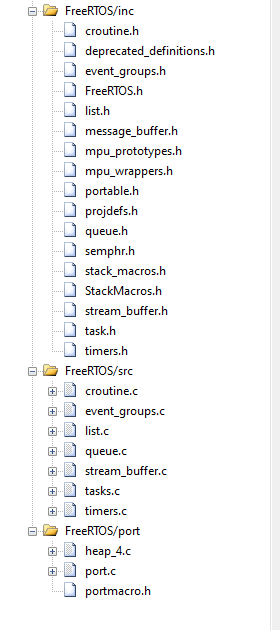
RTOS



注释掉stm32f4xx\_it.h中

void SVC\_Handler(void)

void PendSV\_Handler(void)

void SysTick\_Handler(void)

在FreeRTOSConfig.h中45行

改为 #if defined(\_\_ICCARM\_\_) || defined(\_\_CC\_ARM) ||defined(\_\_GUNC\_\_)

修改为

#define configUSE\_IDLE\_HOOK 0

#define configUSE\_TICK\_HOOK 0

#define configCHECK\_FOR\_STACK\_OVERFLOW 0

#define configUSE\_MALLOC\_FAILED\_HOOK 0

**创建任务**

Viod 函数名（void \* pvParameters）

{

While（1）

{

功能代码

}

}

TaskHandle\_t xHandlename // 定义任务句柄

xTaskCreate( (Ta skFunction\_t) 函数名

( char \* ) "test4", //函数标识，RTOS并不使用

( configSTACK\_DEPTH\_TYPE) 300, //堆栈大小300\*4byte

(void \* ) pvParameters, //函数参数

(UBaseType\_t) 9, // 优先级

(TaskHandle\_t \* ) & xHandlename //任务句柄

);

vTaskDelay(100); //系统延时(ms)

vTaskStartScheduler(); //启动调度器

freeRTOS中：

任务优先级-数值越大优先级越高，任务数量无限制

中断优先级-数值越小优先级越低

configLIBRARY\_LOWEST\_INTERRUPT\_PRIORITY M

configLIBRARY\_MAX\_SYSCALL\_INTERRUPT\_PRIORITY N

M为SVC和PendSv的中断优先级，一般为最低

N 为RTOS可管理(临界区)的最高中断优先级（实际响应顺序的最高优先级）

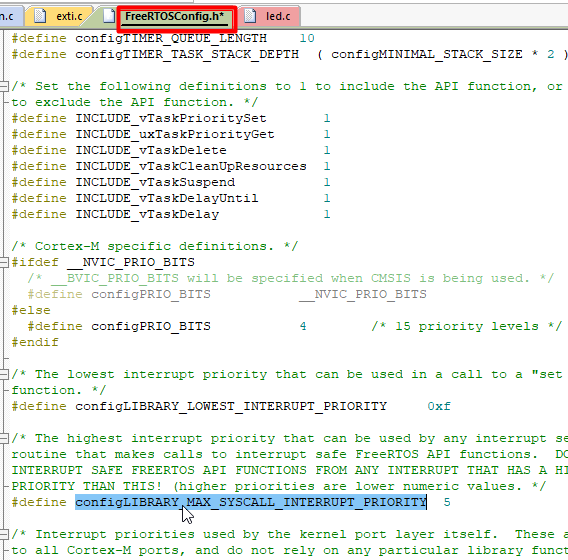
中断优先级数值：0--------------------------N-----------------------15（M）

不受RTOS管理(不可用FromISR) | 受RTOS管理(中断函数内可用FromISR)

===============================================================================

configMAX\_PRIORITIES N

最大任务优先级： 0-(N-1)级



**延时**

void Task()

{

    TickType\_t Time;

    Time=xTaskGetTickCount();//每个任务函数都需要初始化一次，并维护这个值

    While(1)

    {

    //任务代码

        vTaskDelayUntil(&Time,100/portTICK\_PERIOD\_MS);//周期执行,立即进入阻塞态，并在计数器到延时时间时切回就绪态

    }

}

vTaskDelayUntil()在进入阻塞后，如果有其他任务执行，则延时时间会变长

**任务优先级**

改变任务优先级 vTaskPrioritySet()

获取任务优先级 uxTaskPriorityGet()

**任务挂起和解挂**

参数：任务句柄

挂起任务：vTaskResume(xHandlename)

解挂任务：vTaskSuspend(xHandlename)

从中断里解挂任务：xTaskResumeFromISR(xHandlename)

注：此中断的优先级必须受RTOS管理（N--15）

**任务保护**

        taskENTER\_CRITICAL();//进入临界区，保证任务不会被切换或被中断打断

        任务代码

        taskEXIT\_CRITICAL();//退出临界区

有专门为中断函数设计的临界区函数

uint32\_t SaveInterruptState;

SaveInterruptState =taskENTER\_CRITICAL\_FROM\_ISR();保存中断状态

任务代码

taskEXIT\_CRITICAL\_FROM\_ISR(SaveInterruptState)恢复之前的状态

关于临界区的说明

只会关闭上下文的切换，不关闭中断

临界区执行时间太长时建议使用此操作，而不是使用临界区开关中断的方式

vTaskSuspendAll(); //挂起任务调度器

之间不能有FreeRTOS函数！！！！

xTaskResumeAll();//恢复任务调度器

临界区访问函数只会屏蔽和开启受FreeRTOS管理的中断，

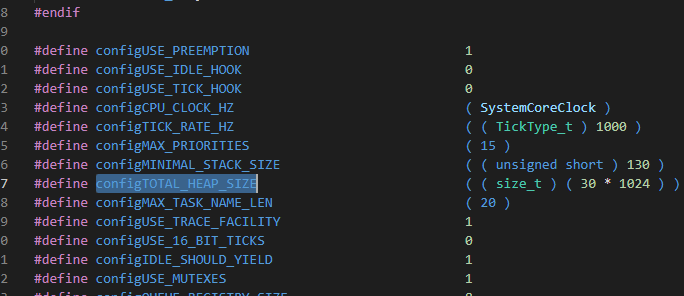
对于高于【configMAX\_SYSCALL\_INTERRUPT\_PRIORITY】中断函数内不能包含FreeRTOS函数

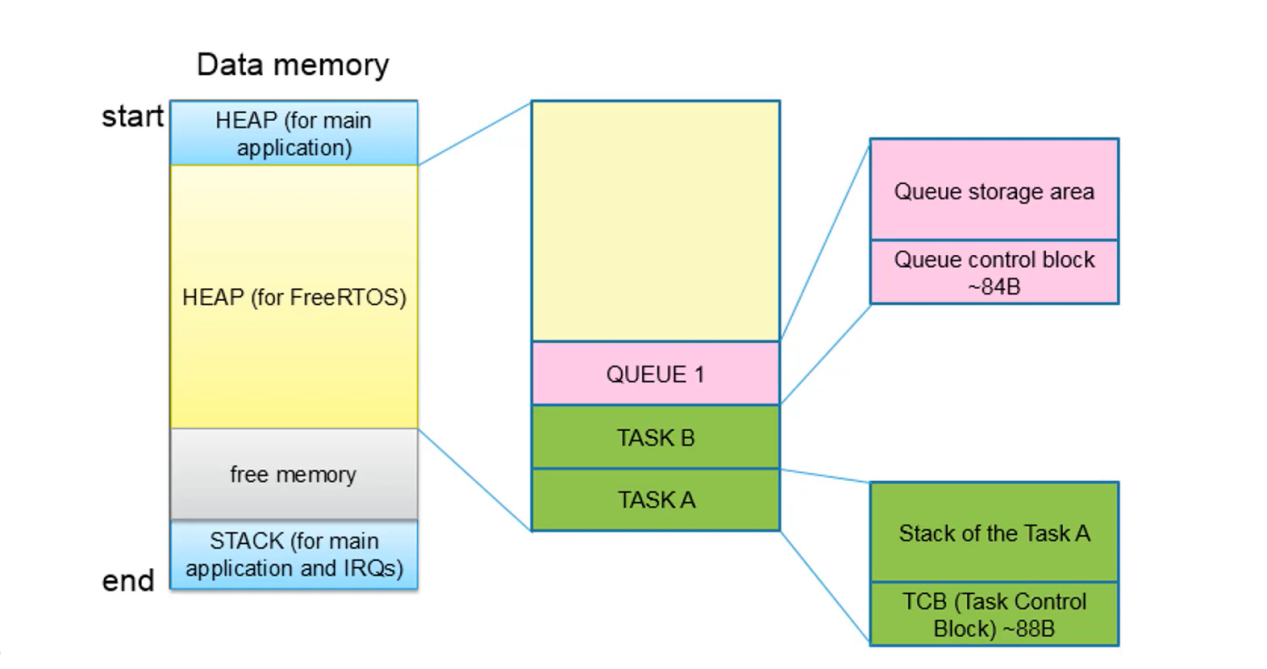
taskENTER\_CRITICAL();进入临界区

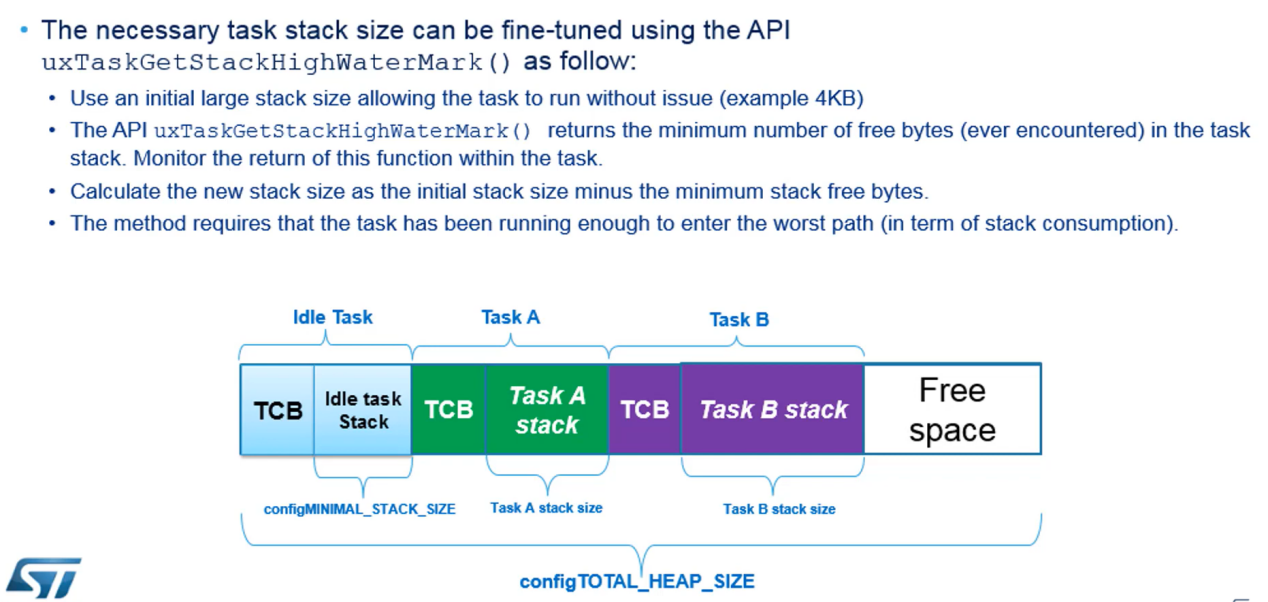
taskEXIT\_CRITICAL();退出临界区

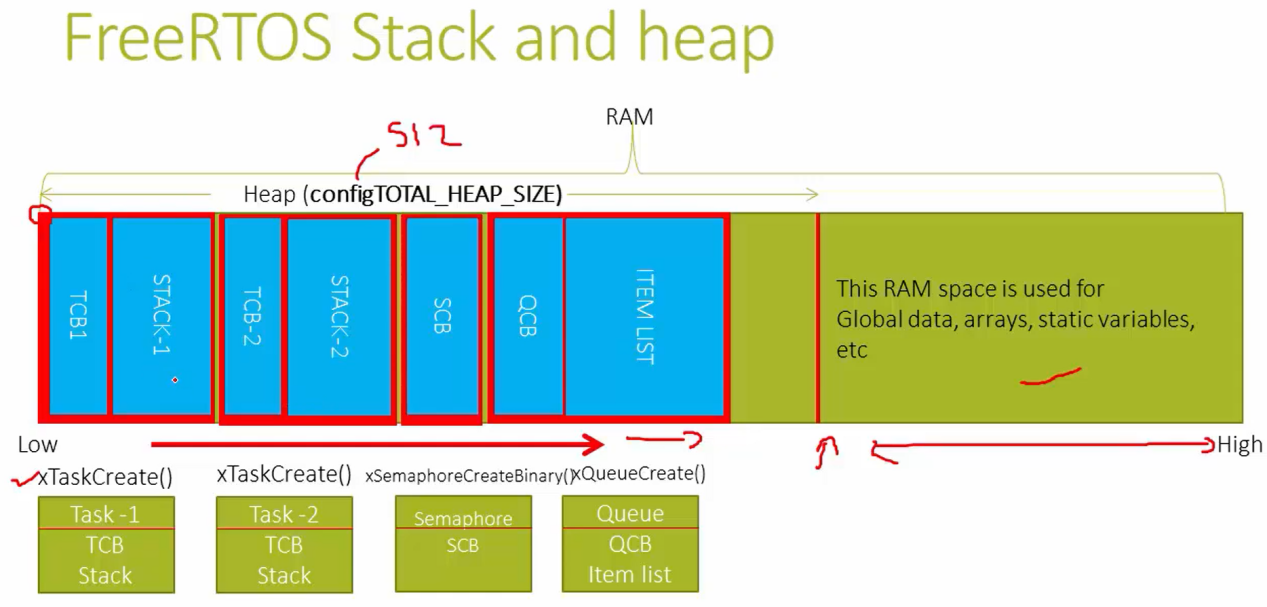
内存管理

Free RTOS所使用的所有内存(RTOS私有的堆和栈[黄色部分])都在下图中设置(单位：字节)

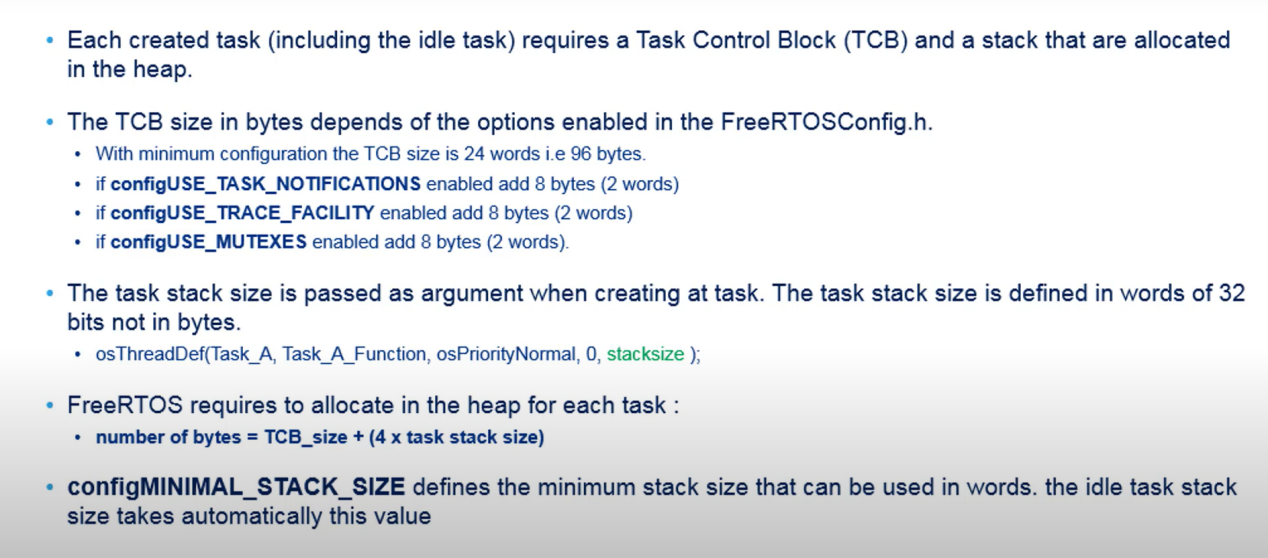








每个任务所使用的内存大小可以按下图估算，**一个任务总的堆大小(字节) = TCB+ usStackDepth\*4**



其中

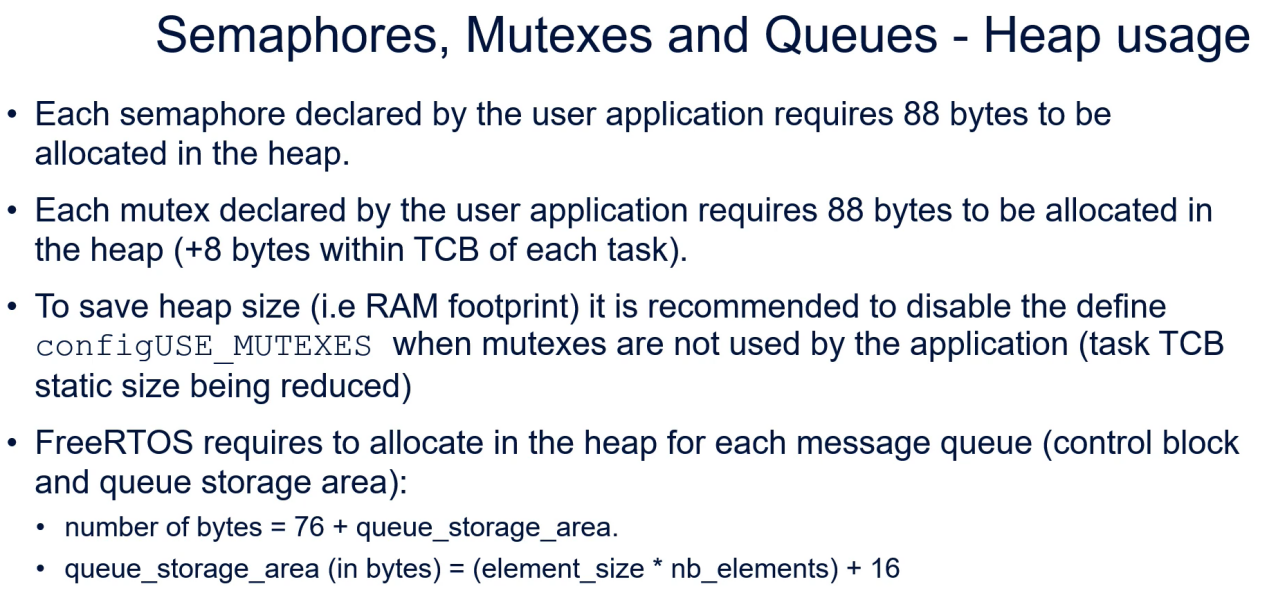
#define configMINIMAL\_STACK\_SIZE

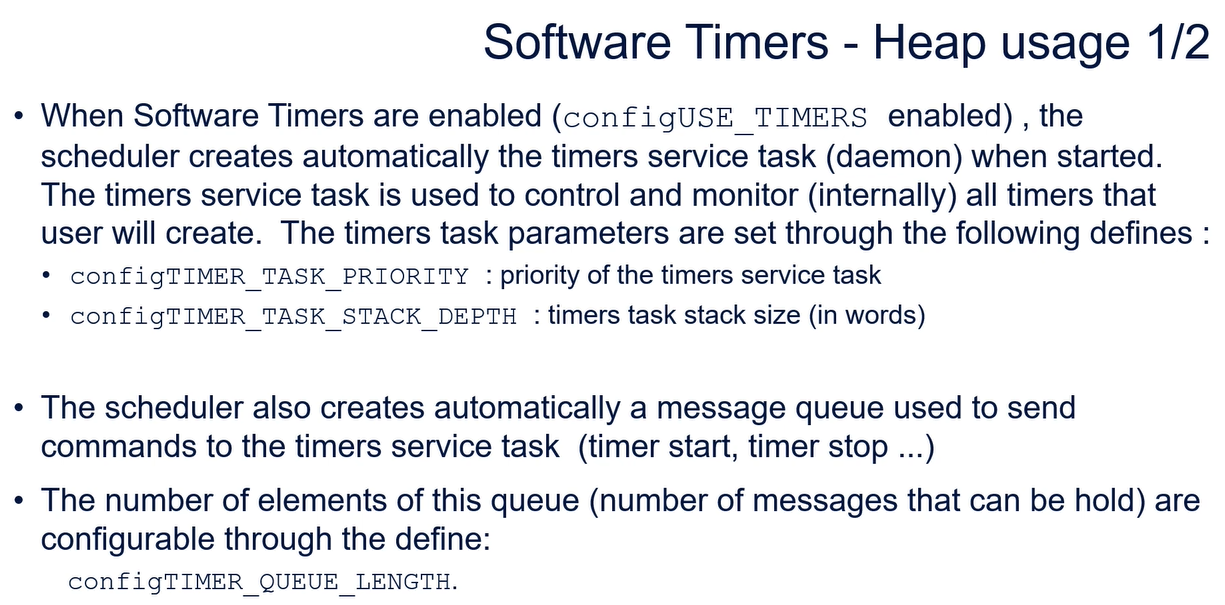
规定了每个任务的最小内存使用大小(单位：32bit(4字节))，空闲任务创建时自动使用这个大小，TCB的大小会随着使用组件的多少而增加。

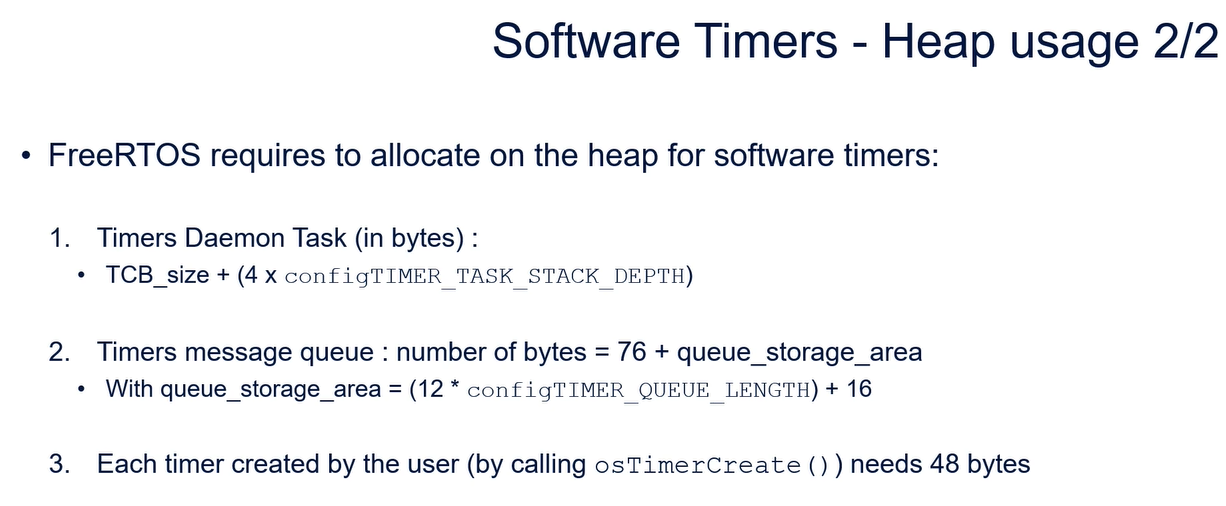
信号量，互斥量和队列的内存分配情况如下图

消息队列的大小可按下图估算

总字节数 = 76 + 数据个数\*单个数据大小+16

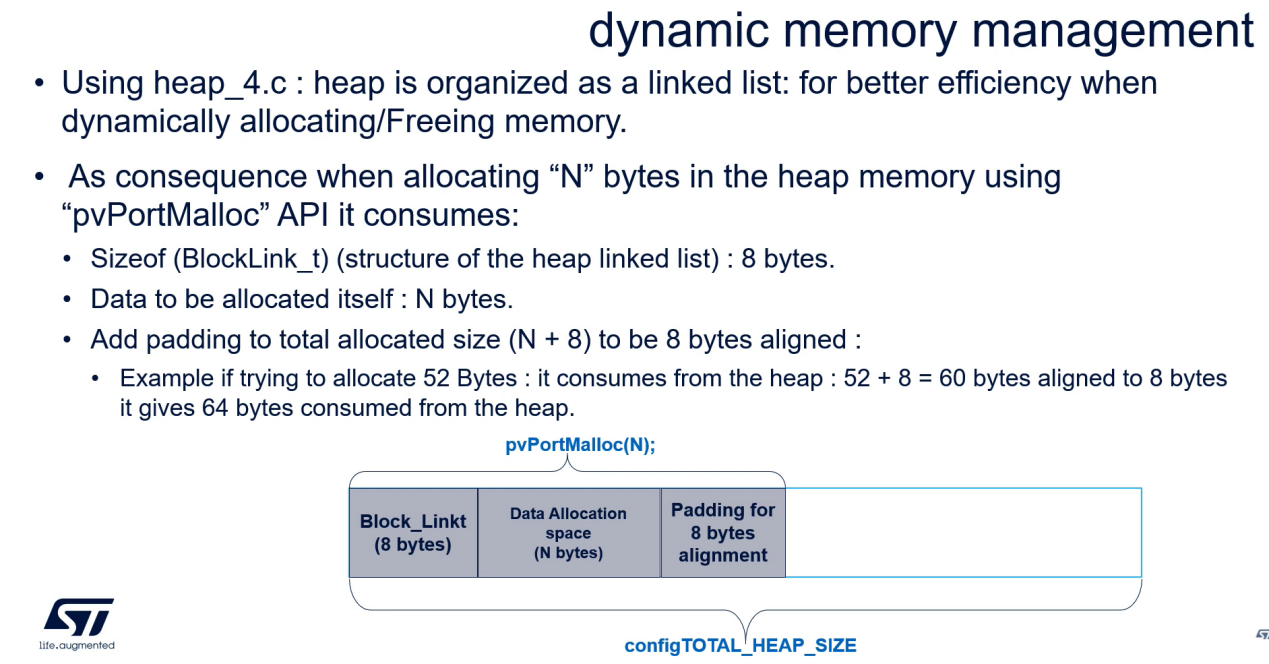




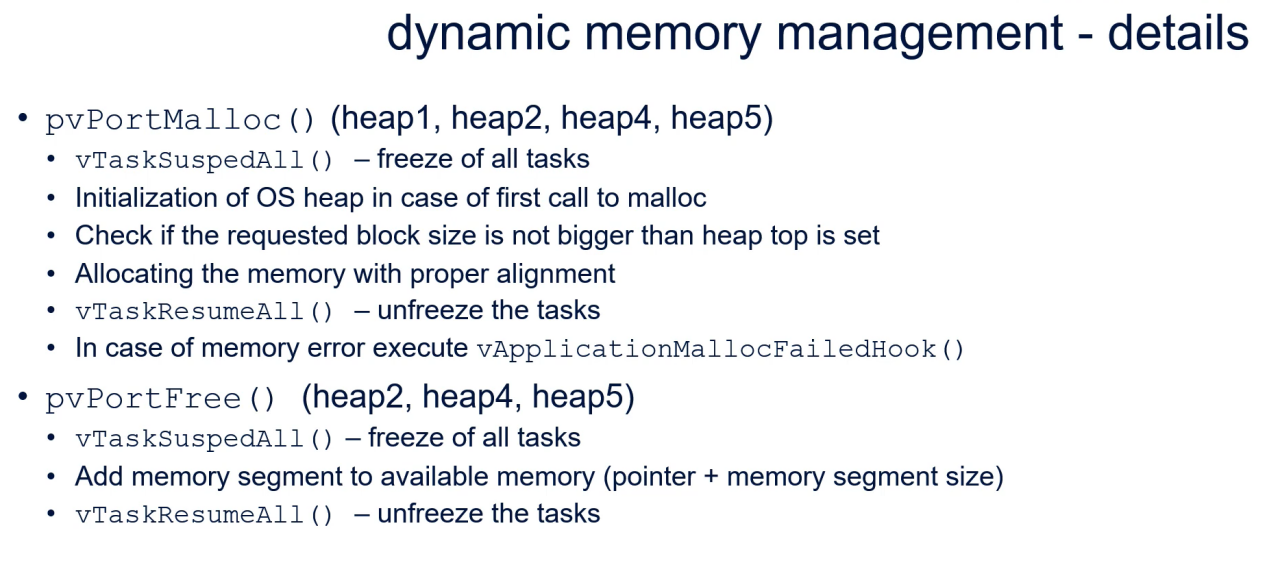


**当使用heap\_4时，每次申请的实际内存大小如下，链表(8 Byte)+申请的数据（N Byte）**

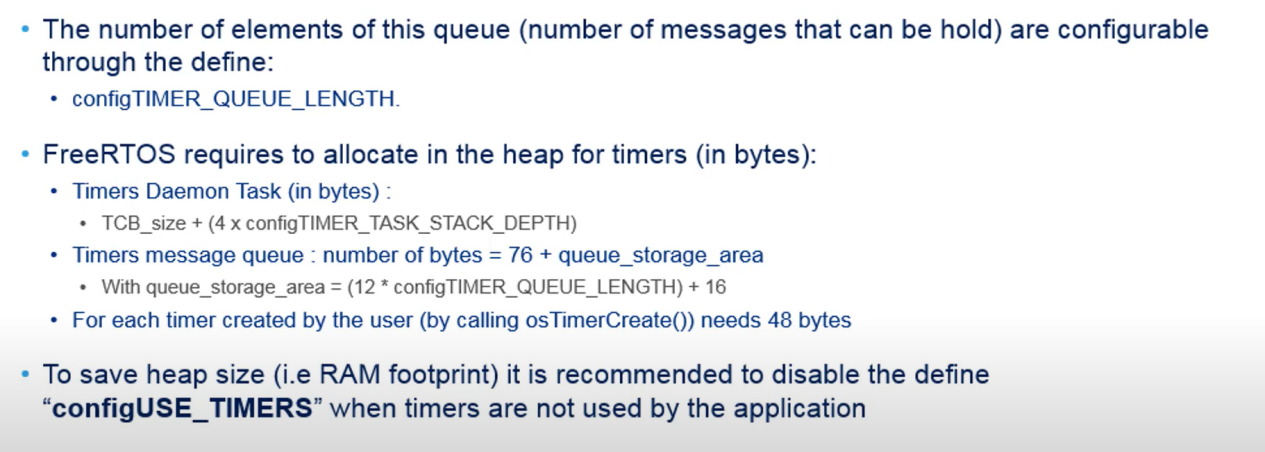
**且还要8字节对齐！！！！！**



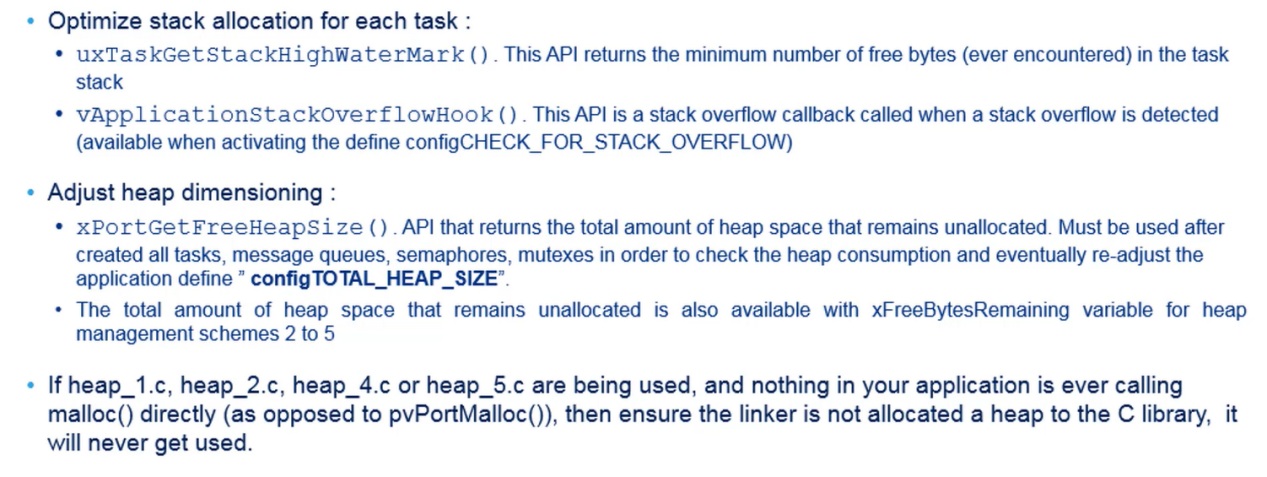
动态内存申请函数的执行过程介绍：



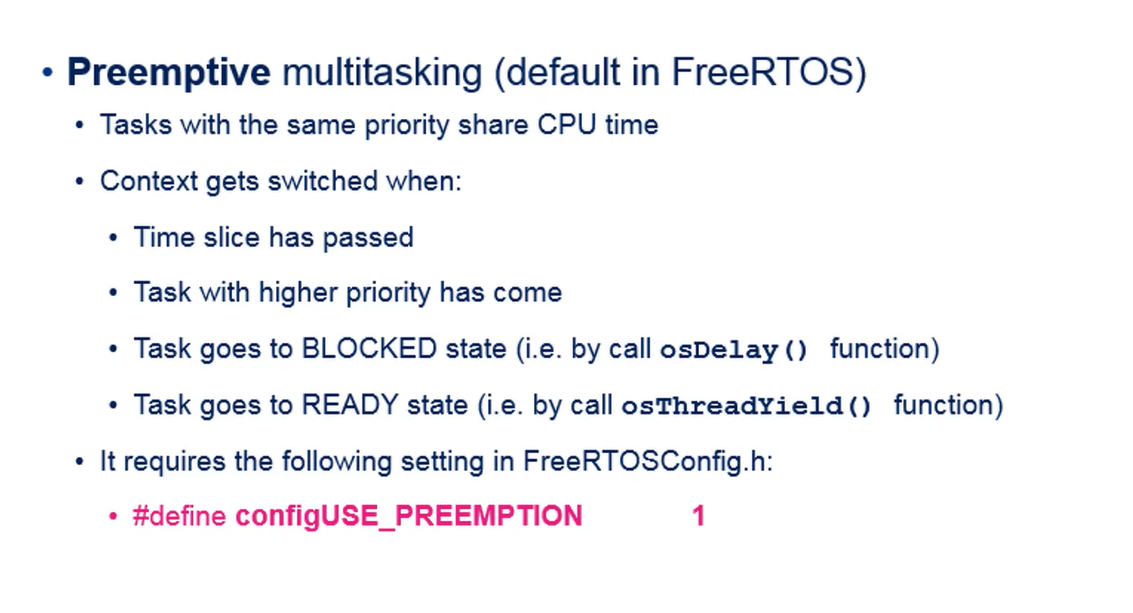
这部分看不懂（TIMER是啥？）



获取任务的内存使用情况可参考下图



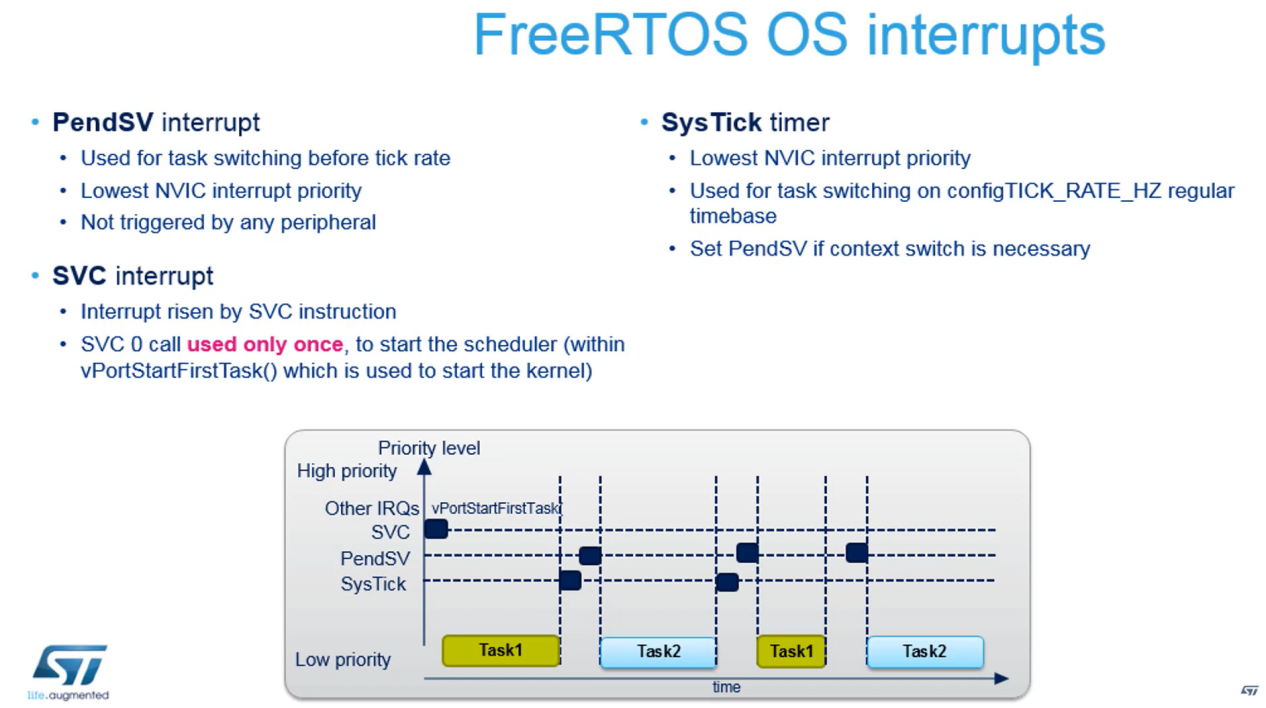
有关任务切换

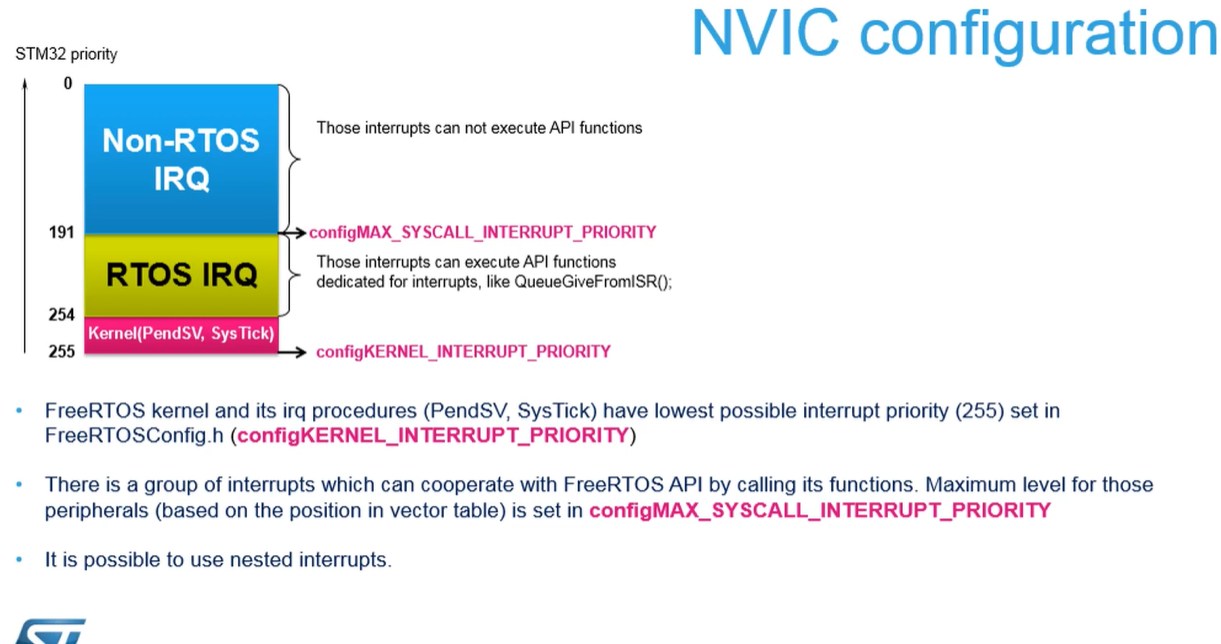


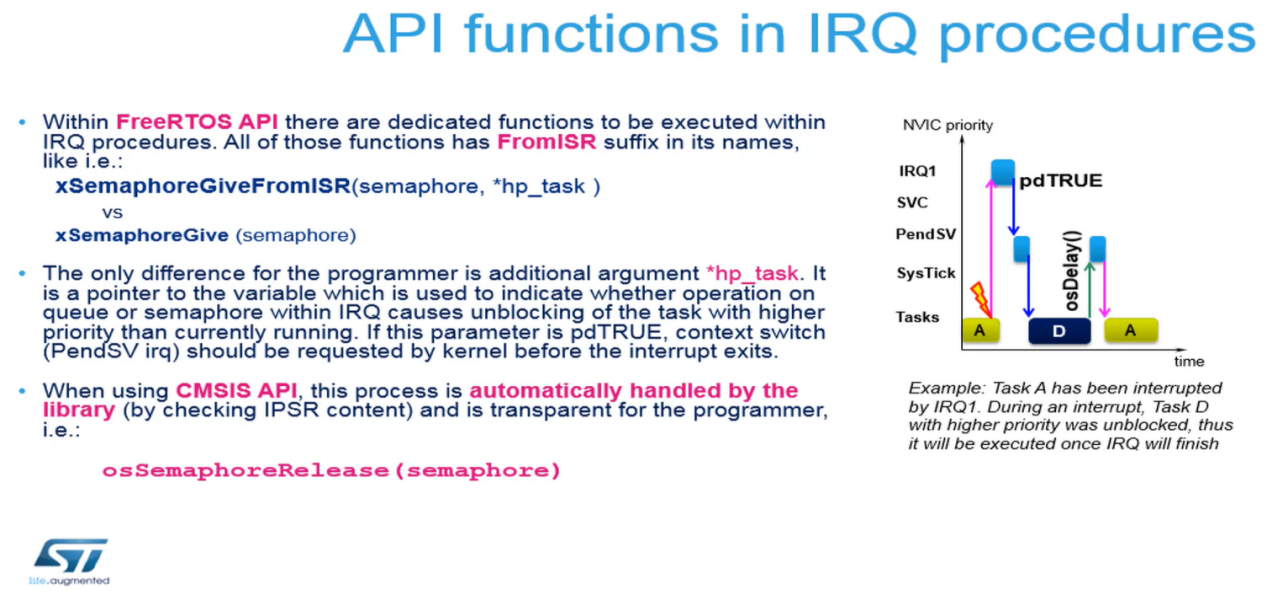


中断与RTOS

启动调度器时创建的第一个任务会调用SVC 0(仅在启动时使用1次)来启动内核





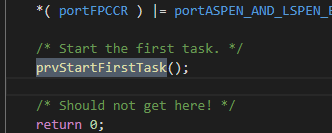


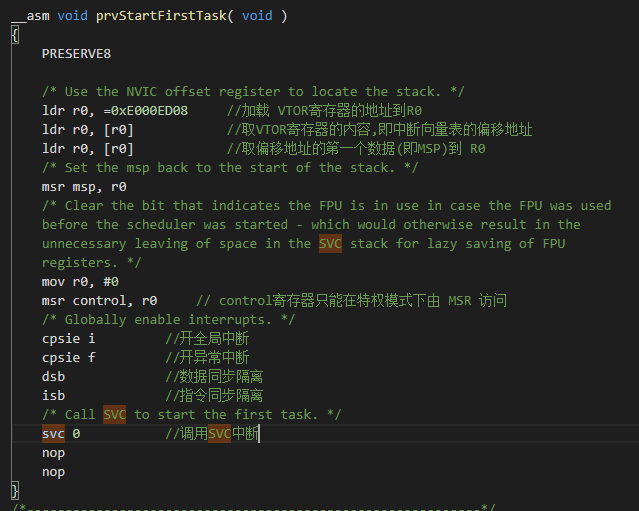
**关于SVC中断函数：**

FreeRTOS通过在启动任务调度器时，在创建的第一个任务中，通过

svc 0

的方式，来进入 vPortSVCHandler()





\_\_asm void vPortSVCHandler( void )

{

    PRESERVE8

    /\* Get the location of the current TCB. \*/

    ldr r3, =pxCurrentTCB   //取 pxCurrentTCB 的地址

    ldr r1, [r3]            //取 pxCurrentTCB 的内容到R1( 即 R1 = pxTopOfStack )  pxTopOfStack为一个指针变量！！！！

    ldr r0, [r1]            //取 pxTopOfStack 指向的内容，即 R0 = 栈地址

    /\* Pop the core registers. \*/

    ldmia r0!, {r4-r11, r14}

    msr psp, r0

    isb

    mov r0, #0

    msr basepri, r0

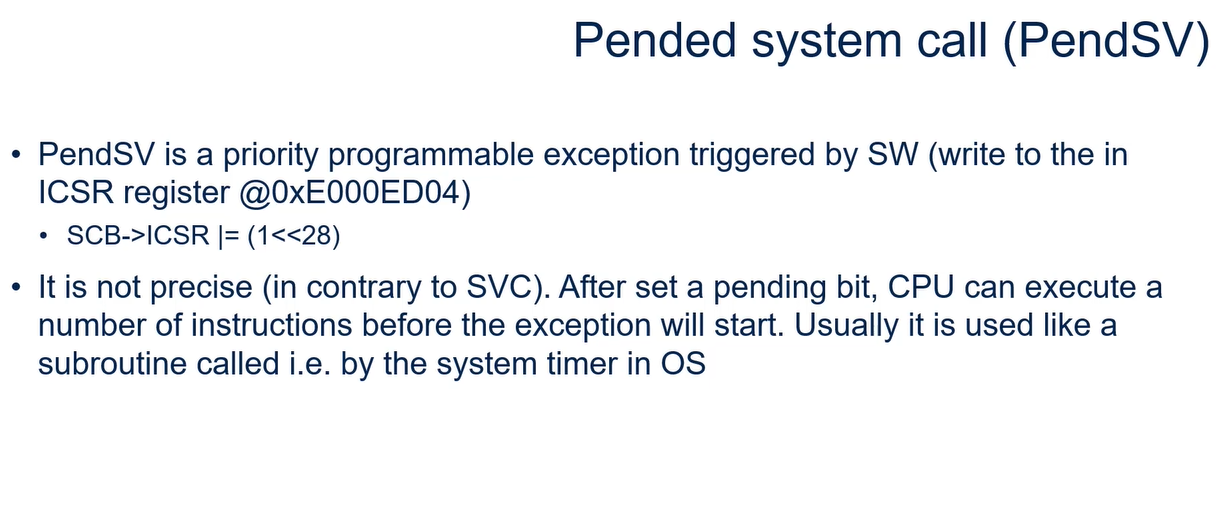
    bx r14

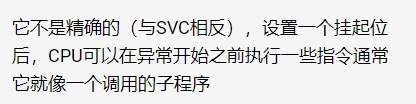
}

关于 PendSV中断函数

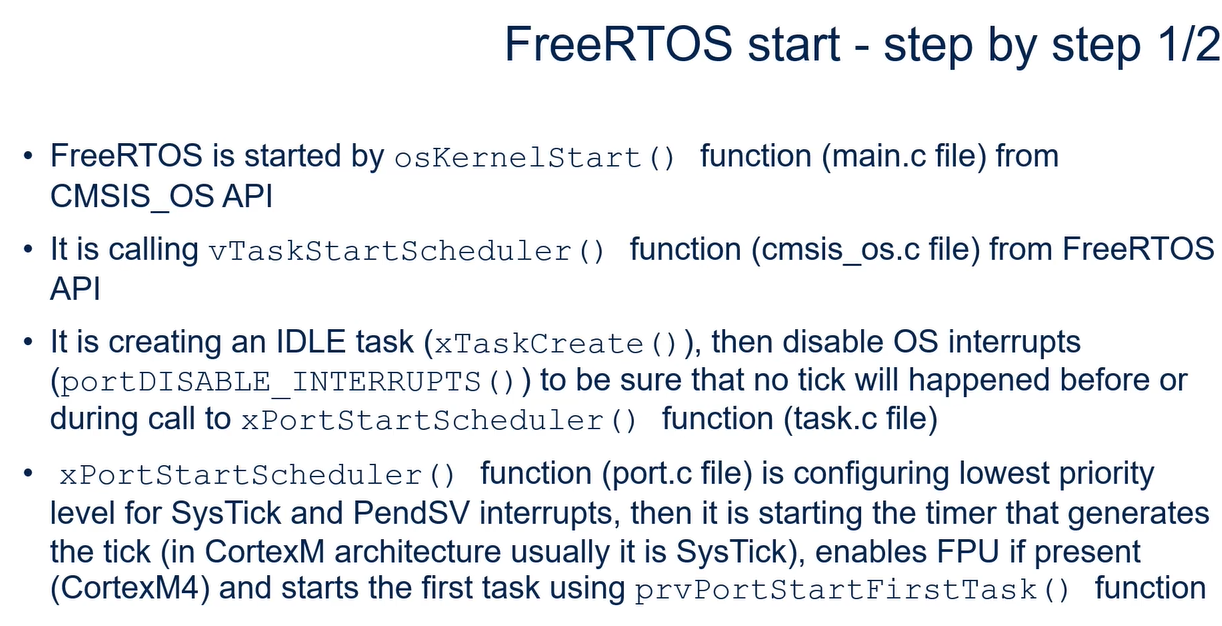
Free RTOS通过向以下寄存器(ICSR)置位

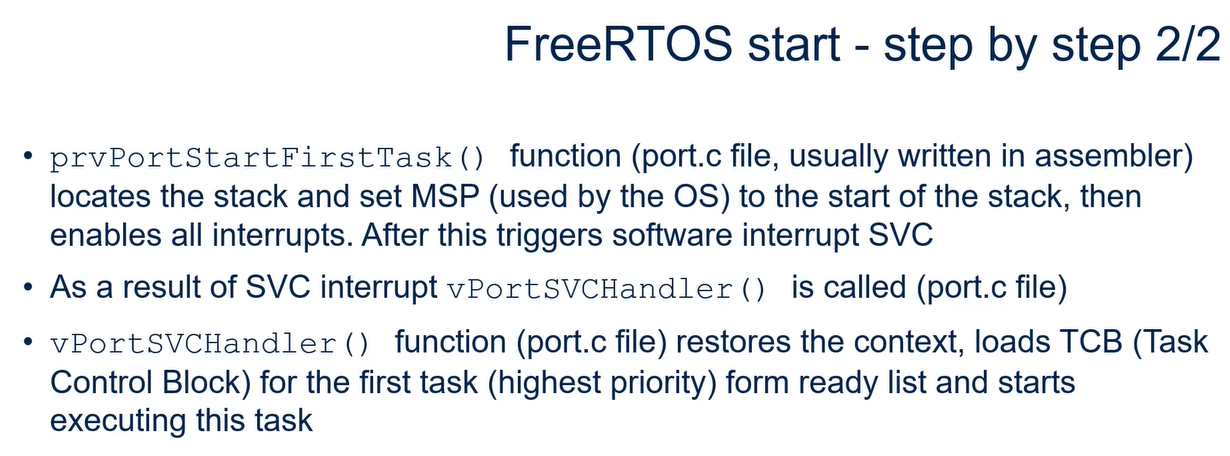
portNVIC\_INT\_CTRL\_REG





**FreeRTOS启动过程：**



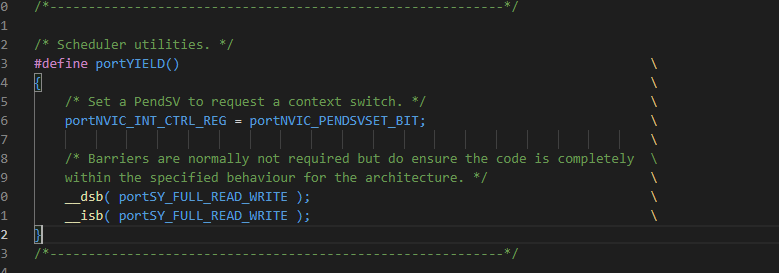


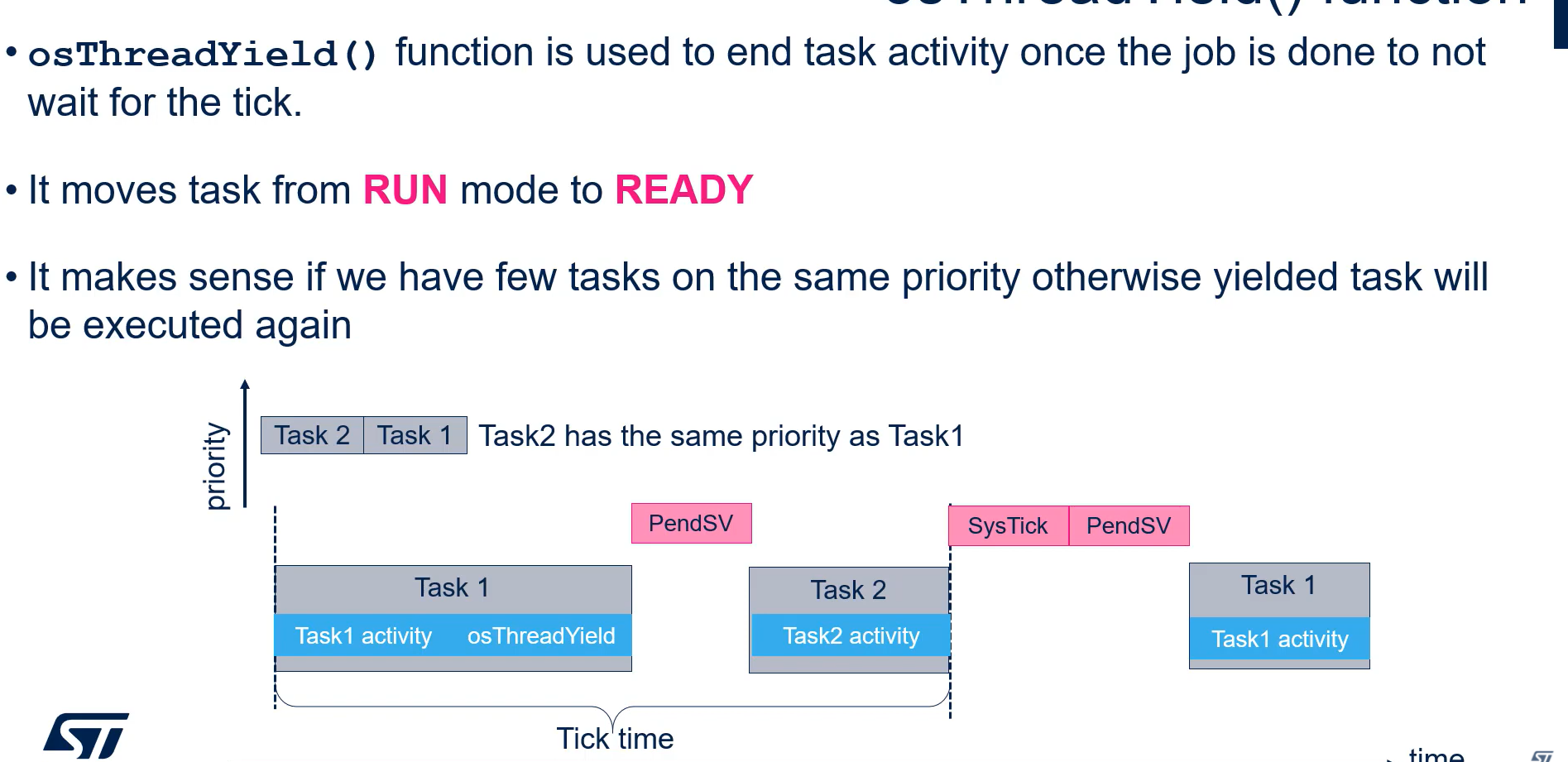
知识点：

优先级反转

IDLE会使任务进入READY状态，而不是阻塞状态！！

Idle会触发一次PendSV,进行上下文切换！！！！





信号量

二值信号量：

【1】创建二值信号量

 SemaphoreHandle\_t xSemaphore = NULL;

    xSemaphore = xSemaphoreCreateBinary();

    if( xSemaphore != NULL )

    {

       //创建成功

    }

给出(Give操作) 和 取走(Take操作)是一对互补操作，Give使信号量有效，Take取走信号量然后使信号量失效。

【2】Take二值信号量

    //获取信号量，如果无效，则阻塞Times时长来等待信号量有效

    if( xSemaphoreTake( xSemaphore, ( TickType\_t ) Times ) )

    {

        //获取信号量成功！！

    }

【3】Give二值信号量

void \_\_interrupt xxxIRQ\_Handler( void )

{

   static portBASE\_TYPE xHigherPriorityTaskWoken;

   xHigherPriorityTaskWoken = pdFALSE; //初始状态设置为FALSE

    // xHigherPriorityTaskWoken 用于指示【被中断的函数】 和 【被信号量阻塞的任务】的优先级比较结果

    // 当【被信号量解除阻塞的任务优先级】 高于 【被中断的函数】 的任务时， xHigherPriorityTaskWoken 会被设置为TURE

  xSemaphoreGiveFromISR( xBinarySemaphore,&xHigherPriorityTaskWoken );

    if( xHigherPriorityTaskWoken == pdTRUE )

    {

       给出信号量以使得等待此信号量的任务解除阻塞。如果解出阻塞的任务的优先级高于当前任务的优先级 – 强制进行一次任务切换，以确保中断直接返回到解出阻塞的任务(优选级更高)。

        vTaskSwitchContext();

    }

}

xSemaphoreGive( xSemaphore )//用于非中断函数中

xSemaphoreGiveFromISR( xSemaphore,&xHigherPriorityTaskWoken ); //IRQ中

https://www.youtube.com/watch?v=hbRcjzmv0wY&list=PLnMKNibPkDnFeFV4eBfDQ9e5IrGL\_dx1Q&index=10