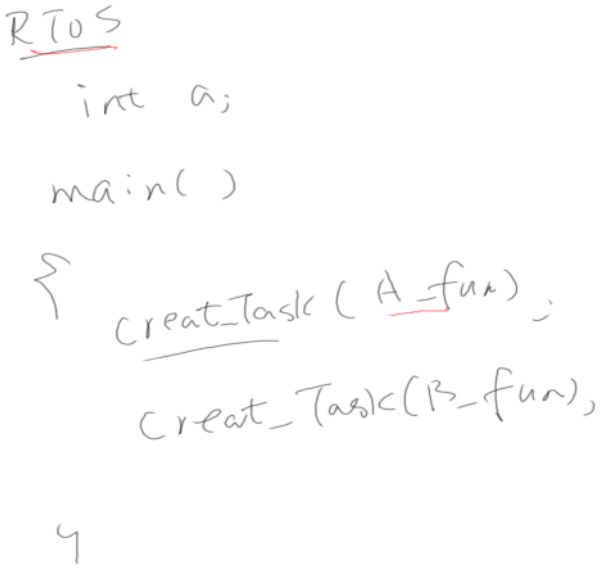
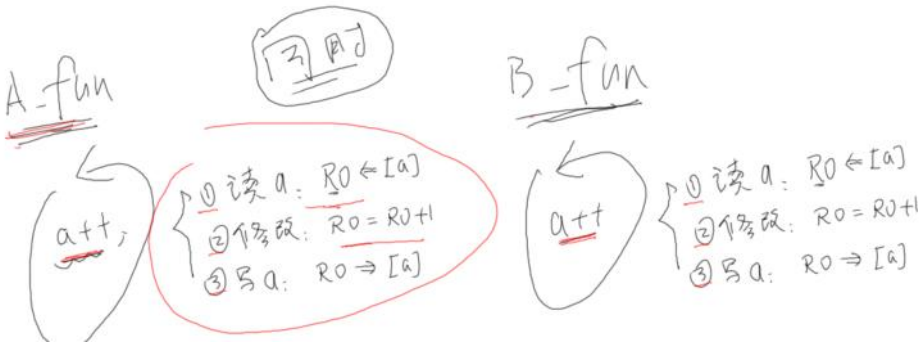
2.队列

2025年9月7日 23:39

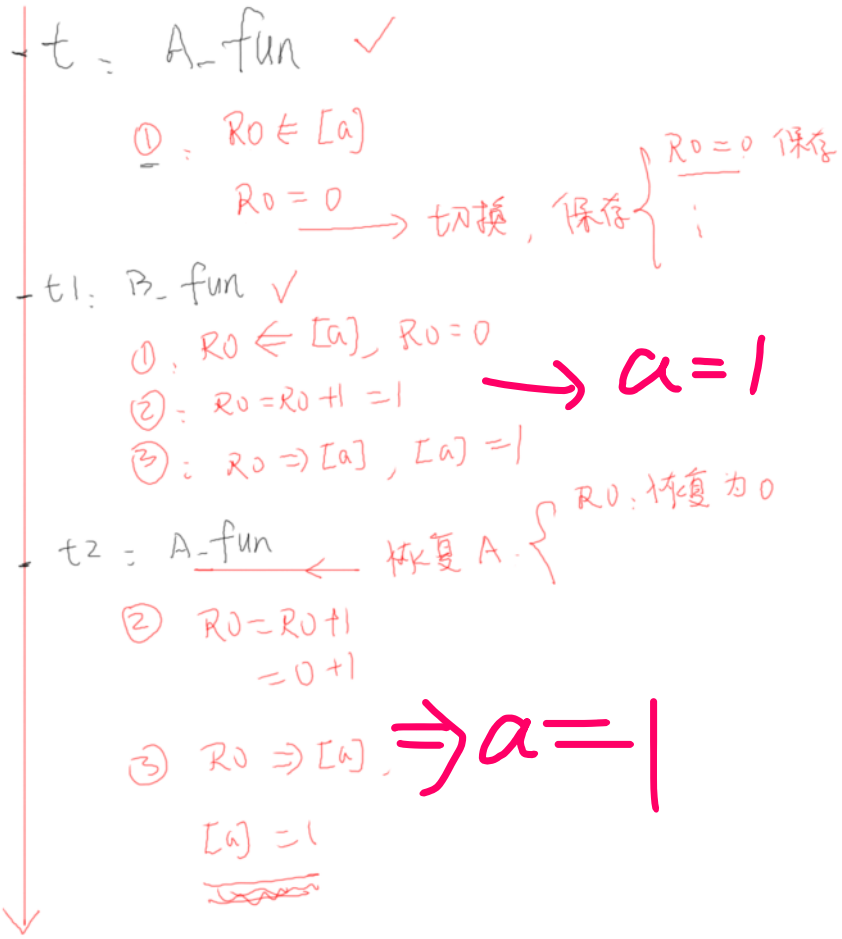
- 一.队列的引入
  - 假设我们有两个任务，分别时**任务A和任务B**，我们**定义一个全局变量a**，任务A和B的**程序均是a++**，如果在FreeRTOS多任务操作系统中，**任务A和B的优先级相同的情况下**，我们来分析一下经过一次系统任务调度之后，**a的值最终是多少**



    - 经过上节课的学习我们可以知道，任务中的一个操作**在ARM架构中需要细分为很多步骤**，就拿**a++**这行代码来说，**首先CPU中的寄存器需要读取a的值，并将其保存到某个寄存器中，接着需要从Flash中读取a++的指令，完成对a变量值的++，最后寄存器需要将a的值重新写入到a的值中。**



    - 就如上图所示，任务A和B都会执行这样一个过程，那么上节课中我们说过，相同任务的优先级是采取时间片轮转的方式，当A任务执行时，假设任务A只执行到① 读取a的值时，此时一个时间片结束，或者此时任务B因为某些事件或着改变任务的优先级过来抢占任务A，此时任务让出CPU的资源，将此时的任务现场（任务上下文，寄存器的值，局部变量等）保存到对应的任务栈中，那么此时全局变量的值还未被修改，被寄存器读取的值为0并保存到寄存器中。任务B抢占任务A之后，任务B开始执行，任务B按照1，2，3的顺序依次执行完之后，此时变量a的值被修改成了1。任务B执行完之后，又轮到任务A执行，此时任务A从任务栈中恢复任务现场，继续执行2和3，由于之前保存到CPU寄存器的值为0，所以此时a的值是保存之前的0而不是1，任务A执行完成之后，此时a的值变为了1。明明任务A和B都执行了，也执行完成了两次a++，可是到最后变量a的值却是1。



    - 所以这种情况下无法保证数据的完整性，我们需要的是在任务A执行时，不要进行任务切换，不要被抢占，即任务A执行时免受外界的打扰。达到一种互斥的效果!
  - 这里我们引入一个新的概念叫做队列。合理的运用队列不仅可以**保护任务在执行的过程中不会被打断**，保证了数据的完整性。而且还可以进行任务间的通信和同步等功能。

二.队列的核心与创建

2.1队列的创建

- 队列的创建需要调用API函数xQueueCreate(uxQueueLength,uxItemSize)来创建一个队列。

```
QueueHandle_t xQueueGenericCreate( const UBaseType_t uxQueueLength,
                                   const UBaseType_t uxItemSize,
                                   const uint8_t ucQueueType )
{
    Queue_t * pxNewQueue = NULL;
    size_t xQueueSizeInBytes;
    uint8_t * pucQueueStorage;

    traceENTER_xQueueGenericCreate( uxQueueLength, uxItemSize, ucQueueType );
```

- xQueueCreate的函数原型就是上面这个函数，我们来分析一下
  - 首先这个函数的类型是QueueHandle类型，类似于任务创建时的TaskHandle类型，是创建队列成功后返回的一个队列句柄，用于后续对队列的操作与管理，如果后续需要操作队列则需要接收用对应类型的变量接收返回值，否则不需要接收。
  - 第一个参数时UBaseType_t类型的变量uxQueueLength：该参数是表示创建队列的深度，即队列**最多可以储存多少个数据项。**

- 第二个参数依然是UBaseType_t类型的uxItemSize：表示队列中每个数据项的大小（以字节为单位）。
- 第三个是uint8_t类型的**ucQueueType**：用于指定队列的类型，不同的类型可能对应不同的队列行为或特性，但是在我们真正创建队列是，**这个参数并不需要给出**，由系统确定。

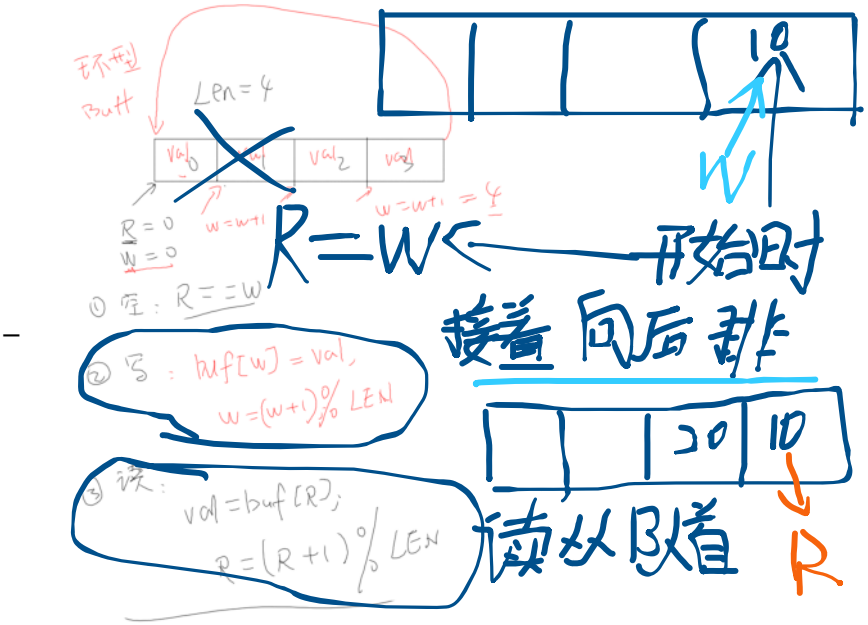
- 经过上面的学习，相信你已经了解了队列的创建方式，那么队列的核心是什么，为什么队列会有这么大的作用呢，接下来我们就来深入了解一下队列的核心。

2.2队列的核心

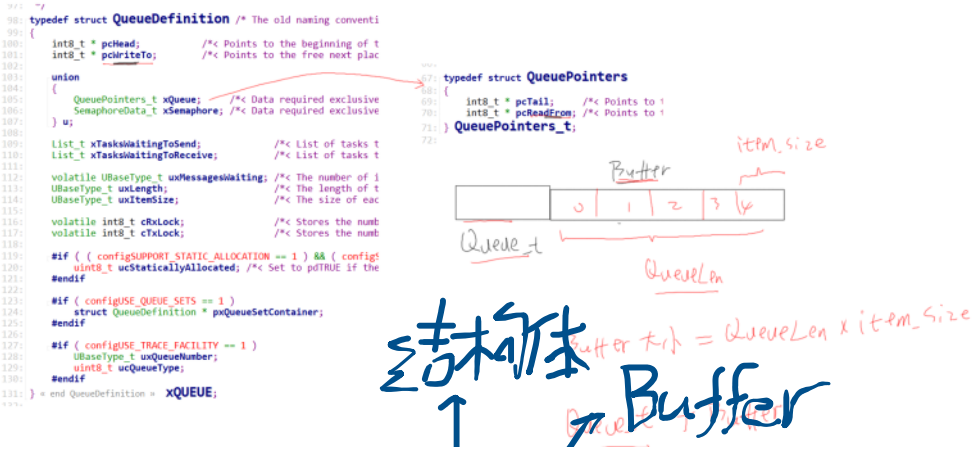
- 队列的核心包括三个部分，分别是关中断，环形缓冲区，链表，接下来我们逐一分析
- **2.2.1 关中断**
  - FreeRTOS为了保证任务并发访问导致的数据不一致，会暂时通过关中断来阻止任务的并发访问，确保在同一时间只有一个任务对数据进行访问，保证了数据的完整性。
  - **那么队列是怎么实现对中断的控制的呢?**
    - 在任务对队列进行读写操作时，在Free RTOS的底层都会调用API函数来开关系统的中断
    - taskENTER_CRITICAL();这是一个宏定义的函数，该函数是用来关闭系统中断的API函数。这个函数的原名叫做进入临界区。就是通过调用这么一个函数，阻止了任务在执行过程中，被别的数据打断或者抢占。
    - 函数的作用
      - 1.关中断（进入临界区，屏蔽当前处理器所有可屏蔽的中断）
      - 2.禁止任务调度器转换
    - taskEXIT_CRITICAL();这个函数和上面那个函数一样，也是一个由宏定义的函数。这个函数和上面那个函数是对应的，这个函数叫退出任务临界区。通过调用这个函数，恢复了任务调度器和系统中断。
    - 函数的作用：
      - 1.递减临界区嵌套计数器，当计数值为为零时，恢复中断使能（退出临界区，恢复进入临界区前的中断屏蔽状态）
      - 2.允许任务调度器重新工作
    - 注意：这两个宏必须严格配对，即两者必须成对使用，即使用了多少次taskENTER_CRITICAL();就要使用多少次taskEXIT_CRITICAL();，否则会导致系统中断被长期屏蔽。每调用一次taskENTER_CRITICAL();，临界区嵌套计数器就会+1，每调用一次taskEXIT_CRITICAL();，临界区嵌套计数器就会-1，**只有当临界区嵌套计数器为0时系统的中断才会正常运行。**
- **2.2.2 环形缓冲区**
  - 关于环形缓冲区他就是一个由用户分配的一个类似于数组的，用于保存数据的一个空间，我们既可以向缓冲区中写数据，也可以从缓冲区中读数据。
  - 一.向缓冲区中**写数据**
    - 环形缓冲区刚创建时里面是空的，此时可以向缓冲区中写入数据，注意开始时是从队尾开始写入数据，此时队首就是队尾（无数据时可以写，如果读的话会进入等待（阻塞态）），每写一次，缓冲区的写位置pcWriteTo就**+1**即向后移动一个，**当写到最后一个数据的是时候，如果此时再向缓冲区中写数据的话，此时pcWriteTo指向的**

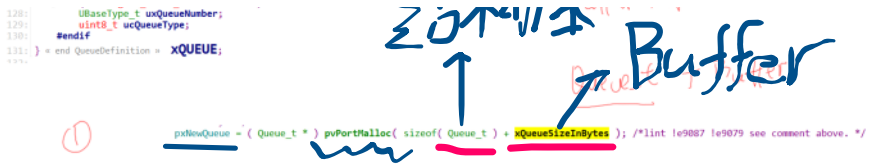
位置就从指向最后一个数据位置变为指向第一个数据位置，此时队列就会进入等待（阻塞态）。所以对于写队列是pcWriteTo指向的位置我们可以总结**W = （W+1）% Length。其中W表示写位置，Length表示队列的深度。**

- 二.从缓冲区中**读数据**
  - 当缓冲区中有数据时，我们可以从缓冲区中读取数据，与写队列类似，都队列也有一个读取位置叫pcReadFrom。开始时，**pcReadFrom为0，指向队首位置**，当有数据别写入队列中时，此时就可以读取写入的数据，每读取一次，pcReadFrom+1，当**读到缓冲区最后一个数据时，此时如果再读取数据，则会和写队列一样，此时pcReadFrom由最后一个位置变为指向缓冲区第一个位置**，此时读取队列也会进入阻塞状态，直到缓冲区中又有数据。同理，我们可以将都队列时的位置总结为：R = （R+1）% Length。R表示读位置，Length表示队列的深度。
- 为了更好的展现环形缓冲区的读写，我们用下图来辅助理解



- 在了解完什么时数据缓冲区之后，接下来我们要结合数据缓冲区来理解队列的核心
- 由队列创建的API函数可以知道，在创建队列的时候我们需要传入两个参数，一个参数是uxQueueLength，表示创建队列的深度，即创建的队列可以保存多少个数据。
- 另外一个参数是uxItemSize，表示每个数据的大小。
- 在创建队列的时候系统不仅会创建一个环形缓冲区，而是会创建一个队列头和一个用户定义的环形缓冲区（大小），环形缓冲区经过上面的学习我们知道了，那么队列头是什么呢。我们深入队列代码的底层看一下。





- 观察了队列底层的代码之后，我们可以发现，队列头是一种类似于任务结构体的数据，我们就可以把它叫做队列结构体，接下来我们来深入分析一下队列结构体中的内容，观察一下队列结构体和环形缓冲区结合是怎么操控队列的
- 三.**队列结构体**
  - 首先在结构体的最上面我们可以看到，队列结构体在**一开始就定义了环形缓冲区的写入位置pcWriteTo和读取位置pcReadFrom**，说明了队列结构体和队列的环形缓冲区有着直接的关系，事实确实如此，**在队列结构体之后紧接着的就是队列环形缓冲区，即在创建队列的时候不仅仅只是创建队列结构体或者是环形缓冲区，而是在创建时，二者共同构成了队列，在创建时分配内存时，pcNewQueue=Queue_t + Buffer。**
  - 接着向下看，我们可以看到**两个List_t类型的链表**，分别是**xTasksWaitingToSend（任务等待发送链表）和xTasksWaitingToReceive（任务等待接收链表）**。那么这两个链表有什么用呢，前面我们说过在读取队列和写入队列时，当缓冲区满时（写队列进入等待）或者缓冲区空时（读队列进入等待），**任务会被记录在这个两个链表中的对应链表**，但注意只是把他们任务记录在链表中，**任务并不在链表中**，因为任务处于等待状态，所以任务会从就绪态任务列表中移到阻塞态任务列表中，所以**任务实际被存储到阻塞链表中**，并不在队列链表中。
  - 对于剩余参数我们来看一下
  - 在队列结构体的开始定义了一个**pcHead**变量，此变量指向**队列存储区域的起始地址**，队列的所有数据都**存放在以该指针为起点的连续内存中。**
  - 在队列链表下面定义了一个volatile UBaseType_t uxMessagesWaiting：该变量表示**待接收消息的数量。**
  - 紧接着是队列的两个参数uxQueueLength即队列深度，uxQueueItem即队列数据大小。
  - 下面的时队列锁和条件编译部分（根据配置项启用不同功能），这里就不再详述。

2.2.3 队列的链表

- 上面在队列结构体中，我们发现了两个队列等待链表，在上面我们已经详述过了，**这两个链表使用与记录那个任务处于等待状态**，以便于当任务满足条件时，**系统从队列等待链表中找到任务，将任务从链表中移除**，与此同时，**任务从阻塞态任务列表中被恢复到之前就绪态任务链表中。**

将核心结合起来看队列（综合）

- 假设有一个队列，有两个任务A（高优先级）和任务B（低优先级）写队列，有两个任务C（高优先级）和任务D（低优先级）去读队列，起始队列中并没有数据。
- 一.**任务A和任务B同时向队列中写数据**
  - 因为A的优先级高，所以开始时，任务A向队列中写入数据，此时任务A的运行过程为
    - 1.关闭FreeRTOS系统的屏蔽中断，调用API函数让任务进入临界区
    - 2.任务A判断队列的数据是否已满，缓冲区中是否还有空间
      - 2.1当缓冲区中还**有空间时**，任务A直接向队列中写入数据，数据被保存到环

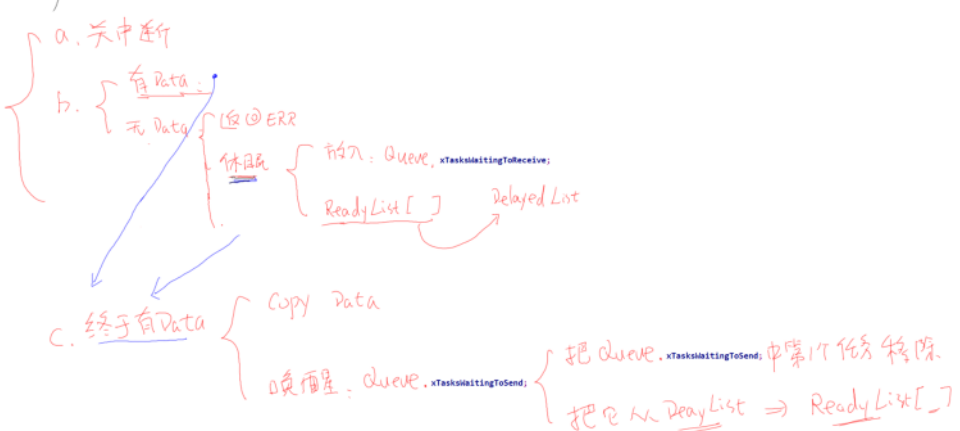
形缓冲区中。

- 2.2当缓冲区中**没有空间时**，任务A想要向队列中写入数据，发现数据已满，此时任务会有以下几个选择
  - 一.当任务不想等待时，此时直接返回ERR
  - 二.任务会进入休眠，此时会进两步操作
    - 1.任务A会被记录到队列写等待链表（**xTasksWaitingToSend）**中
    - 2.任务A会从原来的就绪态列表被转移到阻塞态任务列表
- 2.3任务A阻塞等待着，**当缓冲区有空间了**，此时任务A会进行如下操作
  - 首先，任务A被唤醒
    - 任务A会从**队列写等待链表（xTasksWaitingToSend）中移除**
    - 任务A会由**阻塞态任务列表恢复到原来的就绪态任务列表**
  - 然后，任务A会向缓冲区中写数据（注意：因为任务A向缓冲区中写队列的时候数据已满，所以此时对应的写位置pcWriteTo指向的是第一个数据位）

二.任务C和任务D同时从队列中读取

- 因为C的优先级高，所以开始时，任务C先队列中读取数据，此时任务C的运行过程为
  - 1.关闭FreeRTOS系统的屏蔽中断，调用API函数让任务进入临界区
  - 2.任务A判断队列的数据是否为空，缓冲区中是否有数据
    - 2.1当缓冲区中有数据时，此时任务C**直接从队列中读取数据（Copy）**（注意：任务从队列缓冲区中读取数据是将数据复制，保存到自己的缓冲区中，然后队列会将数据移除，使其不在队列当中了，看似是直接拿走的过程，但其实本质是值的传递，而不是指针传递）
    - 2.2当缓冲区中没有数据时，此时任务C再向队列中读取数据，此时和写队列一样会有两个选择
      - 一.任务不等待，直接返回ERR
      - 二.任务等待，此时任务会进行以下操作
        - 1.任务会被记录到接收数据等待链表（**xTasksWaitingToReceive**）中
        - 2.任务会从原来的**就绪态任务链表被移动到阻塞态任务链表中**

- 2.3当任务的条件满足时，即队列中有数据了，此时任务会进行如下操作
  - 首先任务C被唤醒
    - 任务C会从队列读等待链表（**xTasksWaitingToReceive**）中移除
    - 任务C会从**阻塞态任务列表回到原来的就绪态任务列表**
    - 注意，和写队列一样，当读取数据到最后一位时，此时读位置pcReadFrom会来到第一位



    - **三.如果等待任务超时了**
      - 此时任务会**直接从阻塞态任务列表被移动到原来的就绪态任务列表**
  - 从上面几个方面我们已经了解了队列的创建，已经队列在底层是怎么操作的，那么学习完了队列的内部机制之后，接下来我们要学习如何使用队列。

三.队列的特性与使用

3.1.队列的特性

1.**数据的存储**

往队列中写入数据是**通过字节拷贝把数据复制存储到队列中**，从**队列读取数据是得把队列中的数据拷贝删除。**

2.**可被多任务存取**

- 队列是具有对立权限的内核对象，并不属于或赋予任何任务，所有任务都可以向同一队列写入或读出。

3.**读队列是阻塞**

- 当某个任务试图读一个队列时，其可以指定一个阻塞超时时间。在这段时间中，如果队列为空，该任务将保持阻塞状态以等待队列数据有效。当其它任务或中断服务例程往其等待的队列中写入了数据，该任务将自动由阻塞态转移为就绪态。当等待的时间超过了
- 由于队列可以被多个任务读取，所以对于单个队列而言，也**可能有多个任务处于阻塞状态以等待队列数据有效。**这种情况下，一旦数据有效，只能有一个任务解除阻塞，这个任务就是所有等待任务中优先级最高的。而如果优先级相同，那么则是等待最久的任务。

4.**写队列时阻塞**

- 同读队列一样，任务也可以在写队列时指定一个阻塞超时时间。这个时间是当被写队列已满时，任务进入阻塞态以等待队列空间有效的最长时间。
- 由于队列也可以被多个任务写入，所以对于单个队列而言，也**可能有多个任务处于阻塞**

状态以等待有效的空间。这种情况下，一旦空间有效，只能由一个任务解除阻塞，最高任务就是所有等待任务中优先级最高的。而如果优先级相同，则时等待最久的任务。

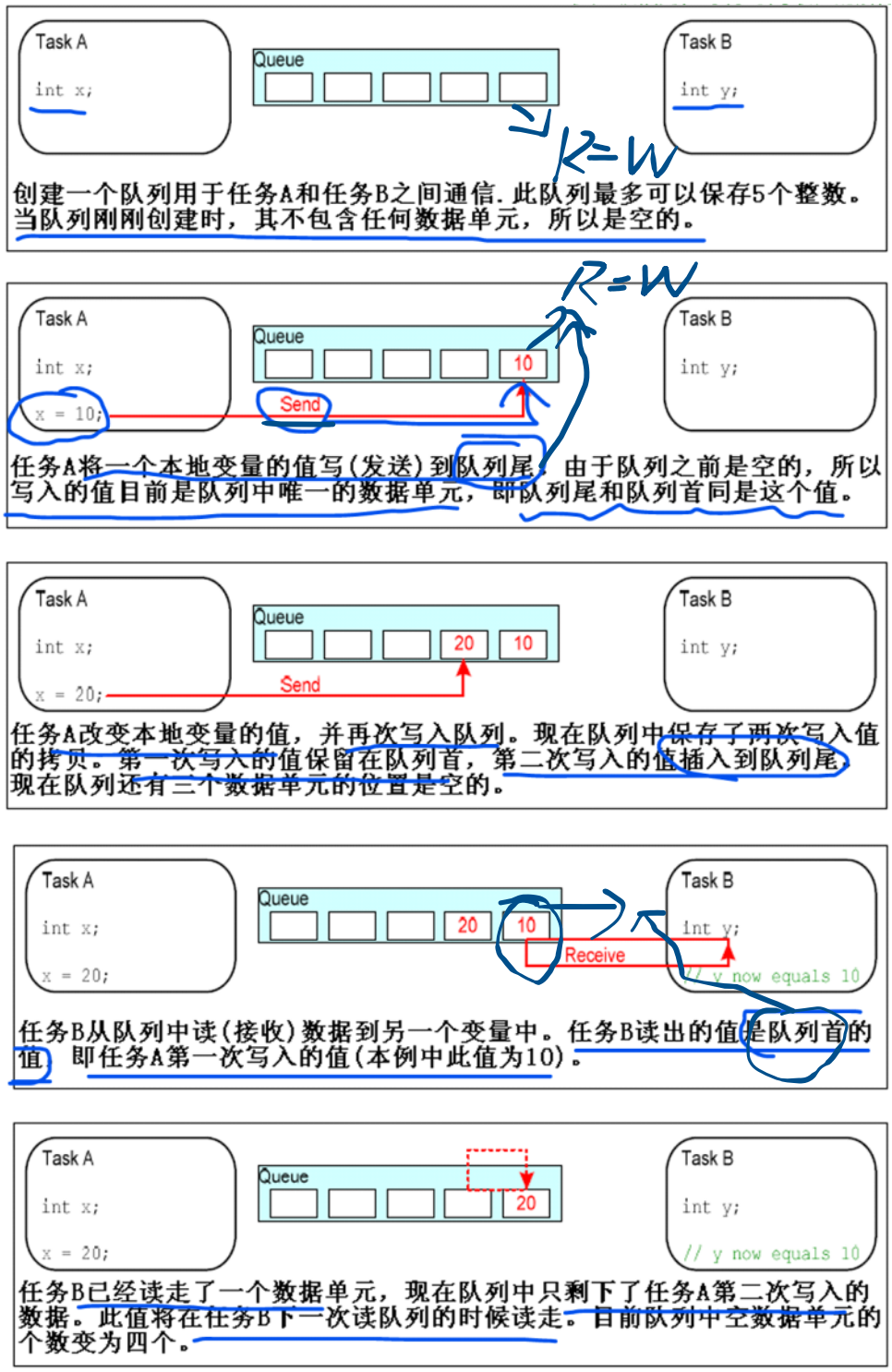


图 19 队列读写过程示例

- 以上就是队列读和写的全部过程

3.2队列的使用

- 1.创建队列，调用API函数 xQueueCreate(uxQueueLength,uxItemSize)
  - xQueueCreate(uxQueueLength,uxItemSize)API函数
    - 队列在使用前必须创建
    - xQueueCreate()用于创建一个队列，并返回一个xQueueHandle类型的队列句柄，用于对其创建的队列进行引用
    - 在创建队列是，FreeRTOS会从堆空间中分配内存空间。分配的内存空间**用于存储队**

列数据结构本身（队列结构体）以及队列中包含的数据单元。

表 7　xQueueCreate()参数与返回值

参数名	描述
uxQueueLength	队列能够存储的最大单元数目，即队列深度。
uxItemSize	队列中数据单元的长度，以字节为单位。
返回值	NULL 表示没有足够的堆空间分配给队列而导致创建失败。 非 NULL 值表示队列创建成功。此返回值应当保存下来，以作为操作此队列的句柄。

- 2.向队列发送数据，调用API函数xQueueSend()和xQueueSendToBack() 与以及 xQueueSendToFront() API 函数
  - **xQueueSend()完全等于xQueueSendToBack()** 。如同函数名一样，这两个函数都是将数据发送到队列的尾部。而**xQueueSendToFront()** 是将数据发送到队列首
  - 注意上面这几个发送函数不能在中断服务程序中调用，如果需要在中断服务程序中调用队列发送函数则调用安全版本的 xQueueSendToFrontFromISR()与 xQueueSendToBackFromISR()，这是专门在中断服务程序中调用的API函数

表 8 xQueueSendToFront()与 xQueueSendToBack()函数参数及返回值

参数名	描述
xQueue	目标队列的句柄。这个句柄即是调用 xQueueCreate()创建该队列时的返回值。
pvItemToQueue	发送数据的指针。其指向将要复制到目标队列中的数据单元。 由于在创建队列时设置了队列中数据单元的长度，所以会从该指针指向的空间复制对应长度的数据到队列的存储区域。
xTicksToWait	阻塞超时时间。如果在发送时队列已满，这个时间即是任务处于阻塞态等待队列空间有效的最长等待时间。

	如果 xTicksToWait 设为 0，并且队列已满，则 xQueueSendToFront()与 xQueueSendToBack()均会立即返回。 阻塞时间是以系统心跳周期为单位的，所以绝对时间取决于系统心跳频率。常量 portTICK_RATE_MS 可以用来把心跳时间单位转换为毫秒时间单位。 如果把 xTicksToWait 设置为 portMAX_DELAY，并且在 FreeRTOSConig.h 中设定 INCLUDE_vTaskSuspend 为 1，那么阻塞等待将没有超时限制。
返回值	有两个可能的返回值: 1. pdPASS 返回 pdPASS 只会有一种情况，那就是数据被成功发送到队列中。 如果设定了阻塞超时时间(xTicksToWait 非 0)，在函数返回之前任务将被转移到阻塞态以等待队列空间有效—在超时到来前能够将数据成功写入到队列，函数则会返回 pdPASS。 2. errQUEUE_FULL 如果由于队列已满而无法将数据写入，则将返回 errQUEUE_FULL。 如果设定了阻塞超时时间（xTicksToWait 非 0），在函数返回之前任务将被转移到阻塞态以等待队列空间有效。但直到超时也没有其它任务或是中断服务例程读取队列而腾出空间，函数则会返回 errQUEUE_FULL。

- xQueueSend()和发送到队列尾部的函数一样，这两者调用哪一个函数都可以。
- 注意上面函数的第二个参数pvItemToQueue，它是发送数据的指针，记得在传参数的时候需要**传入数据的地址**。

- 3.从队列中读取数据，调用xQueueReceive()与 xQueuePeek() API 函数
  - 上面两个读队列函数，我们常用第一种**xQueueReceive（）**，它用于从队列中接收（读取数据单元。接收到的单元同时会从队列中删除。（效果上类似于电脑的剪切，**但是注意，是对数据拷贝完之后再删除的，而不是直接从队列中拿过来的**）
  - **xQueuePeek()**也是从从队列中接收数据单元，不同的是并不从队列中删出接收到的单元。（**效果上类似于电脑中的复制操作**）xQueuePeek()从队列首接收到数据后，不会修改队列中的数据，也不会改变数据在队列中的存储序顺。
  - 和上面向队列中写数据一样，不可以再中断服务程序中调用这两个API函数，需要调用加FromISR后缀的函数

表 9　xQueueReceive()与 xQueuePeek()函数参数与返回值

参数名	描述
xQueue	被读队列的句柄。这个句柄即是调用 xQueueCreate()创建该队列时的返回值。
pvBuffer	接收缓存指针。其指向一段内存区域，用于接收从队列中拷贝来的数据。 数据单元的长度在创建队列时就已经被设定，所以该指针指向的内存区域大小应当足够保存一个数据单元。
xTicksToWait	阻塞超时时间。如果在接收时队列为空，则这个时间是任务处于阻塞状态以等待队列数据有效的最长等待时间。 如果 xTicksToWait 设为 0，并且队列为空，则 xQueueRecieve()与 xQueuePeek()均会立即返回。 阻塞时间是以系统心跳周期为单位的，所以绝对时间取决于系统心跳频率。常量 portTICK_RATE_MS 可以用来把心跳时间单位转换为毫秒时间单位。 如果把 xTicksToWait 设置为 portMAX_DELAY，并且在 FreeRTOSConig.h 中设定 INCLUDE_vTaskSuspend 为 1，那么阻塞等待将没有超时限制。
返回值	有两个可能的返回值: 1. pdPASS 只有一种情况会返回 pdPASS，那就是成功地从队列中读到数据。 如果设定了阻塞超时时间(xTicksToWait 非 0)，在函数返回之前任务将被转移到阻塞态以等待队列数据有效—在超时到来前能够从队列中成功读取数据，函数则会返回 pdPASS。 2. errQUEUE_FULL 如果在读取时由于队列已空而没有读到任何数据，则将返回 errQUEUE_FULL。 如果设定了阻塞超时时间（xTicksToWait 非 0），在函数返回之前任务将被转移到阻塞态以等待队列数据有效。但直到超时也没有其它任务或是中断服务例程往队列中写入数据，函数则会返回 errQUEUE_FULL。

- 这里我们也需要注意第二个参数**pvBuffer**，它是接收数据的缓存指针，在传参的时候也需要**传入接收变量的地址**

- 要**查询队列中环形缓冲区中有效单元的个数**，我们需要调用uxQueueMessagesWaiting() API函数

```
unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle xQueue );
```

程序清单 33 uxQueueMessagesWaiting() API 函数原型

表 10 uxQueueMessagesWaiting()函数参数及返回值

参数名	描述
xQueue	被查询队列的句柄。这个句柄即是调用 xQueueCreate()创建该队列时的返回值。
返回值	当前队列中保存的数据单元个数。返回 0 表明队列为空。

  - 只需要**传入队列句柄，就可以查询到队列中数据的有效个数**
- 到这里队列常用的API函数我们就都了解了，我们上面都是队列用于接收小数据单元，如果数据工作于大数据单元会怎样呢

3.3队列工作于大数据单元

- 如果队列存储的数据单元尺寸较大，那**最好是利用队列来传递数据的指针**，而不是对数据本身在队列上一个个拷贝进或者拷贝出。传递指针无论是在**处理速度上还是在内存空间利用上都更有效。**但是利用队列传递指针需要注意以下几点
  - 1.指针指向的内存空间所有权必须明确
    - 原则上，共享内存在其指针发送到队列之前，只允许发送任务访问。共享内存指针从队列中被读出后，只允许接收任务访问
  - 2.指针指向的内存空间必须有效
    - 如果指针指向的内存空间是动态分配的，只应该有一个任务负责对其内存进行释放。当这段内存被释放之后，就不应该有任何一个任务再访问这段空间。切忌用指针访问任务栈上分配的空间。因为当栈帧改变，栈上的数据将不在有效。