3.信号量

2025年9月25日 20:21

- ## 一、信号量的引入
  - 假设有一个长度为4的停车场，我们用一个量Count来表示此时停车场剩余车位的。在车进入停车场之前要判断一下是否还有位置，如果Count>0，则表示还有停车位，车可以进入，车进入后对Count--，表示此时车位有少了一个，当Count>0不成立时，此时表示停车场已经没有车位了，此时车有两个选择，一是等待别的车开走，然后把车开进停车场。二是直接把车开走。此时Count就是一个**信号量，用来表示此时的环境状态。**
  - 此时会说了，那信号量不就是一个全局变量，用来计数而已，也没什么特别的啊。如果只把信号量当作一个全局变量，那就错了。在FreeRTOS系统中，任务之间会在不经意间别切换或者被别的任务抢占，但是如果信号量只是一个全局变量而已，那就会导致系统中数据的完整性被破坏，比如，此时刚有一辆车判断条件满足，程序执行到Count--时，而此时有一个车也要进入停车场，后来的这辆车抢占了前面那辆车的车位，则此时Count的值就会错乱。如果信号量会有这种原子性操作的漏洞，则就不会出现这个东西了。由此我们就要提到**信号量的另一个特性就是起到保护作用**，当一个任务获取信号量的时候，此时信号量就会将任务保护起来。**此时如果有别的任务也想要获取信号量的话，此时这个任务无论优先级有多高，都无发抢夺当前任务，后来的任务因无法获取到信号量而进入阻塞状态。**这里一定要注意一下，**信号量只是保护别的要获取信号量的任务无法抢占当前获取信号量的任务。即在任务获取信号量的过程中，系统会确保当前任务成功获取到信号量，保证信号量操作的原子性（即操作不会被其他任务或中断打断），当任务成功获取信号量之后，系统还是会进行任务调度**！
- ## 二、信号量的定义
  - 在FreeRTOS系统中，信号量（semaphore）是一种**任务间同步**或**资源共享的机制**，**本质上就是一个计数器**，通过**对计数器的操作（加或减）来实现任务间的协调。（效果上，信号量可以看作是一个Flag标志位，系统通过对标志位来实现某些功能**）
- ## 三、信号量的作用
  - 通过上面对信号量的了解相信你已经知道了信号量的核心作用是什么了
  - 1.资源共享
    - 控制多个任务对有限资源的访问，确保同一时间只有有限任务可以访问资源
  - 2.任务同步
    - 实现任务中的顺序协作
- ## 三、信号量的分类
  - 经过上面的对信号量的了解，我相信已经对信号量有了认识，那么接下来我们来看一下信号量有哪些
  - **1.二值信号量**
    - 三.1.1 定义
      - 首先我们来认识一下什么是二值信号量，通过名字我们不难发现这个信号量应该只有两个值，事实上确实如此，**二值信号量只有两个计数值，分别是0（不可用）和1（可用）两种状态**.
    - 三.1.2 作用及效果
      - **1.实现“事件触发式”任务同步（核心）**
        - 让**“被动等待的任务”**转变成**“主动触发事件”**发生时执行，目的时为了**节省CPU的资源**
        - 简单来说就是由原来的系统每一次任务轮转时，都会由CPU判断任务是否满足条件，变成了，**当有条件满足了，自动由阻塞等待变成了可运行的就绪态。**例如，有两个任务Task1和Task2，Task1开始时获取了信号量，当Task2运行时，它也想要获取信号量，但是信号量被Task1占有无法获取，此时Task2就进入休眠阻塞状态，此时CPU知道Task2因为无法获取信号量而进入阻塞等待，CPU往后就不会在进行判断条件是否满足了，而是当Task1释放信号量之后，系统自动将Task2由阻塞态变成就绪态。
      - **2.保障共享资源的互斥访问（关键）**

- 当多个任务或着中断**需要访问同一个外设（如串口或者传感器等等）或数据时**，通过一个**信号量上锁，确保同一时间只有一个访问者操作，避免了数据的错乱或冲突**
- 比如，Task1要执行C++的代码，而Task2要执行C--，此时如过不对任务做出控制，就会有C的数据错乱的风险，此时，我们让Task1获取二值信号量，只有获得二值信号量的任务才可以访问C资源，避免了多任务同时访问造成的数据错误的风险。

三.1.3 二值信号量的创建

二值信号量的创建有专门的API函数

- SemaphoreHandle_t xSemaphoreCreateBinary（void）
  - 函数**无参数**，但是**有返回值。**创建成功时函数**返回二值信号量句柄**，用于后续的操作管理。否则返回错误。
  - 接下来我们来深入函数创建的底层代码来看一下，FreeRTOS是如何创建一个二值信号量的。

```
#if ( configSUPPORT_DYNAMIC_ALLOCATION == 1 )
    #define xSemaphoreCreateBinary()    xQueueGenericCreate( ( UBaseType_t ) 1, semSEMAPHORE_QUEUE_ITEM_LENGTH,
#endif
```

  - 由此可以看到，**二值信号量的创建函数是一个宏定义的函数，信号量的创建复用了队列的创建函数，参数是固定的。**

```
xQueueGenericCreate( ( UBaseType_t ) 1, semSEMAPHORE_QUEUE_ITEM_LENGTH, queueQUEUE_TYPE_BINARY_SEMAPHORE )
```

```
QueueHandle_t xQueueGenericCreate( const UBaseType_t uxQueueLength,
                                   const UBaseType_t uxItemSize,
                                   const uint8_t ucQueueType )
```

  - 我们来看一下，创建二值信号量给的三个参数
    - 第一个参数是**创建队列的深度**，即队列中可以保存多少个数据，由用户给出，但是这里直接给了1，**表示创建一个深度为1队列**
    - 第二个参数是**队列每个数据空间的大小**，这个直接传入了一个semSEMAPHORE_QUEUE_ITEM_LENGTH通过名字我们可以知道，这是创建信号量时，每个数据的有多大，这里是个宏定义，我们点进去看一下具体的值是多少

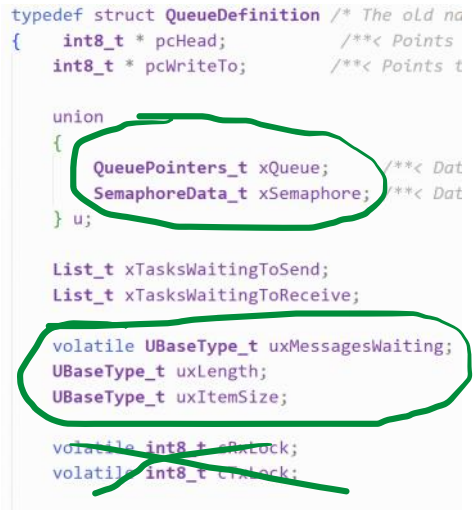
```
typedef QueueHandle_t SemaphoreHandle_t;
```

```
#define semBINARY_SEMAPHORE_QUEUE_LENGTH    ( ( uint8_t ) 1U )
#define semSEMAPHORE_QUEUE_ITEM_LENGTH      ( ( uint8_t ) 0U )
#define semGIVE_BLOCK_TIME                  ( ( TickType_t ) 0U )
```

      - 由图可以知道，**这个参数的值时0**，这就表示此时每个数据最大的空间是0，表示不能保存数据。
    - 第三个参数是**创建队列的种类**，这里传入queueQueue_TYPE_BINARY_SEMAPHORE,翻译过来就是队列种类中的二值信号量，第三个参数可以有以下值

```
/* For internal use only.  These definitions *must* match those in queue.c. */
#define queueQUEUE_TYPE_BASE                ( ( uint8_t ) 0U )
#define queueQUEUE_TYPE_SET                 ( ( uint8_t ) 0U )
#define queueQUEUE_TYPE_MUTEX               ( ( uint8_t ) 1U )
#define queueQUEUE_TYPE_COUNTING_SEMAPHORE  ( ( uint8_t ) 2U )
#define queueQUEUE_TYPE_BINARY_SEMAPHORE    ( ( uint8_t ) 3U )
#define queueQUEUE_TYPE_RECURSIVE_MUTEX     ( ( uint8_t ) 4U )
```

      - 我们可以看到，这里有好多中可以创建的类型，传入第一个表示创建一个队列，传入所选这种，就是创建一个二值信号量。
  - 在上面有幅图片中有一个将队列结构体重定义为**信号量结构体**，接下来我们再次观察队列结构体



- ◇ 在队列结构体中，一位前两参数都是有关数据缓冲区的，因为**信号量的创建时默认无缓冲区，无需接收数据，所以前面两个参数都是默认给NULL**
- ◇ 而下面的**union联合体**部分表示这个结构体**可以根据需要，在队列模式和信号量模式之间切换，共享部分内存资源**
- ◇ 两个List等待链表是二者共用的
- ◇ 而下面的那三个参数，二者之间有很大的差异，对于队列而言分别就是队列的有效数据的数量，队列深度，队列的数据据大小。而**对于信号量而言，第一个参数由原来的有效数据的个数复用为了信号量的计数值，后面两个是固定的1和0**

- ◆ 经过上面对二值信号量创建的深入探索，我们可以知道，**二值信号量的创建就是复用了队列创建的API函数**，只是在传参时，**将队列的uxItemSize设为0，uxLength设为1。意味着创建的队列不存放任务真实数据，仅用其“计数”和“等待链表”的功能。**
- ◆ 从底层代码看，创建二值信号量的本质是：**通过 “复用队列控制块结构→调用队列创建函数→传递信号量特化参数→初始化队列关键字段” 的全流程，将一个 “无数据存储、计数仅 0/1” 的特殊队列，包装成二值信号量。**
- ◆ 所以**信号量本质上就是一个特殊的队列**。

□ 三.1.4 二值信号量的使用

- ◆ **在创建成功二值信号量之后，此时二值信号量是不可以用状态，需要手动释放**（调用xSemaphoreGive（））才可以使用。二值信号量的核心流程如下：
  - ◇ 1.创建二值信号量（初始化）
  - ◇ 2.释放二值信号量（调用xSemaphoreGive（））：**将信号量状态设为“可用（1）”**
  - ◇ 3.对信号量进行操作（获取信号量（xSemaphoreTake（））：**尝试将信号量状态设置为“不可用”，若已被占用则阻塞等待**
- ◆ 接下里我们详细讲述以下二值信号量的使用过程：
- ◆ 创建二值信号量上面有详细过程，这里就不过多叙述
- ◆ 创建成功之后需要先手动释放一次二值信号量，调用xSemaphoreGive（），这里来详细讲述一下此API函数
  - ◇ **xSemaphoreGive（）** 释放信号量函数
  - ◇ `#define xSemaphoreGive( xSemaphore ) xQueueGenericSend( ( QueueHandle_t ) ( xSemaphore ), NULL, semGIVE_BL`
  - ◇ 释放信号量函数也是一个宏定义的函数，真实的实现函数是右边的队列发送的函数，释放信号量API函数要传递一个信号量句柄的参数，接下来我们来看一下真实的实现函数
  - ◇ `xQueueGenericSend( ( QueueHandle_t ) ( xSemaphore ), NULL, semGIVE_BLOCK_TIME, queueSEND_TO_BACK )`
  - ◇

```
BaseType_t xQueueGenericSend( QueueHandle_t xQueue,
                              const void * const pvItemToQueue,
                              TickType_t xTicksToWait,
                              const BaseType_t xCopyPosition )
```

- ◇
```
BaseType_t xQueueGenericSend( QueueHandle_t xQueue,
                              const void * const pvItemToQueue,
                              TickType_t xTicksToWait,
                              const BaseType_t xCopyPosition )
```
  - ◇ 对照函数的原型，我们来看一下释放信号量传递了的参数分别是意思
  - ◇ 第一个参数是**队列句柄**，这里传入的是要**释放的二值信号量的句柄**
  - ◇ 第二个参数是**要发送到队列的消息指针**，这里**传入NULL，因为信号量本身不存放数据**
  - ◇ 第三个参数是**阻塞等待时间**（Free RTOS的时间片单位），这里传入的是**semGIVE_BLOCK_TIE，这个参数宏定义的值是0，说明释放信号量的阻塞的等待时间为0**
  - ◇ 第四个参数是消息存储策略，这个参数我们用不到，这里就不作考虑了
- ◆ 看完了释放信号量复用的函数及其参数，接下来我们深入底层代码去看看是如何操作的
- ◇
```
BaseType_t xQueueGenericSend( QueueHandle_t xQueue,
                              const void * const pvItemToQueue,
                              TickType_t xTicksToWait,
                              const BaseType_t xCopyPosition )
{
    BaseType_t xEntryTimeSet = pdFALSE, xYieldRequired;
    TimeOut_t xTimeOut;
    Queue_t * const pxQueue = xQueue;
    for( ; ; )
    {
        taskENTER_CRITICAL();
        {
            if( ( pxQueue->uxMessagesWaiting < pxQueue->uxLength ) || ( xCopyPosition == queueOVERWRITE ) )
            {
                traceQUEUE_SEND( pxQueue );
            }
                taskEXIT_CRITICAL();
                traceRETURN_xQueueGenericSend( pdPASS );
                return pdPASS;
            }
            else
            {
                if( xTicksToWait == ( TickType_t ) 0 )
                {
                    taskEXIT_CRITICAL();
                    traceQUEUE_SEND_FAILED( pxQueue );
                    traceRETURN_xQueueGenericSend( errQUEUE_FULL );
                    return errQUEUE_FULL;
                }
```
  - ◇ 在将一些系统变量赋了初始值之后，**直接进入临界区（关中断），保护任务此时不会被抢占或打断，保证了操作的原子性。**
  - ◇ 进入临界区后，**检查信号量是否 “可用” （即 uxMessagesWaiting < 1，因为信号量长度为1）。若可用，直接释放信号量并返回成功，对计数值++。**
  - ◇ 如果条件不满足，即信号量不可用，信号量被占用，因为**传参的时候我们传进来的阻塞等待时间为0，所以这里我们就直接返回错误ERR**
- ◆ 讲述完释放信号量了，接下来我们要详细讲述一下获取信号量
  - ◇ **xSemaphoreTake ()** 获取信号量函数
  - ◇
```
#define xSemaphoreTake( xSemaphore, xBlockTime )    xQueueSemaphoreTake( ( xSemaphore ), ( xBlockTime ) )
```
  - ◇ 和前面两个函数一样，获取信号量的函数也是宏定义的函数，真实的实现函数是右边的队列信号量获取函数，注意这个并不是队列的接收函数，我们来看看传入的参数分别是什么
  - ◇ 我们观察可以看到，获取信号量需要给的参数和右边真实实现函数的参数一样
    - 1.**传入一个信号量句柄**，即表示那个信号量要获取
    - 2.**阻塞等待时间**
  - ◇ 接下来我们来深入看一下获取信号量的底层代码是如何实现的

```
BaseType_t xQueueSemaphoreTake( QueueHandle_t xQueue,
                                TickType_t xTicksToWait )
{
```

◇ 接下来我们来深入看一下获取信号量的底层代码是如何实现的

```
BaseType_t xQueueSemaphoreTake( QueueHandle_t xQueue,
                                TickType_t xTicksToWait )
{
    BaseType_t xEntryTimeSet = pdFALSE;
    TimeOut_t xTimeOut;
    Queue_t * const pxQueue = xQueue;
    for( ; ; )
    {
        taskENTER_CRITICAL();
        {
            const UBaseType_t uxSemaphoreCount = pxQueue->uxMessagesWaiting;
            if( uxSemaphoreCount > ( UBaseType_t ) 0 )
            {
                traceQUEUE_RECEIVE( pxQueue );
                pxQueue->uxMessagesWaiting = ( UBaseType_t ) ( uxSemaphoreCount - ( UBaseType_t ) 1 );
                if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToSend ) ) == pdFALSE )
                {
                    if( xTaskRemoveFromEventList( &( pxQueue->xTasksWaitingToSend ) ) != pdFALSE )
                    {
                        queueYIELD_IF_USING_PREEMPTION();
                    }
                    else
                    {
                        mtCOVERAGE_TEST_MARKER();
                    }
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
                taskEXIT_CRITICAL();
                traceRETURN_xQueueSemaphoreTake( pdPASS );
                return pdPASS;
            }
            else
            {
                if( xTicksToWait == ( TickType_t ) 0 )
                {
                    taskEXIT_CRITICAL();
                    traceQUEUE_RECEIVE_FAILED( pxQueue );
                    traceRETURN_xQueueSemaphoreTake( errQUEUE_EMPTY );
                    return errQUEUE_EMPTY;
                }
                else if( xEntryTimeSet == pdFALSE )
                {
                    vTaskInternalSetTimeOutState( &xTimeOut );
                    xEntryTimeSet = pdTRUE;
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
        }
        taskEXIT_CRITICAL();
        vTaskSuspendAll();
        prvLockQueue( pxQueue );
        if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) == pdFALSE )
        {
            if( prvIsQueueEmpty( pxQueue ) != pdFALSE )
            {
                traceBLOCKING_ON_QUEUE_RECEIVE( pxQueue );
                vTaskPlaceOnEventList( &( pxQueue->xTasksWaitingToReceive ), xTicksToWait );
                prvUnlockQueue( pxQueue );
```

- 首先进来还是**进入临界区，关闭中断**。
- 进入临界区后，检查信号量计数（uxMessagesWaiting）是否大于 0（即信号量可用）。
- 若可用，**将信号量计数减 1（表示已获取）**，并检查是否有任务在等待 “释放信号量”，若有则唤醒该任务，最后返回成功（pdPASS）。
- 若**信号量不可用且超时时间为 0，直接退出临界区**，返回 “信号量空” 错误（errQUEUE_EMPTY）。
- 若**给出超时等待时间，任务就是进入等待，从原来的就绪态转移到阻塞态，进行阻塞等待，同时将自己记录到获取信号量等待的列表中。**当条件满足时，会自动转会原来的就绪态。

□ 有关于二值信号量的定义，作用，及操作我们都已经讲完了，下面我们进行下一种信号量的学习。

○ 2.计数信号量

- 上面我们已经学习过了二值信号量了，计数信号量就是相对于二值信号量可以计数的值更过了
- 三.2.1 定义
  - 计数信号量是一种基于一个**整数计数器的同步机制**，专门用于**管理多个相同资源的分配和统计事件发生的次数。相比于二值信号量的“可用（1）”和“不可用（0）”，计数信号量通过计数器的动态变化，支持更复杂的资源调度和事件处理场景。**
- 三.2.2 实现与核心
  - 计数信号量的核心是一个**非负整数计数器**，其行为受两个参数约束：
    - 1.最大计数值（Max Count）:计数器**允许的最大值（创建时指定），同于限制资源的总数**
    - 2.初始计数值（Initial Count）：计数器**初始计数值（创建时指定），用于表示初始时刻可用的资源数量或者处理事件的初始次数**
- 三.2.3 作用及其效果
  - **1.资源池的管理（多实例资源分配）**
    - 当系统中**有多个相同的资源时，利用计数信号量去控制资源的分配，避免冲突**
      - 例如，当系统中由3个SPI的接口，我们只需要创建一个最大计数值为3，初始计数值为3的计数信号量就可以实现对三个资源的控制，每次有一个任务调用SPI时，我们就让计数值-1，当计数值=0时，我们就进行阻塞等待，避免了调用的冲突

2. 资源管理

在这种用法中，信号量的计数值用于表示可用资源的数目。一个任务要获取资源的控制权，其必须先获得信号量——使信号量的计数值减 1。当计数值减至 0，则表示没有可用资源。当任务利用资源完成工作后，将给出(归还)信号量——使信号量的计数值加 1。

用于资源管理的信号量，在创建时其计数值被初始化为可用资源总数。第四章涵盖了使用信号量来管理资源。

  - **2.事件计数（统计事件的发生次数）**
    - 用于**记录某个事件的发生次数，任务通过获取信号量依次处理事件。**
      - 假如我们有一个需要按键按下的次数来执行相应的任务，比如按下一次时打开LED灯，按下两次是调整LED灯的颜色，按下第三次就是LED流水灯等等，我们可以根据按下按键次数的不同来实现相应的任务，用计数信号量来计数就很合适。

1. 事件计数

在这种用法中，每次事件发生时，中断服务例程都会“给出(Give)”信号量——信号量在每次被给出时其计数值加 1。延迟处理任务每处理一个任务都会“获取(Take)”一次信号量——信号量在每次被获取时其计数值减 1。信号量的计数值其实就是已发生事件的数目与已处理事件的数目之间的差值。这种机制可以参考图 31。

用于事件计数的计数信号量，在被创建时其计数值被初始化为 0。

- 三.2.4 计数信号量的创建
  - 创建二值信号量的时候调用的底层位创建队列的函数，现在我们来看一下计数信号量的创建函数
  - 计数信号量创建也有专门的API函数
  - xSemaphoreHandle xSemaphoreCreateCounting(unsigned portBASE_TYPE uxMaxCount, unsigned

portBASE_TYPE uxInitialCount);

- 函数有两个参数，并且会返回**一个信号量句柄**
- 参数1：**uxMaxCount，最大计数值**

uxMaxCount	最大计数值。如果把计数信号量类比于队列的话，uxMaxCount 值就是队列的最大深度。 当此信号量用于对事件计数或锁存事件的话，uxMaxCount 就是可锁存事件的最大数目。 当此信号量用于对一组资源的访问进行管理的话，uxMaxCount 应当设置为所有可用资源的总数。

- 参数2：**uxInitialCount，初始计数值**

uxInitialCount	信号量的初始计数值。 当此信号量用于事件计数的话，uxInitialCount 应当设置为 0——因为当信号量被创建时，还没有事件发生。 当此信号量用于资源管理的话，uxInitialCount 应当等于 uxMaxCount——因为当信号量被创建时，所有的资源都是可用的。

- 看完这个计数信号量创建的API函数，我们深入底层看一下这个计数信号量创建的API函数，当我们点开这个函数我们就会发现，这是一个宏定义的函数

```
#if ( configSUPPORT_DYNAMIC_ALLOCATION == 1 )
    #define xSemaphoreCreateCounting( uxMaxCount, uxInitialCount )    xQueueCreateCountingSemaphore( ( uxMaxCc
#endif
```

- 我们发现，计数信号量底层的创建也是**用队列的底层创建的**，但是和二值信号量的创建函数不同，那么实际创建信号量的函数是右边调用的队列创建计数信号量函数

```
xQueueCreateCountingSemaphore( ( uxMaxCount ), ( uxInitialCount ) )
```

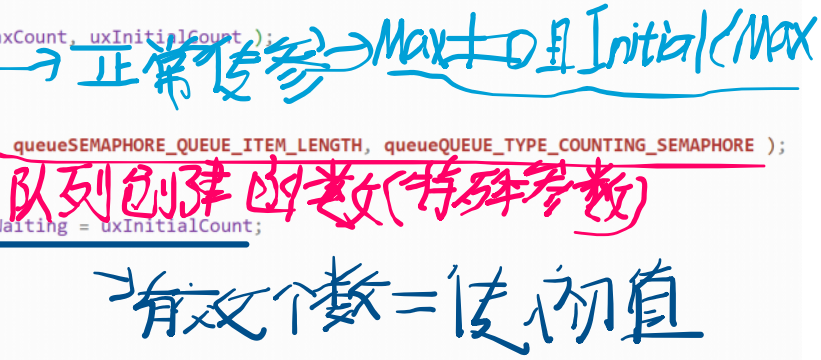
- 我们可以看到，这个函数的两个参数和宏定义的函数参数相同，即就是创建信号量传入的参数，接下来，我们深入底层代码来看一下

```c
#if ( ( configUSE_COUNTING_SEMAPHORES == 1 ) && ( configSUPPORT_DYNAMIC_ALLOCATION == 1 ) )

    QueueHandle_t xQueueCreateCountingSemaphore( const UBaseType_t uxMaxCount,
                                                 const UBaseType_t uxInitialCount )
    {
        QueueHandle_t xHandle = NULL;
        traceENTER_xQueueCreateCountingSemaphore( uxMaxCount, uxInitialCount );
        if( ( uxMaxCount != 0U ) &&
            ( uxInitialCount <= uxMaxCount ) )
        {
            xHandle = xQueueGenericCreate( uxMaxCount, queueSEMAPHORE_QUEUE_ITEM_LENGTH, queueQUEUE_TYPE_COUNTING_SEMAPHORE );
            if( xHandle != NULL )
            {
                ( ( Queue_t * ) xHandle )->uxMessagesWaiting = uxInitialCount;

                traceCREATE_COUNTING_SEMAPHORE();
            }
            else
            {
                traceCREATE_COUNTING_SEMAPHORE_FAILED();
            }
        }
        else
        {
            configASSERT( xHandle );
            mtCOVERAGE_TEST_MARKER();
        }
        traceRETURN_xQueueCreateCountingSemaphore( xHandle );
        return xHandle;
    }

#endif /* ( ( configUSE_COUNTING_SEMAPHORES == 1 ) && ( configSUPPORT_DYNAMIC_ALLOCATION == 1 ) ) */
```



- 首先就是配置**FreeConfig.h函数开启宏**
- 然后就是**初始化句柄，用于创建完成后返回**
- 然后**判断是否满足计数信号量的创建条件**：最大计数值**MaxCount不等于0**，然后**初始计数值InitialCount<最大计数值MaxCount**，满足条件就会**调用一个队列创建函数**，然后**传入特殊的参数用来创建计数信号量**
  - 第一个参数要**传入队列的深度**，这里传入的是**最大计数值MaxCount，就是能够保存的数量**
  - 第二个参数要**传入每个队列空间的大小**，因为是**信号量不用来保存数据所以和创建二值信号量时传入的参数相同，这里传入的是一个宏，其值就是0**
  - 第三个参数要传入的是要**创建的队列的类型**，根据传入不同的值来表示创建不同类型的队列，其值如下

```c
/* For internal use only.  These definitions *must* match those in queue.c. */
#define queueQUEUE_TYPE_BASE                ( ( uint8_t ) 0U )
#define queueQUEUE_TYPE_SET                 ( ( uint8_t ) 0U )
#define queueQUEUE_TYPE_MUTEX               ( ( uint8_t ) 1U )
#define queueQUEUE_TYPE_COUNTING_SEMAPHORE  ( ( uint8_t ) 2U )
#define queueQUEUE_TYPE_BINARY_SEMAPHORE    ( ( uint8_t ) 3U )
#define queueQUEUE_TYPE_RECURSIVE_MUTEX     ( ( uint8_t ) 4U )
```

  - 因为这里创建的是计数信号量，所以这里穿入的是计数信号量类型，前面创建二值信号量对应的就是传入二值信号量类型，传入第一个就表示要创建的是队列
- 当创建成功之后，就是**将uxMessageWaiting复用为此时的有效数据个数，并把传入的初始值赋给它**
- 最后把计数信号量句柄返回即可

- 通过计数信号量创建的底层代码我们可以知道，**计数信号量的本质也是一个队列，是一个特殊的队列**

三.2.5 计数信号量的使用

- 首先要**创建一个计数信号量**，通过调用创建计数信号量的API函数
- 然后就是任务**调用获取信号量函数**，**调用xSemaphoreTake（）函数**，**注意这里与二值信号量不用，因为二值信号量在创建时，uxInitialCount初始计数值给的是0，在获取前需要先手动释放一次才可以**

获取，而计数信号量的初始值在创建时给出，如果不是0就不需要在使用前释放

- 当任务**在获取或者访问完资源之后调用释放信号量函数xSemaphoreGive（）函数**
- 这里的获取和释放信号量调用的函数和上面的二值信号量相同，其底层的代码实现这里就不在详细简述了

四.信号量的总结

- 到这里我们就把信号量的知识全部都讲述完成了，你会发现**其实FreeRTOS中的信号量其本质就是一个特殊的队列**，根据不用信号量的类型在创建时给予不同的参数。而二值信号量又好像一个特殊的计数信号量，他只是将计数的最大值设置为了1。

  信号量是 FreeRTOS 中**基于非负整数计数器的轻量同步机制**，**通过xSemaphoreTake()（计数器减 1）和xSemaphoreGive()（计数器加 1）操作**，**实现多任务间的资源协调与事件处理**；其中二值信号量计数器仅为 0 或 1，**适用于独占式资源管理**，计数信号量计数器可在 0 到最大值间变化，**用于多同类资源分配或事件统计**，**其本质是简化的队列，通过阻塞等待队列管理任务阻塞与唤醒**，**核心价值在于以简洁的计数逻辑平衡多任务竞争与协作的效率和安全性，是同步与资源管理的基础工具。**