FreeRTOS思路梳理

前置知识

中断屏蔽：芯片会有个中断屏蔽相关的寄存器，往这个寄存器中写入一个值，芯片会自动将优先级小于这个值的中断屏蔽掉

滴答定时器：为OS提供心跳，计时，本质仍然是中断

PendSV异常：本质是中断，它的特点是可以缓期执行——在ISR被占用时可以暂时挂起

PSP和MSP：有些芯片会提供有两个堆栈指针，MSP用于OS内核和中断异常处理，PSP用于任务

列表：可以看成是双向链表

列表项：可以看成列表里面的节点

关于任务调度

在FreeRTOS中，任务被抽象成了一个TCB_t结构体，这个结构体中有这个任务所有的信息

在TCB中有两个列表项，一个表示状态，一个表示事件

列表项中存储一个指向所属TCB的指针，所以移动列表项就可以完成对TCB状态的更改——就像衣架，移动钩子，钩子下面的衣服也就跟着移动了

同时列表项中还有一个指向所属列表的指针，它一般用来表示在等待事件、信号量、队列等

FreeRTOS定义了很多种列表，主要分状态列表和事件列表，列表按value值排序，value表示被处理的优先度

状态列表主要是用来表示任务状态，有就绪列表、等待延时列表、挂起列表和等待唤醒列表

唤醒任务、阻塞任务等操作均是通过操作TCB中的列表项到相应的列表中完成

FreeRTOS在PendSV中执行任务调度和任务切换，在每一次进入滴答定时器中断后，会自动悬起一个PendSV中断（P.S.因为PendSV优先级允许缓期执行，所以会ISR未被占用后进行任务调度，适合做任务调度）

在PendSV中断中，OS直接选择就绪列表中优先级最高的那个任务进行切换执行；在任务切换时，会进行上下文切换，即保护现场（将运行环境，寄存器变量等压入任务堆栈中），恢复现场（从任务堆栈中按规则取出数据和寄存器的值进行恢复）

当任务阻塞时，如果有等待时间则将任务状态TCB移动到延时等待列表中，延时列表根据延时时间排序

在每个滴答定时器中断中都会更新系统时间，并检查是否有延时到的任务需要唤醒，唤醒流程——将列表项移到就绪列表中，将此TCB的任务列表项归属置NULL

关于队列和信号量

队列是利用一片内存来存储数据，队列结构体中有两个列表，一个表示入队阻塞的任务（队列满了），一个表示出队阻塞的任务（队列为空）

队列为空时，要出队的任务的事件列表项将被挂载在队列中的列表中，如有等待时间移动状态列表项到延时等待列表中，如果死等则挂起任务

队列满时，入队的任务操作流程相同

当可以入队时，将消息拷贝到消息队列中，然后按优先级唤醒相应因出队阻塞的任务；可以入队时，操作类似

各种信号量底层均是队列，利用队列完成信号量的上锁等待等操作

互斥信号量较其他信号量多了优先级翻转，在等待时，互斥信号量会将正在持有互斥量的任务优先级提高到和自己一样

关于事件组

事件组的实现和队列类似，事件组的结构体中有一个列表，列表存储等待事件组某些事件发生的任务

当任务调用等待某一事件组的某些事件时，会将此任务的事件列表项挂载在此列表下，列表项的value值存储等待哪些事件的发生，同时将其状态切换为延时阻塞或挂起

当事件组被写入某些事件时，遍历列表中的所有任务，查看是否满足要求，满足则唤醒，最后根据要求清除相应的事件标志位

关于任务通知

修改TCB中的元素的值和任务通知的状态变量

中断

切换任务，任务执行均在中断中进行

FreeRTOS中定义了一些宏来管理中断，优先级低于此宏的RTOS可以管理，高于此宏的RTOS无法管理——宏：configMAX_SYSCALL_INTERRUPT_PRIORITY

临界段保护：屏蔽RTOS可以管理的那些中断之后进入临界段，临界段代码要快，因为低优先级的中断被屏蔽后无法响应

中断屏蔽：msr basepri, ulNewBASEPRI；将configMAX_SYSCALL_INTERRUPT_PRIORITY的值写入basepri寄存器中

中断级关中断不可以嵌套，任务级运行嵌套

在任务级中，有变量记录调用开关中断函数的次数，只有开中断和关中断数相等时才会开启中断；但是任务级没有应用这个变量

开关中断

```
#define taskENTER_CRITICAL()        portENTER_CRITICAL()
#define taskENTER_CRITICAL_FROM_ISR() portSET_INTERRUPT_MASK_FROM_ISR()

#define taskEXIT_CRITICAL()         portEXIT_CRITICAL()
#define taskEXIT_CRITICAL_FROM_ISR( x ) portCLEAR_INTERRUPT_MASK_FROM_ISR( x )

#define taskDISABLE_INTERRUPTS()    portDISABLE_INTERRUPTS()
#define taskENABLE_INTERRUPTS()     portENABLE_INTERRUPTS()

#define portDISABLE_INTERRUPTS()                vPortRaiseBASEPRI()
#define portENABLE_INTERRUPTS()                 vPortSetBASEPRI( 0 )
#define portENTER_CRITICAL()                    vPortEnterCritical()
```

```
#define portEXIT_CRITICAL()                     vPortExitCritical()
#define portSET_INTERRUPT_MASK_FROM_ISR()       ulPortRaiseBASEPRI()
#define portCLEAR_INTERRUPT_MASK_FROM_ISR(x)    vPortSetBASEPRI(x)
```

```
static portFORCE_INLINE void vPortSetBASEPRI( uint32_t ulBASEPRI )
{
    __asm
    {
        /* Barrier instructions are not used as this function is only used to
        lower the BASEPRI value. */
        msr basepri, ulBASEPRI
    }
}

static portFORCE_INLINE void vPortRaiseBASEPRI( void )
{
uint32_t ulNewBASEPRI = configMAX_SYSCALL_INTERRUPT_PRIORITY;

    __asm
    {
        /* Set BASEPRI to the max syscall priority to effect a critical
        section. */
        msr basepri, ulNewBASEPRI
        dsb
        isb
    }
}

static portFORCE_INLINE void vPortClearBASEPRIFromISR( void )
{
    __asm
    {
        /* Set BASEPRI to 0 so no interrupts are masked.  This function is only
        used to lower the mask in an interrupt, so memory barriers are not
        used. */
        msr basepri, #0
    }
}

static portFORCE_INLINE uint32_t ulPortRaiseBASEPRI( void )
{
uint32_t ulReturn, ulNewBASEPRI = configMAX_SYSCALL_INTERRUPT_PRIORITY;

    __asm
    {
        /* Set BASEPRI to the max syscall priority to effect a critical
        section. */
        mrs ulReturn, basepri
        msr basepri, ulNewBASEPRI
        dsb
        isb
    }

```

```
    return ulReturn;
}
```

操作basepri寄存器开启或屏蔽低于basepri的中断

systick和pendsv中断

systick用于记录时间，为OS提供心跳计时

pendsv用于执行任务切换，每出发一次systick中断均悬挂一个pendsv中断，当检测到无中断服务在运行时进行任务切换——缓期执行

因为，如果在systick中进行任务切换并保证定时就无法同时保证实时系统的运行和定时的准确性，会对中断产生延时，因为在任务切换中会屏蔽掉中断，无法及时响应外部中断，这是系统不允许的

滴答定时器如果拥有最高优先级，因为其会周期性触发，会经常抢占外部中断，会导致外部中断的相应变慢，这是不可忍受的，想象一下，外部中断是按键或者串口，按键卡顿或者串口接收数据接收到一半跳出；相发，如果设为最低，可能因为外部中断的触发导致定时会不准确，但这是可以忍受的，影响较小，可以先接收数据，之后在一定时间内处理即可

任务切换中断要求拥有最低优先级，因为如果非最低优先级，在系统进入某个中断时，pendsv中断触发，系统进行任务切换，然后执行下一个任务了，无法再回到被打断的中断服务中

所以需要两个中断保证系统时间的准确和中断响应实时性，而且为了外部中断的及时相应要将其均设为最低优先级

列表和列表项

列表可以看成是一个循环双指针的链表

列表：用于调度FreeRTOS中的任务，列表结构体：

```
typedef struct xLIST
{
    listFIRST_LIST_INTEGRITY_CHECK_VALUE                /*< Set to a known value if configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
    configLIST_VOLATILE UBaseType_t uxNumberOfItems; //列表项数量
    ListItem_t * configLIST_VOLATILE pxIndex;       //当前列表项，可以用来遍历列表
    MiniListItem_t xListEnd;                        //迷你列表项，列表项裁剪掉一部分，有些简单的程序不需要完整的列表；还用于表示列表结束
    listSECOND_LIST_INTEGRITY_CHECK_VALUE               /*< Set to a known value if configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
} List_t;
```

列表项：

```c
struct xLIST_ITEM
{
    listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE           /*< Set to a known value if configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
    configLIST_VOLATILE TickType_t xItemValue;          /*< The value being listed.  In most cases this is used to sort the list in descending order. */
    struct xLIST_ITEM * configLIST_VOLATILE pxNext;     //下一个列表项
    struct xLIST_ITEM * configLIST_VOLATILE pxPrevious; //上一个列表项
    void * pvOwner;                                     //指向这个列表项属于哪个任务控制块，属于哪个任务
    void * configLIST_VOLATILE pvContainer;             //指向这个列表项属于哪个列表，列表种类：信号量、事件等列表
    listSECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE          /*< Set to a known value if configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
};
typedef struct xLIST_ITEM ListItem_t;
```

列表主要用于跟踪调度任务，列表项指向任务控制块和所属列表，任务控制块中有两个列表项，分别用来描述任务的状态和事件的状态，当任务状态列表项处于就绪列表且优先级最高，在下一次任务切换时，执行其指向的任务。

任务

任务状态

运行态、就绪态、阻塞态、挂起态

任务控制块

每个任务均有一些属性，RTOS将任务抽象成任务控制块结构体

里面存储有任务的堆栈指针，临界区信息，任务优先级，任务的状态和事件列表项等

任务的创建流程

1. 关闭中断
2. 创建任务 xTaskCreate()
3. 开启中断

动态创建：程序自动分配内存，初始化控制块

静态创建：用户自行分配内存

任务创建函数 xTaskCreate

1. 申请堆栈内存和TCB内存
2. 初始化新任务 prvInitialiseNewTask
3. 将新任务加入到就绪列表中 prvAddNewTaskToReadyList

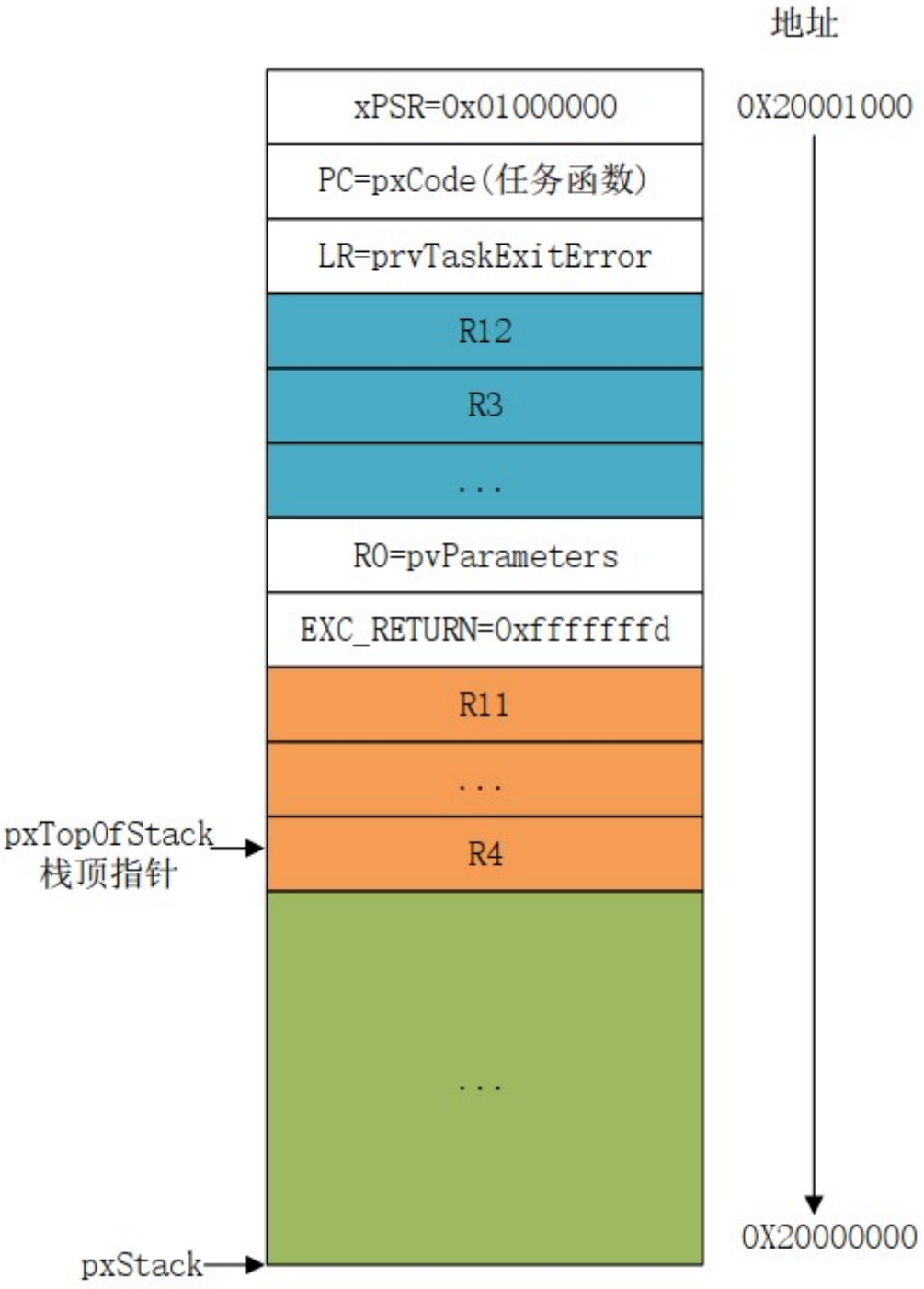
初始化新任务 prvInitialiseNewTask

1. 初始化堆栈
2. 保存任务名
3. 判断任务优先级是否合法
4. 初始化优先级
5. 互斥量相关参数初始化
6. 列表项初始化——状态和事件列表项（设置pvOwner为当前TCB）
7. 事件列表项的value值初始化为configMAX_PRIORITIES - uxPriority
8. 初始化堆栈 pxPortInitialiseStack
9. 任务句柄赋值为当前TCB指针

堆栈初始化 pxPortInitialiseStack

1. 将xPSR寄存器要初始化的值和任务函数地址压入栈中——xPSR寄存器和PC初始化值
2. 将prvTaskExitError压入栈中——LR的值初始化为prvTaskExitError
3. 跳过4个寄存器，R12,R3,R2,R1；栈顶指针减5
4. 将输入参数指针压入栈中——R0初始化为输入参数
5. 跳过8个寄存器，R11, R10, R9, R8, R7, R6, R5 and R4；栈顶指针减8
6. 更新栈顶指针

堆栈结构示例（STM32 M3）



添加新任务到就绪列表

列表项

```
/* Lists for ready and blocked tasks. --------------------*/
PRIVILEGED_DATA static List_t pxReadyTasksLists[ configMAX_PRIORITIES ];/*< Prioritised ready tasks. */
PRIVILEGED_DATA static List_t xDelayedTaskList1;                        /*< Delayed tasks. */
PRIVILEGED_DATA static List_t xDelayedTaskList2;                        /*< Delayed tasks (two lists are used - one for delays that have overflowed the current tick count. */
PRIVILEGED_DATA static List_t * volatile pxDelayedTaskList;             /*< Points to the delayed task list currently being used. */
PRIVILEGED_DATA static List_t * volatile pxOverflowDelayedTaskList;     /*< Points to the delayed task list currently being used to hold tasks that have overflowed the current tick count. */
PRIVILEGED_DATA static List_t xPendingReadyList;                        /*< Tasks that have been readied while the scheduler was suspended.  They will be moved to the ready list when the scheduler is resumed. */
```

列表数组pxReadyTasksLists[]为任务就绪列表，数组中的每个列表表示相同优先级的就绪任务列表

pxDelayedTaskList为延时列表，pxOverflowDelayedTaskList为延时溢出列表这两个指向xDelayedTaskList1和xDelayedTaskList2，实现溢出处理

xPendingReadyList唤醒任务列表

prvAddNewTaskToReadyList

1. 关中断 taskENTER_CRITICAL
2. 判断当前任务TCB，如果没有则将新任务TCB赋给pxCurrentTCB
3. 如果是第一个任务，初始化相应的列表 prvInitialiseTaskLists——ListItem_t结构体的初始化
4. 如果当前任务控制块有有任务，且未开启任务调度器，比较优先级，选择优先级高的赋值给pxCurrentTCB
5. 任务ID赋值 pxNewTCB->uxTCBNumber = uxTaskNumber;
6. 将当前TCB添加到就绪列表中 prvAddTaskToReadyList
7. 开中断
8. 如果开启了任务调度器，比较新任务的优先级和当前任务的优先级，新任务的优先级大于当前任务的优先级，进行任务切换 taskYIELD_IF_USING_PREEMPTION

prvAddTaskToReadyList

```
#define prvAddTaskToReadyList( pxTCB )
\
    traceMOVED_TASK_TO_READY_STATE( pxTCB );
\
    taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority );
\
    vListInsertEnd( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &( ( pxTCB )->xStateListItem ) ); \
    tracePOST_MOVED_TASK_TO_READY_STATE( pxTCB )
```

行2：空

行3：将优先级存储进uxReadyPriorities的位图中

行4：将（pxTCB)->xStateListItem 一个列表项插入到pxReadyTasksLists[(pxTCB)->uxPriority]列表的末尾

行5：空

任务删除流程 vTaskDelete

1. 关中断 taskENTER_CRITICAL
2. 从就绪列表中移除列表项pxTCB->xStateListItem uxListRemove(&(pxTCB->xStateListItem))
3. 判断其是否在等待某个事件，在某个事件列表中，是的话删除 listLIST_ITEM_CONTAINER(&(pxTCB->xEventListItem)) != NULL
4. 判断是否正在运行 pxTCB == pxCurrentTCB
    1. 将其加入到等待删除的列表中 vListInsertEnd(&xTasksWaitingTermination, &(pxTCB->xStateListItem));
    2. 释放内存的任务数量加一，任务资源会在空闲任务中释放
5. 要删除的任务并未运行则直接释放资源 prvDeleteTCB
6. 更新下一任务的解锁事件——还有多久到下一任务 prvResetNextTaskUnblockTime
7. 开中断 taskEXIT_CRITICAL
8. 如果任务调度器开启并且要删除的任务正在执行pxTCB == pxCurrentTCB，执行任务切换

任务挂起流程 vTaskSuspend

1. 关中断 taskENTER_CRITICAL
2. 从就绪列表中移除列表项pxTCB->xStateListItem uxListRemove(&(pxTCB->xStateListItem))
3. 判断是否在等待某个事件，如果在某个事件的等待列表中，则移除它
4. 将其插入到挂起列表的末端 vListInsertEnd(&xSuspendedTaskList, &(pxTCB->xStateListItem));
5. 开中断
6. 更新下一任务的解锁事件——还有多久到下一任务 prvResetNextTaskUnblockTime
7. 如果任务调度器开启了且要挂起的任务为当前任务，进行一次任务切换；如果任务调度器关闭了，但是当前任务为要挂起的任务，手动寻找下一个要运行的任务并进行切换

任务恢复流程 vTaskResume

1. 获取任务控制块
2. 关中断 taskENTER_CRITICAL
3. 判断任务是否被挂起 prvTaskIsTaskSuspended
4. 从挂起的列表中移除列表项 (void) uxListRemove(&(pxTCB->xStateListItem));
5. 将其添加到就绪列表中 prvAddTaskToReadyList(pxTCB);
6. 根据优先级决定是否进行任务切换，如果恢复的任务优先级较当前任务优先级更高则任务切换，反之不切换
7. 开中断 taskEXIT_CRITICAL

任务调度器

任务调度器的开启 vTaskStartScheduler

1. 创建空闲任务和软件定时器
2. 关中断 portDISABLE_INTERRUPTS();
3. 允许运行任务调度器 xSchedulerRunning = pdTRUE;
4. 开启任务调度器，初始化相关硬件 xPortStartScheduler()

开启任务调度器，初始化 xPortStartScheduler

1. 设置pendsv和滴答定时器优先级为最低优先级
2. 设置滴答定时器的周期
3. 初始化临界区嵌套计时器
4. 开始第一个任务 prvStartFirstTask();

开启第一个任务 prvStartFirstTask

1. 获取MSP的初始值，复位MSP
2. 使能中断
3. 数据和指令同步（保证数据和指令的写入和执行，而非在缓冲区）
4. 触发SVC中断

SVC中断服务函数

中断服务函数：SVC_Handler()

1. 取当前任务的TCB块，获取栈顶指针赋值给r0
2. 寄存器出栈
3. 进程栈PSP指针初始化为任务堆栈
4. 指令同步
5. 开启中断

空闲任务

在vTaskStartScheduler函数中会创建一个空闲任务，这个任务优先级最低，功能有：

1. 判断系统是否有任务删除
2. 运行用户设置的空闲任务钩子函数
3. 低功耗模式的开启与处理

调度器的挂起和恢复 vTaskSuspendAll xTaskResumeAll

vTaskSuspendAll：任务挂起所有主要是使得滴答定时器不在更新系统时钟节拍数（xTickCount），系统就会认为时间仍未达到唤醒其他任务的时间，这样就不会进行任务切换了；挂起时候用uxPendedTicks来记录时间节拍；同时将任务调度器挂起，即将更改标志位

xTaskResumeAll：相反，恢复xTickCount计数时钟节拍数；恢复任务调度器，即将更改标志位

任务切换

任务堆栈与保护恢复现场

RTOS对每个任务均维护有一个堆栈，在进行任务调度时，RTOS会将任务的运行环境（程序指针，寄存器或变量的值等）信息压入堆栈中，当任务再次运行时，调度器会从而堆栈中取出数据，还原当时的运行环境；M3核有PSP：任务堆栈指针，会自动恢复部分寄存器的值

PendSV异常

PendSV（可挂起异常）与SVC不同，它是不精确的，可以子啊更高优先级的异常处理内挂起它，当更高优先级处理完成后PendSV执行。

FreeRTOS在PendSV中断中完成任务切换

任务切换的场合

1. 系统调用

```
#define taskYIELD()         portYIELD()
#define portYIELD()
\
{
\
    /* Set a PendSV to request a context switch. */
\
    portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
\

\
    /* Barriers are normally not required but do ensure the code is
completely  \
    within the specified behaviour for the architecture. */
\
    __dsb( portSY_FULL_READ_WRITE );
\
    __isb( portSY_FULL_READ_WRITE );
\
}
#define portNVIC_PENDSVSET_BIT      ( 1UL << 28UL )
#define portNVIC_INT_CTRL_REG       ( * ( ( volatile uint32_t * )
0xe000ed04 ) )
```

将0xe000ed04地址的第28位置1，挂起一个PendSV中断

2. 滴答定时器中断触发

滴答定时器中断服务函数：

```
void SysTick_Handler(void)
{
    //判断任务调度器状态
    if(xTaskGetSchedulerState()!=taskSCHEDULER_NOT_STARTED)
    {
        xPortSysTickHandler();
    }
}

void xPortSysTickHandler( void )
{
    //关中断
    vPortRaiseBASEPRI();
    {
        //更新时钟节拍计数器，并检查是否有任务需要取消阻塞 xTaskIncrementTick
        if( xTaskIncrementTick() != pdFALSE )
        {
            //挂起一个PendSV中断，通过操作寄存器
            portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
        }
    }
```

```
    //开中断
    vPortClearBASEPRIFromISR();
}
```

更新时钟节拍计数器，并检查是否有任务需要取消阻塞 xTaskIncrementTick

PendSV中断服务函数

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

    PRESERVE8

    mrs r0, psp                 //读取任务栈指针
    isb

    ldr r3, =pxCurrentTCB       //获取当前任务TCB地址
    ldr r2, [r3]

    stmdb r0!, {r4-r11}         //将寄存器r4-r11的值压入栈中
    str r0, [r2]                //将新的栈顶指针保存到r0

    stmdb sp!, {r3, r14}
    mov r0, #configMAX_SYSCALL_INTERRUPT_PRIORITY
    msr basepri, r0             //关中断
    dsb
    isb                         //数据和指令同步
    bl vTaskSwitchContext       //加载函数，获取下一个要运行的任务，并更新
pxCurrentTCB
    mov r0, #0
    msr basepri, r0             //开中断
    ldmia sp!, {r3, r14}

    ldr r1, [r3]
    ldr r0, [r1]                //获取新任务的任务栈顶
    ldmia r0!, {r4-r11}         //恢复寄存器r4-r11的值
    msr psp, r0                 //将新的栈顶赋值给任务栈指针，其他的寄存器值会自动恢
复
    isb
    bx r14
    nop
}
```

获取下一个要运行的任务

```
void vTaskSwitchContext( void )
{
    //判断任务调度器状态
    if( uxSchedulerSuspended != ( UBaseType_t ) pdFALSE )
    {
```

```
        /* The scheduler is currently suspended - do not allow a context
switch. */
        xYieldPending = pdTRUE;
    }
    else
    {
        xYieldPending = pdFALSE;
        traceTASK_SWITCHED_OUT();

        /* Check for stack overflow, if configured. */
        taskCHECK_FOR_STACK_OVERFLOW();

        //获取下一个要运行的任务
        taskSELECT_HIGHEST_PRIORITY_TASK();
        traceTASK_SWITCHED_IN();
    }
}
```

- 获取下一个要运行的任务分为通用方法和硬件方式

通用方法

```
#define taskSELECT_HIGHEST_PRIORITY_TASK()
\
{
\
    UBaseType_t uxTopPriority = uxTopReadyPriority;
\
    /* Find the highest priority queue that contains ready tasks. */
\
    while( listLIST_IS_EMPTY( &( pxReadyTasksLists[ uxTopPriority ] ) ) )
\
    {
\
        configASSERT( uxTopPriority );
\
        --uxTopPriority;
\
    }
\

    /* listGET_OWNER_OF_NEXT_ENTRY indexes through the list, so the tasks of
\
    the same priority get an equal share of the processor time. */
\
    listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &( pxReadyTasksLists[
uxTopPriority ] ) );           \
    uxTopReadyPriority = uxTopPriority;
\
} /* taskSELECT_HIGHEST_PRIORITY_TASK */
```

uxTopReadyPriority存储就绪列表的最高优先级，从就绪列表数组中取出一个任务块，将其赋给pxCurrentTCB，移动当前列表指针

硬件方法

```
#define taskSELECT_HIGHEST_PRIORITY_TASK()
\
{
\
    UBaseType_t uxTopPriority;
\
    /* Find the highest priority list that contains ready tasks. */
\
    portGET_HIGHEST_PRIORITY( uxTopPriority, uxTopReadyPriority );
\
    configASSERT( listCURRENT_LIST_LENGTH( &( pxReadyTasksLists[ uxTopPriority ] ) ) > 0 );     \
    listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &( pxReadyTasksLists[ uxTopPriority ] ) );       \
} /* taskSELECT_HIGHEST_PRIORITY_TASK() */
```

portGET_HIGHEST_PRIORITY：获取就绪态的最高优先级

从就绪列表数组中取出一个任务块，将其赋给pxCurrentTCB

优先级数量受位的影响，因为优先级用位图表示

时间片轮转

启用了时间片轮转的宏时，在滴答定时器中断中会对就绪列表中当前优先级任务数量进行判断，如果其他任务则进行任务切换

时间管理

延时函数 vTaskDelay

表示延时多少个系统节拍

1. 挂起任务调度器
2. 将延时任务添加到延时列表中 prvAddCurrentTaskToDelayedList
3. 恢复任务调度器
4. 任务切换

将延时任务添加到延时列表中

```
static void prvAddCurrentTaskToDelayedList( TickType_t xTicksToWait, const BaseType_t xCanBlockIndefinitely )
{
    TickType_t xTimeToWake;
    //获取当前时钟节拍值
    const TickType_t xConstTickCount = xTickCount;

    //将当前任务列表项从就绪列表中移除
    if( uxListRemove( &( pxCurrentTCB->xStateListItem ) ) == ( UBaseType_t ) 0 )
    {
        /* 当前任务必须在就绪列表中，因此无需检查，可以直接调用端口重置宏 */
```

```
        portRESET_READY_PRIORITY( pxCurrentTCB->uxPriority,
uxTopReadyPriority );
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    #if ( INCLUDE_vTaskSuspend == 1 )
    {
        //判断
        if( ( xTicksToWait == portMAX_DELAY ) && ( xCanBlockIndefinitely !=
pdFALSE ) )
        {
            /* 如果等待时间为最大值，且可以被阻塞；挂起任务，将任务挂载到挂起列表末端 */
            vListInsertEnd( &xSuspendedTaskList, &( pxCurrentTCB-
>xStateListItem ) );
        }
        else
        {
            //计算唤醒时间
            xTimeToWake = xConstTickCount + xTicksToWait;

            //将任务状态列表项的值设为唤醒时间
            listSET_LIST_ITEM_VALUE( &( pxCurrentTCB->xStateListItem ),
xTimeToWake );

            //判断是否溢出
            if( xTimeToWake < xConstTickCount )
            {
                //xTimeToWake溢出，将任务移到溢出列表
（pxOverflowDelayedTaskList）中
                vListInsert( pxOverflowDelayedTaskList, &( pxCurrentTCB-
>xStateListItem ) );
            }
            else
            {
                //没有溢出，移到延时列表（pxDelayedTaskList）中
                vListInsert( pxDelayedTaskList, &( pxCurrentTCB-
>xStateListItem ) );

                //更新xNextTaskUnblockTime（取消延时，唤醒任务的最近时刻）
                if( xTimeToWake < xNextTaskUnblockTime )
                {
                    xNextTaskUnblockTime = xTimeToWake;
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
        }
    }
}
```

OS系统时钟节拍

```
BaseType_t xTaskIncrementTick( void )
{
    TCB_t * pxTCB;
    TickType_t xItemValue;
    BaseType_t xSwitchRequired = pdFALSE;

    /* 每个时钟节拍中断调用一次本函数，增加时钟节拍计数器xTickCount的值，并检查是否有任务需要取消阻塞 */
    traceTASK_INCREMENT_TICK( xTickCount );
    if( uxSchedulerSuspended == ( UBaseType_t ) pdFALSE )
    {
        /* 增加系统时钟节拍计数器 */
        const TickType_t xConstTickCount = xTickCount + 1;

        xTickCount = xConstTickCount;

        //如果溢出则交换延时列表指针和溢出列表指针
        if( xConstTickCount == ( TickType_t ) 0U )
        {
            taskSWITCH_DELAYED_LISTS();
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }

        /* 查看是否有任务延时时间到了 */
        if( xConstTickCount >= xNextTaskUnblockTime )
        {
            for( ;; )
            {
                if( listLIST_IS_EMPTY( pxDelayedTaskList ) != pdFALSE )
                {
                    /* 延时列表为空，设置xNextTaskUnblockTime为最大值 */
                    xNextTaskUnblockTime = portMAX_DELAY;
                    break;
                }
                else
                {
                    /* 延时列表不为空 */
                    pxTCB = ( TCB_t * ) listGET_OWNER_OF_HEAD_ENTRY( pxDelayedTaskList );
                    xItemValue = listGET_LIST_ITEM_VALUE( &( pxTCB->xStateListItem ) );

                    /* 判断任务延时是否到了 */
                    if( xConstTickCount < xItemValue )
                    {
                        /* 没到，更新xNextTaskUnblockTime */
                        xNextTaskUnblockTime = xItemValue;
                        break;
                    }
                    else
```

```
51                      {
52                          mtCOVERAGE_TEST_MARKER();
53                      }
54
55                      /* 到了，需要唤醒任务 */
56                      /* 将任务从延时列表中移出 */
57                      ( void ) uxListRemove( &( pxTCB->xStateListItem ) );
58
59                      /* 判断任务是否在等待其他事件，比如信号量等；如果有则移出相应列表
*/
60                      if( listLIST_ITEM_CONTAINER( &( pxTCB->xEventListItem )
) != NULL )
61                      {
62                          ( void ) uxListRemove( &( pxTCB->xEventListItem )
);
63                      }
64                      else
65                      {
66                          mtCOVERAGE_TEST_MARKER();
67                      }
68
69                      /* 将任务加入到就绪列表 */
70                      prvAddTaskToReadyList( pxTCB );
71
72                      /* 抢占式调度？ */
73                      #if (  configUSE_PREEMPTION == 1 )
74                      {
75                          /* 抢占式调度器的话，判断解除阻塞状态的任务优先级和当前任务的
优先级；解除的任务优先级更高的话进行一次任务切换 */
76                          if( pxTCB->uxPriority >= pxCurrentTCB->uxPriority )
77                          {
78                              xSwitchRequired = pdTRUE;
79                          }
80                          else
81                          {
82                              mtCOVERAGE_TEST_MARKER();
83                          }
84                      }
85                      #endif /* configUSE_PREEMPTION */
86                  }
87              }
88          }
89
90          /* 使能了时间片的话还需要处理同优先级下的任务之间的调度 */
91          #if ( ( configUSE_PREEMPTION == 1 ) && ( configUSE_TIME_SLICING ==
1 ) )
92          {
93              if( listCURRENT_LIST_LENGTH( &( pxReadyTasksLists[
pxCurrentTCB->uxPriority ] ) ) > ( UBaseType_t ) 1 )
94              {
95                  xSwitchRequired = pdTRUE;
96              }
97              else
98              {
99                  mtCOVERAGE_TEST_MARKER();
```

```
            }
        }
        #endif /* ( ( configUSE_PREEMPTION == 1 ) && (
configUSE_TIME_SLICING == 1 ) ) */

        //使能了时间钩子?
        #if ( configUSE_TICK_HOOK == 1 )
        {
            if( uxPendedTicks == ( UBaseType_t ) 0U )
            {
                vApplicationTickHook();
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        #endif /* configUSE_TICK_HOOK */
    }
    else
    {
        ++uxPendedTicks;

        /* The tick hook gets called at regular intervals, even if the
        scheduler is locked. */
        #if ( configUSE_TICK_HOOK == 1 )
        {
            vApplicationTickHook();
        }
        #endif
    }

    #if ( configUSE_PREEMPTION == 1 )
    {
        if( xYieldPending != pdFALSE )
        {
            xSwitchRequired = pdTRUE;
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    #endif /* configUSE_PREEMPTION */

    return xSwitchRequired;
}
```

队列

信号量均是用队列实现的，包括二值信号量，计数信号量，互斥信号量，递归互斥信号量

队列结构体

```
typedef struct QueueDefinition
{
    int8_t *pcHead;                 /* 指向队列存储区开始地址 */
    int8_t *pcTail;                 /* 指向队列存储区末尾 */
    int8_t *pcWriteTo;              /* 指向存储区下一个空闲区域 */

    union                           /* 用一个union，节省空间 */
    {
        int8_t *pcReadFrom;         /* 用作队列时，指向第一个出队的队列项首地址 */
        UBaseType_t uxRecursiveCallCount;/* 用作递归互斥量时，用来记录递归互斥量被调用的次数 */
    } u;

    List_t xTasksWaitingToSend;     /* 等待发送列表，因为队列满了而无法入队的任务挂在此列表上，按优先级顺序挂载 */
    List_t xTasksWaitingToReceive;  /* 等待接收队列，因为队列为而接收失败的任务挂在此列表上，按优先级顺序挂载 */

    volatile UBaseType_t uxMessagesWaiting;/* 目前队列中的消息数量 */
    UBaseType_t uxLength;           /* 队列存储区长度 */
    UBaseType_t uxItemSize;         /* 队列项最大长度，以字节为单位 */

    volatile int8_t cRxLock;        /* 队列上锁后，出队的队列项数量；当队列没有上锁，此字段为queueUNLOCKED */
    volatile int8_t cTxLock;        /* 队列上锁后，入队的队列项数量；当队列没有上锁，此字段为queueUNLOCKED */

    #if( ( configSUPPORT_STATIC_ALLOCATION == 1 ) && ( configSUPPORT_DYNAMIC_ALLOCATION == 1 ) )
        uint8_t ucStaticallyAllocated;  /* 静态存储的话，设置为pdTURE */
    #endif

    #if ( configUSE_QUEUE_SETS == 1 )   //队列集
        struct QueueDefinition *pxQueueSetContainer;
    #endif

    #if ( configUSE_TRACE_FACILITY == 1 )   //跟踪调试
        UBaseType_t uxQueueNumber;
        uint8_t ucQueueType;
    #endif

} xQUEUE;

typedef xQUEUE Queue_t;
```

队列创建

```
QueueHandle_t xQueueGenericCreate( const UBaseType_t uxQueueLength, const UBaseType_t uxItemSize, const uint8_t ucQueueType )
{
    Queue_t *pxNewQueue;
    size_t xQueueSizeInBytes;
    uint8_t *pucQueueStorage;

```

```
    configASSERT( uxQueueLength > ( UBaseType_t ) 0 );

    if( uxItemSize == ( UBaseType_t ) 0 )
    {
        /* 队列项大小为0，不需要存储区 */
        xQueueSizeInBytes = ( size_t ) 0;
    }
    else
    {
        /* 分配相应大小的存储区 */
        xQueueSizeInBytes = ( size_t ) ( uxQueueLength * uxItemSize );
    }

    pxNewQueue = ( Queue_t * ) pvPortMalloc( sizeof( Queue_t ) + xQueueSizeInBytes );

    /* 申请内存成功 */
    if( pxNewQueue != NULL )
    {
        pucQueueStorage = ( ( uint8_t * ) pxNewQueue ) + sizeof( Queue_t );

        #if( configSUPPORT_STATIC_ALLOCATION == 1 )
        {
            /* 动态方法创建，这个字段赋为pdFALSE */
            pxNewQueue->ucStaticallyAllocated = pdFALSE;
        }
        #endif /* configSUPPORT_STATIC_ALLOCATION */

        /* 初始化一个新的队列，为队列中的成员变量赋值，处理发送和接收列表 */
        prvInitialiseNewQueue( uxQueueLength, uxItemSize, pucQueueStorage, ucQueueType, pxNewQueue );
    }

    return pxNewQueue;
}
```

队列上锁

```
#define prvLockQueue( pxQueue )                                 \
    taskENTER_CRITICAL();                                       \
    {                                                           \
        if( ( pxQueue )->cRxLock == queueUNLOCKED )             \
        {                                                       \
            ( pxQueue )->cRxLock = queueLOCKED_UNMODIFIED;      \
        }                                                       \
        if( ( pxQueue )->cTxLock == queueUNLOCKED )             \
        {                                                       \
            ( pxQueue )->cTxLock = queueLOCKED_UNMODIFIED;      \
        }                                                       \
    }                                                           \
    taskEXIT_CRITICAL()
```

将cRxLock和cTxLock设置为queueLOCKED_UNMODIFIED

队列解锁

```
static void prvUnlockQueue( Queue_t * const pxQueue )
{
    /* 此函数必须在任务调度器挂起后调用 */

    /* 上锁计数器会记录上锁期间入队或出队的队列项数量 */
    /* 上锁后队列项可以加入或移出队列，但是相应的事件列表不会更新 */
    /* 解锁时要更新相应的事件列表 */
    taskENTER_CRITICAL();
    {
        int8_t cTxLock = pxQueue->cTxLock;

        /* 判断数据在上锁期间是否被加入到队列中 */
        while( cTxLock > queueLOCKED_UNMODIFIED )
        {
            /*****************************************************************/
            /**********************省略队列集代码**********************/
            /*****************************************************************/

            #else /* configUSE_QUEUE_SETS */
            {
                /* 将任务从事件列表中移除 */
                if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToReceive )
) == pdFALSE )
                {
                    if( xTaskRemoveFromEventList( &( pxQueue-
>xTasksWaitingToReceive ) ) != pdFALSE )
                    {
                        /* 从列表中移除任务优先级更高，进行一次任务切换 */
                        vTaskMissedYield();
                    }
                    else
                    {
                        mtCOVERAGE_TEST_MARKER();
                    }
                }
                else
                {
                    break;
                }
            }
            #endif /* configUSE_QUEUE_SETS */

            --cTxLock;
        }

        pxQueue->cTxLock = queueUNLOCKED;
    }
    taskEXIT_CRITICAL();

    /* 处理cRxLock */
    taskENTER_CRITICAL();
    {
        int8_t cRxLock = pxQueue->cRxLock;
```

```
52
53          while( cRxLock > queueLOCKED_UNMODIFIED )
54          {
55              if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToSend ) ) ==
pdFALSE )
56              {
57                  if( xTaskRemoveFromEventList( &( pxQueue-
>xTasksWaitingToSend ) ) != pdFALSE )
58                  {
59                      vTaskMissedYield();
60                  }
61                  else
62                  {
63                      mtCOVERAGE_TEST_MARKER();
64                  }
65
66                  --cRxLock;
67              }
68              else
69              {
70                  break;
71              }
72          }
73
74          pxQueue->cRxLock = queueUNLOCKED;
75      }
76      taskEXIT_CRITICAL();
77  }
```

向队列发送消息

```
1   BaseType_t xQueueGenericSend( QueueHandle_t xQueue, const void * const
    pvItemToQueue, TickType_t xTicksToWait, const BaseType_t xCopyPosition )
2   {
3       BaseType_t xEntryTimeSet = pdFALSE, xYieldRequired;
4       TimeOut_t xTimeOut;
5       Queue_t * const pxQueue = ( Queue_t * ) xQueue;
6
7       configASSERT( pxQueue );
8       configASSERT( !( ( pvItemToQueue == NULL ) && ( pxQueue->uxItemSize !=
    ( UBaseType_t ) 0U ) ) );
9       configASSERT( !( ( xCopyPosition == queueOVERWRITE ) && ( pxQueue-
    >uxLength != 1 ) ) );
10      #if ( ( INCLUDE_xTaskGetSchedulerState == 1 ) || ( configUSE_TIMERS ==
    1 ) )
11      {
12          configASSERT( !( ( xTaskGetSchedulerState() ==
    taskSCHEDULER_SUSPENDED ) && ( xTicksToWait != 0 ) ) );
13      }
14      #endif
15
16      for( ;; )
17      {
18          /* 关中断，进入临界区 */
```

```
        taskENTER_CRITICAL();
        {
            /* 查看队列中是否还有剩余空间，如果采用的覆写方式入队则不用在乎队列是否是满的 */
            if( ( pxQueue->uxMessagesWaiting < pxQueue->uxLength ) || ( xCopyPosition == queueOVERWRITE ) )
            {
                traceQUEUE_SEND( pxQueue );
                /* 将数据拷贝到队列中 */
                xYieldRequired = prvCopyDataToQueue( pxQueue, pvItemToQueue, xCopyPosition );


 /*****************************************************/
                /**********************省略队列集代码**********************/

 /*****************************************************/

                #else /* configUSE_QUEUE_SETS */
                {
                    /* 判断是否有任务在等待队列消息而阻塞 */
                    if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToReceive ) ) == pdFALSE )
                    {
                        /* 有的话移除任务，取消阻塞：将TCB中的事件等待列表项和状态等待列表项从当前所在列表中移除，添加任务到就绪列表 */
                        if( xTaskRemoveFromEventList( &( pxQueue->xTasksWaitingToReceive ) ) != pdFALSE )
                        {
                            /* 解除阻塞态的优先级最高，进行一次任务切换 */
                            queueYIELD_IF_USING_PREEMPTION();
                        }
                        else
                        {
                            mtCOVERAGE_TEST_MARKER();
                        }
                    }
                    else if( xYieldRequired != pdFALSE )
                    {
                        queueYIELD_IF_USING_PREEMPTION();
                    }
                    else
                    {
                        mtCOVERAGE_TEST_MARKER();
                    }
                }

                taskEXIT_CRITICAL();
                return pdPASS;
            }
            /* 如果队列满了 */
            else
            {
                if( xTicksToWait == ( TickType_t ) 0 )
```

```
                {
                    taskEXIT_CRITICAL();
                    /* 阻塞时间为0则直接返回 */
                    traceQUEUE_SEND_FAILED( pxQueue );
                    return errQUEUE_FULL;
                }
                else if( xEntryTimeSet == pdFALSE )
                {
                    /* 阻塞时间不为0则初始化超时时间结构体 */
                    vTaskSetTimeOutState( &xTimeOut );
                    xEntryTimeSet = pdTRUE;
                }
                else
                {
                    /* 时间结构体已经初始化过了 */
                    mtCOVERAGE_TEST_MARKER();
                }
            }
        }
        taskEXIT_CRITICAL();

        /* 挂起任务调度器 */
        vTaskSuspendAll();
        /* 队列上锁 */
        prvLockQueue( pxQueue );

        /* 更新时间并检查是否超时 */
        if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) == pdFALSE )
        {
            /* 未超时 */
            if( prvIsQueueFull( pxQueue ) != pdFALSE )
            {
                traceBLOCKING_ON_QUEUE_SEND( pxQueue );
                /* 将当前任务的事件列表项加入到队列的等待发送列表中 */
                /* 将当前任务的状态列表项从就绪列表中移除，将其加入到延时列表中 */
                vTaskPlaceOnEventList( &( pxQueue->xTasksWaitingToSend ),
xTicksToWait );

                /* 解锁队列 */
                prvUnlockQueue( pxQueue );

                /* 唤醒所有，开启OS时钟计数 */
                if( xTaskResumeAll() == pdFALSE )
                {
                    /* 保证执行一次上下文切换 */
                    portYIELD_WITHIN_API();
                }
            }
            else
            {
                /* 如果队列未满，解锁队列，再试一次 */
                prvUnlockQueue( pxQueue );
                ( void ) xTaskResumeAll();
            }
        }
```

```
        else
        {
            /* 超时 */
            prvUnlockQueue( pxQueue );
            ( void ) xTaskResumeAll();

            traceQUEUE_SEND_FAILED( pxQueue );
            return errQUEUE_FULL;
        }
    }
}
```

读取队列数据

```
BaseType_t xQueueGenericReceive( QueueHandle_t xQueue, void * const pvBuffer, TickType_t xTicksToWait, const BaseType_t xJustPeeking )
{
    BaseType_t xEntryTimeSet = pdFALSE;
    TimeOut_t xTimeOut;
    int8_t *pcOriginalReadPosition;
    Queue_t * const pxQueue = ( Queue_t * ) xQueue;

    configASSERT( pxQueue );
    configASSERT( !( ( pvBuffer == NULL ) && ( pxQueue->uxItemSize != ( UBaseType_t ) 0U ) ) );
    #if ( ( INCLUDE_xTaskGetSchedulerState == 1 ) || ( configUSE_TIMERS == 1 ) )
    {
        configASSERT( !( ( xTaskGetSchedulerState() == taskSCHEDULER_SUSPENDED ) && ( xTicksToWait != 0 ) ) );
    }
    #endif

    for( ;; )
    {
        taskENTER_CRITICAL();
        {
            const UBaseType_t uxMessagesWaiting = pxQueue->uxMessagesWaiting;

            /* 判断队列中是否有数据 */
            if( uxMessagesWaiting > ( UBaseType_t ) 0 )     //如果队列中有数据
            {
                /* 读取数据到pvBuffer中，并更新u.pcReadFrom的值 */
                pcOriginalReadPosition = pxQueue->u.pcReadFrom;

                prvCopyDataFromQueue( pxQueue, pvBuffer );

                /* 判断是否删除读取的队列项 */
                if( xJustPeeking == pdFALSE ) //如果要删除读取的队列项
                {
                    traceQUEUE_RECEIVE( pxQueue );

```

```
35                      /* 更新队列中队列项数量，完成数据的删除 */
36                      pxQueue->uxMessagesWaiting = uxMessagesWaiting - 1;
37
38                      #if ( configUSE_MUTEXES == 1 )      //用户定义了互斥信号量
39                      {
40                          if( pxQueue->uxQueueType == queueQUEUE_IS_MUTEX )
//如果此队列是互斥信号量
41                          {
42                              /* 将pxMutexHolder标记为获取到互斥信号量的任务TCB */
43                              /* 将任务TCB中互斥信号量的数量加一    (
pxCurrentTCB->uxMutexesHeld )++; */
44                              pxQueue->pxMutexHolder = ( int8_t * )
pvTaskIncrementMutexHeldCount();
45                          }
46                          else
47                          {
48                              mtCOVERAGE_TEST_MARKER();
49                          }
50                      }
51                      #endif /* configUSE_MUTEXES */
52
53                      /* 如果等待发送列表不为空，有任务因队列满而阻塞，则唤醒一个发送任
务 */
54                      if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToSend
) ) == pdFALSE )
55                      {
56                          if( xTaskRemoveFromEventList( &( pxQueue-
>xTasksWaitingToSend ) ) != pdFALSE )
57                          {
58                              queueYIELD_IF_USING_PREEMPTION();
59                          }
60                          else
61                          {
62                              mtCOVERAGE_TEST_MARKER();
63                          }
64                      }
65                      else
66                      {
67                          mtCOVERAGE_TEST_MARKER();
68                      }
69                  }
70                  else //如果不删除队列项
71                  {
72                      traceQUEUE_PEEK( pxQueue );
73
74                      /* 不删除队列项，恢复u.pcReadFrom */
75                      pxQueue->u.pcReadFrom = pcOriginalReadPosition;
76
77                      /* 队列中还有数据，查看是否有其他任务阻塞在获取队列数据中，有的话
唤醒 */
78                      if( listLIST_IS_EMPTY( &( pxQueue-
>xTasksWaitingToReceive ) ) == pdFALSE )
79                      {
80                          if( xTaskRemoveFromEventList( &( pxQueue-
>xTasksWaitingToReceive ) ) != pdFALSE )
```

```
                        {
                            queueYIELD_IF_USING_PREEMPTION();
                        }
                        else
                        {
                            mtCOVERAGE_TEST_MARKER();
                        }
                    }
                    else
                    {
                        mtCOVERAGE_TEST_MARKER();
                    }
                }

                taskEXIT_CRITICAL();
                return pdPASS;
            }
            else    //队列中无数据
            {
                if( xTicksToWait == ( TickType_t ) 0 )
                {
                    /* 队列为空且等待时间为0，直接返回 */
                    taskEXIT_CRITICAL();
                    traceQUEUE_RECEIVE_FAILED( pxQueue );
                    return errQUEUE_EMPTY;
                }
                else if( xEntryTimeSet == pdFALSE )
                {
                    /* 确保时间结构体的初始化 */
                    vTaskSetTimeOutState( &xTimeOut );
                    xEntryTimeSet = pdTRUE;
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
        }
        taskEXIT_CRITICAL();

        /* 中断和其他任务可以向队列中发送和接收数据，现在关键部分已退出 */

        vTaskSuspendAll();
        prvLockQueue( pxQueue );

        /* 更新时钟，检查是否超时 */
        if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) == pdFALSE )
        {
            /* 未超时 */
            /* 队列仍为空 */
            if( prvIsQueueEmpty( pxQueue ) != pdFALSE )
            {
                traceBLOCKING_ON_QUEUE_RECEIVE( pxQueue );

                #if ( configUSE_MUTEXES == 1 )      /* 是否使用互斥信号量 */
```

```
136                 {
137                     if( pxQueue->uxQueueType == queueQUEUE_IS_MUTEX )
/* 此队列表示互斥信号量 */
138                     {
139                         taskENTER_CRITICAL();
140                         {
141                             //到这说明队列为空，互斥信号量在其他任务中
142                             //处理优先级继承，避免优先级翻转
143                             vTaskPriorityInherit( ( void * ) pxQueue-
>pxMutexHolder );
144                         }
145                         taskEXIT_CRITICAL();
146                     }
147                     else
148                     {
149                         mtCOVERAGE_TEST_MARKER();
150                     }
151                 }
152                 #endif
153 
154                 /* 将任务的时间列表项插入到队列的等待列表 */
155                 /* 将任务的状态列表项插入到延时列表 */
156                 vTaskPlaceOnEventList( &( pxQueue->xTasksWaitingToReceive
), xTicksToWait );
157                 prvUnlockQueue( pxQueue );
158                 if( xTaskResumeAll() == pdFALSE )
159                 {
160                     portYIELD_WITHIN_API();
161                 }
162                 else
163                 {
164                     mtCOVERAGE_TEST_MARKER();
165                 }
166             }
167             else
168             {
169                 /* 队列不为空，有其他任务或中断往队列中写了数据 */
170                 /* 解锁队列，再试一次 */
171                 prvUnlockQueue( pxQueue );
172                 ( void ) xTaskResumeAll();
173             }
174         }
175         else
176         {
177             prvUnlockQueue( pxQueue );
178             ( void ) xTaskResumeAll();
179 
180             if( prvIsQueueEmpty( pxQueue ) != pdFALSE )
181             {
182                 traceQUEUE_RECEIVE_FAILED( pxQueue );
183                 return errQUEUE_EMPTY;
184             }
185             else
186             {
187                 mtCOVERAGE_TEST_MARKER();
```

```
            }
        }
    }
}
```

取消事件列表任务阻塞 xTaskRemoveFromEventList

```
BaseType_t xTaskRemoveFromEventList( const List_t * const pxEventList )
{
    TCB_t *pxUnblockedTCB;
    BaseType_t xReturn;

    //获取列表中第一个列表项的所属任务TCB
    pxUnblockedTCB = ( TCB_t * ) listGET_OWNER_OF_HEAD_ENTRY( pxEventList );
    configASSERT( pxUnblockedTCB );
    /* 将要解除阻塞的任务事件列表项移出 */
    ( void ) uxListRemove( &( pxUnblockedTCB->xEventListItem ) );

    if( uxSchedulerSuspended == ( UBaseType_t ) pdFALSE )   //任务调度器未被挂起
    {
        /* 将任务TCB的状态列表项从延时列表中移出，将其添加到就绪列表中 */
        ( void ) uxListRemove( &( pxUnblockedTCB->xStateListItem ) );
        prvAddTaskToReadyList( pxUnblockedTCB );
    }
    else
    {
        /* 将任务事件列表项添加到等待唤醒列表中 */
        vListInsertEnd( &( xPendingReadyList ), &( pxUnblockedTCB->xEventListItem ) );
    }

    if( pxUnblockedTCB->uxPriority > pxCurrentTCB->uxPriority )
    {
        /* 如果解除阻塞的优先级较现在运行的任务优先级高 */
        xReturn = pdTRUE;

        /* 标志一下，之后调用一次任务切换 */
        xYieldPending = pdTRUE;
    }
    else
    {
        xReturn = pdFALSE;
    }

    #if( configUSE_TICKLESS_IDLE != 0 )
    {
        /* If a task is blocked on a kernel object then xNextTaskUnblockTime
        might be set to the blocked task's time out time.  If the task is
        unblocked for a reason other than a timeout xNextTaskUnblockTime is
        normally left unchanged, because it is automatically reset to a new
        value when the tick count equals xNextTaskUnblockTime.  However if
        tickless idling is used it might be more important to enter sleep
mode
```

```
        at the earliest possible time - so reset xNextTaskUnblockTime here to
        ensure it is updated at the earliest possible time. */
        prvResetNextTaskUnblockTime();
    }
    #endif

    return xReturn;
}
```

信号量

信号量的本质为队列，各种信号量的底层均是用队列实现的

二值信号量

二值信号量常用于互斥访问或同步，二值信号量和互斥信号量非常像，但是互斥信号量有优先级继承机制，二值信号量没有优先级继承机制，所以二值信号量可能会导致优先级翻转，因为这一问题，二值信号量更适合用于同步，而互斥信号量适合用于简单的互斥访问。

二值信号量底层就是一个长度为1的队列，获取释放等操作和队列的操作相同，只是进行了封装

```
#define xSemaphoreCreateBinary() xQueueGenericCreate( ( UBaseType_t ) 1, semSEMAPHORE_QUEUE_ITEM_LENGTH, queueQUEUE_TYPE_BINARY_SEMAPHORE )
```

计数信号量

计数信号量本质是个长度为uxMaxCount，队列项大小为0的队列；对它的操作和普通的发送消息的队列相似只是做了封装

```
#define xSemaphoreCreateCounting( uxMaxCount, uxInitialCount ) xQueueCreateCountingSemaphore( ( uxMaxCount ), ( uxInitialCount ) )

QueueHandle_t xQueueCreateCountingSemaphore( const UBaseType_t uxMaxCount, const UBaseType_t uxInitialCount )
{
    QueueHandle_t xHandle;

    configASSERT( uxMaxCount != 0 );
    configASSERT( uxInitialCount <= uxMaxCount );

    /* 创建一个队列 queueSEMAPHORE_QUEUE_ITEM_LENGTH==0 */
    xHandle = xQueueGenericCreate( uxMaxCount, queueSEMAPHORE_QUEUE_ITEM_LENGTH, queueQUEUE_TYPE_COUNTING_SEMAPHORE );

    if( xHandle != NULL )
    {
    ( ( Queue_t * ) xHandle )->uxMessagesWaiting = uxInitialCount;

    traceCREATE_COUNTING_SEMAPHORE();
    }
    else
    {
```

```
    traceCREATE_COUNTING_SEMAPHORE_FAILED();
    }

    return xHandle;
}
```

互斥信号量

互斥信号量和二值信号量相似，只是它有优先级继承的特性。当一个互斥信号量被一个低优先级的任务使用时，而此时有个高优先级的任务也在尝试获取这个互斥信号量，这个高优先级的任务将被阻塞。不过，这个高优先级任务会将低优先级任务的优先级提高到和自己一样，这就是优先级继承。优先级继承尽可能得降低了高优先级任务的阻塞时间，并且降低优先级翻转的影响。

互斥信号量本质是一个长度为1，队列项大小为0，队列类型(ucQueueType)为互斥量队列（queueQUEUE_TYPE_MUTEX）的队列。

相比于普通的队列，对互斥信号量的操作多了：

- 更改TCB中记录获取到的互斥信号量数量
- 更新队列（即互斥信号量）的所属任务句柄
- 处理优先级继承

互斥信号量的创建

```
#define xSemaphoreCreateMutex() xQueueCreateMutex( queueQUEUE_TYPE_MUTEX )

QueueHandle_t xQueueCreateMutex( const uint8_t ucQueueType )
{
    Queue_t *pxNewQueue;
    const UBaseType_t uxMutexLength = ( UBaseType_t ) 1, uxMutexSize = ( UBaseType_t ) 0;

    pxNewQueue = ( Queue_t * ) xQueueGenericCreate( uxMutexLength, uxMutexSize, ucQueueType );
    prvInitialiseMutex( pxNewQueue );

    return pxNewQueue;
}
```

优先级继承

释放互斥信号量中的优先级继承 xTaskPriorityDisinherit

和申请互斥信号量时的操作相反

申请互斥信号量中的优先级继承 vTaskPriorityInherit

```
void vTaskPriorityInherit( TaskHandle_t const pxMutexHolder )
{
    TCB_t * const pxTCB = ( TCB_t * ) pxMutexHolder;

    /* 队列锁定时，中断释放了互斥锁；则互斥锁持有者为NULL */
    if( pxMutexHolder != NULL )
    {
```

```
        /* 如果互斥锁持有者优先级低于当前尝试获取互斥锁的任务优先级，则进行优先级翻转 */
        if( pxTCB->uxPriority < pxCurrentTCB->uxPriority )
        {
            /* 判断互斥锁的持有者的事件列表项 */
            if( ( listGET_LIST_ITEM_VALUE( &( pxTCB->xEventListItem ) ) &
taskEVENT_LIST_ITEM_VALUE_IN_USE ) == 0UL )
            {
                /* 没有在等待事件，将持有者事件列表项的优先级赋为当前尝试获取互斥锁任务
的优先级 */
                listSET_LIST_ITEM_VALUE( &( pxTCB->xEventListItem ), (
TickType_t ) configMAX_PRIORITIES - ( TickType_t ) pxCurrentTCB->uxPriority
);
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }

            /* 判断持有者是否处于就绪列表中 */
            if( listIS_CONTAINED_WITHIN( &( pxReadyTasksLists[ pxTCB-
>uxPriority ] ), &( pxTCB->xStateListItem ) ) != pdFALSE )
            {
                /* 在就绪列表中的话，移出任务，更改其优先级，更新就绪列表的最大优先级，
移动任务到新的就绪列表中 */
                if( uxListRemove( &( pxTCB->xStateListItem ) ) == (
UBaseType_t ) 0 )
                {
                    taskRESET_READY_PRIORITY( pxTCB->uxPriority );
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }

                pxTCB->uxPriority = pxCurrentTCB->uxPriority;
                prvAddTaskToReadyList( pxTCB );
            }
            else
            {
                /* 未在就绪列表的话，只更改优先级 */
                pxTCB->uxPriority = pxCurrentTCB->uxPriority;
            }

            traceTASK_PRIORITY_INHERIT( pxTCB, pxCurrentTCB->uxPriority );
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
```

递归互斥信号量

它是一种特殊的互斥信号量，一个任务可以递归得获取它，可以获取的次数不限，但是要求任务释放此信号量的次数和获取它的次数相同

递归互斥量的创建

递归互斥量创建过程和互斥量相同

```
#define xSemaphoreCreateRecursiveMutex() xQueueCreateMutex( queueQUEUE_TYPE_RECURSIVE_MUTEX )
```

递归互斥量的释放

```
#define xSemaphoreGiveRecursive( xMutex )   xQueueGiveMutexRecursive( ( xMutex ) )

BaseType_t xQueueGiveMutexRecursive( QueueHandle_t xMutex )
{
    BaseType_t xReturn;
    Queue_t * const pxMutex = ( Queue_t * ) xMutex;

    configASSERT( pxMutex );

    /* 判断，确保当前释放任务和递归互斥锁的持有者是一个任务 */
    if( pxMutex->pxMutexHolder == ( void * ) xTaskGetCurrentTaskHandle() )
    {
        traceGIVE_MUTEX_RECURSIVE( pxMutex );

        /* u.uxRecursiveCallCount是用来记录递归互斥锁的获取次数的 */
        ( pxMutex->u.uxRecursiveCallCount )--;

        /* 变量为0说明最后一次释放 */
        if( pxMutex->u.uxRecursiveCallCount == ( UBaseType_t ) 0 )
        {
            /* 往队列发送空数据，唤醒其他等待任务 */
            ( void ) xQueueGenericSend( pxMutex, NULL, queueMUTEX_GIVE_BLOCK_TIME, queueSEND_TO_BACK );
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }

        xReturn = pdPASS;
    }
    else
    {
        /* 如果当前释放锁的任务和锁的持有者不一致，报错 */
        xReturn = pdFAIL;

        traceGIVE_MUTEX_RECURSIVE_FAILED( pxMutex );
    }

```

```
    return xReturn;
}
```

递归互斥量的获取

```
#define xSemaphoreTakeRecursive( xMutex, xBlockTime ) xQueueTakeMutexRecursive( ( xMutex ), ( xBlockTime ) )

BaseType_t xQueueTakeMutexRecursive( QueueHandle_t xMutex, TickType_t xTicksToWait )
{
    BaseType_t xReturn;
    Queue_t * const pxMutex = ( Queue_t * ) xMutex;

    configASSERT( pxMutex );

    traceTAKE_MUTEX_RECURSIVE( pxMutex );

    /* 判断锁的持有者和当前获取锁的任务是否为同一任务 */
    if( pxMutex->pxMutexHolder == ( void * ) xTaskGetCurrentTaskHandle() )
    {
        /* 是同一任务，记录锁获取次数的变量加一 */
        ( pxMutex->u.uxRecursiveCallCount )++;
        xReturn = pdPASS;
    }
    else
    {
        /* 尝试获取锁，如果锁未被其他任务持有且为第一次获取锁则获取成功；否则函数返回 pdFAIL */
        xReturn = xQueueGenericReceive( pxMutex, NULL, xTicksToWait, pdFALSE );

        if( xReturn != pdFAIL )
        {
            /* 获取成功，计数加一 */
            ( pxMutex->u.uxRecursiveCallCount )++;
        }
        else
        {
            traceTAKE_MUTEX_RECURSIVE_FAILED( pxMutex );
        }
    }

    return xReturn;
}
```

软件定时器

软件定时器允许设置一段时间，当设置的时间到达后，执行指定的回调函数，因为回调函数在定时器服务任务中执行，所以不能使用会使任务阻塞的API函数

软件定时器不属于FreeRTOS内核的功能，它是通过定时器服务任务来提供，软件定时器运行在prvTimerTask中

FreeRTOS提供API，应用程序通过向定时器命令队列中写命令来操作定时器

在定时器服务任务中会从定时器队列中获取命令，根据命令对定时器时间列表进行更改，执行满足条件的定时器的服务函数

事件组

事件是用位图来表示存储的

事件位可以用来表明某个事件是否发生，事件的好处是可以一对多，多对一，一对一

一个任务可以同时等待好几个事件，同样也可以好几个任务等待同一个事件

事件组结构体

```
typedef struct xEventGroupDefinition
{
    EventBits_t uxEventBits;
    List_t xTasksWaitingForBits;        /* 等待某个位变为1的任务列表 */

    #if( configUSE_TRACE_FACILITY == 1 )
        UBaseType_t uxEventGroupNumber;
    #endif

    #if( ( configSUPPORT_STATIC_ALLOCATION == 1 ) && ( configSUPPORT_DYNAMIC_ALLOCATION == 1 ) )
        uint8_t ucStaticallyAllocated; /* 静态方法标志位，手动释放内存 */
    #endif
} EventGroup_t;
```

创建事件标志组

```
EventGroupHandle_t xEventGroupCreate( void )
{
    EventGroup_t *pxEventBits;

    /* 申请内存 */
    pxEventBits = ( EventGroup_t * ) pvPortMalloc( sizeof( EventGroup_t ) );

    if( pxEventBits != NULL )
    {
        pxEventBits->uxEventBits = 0;
        vListInitialise( &( pxEventBits->xTasksWaitingForBits ) );

        #if( configSUPPORT_STATIC_ALLOCATION == 1 )
        {
            pxEventBits->ucStaticallyAllocated = pdFALSE;
        }
        #endif /* configSUPPORT_STATIC_ALLOCATION */

        traceEVENT_GROUP_CREATE( pxEventBits );
    }
    else
    {
```

```
        traceEVENT_GROUP_CREATE_FAILED();
    }

    return ( EventGroupHandle_t ) pxEventBits;
}
```

申请内存，事件组的标志位和等待事件组列表初始化

清除事件标志位&获取事件标志组的值

```
EventBits_t xEventGroupClearBits( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToClear )
{
    EventGroup_t *pxEventBits = ( EventGroup_t * ) xEventGroup;
    EventBits_t uxReturn;

    configASSERT( xEventGroup );
    configASSERT( ( uxBitsToClear & eventEVENT_BITS_CONTROL_BYTES ) == 0 );

    taskENTER_CRITICAL();
    {
        traceEVENT_GROUP_CLEAR_BITS( xEventGroup, uxBitsToClear );

        uxReturn = pxEventBits->uxEventBits;

        /* 清楚位 */
        pxEventBits->uxEventBits &= ~uxBitsToClear;
    }
    taskEXIT_CRITICAL();

    return uxReturn;
}

#define xEventGroupGetBits( xEventGroup ) xEventGroupClearBits( xEventGroup, 0 )
```

等待事件组

```
EventBits_t xEventGroupWaitBits( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToWaitFor, const BaseType_t xClearOnExit, const BaseType_t xWaitForAllBits, TickType_t xTicksToWait )
{
    /*  xEventGroup：事件组结构体
        uxBitsToWaitFor：等待事件位
        xClearOnExit：是否清除事件位，pdTRUE则清除，pdFALSE不清除
        xWaitForAllBits：是否等待所有的位，pdTRUE为事件均发生，pdFALSE表示至少一个事件发生
        xTicksToWait：等待超时时间      */

    EventGroup_t *pxEventBits = ( EventGroup_t * ) xEventGroup;
    EventBits_t uxReturn, uxControlBits = 0;
    BaseType_t xWaitConditionMet, xAlreadyYielded;
    BaseType_t xTimeoutOccurred = pdFALSE;

```

```
    configASSERT( xEventGroup );
    configASSERT( ( uxBitsToWaitFor & eventEVENT_BITS_CONTROL_BYTES ) == 0 );
    configASSERT( uxBitsToWaitFor != 0 );
    #if ( ( INCLUDE_xTaskGetSchedulerState == 1 ) || ( configUSE_TIMERS == 1 ) )
    {
        configASSERT( !( ( xTaskGetSchedulerState() == taskSCHEDULER_SUSPENDED ) && ( xTicksToWait != 0 ) ) );
    }
    #endif

    vTaskSuspendAll();  //关任务调度器
    {
        const EventBits_t uxCurrentEventBits = pxEventBits->uxEventBits;

        /* 查看事件是否已满足 */
        xWaitConditionMet = prvTestWaitCondition( uxCurrentEventBits, uxBitsToWaitFor, xWaitForAllBits );

        if( xWaitConditionMet != pdFALSE )
        {
            /* 等待事件已经触发，无需等待 */
            uxReturn = uxCurrentEventBits;
            xTicksToWait = ( TickType_t ) 0;

            /* 退出时是否清除位处理 */
            if( xClearOnExit != pdFALSE )
            {
                pxEventBits->uxEventBits &= ~uxBitsToWaitFor;
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        else if( xTicksToWait == ( TickType_t ) 0 )
        {
            /* 未满足，但是等待阻塞时间为0 */
            uxReturn = uxCurrentEventBits;
        }
        else
        {
            /* 任务要阻塞等待事件触发 */

            /* 利用uxControlBits的位存储事件位操作和事件的触发要求 */
            if( xClearOnExit != pdFALSE )
            {
                uxControlBits |= eventCLEAR_EVENTS_ON_EXIT_BIT;
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }

```

```
65              if( xWaitForAllBits != pdFALSE )
66              {
67                  uxControlBits |= eventWAIT_FOR_ALL_BITS;
68              }
69              else
70              {
71                  mtCOVERAGE_TEST_MARKER();
72              }
73
74              /*  将TCB中的事件列表项挂在事件组的等待列表下；事件列表项的value值设为
uxControlBits
75                  value值中，低24位（0-23）表示等待的事件位，第24位表示退出是否清除事
件位，第26位表示是否等待所有事件位，第27位表示等待事件
76                  将状态列表项移动到延时等待列表中 */
77              vTaskPlaceOnUnorderedEventList( &( pxEventBits->xTasksWaitingForBits ), ( uxBitsToWaitFor | uxControlBits ), xTicksToWait );
78              uxReturn = 0;
79
80              traceEVENT_GROUP_WAIT_BITS_BLOCK( xEventGroup, uxBitsToWaitFor );
81          }
82      }
83      /* 开启任务调度器 */
84      xAlreadyYielded = xTaskResumeAll();
85
86      if( xTicksToWait != ( TickType_t ) 0 )
87      {
88          /* 事件不满足，且有等待时间，任务需要阻塞 */
89          if( xAlreadyYielded == pdFALSE )
90          {
91              /* 确保执行一次任务切换，从当前任务切走 */
92              portYIELD_WITHIN_API();
93          }
94          else
95          {
96              mtCOVERAGE_TEST_MARKER();
97          }
98
99          /* 程序运行到这表示任务切换回来了，即超时了或者等到了事件位 */
100
101         /* 重置任务的事件列表项的value值，置为最大优先级-任务优先级 */
102         uxReturn = uxTaskResetEventItemValue();
103
104         /* 判断是因为超时还是等到了事件位 */
105         if( ( uxReturn & eventUNBLOCKED_DUE_TO_BIT_SET ) == ( EventBits_t ) 0 )
106         {
107             taskENTER_CRITICAL();
108             {
109                 /* 超时了 */
110                 uxReturn = pxEventBits->uxEventBits;
111
112                 /* 再判断一下是否满足事件位，事件位可能再任务恢复之间更新 */
```

```
                if( prvTestWaitCondition( uxReturn, uxBitsToWaitFor, xWaitForAllBits ) != pdFALSE )
                {
                    if( xClearOnExit != pdFALSE )
                    {
                        pxEventBits->uxEventBits &= ~uxBitsToWaitFor;
                    }
                    else
                    {
                        mtCOVERAGE_TEST_MARKER();
                    }
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
            taskEXIT_CRITICAL();

            xTimeoutOccurred = pdFALSE;
        }
        else
        {
            /* 任务唤醒已唤醒，因为满足了事件位 */
        }

        /* 任务被阻塞，因此可能设置了控制位 */
        uxReturn &= ~eventEVENT_BITS_CONTROL_BYTES;
    }
    traceEVENT_GROUP_WAIT_BITS_END( xEventGroup, uxBitsToWaitFor, xTimeoutOccurred );

    return uxReturn;
}
```

设置事件组标志位

```
EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet )
{
    ListItem_t *pxListItem, *pxNext;
    ListItem_t const *pxListEnd;
    List_t *pxList;
    EventBits_t uxBitsToClear = 0, uxBitsWaitedFor, uxControlBits;
    EventGroup_t *pxEventBits = ( EventGroup_t * ) xEventGroup;
    BaseType_t xMatchFound = pdFALSE;

    configASSERT( xEventGroup );
    configASSERT( ( uxBitsToSet & eventEVENT_BITS_CONTROL_BYTES ) == 0 );

    pxList = &( pxEventBits->xTasksWaitingForBits );
    pxListEnd = listGET_END_MARKER( pxList );

    vTaskSuspendAll();  //挂起调度器
```

```
    {
        traceEVENT_GROUP_SET_BITS( xEventGroup, uxBitsToSet );

        pxListItem = listGET_HEAD_ENTRY( pxList );

        /* 设置事件标志位，将存储事件的变量置位 */
        pxEventBits->uxEventBits |= uxBitsToSet;

        /* 查看是否有因为新的事件位置一而可以取消阻塞的任务 */
        while( pxListItem != pxListEnd )
        {
            pxNext = listGET_NEXT( pxListItem );
            uxBitsWaitedFor = listGET_LIST_ITEM_VALUE( pxListItem );
            xMatchFound = pdFALSE;

            /* 将value值分割，分为等待事件位部分和事件位设置部分 */
            uxControlBits = uxBitsWaitedFor & eventEVENT_BITS_CONTROL_BYTES;
            uxBitsWaitedFor &= ~eventEVENT_BITS_CONTROL_BYTES;

            if( ( uxControlBits & eventWAIT_FOR_ALL_BITS ) == ( EventBits_t ) 0 )
            {
                /* 事件设置：事件是否满足任意一个 */
                if( ( uxBitsWaitedFor & pxEventBits->uxEventBits ) != ( EventBits_t ) 0 )
                {
                    xMatchFound = pdTRUE;
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
            else if( ( uxBitsWaitedFor & pxEventBits->uxEventBits ) == uxBitsWaitedFor )
            {
                /* 事件设置：要求所有事件位均满足 */
                xMatchFound = pdTRUE;
            }
            else
            {
                /* 不满足事件位要求 */
            }

            if( xMatchFound != pdFALSE )
            {
                /* 事件位满足要求 */

                if( ( uxControlBits & eventCLEAR_EVENTS_ON_EXIT_BIT ) != ( EventBits_t ) 0 )    //判断是否需要清除事件位
                {
                    /* 如果需要清除，将需要清除的位记录下来 */
                    uxBitsToClear |= uxBitsWaitedFor;
                }
                else
```

```
                {
                    mtCOVERAGE_TEST_MARKER();
                }

                /* 存储满足要求在任务事件列表项value值的第25位，同时将任务移到就绪列表中，根据优先级判断是否需要任务切换 */
                ( void ) xTaskRemoveFromUnorderedEventList( pxListItem, pxEventBits->uxEventBits | eventUNBLOCKED_DUE_TO_BIT_SET );
            }

            /* 检查下一个等待项 */
            pxListItem = pxNext;
        }

        /* 退出时清除位，如果不需要清除或者无满足事件位组，uxBitsToClear为0 */
        pxEventBits->uxEventBits &= ~uxBitsToClear;
    }
    ( void ) xTaskResumeAll();  //开启调度器

    return pxEventBits->uxEventBits;
}
```

任务通知

任务通知是一对一或者多对一的，一个任务可以接收任何任务发的通知，也可以向任何任务发送通知；任务通知是个事件

注：任务通知不能为0

发送任务通知：将TCB中的ulNotifiedValue元素按指定方式修改，同时修改TCB中任务通知小黄太，然后将阻塞的任务恢复到就绪列表中

接收任务通知：根据TCB中任务通知的状态和等待时间，将任务移到延时等待列表；如果切换回来，判断是否因为超时，超时返回0，接收到了通知返回接收到的通知值

空闲任务

当FreeRTOS的调度器启动以后就会自动的创建一个空闲任务，这样就可以确保至少有一任务可以运行。此空闲任务使用最低优先级，如果应用中有其他高优先级任务处于就绪态的话这个空闲任务就不会跟高优先级的任务抢占资源。如果某个任务要调用函数vTaskDelete()删除自身，那么这个任务的任务控制块TCB和堆栈等这些由FreeRTOS 系统自动分配的内存会在空闲任务中释放。

空闲任务函数会释放任务的TCB和内存；同时保证有其他任务时让出CPU（合作式调度）；定义了时间片轮转的话轮转相同优先级的任务；定义了空闲任务钩子函数的话执行钩子函数；定义了低功耗的话，执行相应的处理

注：因为要保证FreeRTOS至少有一个任务在运行，所有钩子函数中不能有会导致阻塞的函数

内存管理

FreeRTOS内核在动态创建任务队列的时候会动态的申请RAM，类似于malloc和free实现动态内存管理，但是它们不适用于小型嵌入式系统，因此FreeRTOS使用自己内核的动态内存管理函数

heap_1内存分配方法

heap_1内存分配方式为移动堆栈空闲起始指针，不允许释放内存。heap_1内存管理办法适合于某些小型的应用，这些应用创建一些任务之后便不会删除。

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

    /* portBYTE_ALIGNMENT字节对齐定义，一般为8 */
    /* 确字节保对齐；宏定义为8的话，令字节大小向上取8的倍数 */
    #if( portBYTE_ALIGNMENT != 1 )
    {
        if( xWantedSize & portBYTE_ALIGNMENT_MASK )
        {
            xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize & portBYTE_ALIGNMENT_MASK ) );
        }
    }
    #endif

    vTaskSuspendAll();
    {
        if( pucAlignedHeap == NULL )
        {
            /* 确保堆栈开始指针为8的倍数 */
            pucAlignedHeap = ( uint8_t * ) ( ( ( portPOINTER_SIZE_TYPE ) &ucHeap[ portBYTE_ALIGNMENT ] ) & ( ~( ( portPOINTER_SIZE_TYPE ) portBYTE_ALIGNMENT_MASK ) ) );
        }

        /* 检查是否有足够的剩余RAM来分配内存 */
        if( ( ( xNextFreeByte + xWantedSize ) < configADJUSTED_HEAP_SIZE ) &&
            ( ( xNextFreeByte + xWantedSize ) > xNextFreeByte ) )/* Check for overflow. */
        {
            /* 分配内存，移动堆栈剩余空间的起始指针 */
            pvReturn = pucAlignedHeap + xNextFreeByte;
            xNextFreeByte += xWantedSize;
        }

        traceMALLOC( pvReturn, xWantedSize );
    }
    ( void ) xTaskResumeAll();

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

```

```
    return pvReturn;
}
```

```
void vPortFree( void *pv )
{
    /* 释放内存并不能回收RAM */
    ( void ) pv;

    configASSERT( pv == NULL );
}
```

heap_2内存分配方法

定义了一个空闲内存块链表，链表按空闲内存块大小排序；申请内存时遍历此链表，找到合适大小的空闲内存块，将此内存块的地址向后偏移16位，用来存储此内存块的大小信息；在释放内存时，先将指针向前调整，之后根据内存块大小插入到空闲块链表中

heap_2不会把释放的内存合并，随着不断申请和释放内存，可能会造成内存碎片；当每次申请和释放的内存大小一样时，不会造成内存碎片

```
typedef struct A_BLOCK_LINK //内存块结构体
{
    struct A_BLOCK_LINK *pxNextFreeBlock;    /* 指向下一个空闲的内存块 */
    size_t xBlockSize;                       /* 当前空闲内存块大小 */
} BlockLink_t;
```

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

    vTaskSuspendAll();
    {
        /* 如果时第一次申请内存，初始化空闲内存链表 */
        if( xHeapHasBeenInitialised == pdFALSE )
        {
            prvHeapInit();
            xHeapHasBeenInitialised = pdTRUE;
        }

        /* 判断申请的内存大小 */
        if( xWantedSize > 0 )
        {
            /* 重置申请内存大小，多申请一些内存，在返回的内存起始地址前面存储内存块结构体
*/
            xWantedSize += heapSTRUCT_SIZE;

            /* 保证字节对齐，申请的大小取8的倍数 */
            if( ( xWantedSize & portBYTE_ALIGNMENT_MASK ) != 0 )
            {
                xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize &
portBYTE_ALIGNMENT_MASK ) );
```

```
            }
        }

        if( ( xWantedSize > 0 ) && ( xWantedSize < configADJUSTED_HEAP_SIZE ) )
        {
            /* 获取大于申请内存大小空闲内存块，满足条件的内存块指针为pxBlock，链表前一个节点为pxPreviousBlock */
            pxPreviousBlock = &xStart;
            pxBlock = xStart.pxNextFreeBlock;
            while( ( pxBlock->xBlockSize < xWantedSize ) && ( pxBlock->pxNextFreeBlock != NULL ) )
            {
                pxPreviousBlock = pxBlock;
                pxBlock = pxBlock->pxNextFreeBlock;
            }

            /* 如果是xEnd，表示没有满足的空闲内存块 */
            if( pxBlock != &xEnd )
            {
                /* 返回申请的内存块指针：满足要求的内存块起始指针偏移16位 */
                pvReturn = ( void * ) ( ( ( uint8_t * ) pxPreviousBlock->pxNextFreeBlock ) + heapSTRUCT_SIZE );

                /* 清除满足条件的节点，表示被申请了 */
                pxPreviousBlock->pxNextFreeBlock = pxBlock->pxNextFreeBlock;

                /* 如果内存块大于申请的内存大小，可以将其分成两部分 */
                if( ( pxBlock->xBlockSize - xWantedSize ) > heapMINIMUM_BLOCK_SIZE )
                {
                    /* 新的空闲块起始地址为满足条件的内存块地址加申请内存大小 */
                    pxNewBlockLink = ( void * ) ( ( ( uint8_t * ) pxBlock ) + xWantedSize );

                    /* 重新计算两部分内存块的内存大小 */
                    pxNewBlockLink->xBlockSize = pxBlock->xBlockSize - xWantedSize;
                    pxBlock->xBlockSize = xWantedSize;

                    /* 插入新块到空闲内存链表中,按内存块大小排序 */
                    prvInsertBlockIntoFreeList( ( pxNewBlockLink ) );
                }

                xFreeBytesRemaining -= pxBlock->xBlockSize;
            }
        }

        traceMALLOC( pvReturn, xWantedSize );
    }
    ( void ) xTaskResumeAll();

    #if( configUSE_MALLOC_FAILED_HOOK == 1 )
    {
        if( pvReturn == NULL )
```

```
        {
            extern void vApplicationMallocFailedHook( void );
            vApplicationMallocFailedHook();
        }
    }
    #endif

    return pvReturn;
}
```

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

    if( pv != NULL )
    {
        /* 内存块结构体信息存储在申请内存前面，地址向前偏移获取内存块结构体指针，结构体中有内存块的大小信息 */
        puc -= heapSTRUCT_SIZE;

        /* 类型转换，转成内存块结构体 */
        pxLink = ( void * ) puc;

        vTaskSuspendAll();
        {
            /* 将新块加入到空闲块链表中 */
            prvInsertBlockIntoFreeList( ( ( BlockLink_t * ) pxLink ) );
            xFreeBytesRemaining += pxLink->xBlockSize;
            traceFREE( pv, pxLink->xBlockSize );
        }
        ( void ) xTaskResumeAll();
    }
}
```

heap_3内存分配方法

heap_3内存分配方法就是将malloc和free进行了封装

malloc对比较小的内存申请是直接移动堆栈指针，对大的内存申请使用的是mmap建立文件的内存映射

free函数调用后将内存标记为释放，然后根据情况合并

heap_4内存分配方法

和heap_2分配方式相似，都是定义了一个空闲内存块链表，但是heap_4的是按地址大小来排序的，而且在插入节点时会判断相邻内存块是否可以合并，可以则合并内存块；同时heap_4用位图表示内存块是否被释放，用最高位来表示

```
static void prvInsertBlockIntoFreeList( BlockLink_t *pxBlockToInsert )
{
    BlockLink_t *pxIterator;
    uint8_t *puc;

```

```
    /* 找到要插入的节点位置 */
    for( pxIterator = &xStart; pxIterator->pxNextFreeBlock < pxBlockToInsert; pxIterator = pxIterator->pxNextFreeBlock )
    {
    }

    /* 判断内存块是否可以合并，前一个内存块的末地址和要插入的内存块首地址相等则可以插入 */
    puc = ( uint8_t * ) pxIterator;
    if( ( puc + pxIterator->xBlockSize ) == ( uint8_t * ) pxBlockToInsert )
    {
        /* 合并，直接更改内存块大小即可 */
        pxIterator->xBlockSize += pxBlockToInsert->xBlockSize;
        pxBlockToInsert = pxIterator;
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    /* 判断内存块和下一个内存块是否可以合并 */
    puc = ( uint8_t * ) pxBlockToInsert;
    if( ( puc + pxBlockToInsert->xBlockSize ) == ( uint8_t * ) pxIterator->pxNextFreeBlock )
    {
        /* 可以合并，合并内存块，直接更改指针和内存块大小 */
        if( pxIterator->pxNextFreeBlock != pxEnd )
        {
            pxBlockToInsert->xBlockSize += pxIterator->pxNextFreeBlock->xBlockSize;
            pxBlockToInsert->pxNextFreeBlock = pxIterator->pxNextFreeBlock->pxNextFreeBlock;
        }
        else
        {
            pxBlockToInsert->pxNextFreeBlock = pxEnd;
        }
    }
    else
    {
        /* 不能合并，插入到相应节点 */
        pxBlockToInsert->pxNextFreeBlock = pxIterator->pxNextFreeBlock;
    }

    /* 完成节点的插入 */
    if( pxIterator != pxBlockToInsert )
    {
        pxIterator->pxNextFreeBlock = pxBlockToInsert;
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
```

heap_5内存分配方法

heap_5和heap_4算法相同，实现也基本相同，只是heap_5允许内存跨越多个不连续的内存段；比如内部RAM和外接RAM一起，但是heap_4只能二选一