1.简介

- 支持抢占式,协作式和时间片调度
- 通常占用4~9k固件
- 任务和任务,任务和中断之间可使用任务同时,消息队列,二值信号量,数值信号量,递归互斥信号量和互斥信号量进行通信和同步
- 具有优先级继承特性的互斥信号量
- 高效软件定时器
- 堆栈溢出检测功能
- 任务数量,优先级不限
- 内核时钟优先级和PendSV优先级最低,给外设ISR提供保障

freertos允许使用`INCLUDE_`宏和`config`宏来进行配置裁剪

freertos的堆栈检测功能:

```
#define configCHECK_FOR_STACK_OVERFLOW 1
//堆栈溢出是导致程序不稳定的主要因素,FreeRTOS提供了两种可选机制来帮助检测和调试堆栈溢出,都得设置该宏
//如果使能则需要提供一个钩子函数(回调函数),当内存检测到堆栈溢出后就会调用,函数原型:
void vApplicationStackOverflowHook(TaskHandle_t xTask,char* pcTaskName);
//堆栈溢出如果太严重可能会损毁这两个参数,这种情况需要查看变量pxCurrentPCB 来确定哪个任务发生了堆栈溢出,有的处理器可能会有fault中断来提示这个错误,但堆栈溢出检测会增加上下文切换开销,同样只能在调试时使用
/*----该宏为1时使用方法1----*/
//上下文切换的时候需要保存现场,现场是保存在堆栈中的,这个时候任务堆栈使用率很可能达到最大值,方法一就是不断的检测任务堆栈指针是否指向有效空间,否则调用钩子函数,优点就是快,但不能拿检测所有的
/*----该宏为3时使用方法2----*/
//这个方法会在创建任务的时候向任务堆栈填充一个已知的标记值,它会检测栈后面的几个标记值是否被改写,如果被改写了则调用函数,方法2几乎能检测到所有的堆栈溢出,但也有检测不到,例如溢出值和标记值相同的时候
```

2.M核SVC和PendSV任务切换模型

CortexM核寄存器组:

- r0~r12通用寄存器组, r13(xSP MSP/PSP), r14(LR)保存异常返回标志, 包括返回后进入任务模式还是处理模式、使用PSP还是MSP, r15(PC)
- 程序状态寄存器组(PSRs/xPSR)
- 中断屏蔽寄存器组(PRIMAST, FAULTMAST, BASEPRI)
- 控制寄存器(CONTROL)

它们只能被专用的MSR/MRS指令访问，而且它们也没有与之相关联的访问地址。如：
MRS <gp_reg>, <special_reg> ; 读特殊功能寄存器的值到通用寄存器
MSR <special_reg>, <gp_reg> ; 写通用寄存器的值到特殊功能寄存器

`SVC`是系统服务调用，由`SVC`指令触发调用

freertos使用 **SVC指令**是在 **prvPortStartFirstTask** 函数中使用，目的是开始第一次调度，发生在创建任务完成后的第一次调度

```
#       vTaskStartScheduler     --->
#--->   xPortStartScheduler     --->
#--->   prvPortStartFirstTask
__asm void prvStartFirstTask( void )
{
    PRESERVE8
    /*
     *  - 0xE000ED08是 SCB 模块 VTOR 寄存器的地址
     *  - 这句汇编目的是把 0xE000ED08(VTOR的地址) 保存到 R0 寄存器
     *  - 执行完这句指令(伪指令)，后 R0 寄存器保存的是 0xE000ED08(VTOR的地址)
     */
    /* Use the NVIC offset register to locate the stack. */
    ldr r0, =0xE000ED08
    /*
     *  - 将 0xE000ED08 地址的值(中断向量表的地址) 放到 R0 寄存器中
     *  - 执行完这句指令，后 R0 寄存器保存的是 中断向量表的地址
     */
    ldr r0, [r0]
    /*
     *  - 将 中断向量表 第一项的值 放到 R0 寄存器中
     *  - 中断向量表第一个字保存的 是栈顶地址(cortex-m是满减栈，栈顶即栈的开始)
     *  - 初始转态下使用的是MSP
     *  - 执行完这句指令，后 R0 寄存器保存的是栈顶地址
     */
    ldr r0, [r0]
    /*
     *  - 将 R0的值写入 MSP 中。
     *  - 此时 R0 内存储的是 栈顶地址，这一步其实就是把 MSP 中已经存好的内容完全销毁。
     *  - 在这段代码中,在执行完SVC指令后,会由调度器完全接管代码,这里的SVC异常永远也不会返回
     */
    /* Set the msp back to the start of the stack. */
    msr msp, r0
    /* Globally enable interrupts. */
    cpsie i /* 开启全局中断 */
    cpsie f /* 开启fpu */
    dsb /* 数据同步隔离,确保上面的配置生效 */
    isb /* 指令同步隔离,清空流水线 */
```

```
    /* Call SVC to start the first task. */
    svc 0 /* 触发0号系统调用，永远也不会返回， 从此处开始代码由freertos全面接管
*/
    nop /* svc 不会返回，永远也不会运行到这里 */
    nop
}
__asm void vPortSVCHandler( void )
{
    PRESERVE8
        /*
         * - 将要执行任务的结构体 的 指针的指针(二重指针) 保存到 R3寄存器
        */
    ldr r3, =pxCurrentTCB   /* Restore the context. */
         /*
         * - 将要执行任务的结构体 的 指针(一重指针) 保存到 R1寄存器
         * - 也可以理解为将 要执行任务的结构体 的第一个元素的地址保存到 R1 寄存
器
         * - 结构体第一个元素 保存将要执行任务的栈指针
         * - 所以这一句是将 栈指针(SP) 的地址(是栈指针的存放地址，而不是栈指针)
放到R1中。
         */
    ldr r1, [r3]            /* Use pxCurrentTCBConst to get the
pxCurrentTCB address. */
        /*
         * - 将栈指针 SP 保存到 R0寄存器
         */
    ldr r0, [r1]            /* The first item in pxCurrentTCB is the task
top of stack. */
        /*
         * - 将栈指针中的地址按顺序弹栈(要执行的任务的栈)到{r4-r11}
         * - 这里是要弹栈的内容是 创建任务时 伪造的上下文
         */
    ldmia r0!, {r4-r11}     /* Pop the registers that are not automatically
saved on exception entry and the critical nesting count. */
    /* update the new stack pointer to psp */
    msr psp, r0             /* Restore the task stack pointer. */
    isb /* 指令同步隔离,清空流水线 */
    /* 将basepri寄存器设为0,打开所有中断 */
    mov r0, #0
    msr basepri, r0
        /*
         * - 异常返回，这时r14(LR)中存放的是一个 EXC_RETURN 值，这个值是在创建
任务时伪造的。
         * - EXC_RETURN 的值决定了异常返回后使用的栈（即任务要使用的栈），创建
任务时伪造EXC_RETURN一定要注意：
         *  .bit2 一定要设置为1，要任务返回PSP
         *  .从 "msr psp, r0 " 这句汇编也能看出来，freertos希望任务栈都使用
PSP.
         */
    orr r14, #0xd
    bx r14
    /* return , cpu load (psp update) r0~r3 r12 psr pc r14 automatically */
```

```
    /* r14 in here is null , the r15(pc) will be recovered to pc in cpu */
}
```

PendSV中断, 可悬起的系统调用。两者不同之处在于响应速度

`SVC`中断是要求被立刻得到响应的，而`PendSV`中断则可以延迟被响应，也就是`PendSV`中断可以先“悬起”，待其它重要中断被执行完成后再处理它

`PendSV` 中断的意义在于当需要发生任务切换时，如果当前正在执行一个中断，则等待中断执行完毕后，再进行任务切换。即不允许打断中断来切换任务。

在PendSV异常保存上文时，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，并在异常返回后进入任务模式、返回 Thumb 状态。然后将R14压入任务堆栈，以便下次切换回该任务时获取R14的内容

PendSV异常切换到下文时会将新任务堆栈中的R14读取出来，由于上文将R14做了保存，此时R14=0xFFFFFFFD,表示异常返回后进入任务模式，使用PSP作为堆栈指针出栈,出栈完毕后PSP指向任务栈栈顶。

PendSV 主要进行任务上下文切换:

`上文保存内容`:

- 寄存器的r4~r11（其余的进入异常前CPU自动保存）
- 当前任务栈顶指针
- 全局TCB地址

`下文切换内容`:

- 新任务TCB
- 新任务r4~r11
- 新任务栈顶存入PSP

主要流程为:

1. 产生pendsv中断，cpu自动将psr,pc,lr,r12,r3~r0入栈保护(sp使用psp)
2. 进入pendsv的isr,使用msp
3. 通过异常之前的psp指针把cpu r11~r4寄存器入栈保护
4. 进入临界区
5. 调用函数vTaskSwitchContext找到下一个需要运行的任务,pxCurrentTCB指向需要运行的任务TCB
6. 退出临界区
7. 获取新任务堆栈地址,通过新堆栈地址,将新任务栈中数据出栈到r11~r4
8. 更新psp指针,使之指向新任务栈
9. 调用bl r14指令,退出中断,sp使用psp,cpu根据新的psp自动将新的任务pse pc lr r12 r3~r0数据出栈,
   执行新任务

```
__asm void xPortPendSVHandler( void )
{
    extern uxCriticalNesting;
    extern pxCurrentTCB;
    extern vTaskSwitchContext;

    PRESERVE8

    /*将psp值放到r0,此时sp得值为msp*/
    mrs r0, psp
    isb

    /* 获取当前任务的栈顶 */
    ldr r3, =pxCurrentTCB       /* Get the location of the current TCB. */
    /* r2保存TCB的首地址 */
    ldr r2, [r3]

    /* 将当前的r4~r11按顺序压栈到当前任务的psp栈中 */
    stmdb r0!, {r4-r11}         /* Save the remaining registers. */
    /* 最后将当前TCB的首地址(栈顶地址)保存到当前任务的psp栈中 */
    str r0, [r2]                /* Save the new top of stack into the first
member of the TCB. */

    stmdb sp!, {r3, r14} /*入栈保存R3(即&pxCurrentTCB)和 R14(LR) */
    mov r0, #configMAX_SYSCALL_INTERRUPT_PRIORITY
    msr basepri, r0 /* 关闭中断 */
    dsb
    isb
    bl vTaskSwitchContext /* 在临界段切换就绪队列中优先级最高的任务，更新
pxCurrentTCB */
    mov r0, #0
    msr basepri, r0 /* 打开中断 */
    ldmia sp!, {r3, r14} /* 从主堆栈中恢复寄存器R3和R14的值,此时SP使用的是MSP
*/

    ldr r1, [r3]  /* 将R3存放的pxCurrentTCB的地址赋予R1 */
    ldr r0, [r1]    /* The first item in pxCurrentTCB is the task top of
stack. */
    /* 将r4-f11出栈，将新任务的r4-r11恢复到内核的r4-r11 */
    ldmia r0!, {r4-r11} /* Pop the registers and the critical nesting
count. */
    msr psp, r0 /* 更新psp，使其指向任务堆栈 */
    isb
    bx r14 /* 退出中断，开始使用PSP堆栈指针，硬件自动将r0~r3，r12，lr，pc，
xPSR出栈 */
    nop
}
```

3.内存管理

FreeRTOS提供的内存管理都是从内存堆中分配内存的。默认情况下，FreeRTOS内核创建任务、队列、信号量、事件组、软件定时器都是借助内存管理函数从内存堆中分配内存, 最新的FreeRTOS版本（V9.0.0及其以上版本）可以完全使用静态内存分配方法，也就是不使用任何内存堆。

对于heap_1.c、heap_2.c和heap_4.c这三种内存管理策略，内存堆实际上是一个很大的数组，定义为

```
static unsigned char ucHeap[ configTOTAL_HEAP_SIZE ];
//宏configTOTAL_HEAP_SIZE用来定义内存堆的大小，这个宏在FreeRTOSConfig.h中设置
```

3.1heap_1

这是5个内存管理策略中最简单的一个，它简单到只能申请内存, 不能释放内存,对于大多数嵌入式系统，特别是对安全要求高的嵌入式系统，这种内存管理策略很有用,因为对系统软件来说，逻辑越简单越容易兼顾安全。实际上，大多数的嵌入式系统并不需要动态删除任务、信号量、队列等，而是在初始化的时候一次性创建好，便一直使用，永远不用删除。所以这个内存管理策略实现简洁、安全可靠，使用的非常广泛

可以将heap_1内存管理看作是切面包：初始化的内存就像一根完整的长棍面包，每次申请内存，就从一端切下适当长度的面包返还给申请者，直到面包被分配完毕,就这么简单

heap_1内存管理策略使用两个局部静态变量来跟踪内存分配，变量定义为

```
static size_t xNextFreeByte = ( size_t ) 0;
    //记录已经分配的内存大小，用来定位下一个空闲的内存堆位置。因为内存堆实际上是一个大数组，我们只需要知道已分配内存的大小，就可以用它作为偏移量找到未分配内存的起始地址。变量xNextFreeByte被初始化为0，然后每次申请内存成功后，都会增加申请内存的字节数目。
static uint8_t *pucAlignedHeap = NULL;
    //指向对齐后的内存堆起始位置。为什么要对齐？这是因为大多数硬件访问内存对齐的数据速度会更快。为了提高性能，FreeRTOS会进行对齐操作，不同的硬件架构对齐操作也不尽相同，对于Cortex-M3架构，进行8字节对齐
```

内存申请: pvPortMalloc()

```
void *pvPortMalloc( size_t xWantedSize )
{
void *pvReturn = NULL;
static uint8_t *pucAlignedHeap = NULL;


    /* 确保申请的字节数是对齐字节数的倍数 */
    #if( portBYTE_ALIGNMENT != 1 )
    {
        if( xWantedSize & portBYTE_ALIGNMENT_MASK )
        {
            xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize &
portBYTE_ALIGNMENT_MASK ) );
```

```
        }
    }
    #endif

    vTaskSuspendAll();
    {
        if( pucAlignedHeap == NULL )
        {
            /* 第一次使用,确保内存堆起始位置正确对齐 */
            pucAlignedHeap = ( uint8_t * ) ( ( ( portPOINTER_SIZE_TYPE )
&ucHeap[
portBYTE_ALIGNMENT ] ) & ( ~( ( portPOINTER_SIZE_TYPE )
portBYTE_ALIGNMENT_MASK )
) );
        }


        /* 边界检查,变量xNextFreeByte是局部静态变量,初始值为0 */
        if( ( ( xNextFreeByte + xWantedSize ) < configADJUSTED_HEAP_SIZE )
&&
            ( ( xNextFreeByte + xWantedSize ) > xNextFreeByte ) )
        {
            /* 返回申请的内存起始地址并更新索引 */
            pvReturn = pucAlignedHeap + xNextFreeByte;
            xNextFreeByte += xWantedSize;
        }
    }
    ( void ) xTaskResumeAll();

    #if( configUSE_MALLOC_FAILED_HOOK == 1 )
    {
        if( pvReturn == NULL )
        {
            extern void vApplicationMallocFailedHook( void );
            vApplicationMallocFailedHook();
        }
    }
    #endif

    return pvReturn;
}
```

函数一开始会将申请的内存数量调整到对齐字节数的整数倍，所以实际分配的内存空间可能比申请内存大。

比如对于8字节对齐的系统，申请11字节内存，经过对齐后，实际分配的内存是16字节（8的整数倍）。

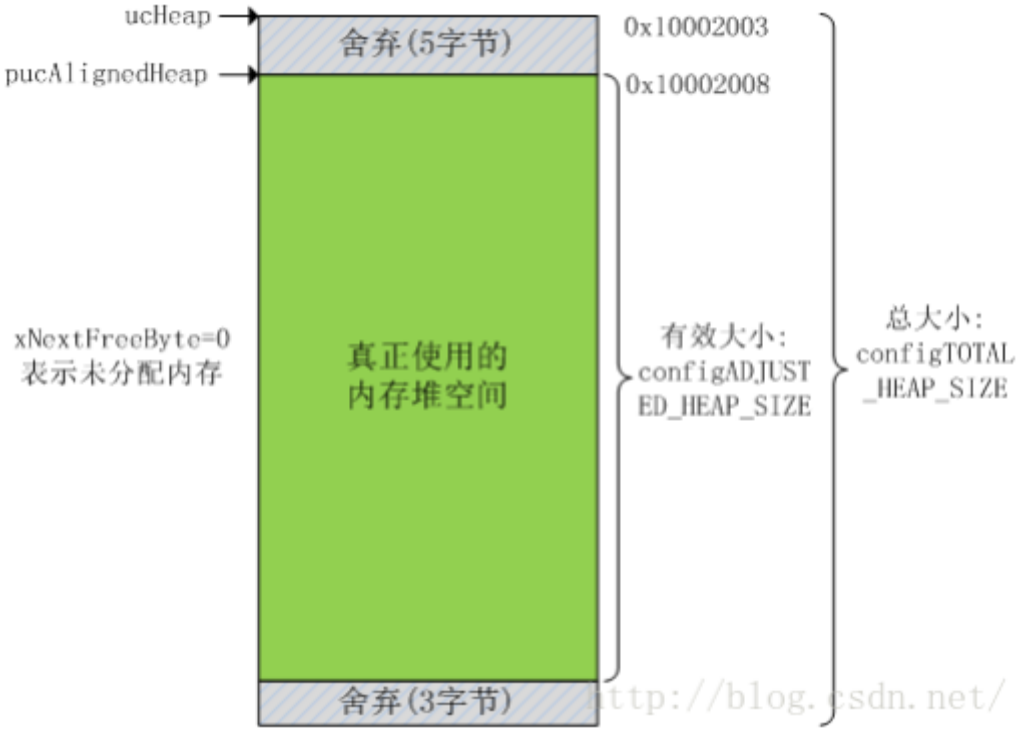
接下来会挂起所有任务，因为内存申请是不可重入的（使用了静态变量）。

如果是第一次执行这个函数，需要将变量pucAlignedHeap指向内存堆区域第一个地址对齐处。

内存堆其实是一个大数组，编译器为这个数组分配的起始地址是随机的，可能不符合我们的

对齐需要，这时候要进行调整。

比如内存堆数组ucHeap从RAM地址0x10002003处开始，系统按照8字节对齐，则对齐后的内存堆如下图所示

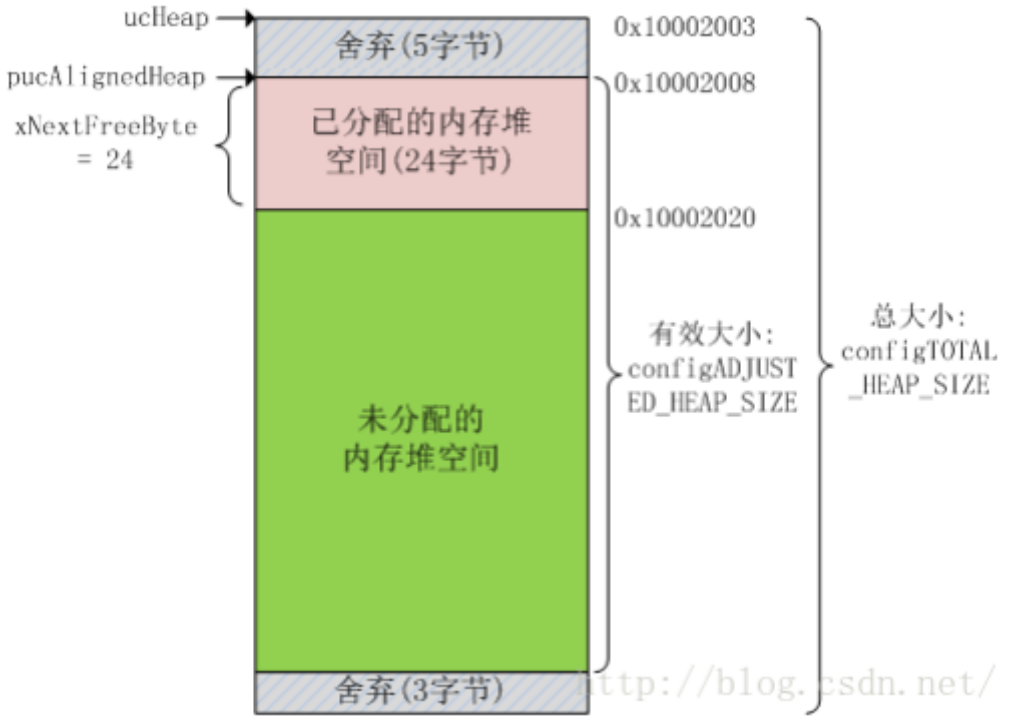


之后进行边界检查，查看剩余的内存堆是否够分配，检查xNextFreeByte + xWantedSize是否溢出。

如果检查通过，则为申请者返回有效的内存指针并更新已分配内存数量计数器xNextFreeByte（从指针pucAlignedHeap开始，偏移量为xNextFreeByte处的内存区域为未分配的内存堆起始位置）。

比如我们首次调用内存分配函数pvPortMalloc(20)，申请20字节内存。

根据对齐原则，我们会实际申请到24字节内存，申请成功后，内存堆示意图如下图所示



内存分配完成后，不管有没有分配成功都恢复之前挂起的调度器。

如果内存分配不成功，这里最可能是内存堆空间不够用了，会调用vApplicationMallocFailedHook()。

这个钩子函数由应用程序提供，通常我们可以打印内存分配设备信息或者点亮也故障指示灯。

获取当前未分配的内存堆大小: xPortGetFreeHeapSize()

函数用于返回未分配的内存堆大小。这个函数也很有用，通常用于检查我们设置的内存堆是否合理，通过这个函数我们可以估计出最坏情况下需要多大的内存堆，以便合理的节省RAM。

```
size_t xPortGetFreeHeapSize( void )
{
    return ( configADJUSTED_HEAP_SIZE - xNextFreeByte );
}
```

heap_1内存管理策略中还有两个函数：vPortFree()和vPortInitialiseBlocks()。但实际上第一个函数什么也不做；第二个函数仅仅将静态局部变量xNextFreeByte设置为0

3.2heap_2

heap_2内存管理策略要比heap_1内存管理策略复杂

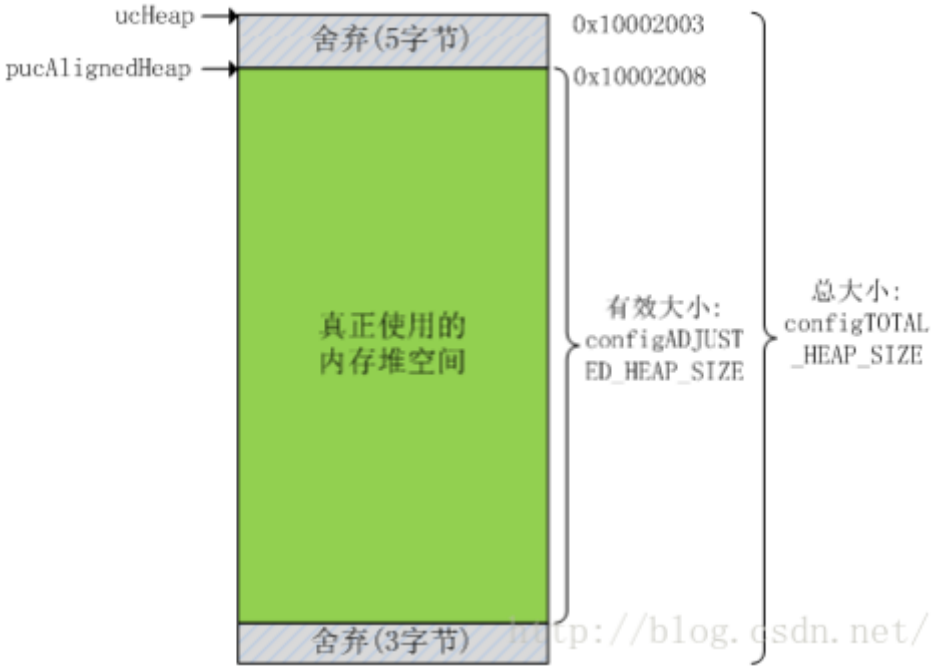
它使用一个最佳匹配算法，允许释放之前已分配的内存块

但是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内存管理策略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

不适用于分配和释放随机字节堆栈空间的应用程序!

局部静态变量pucAlignedHeap指向对齐后的内存堆起始位置。地址对齐的原因在第一种内存管理策略中已经说明。假如内存堆数组ucHeap从RAM地址0x10002003处开始，系统按照8字节对齐，则对齐后的内存堆与第一个内存管理策略一样, 如下图所示：

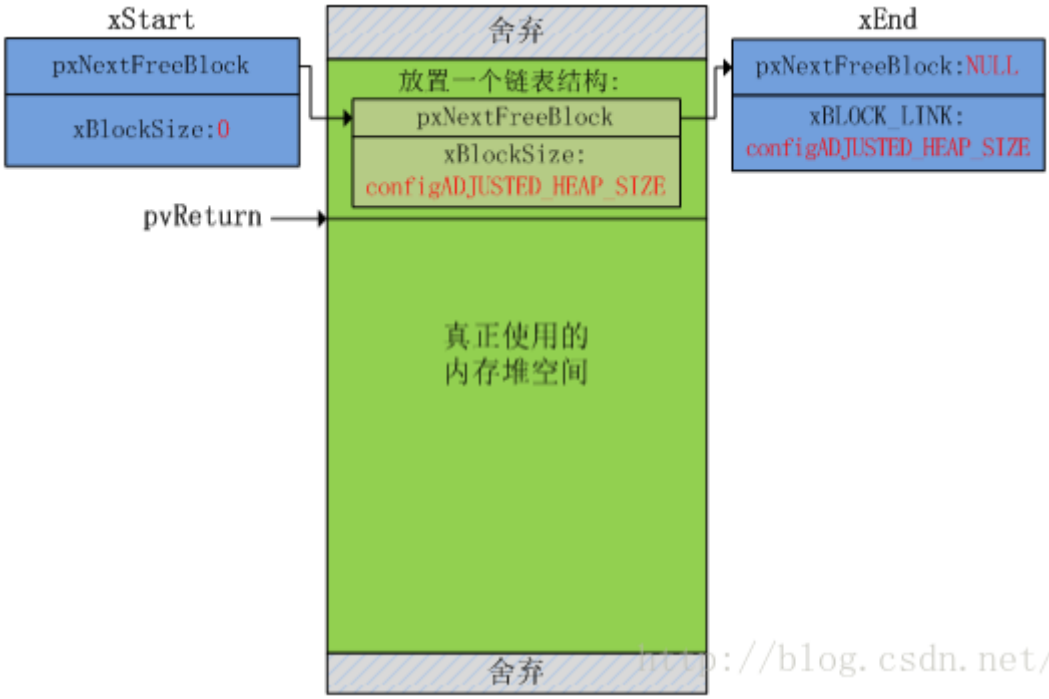


内存申请: pvPortMalloc()

与heap_1内存管理策略不同，heap_2内存管理策略使用链表结构来跟踪记录空闲内存块，将空闲块组成一个链表

```
typedef struct A_BLOCK_LINK
{
    struct A_BLOCK_LINK *pxNextFreeBlock;    /*指向列表中下一个空闲块*/
    size_t xBlockSize;                       /*当前空闲块的大小，包括链表结构大小*/
} BlockLink_t;
```

两个BlockLink_t类型的局部静态变量xStart和xEnd用来标识空闲内存块的起始和结束
刚开始时，整个内存堆有效空间就是一个空闲块，如下所示



整个有效空间组成唯一一个空闲块，在空闲块的起始位置放置了一个链表结构，用于存储这个空闲块的大小和下一个空闲块的地址。

由于目前只有一个空闲块，所以空闲块的pxNextFreeBlock指向链表xEnd，而链表xStart结构的pxNextFreeBlock指向空闲块。

这样，xStart、空闲块和xEnd组成一个单链表，xStart表示链表头，xEnd表示链表尾。

随着内存申请和释放，空闲块可能会越来越多，但它们仍是以xStart链表开头以xEnd链表结尾，根据空闲块的
大小排序，小的在前，大的在后
当申请N字节内存时，实际上不仅需要分配N字节内存，还要分配一个BlockLink_t类型结构体空间，用于描述
这个内存块，结构体空间位于空闲内存块的最开始处。当然，和heap_1内存管理策略一样，申请的内存大小和
BlockLink_t类型结构体大小都要向上扩大到对齐字节数的整数倍。

内存申请过程：

首先计算实际要分配的内存大小，判断申请的内存是否合法。

如果合法则从链表头xStart开始查找，如果某个空闲块的xBlockSize字段大小能容得下要申请的内存则从这块内存取出合适的部分返回给申请者，剩下的内存块组成一个新的空闲块，按照空闲块的大小顺序插入到空闲块链表中，小块在前大块在后。

注意，返回的内存中不包括链表结构，而是紧邻链表结构（经过对齐）后面的位置。

举个例子，如上图所示的内存堆，当调用申请内存函数如果内存堆空间足够大，就将pvReturn指向的地址返回给申请者，而不是静态变量pucAlignedHeap指向的内存堆起始位置！

当多次调用内存申请函数后（没有调用内存释放函数）

函数中使用的一个静态局部变量xFreeBytesRemaining，它用来记录未分配的内存堆大小。这个变
量将提供给函数xPortGetFreeHeapSize()使用，以方便用户估算内存堆使用情况

```
void *pvPortMalloc( size_t xWantedSize )
{
BlockLink_t *pxBlock, *pxPreviousBlock, *pxNewBlockLink;
static BaseType_t xHeapHasBeenInitialised = pdFALSE;
void *pvReturn = NULL;

    /* 挂起调度器 */
    vTaskSuspendAll();
    {
        /* 如果是第一次调用内存分配函数,这里先初始化内存堆 */
        if( xHeapHasBeenInitialised == pdFALSE )
        {
            prvHeapInit();
            xHeapHasBeenInitialised = pdTRUE;
        }
        /* 调整要分配的内存值,需要增加上链表结构体空间,heapSTRUCT_SIZE表示经过
对齐扩展后的结
构体大小 */
        if( xWantedSize > 0 )
        {
            xWantedSize += heapSTRUCT_SIZE;
            /* 调整实际分配的内存大小,向上扩大到对齐字节数的整数倍 */
            if( ( xWantedSize & portBYTE_ALIGNMENT_MASK ) != 0 )
            {
                xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize &
portBYTE_ALIGNMENT_MASK ) );
            }
        }

        if( ( xWantedSize > 0 ) && ( xWantedSize < configADJUSTED_HEAP_SIZE
) )
        {
            /* 空闲内存块是按照块的大小排序的,从链表头xStart开始,小的在前大的在
后,以链表尾
xEnd结束 */
            pxPreviousBlock = &xStart;
```

```
            pxBlock = xStart.pxNextFreeBlock;
            /* 搜索最合适的空闲块 */
            while( ( pxBlock->xBlockSize < xWantedSize ) && ( pxBlock-
>pxNextFreeBlock != NULL ) )
            {
                pxPreviousBlock = pxBlock;
                pxBlock = pxBlock->pxNextFreeBlock;
            }


            /* 如果搜索到链表尾xEnd,说明没有找到合适的空闲内存块,否则进行下一步
处理 */
            if( pxBlock != &xEnd )
            {
                /* 返回内存空间,注意是跳过了结构体BlockLink_t空间. */
                pvReturn = ( void * ) ( ( ( uint8_t * ) pxPreviousBlock-
>pxNextFreeBlock ) + heapSTRUCT_SIZE );


                /* 这个块就要返回给用户,因此它必须从空闲块中去除. */
                pxPreviousBlock->pxNextFreeBlock = pxBlock-
>pxNextFreeBlock;


                /* 如果这个块剩余的空间足够多,则将它分成两个,第一个返回给用户,第
二个作为新的
空闲块插入到空闲块列表中去*/
                if( ( pxBlock->xBlockSize - xWantedSize ) >
heapMINIMUM_BLOCK_SIZE )
                {
                    /* 去除分配出去的内存,在剩余内存块的起始位置放置一个链表结
构并初始化链表
成员 */
                    pxNewBlockLink = ( void * ) ( ( ( uint8_t * ) pxBlock )
+ xWantedSize );
                    pxNewBlockLink->xBlockSize = pxBlock->xBlockSize -
xWantedSize;
                    pxBlock->xBlockSize = xWantedSize;
                    /* 将剩余的空闲块插入到空闲块列表中,按照空闲块的大小顺序,小
的在前大的在后 */
                    prvInsertBlockIntoFreeList( ( pxNewBlockLink ) );
                }
                /* 计算未分配的内存堆大小,注意这里并不能包含内存碎片信息 */
                xFreeBytesRemaining -= pxBlock->xBlockSize;
            }
        }
        traceMALLOC( pvReturn, xWantedSize );
    }
    ( void ) xTaskResumeAll();

    #if( configUSE_MALLOC_FAILED_HOOK == 1 )
    {   /* 如果内存分配失败,调用钩子函数 */
```

```
        if( pvReturn == NULL )
        {
            extern void vApplicationMallocFailedHook( void );
            vApplicationMallocFailedHook();
        }
    }
    #endif
    return pvReturn;
}
```

内存释放: vPortFree()

因为不需要合并相邻的空闲块，第二种内存管理策略的内存释放也非常简单

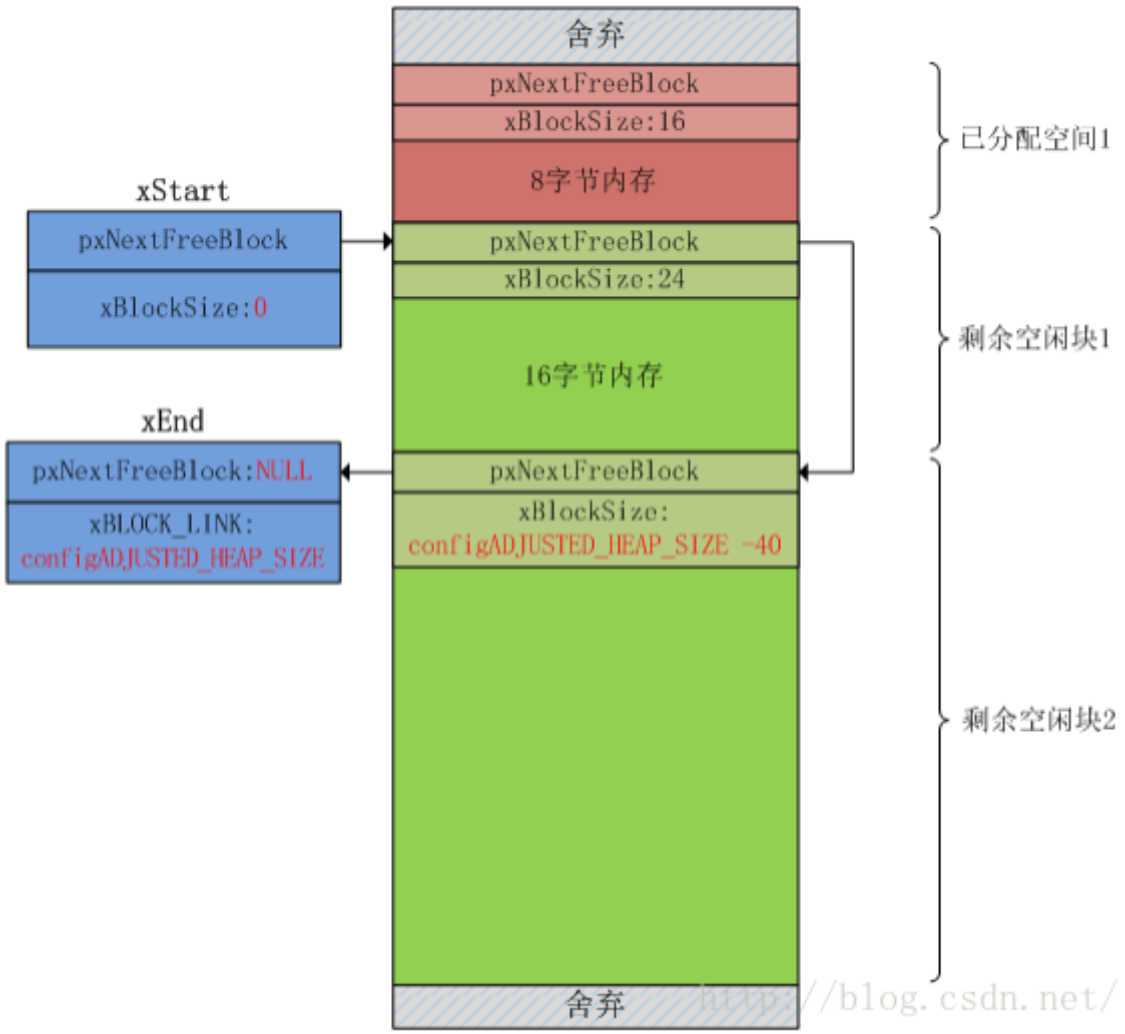
根据传入的参数找到链表结构，然后将这个内存块插入到空闲块列表，更新未分配的内存堆计数器大小

```
void vPortFree( void *pv )
{
uint8_t *puc = ( uint8_t * ) pv;
BlockLink_t *pxLink;
    if( pv != NULL )
    {
        /* 根据传入的参数找到链表结构 */
        puc -= heapSTRUCT_SIZE;
        /* 预防某些编译器警告 */
        pxLink = ( void * ) puc;
        vTaskSuspendAll();
        {
            /* 将这个块添加到空闲块列表 */
            prvInsertBlockIntoFreeList( ( ( BlockLink_t * ) pxLink ) );
            /* 更新未分配的内存堆大小 */
            xFreeBytesRemaining += pxLink->xBlockSize;

            traceFREE( pv, pxLink->xBlockSize );
        }
        ( void ) xTaskResumeAll();
    }
}
```

举一个例子，将上图的pvReturn指向的内存块释放掉，假设（configADJUSTED_HEAP_SIZE-40）远大于要释放的内存块大小， 如下图所示

-



可以看出heap_2内存管理策略的两个特点：

第一，空闲块是按照大小排序的；

第二，相邻的空闲块不会组合成一个大块。

再接着引申讨论一下heap_2内存管理策略的优缺点。

通过对内存申请和释放函数源码分析，一个优点是速度足够快，因为它的实现非常简单，第二个优点是可以动态释放内存。但是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）！

获取未分配的内存堆大小:xPortGetFreeHeapSize()

函数用于返回未分配的内存堆大小。这个函数也很有用，通常用于检查我们设置的内存堆是否合理，通

过这个函数我们可以估计出最坏情况下需要多大的内存堆，以便进行合理的节省RAM。

3.3heap_3

该内存管理策略简单的封装了标准库中的malloc()和free()函数

采用的封装方式是操作内存前挂起调度器、完成后再恢复调度器。

封装后的malloc()和free()函数具备线程保护

heap_1和heap_2策略都是通过定义一个大数组作为内存堆，数组的大小由configTOTAL_HEAP_SIZE指定。

heap_3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。

因此宏configTOTAL_HEAP_SIZE对这种内存管理策略是无效的

3.4heap_4

heap_4与heap_2相似，只不过增加了一个合并算法，将相邻的空闲内存块合并成一个大块

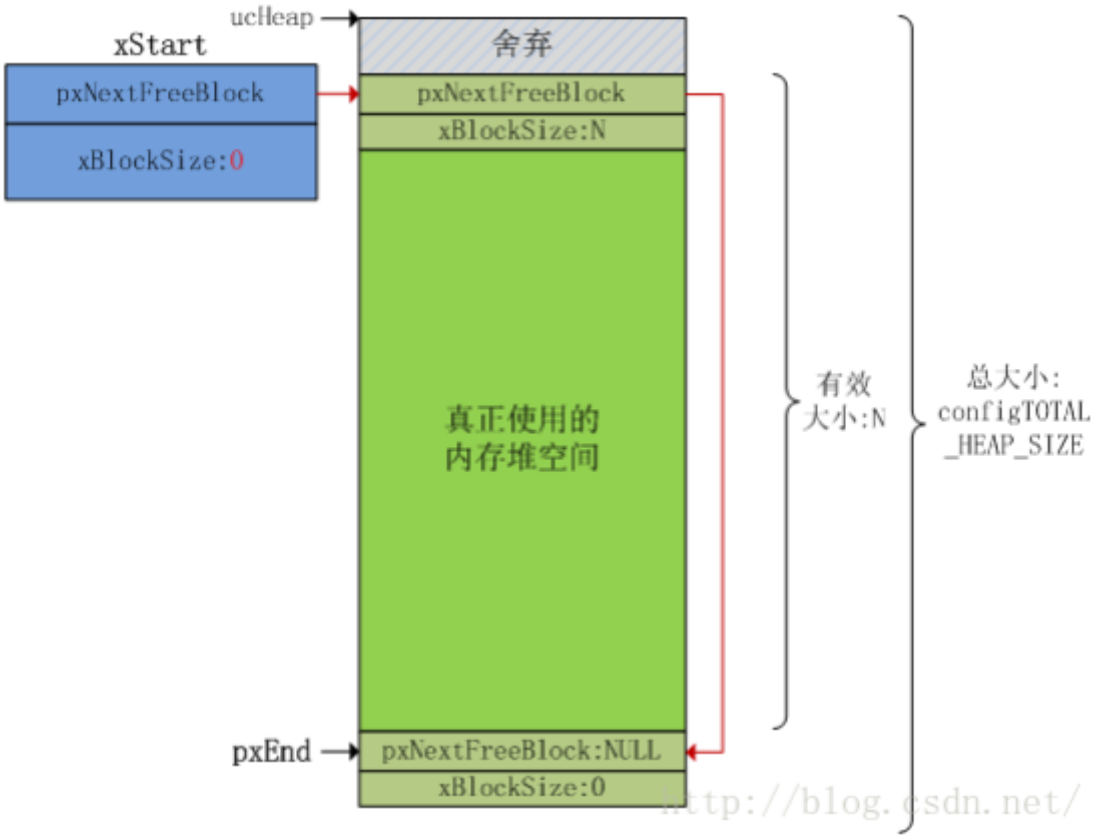
与heap_2内存管理策略一样，空闲内存块也是以单链表的形式组织起来的

BlockLink_t类型的局部静态变量xStart表示链表头，

但heap_4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（heap_2内存管理策略使用静态变量xEnd表示链表尾），如下图所示。

heap_4内存管理策略和heap_2内存管理策略还有一个很大的不同是：

heap_4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



整个有效空间组成唯一一个空闲块，在空闲块的起始位置放置了一个链表结构，用于存储这个空闲块的大小和下一个空闲块的地址。由于目前只有一个空闲块，所以空闲块的pxNextFreeBlock指向指针pxEnd指向的位置，而链表xStart结构的pxNextFreeBlock指向空闲块。xStart表示链表头，pxEnd指向位置表示链表尾

当申请x字节内存时，实际上不仅需要分配x字节内存，还要分配一个BlockLink_t类型结构体空间，用于描述这个内存块，结构体空间位于空闲内存块的最开始处。当然，和heap_1、heap_2内存管理策略一样，申请的内存大小和BlockLink_t类型结构体大小都要向上扩大到对齐字节数的整数倍。

内存申请过程：

首先计算实际要分配的内存大小，判断申请内存合法性，

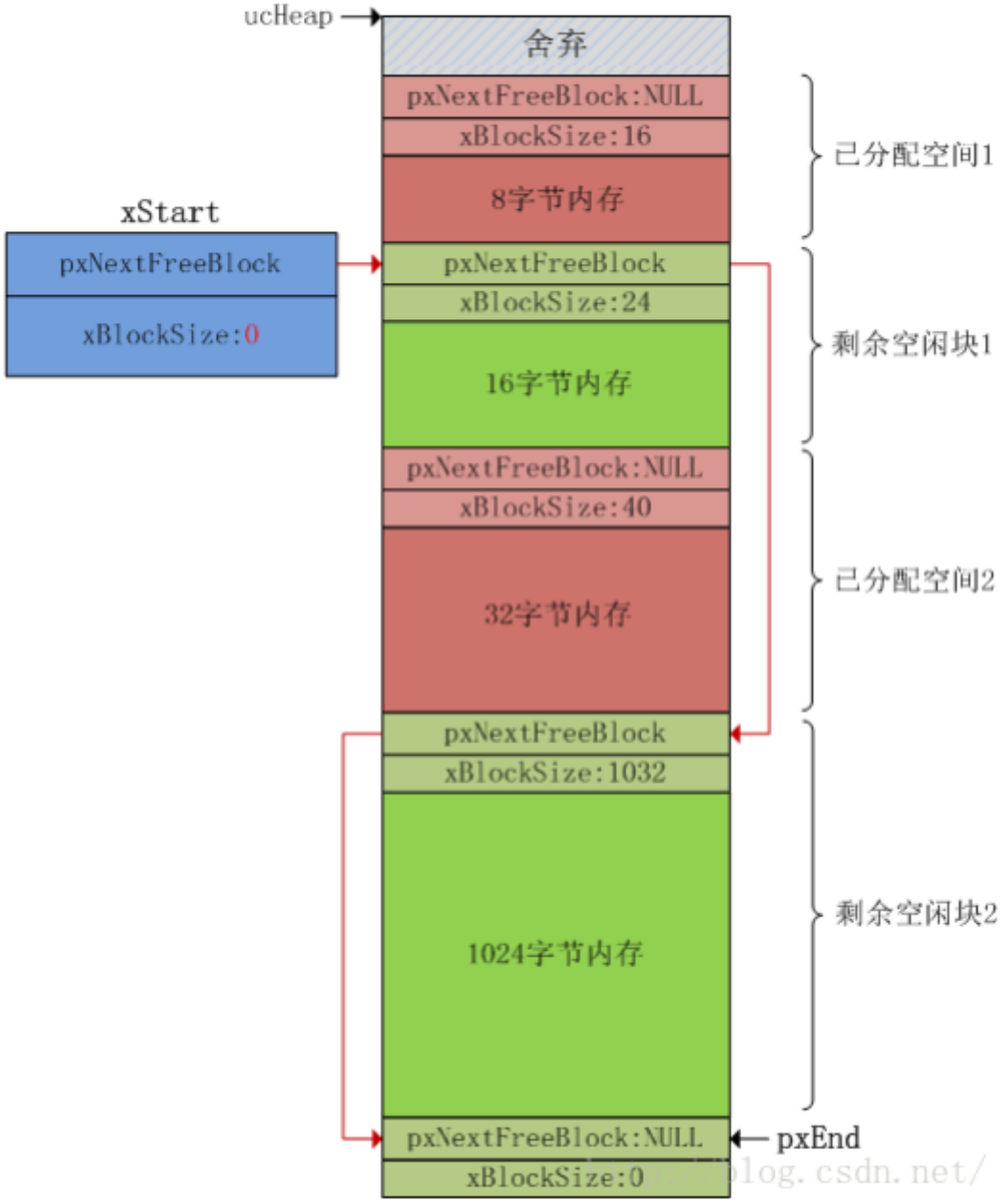
如果合法则从链表头xStart开始查找，

如果某个空闲块的xBlockSize字段大小能容得下要申请的内存，则将这块内存取出合适的部分返回给申请者，剩下的内存块组成一个新的空闲块

按照空闲块起始地址大小顺序插入到空闲块链表中，地址小的在前，地址大的在后。

在插入到空闲块链表的过程中，还会执行合并算法：判断这个块是不是可以和上一个空闲块合并成一个大块，如果可以则合并；

然后再判断能不能和下一个空闲块合并成一个大块，如果可以则合并！合并算法是heap_4内存管理策略和heap_2内存管理策略最大的不同!经过几次内存申请和释放后，可能的内存堆如下图所示：



函数中会用到几个局部静态变量在这里简单说明一下：

- xFreeBytesRemaining：表示当前未分配的内存堆大小
- xMinimumEverFreeBytesRemaining：表示未分配内存堆空间历史最小值。这个值跟xFreeBytesRemaining有很大区别，只有记录未分配内存堆的最小值，才能知道最坏情况下内存堆的使用情况。
- xBlockAllocatedBit：这个变量在第一次调用内存申请函数时被初始化，将它能表示的数值的最高位置1。比如对于32位系统，这个变量被初始化为0x80000000（最高位为1）。内存管理策略使用这个变量来标识一个内存块是否空闲。如果内存块被分配出去，则内存块链表结构成员xBlockSize按位或上这个变量（即xBlockSize最高位置1），在释放一个内存块时，会把xBlockSize的最高位清零。

```
void *pvPortMalloc( size_t xWantedSize )
{
BlockLink_t *pxBlock, *pxPreviousBlock, *pxNewBlockLink;
void *pvReturn = NULL;
    vTaskSuspendAll();
    {
        /* 如果是第一次调用内存分配函数,则初始化内存堆,初始化后的内存堆如图4-1所
示 */
        if( pxEnd == NULL )
        {
            prvHeapInit();
        }
        /* 申请的内存大小合法性检查:是否过大.结构体BlockLink_t中有一个成员
xBlockSize表示块
的大小,这个成员的最高位被用来标识这个块是否空闲.因此要申请的块大小不能使用这个
位.*/
        if( ( xWantedSize & xBlockAllocatedBit ) == 0 )
        {
            /* 计算实际要分配的内存大小,包含链接结构体BlockLink_t在内,并且要向
上字节对齐 */
            if( xWantedSize > 0 )
            {
                xWantedSize += xHeapStructSize;
                /* 对齐操作,向上扩大到对齐字节数的整数倍 */
                if( ( xWantedSize & portBYTE_ALIGNMENT_MASK ) != 0x00 )
                {
                    xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize &
portBYTE_ALIGNMENT_MASK ) );
                    configASSERT( ( xWantedSize & portBYTE_ALIGNMENT_MASK )
== 0
);
                }
            }
            if( ( xWantedSize > 0 ) && ( xWantedSize <= xFreeBytesRemaining
) )
            {
                /* 从链表xStart开始查找,从空闲块链表(按照空闲块地址顺序排列)中
找出一个足够大
的空闲块 */
                pxPreviousBlock = &xStart;
```

```
                pxBlock = xStart.pxNextFreeBlock;
                while( ( pxBlock->xBlockSize < xWantedSize ) && ( pxBlock-
>pxNextFreeBlock != NULL ) )
                {
                    pxPreviousBlock = pxBlock;
                    pxBlock = pxBlock->pxNextFreeBlock;
                }
                /* 如果最后到达结束标识,则说明没有合适的内存块,否则,进行内存分配
操作*/
                if( pxBlock != pxEnd )
                {
                    /* 返回分配的内存指针,要跳过内存开始处的BlockLink_t结构体
*/
                    pvReturn = ( void * ) ( ( ( uint8_t * )
pxPreviousBlock->pxNextFreeBlock ) + xHeapStructSize );
                    /* 将已经分配出去的内存块从空闲块链表中删除 */
                    pxPreviousBlock->pxNextFreeBlock = pxBlock-
>pxNextFreeBlock;
                    /* 如果剩下的内存足够大,则组成一个新的空闲块 */
                    if( ( pxBlock->xBlockSize - xWantedSize ) >
heapMINIMUM_BLOCK_SIZE )
                    {
                        /* 在剩余内存块的起始位置放置一个链表结构并初始化链表成
员 */
                        pxNewBlockLink = ( void * ) ( ( ( uint8_t * )
pxBlock ) + xWantedSize );
                        configASSERT( ( ( ( size_t ) pxNewBlockLink ) &
portBYTE_ALIGNMENT_MASK ) == 0 );
                        pxNewBlockLink->xBlockSize = pxBlock->xBlockSize -
xWantedSize;
                        pxBlock->xBlockSize = xWantedSize;
                        /* 将剩余的空闲块插入到空闲块列表中,按照空闲块的地址大
小顺序,地址小
的在前,地址大的在后 */
                        prvInsertBlockIntoFreeList( pxNewBlockLink );
                    }
                    /* 计算未分配的内存堆空间,注意这里并不能包含内存碎片信息 */
                    xFreeBytesRemaining -= pxBlock->xBlockSize;
                    /* 保存未分配内存堆空间历史最小值 */
                    if( xFreeBytesRemaining <
xMinimumEverFreeBytesRemaining )
                    {
                        xMinimumEverFreeBytesRemaining =
xFreeBytesRemaining;
                    }
                    /* 将已经分配的内存块标识为"已分配" */
                    pxBlock->xBlockSize |= xBlockAllocatedBit;
                    pxBlock->pxNextFreeBlock = NULL;
                }
            }
        }
        traceMALLOC( pvReturn, xWantedSize );
```

```
    }
    ( void ) xTaskResumeAll();


    #if( configUSE_MALLOC_FAILED_HOOK == 1 )
    {   /* 如果内存分配失败,调用钩子函数 */
        if( pvReturn == NULL )
        {
            extern void vApplicationMallocFailedHook( void );
            vApplicationMallocFailedHook();
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    #endif
    configASSERT( ( ( ( size_t ) pvReturn ) & ( size_t )
portBYTE_ALIGNMENT_MASK
) == 0 );
    return pvReturn;
}
```

内存释放: vPortFree():

heap_4内存管理策略的内存释放也比较简单：根据传入的参数找到链表结构，然后将这个内存块插入到空闲块列表，需要注意的是在插入过程中会执行合并算法。最后是将这个内存块标志为“空闲”、更新未分配的内存堆大小，结束。

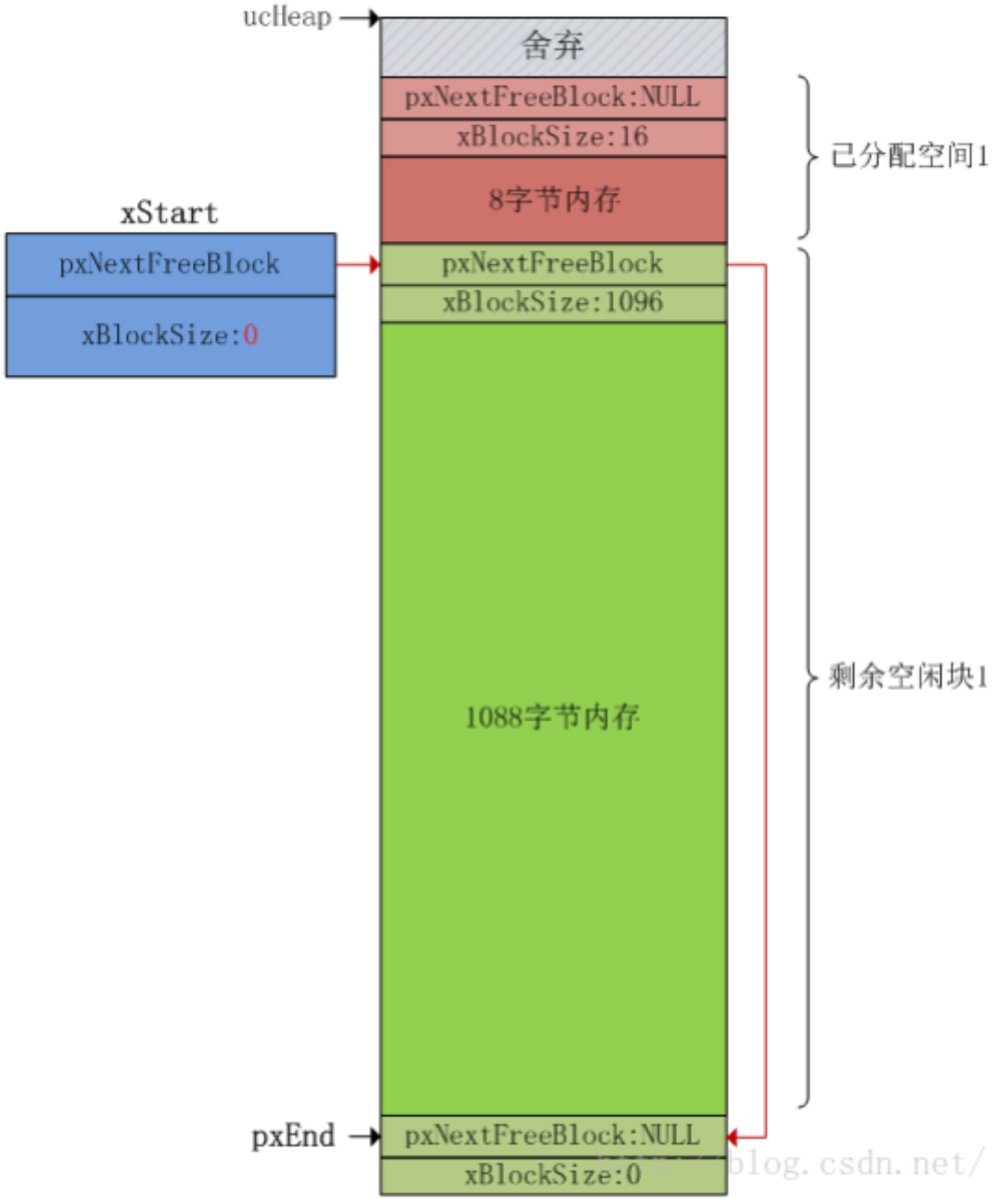
```
void vPortFree( void *pv )
{
uint8_t *puc = ( uint8_t * ) pv;
BlockLink_t *pxLink;

    if( pv != NULL )
    {
        /* 根据参数地址找出内存块链表结构 */
        puc -= xHeapStructSize;
        pxLink = ( void * ) puc;

        /* 检查这个内存块确实被分配出去 */
        if( ( pxLink->xBlockSize & xBlockAllocatedBit ) != 0 )
        {
            if( pxLink->pxNextFreeBlock == NULL )
            {
                /* 将内存块标识为"空闲" */
                pxLink->xBlockSize &= ~xBlockAllocatedBit;
                vTaskSuspendAll();
                {
                    /* 更新未分配的内存堆大小 */
                    xFreeBytesRemaining += pxLink->xBlockSize;
```

```
                    traceFREE( pv, pxLink->xBlockSize );
                    /* 将这个内存块插入到空闲块链表中,按照内存块地址大小顺序 */
                    prvInsertBlockIntoFreeList( ( ( BlockLink_t * ) pxLink
) );
                }
                ( void ) xTaskResumeAll();
            }
        }
    }
}
```

如上图所示的内存堆示意图，如果我们将32字节的“已分配空间2”释放，由于这个内存块的上面和下面都是空闲块，所以在将它插入到空闲块链表的过程在中，会先和“剩余空闲块1”合并，合并后的块再和“剩余空闲块2”合并，这样组成一个大的空闲块，如下图所示



3.5heap_5

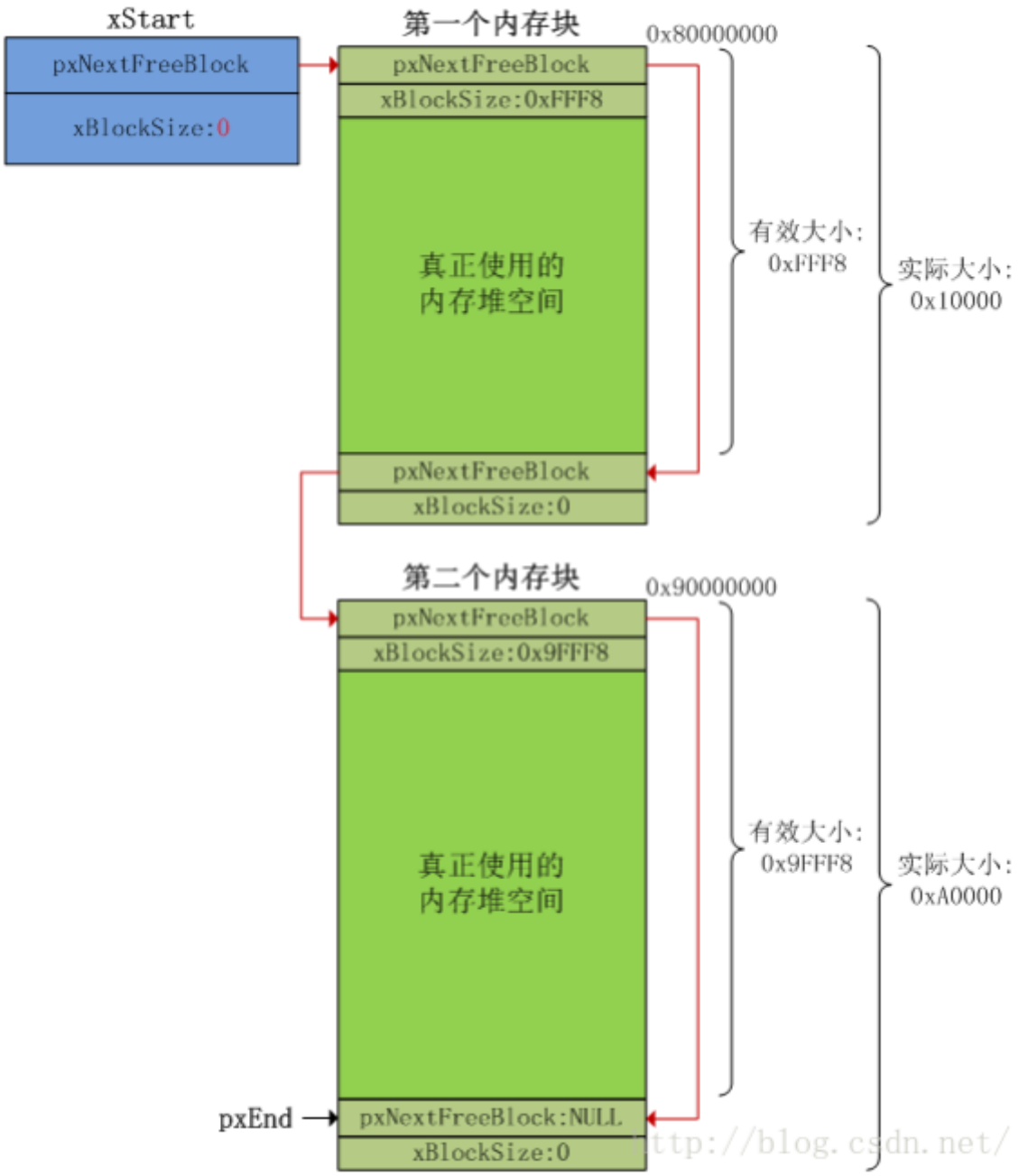
heap_5内存管理策略允许内存堆跨越多个非连续的内存区，并且需要显示的初始化内存堆，除此之外其它操作都和heap_4内存管理策略十分相似。

假设我们为内存堆分配两个内存块，第一个内存块大小为0x10000字节，起始地址为0x80000000；第二个内存块大小为0xa0000字节,起始地址为0x90000000。HeapRegion_t结构体类型数组可以定义如下：

```
HeapRegion_t xHeapRegions[] =
 {
    { ( uint8_t * ) 0x80000000UL, 0x10000 },
    { ( uint8_t * ) 0x90000000UL, 0xa0000 },
    { NULL, 0 }
 };
```

两个内存块要按照地址顺序放入到数组中，地址小的在前，因此地址为0x80000000的内存块必须放数组的第一个位置。数组必须以使用一个NULL指针和0字节元素作为结束，以便让内存管理程序知道何时结束

定义好内存堆数组后，需要应用程序调用vPortDefineHeapRegions()函数初始化这些内存堆：将它们组成一个链表，以xStart链表结构开头，以pxEnd指针指向的位置结束。我们看一下内存堆数组是如何初始化的，以上面的内存堆数组为例，初始化后的内存堆如下图所示



4.中断管理

4.1中断配置

`config_PRIO_BITS`：配置MCU使用几位优先级，STM32使用4位，所以为4

`configLIBRARY_LOWEST_INTERRUPT_PRIORITY`：最低优先级,STM32使用了4位,所以优先级数为16个,最低则为15

`configKERNEL_INTERRUPT_PRIORITY`：设置内核中断优先级，定义如下:

```
#define configKERNEL_INTERRUPT_PRIORITY
(configLIBRARY_LOWEST_INTERRUPT_PRIORITY<<(8- configPRIO_BITS))
//STM32使用4位作为优先级,这4位是高4位,所以要左移4位
//此宏用来设置PendSV和SysTick的中断优先级
//port.c定义
#define portNVIC_PENDSV_PRI \
(((uint32_t) configKERNEL_INTERRUPT_PRIORITY)<<16UL)
```

```
#define portNVIC_SYSTICK_PRI \
(((uint32_t) configKERNEL_INTERRUPT_PRIORITY)<<24UL)
//PendSV和SysTick的中断优先级设置 是操作0xE000_ED20地址,这样写入的是一个32位的数据,SysTick和PendSV的优先级寄存器分别对应这32位数据的最高8位和次高8位,所以一个左移16位,一个左移24位
//PendSV和SysTick优先级在函数 xPortStartScheduler()中设置,在port.c中
    { portNVIC_SHPR3_REG |= portNVIC_PENDSV_PRI;
portNVIC_SHPR3_REG |= portNVIC_SYSTICK_PRI;}
//上述就是配置PendSV和SysTick优先级的,即向指定地址写入数据,显然PendSV和SysTick的中断优先级是最低的
//例如优先级有效位4bit,则PendSV和SYSTICK的优先级为15(即最低优先级)
```

`configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY`：设置FreeRTOS系统可管理的最大优先级,也就是BASEPRI寄存器的阈值优先级,可自由设置,如果设置为5,则表示高于5的优先级(优先级数小于5)不归FreeRTOS管理

`configMAX_SYSCALL_INTERRUPT_PRIORITY`：此宏由`configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY`左移4位而来,原因和`configKERNEL_INTERRUPT_PRIORITY`一样,设置好之后,低于此优先级的中断可以安全的调用FreeRTOS的API函数,高于此优先级的中断FreeRTOS是不会禁止的,中断服务函数也不能调用API

```
//以STM32为例,有16个优先级,0最高,配置:
    configMAX_SYSCALL_INTERRUPT_PRIORITY = 5
    configKERNEL_INTERRUPT_PRIORITY = 15    //SYSTICK and PendSV
    /*结果为:
    优先级0~4的中断不会被FreeRTOS禁止(管理),不会因为执行FreeRTOS的内核而延时,中断不可调
用API,
    优先级5~15的终端可以调用以FromISR结尾的API,可嵌套
    不调用任何API的终端可以使用所有的中断优先级,可以嵌套
    由于高于configMAX_SYSCALL_INTERRUPT_PRIORITY 的优先级不会被内核屏蔽,因此那些对实
时性要求严格的任务就可以使用这些优先级,例如飞行器的避障检测*/
```

4.2临界保护

其接口为`portENABLE_INTERRUPTS()`和`portDISABLE_INTERRUPT()`

```
    #define portDISABLE_INRERRUPTS() vPortSetBasePRI(0)
    #define portENABLE_INTERRUPTS() vPortRaiseBASEPRI()
//原型如下:
Static portFORCE_INLINE void vPortSetBASEPRI(uint32_t ulBASEPRI){
    __asm
    {
        msr basepri, ulBASEPRI //立即数(为0)
    }
}
Static portFORCE_INLINE void vPortRaiseBASEPRI(void){
    Uint32_t ulNewBASEPRI = configMAX_SYSCALL_INTERRUPT_PRIORITY
    __asm
    {
```

```
        msr basepri, ulNewBASEPRI //可管理的最大优先级
        dsb
        isb
    }
}
```

临界段代码也叫做临界区,是指那些必须被完整运行不能打断的代码段,例如有的外设初始化严格的时序,FreeRTOS在进入临界段代码的时候需要关闭中断,当处理完临界段代码以后再打开,FreeRTOS本身就有很多临界段代码,都加了临界段代码保护,用户程序也需要添加临界段保护

- taskENTER_CRITICAL()
- taskEXIT_CRITICAL()
- taskENTER_CRITICAL_FROM_ISR()
- taskEXIT_CRITICAL_FROM_ISR()
  在task.h中定义,区别是前两个是任务级别的临界段保护,后两个是中断级别的临界段保护

任务级临界保护:

```
    #define taskENTER_CRITICAL() portENRER_CRITICAL()
    #define taskEXIT_CRITICAL() portEXIT_CRITICAL()
//在portmarco.h中
    #define portENRER_CRITICAL() vPortEnterCritical()
    #define portEXIT_CRITICAL() vPortExitCritical()
//在port.c中
void vPortEnterCritical(void){
        portDISABLE_INTERRUPTS();   //关闭中断
        uxCriticalNesting++;            //全局变量,记录临界段嵌套次数
        if(uxCriticalNesting==1){
            configASSERT((portNVIC_INT_CTRL_REG&portVECTACTIVE_MASK)==0)
        }
}
void vPortExitCritical(void){
    configASSERT(uxCriticalNesting);
    uxCriticalNesting--;
    if(uxCriticalNesting==0){  //保证了在有多个临界段代码的时候不会因为某一个临界段代码的退
出而打断其他临界段的保护,只有在所有临界段代码都退出的时候才会使能中断
        portENABLE_INTERRUPTS();
    }
}
/* 使用实例 */
    taskENTER_CRITICAL();
    …
    taskEXIT_CRITICAL();
    //注意临界区代码一定要精简,因为进入临界区会关闭中断,这样会导致优先级低于
configMAX_SYSCALL_INTERRUPT_PRIORITY的中断得不到及时的响应
```

中断级临界保护:

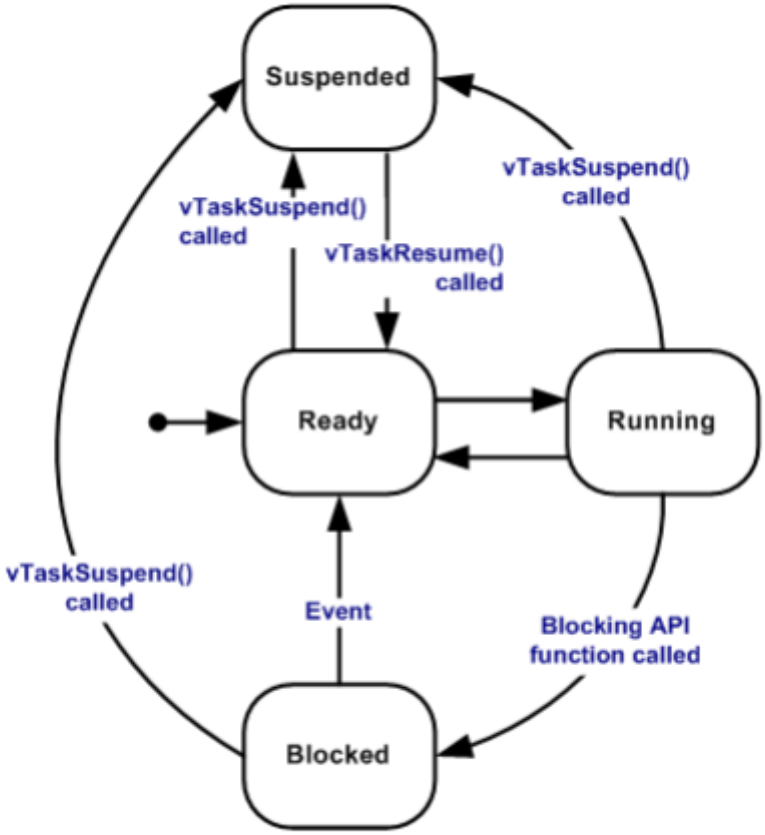
```
taskENTER_CRITICAL_FROM_ISR()
```

```
taskEXIT_CRITICAL_FROM_ISR()
    //这两个函数是用在中断服务程序中的,而且这个中断优先级一定要低于
configMAX_SYSCALL_INTERRUPT_PRIORITY
    //在task.h
#define taskENTER_CRITICAL_FROM_ISR() portSET_INTERRUPT_MASK_FROM_ISR()
#define taskEXIT_CRITICAL_FROM_ISR(x) portCLEAR_INTERRUPT_MASK_FROM_ISR(x)
    //在portmacro.h中
#define portSET_INTERRUPT_MASK_FROM_ISR() ulPortRaiseBASEPRI()
#define portCLEAR_INTERRUPT_MASK_FROM_ISR(x) vPortSetBASEPRI(x)
vPortSetBASEPRI(x)就是给BASEPRI寄存器写入一个值
static portFORCE_INLINE uint32_t ulPortRaiseBASEPRI(void)
{
    Uint32_t ulReturn,ulNewBASEPRI = configMAX_SYSCALL_INTERRUPT_PRIORITY;
    __asm{
        mrs ulReturn,basepri         //read basepri
        msr basepri,ulNewBASEPRI     //write basepri (屏蔽所有低于此的优先级中
断)
        dsb
        isb
    }
    Return ulReturn;         //返回ulReturn,退出临界区代码保护的时候要使用此值
}
/* 使用实例 */
void …Handler(){
    Uint32_t status_value;
    status_value = taskENTER_CRITICAL_FROM_ISR();
    …
    taskEXIT_CRITICAL_FROM_ISR(status_value);
}
```

5.任务调度管理

5.1任务状态模型

- 运行态Running
- 就绪态Ready
- 阻塞态Blocked，等待某个时间(时间,信号,标志位等)被动阻塞
- 挂起(暂停)态Suspend，主动暂停



Valid task state transitions

5.2任务管理控制块

```
typedef struct tskTaskControlBlock
    {
        // 这里栈顶指针必须位于TCB第一项是为了便于上下文切换操作，详见xPortPendSVHandler中
任务切换的操作。
        volatile StackType_t    *pxTopOfStack;

        // MPU相关暂时不讨论
        #if ( portUSING_MPU_WRAPPERS == 1 )
            xMPU_SETTINGS    xMPUSettings;
        #endif
        // 表示任务状态，不同的状态会挂接在不同的状态链表下
        // Running State  ---> pxCurrentTCB
        // Ready State  ---> pxReadyTasksLists (数组+循环链表)
        // Blocked State ---> pxDelayedTaskList
        // Suspended State ---> xSuspendedTaskList
        ListItem_t            xStateListItem;
        // 事件链表项，会挂接到不同事件链表下
        ListItem_t            xEventListItem;
        // 任务优先级，数值越大优先级越高
        UBaseType_t            uxPriority;
        // 指向堆栈起始位置，这只是单纯的一个分配空间的地址，可以用来检测堆栈是否溢出
        StackType_t            *pxStack;
        // 任务名
```

```
        char                pcTaskName[ configMAX_TASK_NAME_LEN ];

        // 指向栈尾，可以用来检测堆栈是否溢出
        #if ( ( portSTACK_GROWTH > 0 ) || ( configRECORD_STACK_HIGH_ADDRESS
== 1 ) )
            StackType_t        *pxEndOfStack;
        #endif

        // 记录临界段的嵌套层数
        #if ( portCRITICAL_NESTING_IN_TCB == 1 )
            UBaseType_t        uxCriticalNesting;
        #endif

        // 跟踪调试用的变量
        #if ( configUSE_TRACE_FACILITY == 1 )
            UBaseType_t        uxTCBNumber;
            UBaseType_t        uxTaskNumber;
        #endif

        // 任务优先级被临时提高时，保存任务原本的优先级
        #if ( configUSE_MUTEXES == 1 )
            UBaseType_t        uxBasePriority;
            UBaseType_t        uxMutexesHeld;
        #endif

        // 任务的一个标签值，可以由用户自定义它的意义，例如可以传入一个函数指针可
以用来做Hook
   函数调用
        #if ( configUSE_APPLICATION_TASK_TAG == 1 )
            TaskHookFunction_t pxTaskTag;
        #endif

        // 任务的线程本地存储指针，可以理解为这个任务私有的存储空间
        #if( configNUM_THREAD_LOCAL_STORAGE_POINTERS > 0 )
            void               *pvThreadLocalStoragePointers[
configNUM_THREAD_LOCAL_STORAGE_POINTERS ];
        #endif

        // 运行时间变量
        #if( configGENERATE_RUN_TIME_STATS == 1 )
            uint32_t           ulRunTimeCounter;
        #endif

        // 支持NEWLIB的一个变量
        #if ( configUSE_NEWLIB_REENTRANT == 1 )
            struct    _reent xNewLib_reent;
        #endif

        // 任务通知功能需要用到的变量
        #if( configUSE_TASK_NOTIFICATIONS == 1 )
            // 任务通知的值
            volatile uint32_t ulNotifiedValue;
```

```
        // 任务通知的状态
        volatile uint8_t ucNotifyState;
    #endif

    // 用来标记这个任务的栈是不是静态分配的
    #if( tskSTATIC_AND_DYNAMIC_ALLOCATION_POSSIBLE != 0 )
        uint8_t    ucStaticallyAllocated;
    #endif

    // 延时是否被打断
    #if( INCLUDE_xTaskAbortDelay == 1 )
        uint8_t ucDelayAborted;
    #endif

    // 错误标识
    #if( configUSE_POSIX_ERRNO == 1 )
        int iTaskErrno;
    #endif

} tskTCB;
typedef tskTCB TCB_t;
```

5.3任务链表

根据优先级和状态可将任务发放到多个任务链表中，内核通过任务链表帮助调度器对任务进行管理

- 就绪态: 每个优先级都有一个对应的对应链表,最多有configMAX_PRIORITIES个
    - `static List_t pxReadyTasksLists[configMAX_PRIORITIES]`
- 阻塞态: 发生在当任务因为延时或等待事件的发生时，延时任务链表里的任务通过延时的先后顺序由小到大排列
    - ```
      static List_t * volatile pxDelayedTaskList;
      static List_t * volatile pxOverflowDelayedTaskList;
      ```
- 挂起(暂停)态: 当任务被挂起时,任务放在暂停列表里
    - `static List_t xSuspendTaskList;`
- Pending态: 当任务从挂起态或阻塞态被恢复时，如果调度器此时已经被挂起，则任务会被放进xPendingReadyList链表，等到调度器恢复时(xTaskResumeAll)再将这些xPendingReadyList里的任务放进就绪链表。

xPendingReadyList链表的作用?为什么不直接将恢复的任务放进就绪链表?（因为进入此链表的前提是调度器被挂起，保持当前任务的运行状态，无法执行抢占操作，调度器不工作就无法将就绪任务放入就绪链表，这时恢复的任务就需要先存放在此链表）
      ```

5.4空闲任务

```
/*
 * -----------------------------------------------------------
 * The Idle task.
 * ----------------------------------------------------------
 *
 * The portTASK_FUNCTION() macro is used to allow port/compiler specific
 * language extensions.  The equivalent prototype for this function is:
 *
 * void prvIdleTask( void *pvParameters );
 *
 */
static portTASK_FUNCTION( prvIdleTask, pvParameters )
{
    ( void ) pvParameters;
    portALLOCATE_SECURE_CONTEXT( configMINIMAL_SECURE_STACK_SIZE );

    for( ; ; ) {
        prvCheckTasksWaitingTermination();
        #if ( configUSE_PREEMPTION == 0 ) {
            taskYIELD();
        }
        #endif /* configUSE_PREEMPTION */
        #if ( ( configUSE_PREEMPTION == 1 ) && ( configIDLE_SHOULD_YIELD ==
1 ) )
{
            if( listCURRENT_LIST_LENGTH( &( pxReadyTasksLists[
tskIDLE_PRIORITY ] ) ) > ( UBaseType_t ) 1 ) {
                /* 如果0优先级的就绪列表长度大于1，那么主动执行一次让步，使其他
任务都执行完毕
之后空闲任务再接着执行 */
                taskYIELD();
            } else {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        #endif

        #if ( configUSE_IDLE_HOOK == 1 ) {
            extern void vApplicationIdleHook( void );
            vApplicationIdleHook();
        }
        #endif /* configUSE_IDLE_HOOK */

        #if ( configUSE_TICKLESS_IDLE != 0 ) {
            TickType_t xExpectedIdleTime;

            xExpectedIdleTime = prvGetExpectedIdleTime();
```

```
            if( xExpectedIdleTime >= configEXPECTED_IDLE_TIME_BEFORE_SLEEP
) {
                vTaskSuspendAll();
                {
                    configASSERT( xNextTaskUnblockTime >= xTickCount );
                    xExpectedIdleTime = prvGetExpectedIdleTime();

                    configPRE_SUPPRESS_TICKS_AND_SLEEP_PROCESSING(
xExpectedIdleTime );
                    if( xExpectedIdleTime >=
configEXPECTED_IDLE_TIME_BEFORE_SLEEP ) {
                        traceLOW_POWER_IDLE_BEGIN();
                        portSUPPRESS_TICKS_AND_SLEEP( xExpectedIdleTime );
                        traceLOW_POWER_IDLE_END();
                    } else {
                        mtCOVERAGE_TEST_MARKER();
                    }
                }
                ( void ) xTaskResumeAll();
            } else {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        #endif /* configUSE_TICKLESS_IDLE */
    }
}
```

5.5调度决策

`决定要运行的任务?`

- 找到最高优先级的运行态，就绪态任务，运行
- 如果任务优先级都相同，轮流执行，排队，链表前面的先运行，运行1个tick后将当前任务重新排到链表尾部。

`如何找到最高优先级的任务?`

- 创建任务的时候，先为TCB和栈申请空间
- 然后使用prvAddNewTaskToReadyList(pxNewTCB)添加到就绪链表中
- 实际上在此函数中调用的proAddTaskToReadyList(pxNewTCB)插入对应优先级的链表尾部
- 在调度器选择任务的时候会从pxReadyTaskLists[优先级]从大到小进行挑选，并会选择最大优先级链表的第一个就绪任务执行

`pxReadyTaskLists就绪任务优先级数组`:

pxReadyTaskLists[]指针数组的大小是可配置的，每个元素放置一个优先级就绪链表的首地址，因为Task有多个优先级，选择任务运行的时候往往需要考虑任务状态以及优先级，根据不同优先级，将就绪任务放在不同优先级的链表中方便调度器进行选择。但pxDelayedTaskList与xPendingReadyList大小都为1，因为阻塞态不需要根据优先级处理

```
/* 创建三个任务 */
xTaskCreate(vTask1, "Task_1", 1000, NULl, 0, NULl);
xTaskCreate(vTask1, "Task_2", 1000, NULl, 0, NULl);
xTaskCreate(vTask1, "Task_3", 1000, NULl, 2, NULl);
-------------------------------------------------
/* 创建三个任务会根据优先级将任务地址链接到对应优先级的链表末尾，vTask3的优先级高，它对应的链表
就与vTask1、vTask2不同。
 * 并根据创建顺序可以看到，相同优先级的任务，vTask2创建的晚，则它在链表末端。当vTask1执行完成之
后，又放回到链表末端 */
pxReadyTaskLists[4]
pxReadyTaskLists[3]
pxReadyTaskLists[2] ---> vTask3   /* 任务调度器从pxReadyTaskLists[高优先级]挑选就会先
挑选vtask3执行 */
pxReadyTaskLists[1]
pxReadyTaskLists[0] ---> vTask1 ---> vTask2 ---> Idle Task(优先级为0 的空闲任务)

pxDelayedTaskList
pxPendingReadyList
```

`谁进行调度?`:

TICK中断，调度系统的核心，一个任务执行一个Tick时长，在Tick中断函数中会进行任务切换（调整任务所在的任务链表）

每次中断会检查有哪些延时任务到期，将其从延时任务列表里移除并加入到就绪列表里。如果就绪任务的优先级都相同，如果开启时间片轮询，就会每个tick执行一个任务，轮询执行

Tick中断做了什么:

- 找到最高优先级的就绪任务
- 保存当前任务信息,并将其插入就绪任务链表末端
- 恢复新task

可抢占：高优先级任务先运行
时间片轮转：同优先级的任务轮流执行
空闲任务礼让：如果其他优先级为0 的任务，空闲任务主动放弃一次运行机会

5.5.1系统调用

执行系统调用就是执行FreeRTOS系统提供的API函数,比如任务切换函数taskYIELD(),FreeRTOS有些API函数也会调用函数taskYIELD(),这些API函数都会导致任务切换,函数taskYIELD()其实就是宏，在task.h中定义

```
    #define taskYIELD() portYIELD()
    //在portmacro.h中定义
    #define portYIELD()                                         \
    {                                                           \
        portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT; \
        //通过向中断控制和状态寄存器ICSR的bit28切入1挂起PendSV来启动PendSV中断
        __dsb( portSY_FULL_READ_WRITE );                        \
        __isb( portSY_FULL_READ_WRITE );                        \
    }
```

中断级的任务切换函数为`portYIELD_FROM_ISR()`,定义如下:

```
  #define    portEND_SWITCHING_ISR(xSwitchRequired) \
             If(xSwitchRequired!=pdFALSE)    portYIELD()
  #define portYIELD_FROM_ISR(x)   portEND_SWITCHING_ISR(x)
```

可以看出`portYIELD_FROM_ISR()`最终也是调用函数`portYIELD()`来完成任务切换的

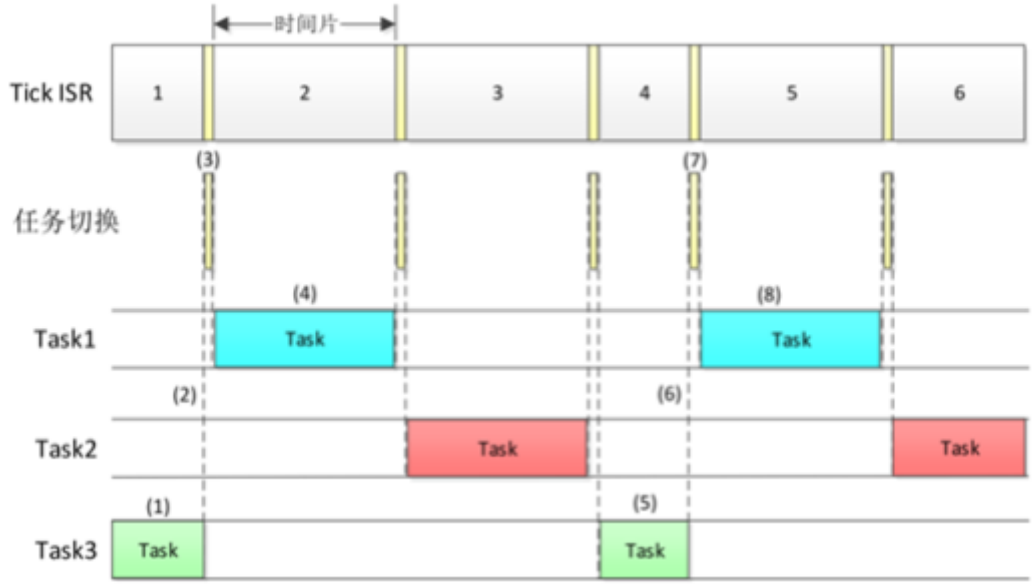
5.5.2内核节拍中断

FreeRTOS中Systick的ISR也会进行任务切换

```
void SysTick_Handler(void){
    if(xTaskGetSchedulerState()!=taskSCHEDULER_NOT_STARTED){
        xPortSysTickHandler();  //if system is running
    }
    HAL_IncTick();
}
void xPortSysTickHandler(void){
    vPortRaiseBASEPRI();    //关中断
    {
        If(xTaskIncrementTick()!=pdFALSE){      //增加时钟计数器xTickCount的
值
            portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
            //通过向中断控制和状态寄存器ICSR的bit28切入1挂起PendSV来启动PendSV
中断
        }
    }
    vPortClearBASEPRIFROMISR(); //开中断
}
```

5.5.3时间片调度

FreeRTOS支持多个任务拥有同一个优先级,在FreeRTOS中允许一个任务运行一个时间片后让出CPU使用权,让同优先级的下一个任务运行，例如在同一优先级下有3个就绪任务:



(1) 任务3正在运行
(2) Systick中断发送,任务3时间片用完,但任务3还没有执行完
(3) FreeRTOS将任务切换到任务1,任务1是同优先级下的下一个就绪任务
(4) 任务1连续运行到时间片用完
(5) 任务3再次获取使用权继续运行
(6) 任务3运行完成,调用任务切换函数portYIELD()强行进行任务切换放弃剩余时间片
(7) FreeRTOS切换到任务1
(8) 任务1执行完其时间片
要使用时间片调度的话宏configUSE_PREEMPTION和宏configUSE_TIME_SLICING必须为1,时间片长度
由宏configTICK_RATE_HZ来决定,例如以上例子里的时间片长度为1ms
而执行systick的任务调度时,会判断调度器的条件
If(xTaskIncrementTick()!=pdFALSE)
xTaskIncrementTick函数里会判断相应的宏是否为1,
判断当前任务优先级下是否还有其他任务
如果当前任务所对应的任务优先级下还有其他任务就返回pdTRUE

5.5.4寻找下一个任务

```
void vTaskSwitchContext( void )
{
    if( uxSchedulerSuspended != ( UBaseType_t ) pdFALSE )    (1)
    {   //如果调度器挂起那就不能进行任务切换
        xYieldPending = pdTRUE;
    }
    else
    {
        xYieldPending = pdFALSE;
        traceTASK_SWITCHED_OUT();
        taskCHECK_FOR_STACK_OVERFLOW();
        //调用该宏获取下一个要运行的任务
        taskSELECT_HIGHEST_PRIORITY_TASK();                (2)
        traceTASK_SWITCHED_IN();
    }
```

```
}
```

FreeRTOS中查找下一个要运行的任务有两种方法, 通用方法和硬件方法

使用方法通过宏configUSE_PORT_OPTIMISED_TASK_SELECTION来决定,为1时使用硬件方法

通用方法:

```
    #define taskSELECT_HIGHEST_PRIORITY_TASK()
    {
        UBaseType_t uxTopPriority = uxTopReadyPriority;
        while(listLIST_IS_EMPTY(&(pxReadyTasksLists[uxTopPriority])))
        {
            //循环查找就绪任务列表数组,一个优先级一个列表,同优先级挂在链表上,
uxTopPriority代表就绪态的最高优先级，从最高优先级开始判断，哪个列表不为空则说明
有就绪任务
            configASSERT( uxTopPriority );
            --uxTopPriority;
        }
        //从对应的列表中获取下一个要运行的任务
        listGET_OWNER_OF_NEXT_ENTRY(pxCurrentTCB,&
(pxReadyTasksLists[uxTopPriority]));
        uxTopReadyPriority = uxTopPriority;
}
```

硬件方法

使用Cortex-M自带的硬件指令计算前导0个数的CLZ (限制优先级个数,比如STM32只能有32个优先级)

```
    #define taskSELECT_HIGHEST_PRIORITY_TASK()
    {
        UBaseType_t uxTopPriority;
        //获取就绪态的最高优先级
portGET_HIGHEST_PRIORITY(uxTopPriority,uxTopReadyPriority );

configASSERT(listCURRENT_LIST_LENGTH(&
(pxReadyTasksLists[uxTopPriority]))>0);
        //从对应的列表中取出要运行的任务
        listGET_OWNER_OF_NEXT_ENTRY(pxCurrentTCB,&
(pxReadyTasksLists[uxTopPriority]));
    }
#define portGET_HIGHEST_PRIORITY(uxTopPriority,uxReadyPriorities)
uxTopPriority=
(31UL-(uint32_t)__clz(uxReadyPriorities))
//__clz(uxReadyPriorities)就是计算uxReadyPriorities的前导零个数,前导零个数就是
指从最高位开始到第一个为1的bit,期间0的个数
//使用硬件方法的时候uxTopPriority就不代表就绪态的最高优先级了,而是使用每个bit代
表一个优先级,bit0代表优先级0,bit31代表优先级31,当某个优先级由就绪任务的话就将其
对应的bit置1,从这里可以看出,硬件方法最多只有32个优先级
```

6.内核时钟/延时模型

6.1vTaskDelay分析

```
void vTaskA( void * pvParameters )
 {
     /* 阻塞500ms. 注:宏pdMS_TO_TICKS用于将毫秒转成节拍数,FreeRTOS V8.1.0及
        以上版本才有这个宏,如果使用低版本,可以使用 500 / portTICK_RATE_MS */
     const portTickType xDelay = pdMS_TO_TICKS(500);
     for( ;; )  {
         //  ...
         //  这里为任务主体代码
         //  ...

         /* 调用系统延时函数,阻塞500ms */
         vTaskDelay( xDelay );
     }
}
```

任务A自调用vTaskDelay()的时刻起进入阻塞状态,FreeRTOS内核会周期性检查任务A的阻塞是否达到,如果阻塞时间达到,则将任务A设置为就绪态。

任务A每次延时都是从调用延时函数vTaskDelay()开始算起,延时是相对于这一时刻开始的,所以叫做相对延时函数。

如果执行任务A的过程中发生中断,则任务A的执行周期会变长，所以使用相对延时函数vTaskDelay()，不能周期性的执行任务A。

```
void vTaskDelay( const TickType_t xTicksToDelay )
{
    BaseType_t xAlreadyYielded = pdFALSE;
    /* 如果延时时间为0,则不会将当前任务加入延时列表 */
    if( xTicksToDelay > ( TickType_t ) 0U )
    {
        vTaskSuspendAll();
        {
            /* 将当前任务从就绪列表中移除,并根据当前系统节拍计数器值计算唤醒时
间,然后将任务加
入延时列表pxDelayedTaskList或者pxOverflowDelayedTaskList中 */
            prvAddCurrentTaskToDelayedList( xTicksToDelay, pdFALSE );
        }
        xAlreadyYielded = xTaskResumeAll();
    }
    /* 强制执行一次上下文切换*/
    if( xAlreadyYielded == pdFALSE )
    {
        portYIELD_WITHIN_API();
    }
}
//=====================================================================
static void prvAddCurrentTaskToDelayedList( TickType_t xTicksToWait, const
BaseType_t xCanBlockIndefinitely )
```

```
{
TickType_t xTimeToWake;
    //读取内核计时器
const TickType_t xConstTickCount = xTickCount;

    #if( INCLUDE_xTaskAbortDelay == 1 )
    {
        pxCurrentTCB->ucDelayAborted = pdFALSE;
    }
    #endif
    //要将当前正在运行的任务添加到延时列表中,肯定要先将当前任务从就绪列表中删除
    if( uxListRemove( &( pxCurrentTCB->xStateListItem ) ) == ( UBaseType_t
) 0 )
    {
        //将当前任务从就绪列表中移除之后要取消任务在uxTopReadyPriority中的标记
        portRESET_READY_PRIORITY( pxCurrentTCB->uxPriority,
uxTopReadyPriority );
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    #if ( INCLUDE_vTaskSuspend == 1 )
    {
        //延时时间最大值为portMAX_DELAY,并且xCanBlockIndefinitely不为
pdFALSE(表示允许阻塞),直接将当前任务添加到挂起列表中
        if( ( xTicksToWait == portMAX_DELAY ) && ( xCanBlockIndefinitely !=
pdFALSE ) )
        {
            //当前任务添加到挂起列表xSuspendedTaskList末尾
            vListInsertEnd( &xSuspendedTaskList, &( pxCurrentTCB-
>xStateListItem ) );
        }
        else
        {
            //计算任务唤醒时间点,也就是当前的时间点加上延时时间值
            xTimeToWake = xConstTickCount + xTicksToWait;
            //唤醒时间点值xTimetoWake写入任务列表中状态列表项的字段中
            listSET_LIST_ITEM_VALUE( &( pxCurrentTCB->xStateListItem ),
xTimeToWake );

            if( xTimeToWake < xConstTickCount )
            {
                //计算的唤醒时间点发生了溢出,添加到overflow列表中
                vListInsert( pxOverflowDelayedTaskList, &( pxCurrentTCB-
>xStateListItem ) );
            }
            else
            {
                //正常情况下添加到delayList中
```

```
                vListInsert( pxDelayedTaskList, &( pxCurrentTCB->xStateListItem ) );
                //xNextTaskUnblockTime是全局变量,保存着距离下一个要取消阻塞的最小时间点值,当xTimetoWake小于这个值的话说明有个更小的时间点
                if( xTimeToWake < xNextTaskUnblockTime )
                {
                    xNextTaskUnblockTime = xTimeToWake;
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
        }
    }
    #else /* INCLUDE_vTaskSuspend */
    {
        xTimeToWake = xConstTickCount + xTicksToWait;
        listSET_LIST_ITEM_VALUE( &( pxCurrentTCB->xStateListItem ), xTimeToWake );

        if( xTimeToWake < xConstTickCount )
        {
            vListInsert( pxOverflowDelayedTaskList, &( pxCurrentTCB->xStateListItem ) );
        }
        else
        {
            vListInsert( pxDelayedTaskList, &( pxCurrentTCB->xStateListItem ) );
            if( xTimeToWake < xNextTaskUnblockTime )
            {
                xNextTaskUnblockTime = xTimeToWake;
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        /* Avoid compiler warning when INCLUDE_vTaskSuspend is not 1. */
        ( void ) xCanBlockIndefinitely;
    }
    #endif /* INCLUDE_vTaskSuspend */
}
```

task.c中定义了局部变量

```
static volatile TickType_t xTickCount = ( TickType_t ) 0U;
```

该变量为全局的系统节拍计数器，下次唤醒任务的时间xTickCount + xTicksToDelay会被记录到任务TCB中,随着任务一起被挂到延时列表。

为了解决xTickCount的溢出问题,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

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

6.2vTaskDelayUntil分析

此函数会阻塞任务,阻塞时间是一个绝对时间,需要按照一定的频率运行的任务可以使用此函数

```
void vTaskB( void * pvParameters )
{
    static portTickType xLastWakeTime;
    const portTickType xFrequency = pdMS_TO_TICKS(500);
    // 使用当前时间初始化变量xLastWakeTime ,注意这和vTaskDelay()函数不同
    xLastWakeTime = xTaskGetTickCount();
    for( ;; )  {
        /* 调用系统延时函数,周期性阻塞500ms */
        vTaskDelayUntil( &xLastWakeTime,xFrequency );
         //  ...
         //  这里为任务主体代码,周期性执行.注意这和vTaskDelay()函数也不同
         //  ...
    }
}
```

当任务B获取CPU使用权后，先调用系统延时函数vTaskDelayUntil()使任务进入阻塞状态。任务B进入阻塞后，其它任务得以执行。

FreeRTOS内核会周期性的检查任务B的阻塞是否达到，如果阻塞时间达到，则将任务B设置为就绪状态。由于任务B的优先级最高，会抢占CPU，接下来执行任务主体代码。任务主体代码执行完毕后，会继续调用系统延时函数vTaskDelayUntil()使任务进入阻塞状态，周而复始。

从调用函数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

```
void vTaskDelayUntil( TickType_t * const pxPreviousWakeTime, const
TickType_t
xTimeIncrement )
{
TickType_t xTimeToWake;
BaseType_t xAlreadyYielded, xShouldDelay = pdFALSE;

    vTaskSuspendAll();
    {
        /* 保存系统节拍中断次数计数器 */
```

```
        const TickType_t xConstTickCount = xTickCount;

        /* 任务主体执行时长 */
        /* (xConstTickCount - *pxPreviousWakeTime) */
        /* 保证任务执行时长小于周期 */
        /* (xConstTickCount - *pxPreviousWakeTime) < xTimeIncrement */
        /* xConstTickCount < xTimeToWake */

        /* 计算任务下次唤醒时间(以系统节拍中断次数表示)   */
        xTimeToWake = *pxPreviousWakeTime + xTimeIncrement;
        /* *pxPreviousWakeTime中保存的是上次唤醒时间,唤醒后需要一定时间执行任务主体代码,如
果上次唤醒时间大于当前时间,说明节拍计数器溢出了 */
        /*------------内核计时器溢出----------------------*/
        if( xConstTickCount < *pxPreviousWakeTime )
        {
            /*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.*/
            /* xTimeToWake溢出,但符合正常延时场景 */
            if( ( xTimeToWake < *pxPreviousWakeTime ) && ( xTimeToWake > xConstTickCount ) )
            {
                xShouldDelay = pdTRUE;
            }
        }
        else
        {
            /* 也都是保证周期性延时时间大于任务主体代码执行时间 */
            /*-----------------内核计时器无溢出----------------*/
            /* 满足任意一种延时场景: 所有无溢出 / xTimeToWake溢出但符合正常延时场景 */
            if( ( xTimeToWake < *pxPreviousWakeTime ) || ( xTimeToWake > xConstTickCount ) )
            {
                xShouldDelay = pdTRUE;
            }
        }

        /* 更新唤醒时间,为下一次调用本函数做准备. */
        *pxPreviousWakeTime = xTimeToWake;

        if( xShouldDelay != pdFALSE )
        {
            /* 将本任务加入延时列表,注意阻塞时间并不是以当前时间为参考,因此减去了当前系统节拍
中断计数器值*/
            prvAddCurrentTaskToDelayedList( xTimeToWake - xConstTickCount, pdFALSE );
        }
    }
    xAlreadyYielded = xTaskResumeAll();
    /* 强制执行一次上下文切换 */
```

```
    if( xAlreadyYielded == pdFALSE )
    {
        portYIELD_WITHIN_API();
    }
}
```

本函数增加了一个参数pxPreviousWakeTime用于指向一个变量，变量保存上次任务解除阻塞的时间。这个变量在任务开始时必须被设置成当前系统节拍中断次数，此后函数vTaskDelayUntil()在内部自动更新这个变量

由于变量xTickCount可能会溢出，所以必须检测各种溢出情况，并且要保证延时周期不得小于任务主体代码执行时间

其实使用vTaskDelayUntil也不能保证任务能周期运行,如果使用函数vTaskDelayUntil()只能保证按照一定的周期性阻塞,进入就绪态,如果有更高优先级或者中断的话还是得等待其他事件

6.3内核时钟节拍

FreeRTOS的系统节拍为全局参数 xTickCount,在函数xTaskIncrementTick()中进行,在tasks.c中定义

```
BaseType_t xTaskIncrementTick( void )
{
TCB_t * pxTCB;
TickType_t xItemValue;
BaseType_t xSwitchRequired = pdFALSE;
    traceTASK_INCREMENT_TICK( xTickCount );
    //判断任务调度器是否挂起
    if( uxSchedulerSuspended == ( UBaseType_t ) pdFALSE )
    {
        //内核计时器+1
        const TickType_t xConstTickCount = xTickCount + 1;
        xTickCount = xConstTickCount;
        //计时器是否溢出
        if( xConstTickCount == ( TickType_t ) 0U )
        {
            //如果溢出将延时列表指针和溢出列表指针pxOverflowDelayedTaskList所指向的列表进行交换,更新xNestTaskUnblockTime
            taskSWITCH_DELAYED_LISTS();
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
        //变量xNestTaskUnblockTime保存着下一个要解除阻塞的任务时间点值,如果xConstTickCount大于xNestTaskUnblockTime的话说明有任务需要解除阻塞了
        if( xConstTickCount >= xNextTaskUnblockTime )
        {
            for( ;; )
            {
                if( listLIST_IS_EMPTY( pxDelayedTaskList ) != pdFALSE )
                {
```

```
                    xNextTaskUnblockTime = portMAX_DELAY;
                    break;
                }
                else
                {
                    //如果不为空,获取延时列表第一个列表项的TCB
                    pxTCB = ( TCB_t * )
listGET_OWNER_OF_HEAD_ENTRY(pxDelayedTaskList );
                    //获取对应TCB的状态列表
                    xItemValue = listGET_LIST_ITEM_VALUE( &( pxTCB-
>xStateListItem ) );
                    //TCB的状态列表保存了任务的唤醒时间点,如果大于当前的节拍,说
明还没有到
                    if( xConstTickCount < xItemValue )
                    {
                        //任务延时时间为到,而且xItemValue已经保存了下一个要唤
醒的任务的唤醒时间,所以要用xItemValue更新xNextTaskUnblockTime
                        xNextTaskUnblockTime = xItemValue;
                        break;
                    }
                    else
                    {
                        mtCOVERAGE_TEST_MARKER();
                    }
                    //延时时间到了,所以先从延时列表中移除
                    ( void ) uxListRemove( &( pxTCB->xStateListItem ) );
                    //检查是否等待某个时间,如果还在等待的话就将任务从相应的时间
列表中移除,因为超时时间到了
                    if( listLIST_ITEM_CONTAINER( &( pxTCB->xEventListItem )
) != NULL )
                    {
                        //将任务从相应的事件列表中移除
                        ( void ) uxListRemove( &( pxTCB->xEventListItem )
);
                    }
                    else
                    {
                        mtCOVERAGE_TEST_MARKER();
                    }
                    //任务延时时间到了,将任务添加到就绪列表
                    prvAddTaskToReadyList( pxTCB );
                    #if (  configUSE_PREEMPTION == 1 )
                    {
                //延时时间到的任务优先级要与正在运行的任务大,标记
xSwitchRequired为pdTRUE,表示需要进行切换
                        if( pxTCB->uxPriority >= pxCurrentTCB->uxPriority )
                        {
                            xSwitchRequired = pdTRUE;
                        }
                        else
                        {
                            mtCOVERAGE_TEST_MARKER();
```

```
                        }
                    }
                    #endif /* configUSE_PREEMPTION */
                }
            }
        }
        #if ( ( configUSE_PREEMPTION == 1 ) && ( configUSE_TIME_SLICING ==
1 ) )
        {
            //如果使能了时间片调度,还要处理跟时间片调度的有关工作
            if( listCURRENT_LIST_LENGTH( &( pxReadyTasksLists[
pxCurrentTCB->uxPriority ] ) ) > ( UBaseType_t ) 1 )
            {
                xSwitchRequired = pdTRUE;
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        #endif
        #if ( configUSE_TICK_HOOK == 1 )
        {
            //如果使能了时间片钩子函数就要执行函数vApplicationTickHook(),用户
编写
            if( uxPendedTicks == ( UBaseType_t ) 0U )
            {
                vApplicationTickHook();
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        #endif /* configUSE_TICK_HOOK */
    }
    else
    {
        //如果调用函数vTaskSuspendAll()挂起了调度器的话,在每个systick中断就不会
更新节拍了,取而代之的是uxPendedTicks来记录挂起节拍,这样在调用xTaskResumeAll()恢
复调度器的时候就会调用uxPendedTicks此函数xTaskIncrementTick(),这样xTickCount就
会恢复,并且那些应该取消阻塞的任务都会取消阻塞
        ++uxPendedTicks;
        #if ( configUSE_TICK_HOOK == 1 )
        {
            vApplicationTickHook();
        }
        #endif
    }
    #if ( configUSE_PREEMPTION == 1 )
    {
        //有时候调用其他API函数会使用变量xYieldPending来标记是否切换
```

```
        if( xYieldPending != pdFALSE )
        {
            xSwitchRequired = pdTRUE;
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    #endif /* configUSE_PREEMPTION */
    return xSwitchRequired;
}
```