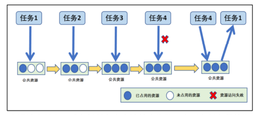
信号量

- 提出的背景
  - 在裸机编程中这样使用过一个变量：用于标**记某个事件是否发生**，或者**标志一下某个东西是否正在被使用**，如果是被占用了的或者没发生，我们就不对它进行操作。
  - 主要用于资源管理/任务同步：某个任务去获取信号量，一旦获取到了就执行操作（一旦释放了信号量，需要同步的任务就获取它并执行，实现了任务“跑进”/同步）
  - 本质上就是一个非负整数，任何任务获取它（占用资源），就会减一（资源减少）——当该整数值为零时，所有试图获取它的任务都将处于阻塞状态
- 主要的类型
  - 二值信号量——用于临界资源访问也可以用于同步功能
    - 同步的机理——信号量在创建后应被置为空，任务 1 获取信号量而进入阻塞，**任务 2 在某种条件发生后，释放信号量**，于是**任务 1 获得信号量得以进入就绪态**，如果任务 1 的优先级是最高的，那么就会立即切换任务，从而达到了两个任务间的同步
      - 可见：信号量的基本操作是**释放和获取——这两个操作基于队列这一数据结构**
      - **<中断也可以释放信号量，因此我们也可以实现中断和任务的同步>**
    - 二值信号量的数据结构
      - **看作只有一个消息的队列**：队列只能为空或满（因此称为二值），我们在运用的时候只需要知道队列中是否有消息即可，而无需关注消息是什么。
  - 计数信号量
    - 计数信号量则可以被认为长度大于 1 的队列
      - 事件计数
        - 运行机制
          每当**某个事件发生时，任务或者中断将释放一个信号量**（信号量计数值加 1），当**事件被处理时**（一般在任务中处理），**处理任务会取走该信号量**（信号量计数值减 1），信号量的**计数值表示还有多少个事件没被处理**
        - 这里的事件是一个抽象的概念，不仅仅是内核的事件/异常，还可以是我们自定义的事件：按键按下/串口得到消息…
      - 资源管理
        - 信号量的**计数值表示系统中可用的资源数目**，任务必须先**获取到信号量才能获取资源访问权**，当信号量的**计数值为零表示系统没有可用的资源**
        - Tips：**使用完资源的时候必须归还信号量**，否则当计数值为 0 的时候任务就无法访问该资源了。
      - 应用——
        1、事件计数
        2、资源管理
  - 互斥信号量——特殊的二值信号量
    - 我们使用信号量的很多时候是为了给临界资源建立一个标志，信号量表示了该临界资源被占用情况（使得临界资源得到有效的保护）
  - 递归信号量
    - 对于已经获取递归互斥量的任务可以重复获取该递归互斥量，该任务拥有递归信号量的所有权
- 信号量的运行机制
  - 二值信号量的运行机制
    - 二值信号量获取——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它任务/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。**

中断/任务释放信号量让二值信号量有效，那么订阅他的任务就能成功获取消息并解除阻塞

  - 计数信号量的运行机制



资源被全部占用后有新任务来申请就会失败，直到有任务释放资源才能成功申请

- 信号量控制块——本质和队列QCB一样

```
typedef struct QueueDefinition
{
    int8_t *pcHead;                 /*< Points to the beginning of the queue storage area.
    int8_t *pcTail;                 /*< Points to the byte at the end of the queue storag
    int8_t *pcWriteTo;              /*< Points to the free next place in the storage area
    union                           /* Use of a union is an exception to the coding stand
    {
        int8_t *pcReadFrom;         /*< Points to the last place that a queued item was
        UBaseType_t uxRecursiveCallCount;/*< Maintains a count of the number of times a r
    } u;

    List_t xTasksWaitingToSend;     /*< List of tasks that are blocked waiting to post on
    List_t xTasksWaitingToReceive;  /*< List of tasks that are blocked waiting to read f

    volatile UBaseType_t uxMessagesWaiting;/*< The number of items currently in the queue.
    UBaseType_t uxLength;           /*< The length of the queue defined as the number of
    UBaseType_t uxItemSize;         /*< The size of each items that the queue will hold.

    volatile int8_t cRxLock;        /*< Stores the number of items received from the queu
    volatile int8_t cTxLock;        /*< Stores the number of items transmitted to the que

    #if( ( configSUPPORT_STATIC_ALLOCATION == 1 ) && ( configSUPPORT_DYNAMIC_ALLOCATION =
        uint8_t ucStaticallyAllocated;  /*< Set to pdTRUE if the memory used by the queue
    #endif

    #if ( configUSE_QUEUE_SETS == 1 )
        struct QueueDefinition *pxQueueSetContainer;
    #endif

    #if ( configUSE_TRACE_FACILITY == 1 )
        UBaseType_t uxQueueNumber;
        uint8_t ucQueueType;
    #endif

} xQUEUE;
```

uxMessagesWaiting=0/1，表示有效信号量个数
uxLength==1，表明长度为1的二值信号量
uxItemSize==0创建二值信号量时，无需申请内存

- 创建信号量
  - 创建完成后的结构

创建完成后的结构

后续我们关心的只是uxMessageWaiting这个变量的值，表示信号量是否有效

  - 本质上调用队列创建的接口

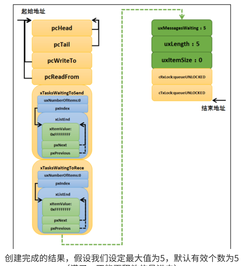
```
#if ( configSUPPORT_DYNAMIC_ALLOCATION == 1 )
    #define xSemaphoreCreateBinary( ) xQueueGenericCreate( ( UBaseType_1 ) 1, semSEMAPHORE_QUEUE_ITEM_LENGTH, queueQUEUE_TYPE_BINARY_SEMAPHORE )
#endif
```

semSEMAPHORE_QUEUE_ITEM_LENGTH：单个消息空间大小=0（不申请内存）
queueQUEUE_TYPE_BINARY_SEMAPHORE：队列类型——作为二值信号量

  - 本质上还是调用队列创建的接口，指定创建计数信号量的队列类型（不分配内存）

```
#if ( configSUPPORT_DYNAMIC_ALLOCATION == 1 )
    QueueHandle_t xQueueCreateCountingSemaphore( const UBaseType_t uxMaxCount, const UBaseType_t uxInitialCount )
{
    QueueHandle_t xReturn;

    configASSERT( uxMaxCount != 0 );
    configASSERT( uxInitialCount <= uxMaxCount );

    xReturn = xQueueGenericCreate( ( uxMaxCount ), ( sizeof( UBaseType_t ) ), queueQUEUE_TYPE_COUNTING_SEMAPHORE );

    if( xReturn != NULL )
    {
        ( ( Queue_t * ) xReturn )->uxMessagesWaiting = uxInitialCount;

        traceCREATE_COUNTING_SEMAPHORE();
    }
    else
    {
        traceCREATE_COUNTING_SEMAPHORE_FAILED();
    }

    return xReturn;
}
```

  - 创建完成的结果，假设我们设定最大值为5，默认有效个数为5（满了，不能再释放信号进去）




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

两个参数：1、**计数信号量的最大值**，当达到这个值的时候，信号量不能再被释放
2、创建计数信号量的初始值。

- 函数接口讲解
  - 信号量的删除

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

本质是一个队列删除的操作

  - 对信号量的两个基本操作——释放/获取
    - **释放——根据前面所说，信号量有效我们才能获取（有效其实就是不为零），让信号量有效的两种方法**
      （1）初始化的时候将其设定为有效
      （2）通过中断/任务释放信号量，让其有效
      - （1）初始化方法，默认的版本将其初始化无效

```
BaseType_1 xQueueGenericReset( QueueHandle_1 xQueue, BaseType_1 xNewQueue )
{
Queue_1 * const pxQueue = ( Queue_1 * ) xQueue;

    configASSERT( pxQueue );

    taskENTER_CRITICAL();
    {
        pxQueue->pcTail = pxQueue->pcHead + ( pxQueue->uxLength * pxQueue->uxItemSize );
        pxQueue->uxMessagesWaiting = ( UBaseType_1 ) 0U;
```

      - （2）释放——本质上为向队列发送消息
        此处为向队尾发送消息，不支持阻塞

```
/*
* 为向队尾发送消息
* \ingroup Semaphores
*/
#define xSemaphoreGive( xSemaphore )    xQueueGenericSend( ( QueueHandle_1 ) ( xSemaphore ), NULL, semGIVE_BLOCK_TIME, queueSEND_TO_BACK )
```

      - 中断版本——调用中断发送接口
        特点：不能释放互斥量（因为你互斥量仅在任务当中有意义）

```
/*
* \ingroup Semaphores
*/
#define xSemaphoreGiveFromISR( xSemaphore, pxHigherPriorityTaskWoken )  xQueueGenericSendFrom( ( QueueHandle_1 ) ( xSemaphore ), ( pxHigherPriorityTaskWoken ) )
```

    - （1）一般的释放操作：如果队列空闲（未满）可释放，进入发送函数

```
BaseType_1 xQueueGenericSend( QueueHandle_1 xQueue, const void * pvItemToQueue, TickType_1 xTicksToWait,
BaseType_1 xCopyPosition )
{
BaseType_1 xEntryTimeSet = ( pdFALSE ), xYieldRequired;
TimeOut_1 xTimeOut;
Queue_1 * const pxQueue = ( Queue_1 * ) xQueue;

    configASSERT( pxQueue );
    configASSERT( ( pvItemToQueue != ( ( void * ) 0 ) ) || ( pxQueue->uxItemSize == ( ( UBaseType_1 ) 0U ) ) );
    configASSERT( ( ( xCopyPosition == queueOVERWRITE ) || ( pxQueue->uxLength != ( ( UBaseType_1 ) 1U ) ) ) );
    #if ( ( INCLUDE_xTaskGetSchedulerState == 1 ) || ( configUSE_TIMERS == 1 ) )
        {
            configASSERT( !( ( xTaskGetSchedulerState() == taskSCHEDULER_SUSPENDED ) && ( xTicksToWait != 0 ) ) );
        }
    #endif

    /* This function relies on the timing assumed wanted to also return
    spurious wakeups from functions (such) this is done in the second
    of execution time efficiency. */
    ( void ) xTemp;
    xTimeOut.xTimeInto = ( ( ( int16_t ) 0 ) );
    ( void ) xTemp;
    /* If the queue is not full then the calling task must be the
    one to place something on the queue unless it is a queue used
    by the ports ISR to communicate with a task.
    If the queue is full then the calling task must block. */
    ( void ) xTemp;
    if( ( pxQueue->uxMessagesWaiting < pxQueue->uxLength ) || ( xCopyPosition == queueOVERWRITE ) )
    {
        traceQUEUE_SEND( pxQueue );
        #if ( configUSE_QUEUE_SETS == 1 )
            {
                const UBaseType_1 uxPreviousMessagesWaiting = pxQueue->uxMessagesWaiting;
```

      - （2）中断版本——本质也是加一的操作（如果未满/否则返回错误）

```
xQueueGenericSendFrom( QueueHandle_1 xQueue, const void * pvItemToQueue,
BaseType_1 * const pxHigherPriorityTaskWoken )
{
BaseType_1 xReturn;
UBaseType_1 uxSavedInterruptStatus;
Queue_1 * const pxQueue = ( Queue_1 * ) xQueue;

    configASSERT( pxQueue );
    configASSERT( ( pvItemToQueue != ( ( void * ) 0 ) ) );
    configASSERT( ( ( pxQueue->uxItemSize == ( ( UBaseType_1 ) 0U ) ) || ( pvItemToQueue != ( ( void * ) 0 ) ) ) );
    configASSERT( ( ( xCopyPosition == queueSEND_TO_BACK ) || ( xCopyPosition == queueSEND_TO_FRONT ) ) );
    #if ( ( configASSERT_DEFINED == 1 ) && ( portASSERT_IF_INTERRUPT_PRIORITY_INVALID() != 0 ) )
        {
            configASSERT( !( ( portCHECK_IF_IN_ISR() == pdTRUE ) && ( ( portMAX_23_BIT_NUMBER_SHIFT == ( ( (
```