# freertos内核分析（结合视频看）

拟制：叶大鹏

审核：

文档部分内容为网络上截取，如有侵权，请联系删除

修订记录

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本** | **修订说明** | **修订人** |
| 2024.11.20 | V1.0 | 第一稿完成 | 叶大鹏 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目录

# 为什么使用RTOS

### 裸机程序框架

/\*主函数\*/

int main()

{

init();//一些初始化

/\*死循环\*/

while(1)

{

do\_something\_1();//执行一些逻辑

do\_something\_2();

}

}

/\*中断服务函数\*/

IRQ\_Handler()

{

set\_flag();//简短的标记操作

}

### RTOS程序框架

/\*主函数\*/

int main()

{

init();//一些初始化

xTaskCreate(task1);

xTaskCreate(task2);

vTaskStartScheduler(); //启动调度器

}

/\*子任务1（死循环）\*/

void task1()

{

while(1)

{

do\_something\_1();//执行一些逻辑(如采集传感器信息)

vTaskDelay();

}

}

/\*子任务2（死循环）\*/

void task2()

{

while(1)

{

do\_something\_2();//执行一些逻辑(如执行电机运动)

vTaskDelay();

}

}

/\*中断服务函数\*/

IRQ\_Handler()

{

set\_event();//触发事件、信号量等

}

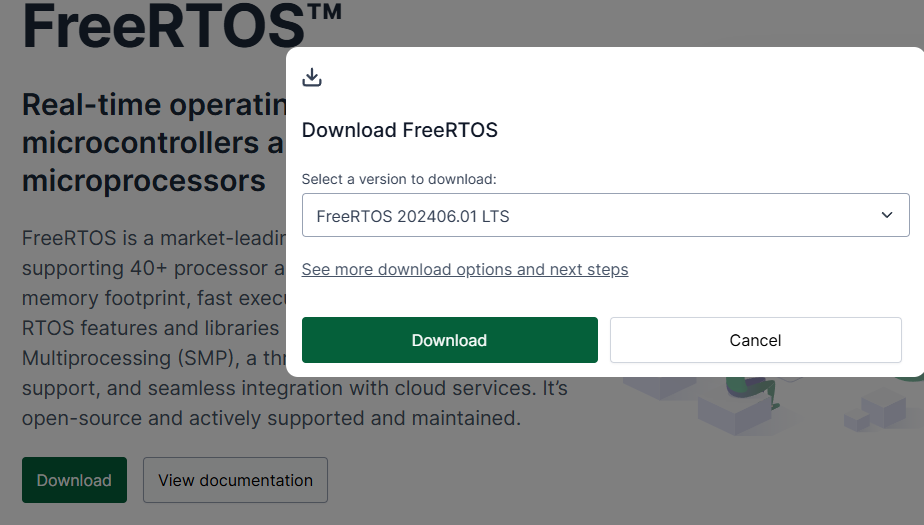
### 总结

单片机引入RTOS，目的还是为了提高单片机的实时性，使用时可以将各个功能模块分别设计为单独的任务，每个任务都是一个死循环，就相当于有多个main函数，达到并行的效果，这样CPU的利用率就提高了。

单片机只有一个CPU（核），那怎么让多个人同时干活呢？其实每个子任务虽然都是死循环，但并不是每个子任务一直都在执行，每个子任务在执行期间，可能需要延时，也可能需要等另一个任务的数据到来，所以在某个任务在等待的时候，CPU就可以停止此任务，然后切换到其它任务执行，这样看起来就是多个人在同时干活了。

### 移植

1. 下载freertos代码，网址https://www.freertos.org/，下载带LTS后缀的，表示长期支持的版本。



1. 复制内核源代码目录下的相关文件到代码工程目录下，我们可以基于裸机模板工程。





对portable目录下的文件进行删减，保留如下需要的目录：



其中：

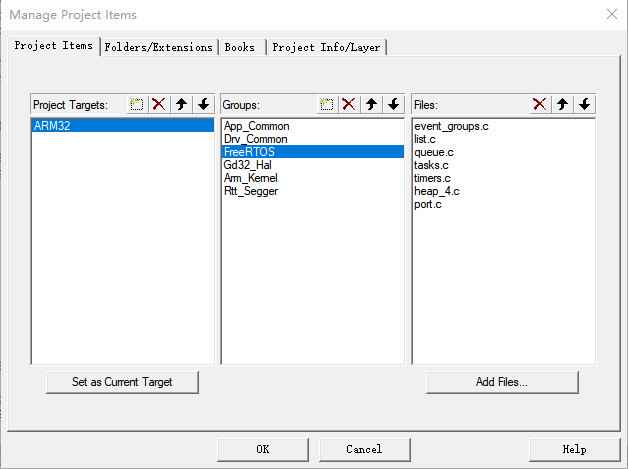
MemMang：内存管理

RVDS：CPU内核相关

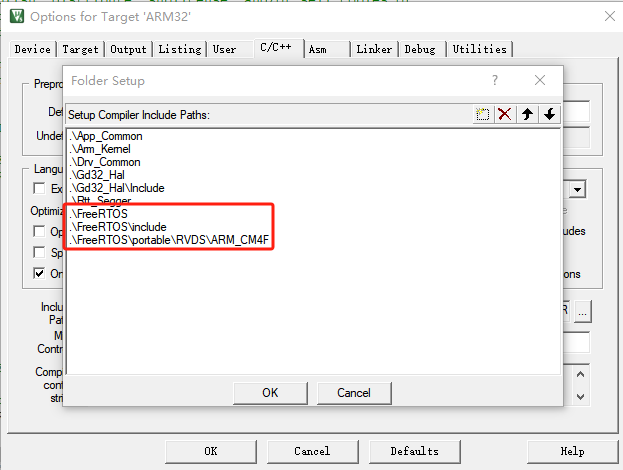
继续对RVDS目录裁剪，由于GD32F303是CM4F内核，所以 RVDS目录下保留如下目录：



1. 添加C文件到代码工程，其中heap\_4.c是位于MemMang目录下，port.c在ARM\_CM4F下。



添加内核头文件引用路径，内核相关头文件存在多个目录下，需要添加多项：



1. 添加FreeRTOSConfig.h内核配置文件，关于这些配置项的详细解释，可以到官网查询：

<https://www.freertos.org/zh-cn-cmn-s/Documentation/02-Kernel/03-Supported-devices/02-Customization>



可以放在这里，方便查找：

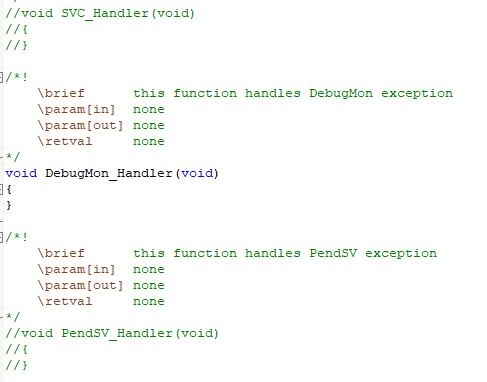


1. 配置[任务调度](https://so.csdn.net/so/search?q=%E4%BB%BB%E5%8A%A1%E8%B0%83%E5%BA%A6&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/QQ1452008/article/details/_blank)相关的中断处理函数

在FreeRTOSConfig.h中对PendSV和SVC服务函数进行宏定义：

1732178964775

记得要删除gd32f30x\_it.c中这两个服务函数和systick服务函数的代码



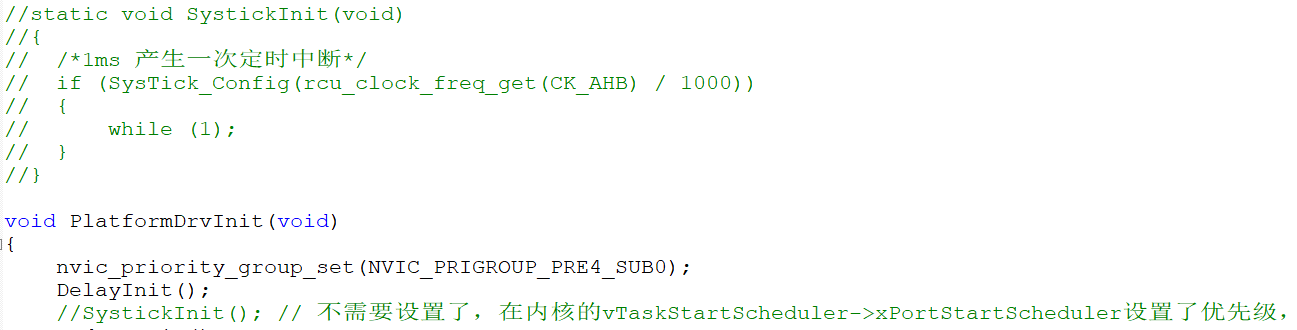
vPortSVCHandler和xPortPendSVHandler这两个函数的具体实现都在ARM\_CM4F\port.c中。

再添加对systick中断服务函数的修改，在ARM\_CM4F\port.c中添加如下代码（如果其它文件中有systick中断服务函数，记得删除）：



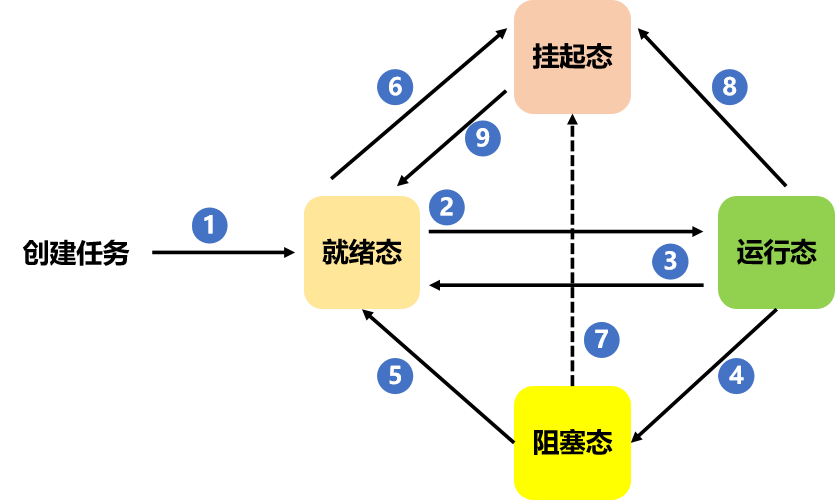
为什么不像上面两个函数进行宏定义，是为了防止进入该中断时，rtos内核还没有开启调度就去执行相关程序。

1. 配置优先级分组。



# 任务调度与切换

### 任务状态



/\* 任务状态枚举. \*/

typedef enum

{

eRunning = 0, /\* 运行态 \*/

eReady, /\* 已读或临时挂起的就绪列表中\*/

eBlocked, /\* 阻塞状态 \*/

eSuspended, /\* 挂起或无线阻塞状态\*/

eDeleted, /\* 正在查询的任务已被删除，但TCB尚未释放\*/

eInvalid /\* 无效状态\*/

} eTaskState;

1. 创建任务→就绪态：

任务创建完成后进入就绪态，表明任务已准备就绪，随时可以运行，只等待调度器进行调度。

2、就绪态→运行态：

发生任务切换时，就绪列表中最高优先级的任务被执行，从而进入运行态。

3、运行态→就绪态：

有更高优先级任务创建或者恢复后，会发生任务调度，此刻就绪列表中最高优先级任务变为运行态，那么原先运行的任务由运行态变为就绪态，依然在就绪列表中，等待最高优先级的任务运行完毕继续运行原来的任务。

1. 运行态→阻塞态：

正在运行的任务发生阻塞（挂起、延时、读信号量等待）时，该任务会从就绪列表中删除，任务状态由运行态变成阻塞态，然后发生任务切换，运行就绪列表中当前最高优先级任务。

1. 阻塞态→就绪态：

阻塞的任务被恢复后（任务恢复、延时时间超时、读信号量超时或读到信号量等），此时被恢复的任务会被加入就绪列表，从而由阻塞态变成就绪态；如果此时被恢复任务的优先级高于正在运行任务的优先级，则会发生任务切换，将该任务将再次转换任务状态，由就绪态变成运行态。

6、7、8 就绪态、阻塞态、运行态→挂起态（Suspended）：

任务可以通过调用vTaskSuspend() API 函数都可以将处于任何状态的任务挂起，被挂起的任务得不到CPU的使用权，也不会参与调度，除非它从挂起态中解除。

9、挂起态→就绪态：

把 一 个 挂 起 状态 的 任 务 恢复的 唯 一 途 径 就 是调 用 vTaskResume() 或 vTaskResumeFromISR() API 函数，如果此时被恢复任务的优先级高于正在运行任务的优先级，则会发生任务切换，将该任务将再次转换任务状态，由就绪态变成运行态。

以上是任务运行的各种状态，看起来有点复杂，可以这样理解：

为简单起见，先不考虑挂起态，在任一时刻，CPU只能处理某一个任务，则该任务就处于运行态，对于其它任务，如果是自己想要延时或阻塞等待时，则处于阻塞态，如果是自己想要执行但因为优先级低而不能执行时，则处于就绪态。

然后，以上状态如何被改变呢？

1.运行态的自己想进入阻塞态，则就绪态的任务即可运行。

2.阻塞态的解除阻塞进入就绪，若该任务的优先级更高，则可抢占当前处于运行的任务，使自己运行，使对方就绪。

有没有发现，阻塞态的任务要想运行，必须先进入就绪态，再进入运行态。

### 创建任务

BaseType\_t xTaskCreate( TaskFunction\_t pxTaskCode,

const char \* const pcName, const configSTACK\_DEPTH\_TYPE usStackDepth,

void \* const pvParameters,

UBaseType\_t uxPriority,

TaskHandle\_t \* const pxCreatedTask )

解释一下这些参数：

pxTaskCode 任务函数名

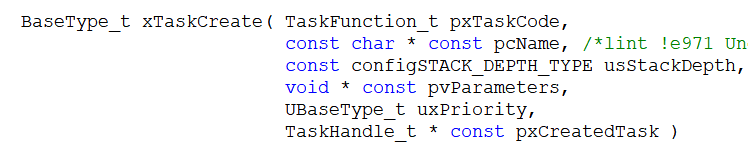
pcName 任务名字 （就是个字符串，别和参数1搞混）

usStackDepth 栈大小，4个字节为单位

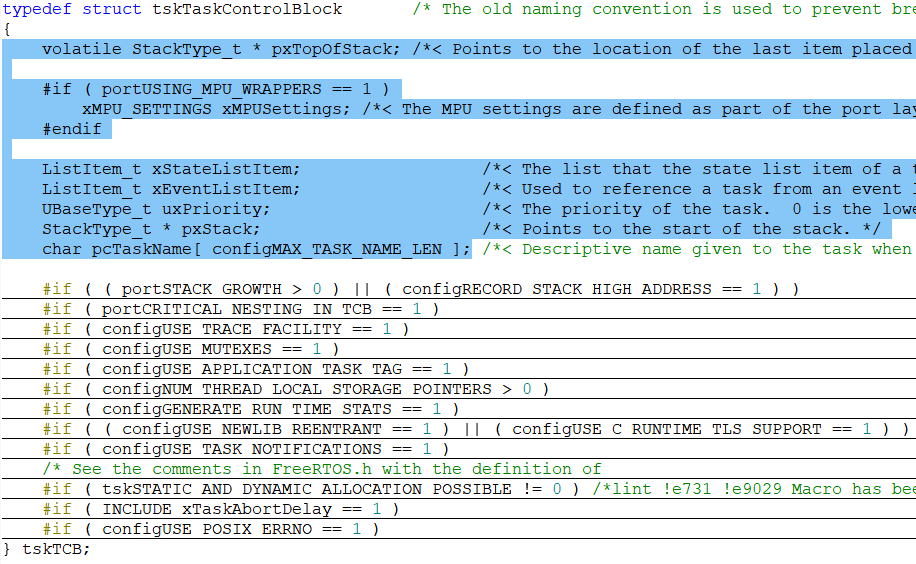
pvParameters 任务函数参数

uxPriority 任务优先级

pxCreatedTask 任务句柄（这个是传出来的，不是传进去的）



1. 定义了一个TCB （任务控制块）的指针，查看这个数据结构的主要内容：



解释一下这些成员：

pxTopOfStack 指针变量，保存对应任务的栈顶地址，随着任务运行会变化

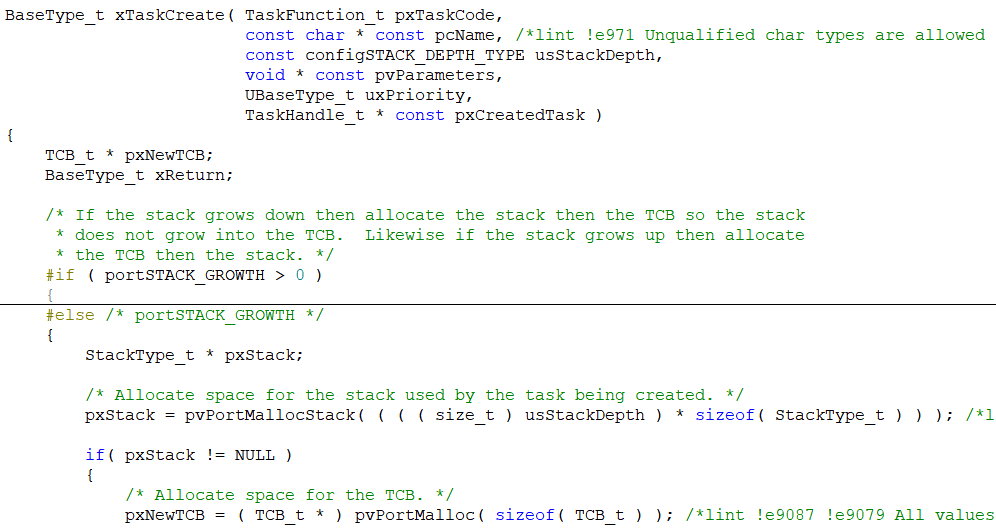
xStateListItem 状态链表节点

xEventListItem 事件链表节点

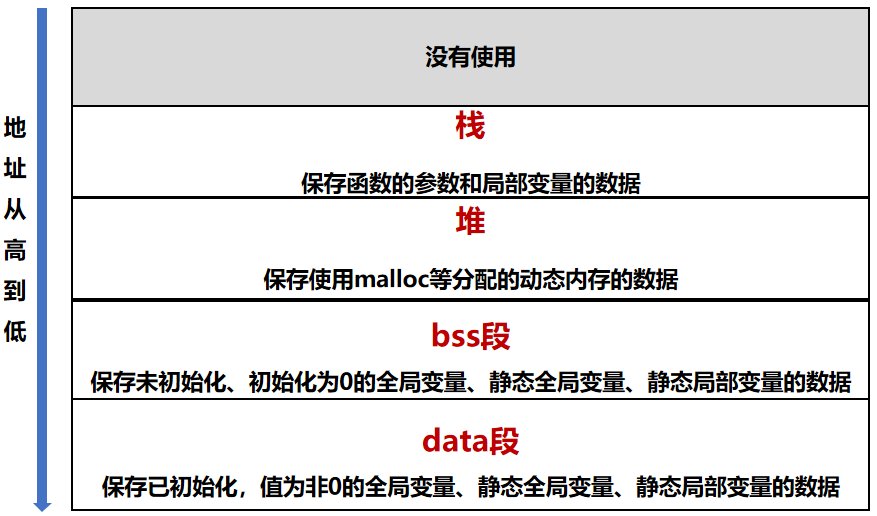
uxPriority 优先级

pxStack 保存对应任务的栈起始地址，不再变化

pcTaskName 数组，保存对应任务的名字



给任务分配栈空间，给任务的TCB分配空间，这个空间实际上是从哪分配的呢？我们在回顾下《ARM32单片机课程》中讲的知识：



这里虽然使用pvPortMalloc申请了内存空间，但是并不是上图的堆中申请的，而是在bss段，也就是全局区，具体在哪呢？在heap4.c里（课程使用的是这个代码）：

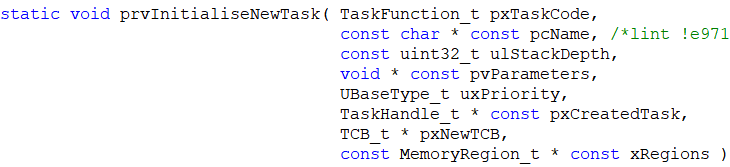
PRIVILEGED\_DATA static uint8\_t ucHeap[ configTOTAL\_HEAP\_SIZE ];

就是一个大数组，元素个数是需要在FreeRTOSConfig.h中定义的。

所以要注意，任务的栈空间不像裸机程序的函数栈空间，通过在startup\_xxxx.s中设定栈空间：

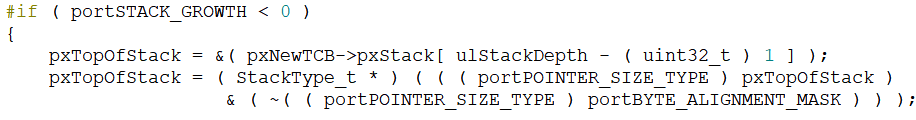
Stack\_Size EQU 0x00000C00

1. 接下来调用prvInitialiseNewTask函数：





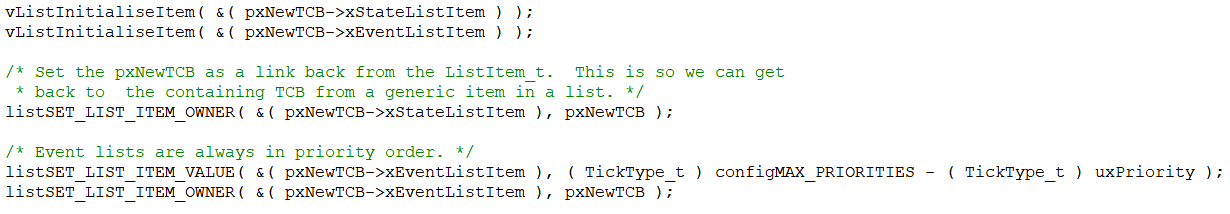
看到这堆参数，基本能猜到要给TCB赋值了，



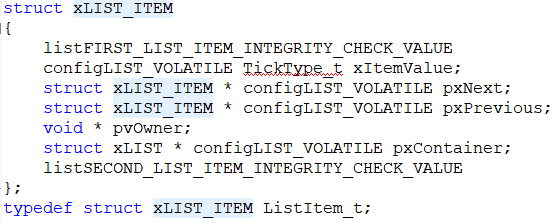
pxTopOfStack = &( pxNewTCB->pxStack[ ulStackDepth - ( uint32\_t ) 1 ] );

给成员pxTopOfStack赋值，栈是向下增长，地址从高到低，所以将动态申请到的高地址（按照数组访问的话，数组的高索引也就对应高地址）。下面那句话是地址对齐用。

然后在这个函数里继续对任务的链表节点进行初始化：

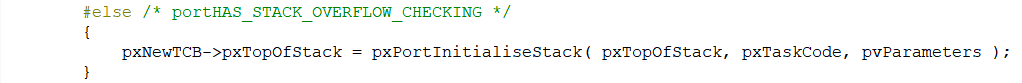


链表是一种数据结构，原理比较复杂，大家需要去学习下我们的《嵌入式C语言课程》中的内容，讲解了RTOS使用到的通用链表。

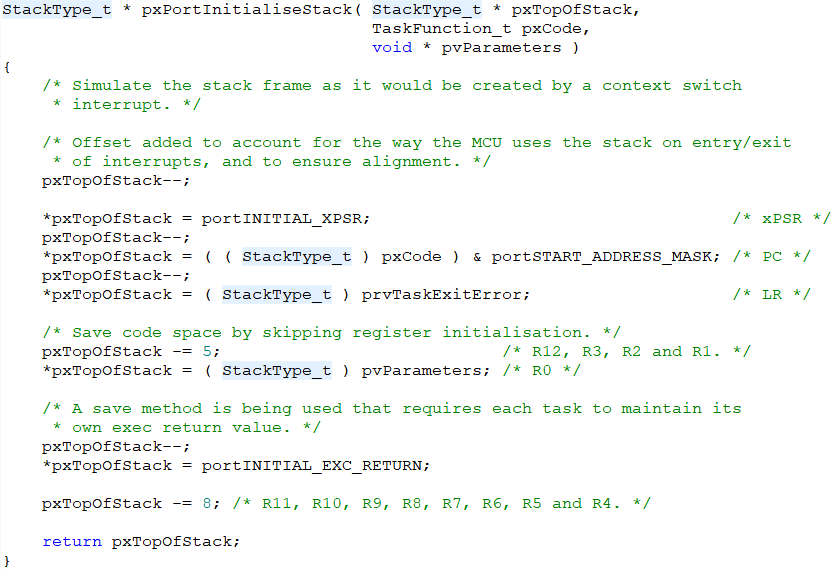


 其中：  
 pvOwner        保存链表节点持有者指针（TCB）  
 pxContainer     保存链表头指针，比如阻塞/就绪链表

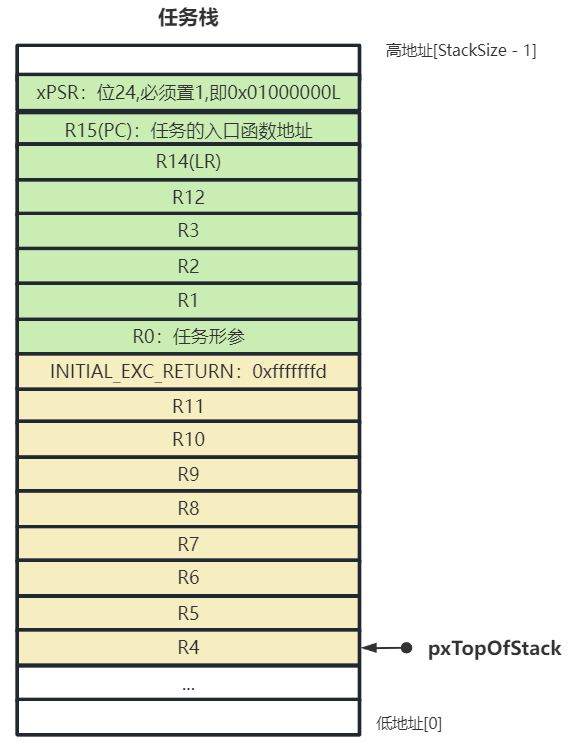
再接着调用pxPortInitialiseStack函数，对任务的栈空间进行初始化：



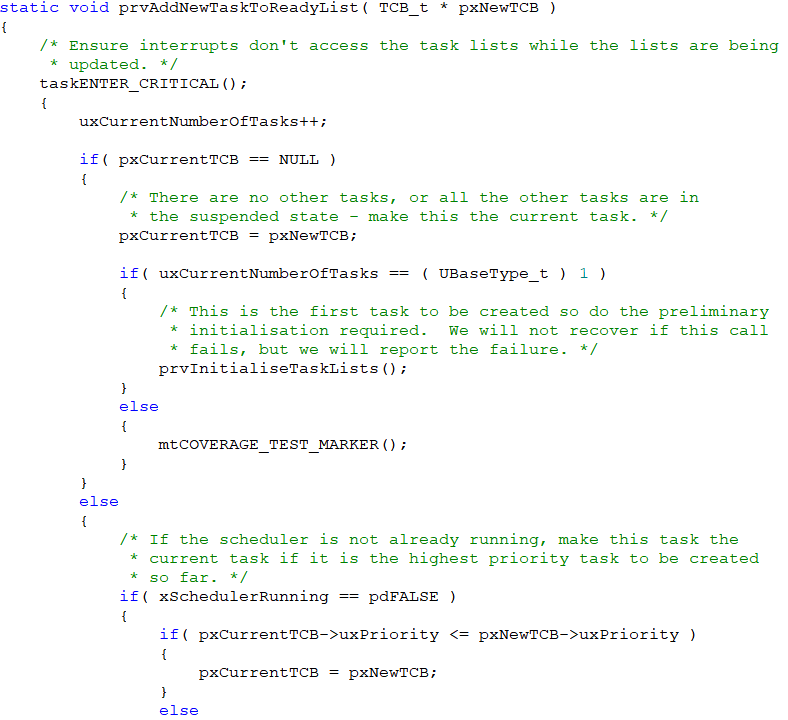
跟进去：



执行完，任务栈的布局是这样的：



1. 再回到xTaskCreate()，添加任务到就绪链表（数组）函数

 prvAddNewTaskToReadyList( pxNewTCB );  
   

当前任务数+1：

uxCurrentNumberOfTasks++;

如果没有任务在执行，将新任务设置为当前任务

pxCurrentTCB = pxNewTCB;

如果是第一个任务，还需要初始化系统的各个链表：

if( uxCurrentNumberOfTasks == ( UBaseType\_t ) 1 )

prvInitialiseTaskLists();

跳进去：

for( uxPriority = ( UBaseType\_t ) 0U; uxPriority < ( UBaseType\_t ) configMAX\_PRIORITIES; uxPriority++ )

{

vListInitialise( &( pxReadyTasksLists[ uxPriority ] ) );

}

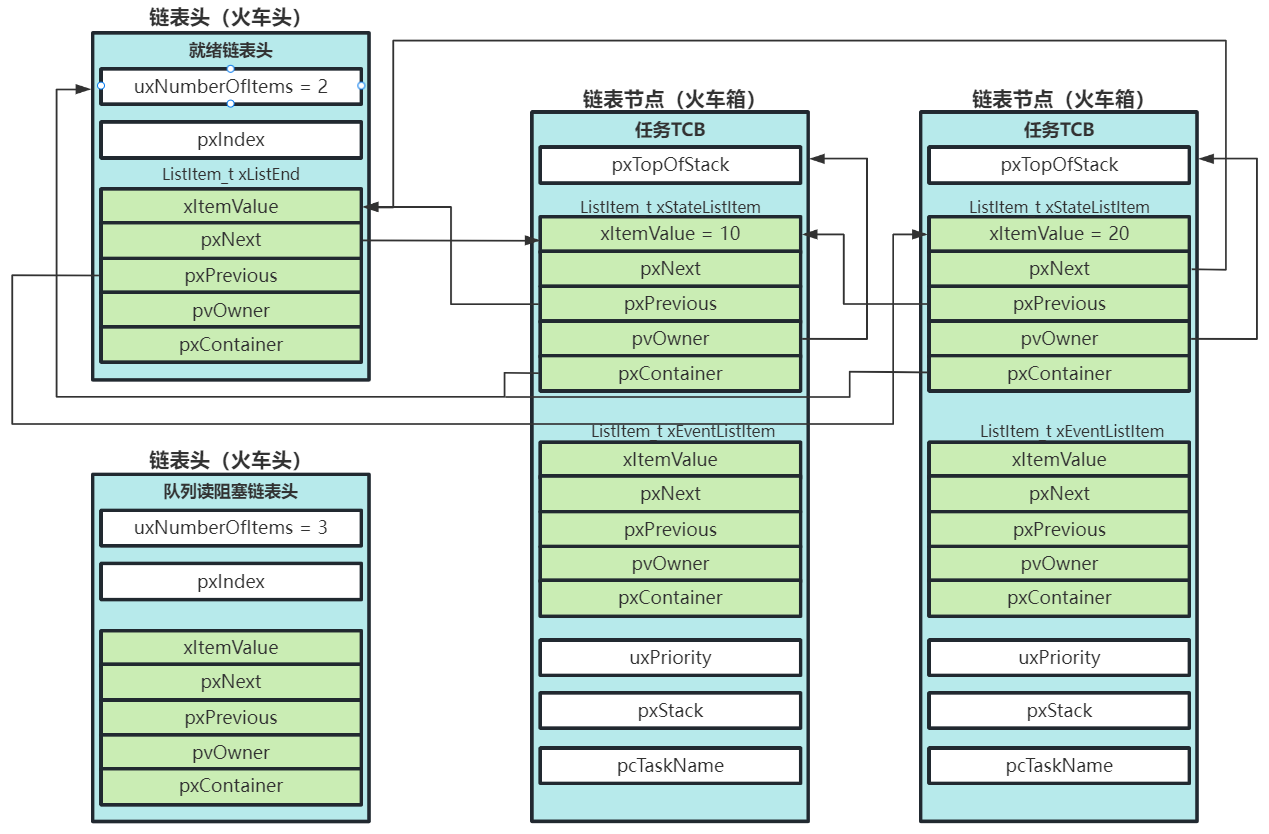
vListInitialise( &xDelayedTaskList1 );

vListInitialise( &xDelayedTaskList2 );

vListInitialise( &xPendingReadyList );

解释一下任务就绪链表，就绪链表有多个，数量就是优先级数量，每一个优先级对应有一个就绪链表：

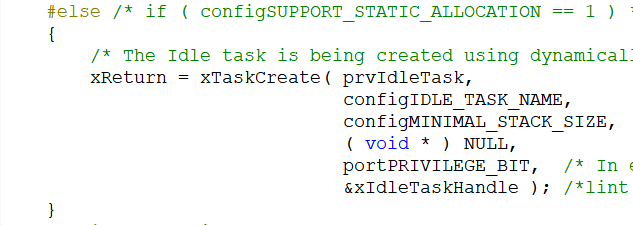




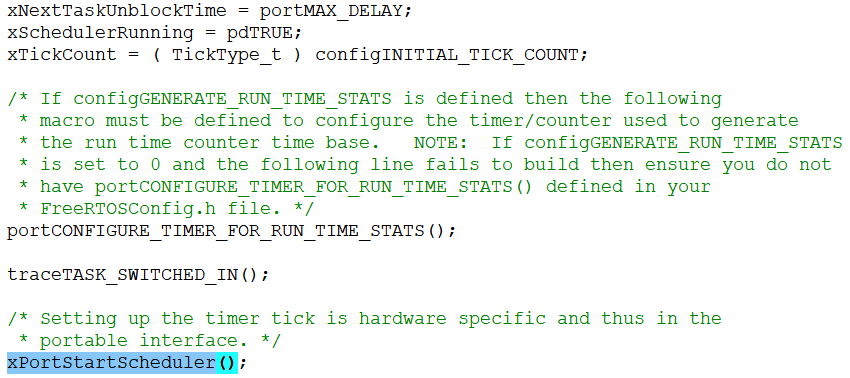
### 启动第一个任务

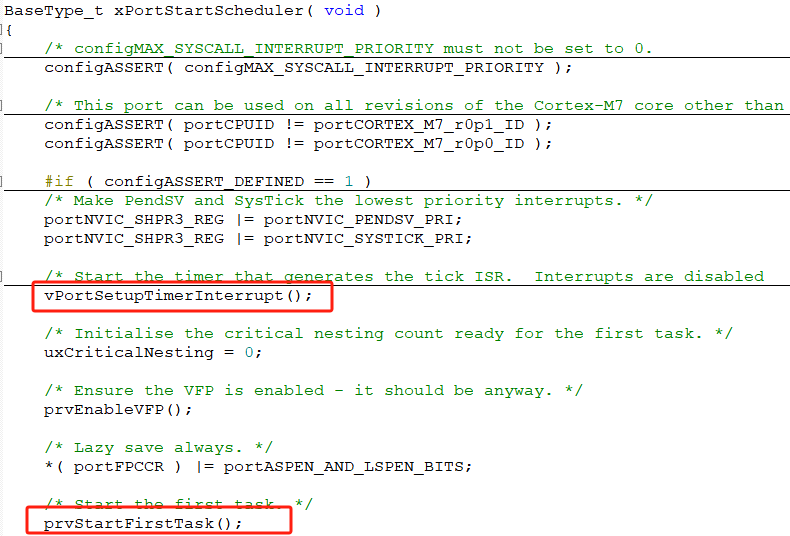
建立好任务之后，就调用vTaskStartScheduler(); 开启任务调度了。

1. 首先建立一个空闲任务，空闲任务优先级为0，cpu空闲时调用（无其他高优先级任务运行）。

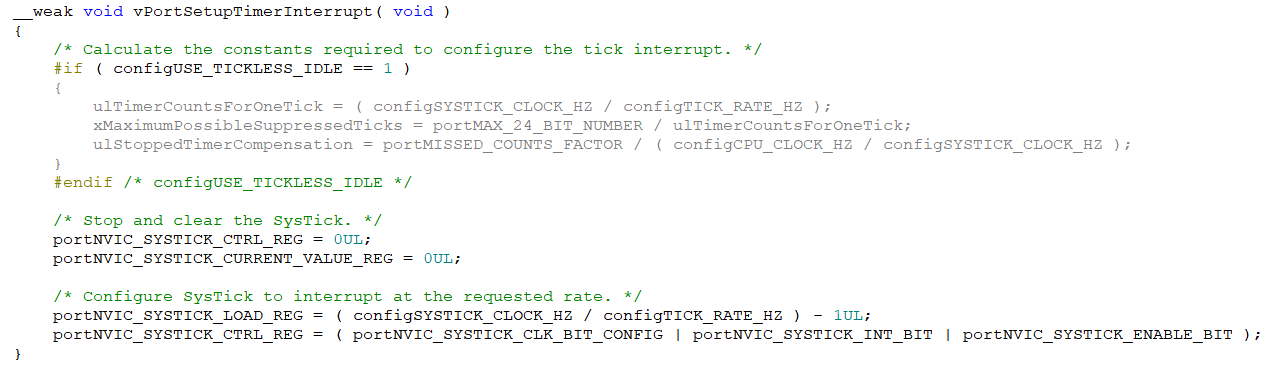


1. 初始化一些变量，开启任务调度标志，Tick计数置0，默认1Tick是1ms，RTOS的调度离不开这个Tick计数器。

  
3. 再进入xPortStartScheduler()：



进入vPortSetupTimerInterrupt()：



portNVIC\_SYSTICK\_CLK\_BIT：

表示设置STK\_CTRL寄存器的bit2(CLKSOURCE),CLKSOURCE=1使用AHB时钟

portNVIC\_SYSTICK\_INT\_BIT：

表示使能systick中断

portNVIC\_SYSTICK\_ENABLE\_BIT：

表示使能systick计数

大家可能关心systick的优先级，是在xPortStartScheduler()函数中设置的：

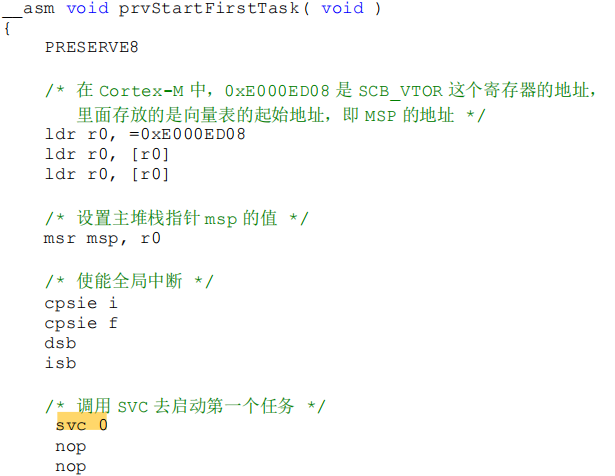
1733195805692

#define portNVIC\_SHPR3\_REG ( \*( ( volatile uint32\_t \* ) 0xe000ed20 ) )

每个中断有一个对应的优先级寄存器，每个寄存器占8位（高4位使用，低4位未使用，高4位包含抢占优先级和子优先级，按照设定，freertos不使用子优先级），4个相邻优先级寄存器组成一个32位寄存器，因此地址0xE000\_ED20~0xE000\_ED23这四个寄存器可拼成一个地址为0xE000ED20的32位寄存器。FreeRTOS在设置PendSV和SysTick的中断优先级时是直接操作地址0xE000\_ED20。



再进入prvStartFirstTask()：



通过svc 0这条指令，触发SVC异常，进入异常服务函数vPortSVCHandler()，在头文件FreeRTOSConfig.h中，有它的宏定义：

#define vPortSVCHandler SVC\_Handler

\_\_asm void vPortSVCHandler( void )

{

PRESERVE8 /\* 8字节对齐\*/

ldr r3, =pxCurrentTCB /\* 加载R3=&pxCurrentTCB \*/

ldr r1, [r3] /\* 加载R1=pxCurrentTCB \*/

ldr r0, [r1] /\* 加载R0=pxCurrentTCB->pxTopOfStack\*/

ldmia r0!, {r4-r11,r14} /\* 将R0用作栈指针,从pxCurrentTCB栈中恢复R4-R11\*/

msr psp, r0 /\* 恢复psp=R0,指向当前任务栈顶R0存储位置\*/

isb /\* 指令同步排序，与流水线相关\*/

mov r0, #0

msr basepri, r0 /\* 停止屏蔽任何中断.\*/

bx r14 /\* 执行bx r14,将切换至用户级线程模式,使用PSP栈指针 自动恢复R0-R3、R12、R14、R15、xPSR，并且PSP指向了 栈底位置\*/

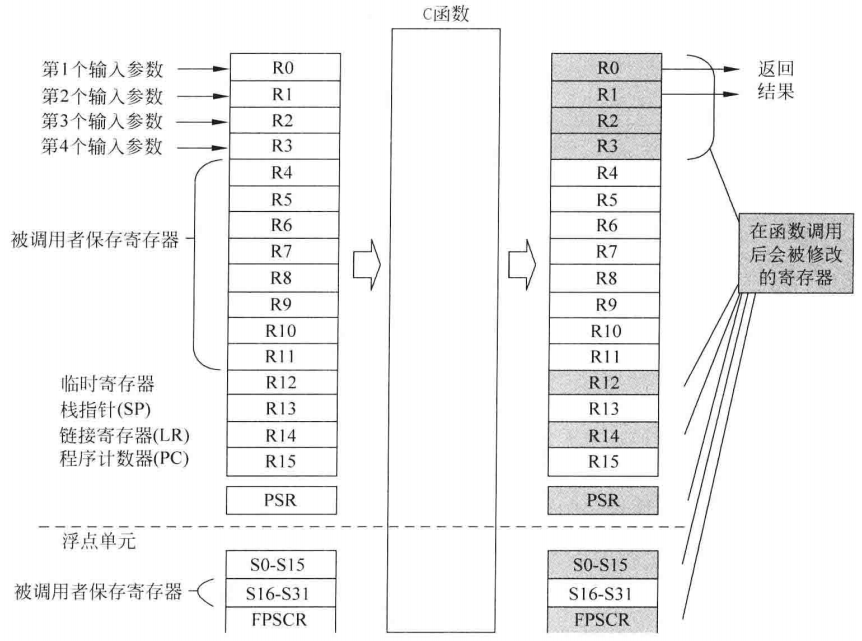
# 1732160255306

这样通过bx r14这条指令，就进入到任务的入口函数开始执行了。

在Cortex-M处理器中有两个栈指针，一个是主栈指针(Main Stack Pointer，即MSP)；另一个是线程栈指针(Processor Stack Pointer，即PSP)。在任何时刻只能使用到其中一个。

复位后，默认使用MSP，在FreeRTOS中，MSP用于OS内核（比如创建任务队列代码）和异常中断处理（比如pendsv/systick/UART），PSP用于应用任务程序（从task入口函数进入执行的程序）。

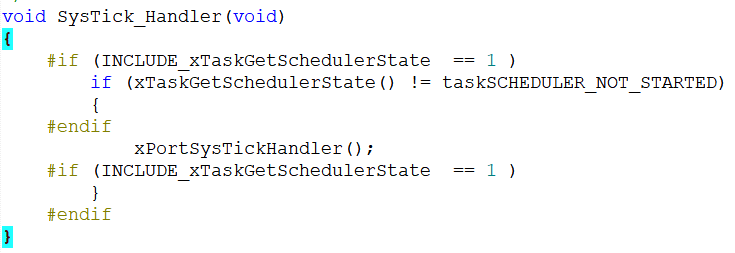
通过设置CPU的CONTROL寄存器的bit[1]选择使用哪个栈指针。CONTROL[1]=0选择主栈指针；CONTROL[1]=1选择线程栈指针。

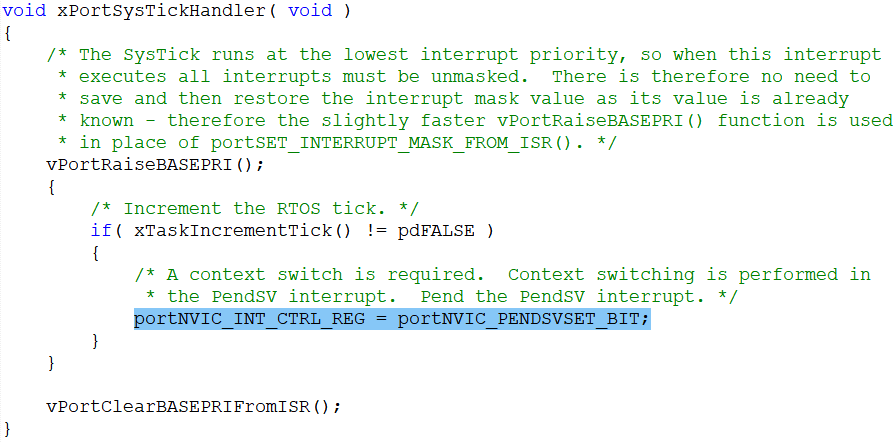


### 任务切换

freertos有两种场景可以触发任务切换，一个是systick中断触发切换（）；另外一个是在普通程序中使用taskYIELD()触发任务切换(比如vTaskDelay函数中，队列发送/接收的内核函数中)，而在中断服务程序中使用portYIELD\_FROM\_ISR()触发任务切换。

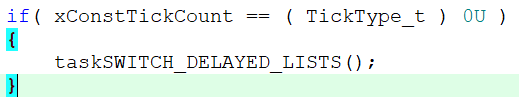
1. systick中断触发



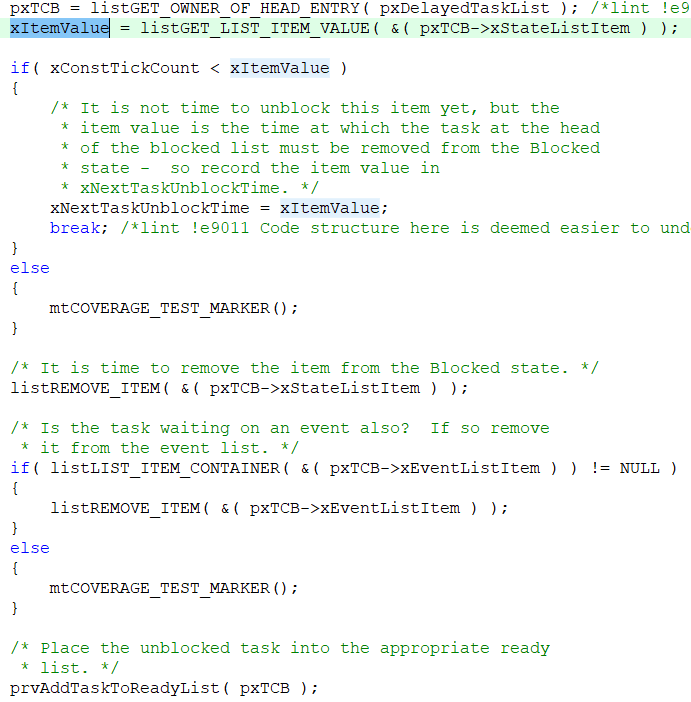


在xTaskIncrementTick()中，主要完成 ：

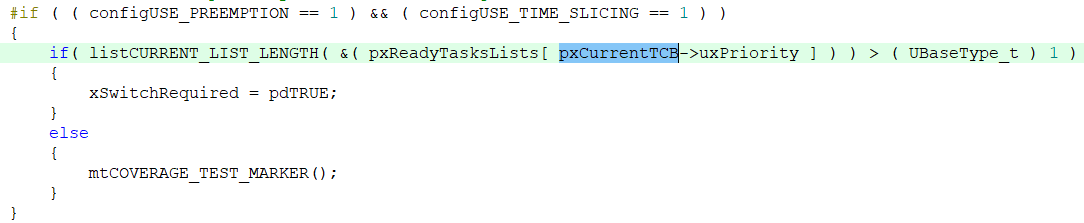
1）判断节拍计数器xTickCount 是否溢出， 溢出轮换延时阻塞链表



1. 判断是否有延时阻塞任务超时，取出插入就绪链表



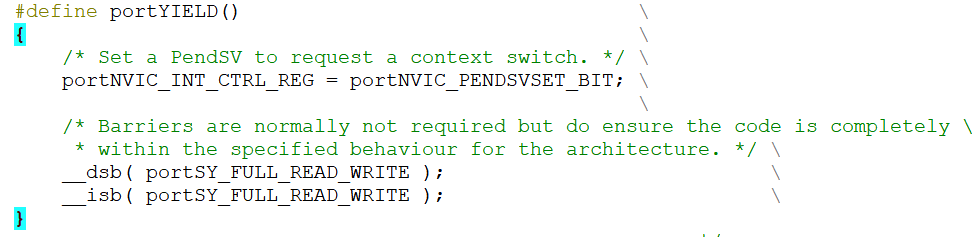
1. 判断是否有同优先级任务在就绪链表中需要轮转



1. 用taskYIELD()触发

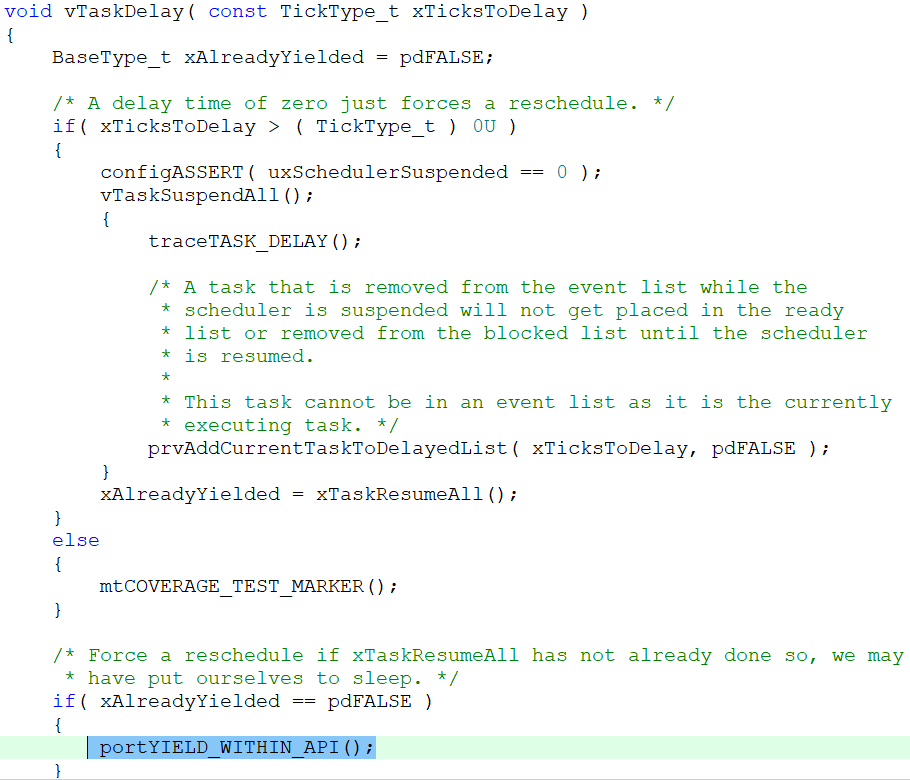
#define taskYIELD() portYIELD()

#define portYIELD\_WITHIN\_API portYIELD



所以，两种场景，殊途同归，最终还是通过向中断控制及状态寄存器ICSR(地址：0xE000 ED04)的第28位写入1，触发PendSV异常，进入服务函数xPortPendSVHandler，从而执行任务切换。

比如在vTaskDelay()函数中：



\_\_asm void xPortPendSVHandler( void )

{

extern uxCriticalNesting;

extern pxCurrentTCB;

extern vTaskSwitchContext;

PRESERVE8

mrs r0, psp /\* 【1.1：环境保存】PSP->R0\*/

isb /\* 【1.2：环境保存】指令同步排序，与流水线相关\*/

ldr r3, =pxCurrentTCB /\* 【1.3：环境保存】加载R3=&pxCurrentTCB\*/

ldr r2, [r3] /\* 【1.4：环境保存】加载R2=pxCurrentTCB \*/

stmdb r0!, {r4-r11, r14} /\* 【1.5：环境保存】R0用作栈指针,手动入栈R4-R11 R14 至切换前的任务栈.\*/

str r0, [r2] /\* 【1.6：环境保存】保存栈顶 pxCurrentTCB->pxTopOfStack=R0\*/

stmdb sp!, {r3, r14} /\* 【2.1：任务切换】入主栈r3-r14 \*/

mov r0, #configMAX\_SYSCALL\_INTERRUPT\_PRIORITY

msr basepri, r0 /\* 【2.2：任务切换】屏蔽部分中断\*/

dsb /\* 【2.3：任务切换】数据同步隔离\*/

isb /\* 【2.4：任务切换】指令同步排序\*/

bl vTaskSwitchContext /\* 【2.5：任务切换】跳转至函数vTaskSwitchContext 执行\*/

mov r0, #0 /\* 【2.6：任务切换】清空R0\*/

msr basepri, r0 /\* 【2.7：任务切换】取消中断屏蔽\*/

ldmia sp!, {r3, r14} /\* 【2.8：任务切换】出主栈 R3-R14\*/

ldr r1, [r3] /\* 【3.1：环境恢复】加载R1=pxCurrentTCB\*/

ldr r0, [r1] /\* 【3.2：环境恢复】加载 R1=pxCurrentTCB->pxTopOfStack. \*/

ldmia r0!, {r4-r11, r14} /\* 【3.3：环境恢复】R0用作栈指针,手动从出栈切换 后的任务出栈R4-R11 R14\*/

msr psp, r0 /\* 【3.4：环境恢复】恢复psp=R0. \*/

isb /\* 【3.5：环境恢复】指令同步排序\*/

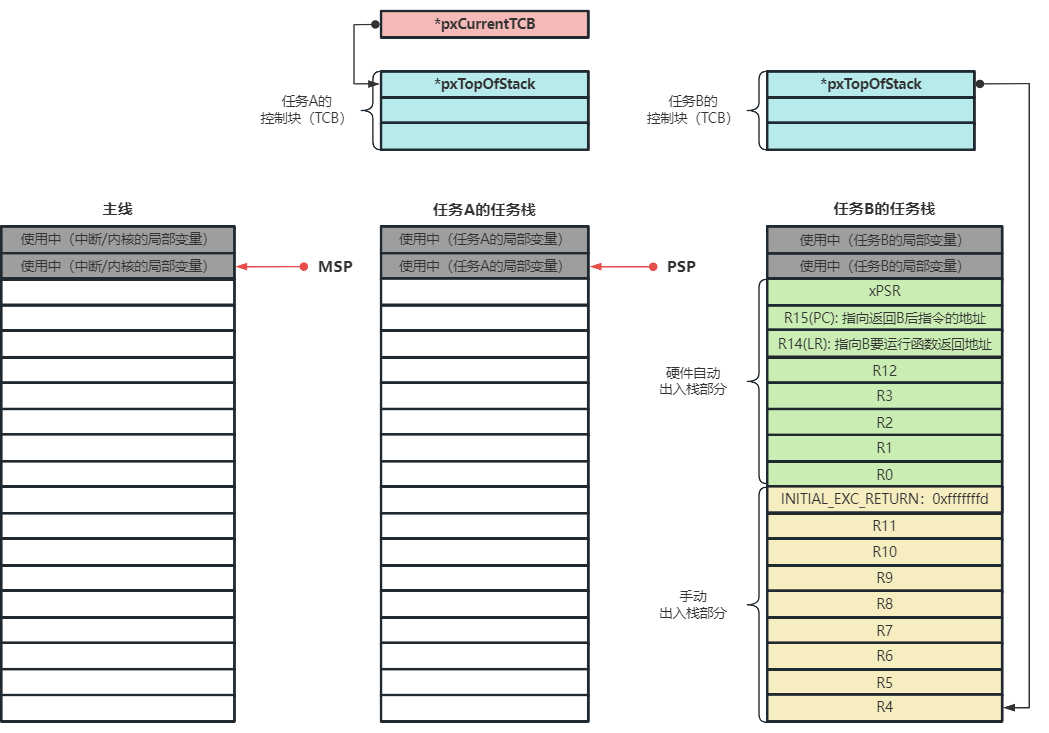
bx r14 /\* 【3.6：环境恢复】R14=0XFFFFFFFDL，将切换至用 户级线程模式, 使PSP栈指针自动恢复R0-R3、R12、 R14、R15、xPSR，并且PSP指向了栈底位置\*/

nop

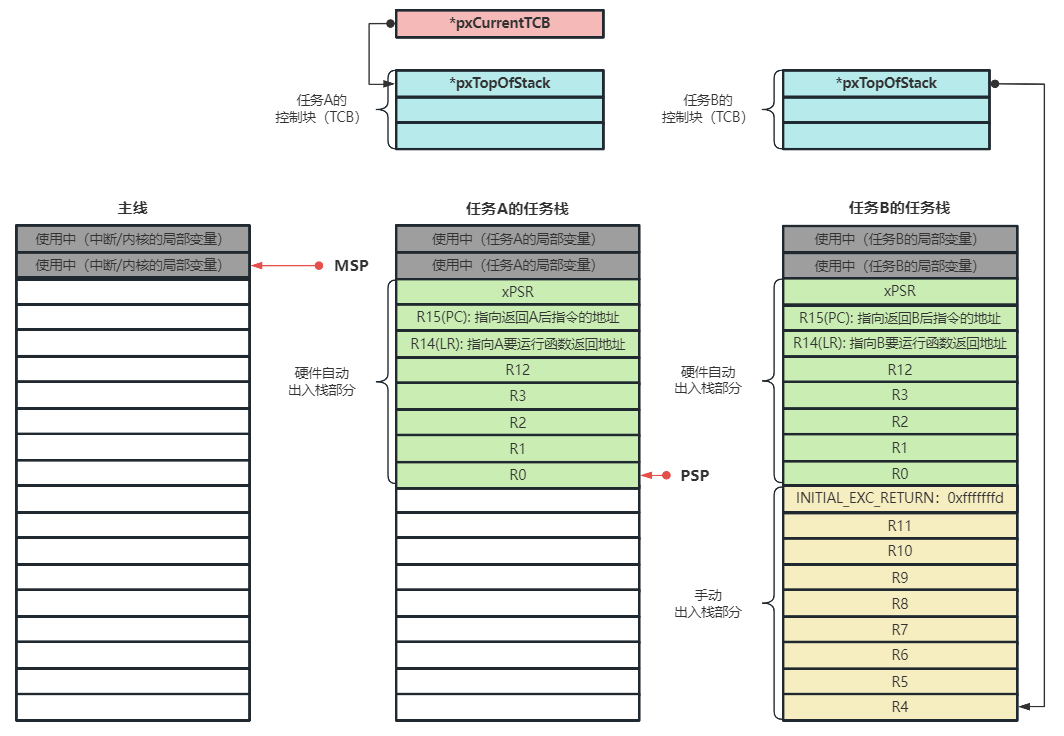
}

上述过程，可以分为3部分，1.任务A由运行态切换到了阻塞态，需要保存A的现场到A的内存栈；2.然后需找下个要运行的任务，为任务切换做准备；3.如果是任务B，则要恢复任务B的现场，A和B切换过程如下所示：

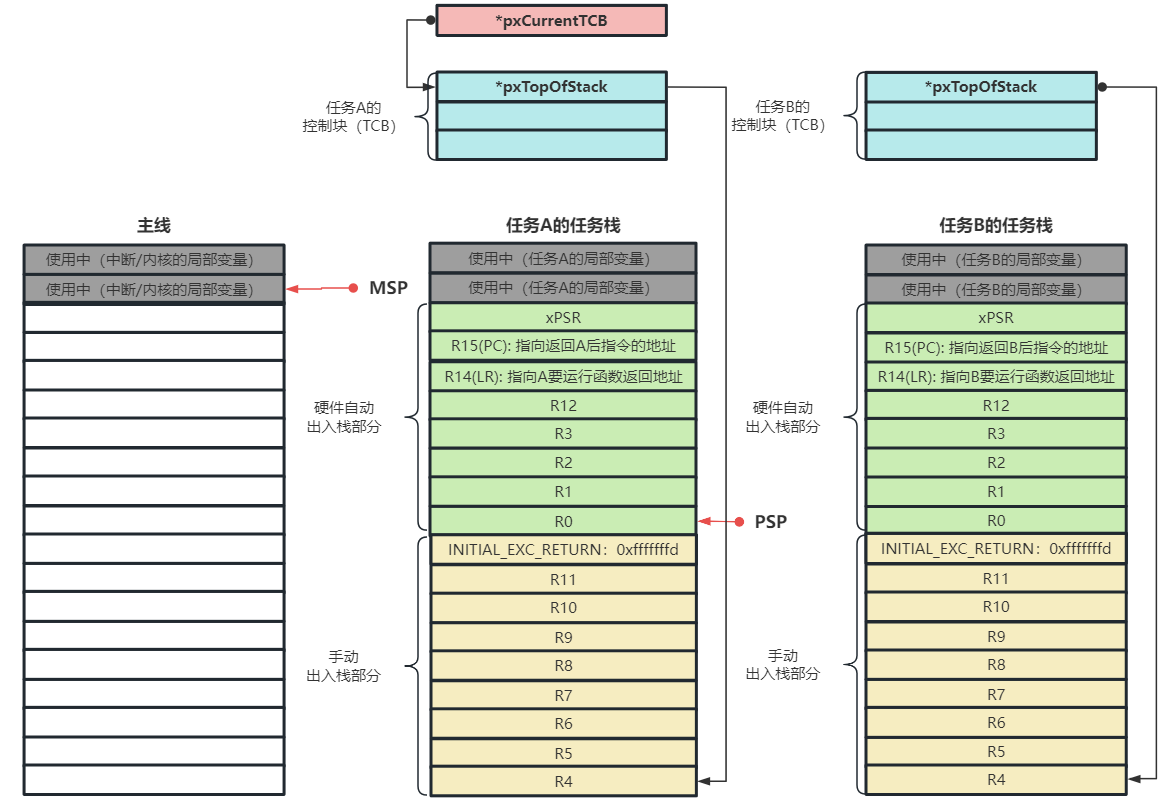
* 在A任务正常运行期间，各个任务的情况：



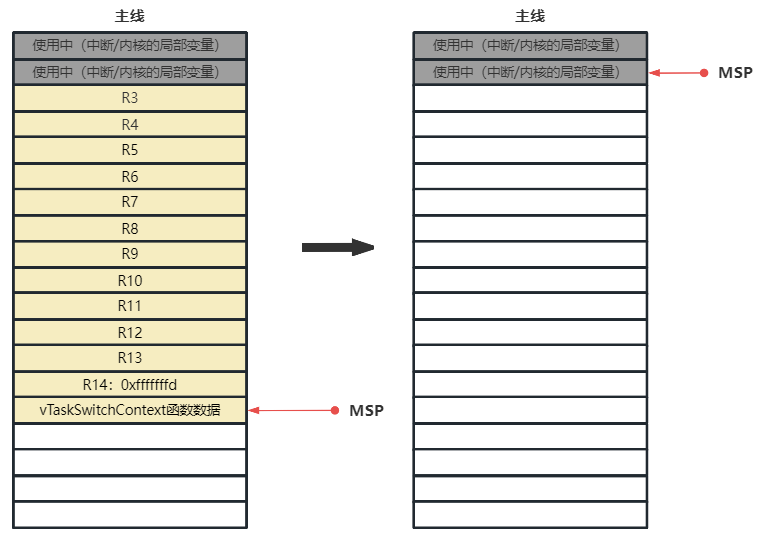
* 进入xPortPendSVHandler()时刻前，因为要进入到异常，CPU自动保存现场，将部分寄存器数据压入到A任务的内存栈中，使用的是PSP。记住，进入异常服务函数，SP对应就变成MSP了：



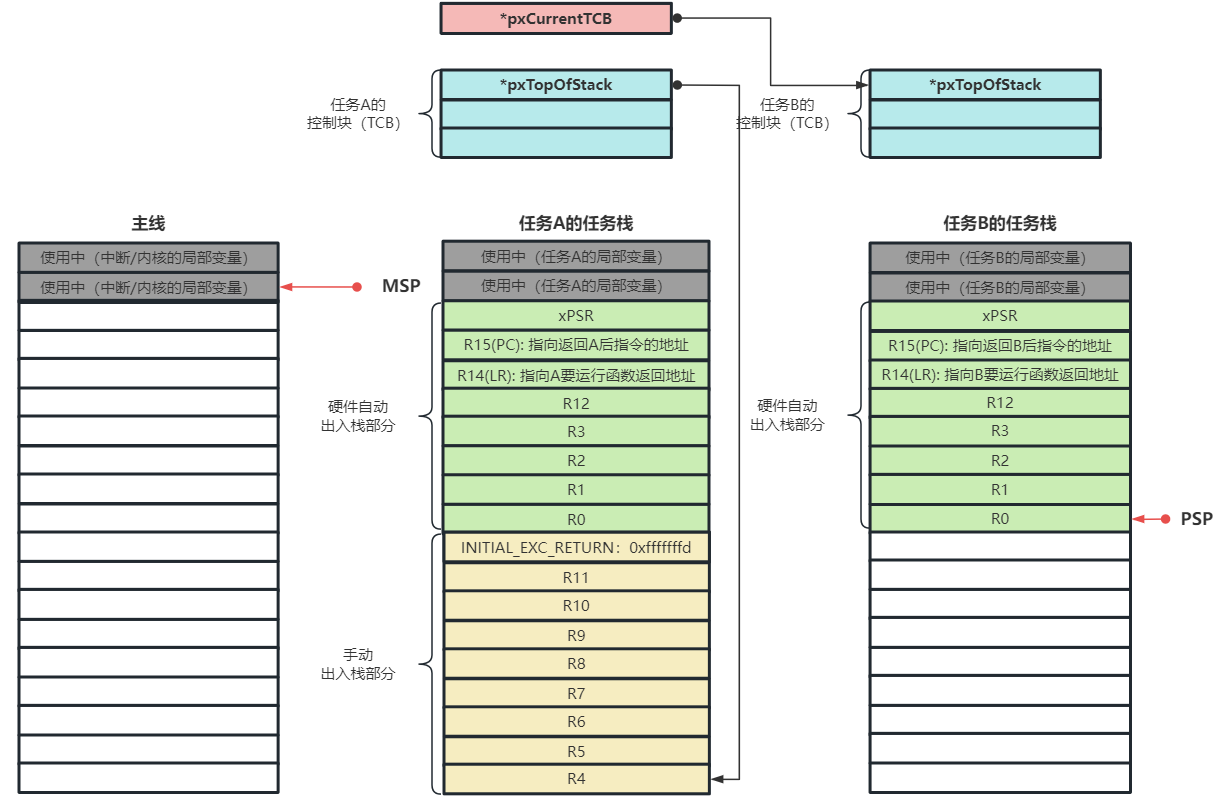
* 进入到异常函数，CPU会使用MSP，再手动将部分寄存器数据压入到A任务的内存栈中，mrs r0, psp，也就是接下来通过R0作为栈指针，将R4-R11R14入栈，再保存栈顶地址到TCB的pxTopOfStack中：



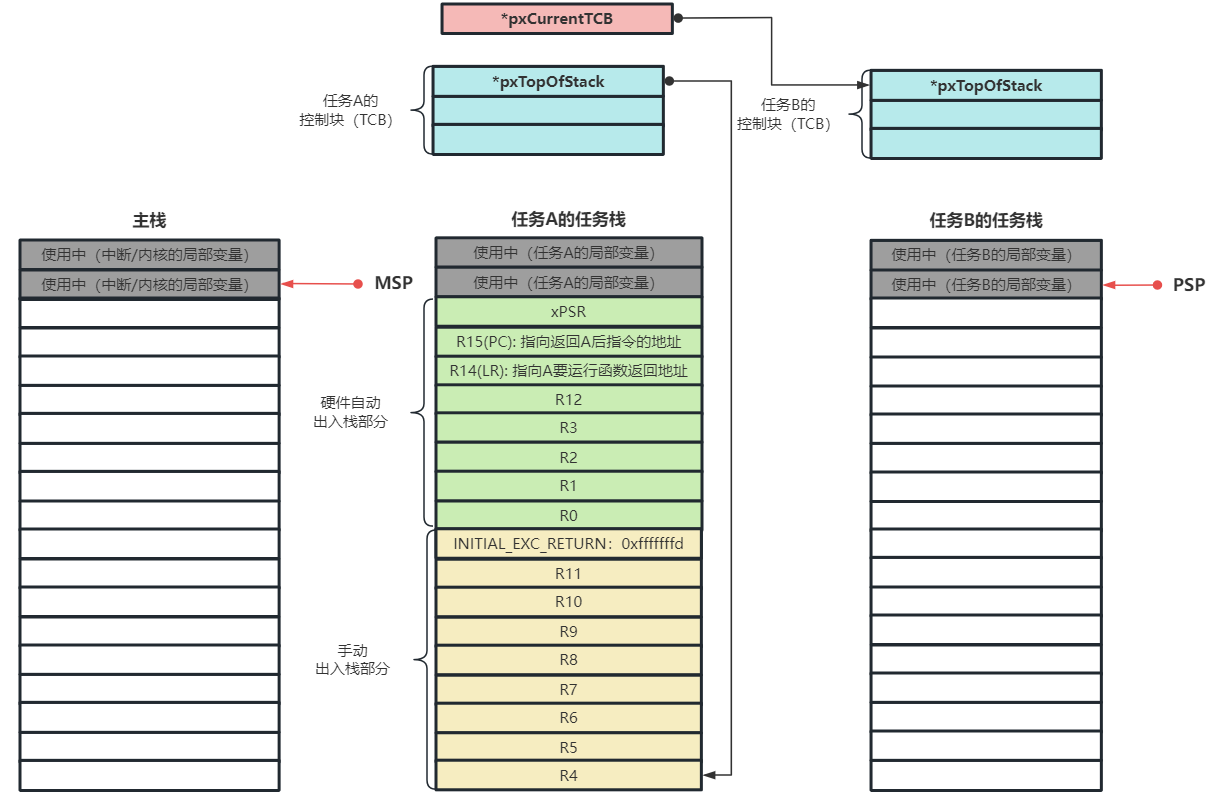
* 将R3-R14压入主栈，进入中断后r14自动更新保存的是0xfffffffd（在进入中断前，r14保存的是任务A当前执行指令所在函数的返回地址，pc保存的是当前执行指令的下一条指令的地址），保存0xfffffffd到主栈，是为了下面再退出中断前恢复给R14，最后调用bx r14指令使用。但是在任务栈中也保存了一份0xfffffffd，代码里也会恢复给R14，还不清楚为什么，在M3的代码中就没有在任务栈保存0xfffffffd。调用vTaskSwitchContext函数，用来寻找下一个要执行的任务，同时pxCurrentTCB指向任务B的任务块TCB：



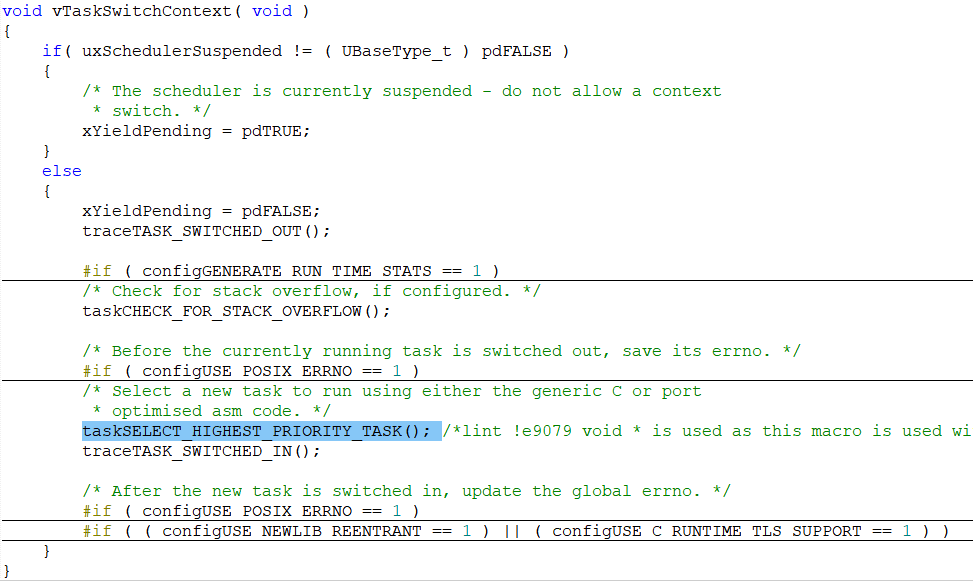
* 恢复任务B的现场，取任务B的栈顶指针交给R0，R0=pxCurrentTCB->pxTopOfStack，将R0充当栈指针，手动将B栈中保存的R4-R11R14出栈赋值给对应寄存器，再将R0赋值给PSP：



* 执行BX R14指令，自动恢复现场，1.因为R14的数值为0xfffffffd，所以SP指针将自动切换到PSP；2.会自动将任务栈里R0-R3、R12、R15(PC)、R14(LR)出栈赋值给对应寄存器：



* 简单分析下vTaskSwitchContext()函数，在里面会调用taskSELECT\_HIGHEST\_PRIORITY\_TASK()获取更高优先级的任务



对于freertos的调度器，它有两种方式寻找下一个最高优先级的任务，分别为特殊方式和常用方式，在FreeRTOSConfig.h中可通过宏定义设置，课程中设置的是1，如下：

/\* 0：使用常用方式来选择下一个要运行的任务；

1：使用特殊方法来选择下一个要运行的任务 \*/

#define configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION 1

* 常用方法，uxTopReadyPriority记录就绪态中最高优先级值，创建任务时会更新值，有任务添加到就绪表时也会更新值。这种方法对任务的数量无限制。

#define taskSELECT\_HIGHEST\_PRIORITY\_TASK() \

{ \

UBaseType\_t uxTopPriority = uxTopReadyPriority; \

\

while( listLIST\_IS\_EMPTY( &( pxReadyTasksLists[ uxTopPriority ] ) ) )\

{ \

configASSERT( uxTopPriority ); \

--uxTopPriority; \

} \

/\* 获取优先级最高任务的任务控制块 \*/

listGET\_OWNER\_OF\_NEXT\_ENTRY(pxCurrentTCB, &( pxReadyTasksLists[ uxTopPriority ] ) ); \

uxTopReadyPriority = uxTopPriority; \

}

* 特殊方法，uxTopReadyPriority每个bit位表示一个优先级，bit0表示优先级0，bit31表示优先级31，使用此方式优先级最大只能是32个。

#define taskSELECT\_HIGHEST\_PRIORITY\_TASK() \

{

UBaseType\_t uxTopPriority; \

/\* 获取优先级最高的任务 \*/

portGET\_HIGHEST\_PRIORITY( uxTopPriority, uxTopReadyPriority );

configASSERT( listCURRENT\_LIST\_LENGTH( &( pxReadyTasksLists[ uxTopPriority ] ) ) > 0 );

/\* 获取优先级最高任务的任务控制块 \*/

listGET\_OWNER\_OF\_NEXT\_ENTRY(pxCurrentTCB, &amp;( pxReadyTasksLists[ uxTopPriority ] ) ); \

}

其中：

#define portGET\_HIGHEST\_PRIORITY( uxTopPriority, uxReadyPriorities ) uxTopPriority = ( 31UL - ( uint32\_t ) \_\_clz( ( uxReadyPriorities ) ) )

\_\_clz( ( uxReadyPriorities ) 是计算uxReadyPriorities 的前导零个数，如：二进制0001 1010 0101 1111的前导零个数为3，可以知道，最高优先级uxTopPriority等于31减去前导零个数。知道最高优先级的优先级，则通过listGET\_OWNER\_OF\_NEXT\_ENTRY()对应最高优先级的列表项，将pxCurrentTCB指向对应的控制块。

对于listGET\_OWNER\_OF\_NEXT\_ENTRY：

#define listGET\_OWNER\_OF\_NEXT\_ENTRY( pxTCB, pxList )

{

List\_t \* const pxConstList = ( pxList );

/\* Increment the index to the next item and return the item, ensuring \*/

/\* we don't return the marker used at the end of the list. \*/

/\* 节点索引指向链表第一个节点调整节点索引指针，指向下一个节点，

\* 如果当前链表有N个节点，当第N次调用该函数时，pxIndex则指向第N个节点 \*/

( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;

/\* 当遍历完链表后，pxIndex 回指到根节点 \*/

if( ( void \* ) ( pxConstList )->pxIndex == ( void \* ) &( ( pxConstList )->xListEnd ) ) \

{

( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;

}

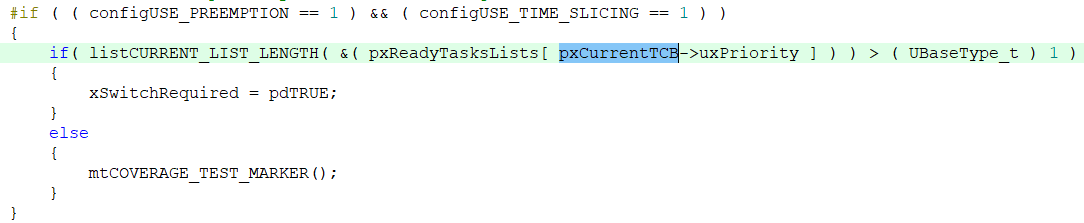
/\* 获取节点的 OWNER，即 TCB \*/

( pxTCB ) = ( pxConstList )->pxIndex->pvOwner;

}

listGET\_OWNER\_OF\_NEXT\_ENTRY()函数的妙处在于它并不是获取链表的第一个节点的 OWNER，怎么理解？假设当前链表有 N 个节点，当第 N 次调用该函数时，链表头（火车头）的pxIndex 则指向第 N 个节点，即每调用一次，节点遍历指针pxIndex 则会向后移动一次，用于指向下一个节点。那么对于同优先级任务的时间片轮转，就是利用了这个功能。

比如，优先级3下有两个任务，当系统第一次切换到优先级为3的任务（包含了任务1和任务 2，它们的优先级相同）时，pxIndex指向任务1，任务1得到执行。当时间片周期1ms过后，发生systick中断，在xTaskIncrementTick()中判断同优先级就绪链表中有大于1个任务，就会触发pendsv进行切换，这个时候 pxIndex 指向任务2，任务2得到执行，任务 1和任务 2 轮流执行，享有相同的CPU时间，即所谓的时间片。



# 队列

队列、信号量、互斥锁都是基于队列的数据结构衍生的，所以我们只需要了解队列的底层实现原理即可。另外它们本身都是线程安全的，在读写操作内部有相关的关中断和互斥访问等机制。

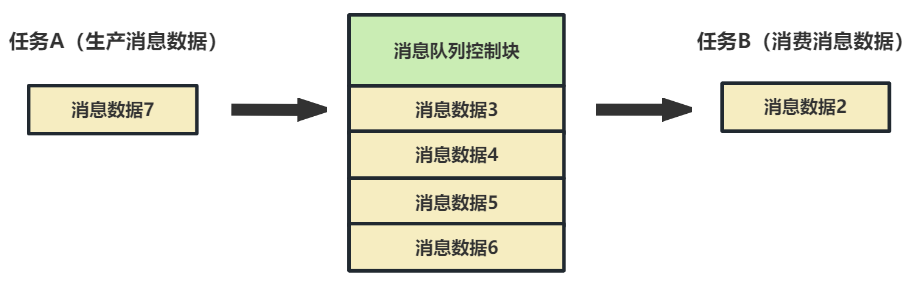
在应用中，使用队列可以达到2种效果：1）相较于全局变量和接口函数来传递数据，队列可以达到阻塞当前任务线程，让出 CPU 使用权的效果；2）使用队列可以达到数据缓存，用空间换时间的效果，对于这种场景的原理，大家可以去学习一下《ARM32课程》中环形队列的内容。

针对第1种效果说明一下：

1. 一个任务线程很少只去做一件事，不然也太浪费了，一个任务本身会消耗很大的资源，如果因为一个事情的数据资源被阻塞，导致任务当中的其他事情无法执行，肯定是无法接受的，比如在显示这个页面时，PM2.5数据用到了一个队列（数据一秒钟生产一次），温湿度数据用到了一个队列，程序逻辑中总不能因为PM2.5数据资源的无法到位而阻塞温湿度数据的显示，这就不太合理了。我看过很多实际产品项目的 RTOS 代码，队列确实用的不多，到时在课程中我会给大家分享两个实际产品项目。当然我们在实战课程的代码里还是有用到，带着大家一起学习。



1. 消息队列毕竟有内存开销，如果只是为了使用而使用，又没有达到阻塞效果也没有起到缓存的功能，那就得不偿失了。



### 创建队列

#define xQueueCreate( uxQueueLength, uxItemSize ) \ xQueueGenericCreate( ( uxQueueLength ), ( uxItemSize ), ( queueQUEUE\_TYPE\_BASE ) )

QueueHandle\_t xQueueGenericCreate( const UBaseType\_t uxQueueLength, const UBaseType\_t uxItemSize,

const uint8\_t ucQueueType )

举例：

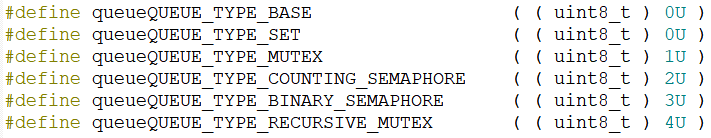
queue = xQueueCreate( 100, sizeof( int ) );

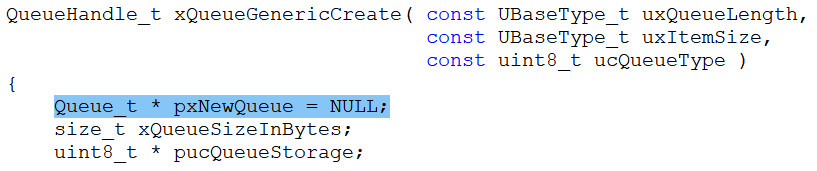
参数说明：

uxQueueLengt 队列长度， 即有多少个队员，实例中为有100个int类型的队员

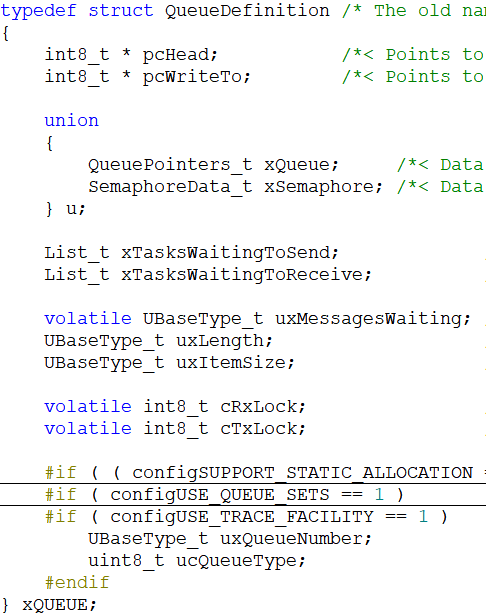
uxItemSize 队员类型大小，每队员内存空间的大小，传参时直接通过sizeof返回

ucQueueType 类型，可以是队列、信号量、互斥锁等。对于消息队列，传入queueQUEUE\_TYPE\_BASE枚举值





1. 首先定义了一个队列指针变量，我们展开这个数据结构：



再将共用体展开，解释下主要成员：

typedef struct QueueDefinition

{

int8\_t \*pcHead; /\*< 指向队列存储区域的开始地址\*/

int8\_t \*pcWriteTo; /\*< 队列写指针,指向储存区下一个空闲的位置\*/

int8\_t \*pcTail; /\*< 指向队列存储区域的结尾地址\*/

int8\_t \*pcReadFrom; /\*< 队列读指针\*/

List\_t xTasksWaitingToSend; /\*< 消息发送阻塞的任务链表。按优先级高->低顺序 存储.\*/

List\_t xTasksWaitingToReceive; /\*< 消息读取阻塞的任务列表。按优先级高->低顺序 存储\*/

volatile UBaseType\_t uxMessagesWaiting; /\*< 当前队员数(待读取的)\*/

UBaseType\_t uxLength; /\*< 队列长度(最大队员数) \*/

UBaseType\_t uxItemSize; /\*< 队列中每个队员大小(bytes)\*/

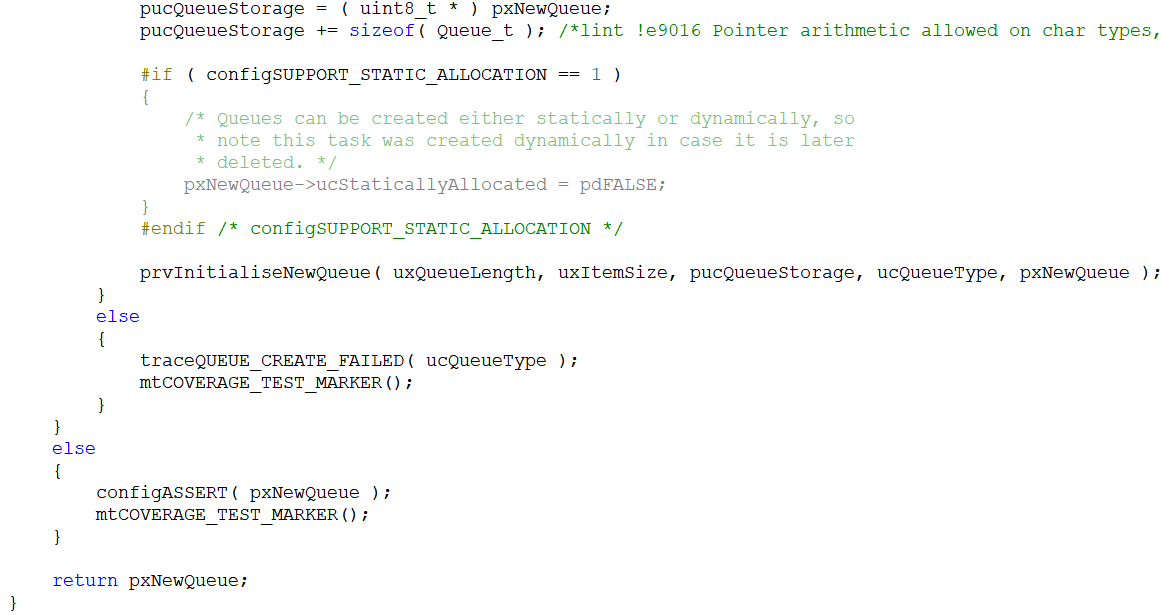
....

} xQUEUE;

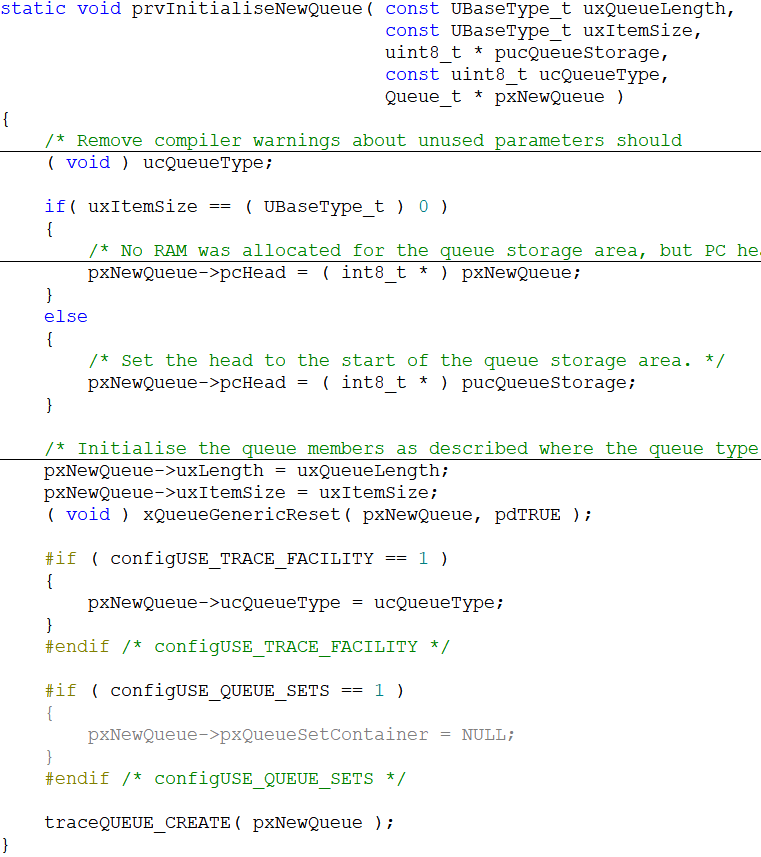
typedef xQUEUE Queue\_t;

再回到这个函数，



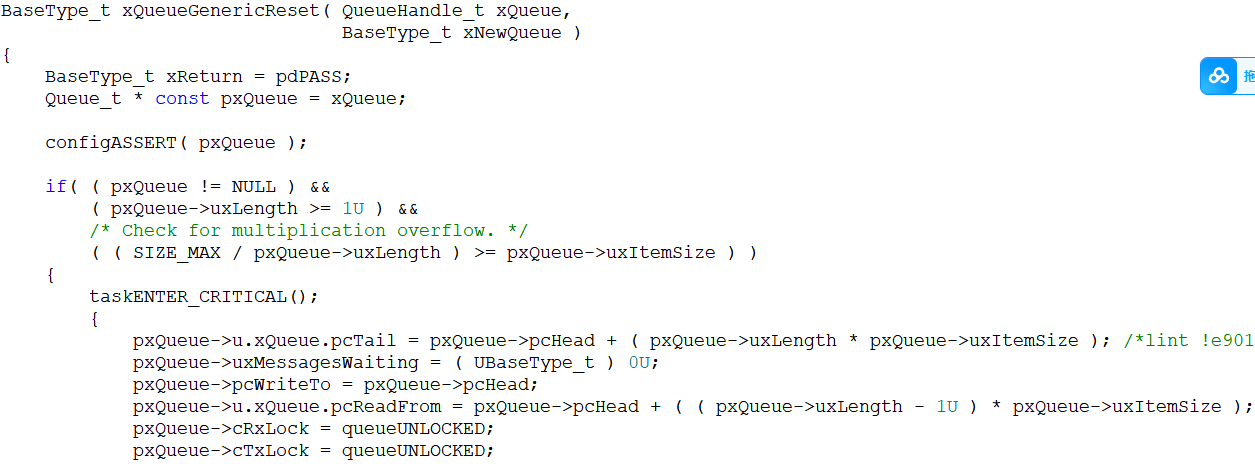
先给队列分配内存空间，包括队列的控制块和队列的队员空间；再将pucQueueStorage指向队员空间的起始位置。

接下来调用prvInitialiseNewQueue()对结构体成员赋值。

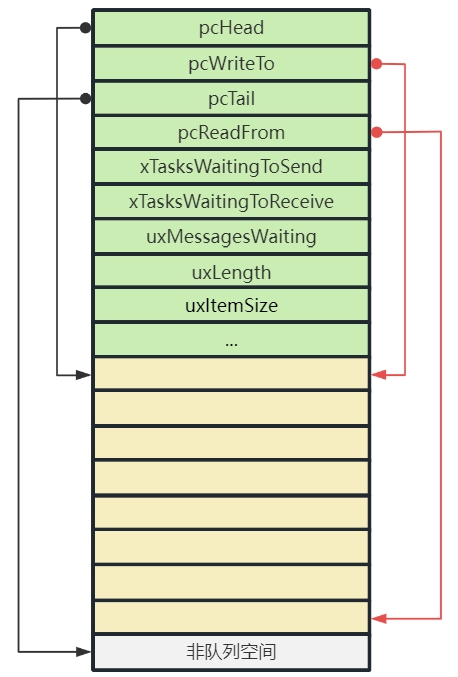


如果队员数据类型是0，将pcHead指向队列的起始位置，否则就将pcHead 指向队员空间的起始位置；再对成员中的队员个数和队员类型进行赋值。

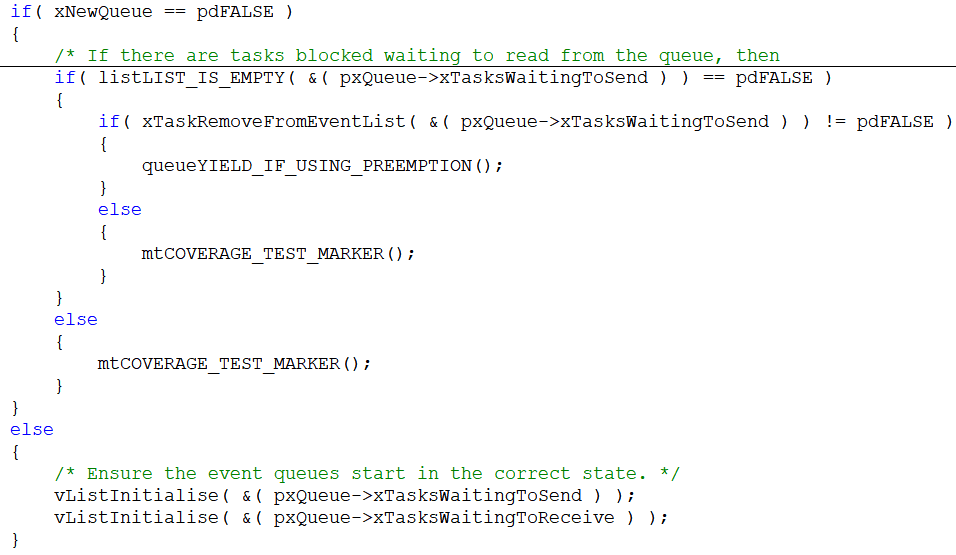
再调用xQueueGenericReset()函数。



经过这几行赋值代码，队列的内存布局是这样的，咱们就拿队员为uint8\_t类型，有8个队员举例，比较好理解:



接下来是创建接收阻塞链表和发送阻塞链表，实际上是创建了链表头结点:

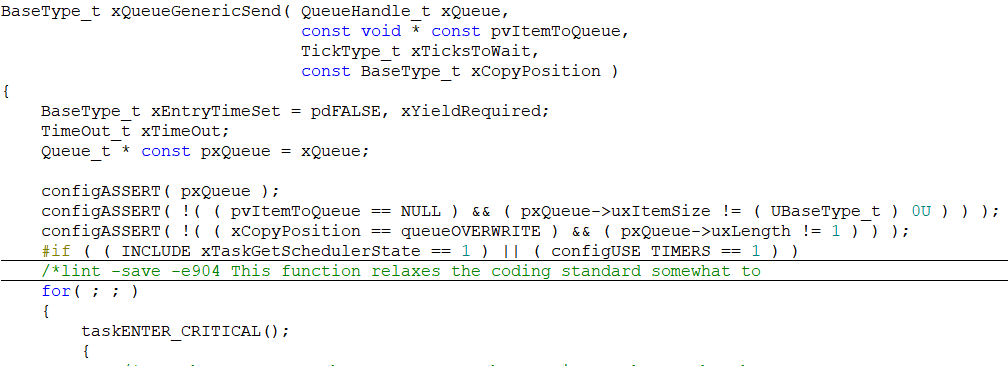


### 写队列

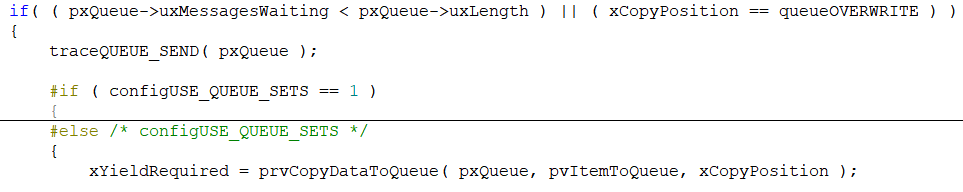
#define xQueueSend( xQueue, pvItemToQueue, xTicksToWait ) \

xQueueGenericSend( ( xQueue ), ( pvItemToQueue ), ( xTicksToWait ), queueSEND\_TO\_BACK )

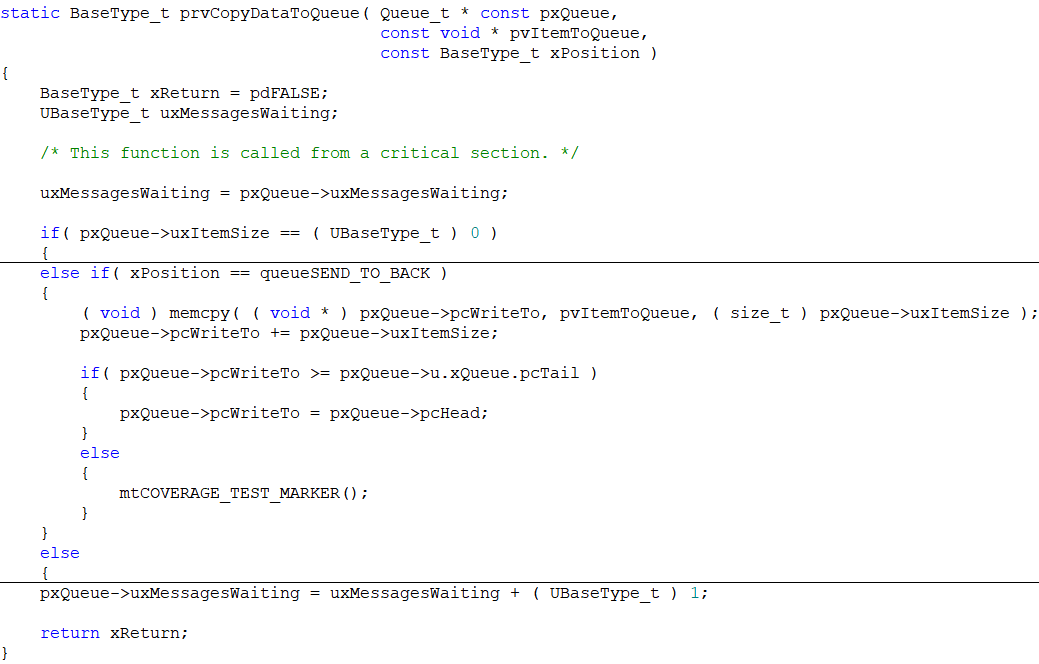
默认是queueSEND\_TO\_BACK方式。



1. 关闭总中断。



1. 判断一下队列是否满了，或者覆盖模式，如果不是就调用prvCopyDataToQueue() 实现入队操作。



我们直接看queueSEND\_TO\_BACK的逻辑，

( void ) memcpy( ( void \* ) pxQueue->pcWriteTo, pvItemToQueue, ( size\_t ) pxQueue->uxItemSize );

用memcpy 把新队员复制到pcWriteTo指向的空间。

 pxQueue->pcWriteTo += pxQueue->uxItemSize;

写完了，把写指针下移，指向待写入的空间

 if( pxQueue->pcWriteTo >= pxQueue->u.xQueue.pcTail )

{

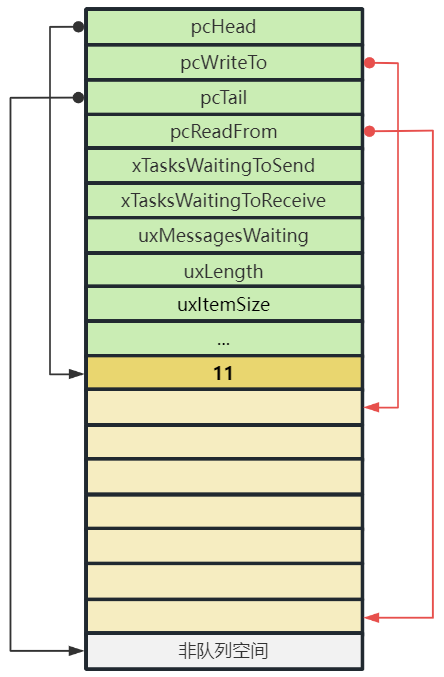
      pxQueue->pcWriteTo = pxQueue->pcHead;

 }

如果写到尾部了，就指向头部。

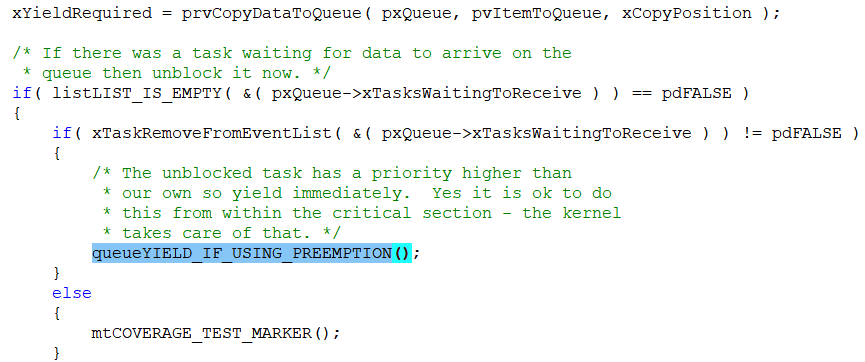
记住，每次在写入队列时写入的是一个队员，不是多个队员。假如现在写入的队员数

据是uint8\_t类型的数据11，内存布局是这样的：

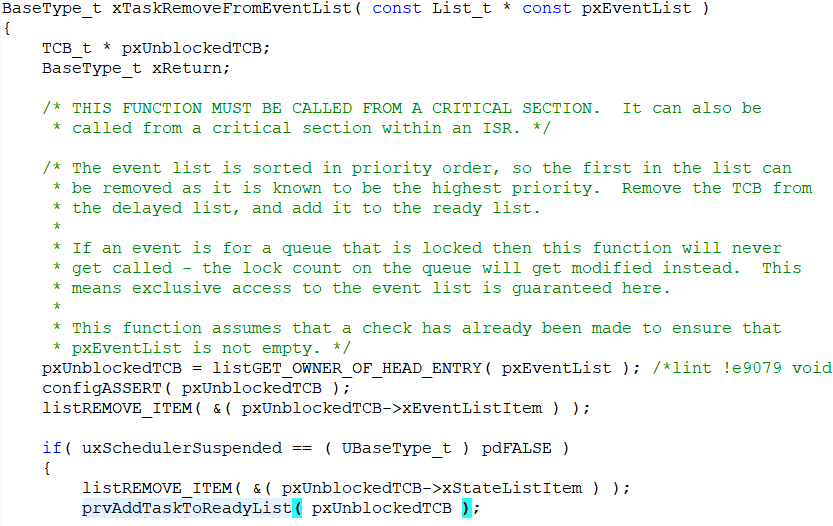


在读写过程中，pcHead和pcTail是不会变化的，变化的是pcWriteTo和pcReadFrom。

1. 再回到xQueueGenericSend()函数



如果队列的接收阻塞链表不为空，再调用xTaskRemoveFromEventList()函数，

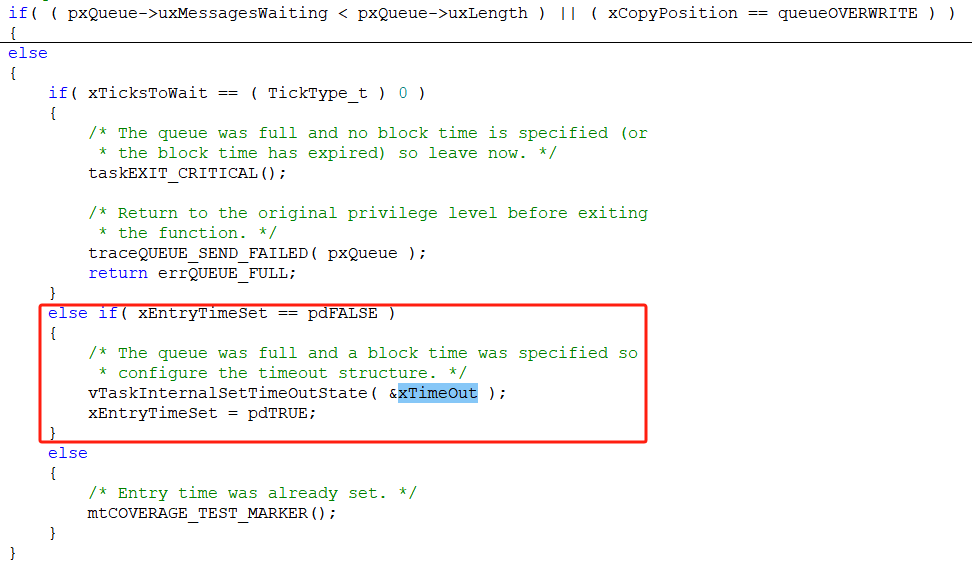


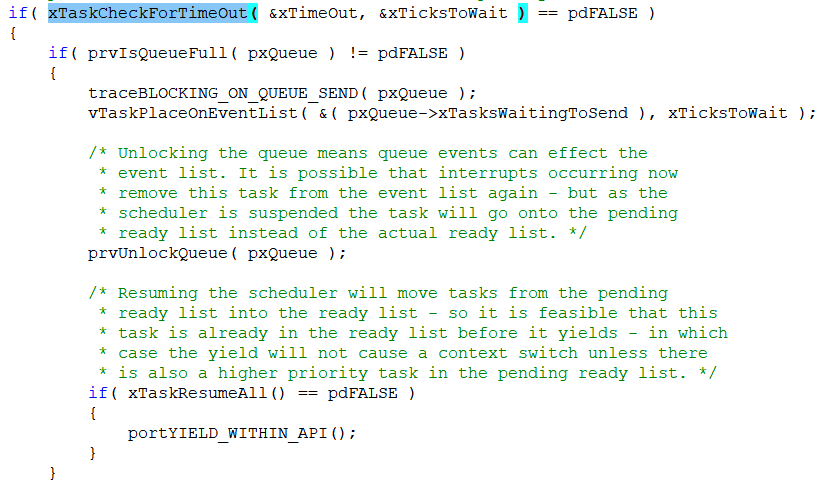
将被阻塞的任务从链表中删除，再将删除的这个任务添加到就绪链表中，接着就等着触发pendsv异常进行任务调度切换就好了。

回到xQueueGenericSend()函数，调用queueYIELD\_IF\_USING\_PREEMPTION()触发pendsv异常，在异常服务函数中选择就绪链表里优先级最高的任务进行调度。

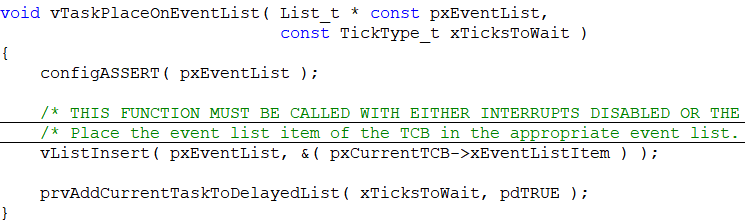
#define queueYIELD\_IF\_USING\_PREEMPTION() portYIELD\_WITHIN\_API()

#define portYIELD\_WITHIN\_API portYIELD

接着往下看

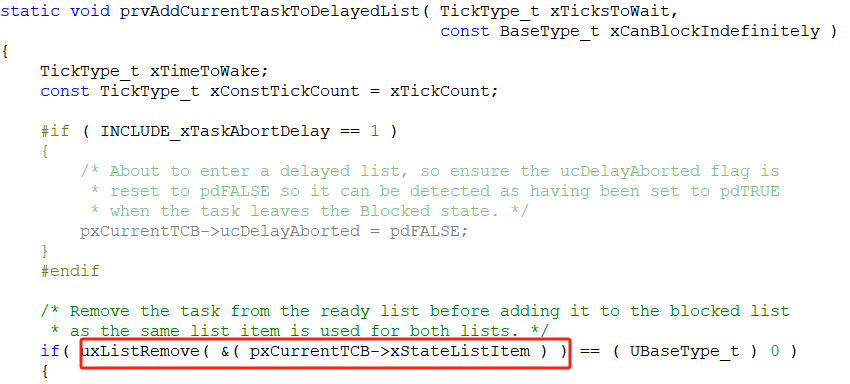
如果队列是满的，并且超时时间不为0，记录系统节拍和溢出次数给下面的代码使用，

将当前任务插入到队列的发送阻塞链表中，跟进去看看，



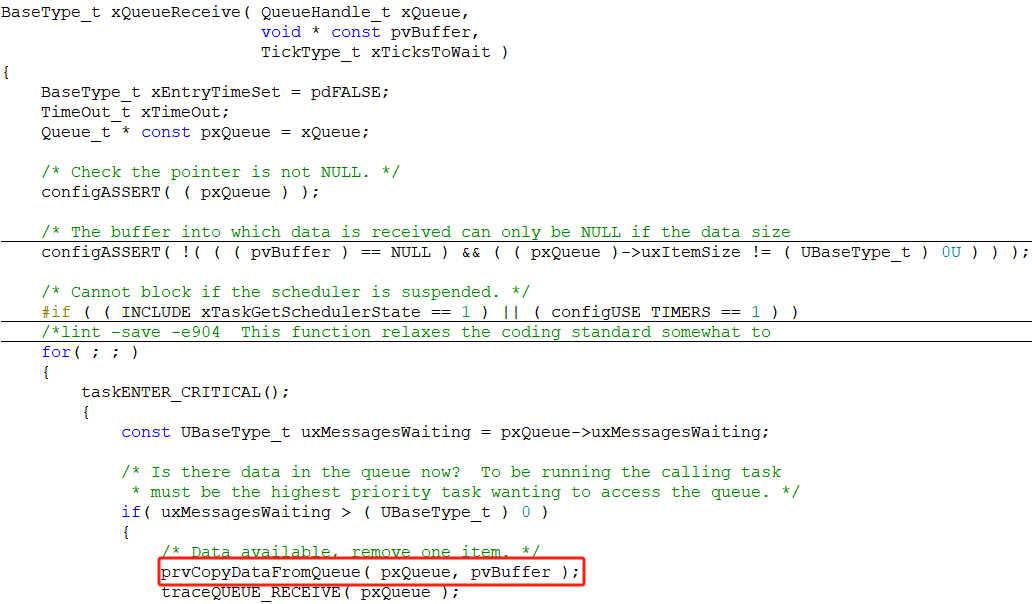
执行vListInsert( pxEventList, &( pxCurrentTCB->xEventListItem ) );插入到队列发送阻塞链表，

执行prvAddCurrentTaskToDelayedList( xTicksToWait, pdTRUE );按照超时时间插入到延时阻塞链表中，并且还会从就绪链表中删除。下面代码就是将运行任务也就是当前发送队列的任务从所在的对应状态链表中删除（它肯定是在就绪链表中），代码很考察链表基本知识。

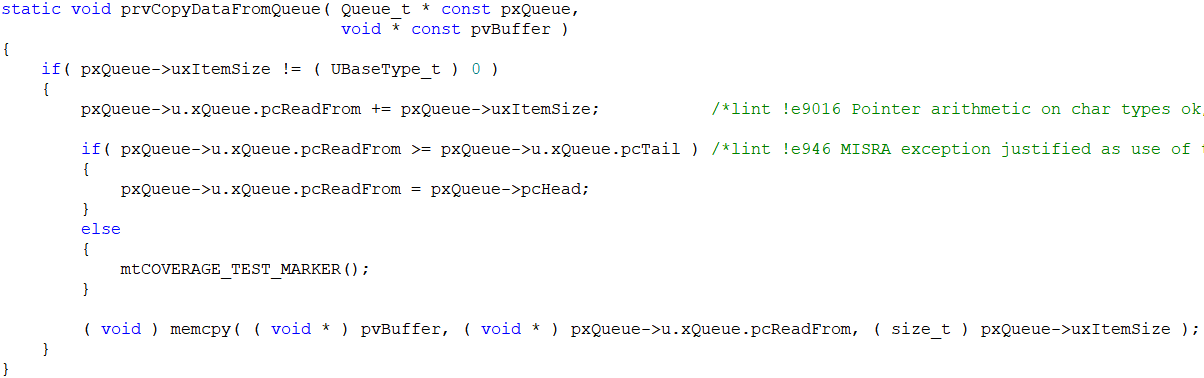


所以当前任务要解除阻塞，要么是队列不为满了，要么是超时了。

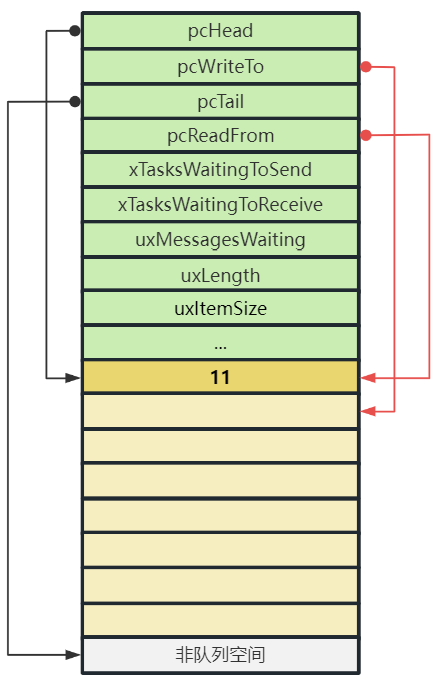
### 读队列



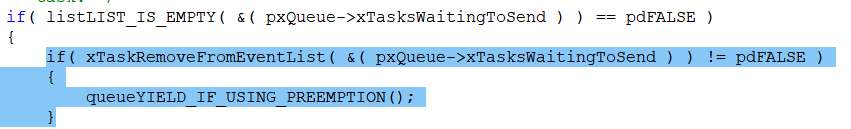
进入prvCopyDataFromQueue函数，



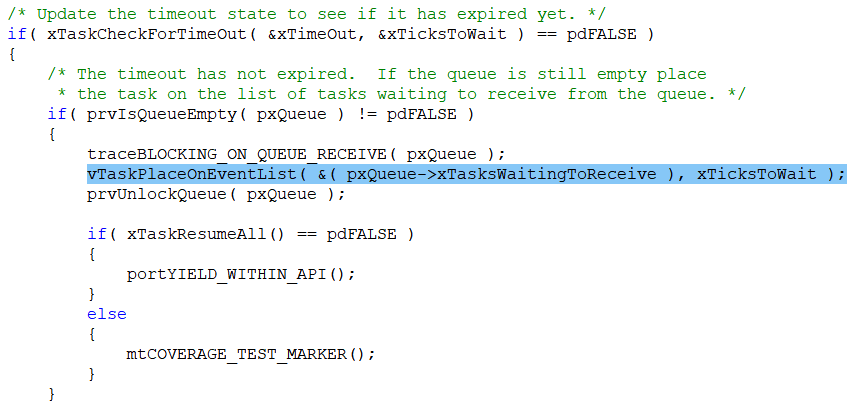
先将读指针下移（很重要，不然梳理不清读取过程），如果指向了尾部，就指向头部。接着拷贝数据。



回到xQueueReceive()函数，



和发送过程类似，从等待发送链表中删除任务，并添加到就绪链表，触发pendsv异常。



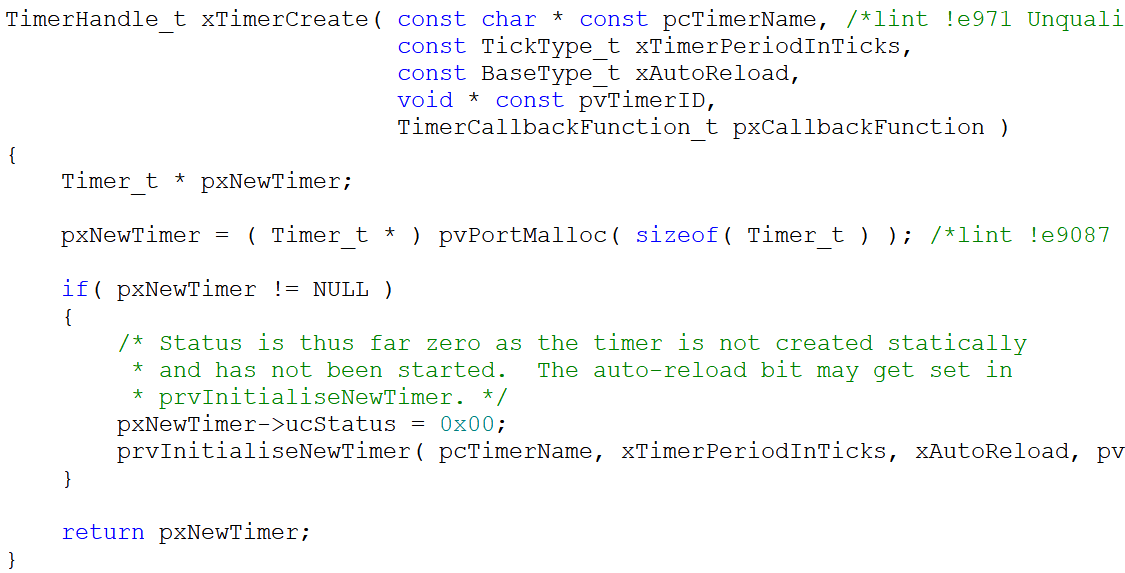
将当前任务添加到等待接收阻塞链表中，也添加到延时阻塞链表中。

总结下，freertos有几种链表，每个优先级对应1个链表，延时阻塞链表（实际上有2个，不会同时用），挂起链表、每个队列（包括信号量/互斥锁）等待发送链表，每个队列（包括信号量/互斥锁）等待接收链表，其实定时器里也有链表。

# 软件定时器

RTOS的软件定时器，在ARM单片机中，是基于SysTick定时器实现的，ms级别，所以它的精度不高，但是可以方便我们在任务中实现一些周期性的工作。

### 创建定时器



参数说明：

pcTimerName: 定时器名字，用处不大，只在调试时可能用到

xTimerPeriodInTicks: 周期，以Tick为单位

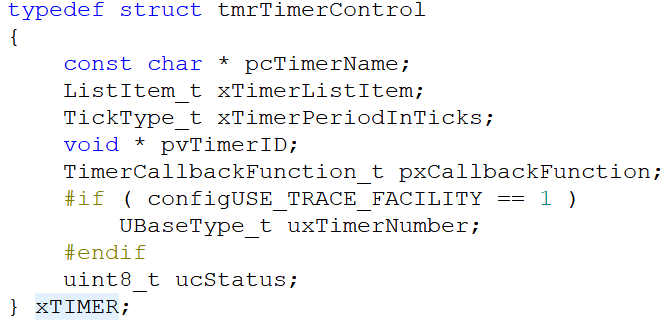
uxAutoReload: 类型，pdTRUE表示定时是周期性的，pdFALSE表示是一次性的

pvTimerID: 回调函数可以使用此参数，比如同一个回调函数分辨是哪个定时器

pxCallbackFunction: 回调函数

返回值: 成功则返回TimerHandle\_t, 否则返回NULL。

展开数据结构：



成员说明：

pcTimerName：定时器名字

xTimerListItem：定时器链表节点，用于将定时器插入到定时器链表之中(等待超时)

xTimerPeriodInTicks：定时器周期，单位为tick(系统节拍周期)

pvTimerID：用于标识定时器的ID

pxCallbackFunction：定时器的回调函数

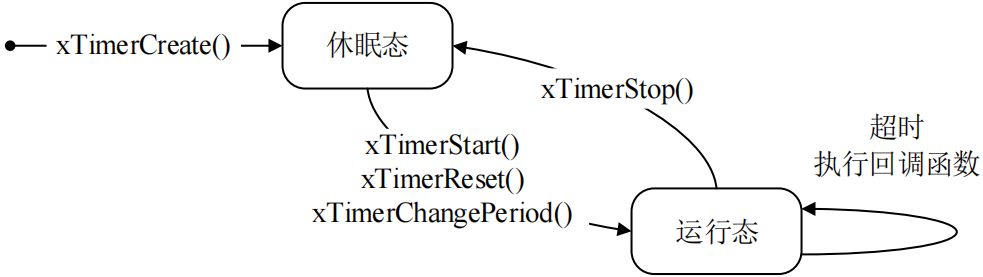
ucStatus：有3种数值：

#define tmrSTATUS\_IS\_ACTIVE ( ( uint8\_t ) 0x01 ) //定时器是否活跃

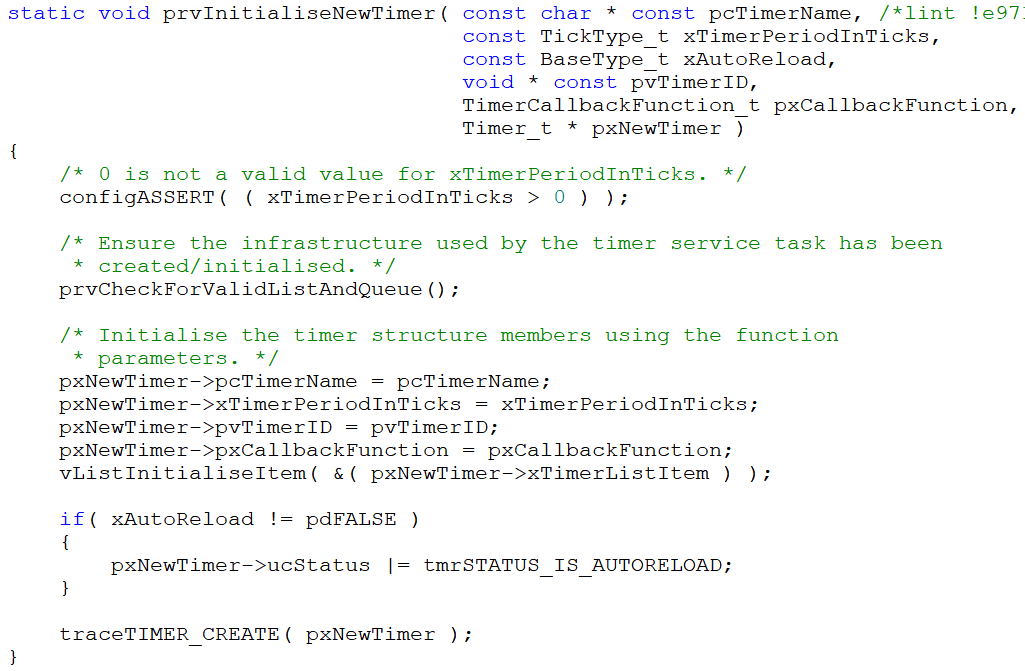
#define tmrSTATUS\_IS\_STATICALLY\_ALLOCATED ( ( uint8\_t ) 0x02 ) //定时器是否静态分配

#define tmrSTATUS\_IS\_AUTORELOAD ( ( uint8\_t ) 0x04 ) //定时器是否周期性

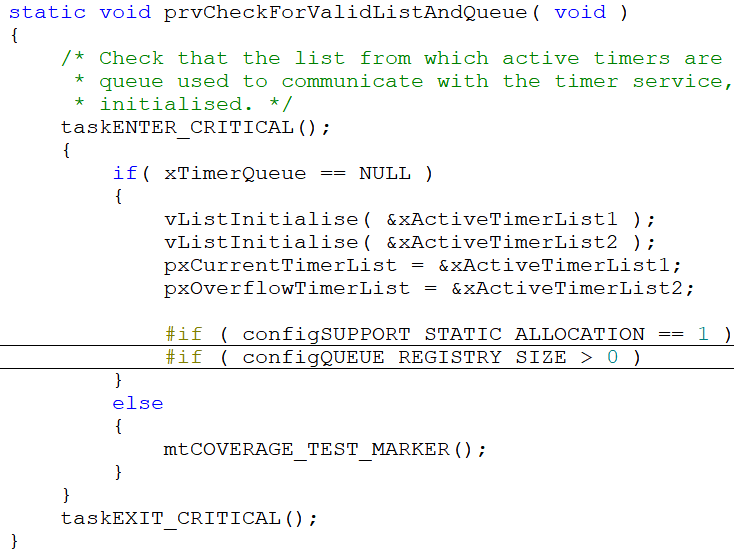
定时器是否活跃是什么意思呢？并不是说定时器一被创建就开始定时，与硬件定时器一样需要一个启动的命令，定时器才会被插入定时器链表，处于活跃态。



再进入prvInitialiseNewTimer()函数：

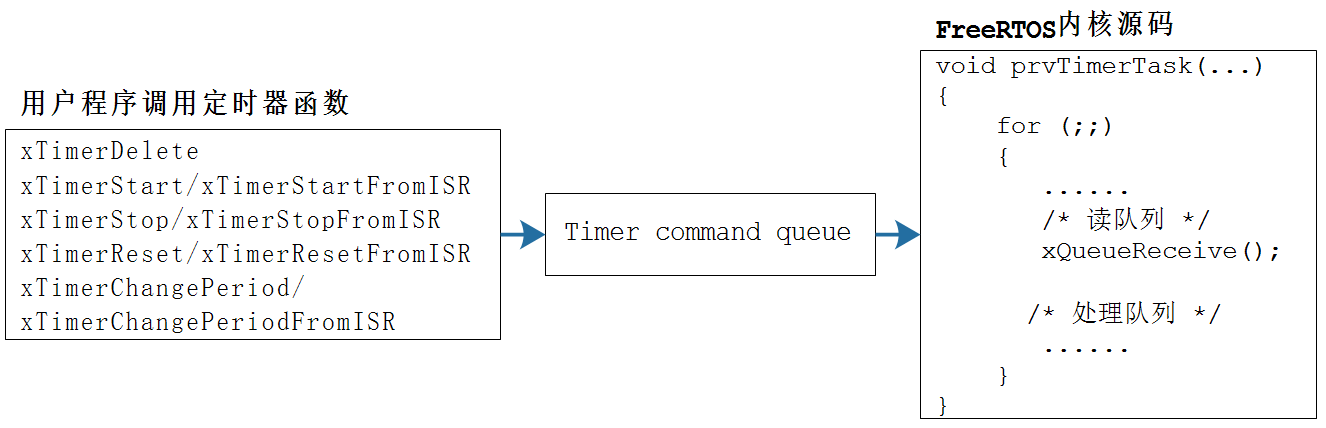


如果是第一次创建定时器，调用prvCheckForValidListAndQueue()会初始化创建定时器的链表头和队列：

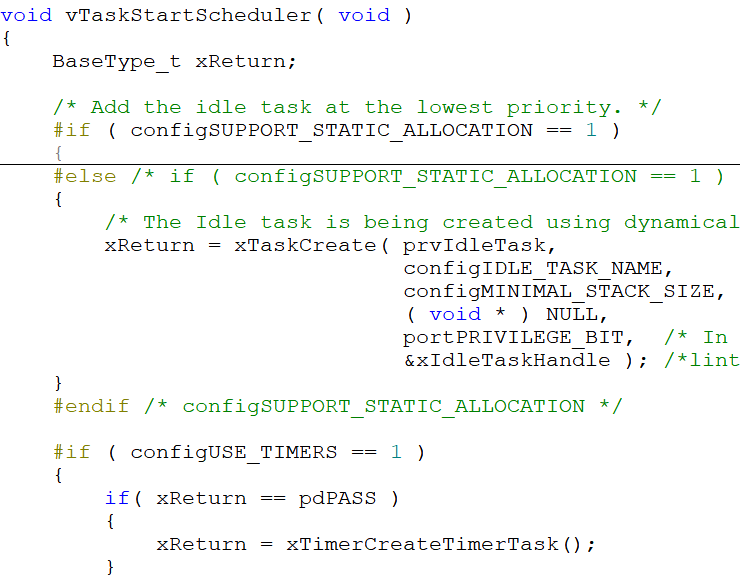


这里创建2个链表头的目的和2个延时阻塞链表是一样的，当tick+超时时间 溢出时，切换使用另外一个链表。另外还会创建一个队列，为什么定时器要使用队列呢？

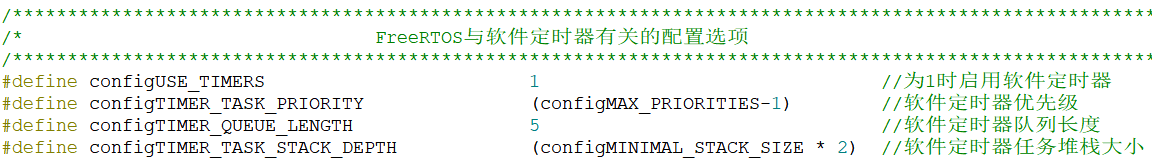
实际上，程序在调用定时器函数去启动/停止定时器时，是向定时器的守护任务通过队列发送了相关命令字。



那么守护任务是在哪里创建的呢？是在vTaskStartScheduler()：



需要在FreeRTOSConfig.h使能开关：



为了尽可能保证定时器的精度，定时器守护任务的优先级通常都会设置为最高档。因为定时器回调函数是在守护任务中运行的，所以回调函数也要像硬件中断服务函数一样，快进快出。同时也为了守护任务不占用太多CPU执行资源，当最小的定时超时时间未到时，且消息队列中也没有消息，则守护任务会进入阻塞状态，直到超时，守护任务会被唤醒去调用该定时器的回调函数。

### 发送命令字

#define xTimerStart( xTimer, xTicksToWait ) \

xTimerGenericCommand( ( xTimer ), tmrCOMMAND\_START, ( xTaskGetTickCount() ), NULL, ( xTicksToWait ) )

#define xTimerStop( xTimer, xTicksToWait ) \

xTimerGenericCommand( ( xTimer ), tmrCOMMAND\_STOP, 0U, NULL, ( xTicksToWait ) )

#define xTimerChangePeriod( xTimer, xNewPeriod, xTicksToWait ) \

xTimerGenericCommand( ( xTimer ), tmrCOMMAND\_CHANGE\_PERIOD, ( xNewPeriod ), NULL, ( xTicksToWait ) )

...

定时器的命令字包括：

#define tmrCOMMAND\_EXECUTE\_CALLBACK\_FROM\_ISR ( ( BaseType\_t ) -2 )

#define tmrCOMMAND\_EXECUTE\_CALLBACK ( ( BaseType\_t ) -1 )

#define tmrCOMMAND\_START\_DONT\_TRACE ( ( BaseType\_t ) 0 )

#define tmrCOMMAND\_START ( ( BaseType\_t ) 1 )

#define tmrCOMMAND\_RESET ( ( BaseType\_t ) 2 )

#define tmrCOMMAND\_STOP ( ( BaseType\_t ) 3 )

#define tmrCOMMAND\_CHANGE\_PERIOD ( ( BaseType\_t ) 4 )

#define tmrCOMMAND\_DELETE ( ( BaseType\_t ) 5 )

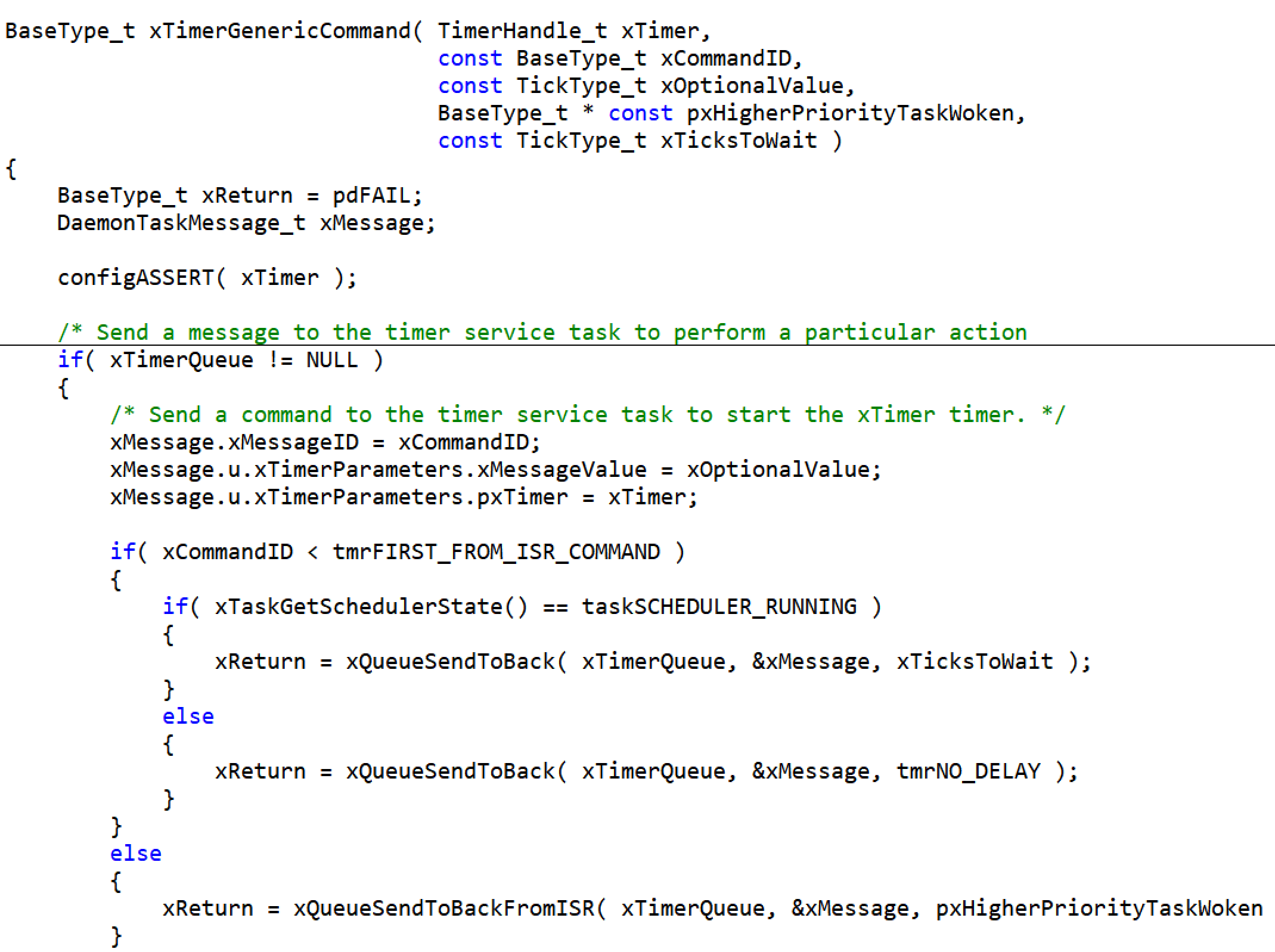
#define tmrFIRST\_FROM\_ISR\_COMMAND ( ( BaseType\_t ) 6 )

#define tmrCOMMAND\_START\_FROM\_ISR ( ( BaseType\_t ) 6 )

#define tmrCOMMAND\_RESET\_FROM\_ISR ( ( BaseType\_t ) 7 )

#define tmrCOMMAND\_STOP\_FROM\_ISR ( ( BaseType\_t ) 8 )

#define tmrCOMMAND\_CHANGE\_PERIOD\_FROM\_ISR ( ( BaseType\_t ) 9 )



参数说明：

xTimer：定时器句柄  
xCommandID：定时器命令ID

ID号-2~-1：直接调用用户指定的函数在守护任务执行前

ID号0~5：定时器的启动、复位、停止、更改周期、删除命令

ID号6~9：中断版的定时器的启动、复位、停止、更改周期命令

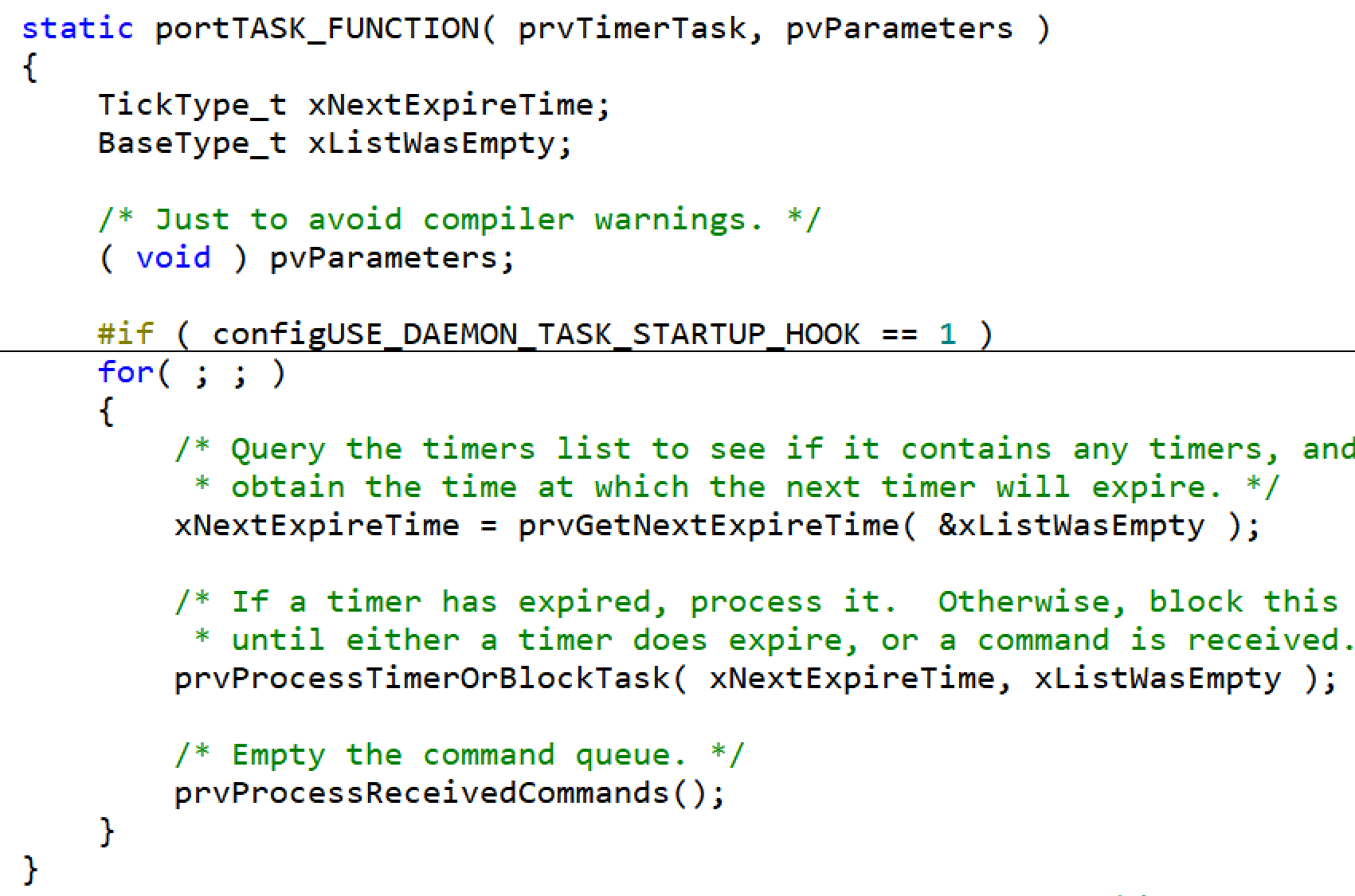
xOptionalValue：消息的value，新的定时周期或者当前定时器的定时值

pxHigherPriorityTaskWoken：用于中断函数中，是否切换任务

xTicksToWait：阻塞时间，因为是写队列，队列满了会写不进去

在函数中，如果命令字小于6，说明是在任务中发送队列数据，否则是在中断中发送队列数据。

### 定时器守护任务

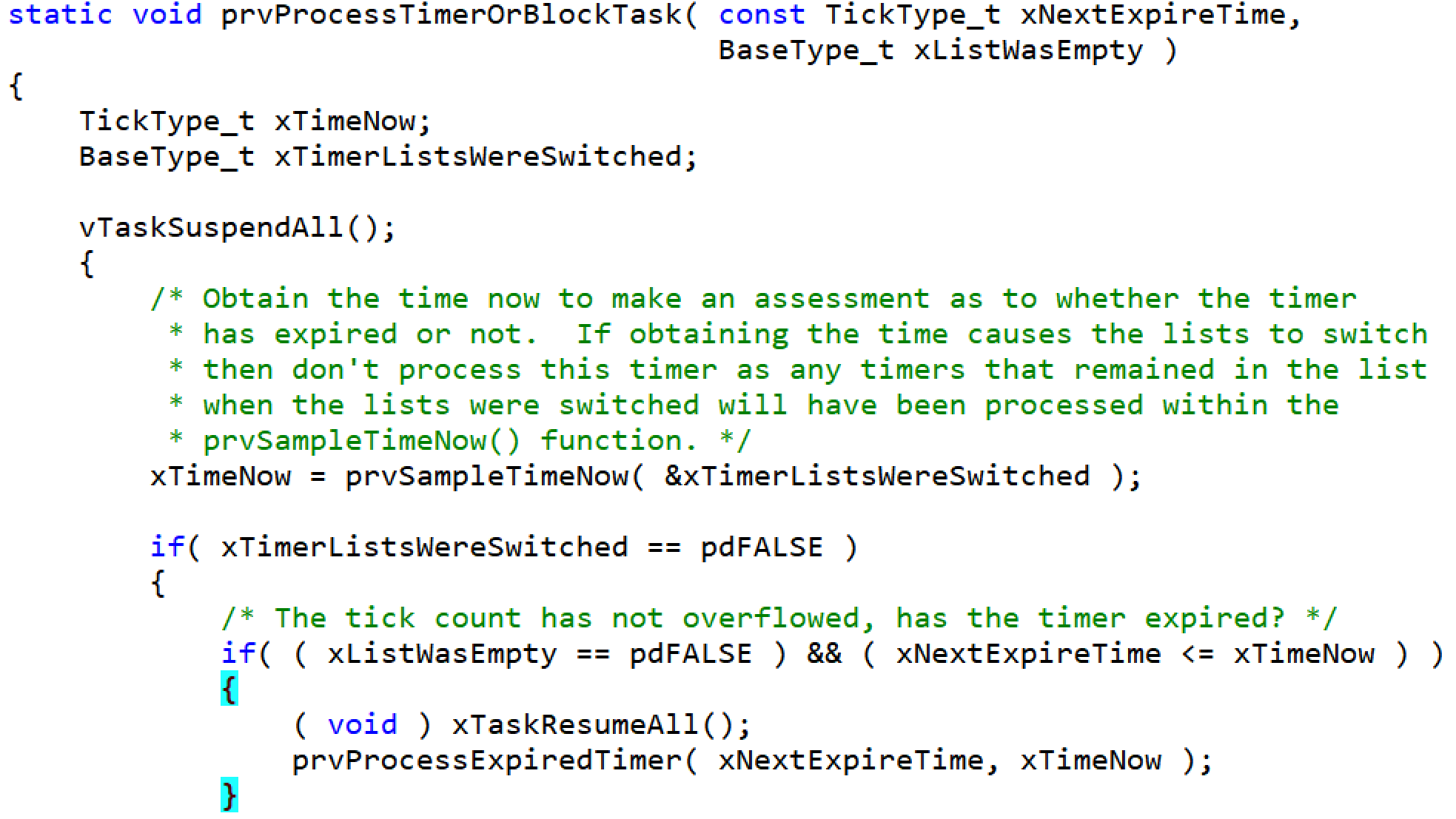


在守护任务中，主要做3件事情：

1. 获取定时器链表中最早的超时时间。  
   2）判断是否超时，超时则调用定时器的回调函数执行，否则阻塞守护任务，直到队列接收到命令或者定时器超时，任务被唤醒。  
   3）读队列，处理接收到的队列消息(命令)。

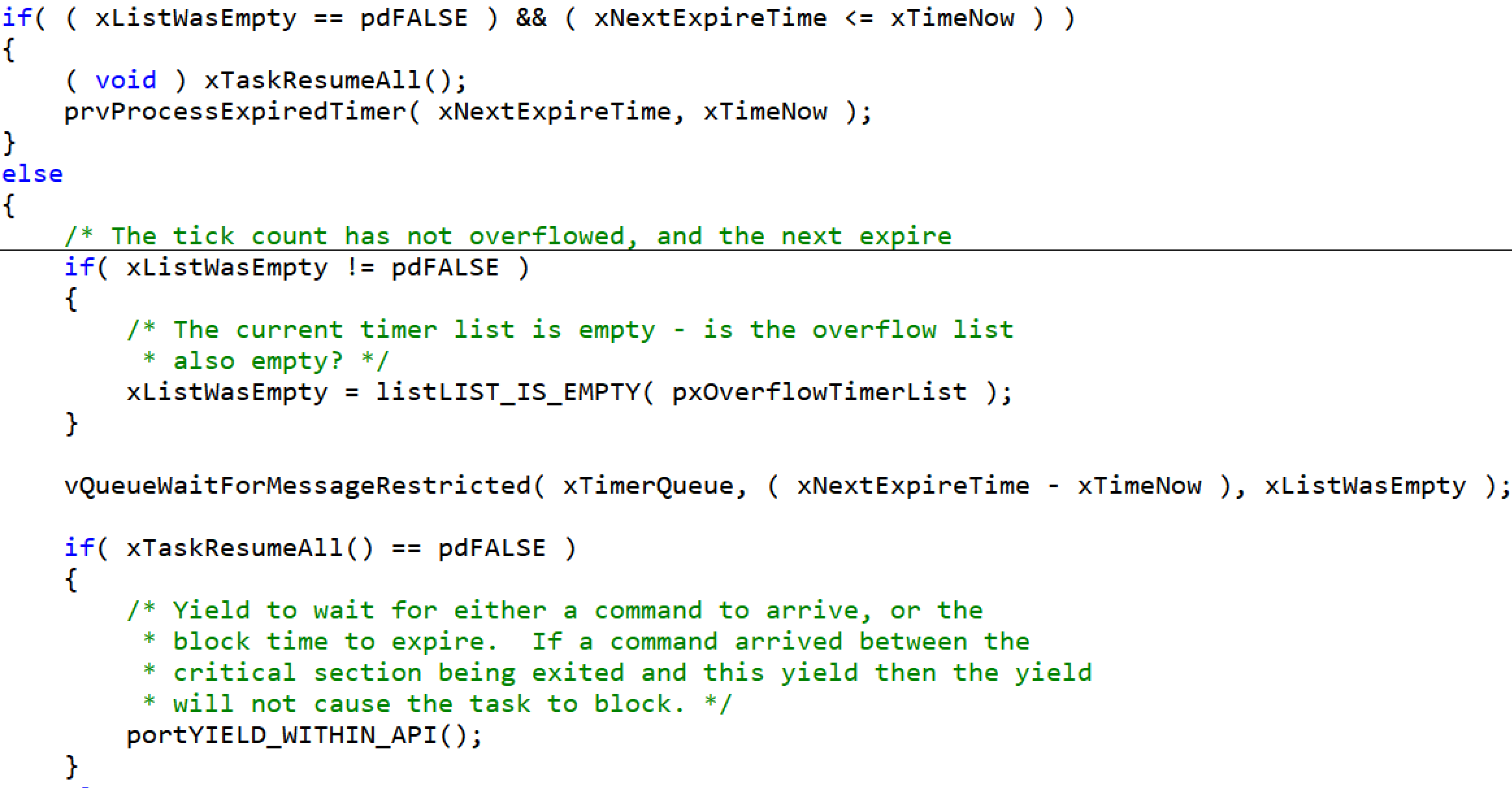
解释一下定时器链表中的计数值，假设创建了一个定时器，它的周期为20个tick，假设当前系统运行时间为100个tick，此时将该定时器插入链表中，计数值就是100+20=120个tick。守护任务在判断定时器是否超时，使用120与当前系统的运行时间进行对比，如果到了120（>=120）,则守护任务会去调用该定时器的回调函数执行。定时器在插入到链表中时，是以升序来排序的。

在判断是否超时时，调用prvProcessTimerOrBlockTask()：



通过prvSampleTimeNow()获取当前的系统时间；在该函数中还会判断当前tick计数值（系统时间）是否溢出，溢出需要切换定时器链表。

如果链表不为空，并且有定时器超时，则调用prvProcessExpiredTimer()，在里面会调用该定时器的回调函数。如果未超时并且队列中无命令字数据，调用vQueueWaitForMessageRestricted()函数让守护任务进入阻塞态插入到延时阻塞链表中（基于xNextExpireTime），并触发pendsv异常切换任务。



守护任务函数的最后，如果未阻塞，调用prvProcessReceivedCommands()去处理队列数据，比如

switch( xMessage.xMessageID )

{

case tmrCOMMAND\_START:

case tmrCOMMAND\_START\_FROM\_ISR:

case tmrCOMMAND\_RESET:

case tmrCOMMAND\_RESET\_FROM\_ISR:

/\* Start or restart a timer. \*/

pxTimer->ucStatus |= tmrSTATUS\_IS\_ACTIVE;

if(prvInsertTimerInActiveList(pxTimer, xMessage.u.xTimerParameters.xMessageValue + pxTimer->xTimerPeriodInTicks, xTimeNow, xMessage.u.xTimerParameters.xMessageValue ) != pdFALSE )

{

此时，会将定时器按升序插入到链表，超时时间为当前系统时间 + 创建定时器设置的定时周期。

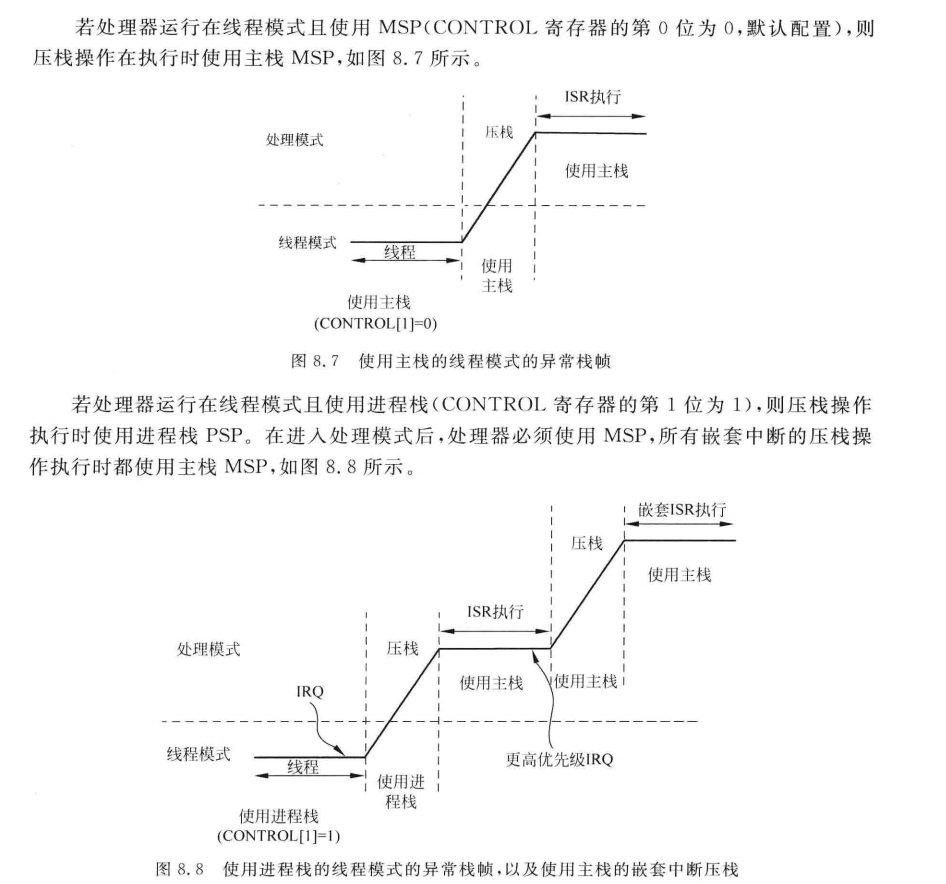
#define xTimerStart( xTimer, xTicksToWait ) \

xTimerGenericCommand( ( xTimer ), tmrCOMMAND\_START, ( xTaskGetTickCount() ), NULL, ( xTicksToWait ) )

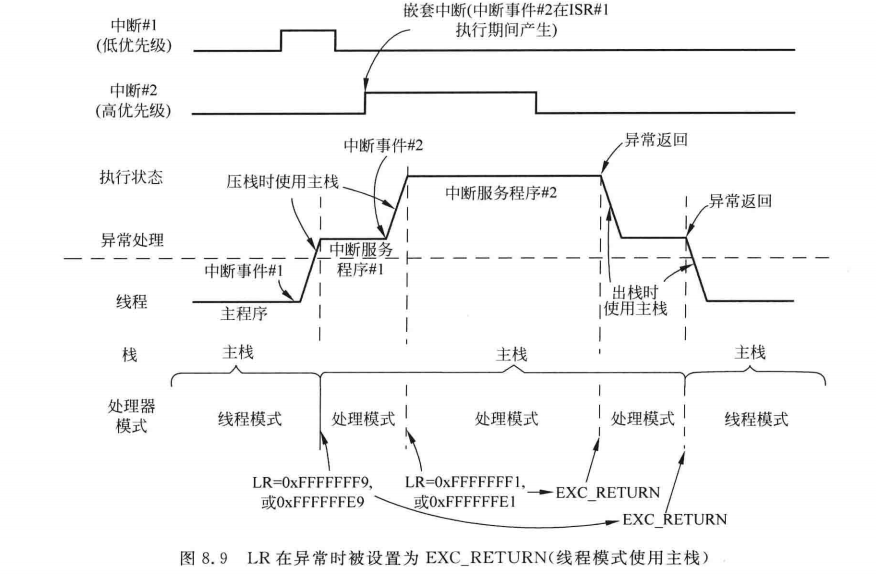
到此，freertos里关于任务、队列、软件定时器的底层实现分析就结束了。

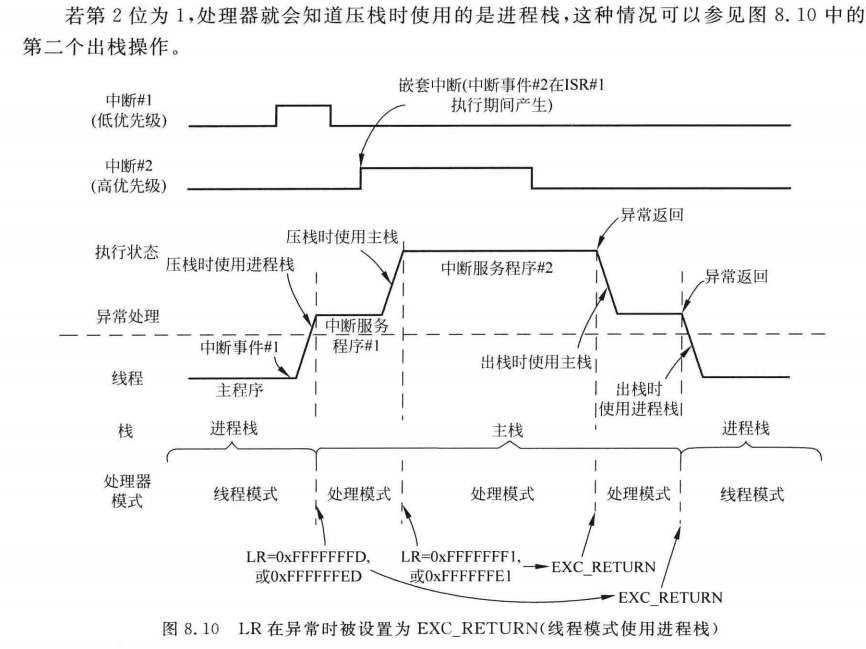
以下内容，大家可以不看，是我自己的一些总结记录。

备注：源自《ARM Cortex-M3与Cortex-M4权威指南》



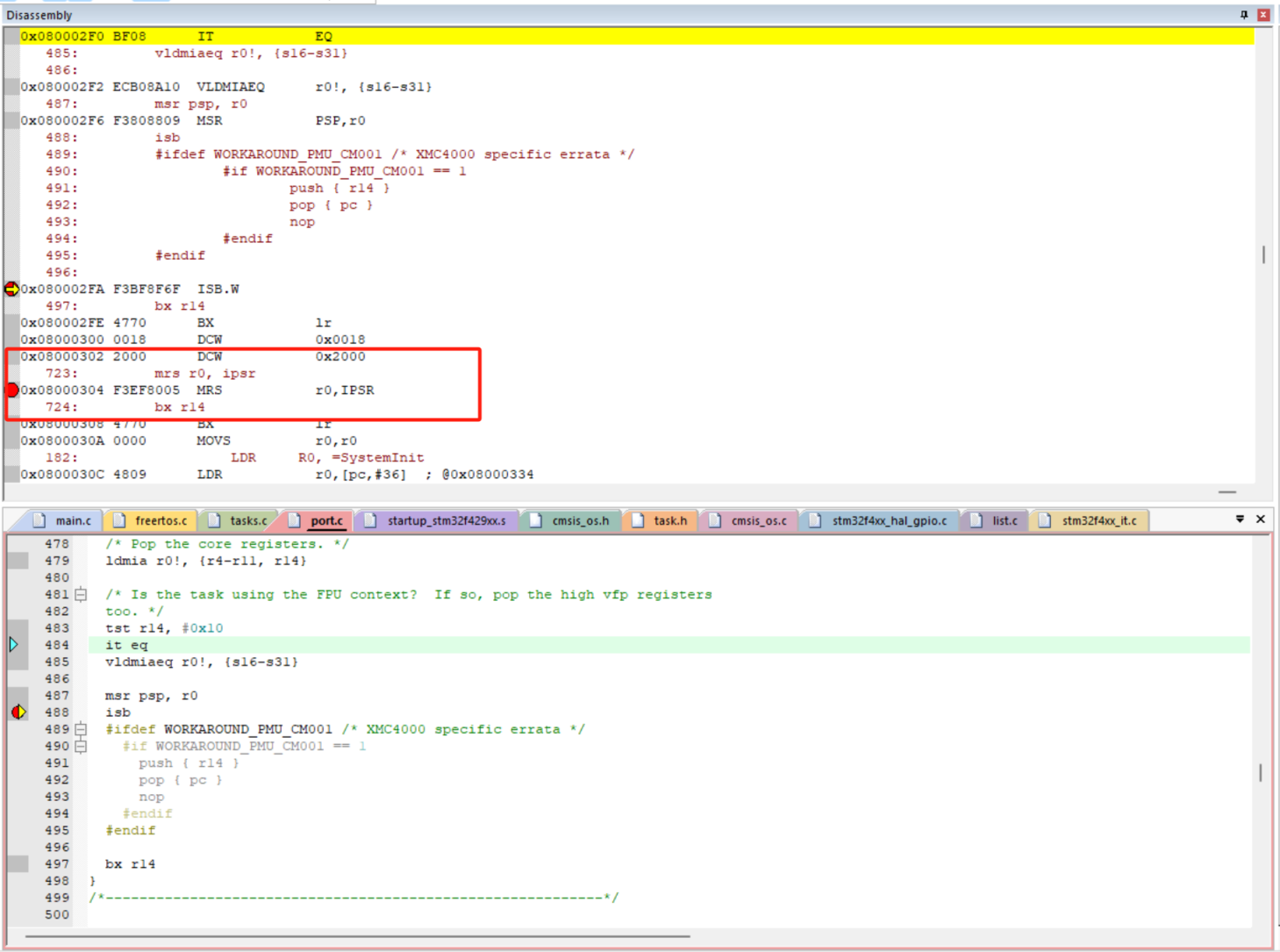






按照这个说法，在进入pendsv异常函数前（包括其他的中断），如果此时是在线程中，使用的是psp，r14会保存0xfffffffd(代表此时使用的是psp)，会自动保存现场到内存中，当异常中断执行完（异常和中断函数里使用的都是msp），恢复现场时根据内存中的0xfffffffd数据，cpu就知道了要返回到线程，使用psp。所以按理说就不需要pendsv函数(xPortPendSVHandler)里最后一行代码bx r14，我试着去掉这句话后，在检测仪产品上，确实还是能够正常运行的，暂时还没搞明白为什么要加这句话，也许是加上会保险一些吧。

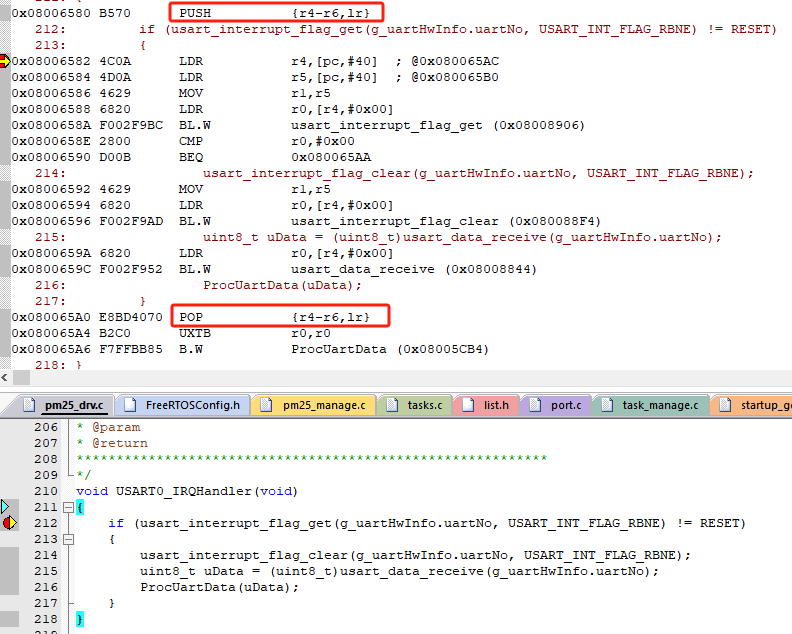
又有进展，经过和海东同学一起分析，之所以去掉bx r14也可以，是因为vPortGetIPSR的汇编代码是紧挨着xPortPendSVHandler汇编代码的，所以还是继续执行了bx r14：





经过验证，如果将vPortGetIPSR换个位置，就出错了。

继续分析，另外一个场景，比如现在正在执行某个任务，突然串口中断来了，那在串口中断函数里我们也没写bx r14 ，为啥能回到任务线程继续执行呢？因为在中断的汇编代码中固定会有将lr压入内存栈的过程：



经过查资料，pop pc 和lx r14作用一样，所以会回到任务线程模式继续执行。

那么注意，此时push压栈是主栈，而进入中断前自动压栈的是任务栈。

