第 5 章

FreeRTOS 任务间同步

本章详细讲述 FreeRTOS 中任务间同步的多种机制，包括信号量、互斥量和事件组，具体包括信号量的类型及其应用场景，互斥量的工作原理和优先级翻转问题，以及事件组的原理和功能；详细讲述这些同步机制的运作机制、控制块、相关函数，并通过实际应用实例来帮助读者理解和实现这些同步机制。

重点内容：

(1) 信号量。

① 二值信号量：介绍二值信号量的概念及应用场景。

② 计数信号量：解释计数信号量的特点及其运作机制。

③ 互斥量信号量：描述互斥量信号量的功能。

④ 递归互斥量：讨论递归互斥量的使用。

⑤ 信号量控制块：说明信号量的控制结构。

⑥ 信号量相关函数：列出与信号量操作相关的 API 函数。

⑦ FreeRTOS 信号量应用实例：通过实例展示如何使用信号量。

(2) 互斥量。

① 优先级翻转问题：解释优先级翻转及其解决方法。

② 互斥量的工作原理：介绍互斥量的工作原理。

③ 互斥量应用场景：列举互斥量的典型应用场景。

④ 互斥量控制块：描述互斥量的控制结构。

⑤ 互斥量函数接口：列出相关函数接口。

⑥ FreeRTOS 互斥量应用实例：通过实例讲述互斥量的实际应用。

(3) 事件组。

① 事件组的原理和功能：介绍事件组的基本概念和功能。

② 事件组的应用场景：讨论事件组在任务同步中的具体应用。

③ 事件组运作机制：解释事件组的运行机制。

④ 事件组控制块：说明事件组的控制结构。

⑤ 事件组相关函数：列出与事件组操作相关的 API 函数。

⑥ FreeRTOS 事件组应用实例：通过实例展示如何在实际应用中使用事件组。

5.1 FreeRTOS 信号量

队列的功能是将进程间需要传递的数据存在其中，所以在有的 RTOS 系统里，队列也被称为“邮箱”(mailbox)。有时进程间需要传递的只是一个标志，用于进程间同步或对一个共享资源的互斥性访问，这时就可以使用信号量(semaphore)或互斥量(mutex)。信号量和互斥量的实现都是基于队列的，信号量更适用于进程间同步，互斥量更适用于共享资源的互斥性访问。

信号量和互斥量都可应用于进程间通信，它们都是基于队列的基本数据结构，但是信号量和互斥量又有一些区别。从队列派生出来的信号量和互斥量的分类如图 5-1 所示。

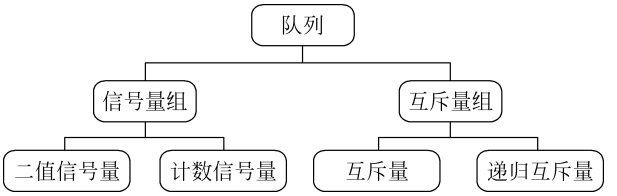

图 5-1 从队列派生出来的信号量和互斥量的分类

5.1.1 二值信号量

二值信号量(binary semaphore)就是只有一个项的队列，这个队列要么是空的，要么是满的，所以相当于只有 0 和 1 两种值。二值信号量就像一个标志，适合用于进程间同步的通信。例如，图 5-2 是使用二值信号量在 ISR 和任务之间进行同步的示意图。

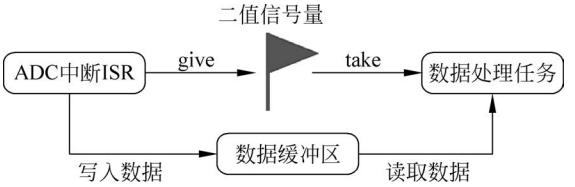

图 5-2 使用二值信号量在 ISR 和任务之间进行同步的示意图

图 5-2 是一个使用二值信号量进行进程间同步的系统，其说明如下。

1. 系统组成

系统有两个进程：

(1) ADC 中断 ISR：负责读取 ADC 转换结果并将结果写入缓冲区。

(2) 数据处理任务：负责读取缓冲区的内容并进行处理。

2. 数据缓冲区

(1) 数据缓冲区是两个任务需要同步访问的对象。

(2) 为了简化分析，假设数据缓冲区仅存储一次转换结果数据。

3. ADC 中断 ISR 的工作流程

(1) 读取数据：ADC 中断 ISR 读取 ADC 转换结果。

(2) 写入缓冲区：将读取的数据写入数据缓冲区。

(3) 释放信号量：释放(give)二值信号量，表示数据缓冲区里已经存入了新的转换结果数据。

4. 数据处理任务的工作流程

(1) 等待信号量：任务尝试获取(take)二值信号量。

(2) 阻塞等待：如果二值信号量无效，任务进入阻塞状态，可以设定一直等待或设定超时时间。

(3) 退出阻塞：当二值信号量变为有效时，数据处理任务退出阻塞状态，进入运行状态。

(4) 处理数据：读取缓冲区的数据并进行处理。

5. 比较标志变量和二值信号量

(1) 标志变量的缺点：如果使用标志变量代替二值信号量进行同步，数据处理任务需要不断地查询标志变量的值，导致频繁的 CPU 占用。

(2) 二值信号量的优势：使用二值信号量可以让数据处理任务在等待数据时进入阻塞状态，从而提高系统效率，减少 CPU 空耗。

二值信号量在进程间同步中能够有效地减少 CPU 资源浪费，使任务可以在真正有数据时才执行处理，从而提高系统的运行效率。

5.1.2　计数信号量

计数信号量(counting semaphore)就是有固定长度的队列，队列的每项是一个标志。计数信号量通常用于对多个共享资源的访问进行控制，其工作原理可用图 5-3 来说明。

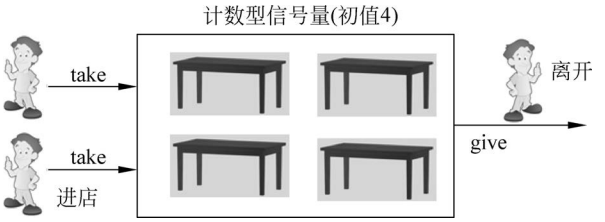

图 5-3　计数信号量的工作原理

(1) 一个计数信号量被创建时设置为初值 4，实际上是队列中有 4 项，表示可共享访问的 4 个资源，这个值只是计数值。可以将这 4 个资源类比为图 5-3 中一个餐馆里的 4 个餐桌，客人就是访问资源的 ISR 或任务。

(2) 当有客人进店时，就是获取信号量，如果有 1 个客人进店了(假设 1 个客人占用 1 张桌子)，计数信号量的值就减 1，计数信号量的值变为 3，表示还有 3 张空余桌子。如果计数信号量的值变为 0，表示 4 张桌子都被占用了，再有客人要进店时就需要等待。在任务中

申请信号量时，可以设置等待超时时间，在等待时，任务进入阻塞状态。

(3) 如果有 1 个客人用餐结束离开了，就是释放信号量，计数信号量的值就加 1，表示可用资源数量增加了 1 个，可供其他要进店的人获取。

由计数信号量的工作原理可知，它适用于管理多个共享资源，例如 ADC 连续数据采集时，一般使用双缓冲区，就可以使用计数信号量管理。

5.1.3 互斥信号量

互斥信号量是一种特殊的二值信号量，通过其特有的优先级继承机制，它更适用于简单的互锁操作，即保护临界资源(关于优先级继承机制的详细讨论将在后续章节中进行)。

1. 互斥信号量的创建与使用

(1) 创建互斥信号量。

当创建一个互斥信号量时，初始可用信号量的数量应为 1，这表示临界资源当前未被占用。

(2) 任务获取互斥信号量。

当任务需要使用临界资源(任何时刻只能被一个任务访问的资源)时，它首先尝试获取互斥信号量。

成功获取信号量后，信号量变为 0，这表示临界资源正在被某个任务使用。

(3) 阻塞其他任务。

当其他任务也尝试访问该临界资源时，由于无法获取信号量，它们会进入阻塞状态，直到信号量再次可用。这种机制保证了临界资源的安全访问。

(4) 任务释放互斥信号量。

当任务完成对临界资源的访问后，它会释放互斥信号量，使信号量数量恢复到 1。这表示临界资源现在可供其他任务使用，阻塞的任务可以继续尝试获取信号量。

2. 互斥信号量的重要性

在操作系统中，用户经常使用信号量来表示临界资源的占用情况。当一个任务需要访问临界资源时，它会首先查询信号量的状态。如果信号量显示资源已被占用，任务将等待或采取其他措施，直到资源可用。

这种机制确保了临界资源在多任务系统中得到有效的保护，避免了资源竞争和潜在的冲突，从而提高了系统的稳定性和性能。

互斥信号量通过有效管理对临界资源的访问，确保了资源的安全使用。它使任务在访问临界资源前能够了解资源状态，并在资源被占用时进行等待，从而防止资源冲突。与普通二值信号量相比，互斥信号量还具备优先级继承机制，使其更加适用于复杂的多任务应用场景。

5.1.4 递归互斥量

递归互斥量(recursive mutex)是一种特殊类型的互斥量，适用于需要递归调用的函数场景。

当一个任务获取一个互斥量后，它不能再次获取相同的互斥量，否则可能会导致死锁。

1. 递归互斥量的优势

递归互斥量在互斥量基础上增加了一些灵活性。

(1) 同一任务的多次获取：当一个任务获取递归互斥量后，它可以在不释放的情况下再次获取该互斥量。这对于一些需要嵌套调用的函数非常有用。

(2) 配对使用：尽管一个任务可以多次获取递归互斥量，但每次获取必须配对一次释放。这意味着获取和释放的次数必须相等，才能最终真正释放该互斥量。

2. 示例说明

(1) 递归调用的任务。

假设一个任务 A 获取了递归互斥量 M。

在该任务未释放互斥量的情况下，它再次调用并获取了相同的递归互斥量 M。

任务 A 必须两次释放 M(对应其两次获取)，才能使其他任务访问该互斥量。

(2) 安全性。

递归互斥量同样不能在中断服务例程(ISR)中使用。

递归互斥量提供了一种在复杂函数调用中安全使用互斥机制的方式。它允许任务在递归调用中多次获取同一个互斥量，但每次获取必须与一次释放配对。这种机制确保了资源的同步访问安全，同时在设计递归函数时提供了更多的灵活性。

5.1.5　信号量应用场景

在嵌入式操作系统中，二值信号量是一种重要的同步手段，常用于任务间以及任务与中断间的同步。二值信号量和互斥信号量是最常用的信号量类型。

在多任务系统中，二值信号量经常被使用。例如，当一个任务需要等待某个事件的发生时，可以使用二值信号量来实现同步。以下是具体的操作流程。

(1) 避免轮询。

任务可以通过轮询的方式不断查询某个标记是否被置位，但这种方式会消耗大量的 CPU 资源，并且妨碍其他任务的执行。

(2) 阻塞等待。

更好的做法是让任务在大部分时间处于阻塞状态，允许其他任务执行，直到某个事件发生时才唤醒该任务。使用二值信号量可以实现这种机制。当任务尝试获取信号量时，如果特定事件尚未发生，信号量为空，任务会进入阻塞状态。

(3) 事件触发。

当事件条件满足时，任务或中断服务程序会释放信号量，表示事件已发生。任务在获取到信号量后被唤醒，执行相应的操作。

(4) 无须归还信号量。

任务执行完毕后，无须归还信号量。这种机制可以大大提高 CPU 的效率，并且确保实时响应。

二值信号量通过简单的 0 和 1 状态，有效地实现了任务间以及任务与中断间的同步。它避免了轮询带来的 CPU 资源浪费，使任务能够在等待事件时进入阻塞状态，从而提高了系统的整体效率和实时响应能力。

在 FreeRTOS 中，信号量是一个非常重要的同步机制，用于任务间及任务和中断服务程序间的通信和协调。信号量主要分为二值信号量、计数信号量和互斥信号量三种类型，每种类型都有其特定的应用场景。

1. 二值信号量

二值信号量只有两个状态：有(1)和无(0)。它主要用于以下场景。

(1) 任务同步：一个任务在完成特定操作后，通过信号量通知另一个任务，例如传感器数据采集完成后通知处理任务进行数据处理。

(2) 中断与任务同步：在中断服务程序完成某个操作后，通过释放信号量通知任务执行相关操作。例如，串口接收完成后，中断服务程序释放信号量，通知任务处理接收到的数据。

(3) 事件触发：用于实现事件处理机制，例如当某个事件发生时，通过设置信号量通知等待该事件的任务。

```
SemaphoreHandle_t xBinarySemaphore;
void vISR_Handler(void) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    xSemaphoreGiveFromISR(xBinarySemaphore, &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}

void vTask(void * pvParameters) {
    for (;;) {
        if (xSemaphoreTake(xBinarySemaphore, portMAX_DELAY) == pdTRUE) {
            // 处理事件
        }
    }
}
```

2. 计数信号量

计数信号量可以递增，代表可用资源或事件的数量。其应用场景包括：

(1) 资源管理：用于管理多个相同类型的资源，例如多个 ADC 通道、多个串口等。当某个资源可用时，信号量递增；任务使用资源前，先请求信号量，使用完后释放信号量。

(2) 事件计数：处理多个事件的场景，例如多个外部事件源，通过计数信号量跟踪事件发生的次数，每次事件发生时释放信号量，任务消费事件时请求信号量。

```
SemaphoreHandle_t xCountingSemaphore;
void vISR_Handler(void) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    xSemaphoreGiveFromISR(xCountingSemaphore, &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

```
void vTask(void * pvParameters) {
    for (;;) {
        if (xSemaphoreTake(xCountingSemaphore, portMAX_DELAY) == pdTRUE) {
            // 处理事件,例如读取多个传感器数据
        }
    }
}
```

3. 互斥信号量

互斥信号量主要用于保护临界区,确保多个任务互斥访问共享资源。它优于二值信号量的地方在于,互斥信号量支持优先级继承机制,防止优先级反转问题。其应用场景包括:

(1) 共享资源保护:用于保护对全局变量、硬件外设等共享资源的访问,确保同一时间只有一个任务可以访问这些资源。

(2) 数据一致性:确保在修改共享数据时,不会受到其他任务的干扰,防止数据竞争。

```
SemaphoreHandle_t xMutex;
void vTask1(void * pvParameters) {
    for (;;) {
        if (xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE) {
            // 访问共享资源
            // 操作完成
            xSemaphoreGive(xMutex);
        }
    }
}

void vTask2(void * pvParameters) {
    for (;;) {
        if (xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE) {
            // 访问共享资源
            // 操作完成
            xSemaphoreGive(xMutex);
        }
    }
}
```

在 FreeRTOS 中,信号量提供了一种有效的任务同步和资源管理机制,使得多个任务及 ISR 可以协作完成复杂的功能。通过合理地使用信号量,可以提高系统的可靠性、响应性和资源利用效率。

5.1.6　二值信号量运作机制

创建信号量时,系统会为创建的信号量对象分配内存,并把可用信号量初始化为用户自定义的个数。二值信号量的最大可用信号量个数为 1。

任何任务都可以从创建的二值信号量资源中获取一个二值信号量,获取成功则返回正确,否则任务会根据用户指定的阻塞超时时间来等待其他任务/中断释放信号量。在等待这段时间,系统将任务变成阻塞态,任务将被挂到该信号量的阻塞等待列表中。

在二值信号量无效时，假如此时有任务获取该信号量，那么任务将进入阻塞态，具体如图 5-4 所示。

假如某个时间中断/任务释放了信号量，其过程具体如图 5-5 所示，那么，由于获取无效信号量而进入阻塞态的任务将获得信号量并且恢复为就绪态，其过程具体如图 5-6 所示。

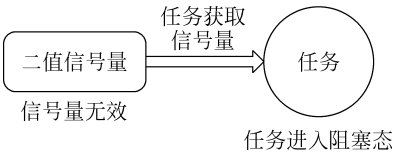

图 5-4　信号量无效时获取

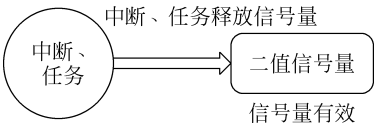

图 5-5　中断和任务释放信号量

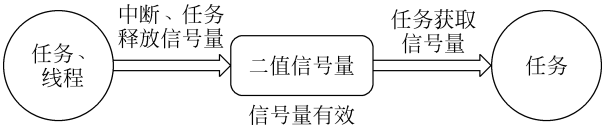

图 5-6　二值信号量运作机制

5.1.7　计数信号量运作机制

计数信号量是一种有效的资源管理工具，允许多个任务同时获取信号量以访问共享资源，但会限制同时访问该资源的任务数目。

计数信号量的运作机制：

(1) 允许多个任务访问。计数信号量允许多个任务同时访问共享资源。

(2) 限制最大任务数。计数信号量设置了可以同时访问资源的最大任务数。如果访问任务数达到这个最大值，其他试图获取该信号量的任务会被阻塞。

(3) 任务阻塞与唤醒。计数信号量可以允许多个任务获取信号量访问共享资源，但会限制任务的最大数目。访问的任务数达到可支持的最大数目时，会阻塞其他试图获取该信号量的任务，直到有任务释放了信号量。

这就是计数型信号量的运作机制，虽然计数信号量允许多个任务访问同一个资源，但是也有限定，比如某个资源限定只能有 3 个任务访问，那么第 4 个任务访问时，会因为获取不到信号量而进入阻塞，等到有任务(比如任务 1)释放掉该资源的时候，第 4 个任务才能获取到信号量从而进行资源的访问。其运作机制具体如图 5-7 所示。

5.1.8　信号量控制块

信号量 API 函数实际上都是宏，它使用现有的队列机制。这些宏定义在 semphr.h 文件中，如果使用信号量或者互斥量，需要包含 semphr.h 头文件。所以，FreeRTOS 的信号量控制块结构体与消息队列结构体是一模一样的，只不过结构体中某些成员变量代表的含义不一样。

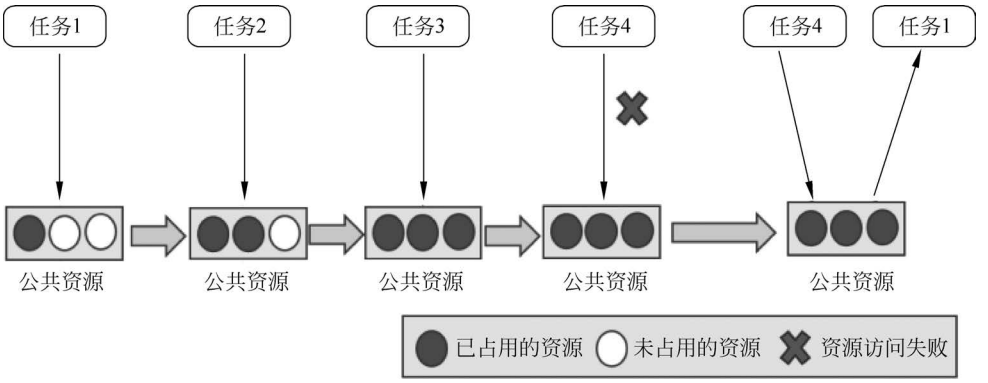

图 5-7　计数信号量运作机制

信号量控制块代码清单如下：

```
1  typedef struct QueueDefinition {
2  int8_t * pcHead;
3  int8_t * pcTail;
4  int8_t * pcWriteTo;
5
6  union {
7  int8_t * pcReadFrom;
8  UBaseType_t uxRecursiveCallCount;
9  } u;
10
11 List_t xTasksWaitingToSend;
12 List_t xTasksWaitingToReceive;
13
14 volatile UBaseType_t uxMessagesWaiting; (1)
15 UBaseType_t uxLength; (2)
16 UBaseType_t uxItemSize; (3)
17
18 volatile int8_t cRxLock;
19 volatile int8_t cTxLock;
20
21 #if((configSUPPORT_STATIC_ALLOCATION == 1)
22 && (configSUPPORT_DYNAMIC_ALLOCATION == 1))
23 uint8_t ucStaticallyAllocated;
24 #endif
25
26 #if (configUSE_QUEUE_SETS == 1)
27 struct QueueDefinition * pxQueueSetContainer;
28 #endif
29
30 #if (configUSE_TRACE_FACILITY == 1)
31 UBaseType_t uxQueueNumber;
32 uint8_t ucQueueType;
33 #endif
```

```
34
35 } xQUEUE;
36
37 typedef xQUEUE Queue_t;
```

FreeRTOS的信号量控制块是通过结构体Queue_t来定义的。该结构体用于实现队列和信号量的数据管理,具体功能如下:

(1) pcHead和pcTail指向队列的头和尾。

(2) pcWriteTo指向当前写入位置。

(3) 联合体成员pcReadFrom或uxRecursiveCallCount可作为读位置指针或递归调用计数。

(4) xTasksWaitingToSend和xTasksWaitingToReceive管理等待队列操作的任务。

(5) uxMessagesWaiting、uxLength和uxItemSize分别表示队列中的消息数量、队列长度和单项大小。

(6) cRxLock和cTxLock用于接收和发送操作的同步。

(7) 额外的成员如ucStaticallyAllocated、pxQueueSetContainer、uxQueueNumber和ucQueueType根据配置宏条件定义,为静态分配、队列集合和跟踪设施提供支持。

这些成员共同作用,实现FreeRTOS中信号量和队列的功能。

5.1.9 相关函数

信号量和互斥量相关的常量和函数定义都在头文件semphr.h中,函数都是宏函数,都是调用文件queue.c中的一些函数实现的。信号量和互斥量操作相关的函数如表5-1所示。

表5-1 信号量和互斥量操作相关的函数

函 数 名	功 能 描 述
xSemaphoreCreateBinary	创建二值信号量
xSemaphoreCreateBinaryStatic	创建二值信号量,静态分配内存
xSemaphoreCreateCounting	创建计数型信号量
xSemaphoreCreateCountingStatic	创建计数型信号量,静态分配内存
xSemaphoreCreateMutex	创建互斥量
xSemaphoreCreateMutexStatic	创建互斥量,静态分配内存
xSemaphoreCreateRecursiveMutex	创建递归互斥量
xSemaphoreCreateRecursiveMutexStatic	创建递归互斥量,静态分配内存
vSemaphoreDelete	删除这4种信号量或互斥量
xSemaphoreGive	释放二值信号量、计数型信号量、互斥量
xSemaphoreGiveFromISR	xSemaphoreGive的ISR版本,但不能用于互斥量
xSemaphoreGiveRecursive	释放递归互斥量
xSemaphore Take	获取二值信号量、计数型信号量、互斥量
xSemaphore TakeFromISR	xSemaphoreTake的ISR版本,但不用于互斥量
xSemaphore TakeRecursive	获取递归互斥量

1. 创建信号量函数

1) 创建二值信号量

xSemaphoreCreateBinary 用于创建一个二值信号量,并返回一个句柄。其实二值信号量和互斥量都共同使用一个类型 SemaphoreHandle_t 的句柄(.h 文件 79 行),该句柄的原型是一个 void 型的指针。使用该函数创建的二值信号量是空的,在使用函数 xSemaphoreTake 获取之前必须先调用函数 xSemaphoreGive 释放后才可以获取。如果是使用老式的函数 vSemaphoreCreateBinary 创建的二值信号量,则为 1,在使用之前不用先释放。要想使用该函数必须在 FreeRTOSConfig.h 中把宏 configSUPPORT_DYNAMIC_ALLOCATION 定义为 1,即开启动态内存分配。其实该宏在 FreeRTOS.h 中默认定义为 1,即所有 FreeRTOS 的对象在创建的时候都默认使用动态内存分配方案。

xSemaphoreCreateBinary 函数原型如下:

```
#define xSemaphoreCreateBinary() xQueueGenericCreate((UBaseType_t) 1,
                                 semSEMAPHORE_QUEUE_ITEM_LENGTH, \
                                 queueQUEUE_TYPE_BINARY_SEMAPHORE)
```

xSemaphoreCreateBinary 是 FreeRTOS 中用于创建二值信号量的函数。二值信号量可以用于任务之间的同步或简单的资源管理。函数本质上是通过调用 xQueueGenericCreate 函数创建一个具有 1 个项目的队列,并指定队列类型为二值信号量。

(1) 功能说明。

① 创建一个初始状态为未获取的二值信号量。

② 信号量只能有两种状态:已获取(1)或未获取(0)。

③ 当信号量被获取(由任务调用 xSemaphoreTake),其状态变为已获取。

④ 当信号量被释放(由任务调用 xSemaphoreGive),其状态变为未获取。

该宏定义通过调用 xQueueGenericCreate 创建一个项目长度为 1、类型为二值信号量的队列。

(2) 应用实例。

以下是一个使用二值信号量进行任务同步的例子。一个任务 TaskA 负责给予信号量,另一个任务 TaskB 负责等待信号量执行相关操作。

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

// 二值信号量句柄
SemaphoreHandle_t xBinarySemaphore;

// Task A:释放信号量
void TaskA(void * pvParameters) {
    for(;;) {
        // 模拟某些工作
        vTaskDelay(pdMS_TO_TICKS(1000));
```

```
            // 释放信号量,通知 TaskB
            xSemaphoreGive(xBinarySemaphore);
        }
    }

    // Task B:等待信号量
    void TaskB(void * pvParameters) {
        for(;;) {
            // 等待信号量(最大等待时间为 portMAX_DELAY)
            if(xSemaphoreTake(xBinarySemaphore, portMAX_DELAY) == pdTRUE) {
                // 收到信号量后执行某些操作
                printf("Task B 收到信号量!\n");
            }
        }
    }

    int main(void) {
        // 创建二值信号量
        xBinarySemaphore = xSemaphoreCreateBinary();

        if (xBinarySemaphore != NULL) {
            // 创建 Task A 和 Task B
            xTaskCreate(TaskA, "TaskA", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
            xTaskCreate(TaskB, "TaskB", configMINIMAL_STACK_SIZE, NULL, 1, NULL);

            // 启动调度器
            vTaskStartScheduler();
        }

        // 当调度器启动失败时,程序会一直运行以下代码
        for(;;);

        return 0;
    }
```

上面代码解释如下：

① 创建信号量：在 main 函数中使用 xSemaphoreCreateBinary 创建二值信号量，并存储在 xBinarySemaphore 句柄中。

② 创建任务：创建了两个任务 TaskA 和 TaskB，分别负责释放信号量和等待信号量。

③ Task A：每隔 1s 释放一次信号量，通知 TaskB。

④ Task B：无限期地等待信号量，收到信号量后执行某些操作。

这个实例展示了如何使用二值信号量在 FreeRTOS 中实现任务间的简单同步。

2）创建计数信号量

xSemaphoreCreateCounting 用于创建一个计数信号量。要想使用该函数，必须在 FreeRTOSConfig.h 中把宏 configSUPPORT_DYNAMIC_ALLOCATION 定义为 1，即开启动态内存分配。其实该宏在 FreeRTOS.h 中默认定义为 1，即所有 FreeRTOS 的对象在

创建的时候都默认使用动态内存分配方案。

计数信号量跟二值信号量的创建过程相似，其实也是间接调用 xQueueGenericCreate 函数进行创建。

xSemaphoreCreateCounting 函数原型如下：

```
#define xSemaphoreCreateCounting(uxMaxCount, uxInitialCount) \
        xQueueCreateCountingSemaphore((uxMaxCount),(uxInitialCount))
```

删除信号量过程其实就是删除消息队列过程，因为信号量就是消息队列，只不过是无法存储消息的队列而已。

vSemaphoreDelete()函数原型如下：

```
#define vSemaphoreDelete(xSemaphore) \
        vQueueDelete((QueueHandle_t) (xSemaphore))
```

2. 信号量删除函数

vSemaphoreDelete 用于删除一个信号量，包括二值信号量、计数信号量、互斥量和递归互斥量。如果有任务阻塞在该信号量上，那么不要删除该信号量。

vSemaphoreDelete()函数原型如下：

```
void vSemaphoreDelete(SemaphoreHandle_t xSemaphore)
```

3. 信号量释放函数

与消息队列的操作一样，信号量的释放可以在任务、中断中使用，所以需要有不一样的 API 函数在不一样的上下文环境中调用。

当信号量有效时，任务才能获取信号量，那么是什么函数使得信号量变得有效？在创建的时候进行初始化，将它可用的信号量个数设置一个初始值。在二值信号量中，该初始值的范围是 0～1(旧版本的 FreeRTOS 中创建二值信号量默认是有效的，而新版本则默认是无效的)，假如初始值为 1 个可用的信号量，被申请一次就变得无效了，那就需要释放信号量。FreeRTOS 提供了信号量释放函数，每调用一次该函数就释放一个信号量。但是有个问题，能不能一直释放？很显然，这是不能的，无论信号量是二值信号量还是计数信号量，都要注意可用信号量的范围。当用作二值信号量时，必须确保其可用值在 0～1 范围内；用作计数信号量时，由用户在创建时指定 uxMaxCount，其最大可用信号量不允许超出 uxMaxCount，这代表不能一直调用信号量释放函数来释放信号量，其实一直调用也是无法释放成功的。

(1) xSemaphoreGive(任务)。

xSemaphoreGive 是一个用于释放信号量的宏，真正的实现过程是调用消息队列通用发送函数。释放的信号量对象必须是已经被创建的，可以用于二值信号量、计数信号量、互斥量的释放，但不能释放由函数 xSemaphoreCreateRecursiveMutex 创建的递归互斥量。此外，该函数不能在中断中使用。

xSemaphoreGive 函数原型如下：

```
#define xSemaphoreGive(xSemaphore) \
        xQueueGenericSend((QueueHandle_t) (xSemaphore), \
                          NULL, \
                          semGIVE_BLOCK_TIME, \
                          queueSEND_TO_BACK)
```

从该宏定义可以看出，释放信号量实际上是一次入队操作，并且不允许入队阻塞，因为阻塞时间为 semGIVE_BLOCK_TIME，该宏的值为 0。

通过消息队列入队过程分析，可以将释放一个信号量的过程简化：如果信号量未满，控制块结构体成员 uxMessageWaiting 就会加 1，然后判断是否有阻塞的任务，如果有就会恢复阻塞的任务，然后返回成功信息（pdPASS）；如果信号量已满，则返回错误代码（err_QUEUE_FULL）。

（2）xSemaphoreGiveFromISR（中断）。

xSemaphoreGiveFromISR 用于释放一个信号量，带中断保护。被释放的信号量可以是二进制信号量和计数信号量。和普通版本的释放信号量 API 函数有些许不同，它不能释放互斥量，这是因为互斥量不可以在中断中使用，互斥量的优先级继承机制只能在任务中起作用，而在中断中毫无意义。带中断保护的信号量释放其实也是一个宏，真正调用的函数是 xQueueGiveFromISR。

SemaphoreGiveFromISR 函数原型如下：

```
#define xSemaphoreGiveFromISR(xSemaphore, pxHigherPriorityTaskWoken) \
        xQueueGiveFromISR((QueueHandle_t) \
                          (xSemaphore), \
                          (pxHigherPriorityTaskWoken))
```

如果可用信号量未满，控制块结构体成员 uxMessageWaiting 就会加 1，然后判断是否有阻塞的任务，如果有的话就会恢复阻塞的任务，然后返回成功信息（pdPASS）。如果恢复的任务优先级比当前任务优先级高，那么在退出中断前要进行任务切换一次；如果信号量满，则返回错误代码（err_QUEUE_FULL），表示信号量满。

4. 信号量获取函数

与消息队列的操作一样，信号量的获取可以在任务、中断（中断中使用并不常见）中使用，所以需要有不一样的 API 函数在不一样的上下文环境中调用。

与释放信号量对应的是获取信号量。当信号量有效时，任务才能获取信号量，当任务获取了某个信号量时，该信号量的可用个数就减 1，当它减到 0 时，任务就无法再获取了，并且获取的任务会进入阻塞态（假如用户指定了阻塞超时时间）。如果某个信号量中当前拥有 1 个可用的信号量，被获取一次就变得无效了，那么此时另外一个任务获取该信号量时，就会无法获取成功，该任务便会进入阻塞态，阻塞时间由用户指定。

（1）xSemaphoreTake（任务）。

xSemaphoreTake 函数用于获取信号量，可以是二值信号量、计数信号量、互斥量，不带中断保护。获取的信号量对象可以是二值信号量、计数信号量和互斥量，但是递归互斥量并不能使用

这个 API 函数获取。其实获取信号量是一个宏，真正调用的函数是 xQueueGenericReceive。该宏不能在中断使用，而是必须由具体中断保护功能的 xQueueReceiveFromISR 版本代替。

xSemaphoreTake 函数原型如下：

```
#define xSemaphoreTake(xSemaphore, xBlockTime)
    xQueueGenericReceive((QueueHandle_t) (xSemaphore),NULL,(xBlockTime),pdFALSE)
```

从该宏定义可以看出释放信号量实际上是一次消息出队操作，阻塞时间由用户指定 xBlockTime，当有任务试图获取信号量时，当且仅当信号量有效时，任务才能读获取到信号量。如果信号量无效，在用户指定的阻塞超时时间中，该任务将保持阻塞状态以等待信号量有效。若其他任务或中断释放了有效的信号量，该任务将自动由阻塞态转移为就绪态。当任务等待的时间超过了指定的阻塞时间时，即使信号量中还是没有可用信号量，任务也会自动从阻塞态转移为就绪态。

通过前面消息队列出队过程分析，可以将获取一个信号量的过程简化：如果有可用信号量，控制块结构体成员 uxMessageWaiting 就会减 1，然后返回获取成功信息(pdPASS)；如果信号量无效并且阻塞时间为 0，则返回错误代码(errQUEUE_EMPTY)；如果信号量无效并且用户指定了阻塞时间，则任务会因为等待信号量而进入阻塞状态，任务会被挂接到延时列表中。

(2) xSemaphoreTakeFromISR(中断)。

xSemaphoreTakeFromISR 是函数 xSemaphoreTake 的中断版本，用于获取信号量，是一个不带阻塞机制获取信号量的函数，获取对象必须由是已经创建的信号量。信号量类型可以是二值信号量和计数信号量，它与 xSemaphoreTake 函数不同，不能用于获取互斥量，因为互斥量不可以在中断中使用，并且互斥量特有的优先级继承机制只能在任务中起作用，而在中断中毫无意义。

xSemaphoreTakeFromISR 函数原型如下：

```
xSemaphoreTakeFromISR(SemaphoreHandle_t xSemaphore,
                      signed BaseType_t * pxHigherPriorityTaskWoken)
```

5.1.10　FreeRTOS 信号量应用实例

1. 二值信号量同步实例

信号量同步实验是在 FreeRTOS 中创建两个任务，一个是获取信号量任务，一个是释放互斥量任务，两个任务独立运行。获取信号量任务一直在等待信号量，其等待时间是 portMAX_DELAY，等到获取到信号量之后，任务开始执行任务代码，如此反复等待其他任务释放的信号量。

释放信号量任务检测按键是否按下，如果按下则释放信号量，此时释放信号量会唤醒获取任务，获取任务开始运行，然后形成两个任务间的同步。因为如果没按下按键，那么信号量就不会释放，只有当信号量释放时，获取信号量的任务才会被唤醒，如此一来就实现了任务与任务的同步，同时程序的运行会在串口打印出相关信息。

二值信号量源代码如下：

```
/**********************************************************************
  * @file  main.c
  * @brief   FreeRTOS V9.0.0 + STM32 二值信号量同步
  * 实验平台:野火 STM32F407 霸天虎开发板
 **********************************************************************/
 /**********************************************************************
*                                包含的头文件
**********************************************************************/
/* FreeRTOS 头文件 */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
/* 开发板硬件 bsp 头文件 */
#include "bsp_led.h"
#include "bsp_debug_usart.h"
#include "bsp_key.h"
/**************************** 任务句柄 ********************************/
/*
 * 任务句柄是一个指针,用于指向一个任务,当任务创建好之后,它就具有了一个任务句柄
 * 以后要想操作这个任务都需要通过这个任务句柄,如果是自身的任务操作自己,那么
 * 这个句柄可以为 NULL
 */
static TaskHandle_t AppTaskCreate_Handle = NULL;            /* 创建任务句柄 */
static TaskHandle_t Receive_Task_Handle = NULL;             /* LED 任务句柄 */
static TaskHandle_t Send_Task_Handle = NULL;                /* KEY 任务句柄 */

/**************************** 内核对象句柄 ********************************/
/*
 * 信号量,消息队列,事件标志组,软件定时器这些都属于内核的对象,要想使用这些内核
 * 对象,必须先创建,创建成功之后会返回一个相应的句柄。实际上就是一个指针,后续
 * 就可以通过这个句柄操作这些内核对象
 *
 * 内核对象是一种全局的数据结构,通过这些数据结构可以实现任务间的通信,
 * 任务间的事件同步等各种功能。这些功能的实现是通过调用这些内核对象的函数
 * 来完成的
 *
 */
SemaphoreHandle_t BinarySem_Handle =NULL;

/**************************** 全局变量声明 ********************************/
/*
 * 在写应用程序时,可能需要用到一些全局变量
 */


/**************************** 宏定义 ********************************/
/*
```

```
 * 在写应用程序时,可能需要用到一些宏定义
 */

/*
*************************************************************************
*                                函数声明
*************************************************************************
*/
static void AppTaskCreate(void);                                    /* 用于创建任务 */

static void Receive_Task(void* pvParameters);                       /* Receive_Task 任务实现 */
static void Send_Task(void* pvParameters);                          /* Send_Task 任务实现 */

static void BSP_Init(void);                                         /* 用于初始化板载相关资源 */

/***********************************************************************
 * @brief 主函数
 * @param 无
 * @retval 无
 * @note  第一步:开发板硬件初始化
          第二步:创建 App 任务
          第三步:启动 FreeRTOS,开始多任务调度
 ***********************************************************************/
int main(void)
{
  BaseType_t xReturn = pdPASS;      /* 定义一个创建信息返回值,默认为 pdPASS */

  /* 开发板硬件初始化 */
  BSP_Init();
printf("这是一个 FreeRTOS 二值信号量同步实例!\n");
  printf("按下 KEY1 或者 KEY2 进行任务与任务间的同步\n");
  /* 创建 AppTaskCreate 任务 */
  xReturn = xTaskCreate((TaskFunction_t)AppTaskCreate,          /* 任务入口函数 */
                        (const char*    )"AppTaskCreate",     /* 任务名字 */
                        (uint16_t       )512,                 /* 任务栈大小 */
                        (void*          )NULL,                /* 任务入口函数参数 */
                        (UBaseType_t    )1,                   /* 任务的优先级 */
                        (TaskHandle_t*  )&AppTaskCreate_Handle);  /* 任务控制块指针 */
  /* 启动任务调度 */
  if(pdPASS == xReturn)
    vTaskStartScheduler();              /* 启动任务,开启调度 */
  else
    return -1;

  while(1);                             /* 正常不会执行到这里 */
}


/***********************************************************************
  * @ 函数名  : AppTaskCreate
```

```
  * @ 功能说明: 为了方便管理,所有的任务创建函数都放在这个函数里面
  * @ 参数     : 无
  * @ 返回值   : 无
  ********************************************************************/
static void AppTaskCreate(void)
{
  BaseType_t xReturn = pdPASS;          /* 定义一个创建信息返回值,默认为 pdPASS */

  taskENTER_CRITICAL();                 //进入临界区

  /* 创建 BinarySem */
  BinarySem_Handle = xSemaphoreCreateBinary();
  if(NULL != BinarySem_Handle)
    printf("BinarySem_Handle 二值信号量创建成功!\r\n");

  /* 创建 Receive_Task 任务 */
  xReturn = xTaskCreate((TaskFunction_t)Receive_Task,           /* 任务入口函数 */
                        (const char* )"Receive_Task",     /* 任务名字 */
                        (uint16_t )512,                   /* 任务栈大小 */
                        (void* )NULL,                     /* 任务入口函数参数 */
                        (UBaseType_t )2,                  /* 任务的优先级 */
                        (TaskHandle_t* )&Receive_Task_Handle); /* 任务控制块指针 */
  if(pdPASS == xReturn)
    printf("创建 Receive_Task 任务成功!\r\n");

  /* 创建 Send_Task 任务 */
  xReturn = xTaskCreate((TaskFunction_t)Send_Task,              /* 任务入口函数 */
                        (const char* )"Send_Task",        /* 任务名字 */
                        (uint16_t )512,                   /* 任务栈大小 */
                        (void* )NULL,                     /* 任务入口函数参数 */
                        (UBaseType_t )3,                  /* 任务的优先级 */
                        (TaskHandle_t* )&Send_Task_Handle);  /* 任务控制块指针 */
  if(pdPASS == xReturn)
    printf("创建 Send_Task 任务成功!\n\n");

  vTaskDelete(AppTaskCreate_Handle);            //删除 AppTaskCreate 任务

  taskEXIT_CRITICAL();                          //退出临界区
}
/**********************************************************************
  * @ 函数名   : Receive_Task
  * @ 功能说明: Receive_Task 任务主体
  * @ 参数     :
  * @ 返回值   : 无
  ********************************************************************/
static void Receive_Task(void* parameter)
{
  BaseType_t xReturn = pdPASS;          /* 定义一个创建信息返回值,默认为 pdPASS */
  while (1)
```

```
  {
    //获取二值信号量 xSemaphore,若没获取到则一直等待
    xReturn = xSemaphoreTake(BinarySem_Handle,         /* 二值信号量句柄 */
                             portMAX_DELAY);           /* 等待时间 */
    if(pdTRUE == xReturn)
      printf("BinarySem_Handle 二值信号量获取成功!\n\n");
      LED1_TOGGLE;
  }
}

/**********************************************************************
  * @ 函数名  : Send_Task
  * @ 功能说明: Send_Task 任务主体
  * @ 参数    :
  * @ 返回值  : 无
  **********************************************************************/
static void Send_Task(void* parameter)
{
  BaseType_t xReturn = pdPASS;        /* 定义一个创建信息返回值,默认为 pdPASS */
  while (1)
  {
    /* KEY1 被按下 */
    if(Key_Scan(KEY1_GPIO_PORT,KEY1_PIN) == KEY_ON)
    {
      xReturn = xSemaphoreGive(BinarySem_Handle);       //给出二值信号量
      if(xReturn == pdTRUE)
        printf("BinarySem_Handle 二值信号量释放成功!\r\n");
      else
        printf("BinarySem_Handle 二值信号量释放失败!\r\n");
    }
    /* KEY2 被按下 */
    if(Key_Scan(KEY2_GPIO_PORT,KEY2_PIN) == KEY_ON)
    {
      xReturn = xSemaphoreGive(BinarySem_Handle);       //给出二值信号量
      if(xReturn == pdTRUE)
        printf("BinarySem_Handle 二值信号量释放成功!\r\n");
      else
        printf("BinarySem_Handle 二值信号量释放失败!\r\n");
    }
    vTaskDelay(20);
  }
}
/**********************************************************************
  * @ 函数名  : BSP_Init
  * @ 功能说明: 板级外设初始化,所有板子上的初始化均可放在这个函数里面
  * @ 参数    : 无
  * @ 返回值  : 无
  **********************************************************************/
static void BSP_Init(void)
{
```

```
/*
 * STM32 中断优先级分组为 4,即 4bit 都用来表示抢占优先级,范围为 0～15
 * 优先级分组只需要分组一次即可,以后如果有其他的任务需要用到中断,
 * 都统一用这个优先级分组,千万不要再分组
 */
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);

/* LED 初始化 */
LED_GPIO_Config();

/* 串口初始化 */
Debug_USART_Config();

  /* 按键初始化 */
  Key_GPIO_Config();

}

/********************************** END OF FILE **************************/
```

上述代码是一个 FreeRTOS 应用实例,在野火 STM32 开发板上使用二值信号量来进行任务间的同步。主要功能包括以下部分:

(1) 初始化硬件:在 main 函数中调用 BSP_Init,初始化板载 LED、USART 和按键等硬件资源。

(2) 创建任务:调用 xTaskCreate 函数创建三个任务。

AppTaskCreate:用于创建其他任务和信号量。

Receive_Task:等待二值信号量,获取到信号量后切换 LED 的状态。

Send_Task:检测按键状态,按下按键时,释放二值信号量。

(3) 二值信号量:在 AppTaskCreate 任务中,调用 xSemaphoreCreateBinary 函数创建一个二值信号量,并保存在 BinarySem_Handle 中。任务之间通过该信号量进行同步。

Send_Task 任务在检测到按键按下后,通过 xSemaphoreGive 函数释放信号量,表示一个事件发生。

Receive_Task 任务通过 xSemaphoreTake 函数等待获取信号量,获取到信号量后执行相应操作(切换 LED 状态)。

(4) 任务调度:创建任务后,通过 vTaskStartScheduler 启动 FreeRTOS 任务调度,使得各个任务并发运行。

代码整体演示了如何利用二值信号量在 FreeRTOS 中实现任务间的事件同步,并展示了基本的硬件操作和任务创建流程。

将程序编译好,用 USB 线连接计算机和 STM32 开发板的 USB 接口(对应丝印为 USB 转串口),用 DAP 仿真器把配套程序下载到野火 STM32 开发板(这里为野火霸天虎 STM32F407 开发板),二值信号量程序下载界面如图 5-8 所示。

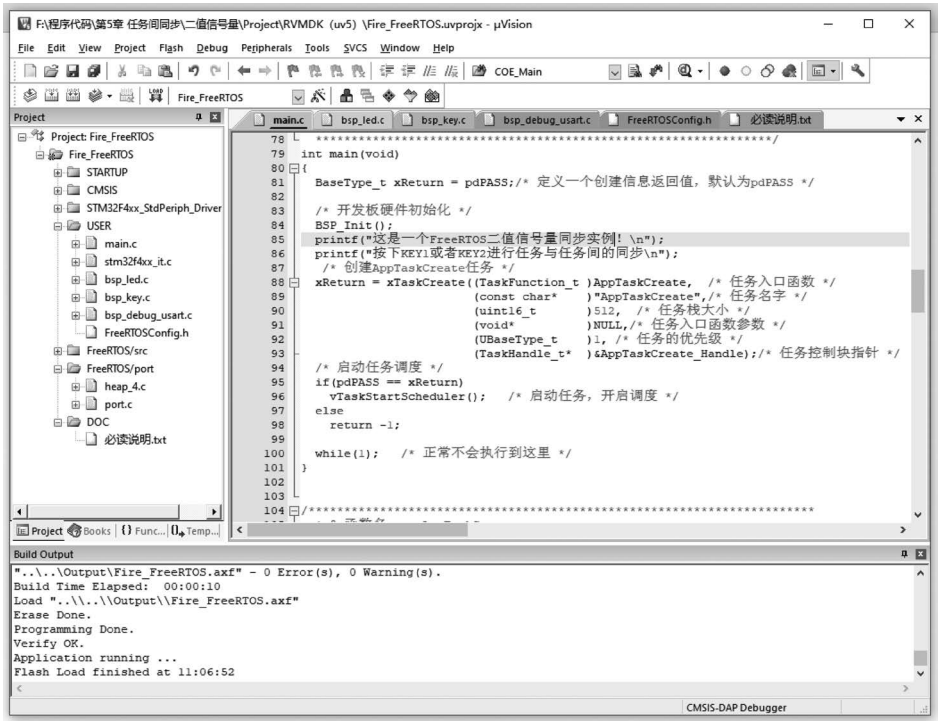

图 5-8　二值信号量程序下载界面

在计算机上打开野火串口调试助手 FireTools，然后复位开发板就可以在调试助手中看到串口的打印信息，它里面输出了信息表明任务正在运行中。按下开发板的按键，串口打印任务运行的信息，表明两个任务同步成功。

二值信号量实例运行结果如图 5-9 所示。

2. 计数信号量实例

计数型信号量实验模拟停车场工作运行。在创建信号量时初始化 5 个可用的信号量，并且创建了两个任务：一个是获取信号量任务，一个是释放信号量任务，两个任务独立运行。获取信号量任务通过按下 KEY1 按键进行信号量的获取，模拟停车场停车操作，其等待时间是 0，在串口调试助手输出相应信息。释放信号量任务通过按下 KEY2 按键进行信号量的释放，模拟停车场取车操作，在串口调试助手输出相应信息。

(1) 计数信号量源代码。

计数信号量源代码从略，请参考本书的数字资源。

(2) 计数信号量实例运行结果。

将程序编译好，用 USB 线连接计算机和 STM32 开发板的 USB 接口(对应丝印为 USB 转串口)，用 DAP 仿真器把配套程序下载到野火 STM32 开发板(这里为野火霸天虎 STM32F407 开发板)，计数信号量程序下载界面如图 5-10 所示。

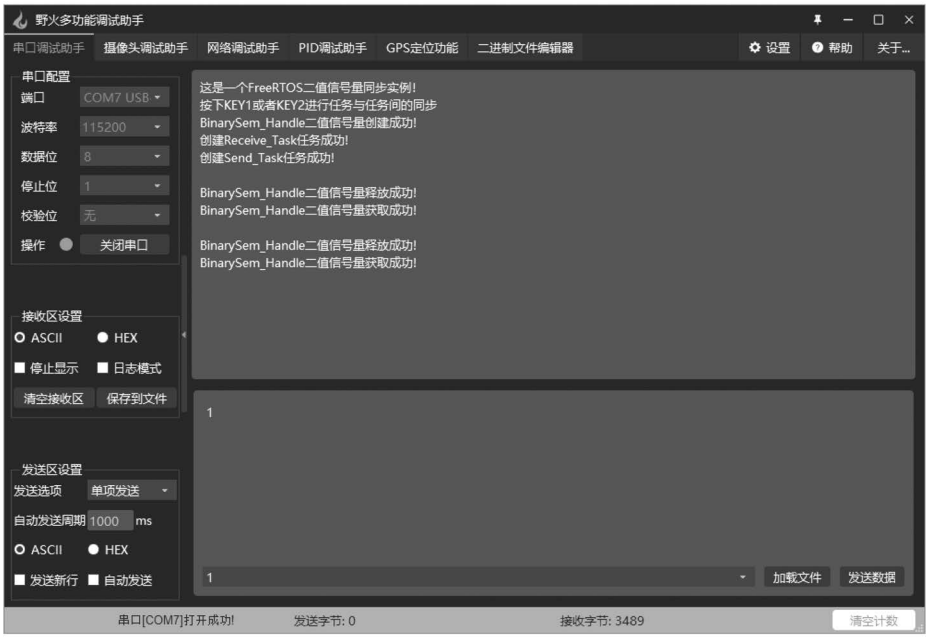

图 5-9　二值信号量实例运行结果

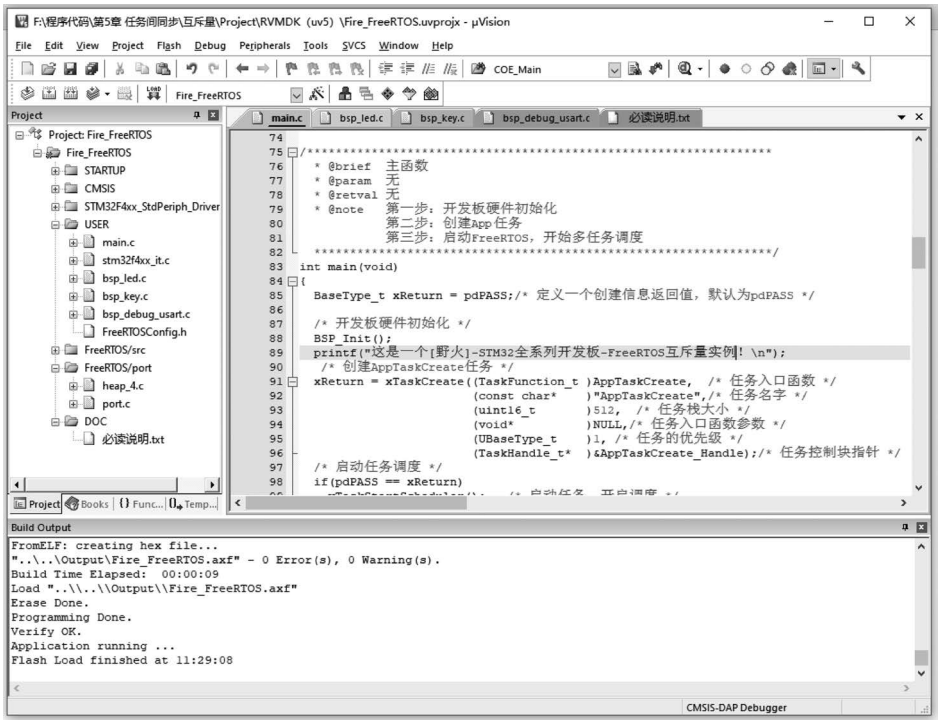

图 5-10　数信号量程序下载界面

在计算机上打开野火串口调试助手 FireTools，然后复位开发板就可以在调试助手中看到串口的打印信息。按下开发板的 KEY1 按键获取信号量模拟停车，按下 KEY2 按键释放信号量模拟取车，在串口调试助手中可以看到运行结果，具体如图 5-11 所示。

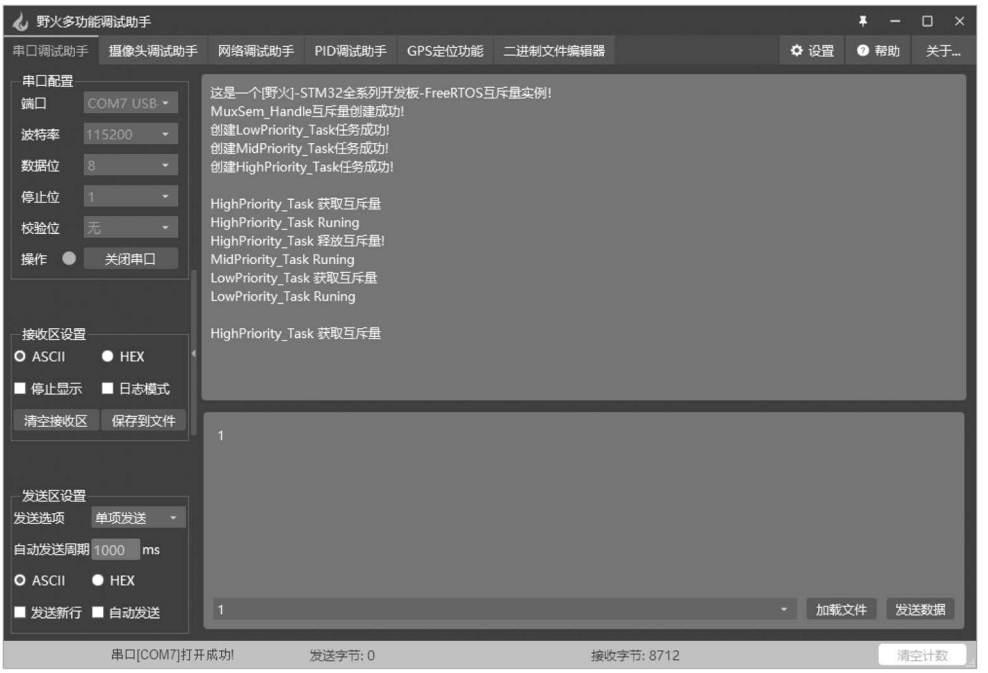

图 5-11　计数信号量实例运行结果

5.2　FreeRTOS 互斥量

使用信号量进行互斥型资源访问控制时，容易出现优先级翻转(priority inversion)问题。互斥量是对信号量的一种改进，增加了优先级继承机制，虽不能完全消除优先级翻转问题，但是可以缓减该问题。在本节中，先介绍出现优先级翻转问题的原因，再介绍引入优先级继承机制后，互斥量解决优先级翻转问题的工作原理。

5.2.1　优先级翻转问题

二值信号量适用于进程间同步，但是二值信号量也可以用于互斥型资源访问控制，只是在这种应用场景下，容易出现优先级翻转问题。使用图 5-12 所示的 3 个任务的运行过程时序图，可以比较直观地说明优先级翻转问题的原理。

在图 5-12 中，有 3 个任务，分别是低优先级的 TaskLP、中等优先级的 TaskMP 和高优先级的 TaskHP，它们的运行过程可描述如下。

(1) 在 t_1 时刻，低优先级任务 TaskLP 处于运行状态，并且获取了一个二值信号量

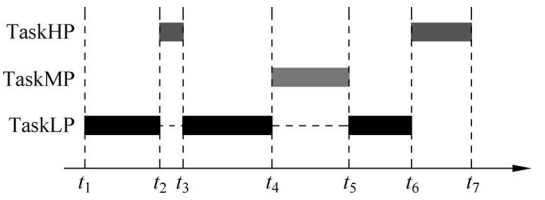

图 5-12　使用二值信号量时 3 个任务的运行过程时序图

semp。

(2) 在 t_2 时刻,高优先级任务 TaskHP 进入运行状态,它申请二值信号量 semp,但是二值信号量被任务 TaskLP 占用,所以 TaskHP 在 t_3 时刻进入阻塞等待状态,TaskLP 进入运行状态。

(3) 在 t_4 时刻,中等优先级任务 TaskMP 抢占了 TaskLP 的 CPU 使用权,TaskMP 不使用二值信号量,所以它一直运行到 t_5 时刻才进入阻塞状态。

(4) 从 t_5 时刻开始,TaskLP 又进入运行状态,直到 t_6 时刻释放二值信号量 semp,TaskHP 才能进入运行状态。

高优先级的任务 TaskHP 需要等待低优先级的任务 TaskLP 释放二值信号量之后才可以运行,这也是期望的运行效果。但是在 t_4 时刻,虽然任务 TaskMP 的优先级比 TaskHP 低,但是它先于 TaskHP 抢占了 CPU 的使用权,这破坏了基于优先级抢占式执行的原则,对系统的实时性是有不利影响的。

5.2.2　互斥量的工作原理

在图 5-12 所示的运行过程中,不希望在 TaskHP 等待 TaskLP 释放信号量的过程中,被一个比 TaskHP 优先级低的任务抢占 CPU 的使用权。也就是说,在图 5-12 中,不希望在 t_4 时刻出现 TaskMP 抢占 CPU 使用权的情况。

为此,FreeRTOS 在二值信号量的功能基础 TaskLP 上引入了优先级继承(priority inheritance)机制,这就是互斥量。使用了互斥量后,图 5-12 的 3 个任务运行过程变为图 5-13 所示的时序图。

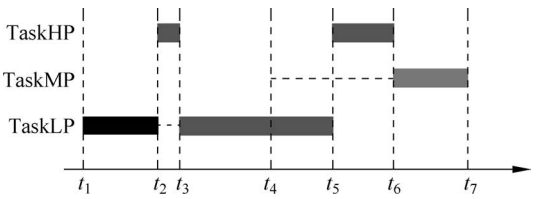

图 5-13　使用互斥量时 3 个任务的运行过程时序图

(1) 在 t_1 时刻,低优先级任务 TaskLP 处于运行状态,并且获取了一个互斥量 mutex。

(2) 在 t_2 时刻,高优先级任务 TaskHP 进入运行状态,它申请互斥量 mutex,但是互斥量被任务 TaskLP 占用,所以 TaskHP 在 t_3 时刻进入阻塞等待状态,TaskLP 进入运行

状态。

(3) 在 t_3 时刻，FreeRTOS 将 TaskLP 的优先级临时提高到与 TaskHP 相同的级别，这就是优先级继承。

(4) 在 t_4 时刻，中等优先级任务 TaskMP 进入就绪状态，发生任务调度，但是因为 TaskLP 的临时优先级高于 TaskMP，所以 TaskMP 无法获得 CPU 的使用权，只能继续处于就绪状态。

(5) 在 t_5 时刻，任务 TaskLP 释放互斥量，任务 TaskHP 立刻抢占 CPU 的使用权，并恢复 TaskLP 原来的优先级。

(6) 在 t_6 时刻，TaskHP 进入阻塞状态后，TaskMP 才进入运行状态。

从图 5-13 的运行过程可以看到，互斥量引入了优先级继承机制，临时提升了占用互斥量的低优先级任务 TaskLP 的优先级，与申请互斥量的高优先级任务 TaskHP 的优先级相同，这样就避免了被中间优先级的任务 TaskMP 抢占 CPU 的使用权，保证了高优先级任务运行的实时性。

互斥量特别适用于互斥型资源访问控制。

使用互斥量可以减缓优先级翻转的影响，但是不能完全消除优先级翻转的问题。例如，在图 5-13 中，若 TaskMP 在 t_2 时刻之前抢占了 CPU，在 TaskMP 运行期间 TaskHP 可以抢占 CPU，但是因为要等待 TaskLP 释放占用的互斥量，要进入阻塞状态等待，还是会让 TaskMP 占用 CPU 运行。

5.2.3　互斥量应用场景

互斥量的适用情况比较单一，因为它是信号量的一种，并且以锁的形式存在。在初始化时，互斥量处于开锁状态，而被任务持有时则立刻转为闭锁状态。互斥量更适用于可能引起优先级翻转的情况。递归互斥量更适用于任务可能多次获取互斥量的情况，这样可以避免同一任务多次递归持有而造成死锁的问题。

多任务环境下往往存在多个任务竞争同一临界资源的应用场景，互斥量可用于对临界资源的保护从而实现独占式访问。另外，互斥量可以降低信号量中存在的优先级翻转问题带来的影响。

比如，有两个任务需要对串口发送数据，其硬件资源只有一个，那么两个任务不能同时发送，否则会导致数据错误。此时就可以用互斥量对串口资源进行保护，当一个任务正在使用串口时，另一个任务无法使用串口，等到一个任务使用串口完毕之后，另一个任务才能获得串口的使用权。

需要注意的是，互斥量不能在中断服务函数中使用，因为其特有的优先级继承机制只在任务中起作用，在中断的上下文环境中毫无意义。

在 FreeRTOS 中，互斥量是一种用于管理资源访问的同步原语，特别是为了保护共享资源在多任务环境中的一致性，从而避免资源竞争和数据不一致的问题。互斥量不仅仅是一个简单的二值信号量，它还支持优先级继承机制，用以解决优先级反转问题。以下是互斥

量的典型应用场景。

1. 保护共享资源

当多个任务需要访问同一个共享资源(如全局变量、数据结构、硬件外设等)时,使用互斥量可以确保同一时间只有一个任务能够访问该资源,从而避免资源竞争和数据不一致。

```
SemaphoreHandle_t xMutex;
int sharedResource = 0;

void Task1(void * pvParameters) {
    for (;;) {
        if (xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE) {
            // 访问和操作共享资源
            sharedResource++;
            xSemaphoreGive(xMutex);
        }
    }
}

void Task2(void * pvParameters) {
    for (;;) {
        if (xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE) {
            // 访问和操作共享资源
            sharedResource--;
            xSemaphoreGive(xMutex);
        }
    }
}
```

2. 串行设备访问

在多个任务访问同一个串行设备(如 UART、SPI、I2C 等)的场景下,互斥量可以确保只有一个任务在同一时间内进行数据传输,避免数据冲突和通信错误。

```
SemaphoreHandle_t xUartMutex;
void UartTask1(void * pvParameters) {
    for (;;) {
        if (xSemaphoreTake(xUartMutex, portMAX_DELAY) == pdTRUE) {
            // 发送数据到 UART
            uart_send("Message from Task1");
            xSemaphoreGive(xUartMutex);
        }
    }
}

void UartTask2(void * pvParameters) {
    for (;;) {
        if (xSemaphoreTake(xUartMutex, portMAX_DELAY) == pdTRUE) {
            // 发送数据到 UART
            uart_send("Message from Task2");
            xSemaphoreGive(xUartMutex);
```

```
        }
    }
}
```

3. 文件系统访问

在嵌入式系统中，多个任务可能需要访问文件系统、读写文件。使用互斥量可以保护文件系统操作，确保同一时间只有一个任务进行文件系统的读写操作，防止文件系统损坏。

```
SemaphoreHandle_t xFileMutex;
void FileTask1(void * pvParameters) {
    for (;;) {
        if (xSemaphoreTake(xFileMutex, portMAX_DELAY) == pdTRUE) {
            // 读写文件操作
            file_write("file.txt", "Data from Task1");
            xSemaphoreGive(xFileMutex);
        }
    }
}

void FileTask2(void * pvParameters) {
    for (;;) {
        if (xSemaphoreTake(xFileMutex, portMAX_DELAY) == pdTRUE) {
            // 读写文件操作
            file_write("file.txt", "Data from Task2");
            xSemaphoreGive(xFileMutex);
        }
    }
}
```

4. 优先级反转问题的解决

在高优先级任务等待低优先级任务释放资源的场景下，可能会发生优先级反转问题。FreeRTOS 中的互斥量具有自动的优先级继承机制，当高优先级任务等待互斥量时，持有互斥量的低优先级任务会自动提升到高优先级，从而减少高优先级任务的等待时间。

```
SemaphoreHandle_t xSharedResourceMutex;
void HighPriorityTask(void * pvParameters) {
    for (;;) {
        if (xSemaphoreTake(xSharedResourceMutex, portMAX_DELAY) == pdTRUE) {
            // 访问共享资源
            xSemaphoreGive(xSharedResourceMutex);
        }
    }
}

void LowPriorityTask(void * pvParameters) {
    for (;;) {
        if (xSemaphoreTake(xSharedResourceMutex, portMAX_DELAY) == pdTRUE) {
            // 模拟长时间的资源占用
            vTaskDelay(pdMS_TO_TICKS(1000));
            xSemaphoreGive(xSharedResourceMutex);
```

```
        }
    }
}
```

互斥量在 FreeRTOS 中具有非常广泛的应用场景，包括但不限于保护共享数据、串行设备访问、文件系统操作以及解决优先级反转问题。合理使用互斥量可以显著提高系统的可靠性和响应性。

5.2.4 互斥量的运作机制

在多任务环境下，多个任务可能需要访问同一个临界资源。为了防止资源冲突，FreeRTOS 提供了互斥量来进行资源保护。互斥量如何避免这种冲突呢？

互斥量的工作机制如下（见图 5-14）：

（1）获取互斥量。当任务需要访问临界资源时，首先需要获取互斥量。一旦任务成功获取了互斥量，互斥量立即变为闭锁状态（锁定）。

（2）阻塞其他任务。在互斥量被锁定的情况下，其他任务将无法获取互斥量，因此也无法访问该临界资源。这些任务会根据用户自定义的等待时间进行等待，直到互斥量被释放。

（3）释放互斥量。当持有互斥量的任务完成对资源的访问后，会释放互斥量。此时，其他等待中的任务可以尝试获取互斥量。

（4）确保单一访问。通过这种机制，互斥量确保在任何时刻只有一个任务可以访问临界资源，从而保证了资源操作的安全性。

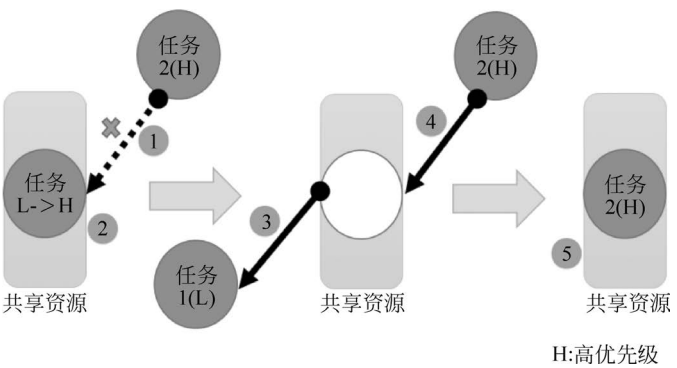

图 5-14 互斥量的运作机制

图 5-14①：因为互斥量具有优先级继承机制，一般选择使用互斥量对资源进行保护，如果资源被占用，无论是什么优先级的任务想要使用该资源都会被阻塞。

图 5-14②：假如正在使用该资源的任务 1 比阻塞中的任务 2 的优先级还低，那么任务 1 将被系统临时提升到与高优先级任务 2 相等的优先级（任务 1 的优先级从 L 变成 H）。

图 5-14③：当任务 1 使用完资源之后，释放互斥量，此时任务 1 的优先级会从 H 变回原来的 L。

图 5-14④和⑤：任务 2 此时可以获得互斥量，然后进行资源的访问，当任务 2 访问资源

时，该互斥量的状态又为闭锁状态，其他任务无法获取互斥量。

5.2.5　互斥量控制块

互斥量的 API 函数实际上都是宏，它使用现有的队列机制，这些宏定义在 semphr.h 文件中，如果使用互斥量，需要包含 semphr.h 头文件。所以，FreeRTOS 的互斥量控制块结构体与消息队列结构体是一模一样的，只不过结构体中某些成员变量代表的含义不一样。

互斥量控制块代码清单如下：

```
1  typedef struct QueueDefinition {
2  int8_t * pcHead;
3  int8_t * pcTail;
4  int8_t * pcWriteTo;
5
6  union {
7  int8_t * pcReadFrom;
8  UBaseType_t uxRecursiveCallCount;
9  } u;
10
11 List_t xTasksWaitingToSend;
12 List_t xTasksWaitingToReceive;
13
14 volatile UBaseType_t uxMessagesWaiting;
15 UBaseType_t uxLength;
16 UBaseType_t uxItemSize;
17
18 volatile int8_t cRxLock;
19 volatile int8_t cTxLock;
20
21 #if((configSUPPORT_STATIC_ALLOCATION == 1)
22 && (configSUPPORT_DYNAMIC_ALLOCATION == 1))
23 uint8_t ucStaticallyAllocated;
24 #endif
25
26 #if (configUSE_QUEUE_SETS == 1)
27 struct QueueDefinition * pxQueueSetContainer;
28 #endif
29
30 #if (configUSE_TRACE_FACILITY == 1)
31 UBaseType_t uxQueueNumber;
32 uint8_t ucQueueType;
33 #endif
34
35 } xQUEUE;
36
37 typedef xQUEUE Queue_t;
```

代码清单展示了 FreeRTOS 中用于实现互斥量的控制块结构体 Queue_t。这个控制块

也用于实现队列和信号量，是 FreeRTOS 中的通用数据结构。

(1) 功能说明。

① 基本指针。

pcHead 和 pcTail：指向队列的头和尾，主要用于队列结构。

pcWriteTo：指向当前写入位置。

② 联合体。

u.pcReadFrom：指示当前读取位置，队列使用。

u.uxRecursiveCallCount：递归调用计数，仅用于互斥量。

③ 任务等待列表。

xTasksWaitingToSend：等待发送的任务列表。

xTasksWaitingToReceive：等待接收的任务列表。

④ 队列属性。

uxMessagesWaiting：当前队列中消息的数量，互斥锁中未使用。

uxLength 和 uxItemSize：队列的长度和单个项目的大小。

⑤ 锁。

cRxLock 和 cTxLock：接收和发送锁，用于同步操作。

⑥ 配置依赖部分。

ucStaticallyAllocated：指示队列是否是静态分配，仅在支持静态和动态分配时定义。

pxQueueSetContainer：指向包含该队列的队列集合，仅在启用队列集合时定义。

uxQueueNumber 和 ucQueueType：队列编号和类型，仅在启用跟踪功能时定义。

(2) 互斥量特有部分。

uxRecursiveCallCount：跟踪同一任务多次获取互斥量的次数，支持递归互斥量。

互斥量使用时，其他成员如队列长度和消息等待等属性未被利用。

这个结构体是 FreeRTOS 实现队列、信号量和互斥量的基础数据结构，通过不同成员和配置的结合，能灵活应用于不同的同步机制。互斥量特别利用了递归调用计数来支持递归锁的功能。

5.2.6 互斥量函数接口

1. 互斥量创建函数 xSemaphoreCreateMutex

xSemaphoreCreateMutex 用于创建一个互斥量，并返回一个互斥量句柄。该句柄的原型是一个 void 型的指针，在使用之前必须先由用户定义一个互斥量句柄。要想使用该函数，必须在 FreeRTOSConfig.h 中把宏 configSUPPORT_DYNAMIC_ALLOCATION 定义为 1，即开启动态内存分配。其实该宏在 FreeRTOS.h 中默认定义为 1，即所有 FreeRTOS 的对象在创建时都默认使用动态内存分配方案，同时还需在 FreeRTOSConfig.h 中把 configUSE_MUTEXES 宏定义打开，表示使用互斥量。

xSemaphoreCreateMutex 函数原型如下：

```
#define xSemaphoreCreateMutex() xQueueCreateMutex(queueQUEUE_TYPE_MUTEX)
```

xSemaphoreCreateMutex是FreeRTOS提供的一个宏，用于创建一个互斥量。互斥量是一种用于管理对共享资源访问的同步工具，确保在同一时刻只有一个任务可以访问共享资源。这对于避免数据竞争和确保数据一致性非常关键。

(1) 功能说明。

xSemaphoreCreateMutex用于创建一个标准的互斥量。该宏实际上是调用xQueueCreateMutex(queueQUEUE_TYPE_MUTEX)函数，生成一个作为互斥量管理结构的队列。

互斥量具有以下特点。

① 互斥：确保同时只有一个任务可以持有资源访问权。

② 递归特性：同一个任务可以多次“获得”同一个互斥量，但在每次操作之前都必须对应一个“释放”操作。

③ 优先级继承：当一个高优先级的任务被阻塞时，如果持有互斥量的是一个低优先级任务，则低优先级任务将暂时“继承”高优先级任务的优先级，以避免优先级反转问题。

(2) 示例代码。

下面是一个简单的应用示例，展示如何使用xSemaphoreCreateMutex来创建和使用一个互斥量。

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

// 全局互斥量句柄
SemaphoreHandle_t xMutex;

// 共享资源
int sharedResource = 0;

// 任务1:修改共享资源
void vTask1(void *pvParameters)
{
    for(;;)
    {
        // 尝试获得互斥量
        if(xSemaphoreTake(xMutex, (TickType_t) 10) == pdTRUE)
        {
            // 获得互斥量,安全地访问共享资源
            sharedResource++;
            printf("Task1 modified sharedResource to %d\n", sharedResource);

            // 释放互斥量
            xSemaphoreGive(xMutex);
```

```
        }

        // 模拟任务处理时间
        vTaskDelay(pdMS_TO_TICKS(100));
    }
}

// 任务 2:修改共享资源
void vTask2(void * pvParameters)
{
    for(;;)
    {
        // 尝试获得互斥量
        if(xSemaphoreTake(xMutex, (TickType_t) 10) == pdTRUE)
        {
            // 获得互斥量,安全地访问共享资源
            sharedResource++;
            printf("Task2 modified sharedResource to %d\n", sharedResource);

            // 释放互斥量
            xSemaphoreGive(xMutex);
        }

        // 模拟任务处理时间
        vTaskDelay(pdMS_TO_TICKS(150));
    }
}

int main(void)
{
    // 创建互斥量
    xMutex = xSemaphoreCreateMutex();

    if (xMutex != NULL)
    {
        // 创建任务
        xTaskCreate(vTask1, "Task1", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
        xTaskCreate(vTask2, "Task2", configMINIMAL_STACK_SIZE, NULL, 1, NULL);

        // 启动调度程序
        vTaskStartScheduler();
    }

    // 如果创建互斥量失败,不会执行这里
    for(;;);

    return 0;
}
```

(3) 代码说明。

① 创建互斥量 xMutex。

```
xMutex = xSemaphoreCreateMutex();
```

如果 xMutex 创建成功，将返回一个有效的互斥量句柄，否则返回 NULL。

② 任务 vTask1 和 vTask2。

任务 vTask1 和任务 vTask2 尝试获得互斥量 xMutex。如果成功，它们分别增加共享资源 sharedResource，然后释放互斥量。

xSemaphoreTake 尝试获得互斥量，等待时间为 10 个系统 tick。

xSemaphoreGive 释放互斥量，使其他任务可以访问共享资源。

③ 启动调度程序。

创建任务之后，通过 vTaskStartScheduler 启动 FreeRTOS 调度程序，开始任务调度。

这个简单的例子展示了如何使用 xSemaphoreCreateMutex 创建一个互斥量，并在多个任务之间保护对共享资源的访问，从而避免竞态条件和数据不一致问题。

2. 递归互斥量创建函数 xSemaphoreCreateRecursiveMutex

xSemaphoreCreateRecursiveMutex 用于创建一个递归互斥量，不是递归的互斥量由函数 xSemaphoreCreateMutex 或 xSemaphoreCreateMutexStatic 创建，且只能被同一个任务获取一次，如果同一个任务想再次获取则会失败。递归信号量则相反，它可以被同一个任务获取很多次，获取多少次就需要释放多少次。递归信号量与互斥量一样，都实现了优先级继承机制，可以降低优先级反转的危害。

要想使用该函数，必须在 FreeRTOSConfig. h 中把宏 configSUPPORT_DYNAMIC_ALLOCATION 和 configUSE_RECURSIVE_MUTEXES 均定义为 1。宏 configSUPPORT_DYNAMIC_ALLOCATION 定义为 1，即表示开启动态内存分配。其实该宏在 FreeRTOS. h 中默认定义为 1，即所有 FreeRTOS 的对象在创建时都默认使用动态内存分配方案。

xSemaphoreCreateRecursiveMutex 函数原型如下：

```
#define xSemaphoreCreateRecursiveMutex()
        xQueueCreateMutex(queueQUEUE_TYPE_RECURSIVE_MUTEX)
```

3. 互斥量删除函数 vSemaphoreDelete

互斥量的本质是信号量，直接调用 vSemaphoreDelete 函数进行删除即可。

在 FreeRTOS 中，vSemaphoreDelete 是用于删除互斥量或信号量的函数。互斥量和信号量在 FreeRTOS 中用于任务间的同步和资源访问控制，而当这些同步对象不再需要时，通过 vSemaphoreDelete 可以释放它们占用的资源。

(1) 功能简介。

① 释放资源：vSemaphoreDelete 函数会释放与互斥量或信号量相关的所有资源，其中包括在 FreeRTOS 内存堆(heap)中为其分配的内存。

② 防止内存泄漏：通过及时删除不再需要的互斥量或信号量，可以有效防止内存泄漏，保持系统的稳定性和运行效率。

(2) 使用示例。

以下是如何创建、使用和删除互斥量的一个示例代码。

```
// 创建一个互斥量
SemaphoreHandle_t xMutex = xSemaphoreCreateMutex();

if (xMutex != NULL) {
    // 使用互斥量进行资源访问控制
    if (xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE) {
        // 访问临界资源
        xSemaphoreGive(xMutex);
    }

    // 当互斥量不再需要时,删除它
    vSemaphoreDelete(xMutex);
}
```

(3) 注意事项。

① 任务知晓：在删除互斥量或信号量前,应确保没有任务在等待或试图使用被删除的同步对象,否则会导致不确定的行为。

② 系统安全性：恰当使用删除函数有助于提升系统的安全性,确保资源合理分配和释放。

通过及时删除不再需要的互斥量和信号量,vSemaphoreDelete 函数在资源管理和操作系统的稳定性方面扮演着重要角色。

4. 互斥量获取函数 xSemaphoreTake

当互斥量处于开锁的状态时,任务才能获取互斥量成功,当任务持有了某个互斥量时,其他任务就无法获取这个互斥量,需要等到持有互斥量的任务进行释放后,其他任务才能获取成功,任务通过互斥量获取函数来获取互斥量的所有权。任务对互斥量的所有权是独占的,任意时刻互斥量只能被一个任务持有,如果互斥量处于开锁状态,那么获取该互斥量的任务将成功获得该互斥量,并拥有互斥量的使用权；如果互斥量处于闭锁状态,获取该互斥量的任务将无法获得互斥量,任务将被挂起。在任务被挂起之前,会进行优先级继承,如果当前任务优先级比持有互斥量的任务优先级高,那么将会临时提升持有互斥量任务的优先级。互斥量的获取函数是一个宏定义,实际调用的函数就是 xQueueGenericReceive。

xSemaphoreTake 函数原型如下：

```
#define xSemaphoreTake(xSemaphore, xBlockTime) \
        xQueueGenericReceive((QueueHandle_t) (xSemaphore), NULL, (xBlockTime), pdFALSE)
```

xQueueGenericReceive 函数其实就是消息队列获取函数,只不过如果使用了互斥量,这个函数会稍微有点不一样。因为互斥量本身有优先级继承机制,所以在这个函数里面会使用宏定义进行编译。如果获取的对象是互斥量,那么这个函数就拥有优先级继承算法；如果获取对象不是互斥量,就没有优先级继承机制。

5. 递归互斥量获取函数 xSemaphoreTakeRecursive

xSemaphoreTakeRecursive 是一个用于获取递归互斥量的宏,与互斥量的获取函数一

样，xSemaphoreTakeRecursive 也是一个宏定义，它最终使用现有的队列机制，实际执行的函数是 xQueueTakeMutexRecursive。

互斥量之前必须由 xSemaphoreCreateRecursiveMutex 这个函数创建。要注意的是，该函数不能用于获取由函数 xSemaphoreCreateMutex 创建的互斥量。要想使用该函数，必须在头文件 FreeRTOSConfig.h 中把宏 configUSE_RECURSIVE_MUTEXES 定义为 1。

xSemaphoreTakeRecursive 函数原型如下：

```
#define xSemaphoreTakeRecursive(xMutex, xBlockTime)
      xQueueTakeMutexRecursive((xMutex), (xBlockTime))
```

6. 互斥量释放函数 xSemaphoreGive

任务想要访问某个资源时，需要先获取互斥量，然后进行资源访问，在任务使用完该资源时，必须及时归还互斥量，这样其他任务才能对资源进行访问。在前面的讲解中，当互斥量有效时，任务才能获取互斥量，那么，是什么函数使得信号量变得有效呢？FreeRTOS 提供了互斥量释放函数 xSemaphoreGive，任务可以调用 xSemaphoreGive 函数进行释放互斥量，表示已经用完了。互斥量的释放函数与信号量的释放函数一致，都是调用 xSemaphoreGive 函数。需要注意的是，互斥量的释放只能在任务中，不允许在中断中释放互斥量。

使用该函数接口时，只有已持有互斥量所有权的任务才能释放它，当任务调用 xSemaphoreGive 函数时会将互斥量变为开锁状态，等待获取该互斥量的任务将被唤醒。

如果任务的优先级被互斥量的优先级翻转机制临时提升，那么当互斥量被释放后，任务的优先级将恢复为原本设定的优先级。

xSemaphoreGive 函数原型如下：

```
#define xSemaphoreGive(xSemaphore) \
        xQueueGenericSend((QueueHandle_t) (xSemaphore), NULL, semGIVE_BLOCK_TIME, queueSEND
_TO_BACK)
```

互斥量、信号量的释放就是调用 xQueueGenericSend 函数，但是互斥量的处理还是有一些不一样的地方，因为它有优先级继承机制，在释放互斥量的时候需要恢复任务的初始优先级。

7. 递归互斥量释放函数 xSemaphoreGiveRecursive

xSemaphoreGiveRecursive 是一个用于释放递归互斥量的宏。要想使用该函数，必须在头文件 FreeRTOSConfig.h 把宏 configUSE_RECURSIVE_MUTEXES 定义为 1。

xSemaphoreGiveRecursive 函数原型如下：

```
#define xSemaphoreGiveRecursive(xMutex) \
        xQueueGiveMutexRecursive((xMutex))
```

xSemaphoreGiveRecursive 函数用于释放一个递归互斥量。已经获取递归互斥量的任务可以重复获取该递归互斥量。使用 xSemaphoreTakeRecursive 函数成功获取几次递归互斥量，就要使用 xSemaphoreGiveRecursive 函数返还几次，在此之前递归互斥量都处于无效

状态，其他任务就无法获取该递归互斥量。使用该函数接口时，只有已持有互斥量所有权的任务才能释放它，每释放一次该递归互斥量，它的计数值就减 1。当该互斥量的计数值为 0 时（持有任务已经释放所有的持有操作），互斥量则变为开锁状态，等待在该互斥量上的任务将被唤醒。如果任务的优先级被互斥量的优先级翻转机制临时提升，那么当互斥量被释放后，任务的优先级将恢复为原本设定的优先级。

5.2.7 FreeRTOS 互斥量应用实例

互斥量实例是基于优先级翻转实验进行修改的，目的是测试互斥量的优先级继承机制是否有效。

互斥量实例是在 FreeRTOS 中创建了三个任务与一个二值信号量，任务分别是高优先级任务、中优先级任务、低优先级任务，用于模拟产生优先级翻转。低优先级任务在获取信号量时，被中优先级打断，中优先级的任务执行时间较长，因为低优先级还未释放信号量，所以高优先级任务就无法取得信号量继续运行，此时就发生了优先级翻转。任务在运行中，使用串口打印出相关信息。

1. 互斥量源代码

```
/**********************************************************************
  * @file   main.c
  * @brief  FreeRTOS V9.0.0 + STM32 互斥量
  **********************************************************************
   * 实验平台:野火 STM32F407 霸天虎开发板

 **********************************************************************/
 /**********************************************************************
  *                              包含的头文件
 **********************************************************************/
/* FreeRTOS 头文件 */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
/* 开发板硬件 bsp 头文件 */
#include "bsp_led.h"
#include "bsp_debug_usart.h"
#include "bsp_key.h"
/****************************** 任务句柄 ********************************/
/*
 * 任务句柄是一个指针，用于指向一个任务，当任务创建好之后，它就具有了一个任务句柄
 * 以后要想操作这个任务都需要通过这个任务句柄，如果是自身的任务操作自己，那么
 * 这个句柄可以为 NULL
 */
static TaskHandle_t AppTaskCreate_Handle = NULL;          /* 创建任务句柄 */
static TaskHandle_t LowPriority_Task_Handle = NULL;       /* LowPriority_Task 任务句柄 */
```

```
static TaskHandle_t MidPriority_Task_Handle = NULL;      /* MidPriority_Task 任务句柄 */
static TaskHandle_t HighPriority_Task_Handle = NULL;     /* HighPriority_Task 任务句柄 */
/************************** 内核对象句柄 ***************************/
/*
 * 信号量、消息队列、事件标志组、软件定时器都属于内核的对象,要想使用这些内核
 * 对象,必须先创建,创建成功之后会返回一个相应的句柄。实际上就是一个指针,后续
 * 就可以通过这个句柄操作这些内核对象
 *
 * 内核对象其实就是一种全局的数据结构,通过这些数据结构可以实现任务间的通信、
 * 任务间的事件同步等各种功能。这些功能的实现是通过调用这些内核对象的函数
 * 来完成的
 */
SemaphoreHandle_t MuxSem_Handle = NULL;

/************************** 全局变量声明 ***************************/
/*
 * 在写应用程序时,可能需要用到一些全局变量
 */
/************************** 宏定义 ***************************/
/*
 * 在写应用程序时,可能需要用到一些宏定义
 */
/*****************************************************************
 *                              函数声明
 *****************************************************************/
static void AppTaskCreate(void);                         /* 用于创建任务 */

static void LowPriority_Task(void* pvParameters);        /* LowPriority_Task 任务实现 */
static void MidPriority_Task(void* pvParameters);        /* MidPriority_Task 任务实现 */
static void HighPriority_Task(void* pvParameters);       /* MidPriority_Task 任务实现 */

static void BSP_Init(void);                              /* 用于初始化板载相关资源 */
/*****************************************************************
  * @brief  主函数
  * @param 无
  * @retval 无
  * @note 第一步:开发板硬件初始化
          第二步:创建 App 任务
          第三步:启动 FreeRTOS,开始多任务调度
  ****************************************************************/
int main(void)
{
  BaseType_t xReturn = pdPASS;     /* 定义一个创建信息返回值,默认为 pdPASS */

  /* 开发板硬件初始化 */
  BSP_Init();
printf("这是一个[野火]-STM32 全系列开发板-FreeRTOS 互斥量实例!\n");
```

```
    /* 创建 AppTaskCreate 任务 */
    xReturn = xTaskCreate((TaskFunction_t)AppTaskCreate,          /* 任务入口函数 */
                          (const char*    )"AppTaskCreate",   /* 任务名字 */
                          (uint16_t       )512,               /* 任务栈大小 */
                          (void*          )NULL,              /* 任务入口函数参数 */
                          (UBaseType_t    )1,                 /* 任务的优先级 */
                          (TaskHandle_t*  )&AppTaskCreate_Handle);  /* 任务控制块指针 */
    /* 启动任务调度 */
    if(pdPASS == xReturn)
      vTaskStartScheduler();                  /* 启动任务,开启调度 */
    else
      return -1;

    while(1);                                 /* 正常不会执行到这里 */
}
/**********************************************************************
**
  * @ 函数名  : AppTaskCreate
  * @ 功能说明: 为了方便管理,所有的任务创建函数都放在这个函数里面
  * @ 参数    : 无
  * @ 返回值  : 无
  **********************************************************************
**/
static void AppTaskCreate(void)
{
    BaseType_t xReturn = pdPASS;              /* 定义一个创建信息返回值,默认为 pdPASS */

    taskENTER_CRITICAL();                     //进入临界区

    /* 创建 MuxSem */
    MuxSem_Handle = xSemaphoreCreateMutex();
    if(NULL != MuxSem_Handle)
      printf("MuxSem_Handle 互斥量创建成功!\r\n");

    xReturn = xSemaphoreGive(MuxSem_Handle);        //给出互斥量
  // if(xReturn == pdTRUE)
  //  printf("释放信号量!\r\n");

    /* 创建 LowPriority_Task 任务 */
    xReturn = xTaskCreate((TaskFunction_t)LowPriority_Task,       /* 任务入口函数 */
                          (const char*    )"LowPriority_Task",  /* 任务名字 */
                          (uint16_t       )512,                 /* 任务栈大小 */
                          (void*          )NULL,                /* 任务入口函数参数 */
                          (UBaseType_t    )2,                   /* 任务的优先级 */
                          (TaskHandle_t*  )&LowPriority_Task_Handle); /* 任务控制块指针 */
    if(pdPASS == xReturn)
      printf("创建 LowPriority_Task 任务成功!\r\n");
```

```
  /* 创建 MidPriority_Task 任务 */
  xReturn = xTaskCreate((TaskFunction_t)MidPriority_Task,       /* 任务入口函数 */
                        (const char*   )"MidPriority_Task",   /* 任务名字 */
                        (uint16_t      )512,                  /* 任务栈大小 */
                        (void*         )NULL,                 /* 任务入口函数参数 */
                        (UBaseType_t   )3,                    /* 任务的优先级 */
                        (TaskHandle_t* )&MidPriority_Task_Handle);  /* 任务控制块指针 */
  if(pdPASS == xReturn)
    printf("创建 MidPriority_Task 任务成功!\n");

  /* 创建 HighPriority_Task 任务 */
  xReturn = xTaskCreate((TaskFunction_t)HighPriority_Task,      /* 任务入口函数 */
                        (const char*    )"HighPriority_Task", /* 任务名字 */
                        (uint16_t       )512,                 /* 任务栈大小 */
                        (void*          )NULL,                /* 任务入口函数参数 */
                        (UBaseType_t    )4,                   /* 任务的优先级 */
                        (TaskHandle_t*  )&HighPriority_Task_Handle); /* 任务控制块指针 */
  if(pdPASS == xReturn)
    printf("创建 HighPriority_Task 任务成功!\n\n");

  vTaskDelete(AppTaskCreate_Handle);           //删除 AppTaskCreate 任务

  taskEXIT_CRITICAL();                         //退出临界区
}
/**********************************************************************
  * @ 函数名  : LowPriority_Task
  * @ 功能说明: LowPriority_Task 任务主体
  * @ 参数    :
  * @ 返回值  : 无
  **********************************************************************/
static void LowPriority_Task(void* parameter)
{
  static uint32_t i;
  BaseType_t xReturn = pdPASS;        /* 定义一个创建信息返回值,默认为 pdPASS */
  while (1)
  {
    printf("LowPriority_Task 获取互斥量\n");
    //获取互斥量 MuxSem,若没获取到则一直等待
      xReturn = xSemaphoreTake(MuxSem_Handle,       /* 互斥量句柄 */
                               portMAX_DELAY);      /* 等待时间 */
    if(pdTRUE == xReturn)
    printf("LowPriority_Task Runing\n\n");

    for(i=0;i<4000000;i++)                          //模拟低优先级任务占用互斥量
        {
            taskYIELD();                            //发起任务调度
```

```
        }

    printf("LowPriority_Task 释放互斥量!\r\n");
    xReturn = xSemaphoreGive(MuxSem_Handle);        //给出互斥量

        LED1_TOGGLE;

    vTaskDelay(1000);
  }
}
/**********************************************************************
  * @ 函数名  : MidPriority_Task
  * @ 功能说明: MidPriority_Task 任务主体
  * @ 参数    :
  * @ 返回值  : 无
  **********************************************************************/
static void MidPriority_Task(void* parameter)
{
  while (1)
  {
   printf("MidPriority_Task Runing\n");
   vTaskDelay(1000);
  }
}
/**********************************************************************
  * @ 函数名  : HighPriority_Task
  * @ 功能说明: HighPriority_Task 任务主体
  * @ 参数    :
  * @ 返回值  : 无
  **********************************************************************/
static void HighPriority_Task(void* parameter)
{
  BaseType_t xReturn = pdTRUE;              /* 定义一个创建信息返回值,默认为 pdPASS */
  while (1)
  {
    printf("HighPriority_Task 获取互斥量\n");
    //获取互斥量 MuxSem,若没获取到则一直等待
      xReturn = xSemaphoreTake(MuxSem_Handle,         /* 互斥量句柄 */
                               portMAX_DELAY);        /* 等待时间 */
    if(pdTRUE == xReturn)
      printf("HighPriority_Task Runing\n");
        LED1_TOGGLE;

    printf("HighPriority_Task 释放互斥量!\r\n");
    xReturn = xSemaphoreGive(MuxSem_Handle);          //给出互斥量

    vTaskDelay(1000);
```

```
  }
}
/*********************************************************************
  * @ 函数名  : BSP_Init
  * @ 功能说明: 板级外设初始化,所有板子上的初始化均可放在这个函数里面
  * @ 参数    : 无
  * @ 返回值  : 无
  *********************************************************************/
static void BSP_Init(void)
{
/*
 * STM32 中断优先级分组为 4,即 4bit 都用来表示抢占优先级,范围为 0～15
 * 优先级分组只需要分组一次即可,以后如果有其他的任务需要用到中断,
 * 都统一用这个优先级分组,千万不要再分组
 */
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);

/* LED 初始化 */
LED_GPIO_Config();

/* 串口初始化 */
Debug_USART_Config();

 /* 按键初始化 */
 Key_GPIO_Config();

}

/*********************************** END OF FILE ***********************************/
```

本代码展示了如何在 FreeRTOS 中使用互斥量来实现任务间对共享资源的安全访问。使用互斥量,避免了多个任务同时访问共享资源导致的数据竞争问题。互斥量也实现了优先级继承功能,当一个高优先级任务被阻塞等待互斥量时,拥有互斥量的低优先级任务会临时提升其优先级,避免优先级反转。

示例说明如下:

(1) 任务创建。

主任务 AppTaskCreate 创建三个不同优先级的任务(低、中、高)以及互斥量 MuxSem_Handle。

(2) 低优先级任务。

低优先级任务 LowPriority_Task 尝试获取互斥量,获取后模拟占用一定的时间,完成后释放互斥量。

任务在每次迭代后等待 1s。

(3) 中优先级任务。

中优先级任务 MidPriority_Task 每秒打印一次运行状态,但不涉及互斥量的获取。

(4) 高优先级任务。

高优先级任务 HighPriority_Task 同样尝试获取互斥量,获取后打印运行状态并释放互斥量。

任务在每次迭代后等待 1s。

(5) 系统资源初始化。

在 BSP_Init 中完成开发板上的初始化工作,如 LED、串口和按键的配置。

此代码示例通过互斥量确保高优先级任务仍能在低优先级任务占用资源时适当地调度,演示了互斥量在任务同步中扮演的重要角色。

在 FreeRTOS 中创建和管理互斥量来保护共享资源,确保多个任务对资源的安全访问,同时保障系统的实时响应和数据一致性。

2. 互斥量实例下载与运行结果

将程序编译好,用 USB 线连接计算机和 STM32 开发板的 USB 接口(对应丝印为 USB 转串口),用 DAP 仿真器把配套程序下载到野火 STM32 开发板(这里为野火霸天虎 STM32F407 开发板),互斥量程序下载界面如图 5-15 所示。

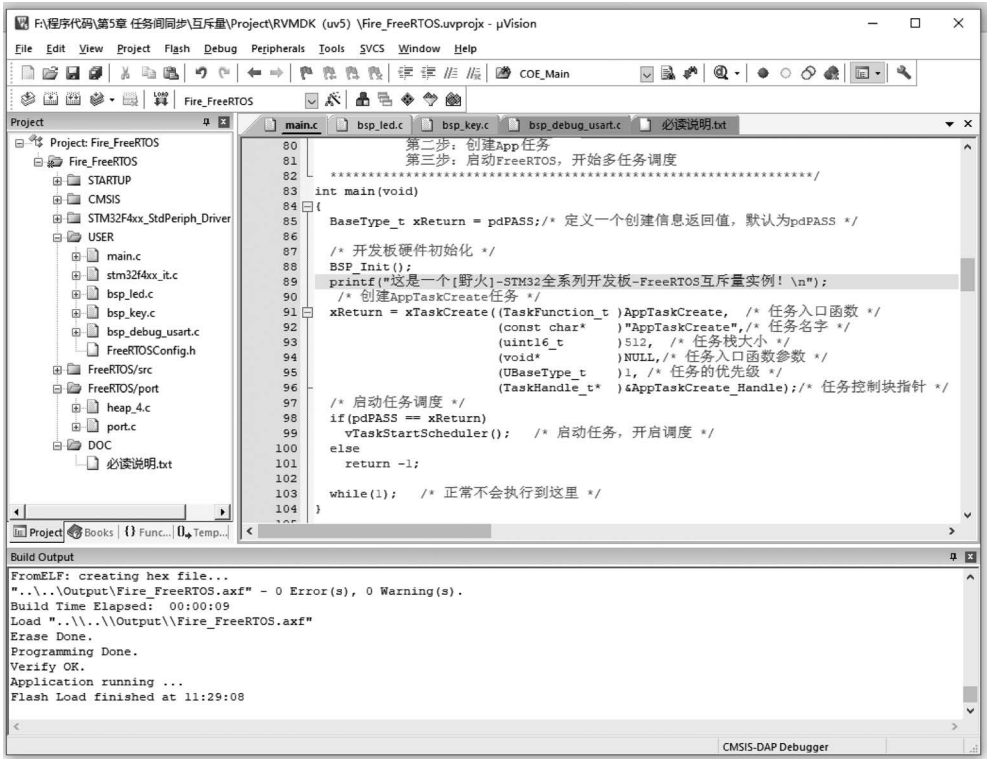

图 5-15 互斥量程序下载界面

在计算机上打开野火串口调试助手 FireTools,然后复位开发板就可以在调试助手中看到串口的打印信息,它里面输出了信息表明任务正在运行中。按下开发板的按键,串口打印

任务运行的信息，表明两个任务同步成功。

互斥量实例运行结果如图 5-16 所示。

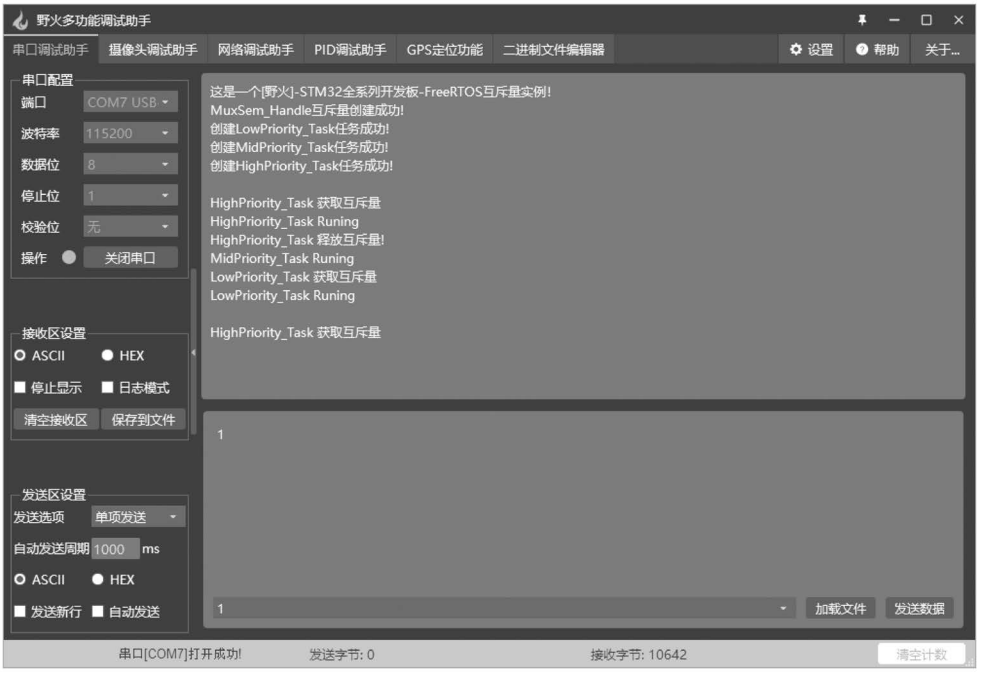

图 5-16　互斥量实例运行结果

5.3　FreeRTOS 事件组

事件组(event group)是 FreeRTOS 中另一种进程间通信技术。与前面介绍的队列、信号量等进程间通信技术相比，它具有不同的特点。事件组适用于多个事件触发一个或多个任务运行，可以实现事件的广播，还可以实现多个任务的同步运行。

在 FreeRTOS 中，事件组是一种数据结构和同步机制，用于任务之间的事件通知和同步。事件组由一组位(bit)组成，每一位可以表示系统中的某个事件状态。当事件发生时，相关位可以被设置，在满足特定条件时，其他任务可以被唤醒或做出相应的处理。

(1) 主要特性和功能。

① 事件位(event bits)。

事件组由一组位组成，通常为 32 位(对于 32 位系统)，每一位代表一个独立的事件。事件位可以被独立设置或清除。

② 同步和通知。

一个或多个任务可以等待一个或多个事件位的设置，通过这种方式实现任务间的同步。等待方式可以是“任意一个”位(逻辑或)被设置，也可以是“所有”位(逻辑与)被设置。

③ 任务唤醒。

当一个或多个事件位设置后，等待这些事件的任务可以被唤醒继续执行。

④ 超时机制。

任务可以指定等待某些事件位的时间，如果在指定时间内事件未发生，任务将超时并继续执行其他操作。

(2) 核心 API 函数。

① 创建事件组：xEventGroupCreate 用于创建一个新的事件组。

② 设置/清除事件位。

xEventGroupSetBits：设置一个或多个事件位。

xEventGroupClearBits：清除一个或多个事件位。

③ 等待事件位。

xEventGroupWaitBits：等待一个或多个事件位被设置，可选择逻辑与或逻辑或方式等待。

④ 获取事件位状态。

xEventGroupGetBits：获取当前事件组的事件位状态。

5.3.1 事件组的原理和功能

事件组是 FreeRTOS 中的一种进程间通信技术，允许任务等待一个或多个事件的组合，并在事件发生时解除所有等待该事件的任务的阻塞状态，适用于多任务之间的同步和事件广播。下面介绍事件组的功能特点和工作原理。

1. 事件组的功能特点

事件组有如下特点。

(1) 一次进程间通信通常只处理一个事件，例如等待一个按键的按下，而不能等待多个事件的发生，例如等待 Key1 键和 Key2 键先后按下。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。

(2) 可以有多个任务等待一个事件的发生，但是在事件发生时，只能解除最高优先级任务的阻塞状态，而不能同时解除多个任务的阻塞状态。也就是说，队列或信号量具有排他性，不能解决某些特定的问题。例如当某个事件发生时，需要两个或多个任务同时解除阻塞状态作出响应。

事件组是 FreeRTOS 中另一种进程间通信技术，与队列和信号量不同，它有自己的特点，具体如下。

(1) 事件组允许任务等待一个或多个事件的组合。例如，先后按下 Key1 键和 Key2 键，或只按下其中一个键。

(2) 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 LED1 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作出响应(或运算关系)，或一组事件都发生后作出响应(与运算关系)；将事件广播给多个任务；多个任务之间的同步。

2. 事件组的工作原理

事件组是 FreeRTOS 中的一种对象，FreeRTOS 中默认是可以使用事件组的，无须设置参数。使用之前需要用函数 xEventGroupCreate 或 xEventGroupCreateStatic 创建事件组对象。

一个事件组对象有一个内部变量存储事件标志，变量的位数与参数 configUSE_16_BIT_TICKS 有关，当 configUSE_16_BIT_TICKS 为 0 时，这个变量是 32 位的，否则，是 16 位的。STM32 MCU 是 32 位的，所以事件组内部变量是 32 位的。

事件标志只能是 0 或 1，用单独的一个位存储。一个事件组中的所有事件标志保存在个 EventBits_t 类型的变量里，所以一个事件又称为一个“事件位”。在一个事件组变量中，如果一个事件位被置为 1，就表示这个事件发生了；如果被置为 0，就表示这个事件还未发生。

32 位的事件组变量存储结构如图 5-17 所示。其中的 31～24 位是保留的，23～0 位是事件位。每一位是一个事件标志(event flag)，事件发生时，相应的位会被置为 1。所以，32 位的事件组最多可以处理 24 个事件。

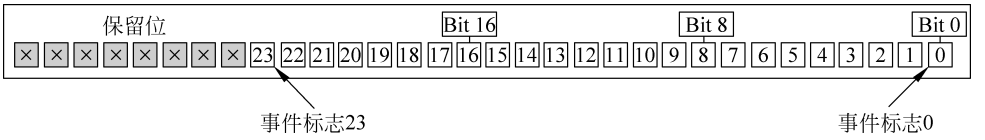

图 5-17　EventBits_t 类型事件组变量存储结构(32 位)

事件组基本工作原理示意图如图 5-18 所示。各部分的功能和工作流程如下。

(1) 设置事件组中的位与某个事件对应，如 EventA 对应 Bit2，EventB 对应 Bit0。在检测到事件发生时，通过函数 xEventGroupSetBits 将相应的位置为 1，表示事件发生了。

(2) 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)，也可以是某个事件发生(事件位的或运算)。

(3) 假设图 5-18 中的 Task1 和 Task2 都在阻塞状态等待各自的事件发生，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，可以使多个任务同时解除阻塞后运行。

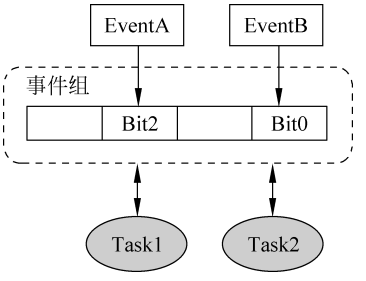

图 5-18　事件组基本工作原理示意图

事件组是 FreeRTOS 中的一种进程间通信技术，用于管理和同步多个任务。图 5-18 中展示了事件组的基本工作原理，具体如下：

(1) 事件组：事件组包含多个事件位，每个位用于表示一个事件的状态。图 5-18 中展

示了一个事件组，其中包含两个事件位：Bit2 和 Bit0。

（2）事件 A 和事件 B：事件 A 和事件 B 分别对应事件组中的 Bit2 和 Bit0。当事件 A 发生时，Bit2 位被置为 1；当事件 B 发生时，Bit0 位被置为 1。

（3）任务 Task1 和 Task2：任务 Task1 和 Task2 分别等待事件 A 和事件 B 的发生。Task1 等待 Bit2 位被置为 1，Task2 等待 Bit0 位被置为 1。

（4）事件触发与任务解除阻塞。

当事件 A 发生时，通过函数 xEventGroupSetBits 将事件组中的 Bit2 位置为 1，表示事件 A 发生了。此时，Task1 被解除阻塞状态，开始运行。

同样，当事件 B 发生时，通过函数 xEventGroupSetBits 将事件组中的 Bit0 位置为 1，表示事件 B 发生了。此时，Task2 被解除阻塞状态，开始运行。

（5）同步与广播功能：事件组允许多个任务同时等待不同的事件，并在事件发生时同步解除阻塞状态。例如，当 Bit2 和 Bit0 都被置为 1 时，Task1 和 Task2 都会被解除阻塞状态，开始运行。这种特性使事件组适用于多任务之间的同步和事件广播。

通过事件组，FreeRTOS 能够有效地管理和同步多个任务，确保系统在处理复杂事件时的实时性和效率。

除了图 5-18 中的基本功能，事件组还可以使多个任务同步运行。事件组允许任务等待多个事件的组合，并在所有指定事件都发生时同时解除阻塞状态，实现多任务的同步。例如，可以设置多个任务等待同一个事件组中的不同事件，当这些事件都发生时，所有等待的任务会同时从阻塞状态中被解除，从而同步执行。这种特性在需要多个任务同时响应某些条件时非常有用，如复杂的状态监控和多任务协调。

5.3.2 事件组的应用场景

FreeRTOS 的事件组用于事件类型的通信，无数据传输，也就是说，可以用事件来做标志位，判断某些事件是否发生了，然后根据结果进行处理。为什么不直接用变量做标志呢？那样岂不是更有效率？若是在裸机编程中，用全局变量是最有效的方法，但是在操作系统中，使用全局变量就要考虑以下问题了。

（1）如何对全局变量进行保护？如何处理多任务同时对它进行访问的情况？

（2）如何让内核对事件进行有效管理？如果使用全局变量，就需要在任务中轮询查看事件是否发送，这会造成 CPU 资源的浪费。此外，用户还需要自己去实现等待超时机制。所以，在操作系统中最好还是使用系统提供的通信机制。

在某些场合，可能需要多个事件发生后才能进行下一步操作，比如一些危险机器的启动，需要检查各项指标，当指标不达标时就无法启动。但是检查各个指标时，不会立刻检测完毕，所以需要事件来做统一的等待。当所有的事件都完成时，机器才允许启动，这只是事件的应用之一。

事件可用于多种场合，能够在一定程度上替代信号量，用于任务与任务间、中断与任务间的同步。一个任务或中断服务例程发送一个事件给事件对象，而后等待的任务被唤醒并

对相应的事件进行处理。但是事件与信号量不同的是，事件的发送操作是不可累计的，而信号量的释放动作是可累计的。事件的另一个特性是，接收任务可等待多个事件，即多个事件对应一个任务或多个任务。同时按照任务等待的参数，可选择是“逻辑或”触发还是“逻辑与”触发。这个特性也是信号量等所不具备的，信号量只能识别单一同步动作，而不能同时等待多个事件的同步。

各个事件可分别发送或一起发送给事件对象，而任务可以等待多个事件，任务仅对感兴趣的事件进行关注。当有它们感兴趣的事件发生并且符合条件时，任务将被唤醒并进行后续的处理动作。

在 FreeRTOS 中，事件组是一种强大的同步机制，用于任务之间的通信和协调。与信号量(semaphore)不同，事件组允许多个任务同步到多个事件标志上，可以同时等待多个事件的发生，并支持位操作，以实现复杂的同步逻辑。以下是事件组的典型应用场景。

1. 任务同步

多个任务可能需要在某一特定时刻同步执行，通过事件组中的事件标志，可以实现这一功能。例如，有两个任务分别在处理不同的数据，当这两个任务都完成时，可以设置事件标志，通知协调任务进行下一步操作。

```
EventGroupHandle_t xEventGroup;
#define TASK1_COMPLETE_BIT (1 << 0)
#define TASK2_COMPLETE_BIT (1 << 1)

void Task1(void * pvParameters) {
    for (;;) {
        // 任务 1 的处理逻辑
        // ...
        xEventGroupSetBits(xEventGroup, TASK1_COMPLETE_BIT);
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

void Task2(void * pvParameters) {
    for (;;) {
        // 任务 2 的处理逻辑
        // ...
        xEventGroupSetBits(xEventGroup, TASK2_COMPLETE_BIT);
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

void CoordinatorTask(void * pvParameters) {
    const EventBits_t xBitsToWaitFor = (TASK1_COMPLETE_BIT | TASK2_COMPLETE_BIT);
    for (;;) {
        EventBits_t xEventGroupValue = xEventGroupWaitBits(
            xEventGroup,
            xBitsToWaitFor,
            pdTRUE,            // 清除已设置的事件位
```

```
            pdTRUE,             // 等待所有事件位被设置
            portMAX_DELAY
        );

        if ((xEventGroupValue & xBitsToWaitFor) == xBitsToWaitFor) {
            // 两个任务都完成,进行协调操作
            // ...
        }
    }
}
```

2. 多事件触发

一个任务可能需要等待多个独立事件的发生,使用事件组可以实现这一功能。例如,在一个传感器网络中,一个任务需要等待不同传感器的数据到达,然后进行统一处理。

```
EventGroupHandle_t xEventGroup;

#define SENSOR1_DATA_BIT (1 << 0)
#define SENSOR2_DATA_BIT (1 << 1)

void Sensor1Task(void *pvParameters) {
    for (;;) {
        // 读取传感器 1 的数据
        // ...
        xEventGroupSetBits(xEventGroup, SENSOR1_DATA_BIT);
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

void Sensor2Task(void *pvParameters) {
    for (;;) {
        // 读取传感器 2 的数据
        // ...
        xEventGroupSetBits(xEventGroup, SENSOR2_DATA_BIT);
        vTaskDelay(pdMS_TO_TICKS(700));
    }
}

void DataProcessingTask(void *pvParameters) {
    for (;;) {
        EventBits_t xEventGroupValue = xEventGroupWaitBits(
            xEventGroup,
            SENSOR1_DATA_BIT | SENSOR2_DATA_BIT,
            pdTRUE, // 清除已设置的事件位
            pdFALSE, // 任一事件位被设置即可退出等待
            portMAX_DELAY
        );

        if (xEventGroupValue & SENSOR1_DATA_BIT) {
            // 处理传感器 1 的数据
```

```
            // ...
        }

        if (xEventGroupValue & SENSOR2_DATA_BIT) {
            // 处理传感器 2 的数据
            // ...
        }
    }
}
```

3. 任务与中断同步

中断服务程序可以通过事件组通知任务处理特定事件，例如当数据到达时，中断服务程序设置某个事件标志，通知任务去处理数据。

```
EventGroupHandle_t xEventGroup;

#define DATA_READY_BIT (1 << 0)

void ISR_Handler(void) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    xEventGroupSetBitsFromISR(xEventGroup, DATA_READY_BIT, &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}

void DataHandlerTask(void * pvParameters) {
    for (;;) {
        xEventGroupWaitBits(xEventGroup, DATA_READY_BIT, pdTRUE, pdFALSE, portMAX_DELAY);
        // 处理数据
        // ...
    }
}
```

4. 状态监控

事件组可以用于监控系统的各种状态，例如任务执行状态、硬件状态等，汇总这些状态信息以便于任务进行决策。

```
EventGroupHandle_t xEventGroup;

#define TASK_RUNNING_BIT (1 << 0)
#define HARDWARE_READY_BIT (1 << 1)

void MonitoringTask(void * pvParameters) {
    for (;;) {
        EventBits_t xEventGroupValue = xEventGroupGetBits(xEventGroup);

        if ((xEventGroupValue & TASK_RUNNING_BIT) != 0) {
            // 任务正在运行
        }

        if ((xEventGroupValue & HARDWARE_READY_BIT) != 0) {
```

```
            // 硬件准备就绪
        }

        vTaskDelay(pdMS_TO_TICKS(500));
    }
}
```

事件组在 FreeRTOS 中提供了一种强大而灵活的方式来处理多任务同步、事件触发和状态监控等复杂场景。通过使用事件组，开发者可以更有效地管理任务之间、任务与中断之间的通信和协调，从而提高系统的效率和可靠性。

5.3.3 事件组运作机制

在嵌入式系统中，事件组允许任务接收和处理一个或多个事件，用户可以根据需要选择接收单个或多个事件。

接收事件的过程如下：

(1) 选择事件类型。

当任务接收事件时，可以根据感兴趣的事件类型选择接收单个或多个事件。

(2) 清除事件选项。

接收事件成功后，任务需要决定是否清除已接收的事件类型。这可以通过 xClearOnExit 选项来实现：

① 如果设置了 xClearOnExit，则在成功接收事件后，相关的事件位会被自动清除。

② 如果未设置 xClearOnExit，则接收到的事件位不会被清除，需要用户显式地清除。

(3) 读取模式。

用户可以自定义读取模式，通过传入参数 xWaitForAllBits 来选择。

① 等待所有感兴趣的事件：如果设置了 xWaitForAllBits，任务将等待所有设置的事件位都被置位。

② 等待任意一个感兴趣的事件：如果未设置 xWaitForAllBits，任务只需等待任意一个设置的事件位被置位。

设置事件时，对指定事件写入指定的事件类型，设置事件集合的对应事件位为 1，可以一次同时写多个事件类型，设置事件成功可能会触发任务调度。

清除事件时，根据写入的参数事件句柄和待清除的事件类型，对事件对应位进行清零操作。事件不与任务相关联，事件相互独立，一个 32 位的变量用于标识该任务发生的事件类型，其中每一位表示一种事件类型(0 表示该事件类型未发生，1 表示该事件类型已经发生)，共有 24 种事件类型，具体如图 5-19 所示。

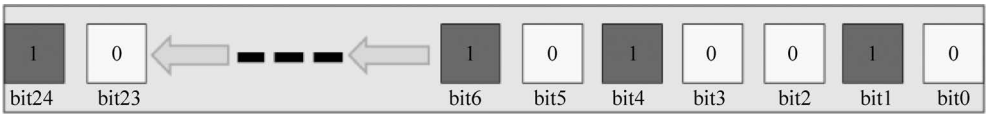

图 5-19 事件集合 set(一个 32 位的变量)

事件唤醒机制，即任务因为等待某个或者多个事件发生而进入阻塞态，当事件发生时会被唤醒，其过程具体如图 5-20 所示。

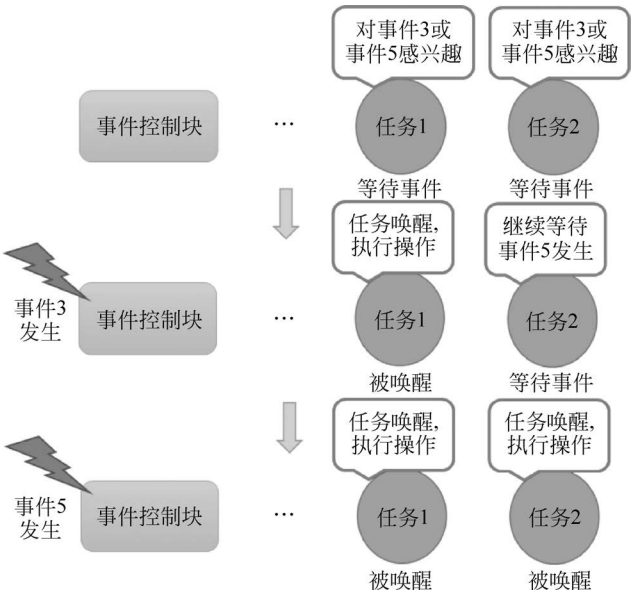

图 5-20　事件唤醒任务示意图

任务 1 对事件 3 或事件 5 感兴趣(逻辑或)，当发生其中某一个事件时都会被唤醒，并且执行相应操作。任务 2 对事件 3 与事件 5 感兴趣(逻辑与)，当且仅当事件 3 与事件 5 都发生时，任务 2 才会被唤醒，如果只有其中一个事件发生，那么任务还是会继续等待另一个事件发生。如果在接收事件函数中设置了清除事件位 xClearOnExit，那么当任务唤醒后将把事件 3 和事件 5 的事件标志清零，否则事件标志将依然存在。

5.3.4　事件组控制块

事件标志组存储在一个 EventBitst 类型的变量中，该变量在事件组结构体中定义。

如果宏 configUSE16BIT T ICKS 定义为 1，那么变量 uxEventBits 就是 16 位的，其中有 8 个位用来存储事件组；如果宏 configUSE16BIT TICKS 定义为 0，那么变量 uxEventBits 就是 32 位的，其中有 24 个位用来存储事件组，每一位代表一个事件的发生与否，利用逻辑或、逻辑与等实现不同事件的不同唤醒处理。在 STM32 中，uxEventBits 是 32 位的，所以有 24 个位用来实现事件组。除了事件标志组变量之外，FreeRTOS 还使用一个链表来记录等待事件的任务，所有在等待此事件的任务均会被挂载到等待事件列表 xTasksWaitingForBits 中。

事件控制块代码清单如下：

```
1  typedef struct xEventGroupDefinition {
2  EventBits_t uxEventBits;
3  List_t xTasksWaitingForBits;
4
```

```
5  #if(configUSE_TRACE_FACILITY == 1)
6  UBaseType_t uxEventGroupNumber;
7  #endif
8
9  #if((configSUPPORT_STATIC_ALLOCATION == 1) \
10 && (configSUPPORT_DYNAMIC_ALLOCATION == 1))
11 uint8_t ucStaticallyAllocated;
12 #endif
13 } EventGroup_t;
```

FreeRTOS中的事件控制块(Event Control Block,ECB)是用于管理事件组的结构体。事件组允许多个任务在同一个或不同的事件组上等待一个或多个事件标志。任务创建的事件组用于事件的同步,即任务可以等待某个事件位被设置或清除,进而实现任务之间的通信和同步。

事件控制块代码说明如下:

(1) 数据结构定义。

```
typedef struct xEventGroupDefinition {
    EventBits_t uxEventBits;                // 事件位,用于存储各个事件标志的状态
    List_t xTasksWaitingForBits;            // 任务列表,存储等待这些事件位的任务

    #if(configUSE_TRACE_FACILITY == 1)
    UBaseType_t uxEventGroupNumber;         // 用于跟踪调试的事件组编号
    #endif

    #if((configSUPPORT_STATIC_ALLOCATION == 1) && \
        (configSUPPORT_DYNAMIC_ALLOCATION == 1))
    uint8_t ucStaticallyAllocated;          // 标识事件组是静态分配还是动态分配
    #endif
} EventGroup_t;
```

(2) 字段说明。

① uxEventBits。

类型:EventBits_t。

描述:用于存储各事件位的状态,每一位代表一个独立的事件。

② xTasksWaitingForBits。

类型:List_t。

描述:等待一个或多个事件位改变的任务列表。

③ uxEventGroupNumber (可选)。

类型:UBaseType_t。

描述:事件组编号,用于跟踪和调试(启用跟踪功能时)。

④ ucStaticallyAllocated (可选)。

类型:uint8_t。

描述:标明事件组是静态分配(由用户提供内存)还是动态分配(由内存管理器分配)。

事件控制块使用说明如下。

事件控制块通过 FreeRTOS 提供的 API 函数进行操作和管理。

① 创建事件组。

xEventGroupCreate：返回一个事件组句柄，并初始化事件控制块。

② 设置/清除事件位。

xEventGroupSetBits：设置一个或多个事件位。

xEventGroupClearBits：清除一个或多个事件位。

③ 等待事件位。

xEventGroupWaitBits：使任务等待一个或多个事件位的变化。

通过使用事件组和事件控制块，FreeRTOS 实现了任务间同步和通信的高级机能，确保多任务操作的协调性和实时性。事件组可以简化复杂的同步问题，提高系统的可靠性和响应速度。

5.3.5　事件组相关函数

事件组相关的函数在文件 event_groups.h 中定义，在文件 event_groups.c 中实现。事件组相关的函数在 FreeRTOS 中总是可以使用的，无须设置参数。

事件组相关的函数清单如表 5-2 所示，这些函数可分为 3 组。

表 5-2　事件组相关的函数

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，等待事件组条件成立后解除阻塞状态
	xEventGroupSync	用于多任务同步

1. 事件创建函数 xEventGroupCreate

xEventGroupCreate 用于创建一个事件组，并返回对应的句柄。要想使用该函数，必须在头文件 FreeRTOSConfig.h 定义宏 configSUPPORT_DYNAMIC_ALLOCATION 为 1（在 FreeRTOS.h 中默认定义为 1）且需要把 FreeRTOS/source/event_groups.c 这个 C 文件添加到工程中。

每个事件组只需要很少的 RAM 空间来保存事件的发生状态。如果使用函数 xEventGroupCreate 来创建一个事件,那么需要的 RAM 是动态分配的。如果使用函数 xEventGroupCreateStatic 来创建一个事件,那么需要的 RAM 是静态分配的。

事件创建函数,顾名思义,就是创建一个事件,与其他内核对象一样,都是需要先创建才能使用的资源。FreeRTOS 给用户提供了一个创建事件的函数 xEventGroupCreate,当创建一个事件时,系统会首先给用户分配事件控制块的内存空间,然后对该事件控制块进行基本的初始化,创建成功返回事件句柄;创建失败返回 NULL。

xEventGroupCreate()源码如下:

```
#if(configSUPPORT_DYNAMIC_ALLOCATION == 1)
EventGroupHandle_t xEventGroupCreate(void)
{
    EventGroup_t * pxEventBits;
    /* 分配事件控制块的内存 */
    pxEventBits = (EventGroup_t * ) pvPortMalloc(sizeof(EventGroup_t)); //

    if (pxEventBits != NULL) { //
        pxEventBits->uxEventBits = 0;              // 初始化事件位为 0
        vListInitialise(&(pxEventBits->xTasksWaitingForBits));   // 初始化等待任务列表

        #if(configSUPPORT_STATIC_ALLOCATION == 1)
        {
            /*
            如果同时支持静态和动态分配内存,标志该事件组为动态分配
            */
            pxEventBits->ucStaticallyAllocated = pdFALSE; // (3)
        }
        #endif

        traceEVENT_GROUP_CREATE(pxEventBits);        // 跟踪调试信息
    } else {
        traceEVENT_GROUP_CREATE_FAILED();            // 内存分配失败时的调试信息
    }

    return (EventGroupHandle_t) pxEventBits;         // 返回事件组句柄
}
#endif
```

xEventGroupCreate 是 FreeRTOS 中用于创建事件组的函数。事件组是一个用于任务间同步的机制,通过事件标志管理多个任务的执行。该函数分配并初始化事件控制块,使得后续的任务能够使用该事件组进行同步和通信。

字段和代码行说明如下:

(1) 内存分配。

“pxEventBits=(EventGroup_t *) pvPortMalloc(sizeof(EventGroup_t));”:使用动态内存分配功能事件控制块分配内存。

(2) 内存分配成功检查。

if(pxEventBits != NULL)：检查内存分配是否成功。

(3) 初始化事件控制块。

“pxEventBits-> uxEventBits=0;”：初始化事件位为0。

“vListInitialise(&(pxEventBits-> xTasksWaitingForBits));”：初始化等待任务列表。

(4) 静态分配标识。

“pxEventBits-> ucStaticallyAllocated=pdFALSE;”：如果同时支持静态和动态分配，标记此事件组为动态分配。

(5) 跟踪调试信息。

“traceEVENT_GROUP_CREATE(pxEventBits);”：记录创建事件组的调试信息。

“traceEVENT_GROUP_CREATE_FAILED();”：记录创建事件组失败的调试信息。

(6) 返回值。

“return (EventGroupHandle_t) pxEventBits;”：返回事件组的句柄，即指向新创建的事件控制块的指针。

xEventGroupCreate函数通过动态内存分配为事件组创建和初始化事件控制块。当分配成功时，该事件控制块包含一个初始化的事件位和任务等待列表，可用于任务间的同步和通信。如果内存分配失败，函数会记录失败情况并返回NULL。该函数为FreeRTOS提供了动态创建事件组的能力，使得事件组的使用更加灵活和高效。

2. 事件删除函数 vEventGroupDelete

在很多场合，某些事件是只用一次的，就好比事件应用场景中的危险机器的启动，假如各项指标都达到了，并且机器启动成功了，这个事件之后可能就没用了，就可以进行销毁了。想要删除事件怎么办？FreeRTOS给用户提供了一个删除事件的函数——vEventGroupDelete，使用它就能将事件进行删除。当系统不再使用事件对象时，可以通过删除事件对象控制块来释放系统资源。

vEventGroupDelete()函数原型如下：

```
void vEventGroupDelete(EventGroupHandle_t xEventGroup)
```

3. 事件组置位函数 xEventGroupSetBits

xEventGroupSetBits用于置位事件组中指定的位，当位被置位之后，阻塞在该位上的任务将会被解锁。使用该函数接口时，通过参数指定的事件标志来设定事件的标志位，然后遍历等待在事件对象上的事件等待列表，判断是否有任务的事件激活要求与当前事件对象标志值匹配，如果有，则唤醒该任务。简单来说，就是设置用户自己定义的事件标志位为1，并且看有没有任务在等待这个事件，有的话就唤醒它。

EventGroupSetBits()函数原型如下：

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

4. 事件组置位函数 xEventGroupSetBitsFromISR

xEventGroupSetBitsFromISR是xEventGroupSetBits的中断版本，用于置位事件组中

定的位。置位事件组中的标志位是一个不确定的操作，因为阻塞在事件组的标志位上的任务的个数是不确定的。FreeRTOS是不允许不确定的操作在中断和临界段中发生的，所以EventGroupSetBitsFromISR给FreeRTOS的守护任务发送一个消息，让置位事件组的操作在守护任务里面完成，守护任务是基于调度锁而非临界段的机制来实现的。

需要注意的是，正如上文提到的那样，在中断中事件标志的置位是在守护任务（也叫软件定时器服务任务）中完成的，因此FreeRTOS的守护任务与其他任务一样，都是系统调度器根据其优先级进行任务调度的，但守护任务的优先级必须比任何任务的优先级都要高，保证在需要的时候能立即切换任务从而达到快速处理的目的。因为这是在中断中让事件标志位置位，其优先级由FreeRTOSConfig.h中的宏configTIMER_TASK_PRIORITY来定义。

其实xEventGroupSetBitsFromISR函数真正调用的也是xEventGroupSetBits函数，只不过是在守护任务中进行调用的，所以它实际上执行的上下文环境依旧是在任务中。

要想使用该函数，必须把configUSE_TIMERS和INCLUDE_xTimerPendFunctionCall这些宏在FreeRTOSConfig.h中都定义为1，并且把FreeRTOS/source/event_groups.c这个C文件添加到工程中编译。

xEventGroupSetBitsFromISR函数原型如下：

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

5. 等待事件函数xEventGroupWaitBits

既然标记了事件的发生，那么怎么知道到底有没有发生，这也需要一个函数来获取事件是否已经发生。FreeRTOS提供了一个等待指定事件的函数——EventGroupWaitBits，通过这个函数，任务可以知道事件标志组中有哪些位，有什么事件发生了，然后通过“逻辑与”“逻辑或”等操作对感兴趣的事件进行获取，并且这个函数实现了等待超时机制，当且仅当任务等待的事件发生时，任务才能获取到事件信息。

在这段时间中，如果事件一直没发生，该任务将保持阻塞状态以等待事件发生。当其他任务或中断服务程序往其等待的事件设置对应的标志位时，该任务将自动由阻塞态转为就绪态。

当任务等待的时间超过了指定的阻塞时间时，即使事件还未发生，任务也会自动从阻塞态转移为就绪态。这体现了操作系统的实时性。如果事件正确获取（等待到）则返回对应的事件标志位，由用户判断再做处理，因为在事件超时的时候也会返回一个不能确定的事件值，所以需要判断任务所等待的事件是否真的发生。

EventGroupWaitBits用于获取事件组中的一个或多个事件发生标志，当要读取的事件标志位没有被置位时，任务将进入阻塞等待状态。要想使用该函数，必须把reeRTOS/source/event_groups.c这个C文件添加到工程中。

xEventGroupWaitBits()函数原型如下：

```
EventBits_t xEventGroupWaitBits(const EventGroupHandle_t xEventGroup,
                                const EventBits_t uxBitsToWaitFor,
                                const BaseType_t xClearOnExit,
```

```
                                const BaseType_t xWaitForAllBits,
                                TickType_t xTicksToWait)
```

5.3.6　FreeRTOS 事件组应用实例

事件组实例在 FreeRTOS 中创建了两个任务，一个是设置事件任务，一个是等待事件任务，两个任务独立运行。设置事件任务通过检测按键的按下情况设置不同的事件标志位，等待事件任务则获取这两个事件标志位，并且判断两个事件是否都发生，如果是则输出相应信息，LED 进行翻转。等待事件任务的等待时间是 portMAX_DELAY，一直在等待事件的发生，等待到事件之后清除对应的事件标记位。

1. 事件源代码

```
/**************************************************************************
  * @file  main.c
  * @brief  FreeRTOS V9.0.0 + STM32 事件
  * 实验平台:野火 STM32F407 霸天虎开发板
**************************************************************************/

/**************************************************************************
*                                包含的头文件
**************************************************************************/
/* FreeRTOS 头文件 */
#include "FreeRTOS.h"
#include "task.h"
#include "event_groups.h"
/* 开发板硬件 bsp 头文件 */
#include "bsp_led.h"
#include "bsp_debug_usart.h"
#include "bsp_key.h"
/****************************** 任务句柄 ********************************/
/*
  * 任务句柄是一个指针,用于指向一个任务,当任务创建好之后,它就具有了一个任务句柄
  * 以后要想操作这个任务都需要通过这个任务句柄,如果是自身的任务操作自己,那么
  * 这个句柄可以为 NULL
  */
static TaskHandle_t AppTaskCreate_Handle = NULL;          /* 创建任务句柄 */
static TaskHandle_t LED_Task_Handle = NULL;               /* LED_Task 任务句柄 */
static TaskHandle_t KEY_Task_Handle = NULL;               /* KEY_Task 任务句柄 */

/****************************** 内核对象句柄 ****************************/
/*
 * 信号量、消息队列、事件标志组、软件定时器这些都属于内核的对象,要想使用这些内核
 * 对象,必须先创建,创建成功之后会返回一个相应的句柄。实际上就是一个指针,后续
 * 就可以通过这个句柄操作这些内核对象
 *
 * 内核对象其实就是一种全局的数据结构,通过这些数据结构可以实现任务间的通信、
 * 任务间的事件同步等各种功能。这些功能的实现是通过调用这些内核对象的函数
 * 来完成的
```

```
 *
 */
static EventGroupHandle_t Event_Handle = NULL;

/************************** 全局变量声明 ****************************/
/*
  * 当写应用程序时,可能需要用到一些全局变量
*/
/************************** 宏定义 ********************************/
/*
  * 在写应用程序时,可能需要用到一些宏定义
  */
#define KEY1_EVENT (0x01 << 0)              //设置事件掩码的位 0
#define KEY2_EVENT (0x01 << 1)              //设置事件掩码的位 1

/*
*************************************************************************
*                                函数声明
*************************************************************************
*/
static void AppTaskCreate(void);            /* 用于创建任务 */

static void LED_Task(void* pvParameters);   /* LED_Task 任务实现 */
static void KEY_Task(void* pvParameters);   /* KEY_Task 任务实现 */

static void BSP_Init(void);                 /* 用于初始化板载相关资源 */

/*************************************************************************
  * @brief  主函数
  * @param 无
  * @retval 无
  * @note  第 1 步:开发板硬件初始化
           第 2 步:创建 App 任务
           第 3 步:启动 FreeRTOS,开始多任务调度
  ************************************************************/
int main(void)
{
  BaseType_t xReturn = pdPASS;        /* 定义一个创建信息返回值,默认为 pdPASS */

  /* 开发板硬件初始化 */
  BSP_Init();
    printf("这是一个 FreeRTOS 事件标志组实例!\n");
  /* 创建 AppTaskCreate 任务 */
  xReturn = xTaskCreate((TaskFunction_t)AppTaskCreate,       /* 任务入口函数 */
                        (const char*    )"AppTaskCreate",  /* 任务名字 */
                        (uint16_t       )512,              /* 任务栈大小 */
                        (void*          )NULL,             /* 任务入口函数参数 */
                        (UBaseType_t    )1,                /* 任务的优先级 */
                        (TaskHandle_t*  )&AppTaskCreate_Handle);  /* 任务控制块指针 */
  /* 启动任务调度 */
```

```
  if(pdPASS == xReturn)
    vTaskStartScheduler();          /* 启动任务,开启调度 */
  else
    return -1;

  while(1);                         /* 正常不会执行到这里 */
}
/**********************************************************************
  * @ 函数名  : AppTaskCreate
  * @ 功能说明: 为了方便管理,所有的任务创建函数都放在这个函数里面
  * @ 参数    : 无
  * @ 返回值  : 无
  **********************************************************************/
static void AppTaskCreate(void)
{
  BaseType_t xReturn = pdPASS;      /* 定义一个创建信息返回值,默认为 pdPASS */

  taskENTER_CRITICAL();             //进入临界区

  /* 创建 Event_Handle */
  Event_Handle = xEventGroupCreate();
  if(NULL != Event_Handle)
    printf("Event_Handle 事件创建成功!\r\n");

  /* 创建 LED_Task 任务 */
  xReturn = xTaskCreate((TaskFunction_t)LED_Task,         /* 任务入口函数 */
                        (const char*    )"LED_Task",    /* 任务名字 */
                        (uint16_t       )512,           /* 任务栈大小 */
                        (void*          )NULL,          /* 任务入口函数参数 */
                        (UBaseType_t    )2,             /* 任务的优先级 */
                        (TaskHandle_t*  )&LED_Task_Handle);  /* 任务控制块指针 */
  if(pdPASS == xReturn)
    printf("创建 LED_Task 任务成功!\r\n");

  /* 创建 KEY_Task 任务 */
  xReturn = xTaskCreate((TaskFunction_t)KEY_Task,         /* 任务入口函数 */
                        (const char*    )"KEY_Task",    /* 任务名字 */
                        (uint16_t       )512,           /* 任务栈大小 */
                        (void*          )NULL,          /* 任务入口函数参数 */
                        (UBaseType_t    )3,             /* 任务的优先级 */
                        (TaskHandle_t*  )&KEY_Task_Handle);  /* 任务控制块指针 */
  if(pdPASS == xReturn)
    printf("创建 KEY_Task 任务成功!\n");

  vTaskDelete(AppTaskCreate_Handle);          //删除 AppTaskCreate 任务

  taskEXIT_CRITICAL();                        //退出临界区
}

/**********************************************************************
```

```
  * @ 函数名   : LED_Task
  * @ 功能说明: LED_Task 任务主体
  * @ 参数     :
  * @ 返回值   : 无
  ********************************************************************/
static void LED_Task(void* parameter)
{
  EventBits_t r_event; /* 定义一个事件接收变量 */
  /* 任务都是一个无限循环,不能返回 */
  while (1)
    {
    /*****************************************************************
      * 等待接收事件标志
      *
      * 如果 xClearOnExit 设置为 pdTRUE,那么在 xEventGroupWaitBits 返回之前,
      * 如果满足等待条件(如果函数返回的原因不是超时),那么在事件组中设置
      * 的 uxBitsToWaitFor 中的任何位都将被清除
      * 如果 xClearOnExit 设置为 pdFALSE,
      * 则在调用 xEventGroupWaitBits 时,不会更改事件组中设置的位
      *
      * xWaitForAllBits 如果 xWaitForAllBits 设置为 pdTRUE,则当 uxBitsToWaitFor 中
      * 的所有位都设置或指定的块时间到期时,xEventGroupWaitBits 才返回
      * 如果 xWaitForAllBits 设置为 pdFALSE,则当 uxBitsToWaitFor 中设置的任何
      * 一个位置 1 或指定的块时间到期时,xEventGroupWaitBits 都会返回
      * 阻塞时间由 xTicksToWait 参数指定
      *************************************************************/
    r_event = xEventGroupWaitBits(Event_Handle,             /* 事件对象句柄 */
                                  KEY1_EVENT|KEY2_EVENT, /* 接收线程感兴趣的事件 */
                                  pdTRUE,                   /* 退出时清除事件位 */
                                  pdTRUE,                   /* 满足感兴趣的所有事件 */
                                  portMAX_DELAY);           /* 指定超时事件,一直等 */

    if((r_event & (KEY1_EVENT|KEY2_EVENT)) == (KEY1_EVENT|KEY2_EVENT))
    {
      /* 如果接收完成并且正确 */
      printf ("KEY1 与 KEY2 都按下\n");
      LED1_TOGGLE;                                          //LED1 反转
    }
    else
      printf ("事件错误!\n");
  }
}

/*********************************************************************
  * @ 函数名   : KEY_Task
  * @ 功能说明: KEY_Task 任务主体
  * @ 参数     : 无
  * @ 返回值   : 无
  ********************************************************************/
static void KEY_Task(void* parameter)
```

```
{
    /* 任务都是一个无限循环,不能返回 */
  while (1)
  {
    if(Key_Scan(KEY1_GPIO_PORT,KEY1_PIN) == KEY_ON)       //如果 KEY2 被单击
        {
      printf ("KEY1 被按下\n");
              /* 触发一个事件 1 */
              xEventGroupSetBits(Event_Handle,KEY1_EVENT);
          }

          if(Key_Scan(KEY2_GPIO_PORT,KEY2_PIN) == KEY_ON)     //如果 KEY2 被单击
          {
      printf ("KEY2 被按下\n");
              /* 触发一个事件 2 */
              xEventGroupSetBits(Event_Handle,KEY2_EVENT);
          }
          vTaskDelay(20);                                     //每 20ms 扫描一次
    }
}

/**********************************************************************
  * @ 函数名  : BSP_Init
  * @ 功能说明: 板级外设初始化,所有板子上的初始化均可放在这个函数里面
  * @ 参数    :
  * @ 返回值  : 无
  *********************************************************************/
static void BSP_Init(void)
{
    /*
      * STM32 中断优先级分组为 4,即 4bit 都用来表示抢占优先级,范围为 0～15
      * 优先级分组只需要分组一次即可,以后如果有其他的任务需要用到中断,
      * 都统一用这个优先级分组,千万不要再分组
      */
      NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);

      /* LED 初始化 */
      LED_GPIO_Config();

      /* 串口初始化 */
      Debug_USART_Config();

  /* 按键初始化 */
  Key_GPIO_Config();

}
/****************************** END OF FILE **************************/
```

该代码展示了如何在 FreeRTOS 中使用事件标志组来实现任务间的同步与通信,其运行平台为野火 STM32 全系列开发板。主要功能是通过按键触发事件,当两个按键都被按

下时，LED 状态变化。

（1）功能。

① 硬件初始化：初始化开发板上的 LED、串口、按键。

② 创建任务：包含创建设备初始化任务、LED 任务、按键任务。

③ 事件标志组：使用 FreeRTOS 的事件标志组功能，在按键按下时设置事件标志组的位，LED 任务通过等待事件标志组的位变化来同步控制 LED 的状态。

（2）任务分工。

① AppTaskCreate：创建并管理其他任务及事件标志组，任务创建成功后自删除。

② LED_Task：等待事件标志组，当检测到两个按键事件都发生时，打印信息并反转 LED 状态。

③ KEY_Task：扫描按键状态，若按下则设置相应的事件标志组位。

这段代码可以实现按键控制 LED 的功能，并展示了如何利用 FreeRTOS 的事件标志组来进行任务间的同步与通信。LED 任务等待事件标志组的变化，按键任务设置事件标志组。通过这种方式实现高效的任务间通信。

2. 事件实例下载与运行结果

将程序编译好，用 USB 线连接计算机和 STM32 开发板的 USB 接口（对应丝印为 USB 转串口），用 DAP 仿真器把配套程序下载到野火 STM32 开发板（这里为野火霸天虎 STM32F407 开发板），事件程序下载界面如图 5-21 所示。

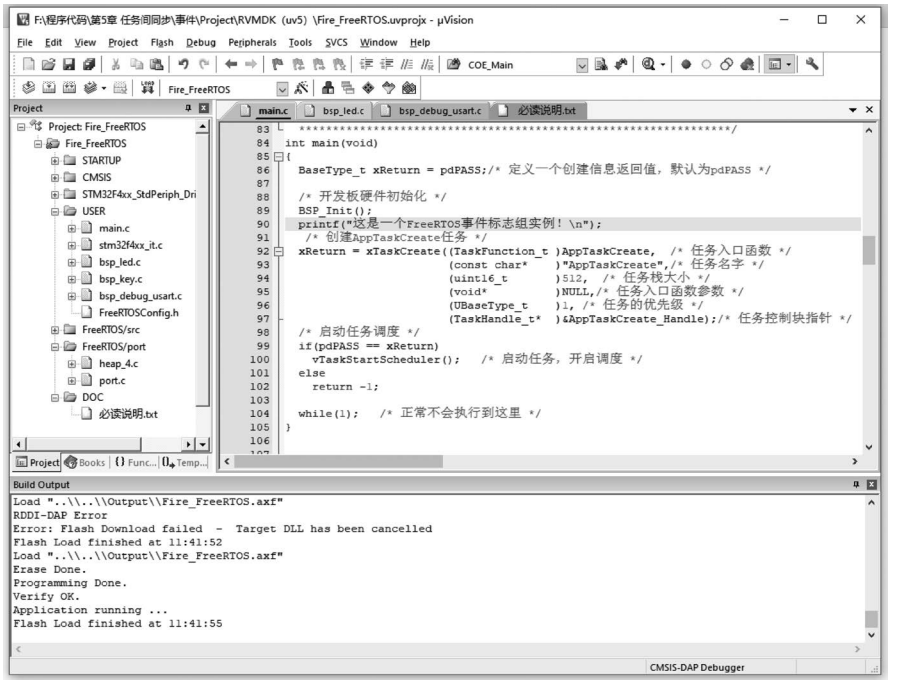

图 5-21　事件程序下载界面

在计算机上打开野火串口调试助手 FireTools，然后复位开发板就可以在调试助手中看到串口的打印信息，它里面输出了信息表明任务正在运行中，按下开发板的 KEY1 按键发送事件 1，按下 KEY2 按键发送事件 2；按下 KEY1 与 KEY2，在串口调试助手中可以看到运行结果，并且当事件 1 与事件 2 都发生时，开发板的 LED 会进行翻转。

事件实例运行结果如图 5-22 所示。

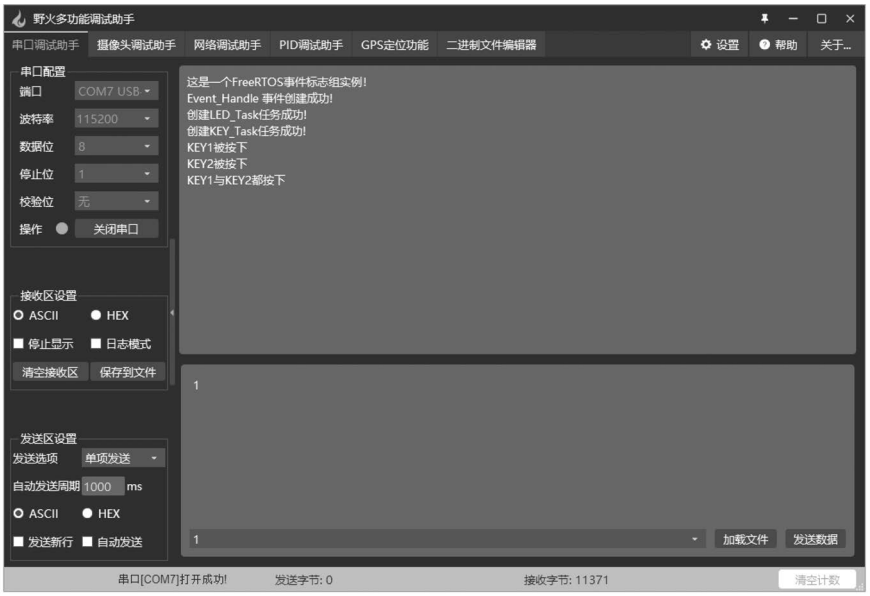

图 5-22　事件实例运行结果