未扑先知：结构体指针的强制转换比较灵活，只需要记住指针就是地址即可。指针的操作就是对地址的操作，void类型的指针不能进行操作，因为无法知道加1时地址应该加多少。freeRTOS优先级，数值越大，优先级越高。

1. 列表项(list.h list.c)

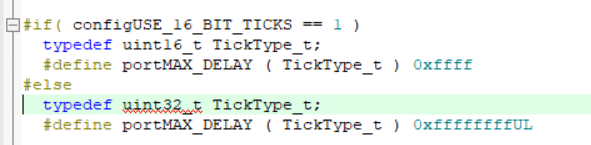
列表项中的结构体：



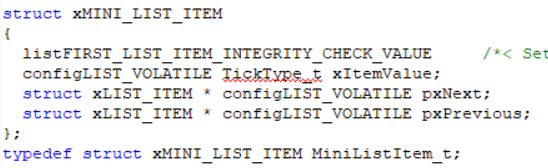


结构体xLIST\_ITEM详解：

xItemValue用来标识xLIST\_ITEM的值，TickType\_t 是类型uint32\_t或uint16\_t，由portmacro.h中的configUSE\_16\_BIT\_TICKS定义，并且还与portMAX\_DELAY息息相关，xItemValue一般用作排序，后面list.c文件中的函数void vListInsert（）函数会用到。



下面pxNext与pxPrevious两个则是指向前级列表项节点及后级列表项节点的指针，configLIST\_VOLATILE看作是volatile关键字即可，pvOwner指针变量，指向拥有该节点的内核对象，一般是TCB；pvContainer是一个指针，主要指向节点所在的列表，

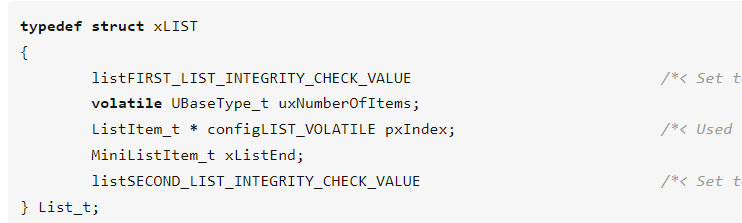




结构体xMINI\_LIST\_ITEM详解：

xItemValue用来标识xMINI\_LIST\_ITEM的值，TickType\_t 是类型uint32\_t或uint16\_t，由portmacro.h中的configUSE\_16\_BIT\_TICKS定义，并且还与portMAX\_DELAY息息相关，xItemValue一般用作排序，pxNext和pxPrevious为xLIST\_ITEM指针变量，分别指向下一个xLIST\_ITEM节点和上一个xLIST\_ITEM节点

xLIST结构体详解：

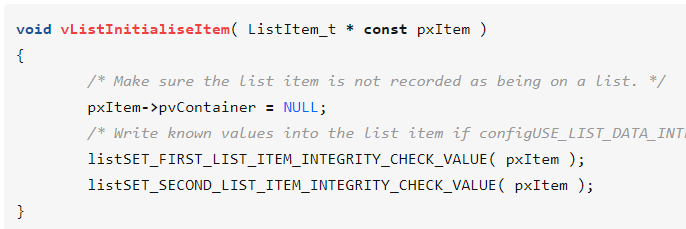




UBaseType\_t为无符号长整形数据类型，参数uxNumberOfItems链表节点计数器，用来表示该链表下有多少个链表节点的，pxIndex指针变量，用于遍历链表的所有节点，指向头节点内xListEnd的地址，xListEnd是一个MiniListItem\_t节点，是链表的最后一个节点。因为FreeRTOS定义的链表是循环双向链表，因此XListEnd也是链表的第一节点。

1. 列表函数详解

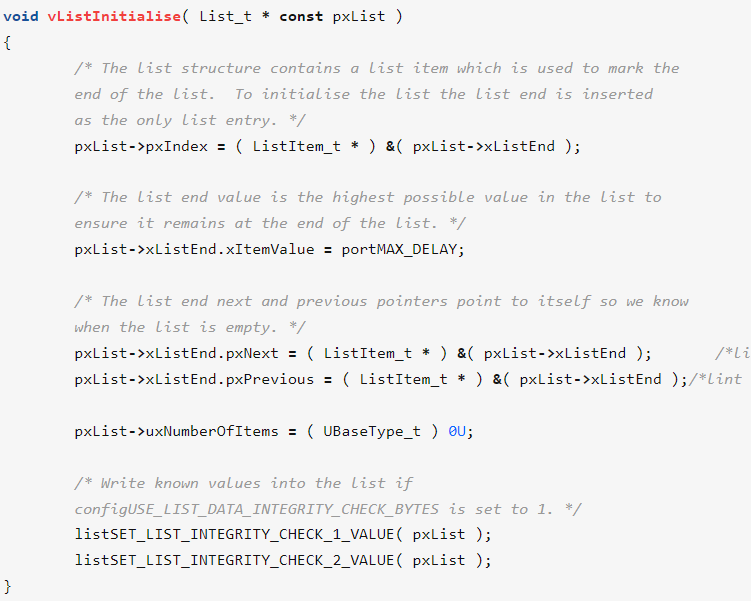
普通链表初始化函数void vListInitialiseItem( ListItem\_t \* const pxItem ）



让此列表项的pvContainer为NULL，用以标识不属于任何链表。



链表头节点（最后一个节点）的初始化void vListInitialise( List\_t \* const pxList )



初始化时让pxIndex指向其结构体系中的列表结尾项（ListEnd），并且让列表结尾项中的xItemValue的值设为portMAX\_DELAY。让列表结尾项（ListEnd）中的pxNext及pxPrevious指向自身的列表结尾项（ListEnd）。接着设置uxNumberOfItems的值为0，uxNumberOfItems记录本链表种普通节点（ListItem\_t节点）的总数量。



链表新增普通链表节点。void vListInsertEnd( List\_t \* const pxList, ListItem\_t \* const pxNewListItem )



传入一个头节点，接着传入一个列表项。

首先将pxList头结点中的pxIndex指向的节点单元赋值给临时变量。让新增节点的pxNext指针指向pxIndex的地址，让pxIndex指向的单元的pxPrevious的指向的地址赋值给新增节点的pxPrevious。让PxIndex之前节点的下一个节点指向新节点，接着让pxIndex前的节点指向新节点。



第一步申明一个临时节点;



第二步设定新插入节点的前节点及后节点指向级临时节点串联。



最后再让pvContainer指向头结点。

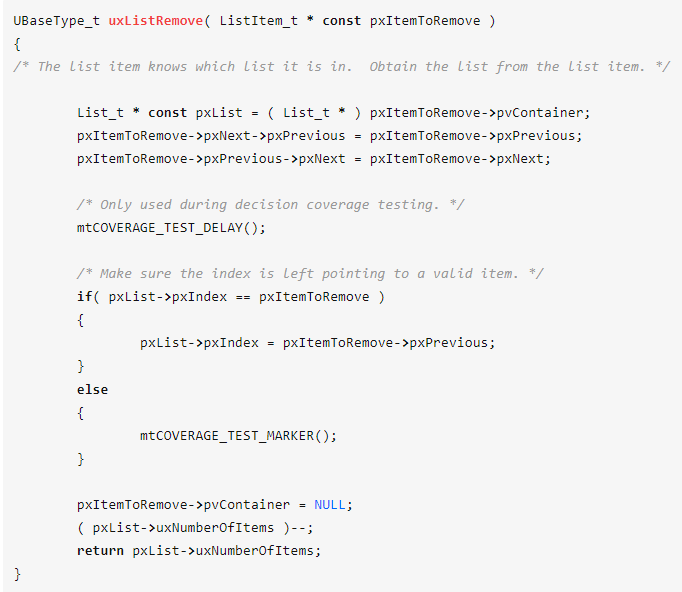
当插入第二个节点时：





由此可见，插入节点的本质就是在头结点中的xListEnd前，插入节点，也就是链表节点插入链表。

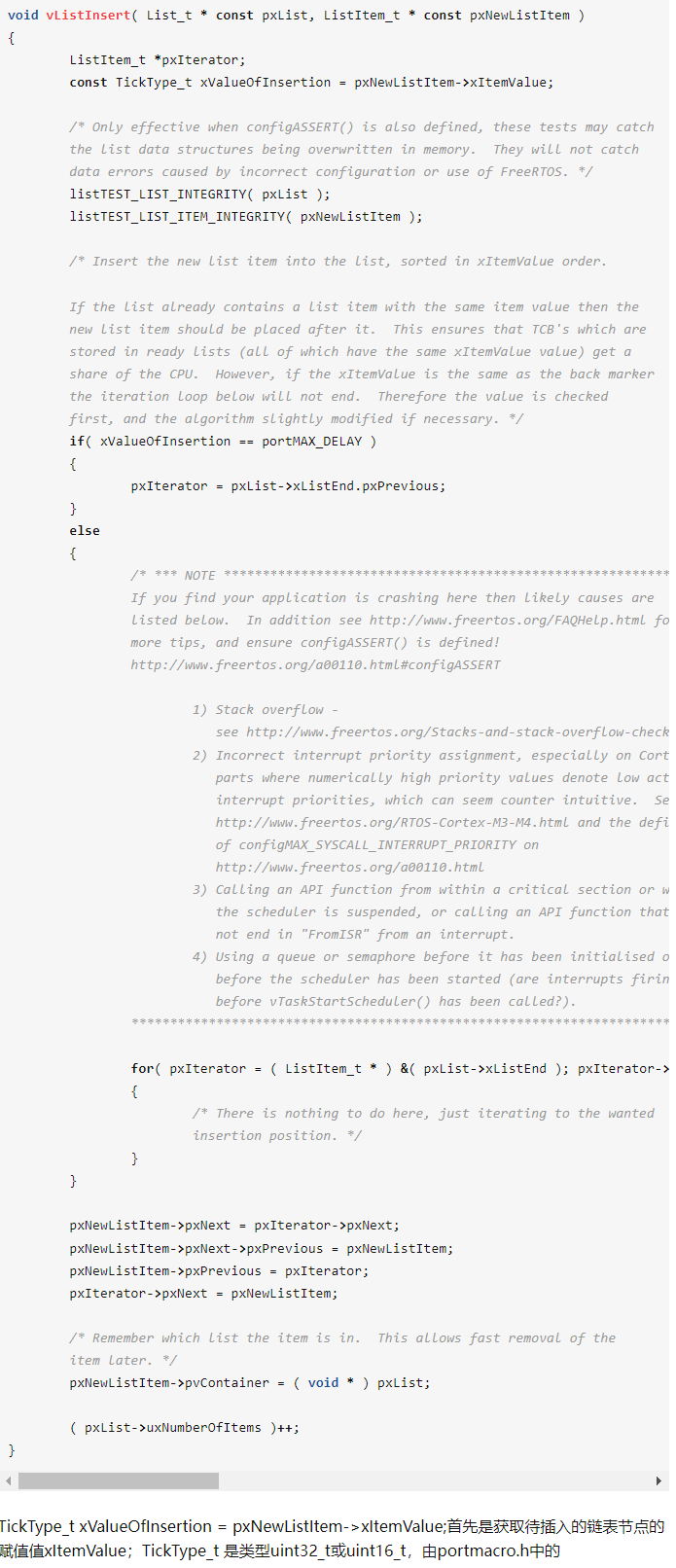
链表删除普通链表节点。UBaseType\_t uxListRemove( ListItem\_t \* const pxItemToRemove )负责删除链表中的普通链表节点；



传入所要删除的节点，返回删除节点链表的数量。

首先定义了一个临时的链表头指针pxList，让他与所要删除的节点所在的链表地址一致。接着把所删除节点的前一个节点指向所删除节点的后一个节点，把所删除节点的后一个节点指向所删除节点的前一个节点。接着删除节点的容器为空，表示不存在任何一个链表，接着链表元素减减，之后返回所剩的节点数。

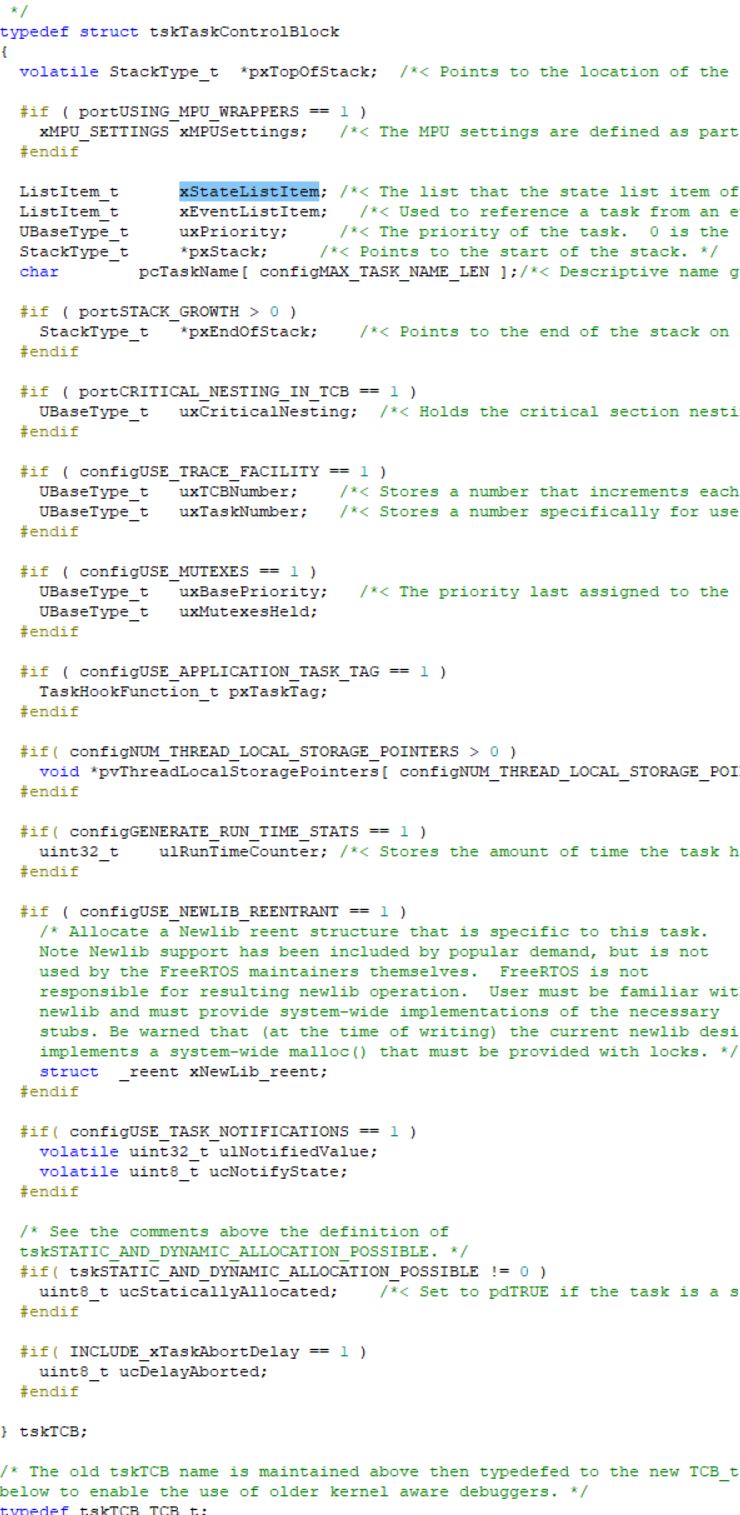
将普通节点按照升序排序增加到链表中void vListInsert( List\_t \* const pxList, ListItem\_t \* const pxNewListItem )



传入链表头结点及列表项节点。首先声明一个指向列表项节点的指针，大概为pxIterator，其次声明一个变量至用于存储当前节点的xItemValue值。如果插入的列表项的值为最大值，那么就让pxIterator的值指向链表最后节点之前的节点。否则就找到比插入节点的xItemValue值大的节点的之前的节点，并且把地址给pxIterator。接着就是常规的插入操作，比较易懂。

二、任务的定任务切换的实现（task.c task.h）

任务控制块

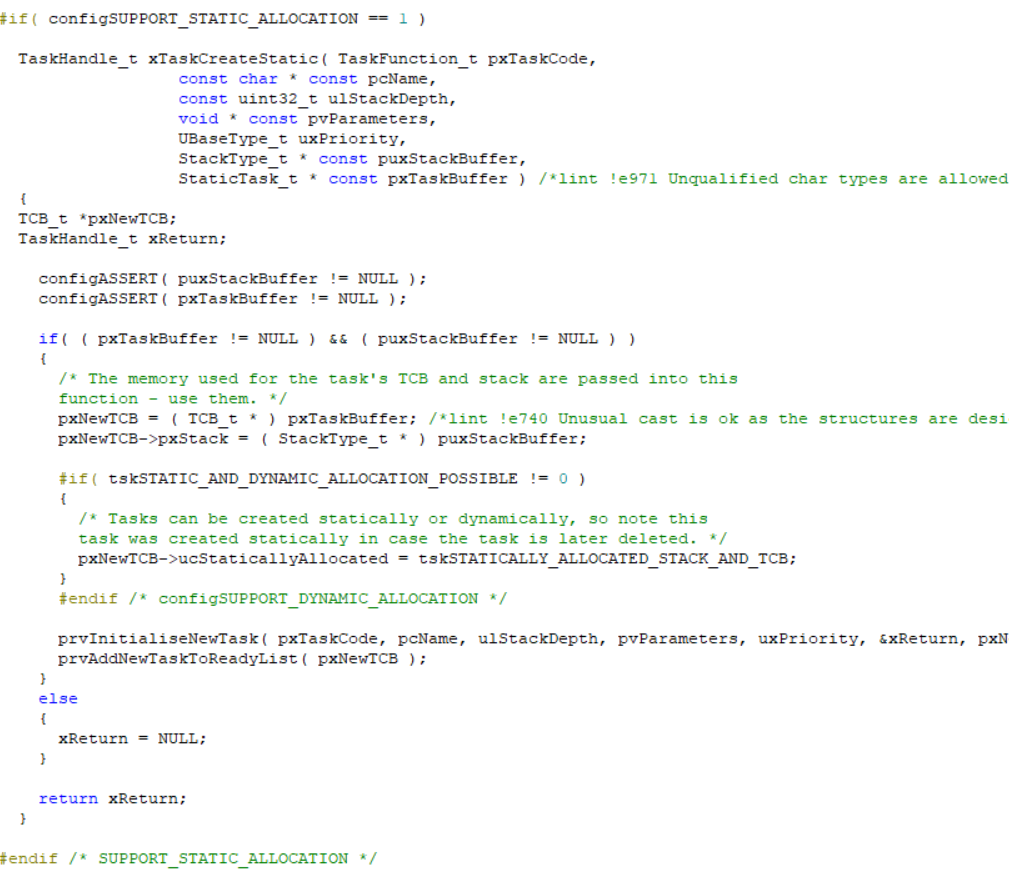


任务控制块程序详解



首先定义了一个栈顶，StackType\_t \*pxToOfStack,StackType\_t为无符号32位整型，其次还有两个个列表项数据结构，xStateListItem及xEvevtListItem。接着定义了任务的优先级uxPriority，数据类型为UBaseType\_t,为32位无符号长整型，pxStack为任务栈的起始地址，任务名字为pcTaskName。Freertos中用TCB\_t重定义了tskTCB。

静态任务创建函数xTaskCreateStatic（）：



返回值为任务句柄TaskHandle\_t，其本质为一个空指针，此指针指向任务控制块，空指针只能起标示作用，无法进行指针操作。第一个参数类型定义，typedef void (\*TaskFunction\_t)( void \* )说明这是一个返回值为void且输入参数为void\*的函数指针TaskFunction，第二个参数pcName为输入的名字，是一个char类型的指针，指针指向的指针的指向都不能改变。第三个参数为堆栈的深度也即就是大小，由ulStackDepth指示，pvParameters则为创建任务的优先级。puxStackBuffer为任务栈起始地址，StackType\_t本质上为32位无符号整型。首先定义了一个任务控制块指针pxNewTCB，创建了一个任务句柄为xReturn，此值作为函数的返回值。此处需要了解一个知识：断言函数。

assert() 会对表达式expression进行检测：

如果expression的结果为0（条件不成立），那么断言失败，表示程序出错，assert()会向标准输出设备(一般是显示器)打印一条错误信息，并调用abort（）函数终止程序的执行。如果expression的结果为非0(条件成立)，那么断言成功，表明程序正确，assert()不进行任何操作。Freertos的断言功能，代码如下所示：

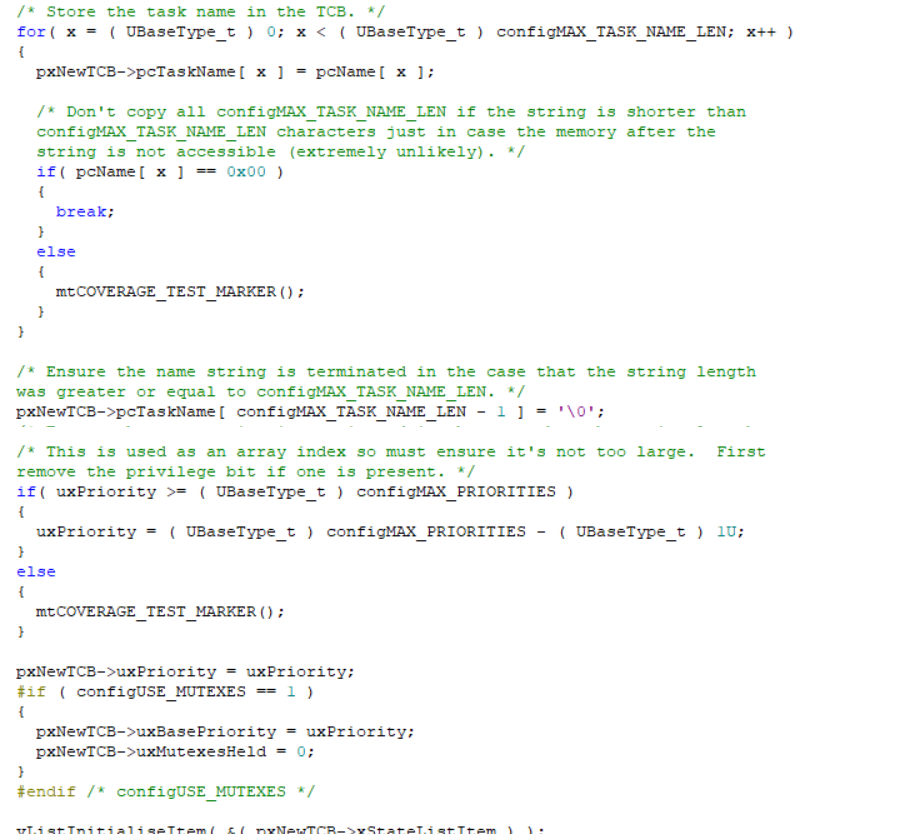


FreeRTOS的断言功能在调试阶段是非常有用的，可以有效地检查参数错误和运行中的错误，但在正式发布软件时，请将此功能关闭，因为断言功能会增加工程代码大小并降低工程执行效率。关闭断言也比较简单，如果FreeRTOSConfig.h文件中有断言的宏定义，将其注释掉即可，如果没有宏定义，默认在FreeRTOS.h文件中就是关闭的。接着用cofigASSERT断言指针是否非空。两个都不为空则进入下列代码，让任务控制块的任务栈指向所传入的参数，然后程序进入prvInitialiseNewTask（）函数运行，接下来看此函数的相关解析。由于这个函数很长，我们需要逐步分析：

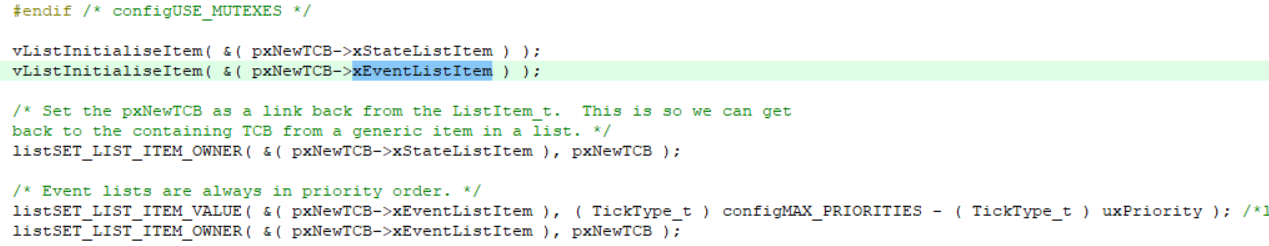


可以清楚地看到，他的输入参数为任务函数pxTaskCode，任务名字pcName

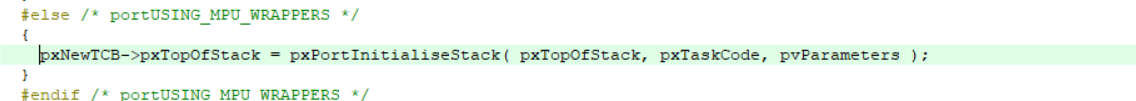
，堆栈大小ulStackDepth，任务参数pxCreatedTask,任务控制块pxNewTCB，最后一个不用管，传值传了个空。函数首先定义了一个栈顶指针pxTopOfStack及x值，接着看到portSTACK\_GROWTH的值，获取栈顶地址需要用栈初始指针加栈的深度减1，这样的等于定位到数组的最后一个字节，这是因为栈顶是高字节在上，而低字节在下的。接着要做栈顶八字节对其，为什么要做八字节对齐呢？这是因为有些浮点型操作是八字节的，按道理只需要4字节对齐，因为是32位的微型处理器。portPOINTER\_SIZE\_TYPE为32位无符号整形，八字节对齐的原理为portBYTE\_ALIGN为7，这样操作的话等同于把地址的最后三个位都变成0，这样的话就做到了八字节对齐。

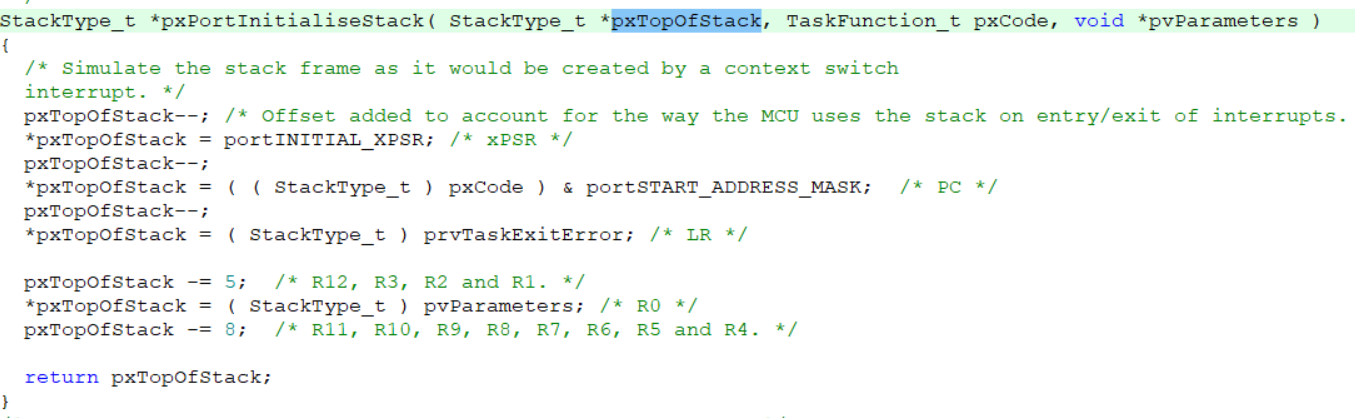


接着传入的名字参数赋值给任务控制块的pcTaskName，此处添加了两个结束符，名字后面添加了一个结束符，总后面添加了一个结束符接着赋值优先级。如果优先级比最大优先级还大，那么优先级就是最大优先级减一，接着把这个任务优先级赋值给任务控制块。

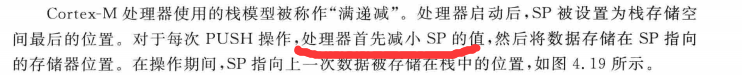


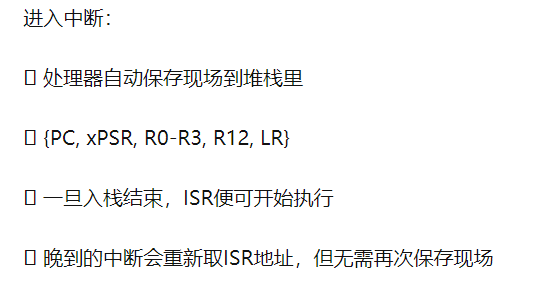
接着初始化任务控制块中的xStateListItem及xEventListItem节点，并且设置任务状态节点（xStateListItem）的拥有者为任务控制块（PxNewTCB），设置事件节点（xEventListItem）的值为最大优先级减去当前任务优先级，并设置任务事件节点（xEventListItem）的拥有者为任务控制块（PxNewTCB）。接着调用下列函数初始化任务堆栈：

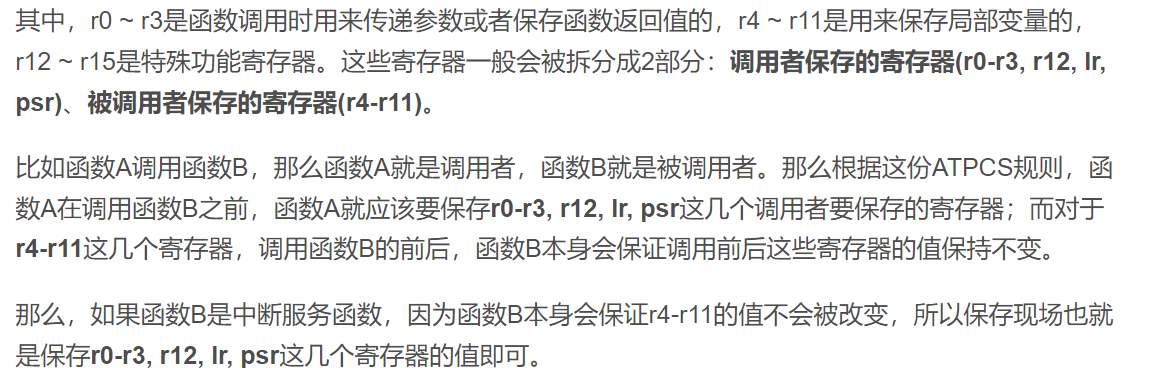


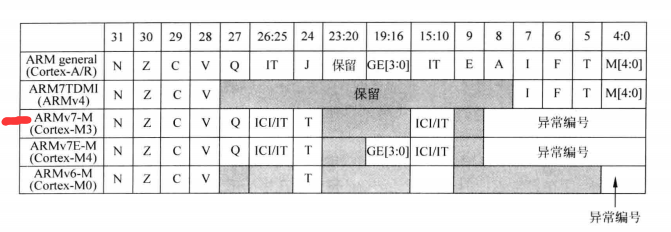


接着我们分析任务栈初始化函数pxPortInitialiseStack(),输入参数（\*pxTopOfStack）为栈顶指针，任务函数(pxCode)，及携带的参数(pvParamrters)。 调用函数pxPortInitialiseStack()后，相当于执行了一次系统节拍时钟中断:将一些重要寄存器入栈。虽然任务还没开始执行,也并没有中断发生,但看上去就像寄存器已经被入栈了，并且部分堆栈值被修改成了我们需要的已知值。对于不同的硬件架构，入栈的寄存器也不相同，所以我们看到这个函数是由移植层提供的。对于Cortex-M3架构，需要依次入栈xPSR（此处portINITAL\_XPSR的值为0x01000000 ,其中bit24被置1，表示是Thumb状态）、PC（寄存器PC被初始化为任务函数指针vTask\_A与portSTART\_ADDRESS\_MASK（0xfffffffeUL）这样当某次任务切换后，任务A获得CPU控制权，任务函数vTask\_A被出栈到PC寄存器，之后会执行任务A的代码）、LR（prvTaskExitError，这是一个防止错误乱跳的函数）、R12、R3~R0、R11~R4，假设堆栈是向下生长的，初始化后的堆栈如图3-1所示。首先减1的目的：添加的偏移量是为了说明 MCU 在中断进入/退出时使用堆栈的方式。详细内容可参考[(10条消息) STM32异常与中断过程详解\_luobeihai的博客-CSDN博客\_stm32中断处理过程](https://blog.csdn.net/luobeihai/article/details/117904999)。











初始化后堆栈大概为这个样子。最后让任务句柄指向任务控制块。



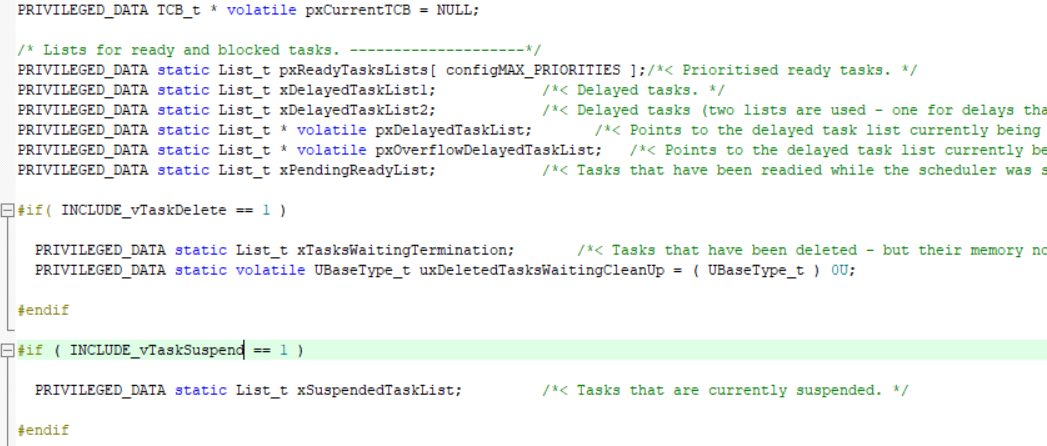
接着调用prvAddNewTaskToReadyList（）函数把创建的任务控制块放入就绪链表中。参数为创建的任务控制块。



首先taskENTER\_CRITICAL()函数为进入临界段函数，进入临界段函数的作用是关闭所有中断。接着让当前任务数全局变量(uxCurrentNumberOfTasks)加1,接着判断当前任务控制块的指向，如果为空则让当前的任务控制块指向新创建任务的任务控制块，接着继续判断前任务数全局变量(pxCurrentTCB)是否为1，为1的话说明是第一个任务，第一个任务则需要进入prvInitialiseTaskLists();这个函数的作用是初始化任务列表。接着我们来看看这个函数：

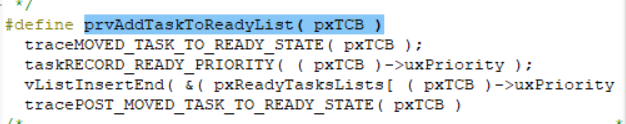


首先进入一个for循环，configMAX\_PRIORITIES为freertos的能管理的最大任务优先级，次吃设置的是32。说明freertos最大能管理32位优先级的任务，task.c中噪音近定义了任务相关链表如下所示：



接着程序会初始化任务就绪列表链表（pxReadyTasksLists）、延时链表（xDelayedTaskList1）、（xDelayedTaskList2）及挂起链表（xSuspendedTaskList）xSuspendedTaskList链表接着让pxDelayedTaskList指针指向延时链表1（xDelayedTaskList1），让溢出延时链表指针（pxOverflowDelayedTaskList）指针指向延时链表2（xDelayedTaskList1），函数完。

如果当前任务控制块非空，如果当前任务调度器未开启，则会判断当前任务控制块与新创建的任务控制快的优先级，谁高则把当前任务控制快的指向指给谁。接着uxTaskNumber++；接着调用prvAddTaskToReadyList( pxTCB )函数，，其本质为



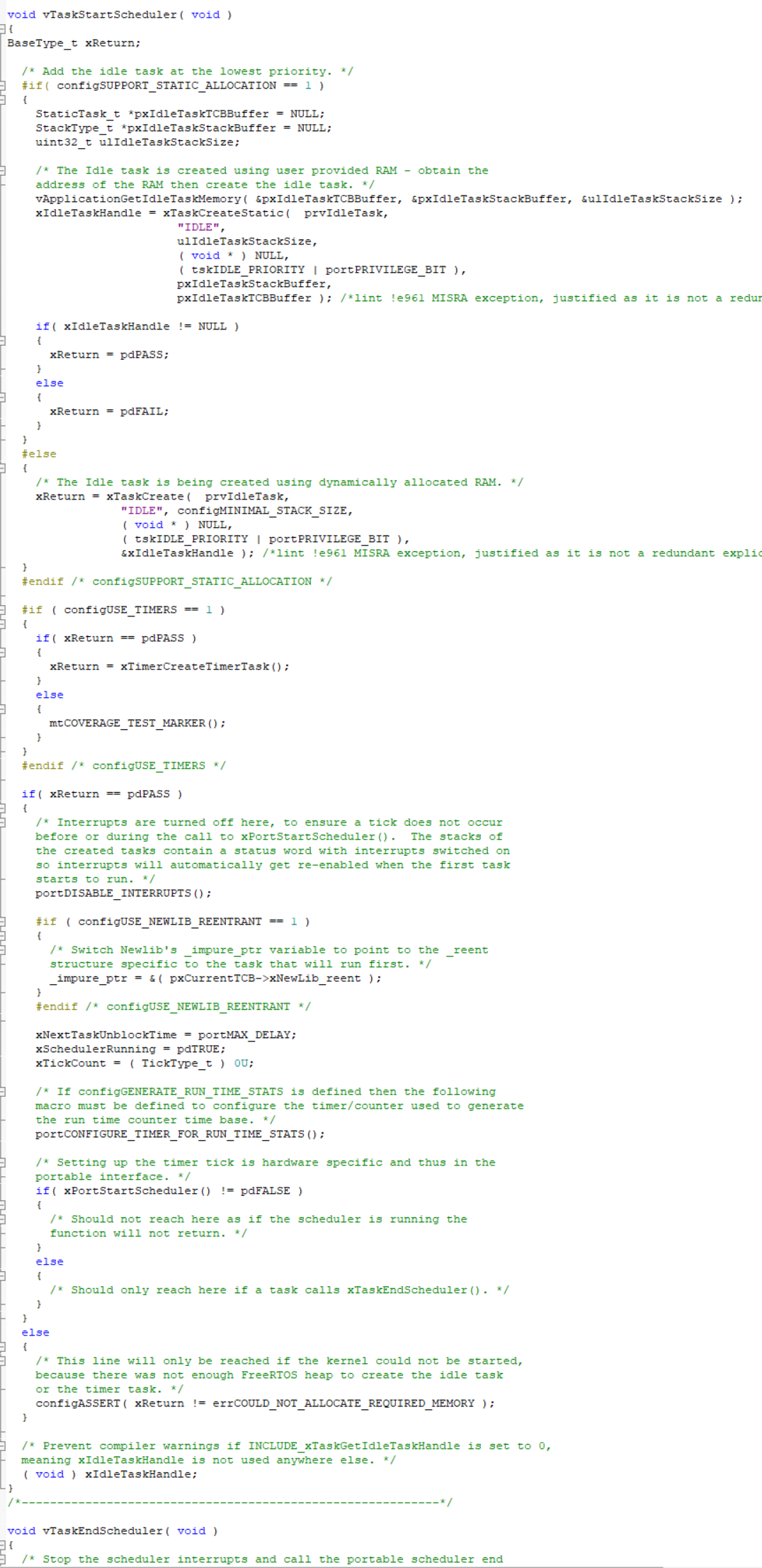
逐个分析：taskRECORD\_READY\_PRIORITY( ( pxTCB )->uxPriority );感觉次数用了一个变量的某个位指示优先级。

#define taskRECORD\_READY\_PRIORITY(uxPriority) portRECORD\_READY\_PRIORITY( uxPriority, uxTopReadyPriority ) ，uxTopReadyPriority 为一个为0的变量，uxTopReadyPriority第几位为1就是命优先级为几的就绪链表有任务。接着执行

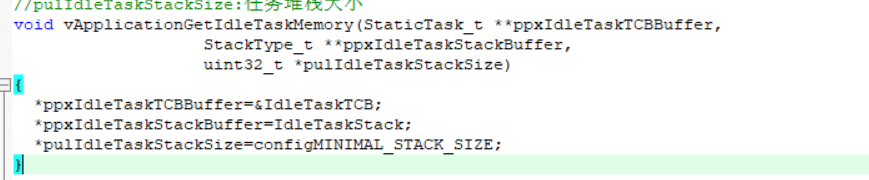
vListInsertEnd( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &( ( pxTCB )->xStateListItem ) );

作用是将任务控制块放到对应优先级的链表中。

接着看到启动任务调度器函数：void vTaskStartScheduler( void )

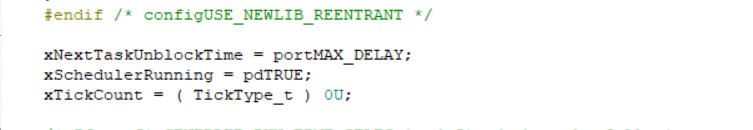


首先看到宏定义是否允许静态创建变量，首先我们允许静态创建变量，首先创建一个任务控制块指针（pxIdleTaskTCBBuffer）及创建任务堆栈指针（pxIdleTaskStackBuffer），接着创建创建了一个32位的无符号整形指示空闲任务堆栈大小(ulIdleTaskStackSize)。接着调用vApplicationGetIdleTaskMemory(&pxIdleTaskTCBBuffer,&pxIdleTaskStackBuffer, &ulIdleTaskStackSize );如下，作用后就是初始化空闲任务控制块，空闲任务堆栈及空闲任务堆栈大小。



接着创建空闲任务，创建定时器管理任务，xReturn = xTimerCreateTimerTask();定时器管理任务分析如下：





先看到 prvCheckForValidListAndQueue( void )函数，函数作用是创建列表与消息队列。



首先初始化了两个列表（xActiveTimerList1、xActiveTimerList2）。并且让pxCurrentTimerList指针指向xActiveTimerList1，pxOverflowTimerList指针指向xActiveTimerList2，接着执行

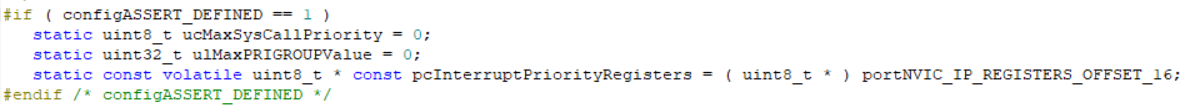
xTimerQueue=xQueueCreateStatic((UBaseType\_t ) configTIMER\_QUEUE\_LENGTH, sizeof( DaemonTaskMessage\_t ), &( ucStaticTimerQueueStorage[ 0 ] ), &xStaticTimerQueue );configTIMER\_QUEUE\_LENGTH的值为5，代表的是软件定时器的长度为5。

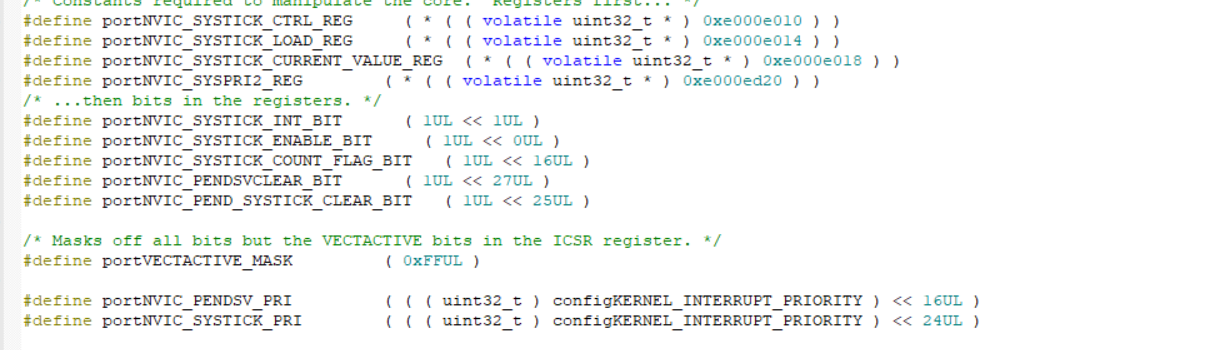
接着设置xNextTaskUnblockTime，暂时不知道有啥用，接着会把调度器的状态设为pdTRUE，说明调度器正在运行，接着把xTickCount的值设为0。

接着就是进行最为关键的函数：xPortStartScheduler()。其作用为启动任务调度器



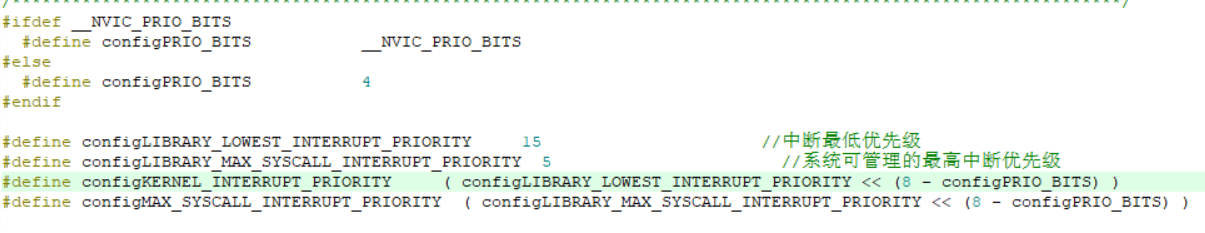




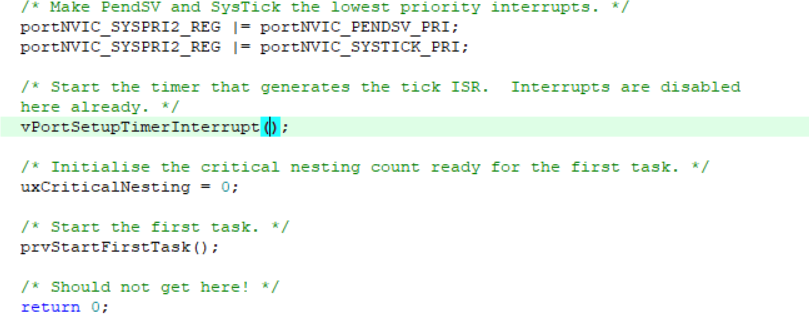




这里是将地址（0xE000ED20、0xE000ED21、0xE000ED22、0xE000ED23）组成一个32位的数值，低位存在低地址，高位存在高地址，这就是为什么portNVIC\_PENDSV\_PRI要位移的原因。如下图所示：

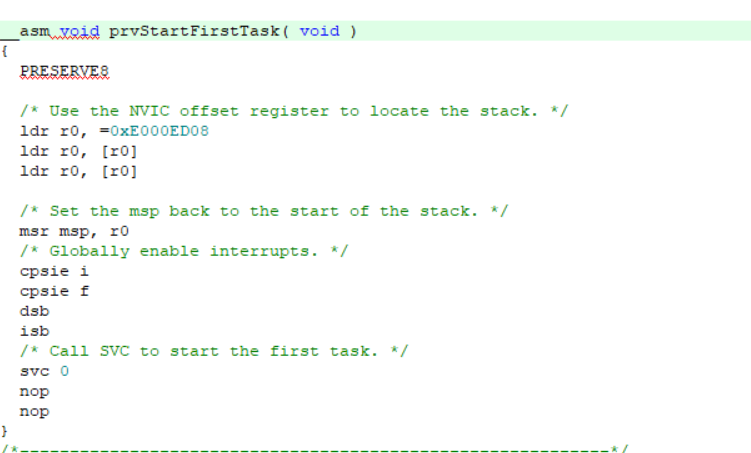


configPRIO\_BITS的值为4，表示用于cortexm3的中断的位数为4,下面就是简单的位移操作，简单易懂。为什么要把pendsv和systick的中断优先级配置成最低？因为系统要先响应其他硬件中断的优先级。

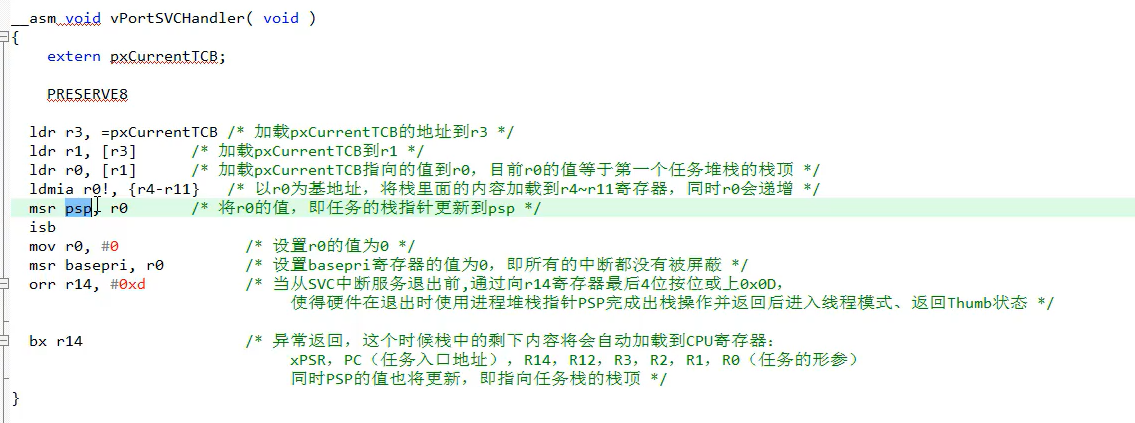


设置定时器相关函数：暂时不明白：vPortSetupTimerInterrupt();

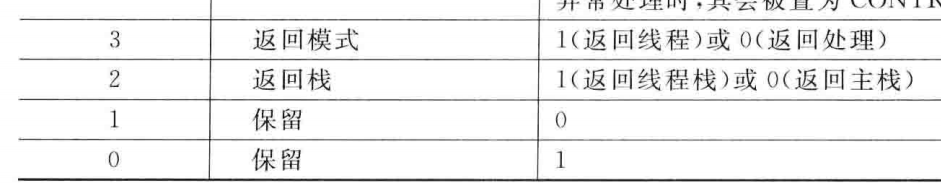
开启第一个任务函数解析：prvStartFirstTask()

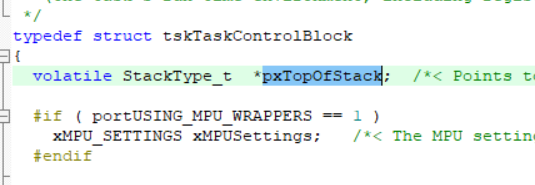


首先是八字节对其关键字PRESERVE8，因为是汇编文件，所以应该八字节对其。Cortexm3硬件中，0xE00ED08地址为SCB->VTOR（向量表偏移地址）寄存器，存储向量表的起始地址。接着把向量表的地址给r0,接着取出向量表0字节数据给r0，接着把r0的值赋值给msp， MRS指令用于将程序状态寄存器的内容传送到通用寄存器中，初始化主堆栈msp，接着调用svc中断。Svc中断如下所示。



这一段与cortexm3内核有关，因为是汇编指令，所以应该加八字节对其关键字（PRSERVE8）。pxCurrentTCB是一个指向就绪控制块地址的指针，加载pxCurrentTCB值赋值给r1，接着加载加载任务控制块值（任务控制块第一个值为栈顶）赋值给r0。压栈赋值，接着让psp栈指针指向r11的位置，因为cortexm3压栈过程为栈顶减一优先，所以指向的为r11所在的位置，ISB指令同步隔离。最严格：它会清洗流水线，以保证所有它前面的指令都执行完毕之后，才执行它后面的指令。接着设置BASEPRI的值为0，表明不屏蔽任何中断。ORR 指令时按位或，所以 ORR R14, #0xd 相当于 R14 |= 0xd；这个操作也和体系架构相关，R14 是链接寄存器 LR，在 ISR 中（此刻我们在 SVC 的 ISR 中），它记录了异常返回值 EXC\_RETURN。Oxd0表示的就是返回线程及返回线程栈。 最后执行 bx R14，告诉处理器ISR完成，需要返回，此刻处理器便会使用 PSP 做为堆栈指针，进行出栈操作，将xPSR、PC、LR、R12、R3~R0 出栈，初始化的时候，PC 被我们赋值成为了执行任务的函数的入口，所以呢，就正常跳入到了优先级最高的 Ready 状态的第一个任务的入口函数了；

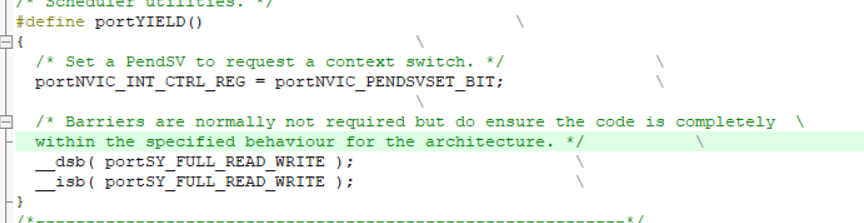


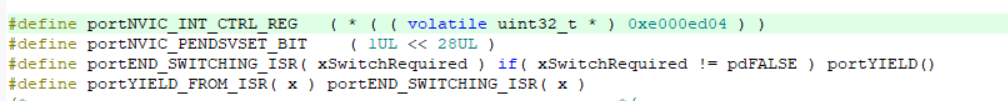


而任务控制块的第一个地址就是栈顶，任务初始化之后的栈顶如下所示：



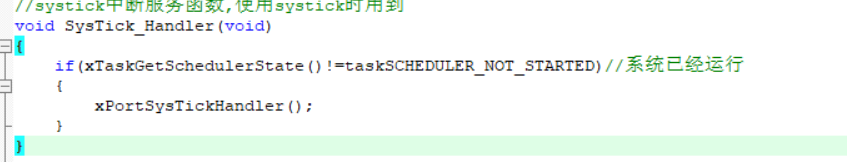
任务切换函数taskYIELD()，portYIELD的实现很简单，实际就是将pendsv的悬起位置1，当没有其他中断运行的时候响应pendsv中断，去执行我们写好的pendsv中断服务函数，在里面实现任务切换。

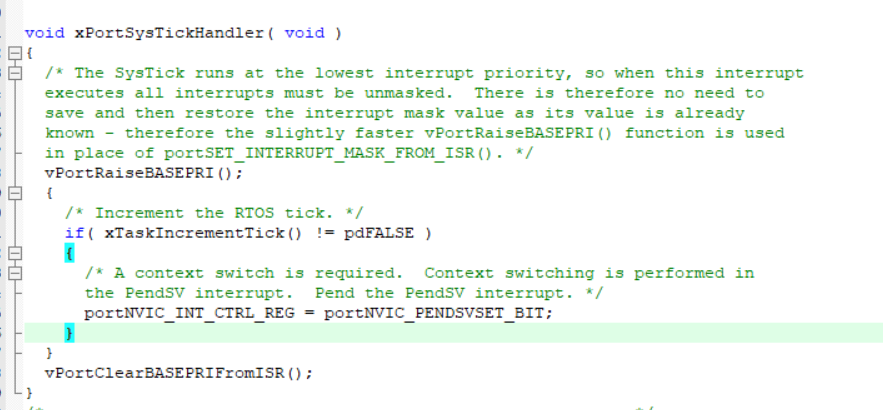




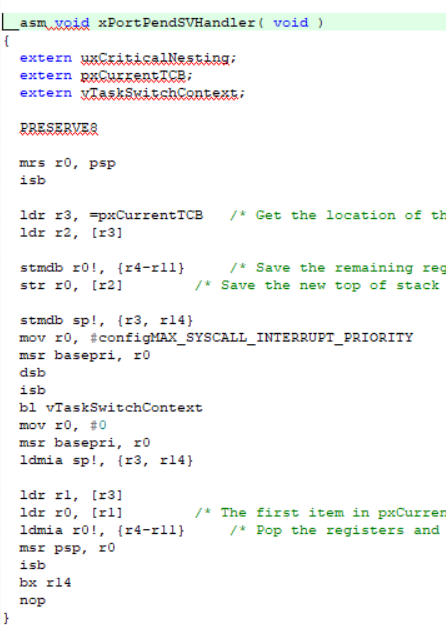
其中已经定义了悬起位的地址，方便赋值。portSY\_FULL\_READ\_WRITE的值为15，

systick函数的本质就是为了进入pendsv中断。

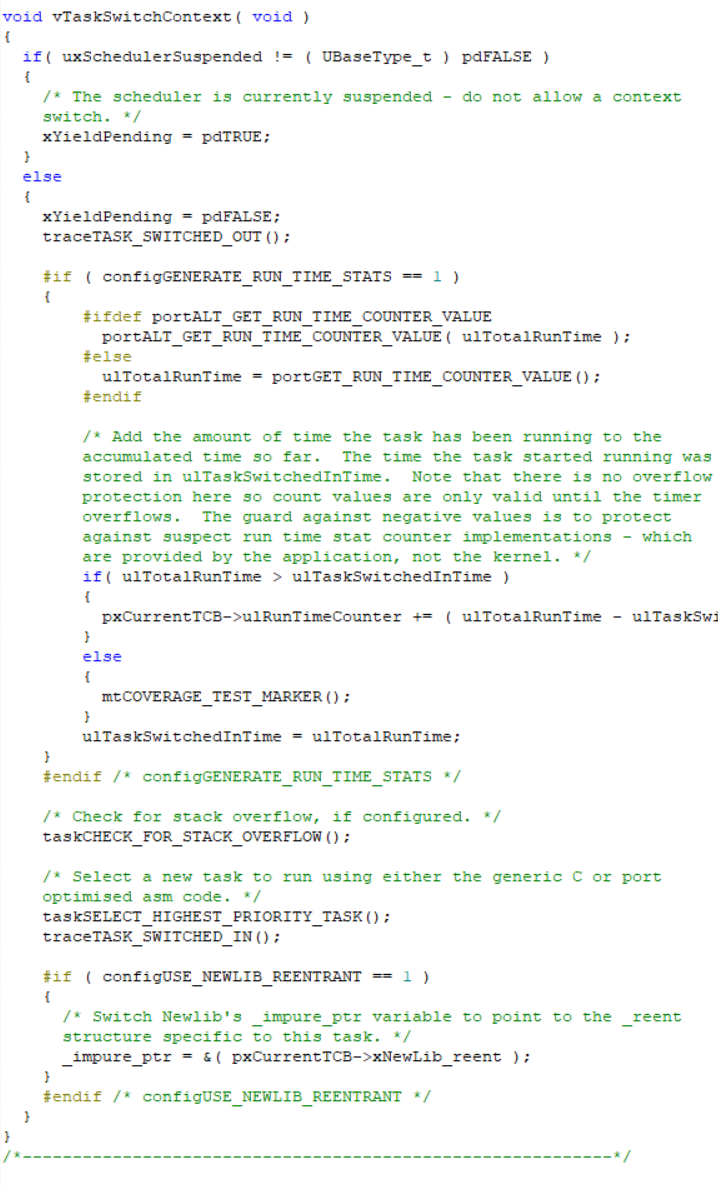




函数在其中悬起pendsv的中断，意味着函数会进入pendsv中断。现在看到pendsv中断中的代码。



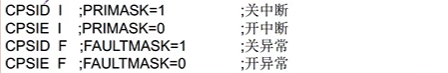
接着会进入任务切换函数

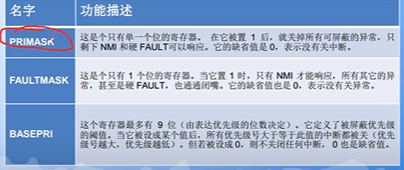


接着会切换任务正常运行。

**什么是临界段？**

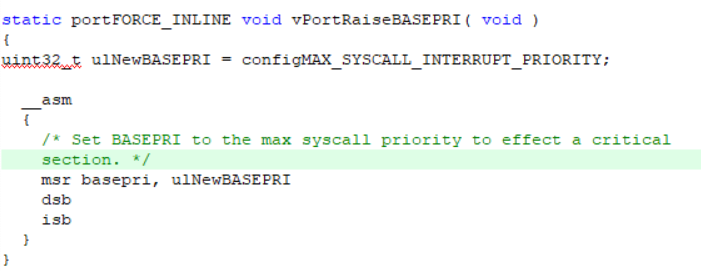
临界段用一句话概括就是一段在执行的时候不能被中断的代码段。什么情况下临界段会被打断？

一个是系统调度，还有一个就是外部中断。因此FreeRTos对临界段的保护最终还是回到对中断的开和管的控制。  




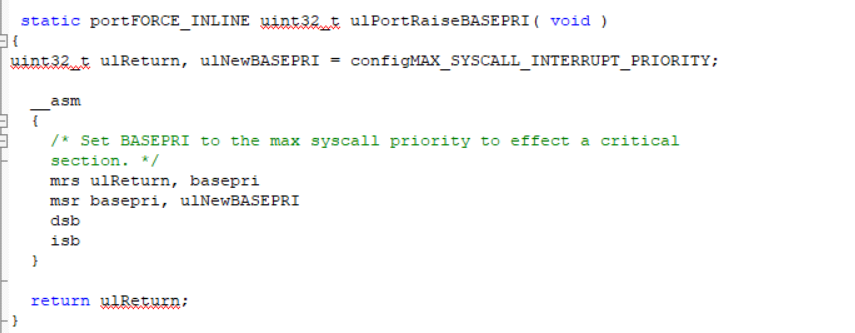
进入临界段（无中断）：

此函数不支持嵌套



进入临界段（中断中）：

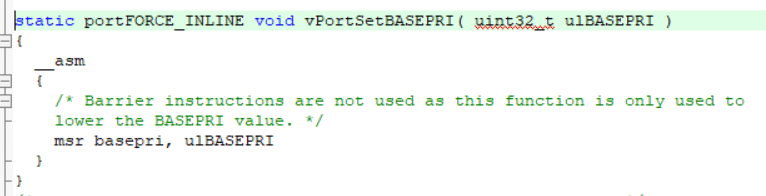
支持嵌套，有返回值



退出临界段（无中断）：

设置为打开一切中断

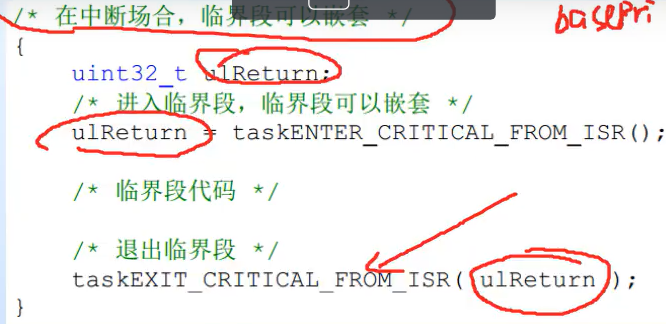




退出临界段（有中断）：



总结：在中断中临界段的用法如下所示：



而在非中断中，临界段不能嵌套。

空闲任务与阻塞延时实现:

上一章中，在任务题体内延时使用的是软件延时，即还是让CPU空等拉达到延时的效果，而使用RTOS的很大优势就是榨干CPU的性能，永远不能让塔闲着。

RTOS中的延时叫阻塞延时，即任务需要延时的时候，任务会放弃CPU的使用权，CPU可以去干其他的事情，当任务延时时间到，重新获取CPU使用权，任务继续运行，这样就重新利用了CPU的资源。如果没有其他任务运行cpu就运行空闲任务。启动调动器会创建空闲任务（空闲任务不允许阻塞）。

vTaskDelay()

阻塞延时的阻塞是指任务调用该函数时，任务会被剥夺CPU使用权，然后进入阻塞状态，直到延时结束，任务重新获取CPU使用权才可以继续运行。需要在任务控制块中添加成员变量xTicksToDelay