<https://blog.csdn.net/zhzht19861011/article/list/2>

1、xTaskCreate()函数的堆栈大小参数一样，堆栈大小不是以字节为单位而是以字为单位的，

2、设置宏configUSE\_TASK\_NOTIFICATIONS为1（或不定义宏configUSE\_TASK\_NOTIFICATIONS）将会开启任务通知功能，开启后，每个任务多增加8字节RAM。每个RTOS任务具有一个32位的通知值，RTOS任务通知相当于直接向任务发送一个事件，接收到通知的任务可以解除任务的阻塞状态。

3、互斥型信号量必须是同一个任务申请，同一个任务释放，其他任务释放无效。

* 二进制信号量，一个任务申请成功后，可以由另一个任务释放。
* 互斥型信号量是二进制信号量的子集

4、   默认情况下（宏configUSE\_TIME\_SLICING未定义或者宏configUSE\_TIME\_SLICING设置为1），FreeRTOS使用基于时间片的优先级抢占式调度器。这意味着RTOS调度器总是运行处于最高优先级的就绪任务，在每个RTOS 系统节拍中断时在相同优先级的多个任务间进行任务切换。

5、configKERNEL\_INTERRUPT\_PRIORITY用来设置RTOS内核自己的中断优先级。调用API函数的中断必须运行在这个优先级；不调用API函数的中断，可以运行在更高的优先级，所以这些中断不会被因RTOS内核活动而延时。 因为RTOS内核中断不允许抢占用户使用的中断，因此这个宏一般定义为硬件最低优先级。。configMAX\_SYSCALL\_INTERRUPT\_PRIORITY用来设置可以在中断服务程序中安全调用FreeRTOS API函数的最高中断优先级。优先级小于等于这个宏所代表的优先级时，程序可以在中断服务程序中安全的调用FreeRTOS API函数；如果优先级大于这个宏所代表的优先级，表示FreeRTOS无法禁止这个中断，在这个中断服务程序中绝不可以调用任何API函数。

       不调用API的中断可以运行在比configMAX\_SYSCALL\_INTERRUPT\_PRIORITY高的优先级，这些级别的中断不会被FreeRTOS禁止，因此不会因为执行RTOS内核而被延时。

6、  Cortex-M构架自身最多允许256级可编程优先级（优先级配置寄存器最多8位，所以优先级范围从0x00~0xFF）；Cortex-M内核的中断优先级寄存器是以最高位（MSB）对齐的。比如，如果使用了3位来表达优先级，则这3个位位于中断优先级寄存器的bit5、bit6、bit7位。剩余的bit0~bit4可以设置成任何值，但为了兼容，最好将他们设置成1.

对于Cortex-M硬件，RTOS使用到硬件的PendSV和SysTick硬件中断，在函数xPortStartScheduler()中（该函数在port.c中，由启动调度器函数vTaskStartScheduler()调用），将PendSV和SysTick硬件中断优先级寄存器设置为宏configKERNEL\_INTERRUPT\_PRIORITY指定的值。

 RTOS内核使用Cortex-M内核的BASEPRI寄存器来实现临界区（注：BASEPRI为优先级屏蔽寄存器，优先级数值大于或等于该寄存器的中断都会被屏蔽，优先级数值越大，逻辑优先级越低，但是**为零时不屏蔽任何中断**）。那些需要在中断调用时保护的API函数，FreeRTOS使用寄存器BASEPRI实现中断保护临界区。当进入临界区时，将寄存器BASEPRI的值设置成configMAX\_SYSCALL\_INTERRUPT\_PRIORITY，当退出临界区时，将寄存器BASEPRI的值设置成0。

# 二、内存管理：

1.heap\_1.c

      这是所有实现中最简单的一个。一旦分配内存之后，它甚至不允许释放分配的内存。API函数xPortGetFreeHeapSize()返回未分配的堆栈空间总大小。

heap\_1功能简介：

用于从不会删除任务、队列、信号量、互斥量等的应用程序（实际上大多数使用FreeRTOS的应用程序都符合这个条件）；执行时间是确定的并且不会产生内存碎片；实现和分配过程非常简单，需要的内存是从一个静态数组中分配的，意味着这种内存分配通常只是适用于那些不进行动态内存分配的应用。

****2.heap\_2.c****

      和方案1不同，这个方案使用一个最佳匹配算法，它允许释放之前分配的内存块。它不会把相邻的空闲块合成一个更大的块（换句话说，这会造成内存碎片）。

      heap\_2功能简介：

* 可以用于重复的分配和删除具有相同堆栈空间的任务、队列、信号量、互斥量等等，并且不考虑内存碎片的应用程序。
* 不能用在分配和释放随机字节堆栈空间的应用程序

如果一个应用程序动态的创建和删除任务，并且分配给任务的堆栈空间总是同样大小，那么大多数情况下heap\_2.c是可以使用的。但是，如果分配给任务的堆栈不总是相等，那么释放的有效内存可能碎片化，形成很多小的内存块。最后会因为没有足够大的连续堆栈空间而造成内存分配失败。在这种情况下，heap\_4.c是一个很好的选择。

如果一个应用程序动态的创建和删除队列，并且在每种情况下队列存储区域（队列存储区域指队列项数目乘以每个队列长度）都是同样的，那么大多数情况下heap\_2.c可以使用。但是，如果队列存储区在每种情况下并不总是相等，那么释放的有效内存可能碎片化，形成很多小的内存块。最后会因为没有足够大的连续堆栈空间而造成内存分配失败。在这种情况下，heap\_4.c是一个很好的选择。

* 应用程序直接调用pvPortMalloc() 和 vPortFree()函数，而不仅是通过FreeRTOS API间接调用。
* 如果你的应用程序中的队列、任务、信号量、互斥量等等处在一个不可预料的顺序，则可能会导致内存碎片问题，虽然这是小概率事件，但必须牢记。
* 不具有确定性，但是它比标准库中的malloc函数具有高得多的效率。

它的缺点也非常明显：由于在释放内存时不会将相邻的内存块合并，所以这可能造成内存碎片。这就对其应用的场合要求极其苛刻：第一，每次创建或释放的任务、信号量、队列等必须大小相同，如果分配或释放的内存是随机的，绝对不可以用这种内存管理策略；第二，如果申请和释放的顺序不可预料，也很危险。举个例子，对于一个已经初始化的10KB内存堆，先申请48字节内存，然后释放；再接着申请32字节内存，那么一个本来48字节的大块就会被分为32字节和16字节的小块，如果这种情况经常发生，就会导致每个空闲块都可能很小，最终在申请一个大块时就会因为没有合适的空闲块而申请失败（并不是因为总的空闲内存不足）！

****3.heap\_3.c****

     第三种内存管理策略与前两种不同，它不再需要通过数组定义内存堆，而是需要使用编译器设置内存堆空间，一般在启动代码中设置。

heap\_3.c简单的包装了标准库中的malloc()和free()函数，包装后的malloc()和free()函数具备线程保护。

****4.heap\_4.c****

      这个方案使用一个最佳匹配算法，但不像方案2那样。它会将相邻的空闲内存块合并成一个更大的块（包含一个合并算法）。

但第四种内存管理策略的链表尾保存在内存堆空间最后位置，并使用BlockLink\_t指针类型局部静态变量pxEnd指向这个区域（第二种内存管理策略使用静态变量xEnd表示链表尾）。

第四种内存管理策略和第二种内存管理策略还有一个很大的不同是：第四种内存管理策略的空闲块链表不是以内存块大小为存储顺序，而是以内存块起始地址大小为存储顺序，地址小的在前，地址大的在后。这也是为了适应合并算法而作的改变。

 heap\_4.c功能简介：

* 可用于重复分配、删除任务、队列、信号量、互斥量等等的应用程序

heap\_4.c还特别适用于移植层代码，可以直接使用pvPortMalloc()和 vPortFree()函数来分配和释放内存

5.heap\_5.c（V8.1.0新增）

      这个方案同样实现了heap\_4.c中的合并算法，并且允许堆栈跨越多个非连续的内存区。

第一、第二和第四种内存管理策略都是利用一个大数组作为内存堆使用，并且只需要应用程序指定数组的大小（通过宏configTOTAL\_HEAP\_SIZE定义），数组定义由内存管理策略实现。第五种内存管理策略有些不同，首先它允许跨内存区定义多个内存堆，比如在片内RAM中定义一个内存堆，还可以在片外RAM再定义内存堆；其次，用户需要指定每个内存堆区域的起始地址和内存堆大小、将它们放在一个HeapRegion\_t结构体类型数组中，并需要在使用任何内存分配和释放操作前调用vPortDefineHeapRegions()函数初始化这些内存堆。

# 三、任务

创建新的任务并将它加入到任务就绪列表。我们知道子函数调用通过寄存器R0~R3传递参数，在文章的最开始讲xTaskCreate()函数时，提到这个函数有一个空指针类型的参数pvParameters，当任务创建时，它作为一个参数传递给任务，所以这个参数被保存到R0中，用来向任务传递参数。在tasks.c中只定义了这一个全局变量。这个变量用来指向当前正在运行的任务TCB。任务切换时会知道，任务切换就是找到优先级最高的就绪任务，而找出的这个最高优先级任务的TCB，就被赋给变量pxCurrentTCB。

Cortex-M3的中断优先级有些违反直觉：Cortex-M3中断优先级数值越大，表示优先级越低。而FreeRTOS的任务优先级则与之相反：优先级数值越大的任务，优先级越高。

每个任务在自己的环境中运行，不依赖于系统中的其它任务或者RTOS调度器。在任何时刻，只有一个任务得到运行，RTOS调度器决定运行哪个任务。调度器会不断的启动、停止每一个任务，宏观看上去就像整个应用程序都在执行。作为任务，不需要对调度器的活动有所了解，在任务切入切出时保存上下文环境（寄存器值、堆栈内容）是调度器主要的职责。为了实现这点，每个任务都需要有自己的堆栈。当任务切出时，它的执行环境会被保存在该任务的堆栈中，这样当再次运行时，就能从堆栈中正确的恢复上次的运行环境。

挂起：处于挂起状态的任务同样对调度器无效。仅当明确的分别调用vTaskSuspend() 和xTaskResume() API函数后，任务才会进入或退出挂起状态。不可以指定超时周期事件（不可以通过设定超时事件而退出挂起状态）；

    低优先级数值代表低优先级。空闲任务（idle task）的优先级为0（tskIDLE\_PRIORITY）。

在较新的FreeRTOS移植包中，如果试图从一个任务中返回，将会调用configASSERT()（如果定义的话）。如果一个任务确实要退出函数，那么这个任务应调用vTaskDelete(NULL)函数，以便处理一些清理工作。



**任务函数返回为void，参数只有一个void类型指针**。所有的任务函数都应该是这样。 任务函数决不应该返回，因此通常任务函数都是一个死循环。任务由xTaskCreate()函数创建，由vTaskDelete()函数删除。

空闲任务是启动RTOS调度器时由内核自动创建的任务，这样可以确保至少有一个任务在运行。空闲任务具有最低任务优先级，这样如果有其它更高优先级的任务进入就绪态就可以立刻让出CPU。删除任务后，空闲任务用来释放RTOS分配给被删除任务的内存。因此，在应用中使用vTaskDelete()函数后确保空闲任务能获得处理器时间就很重要了。除此之外，空闲任务没有其它有效功能，所以可以被合理的剥夺处理器时间，并且它的优先级也是最低的。

空闲任务钩子是一个函数，每一个空闲任务周期被调用一次。如果你想将任务程序功能运行在空闲优先级上，可以有两种选择：在一个空闲任务钩子中实现这个功能：因为FreeRTOS必须至少有一个任务处于就绪或运行状态，因此钩子函数不可以调用可能引起空闲任务阻塞的API函数（比如vTaskDelay()或者带有超时事件的队列或信号量函数）；创建一个具有空闲优先级的任务去实现这个功能：这是个更灵活的解决方案，但是会带来更多RAM开销。

创建一个空闲钩子步骤如下

在FreeRTOSConfig.h头文件中设置configUSE\_IDLE\_HOOK为1；

定义一个函数，名字和参数原型如下所示：

void vApplicationIdleHook( void );

**通常，使用这个空闲钩子函数设置CPU进入低功耗模式。**

**函数返回错误码，具体参见projdefs.h**

从V9.0版本开始，如果一个任务删除另外一个任务，被删除任务的堆栈和TCB立即释放。如果一个任务删除自己，则任务的堆栈和TCB和以前一样，通过空闲任务删除。 这样做的原因也是显而易见的：如果当前任务删除自己，任务还在运行，怎么可以把这个正运行的TCB和堆栈空间释放掉呢？是不是要等到这个任务交出CPU控制权后再删除，空闲任务正合适做这个删除工作。

RTOS任务间同步一般是使用信号量，任务间通讯一般使用队列。 全局变量做同步一般用在不使用RTOS的场合。 FreeRTOS提供的信号量包括二进制信号、计数信号量、互斥型信号量和递归互斥型信号量，覆盖了各种任务同步情况。 使用信号量同步可以方便的进入阻塞状态，进入阻塞的任务不占用CPU时间，可以指定阻塞超时时间，当信号量有效或者阻塞超时发生，由RTOS内核自动调度任务。 FreeRTOS V8.2.0及其以上版本新增“任务通知”功能，可以更方便的实现任务同步，相比信号量，速度更快，代码更少。

vTaskDelay()函数在文件FreeRTOSConfig.h中，宏INCLUDE\_vTaskDelay 必须设置成1，此函数才能有效。

xTaskResumeFromISR()不可用于任务和中断间的同步，如果中断恰巧在任务被挂起之前到达，这就会导致一次中断丢失（任务还没有挂起，调用xTaskResumeFromISR()函数是没有意义的，只能等下一次中断）。这种情况下，可以使用信号量作为同步机制。在文件FreeRTOSConfig.h中，宏INCLUDE\_vTaskSuspend 和 INCLUDE\_xTaskResumeFromISR 必须设置成1，此函数才有效。

uxTaskGetSystemState这个函数仅用来调试用，调用此函数会挂起所有任务，直到函数最后才恢复挂起的任务，因此任务可能被挂起很长时间。

volatile TickType\_t xTaskGetTickCount(void ); 这个函数不能在ISR中调用。在ISR中用xTaskGetTickCountFromISR()，原型为volatileTickType\_t xTaskGetTickCountFromISR( void )。

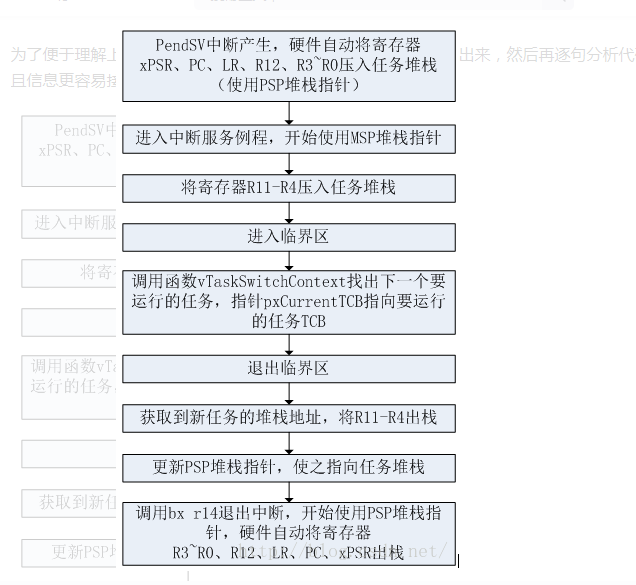
任务因为等待某事件而进入阻塞状态，通常情况下任务会设置一个等待超时周期。如果在等待事件超时，任务会退出阻塞状态。想象一个这样的应用，某任务等待一个事件而进入阻塞状态，但是事件迟迟不发生，超时后任务退出阻塞状态继续执行任务。假如任务等待的事件仍然没有发生，则任务又会阻塞在该事件下。只要任务等待的事件一直不发生，这个任务进入阻塞然后超时退出阻塞，再进入阻塞的循环就会一直存在。是不是可以设定一个总超时时间，只要总阻塞时间大于这个总超时时间，则可以结束这个任务或进行相应记录？freeRTOS提供了两个API函数来完成这个功能，这就是vTaskSetTimeOutState()和xTaskCheckForTimeOut()。

在Cortex-M3架构中，FreeRTOS为了任务启动和任务切换使用了三个异常：SVC、PendSV和SysTick。SVC（系统服务调用）用于任务启动，有些操作系统不允许应用程序直接访问硬件，而是通过提供一些系统服务函数，通过SVC来调用；PendSV（可挂起系统调用）用于完成任务切换，它的最大特性是如果当前有优先级比它高的中断在运行，PendSV会推迟执行，直到高优先级中断执行完毕；SysTick用于产生系统节拍时钟，提供一个时间片，如果多个任务共享同一个优先级，则每次SysTick中断，下一个任务将获得一个时间片。（<https://blog.csdn.net/guozhongwei1/article/details/49544671>）

**任务切换：**

FreeRTOS有两种方法触发任务切换：

* 执行系统调用，比如普通任务可以使用taskYIELD()强制任务切换，中断服务程序中使用portYIELD\_FROM\_ISR()强制任务切换；
* 系统节拍时钟中断



对于Cortex-M3硬件，当系统复位后，默认使用MSP指针。MSP指针用于操作系统内核以及处理异常（也就是说中断服务程序中默认强制使用MSP指针，这是硬件自动设置的）。任务（进程）使用PSP指针，操作系统负责从MSP指针切换到PSP指针。这个过程在《FreeRTOS高级篇3---启动调度器》一文的最后部分中进行了讲解：在SVC中断服务程序中启动第一个任务，当从SVC中断服务退出前，通过向r14寄存器最后4位按位或上0x0D，使得硬件在退出时使用进程堆栈指针PSP完成出栈操作并返回后进入线程模式、返回Thumb状态。

在任务中定义的局部变量，会优先使用寄存器，寄存器不够时就使用任务堆栈的空间。如果在任务中调用其它函数，则调用前的保存信息也存到任务堆栈中去。根据任务代码来估算任务堆栈的大小是件十分重要的技能。

# 四、内核控制

**taskYIELD**：用于强制上下文切换的宏。在中断服务程序中的等价版本为portYIELD\_FROM\_ISR，这也是个宏，实际调用portYIELD，/\* Set a PendSV to request a context switch. \*/

**taskENTER\_CRITICAL：**用于进入临界区的宏。在临界区中不会发生上下文切换。先禁止所有RTOS可屏蔽中断，这可以通过向basepri 寄存器写入configMAX\_SYSCALL\_INTERRUPT\_PRIORITY来实现。basepri寄存器被设置成某个值后，所有优先级号大于等于此值的中断都被禁止，但若被设置为0，则不关闭任何中断，0为默认值。然后临界区嵌套计数器增1。

taskEXIT\_CRITICAL：用于退出临界区的宏。先将临界区嵌套计数器减1，如果临界区计数器为零，则使能所有RTOS可屏蔽中断，这可以通过向basepri 寄存器写入0来实现。

当调用vTaskStartScheduler()后，空闲任务被自动创建。如果configUSE\_TIMERS被设置为1，定时器后台任务也会被创建。

void vTaskSuspendAll( void ); 挂起调度器，但不禁止中断。当调度器挂起时，不会进行上下文切换。调度器挂起后，正在执行的任务会一直继续执行，内核不再调度（意味着当前任务不会被切换出去），直到该任务调用了xTaskResumeAll ()函数。 内核调度器挂起期间，那些可以引起上下文切换的API函数（如vTaskDelayUntil()、xQueueSend()等）决不可使用。

xTaskResumeAll()仅恢复调度器，它不会恢复那些被vTaskSuspend()函数挂起的任务。

 如果RTOS使能tickless空闲功能，每当只有空闲任务被执行时，系统节拍时钟中断将会停止，微控制器进入低功耗模式。当微控制器退出低功耗后，系统节拍计数器必须被调整，将进入低功耗的时间弥补上。

# 五、任务通知

每个RTOS任务都有一个32位的通知值，任务创建时，这个值被初始化为0。RTOS任务通知相当于直接向任务发送一个事件，接收到通知的任务可以解除阻塞状态，前提是这个阻塞事件是因等待通知而引起的。  使用API函数xTaskNotify()和xTaskNotifyGive()（中断保护等价函数为xTaskNotifyFromISR()和vTaskNotifyGiveFromISR()）发送通知，在接收RTOS任务调用API函数xTaskNotifyWait()或ulTaskNotifyTake()之前，这个通知都被保持着。如果接收RTOS任务已经因为等待通知而进入阻塞状态，则接收到通知后任务解除阻塞并清除通知。

禁止这个功能，禁止后每个任务节省8字节内存。但它也有一些限制：

1. 只能有一个任务接收通知事件。
2. 接收通知的任务可以因为等待通知而进入阻塞状态，但是发送通知的任务即便不能立即完成通知发送也不能进入阻塞状态。

   API函数xTaskNotifyTake()有两种方法处理任务的通知值，一种方法是在函数退出时将通知值清零，这种方法适用于实现二进制信号量；另外一种方法是在函数退出时将通知值减1，这种方法适用于实现计数信号量。

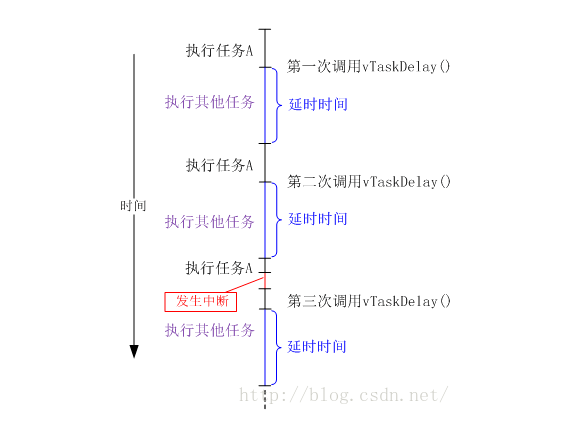
FreeRTOS的信号量是使用队列机制实现的，数据结构也完全是队列的那一套。而任务通知则不同，它的数据结构嵌在任务TCB。任务通知的数据结构包含在任务TCB中，只要任务存在，任务通知数据结构就已经创建完毕，可以直接使用！只有任务可以等待通知，中断服务函数中不可以。

 等待通知API函数只能用在任务中，没有带中断保护版本，因此只有两个API函数：ulTaskNotifyTake()和xTaskNotifyWait ()。

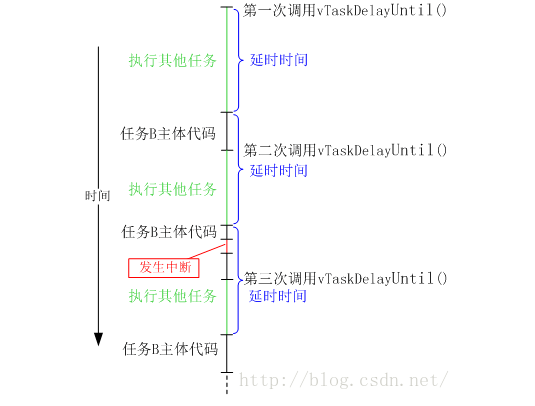
tasks.c中定义了很多局部静态变量，其中有一个变量xTickCount，为了解决xTickCount溢出问题，FreeRTOS使用了两个延时列表：xDelayedTaskList1和xDelayedTaskList2，并使用两个列表指针类型变量pxDelayedTaskList和pxOverflowDelayedTaskList分别指向上面的延时列表1和延时列表2（在创建任务时将延时列表指针指向延时列表）。顺便说一下，上面的两个延时列表指针变量和两个延时列表变量都是在tasks.c中定义的静态局部变量。

        如果内核判断出xTickCount+ xTicksToDelay溢出，就将当前任务挂接到列表指针pxOverflowDelayedTaskList指向的列表中，否则就挂接到列表指针pxDelayedTaskList指向的列表中。

        每次系统节拍时钟中断，中断服务函数都会检查这两个延时列表，查看延时的任务是否到期，如果时间到期，则将任务从延时列表中删除，重新加入就绪列表。如果新加入就绪列表的任务优先级大于当前任务，则会触发一次上下文切换。



调用函数vTaskDelayUntil()开始，每隔固定+  
-周期，任务B的主体代码就会被执行一次，即使任务B在执行过程中发生中断，也不会影响这个周期性，只是会缩短其它任务的执行时间！  
 */\*只有当周期性延时时间大于任务主体代码执行时间,才会将任务挂接到延时列表.\**



# 六、Software Timers

<https://blog.csdn.net/qq_43400642/article/details/100773478>

<https://www.cnblogs.com/yangguang-it/p/7192677.html>

如果使用xTimerCreate（）创建计时器，则会自动从[FreeRTOS堆](https://www.freertos.org/a00111.html)分配此RAM 。如果使用[xTimerCreateStatic（）](https://www.freertos.org/xTimerCreateStatic.html)创建软件定时器， 那么RAM由应用程序编写者提供。

仅当[configTICK\_RATE\_HZ](https://www.freertos.org/a00110.html" \l "configTICK_RATE_HZ)小于或等于1000时，才能使用pdMS\_TO\_TICKS（） 。

# 七、信号量

互斥量与二进制信号量最大的不同是：互斥量具有优先级继承机制。也就是说，如果一个互斥量（令牌）正在被一个低优先级任务使用，此时一个高优先级企图获取这个互斥量，高优先级任务会因为得不到互斥量而进入阻塞状态，正在使用互斥量的低优先级任务会临时将自己的优先级提升，提升后的优先级与与进入阻塞状态的高优先级任务相同。这个优先级提升的过程叫做优先级继承。这个机制用于确保高优先级任务进入阻塞状态的时间尽可能短，以及将已经出现的“优先级翻转”影响降低到最小。

  在很多场合中，某个硬件资源只有一个，当低优先级任务占用该资源的时候，即便高优先级任务也只能乖乖的等待低优先级任务释放资源。这里高优先级任务无法运行而低优先级任务可以运行的现象称为“优先级翻转”。为什么优先级继承能够降低优先级翻转的影响呢？举个例子，现在有任务A、任务B和任务C，三个任务的优先级顺序为任务C>任务B>任务A。任务A和任务C都要使用某一个硬件资源，并且当前任务A占有该资源。

      先看没有优先级继承的情况：任务C也要使用该资源，但是此时任务A正在使用这个资源，因此任务C进入阻塞，此时三个任务的优先级顺序没有发生变化。在任务C进入阻塞之后，某硬件产生了一次中断，唤醒了一个事件，该事件可以解除任务B的阻塞状态。在中断结束后，因为任务B的优先级是大于任务A的，所以任务B抢占任务A的CPU权限。那么任务C的阻塞时间就至少为：中断处理时间+任务B的运行时间+任务A的运行时间。

      再看有优先级继承的情况：任务C也要使用该资源，但是此时任务A正在使用这个资源，因此任务C进入阻塞，此时由于优先级A会继承任务C的优先级，三个任务的优先级顺序发生了变化，新的优先级顺序为：任务C=任务A>任务B。在任务C进入阻塞之后，某硬件产生了一次中断，唤醒了一个事件，该事件可以解除任务B的阻塞状态。在中断结束后，因为任务A的优先级临时被提高，大于任务B的优先级，所以任务A继续获得CPU权限。任务A完成后，处于高优先级的任务C会接管CPU。所以任务C的阻塞时间为：中断处理时间+任务A的运行时间。看，任务C的阻塞时间变小了，这就是优先级继承的优势。

# 八、列表和类表项

xItemValue是列表项值，通常是一个被跟踪的任务优先级或是一个调度事件的计数器值。如果任务因为等待从队列取数据而进入阻塞状态，则任务的事件列表项的列表项值保存任务优先级有关信息，状态列表项的列表项值保存阻塞时间有关的信息。这个变量被configLIST\_VOLATILE修饰，configLIST\_VOLATILE被映射成C语言关键字volatile，表明这个变量是“易变的”，告诉编译器不得对这个变量进行代码优化，因为列表项的成员可能会在中断服务程序中被更新。

列表项所在的位置取决于列表项的列表项值，按照升序序排列。

# 九、队列

发送到队列的消息是通过拷贝实现的，这意味着队列存储的数据是原数据，而不是原数据的引用。

挂起调度器并不会禁止中断，中断服务函数仍然可以操作队列事件列表，可能会解除任务阻塞、可能会进行上下文切换，这是不允许的。于是，解决办法是不但挂起调度器，还要给队列上锁！

创建互斥量时会先释放一个互斥量，表示这个资源可以使用。任务想访问资源时，先获取互斥量，等使用完资源后，再释放它。也就是说互斥量一旦创建好后，要先获取，后释放，要在同一个任务中获取和释放。

  有了上面的基础理论，我们就很好理解为什么释放互斥量会比较复杂了。还是可以简化为两种情况：第一，如果队列未满，除了队列结构体成员uxMessageWaiting加1外，还要判断获取互斥量的任务是否有优先级继承，如果有的话，还要将任务的优先级恢复到原始值。当然，恢复到原来值也是有条件的，就是该任务必须在没有使用其它互斥量的情况下，才能将继承的优先级恢复到原始值。然后判断是否有阻塞的任务，有的话解除阻塞，最后返回成功信息（pdPASS）；第二，如果如果队列满，返回错误代码（err\_QUEUE\_FULL），表示队列满。

# 十、系统节拍时钟

 系统节拍中断服务程序会调用函数xTaskIncrementTick()来完成主要工作，如果该函数返回值为真（不等于pdFALSE），说明处于就绪态任务的优先级比当前运行的任务优先级高。这会触发一次PendSV中断，进行上下文切换。如果调度器挂起，正在执行的任务会一直继续执行，内核不再调度（意味着当前任务不会被切换出去），直到该任务调用了xTaskResumeAll()函数。

  对于队列以及使用队列机制的信号量、互斥量等，在中断服务程序中调用了这些API函数，将任务从阻塞中解除，则需要调用函数xTaskRemoveFromEventList()将任务的事件列表项从事件列表中移除。在移除事件列表项的过程中，会判断解除的任务优先级是否大于当前任务的优先级，如果解除的任务优先级更高，会将变量xYieldPending设置为pdTRUE。在下一次系统节拍中断服务函数中，触发一次任务切换。

# 十一、事件组

<https://www.cnblogs.com/yangguang-it/p/7189607.html>

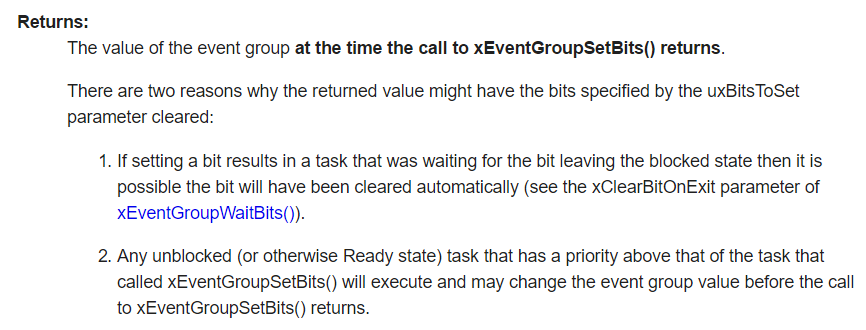
<https://www.cnblogs.com/yangguang-it/p/7932467.html>

任务间事件标志组的实现是指各个任务之间使用事件标志组实现任务的通信或者同步机制。

事件标志组是实现多任务同步的有效机制之一，使用全局变量相比事件标志组主要有如下三个问题：

1. 使用事件标志组可以让 RTOS 内核有效地管理任务，而全局变量是无法做到的，任务的超时等机制需要用户自己去实现。
2. 使用了全局变量就要防止多任务的访问冲突，而使用事件标志组则处理好了这个问题，用户无需担心。
3. 使用事件标志组可以有效地解决中断服务程序和任务之间的同步问题。

用户通过参数 uxBitsToSet 设置的标志位并不一定会保留到此函数的返回值中，下面举两种情况：

  
a. 如果设置一个位导致等待该位的任务离开阻塞（注意是离开阻塞态，即使没有进入运行态，只要离开阻塞态即可）状态，则该位可能会被自动清除（请参阅xEventGroupWaitBits（）的xClearBitOnExit参数）。  
b. 调用此函数的任务是一个低优先级任务，通过此函数设置了事件标志后，让一个等待此事件标志  
的高优先级任务就绪了，会立即切换到高优先级任务去执行，相应的事件标志位会被函数  
xEventGroupWaitBits 清除掉，等从高优先级任务返回到低优先级任务后，函数xEventGroupSetBits 的返回值已经被修改。

函数 xEventGroupSetBitsFromISR 对事件标志组的操作是不确定性操作，因为不知道当前有多少个任务在等待此事件标志。而 FreeRTOS 不允许在中断服务程序和临界段中执行不确定性操作。 为了不在中断服务程序中执行，就通过此函数给 FreeRTOS 的 daemon 任务（就是 FreeRTOS 的定时器任务）发送消息，在 daemon 任务中执行事件标志的置位操作。 同时也为了不在临界段中执行此不确定操作，将临界段改成由调度锁来完成。这样不确定性操作在中断服务程序和临界段中执行的问题就都得到解决了。