特性

- 支持抢占式,协作式,和时间片调度
- 通常占用4k~9k的固件
- 支持实时任务和协程
- 任务与任务,任务与中断之间可以使用任务通知,消息队列,二值信号量,数值型信号量,递归互斥型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)目的是为了给外设ISR提供保障

1.freertos系统配置

使用的时候需要根据需求来配置FreeRTOS,不同架构的MCU在使用的时候配置也不同。

`INCLUDE_`宏

使用`INCLUDE_`开头的宏用来表示使能或者失能FreeRTOS相应的API函数,0表示不能使用函数,1表示使能对应的函数,即条件编译

许多协议栈,`RTOS`系统和`GUI`库都是使用条件编译的方法来完成配置和裁剪的

```
INCLUDE_xSemaphoreGetMutexHolder  ->
    使能函数 xQueueGetMutexHolder()
INCLUDE_xTaskAbortDelay ->
    使能函数xTaskAbortDelay()
INCLUDE_xTaskDelay  ->
    使能函数 vTaskDelay()
INCLUDE_vTaskDelayUntil ->
    使能函数 vTaskDelayUntil()
INCLUDE_vTaskDelete ->
    使能函数 vTaskDelete()
INCLUDE_xTaskGetCurrentTaskHandle ->
    使能函数 xTaskGetCurrentTaskHandle()
INCLUDE_xTaskGetHandle  ->
    使能函数 xTaskGetHandle()
INCLUDE_xTaskGetIdleTaskHandle  ->
    使能函数    xTaskGetIdleTaskHandle()
INCLUDE_xTaskGetSchedulerState  ->
    使能函数 xTaskGetSchedulerState()
INCLUDE_uxTaskGetStackHighWaterMark ->
    使能函数 uxTaskGetStackHighWaterMart()
INCLUDE_uxTaskPriorityGet   ->
    使能函数 uxTaskPriorityGet()
INCLUDE_vTaskPrioritySet        ->
    使能函数 vTaskPrioritySet()
INCLUDE_xTaskResumeFromISR  ->
    使能函数 xTaskResumeFromISR()
```

```
INCLUDE_eTaskGetState        ->
    使能函数 eTaskGetState()
INCLUDE_vTaskSuspend             ->
    使能函数vTaskSuspend(), vTaskResume(),
prvTaskIsTaskSuspended(),xTaskResumeFromISR()
INCLUDE_xTimerPendFunctionCall和configUSE_TIMERS ->
    使能函数 xTimerPendFunctionCall() 和xTimerPendFunctionCallFromISR()
```

config宏

config 宏和 INCLUDE_ 宏一样,都是用来完成配置裁剪的。

- configAPPLICATION_ALLOCATED_HEAP:

  默认情况下 FreeRTOS 的堆内存时由编译器来分配的,将这个宏定义1的话可以由用户自行设置,堆内存在 heap_1 , heap_2 , heap_3 , heap_4 , heap_5 中都有定义

- configASSERT:

  断言,类似 C 标准库中的 assert() 函数,调试代码的时候可以检查传入的参数是否合理, FreeRTOS 内核中的关键点都会调用 configASSERT(x) ,当 x=0 时说明有错误发生,主要用于调试, configASSERT() 需要在 FreeRTOSConfig.h 文件中定义,例如:

  ```
  #define configASSERT((x)) if((x)==0) vAssertCalled(__FILE,__LINE__);
  //vAssertCalled需要用户自行定义,例如
  #define vAssertCalled(char,char) printf("Errors:%s,%d\r\n",char,int)
  //调试完成的时候,尽量去掉断言,防止增加开销
  ```

- configCHECK_FOR_STACK_OVERFLOW:

  设置堆栈溢出检测,每个任务都有一个堆栈,使用函数 xTaskCreate() 创建一个任务则任务的堆栈自动从 FreeRTOS 的堆(ucHeap)中分配,大小由函数参数决定,使用函数 xTaskCreateStatic() 创建,则堆栈由用户设置,任务堆栈作为参数,一般是一个数组

  ```
  //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,否则条用钩子函数,优点就是快,但不能拿检测所有的
  /*----该宏为1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
  ```

- configCPU_CLOCK_HZ:

  设置 cpu 的频率

- configSUPPORT_DYNAMIC_ALLOCATION:

设置为1的话在创建 FreeRTOS 的内核对象所需要的 RAM 就会从 FreeRTOS 的堆中动态获取内存,定义为0的时候 RAM 需要由用户自行提供,默认为1

- configENABLE_BACKWORD_COMPATIBILITY:

  FreeRTOS.h 中有一些列的 #define 宏定义,这些宏定义都是一些数据类型名字, V8.0.0 之前的 FreeRTOS 中会使用这些数据类型,这些宏保证了你的代码从 V8.0.0 之前的版本升级到最新版本的时候不需要作出修改,默认为1

- configGENERATE_RUN_TIME_STATS:

  设置1开启时间统计功能,相应的 API 函数会编译,为0时关闭时间统计

  如果为1还需要:

  portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() 初始化一个外设来作为时间统计的基准时钟

  portGET_RUN_TIME_COUNTER_VALUE() 或 portALT_GET_RUN_TIME_COUNTER_VALUE(time) 用来返回当前基准时钟的时钟值

- configIDLE_SHOULD_YIELD:

  此宏定义了与空闲任务(idle Task)处于同优先级的其他用户任务的行为,为0时空闲任务不会为其他同优先级任务让出 CPU 使用权,为1时空闲任务就会处于同优先级的用户任务让出 CPU 使用权,除非没有就绪的用户任务,这样花费在空闲任务上的时间就会很少,一般都关闭这个功能

- configKERNEL_INTERRUPT_PRIORITY,configMAX_SYSCALL_INTERRUPT,configMAX_API_CALL_INTERRUPT_PRIORITY:

  与FreeRTOS的中断配置有关

- configMAX_CO_ROUTINE_PRIORITIES:

  设置分配给协程的最大优先级,优先级最低为0,最大为configMAX_CO_ROUTINE_PRIORITIES-1

- configMAX_PRIORITIES:

  设置任务优先级数量,设置好后任务优先级从0到configMAX_PRIORITIES-1 , 0最低

- configMAX_TASK_NAME_LEN:

  设置任务名最大长度

- configMINIMAL_STACK_SIZE:

  设置空闲任务的最小任务堆栈大小,以字为单位

- configNUM_THREAD_LOCAL_STORAGE_POINTERS:

  设置每个任务的本地存储指针数组大小,任务控制块中有本地存储数组指针,用户应用程序可以在这些本地存储中存入一些数据

- configQUEUE_REGISTRY_SIZE:

  设置可注册的队列和信号量的最大数量,在使用内核调试器查看信号量和队列的时候需要设置此宏,而且要先将消息队列和信号量进行注册,只有注册了的队列和信号才会在内核调试器中看到,如果不使用内核调试器的话设置0即可

- configSUPPORT_STATIC_ALLOCATION:

  定义1时,在创建内核对象时需要用户指定RAM,为0时就会使用heap.c中动态内存管理函数来自动申请RAM

- config_TICK_RATE_HZ:

  设置FreeRTOS的系统时钟节拍频率单位为HZ,就是SysTick的中断频率

- configTIMER_QUEUE_LENGTH:

配置FreeRTOS软件定时器,软件定时器API函数会通过命令队列向软件定时器任务发送消息,这个宏用来设置软件定时器的命令队列长度

- `configTIMER_TASK_PRIORITY`:

  设置软件定时器的任务优先级

- `configTIMER_TASK_STACK_DEPTH`:

  设置定时器服务任务的任务堆栈大小

- `configTOTAL_HEAP_SIZE`:

  设置堆大小,如果使用了动态管理内存的话,`FreeRTOS`在创建任务,信号量,队列等的时候就会使用`heap_x.c`中的内存申请函数来申请内存,这些内存就是从堆`ucHeap[configTOTAL_HEAP_SIZE]`中申请的

- `configUSE_16_BIT_TICKS`:

  设置系统节拍计数器变量数据类型,系统节拍计数器变量类型为`TickType_t`,为1时则是16位 为2时则是32位

- `configUSE_APPLICATION_TASK_TAG`:

  设置1的话函数`configUSE_APPLICATION_TASK_TAG()`和`xTaskCallpplicationTaskHook()`就会编译

- `configUSE_CO_ROUTINES`:

  为1时启用协程,协程可以节省开销,但功能有限,建议关闭

- `configUSE_COUNTING_SEMAPHORES`:

  设置1启用计数型信号量,相关`API`函数就会编译

- `configUSE_DAEMON_TASK_STARTUP_HOOK`:

  宏`configUSE_TIMERS`和`configUSE_DAEMON_TASK_STARTUP_HOOK`都为1的时候需要定义函数`vApplicationDaemonTaskStartupHook()`

- `configUSE_IDLE_HOOK`:

  为1时使用空闲任务函数`vApplicationIdleHook()`

- `configUSE_MALLOC_FAILED_HOOK`:

  为1时启用内存分配失败函数,需要用户实现

- `configUSE_MUTEXES`:

  为1时使用互斥信号量

- `configUSE_PORT_OPTIMISED_TASK_SELECTION`:

  当此宏为1的时候，切换任务时使用硬件计算前导零(`CLZ`)指令，为0的时候，使用遍历通用方法。

- `configUSE_PREEMPTION`:

  为1时使用抢占式调度器,为0时使用协程,如果使用抢占式调度器的话内核会在每个时钟节拍中断中进行任务切换,当使用协程的话会在如下地方进行任务切换:

  一个任务调用了函数`taskTIELD()`

  一个任务调用了可以使任务进入阻塞态的`API`函数

  应用程序明确定义了在中断中执行上下文切换

- `configUSE_QUEUE_SETS`:

  为1时启用队列集功能

- `configUSE_RECURSIVE_MUTEXES`：

  为1时使用递归互斥信号量,相关API函数会编译

- `configUSE_STATS_FORMATTING_FUNCTIONS`：

  宏`configUSE_STATS_FORMATTING_FUNCTIONS`和`configUSE_TRACE_FACILITY`都为1时函数`vTaskList()`和`vTaskGetRunTimeStats()`会编译

- `configUSE_TASK_NOTIFICATIONS`：

  为1时使用任务通知功能,开启了功能每个任务会多消耗8个字节

- `configUSE_TICK_HOOK`：

  为1时使能时间片钩子函数,用户需要实现:
  `void vApplicationTickHook(void)`

- `configUSE_TICKLESS_IDLE`：

  为1时使能低功耗`tickless`模式

- `configUSE_TIMERS`：

  为1时使用软件定时器,而且宏`configTIMER_TASK_PRIORITY`,`configTIMER_QUEUE_LENGTH`和`configTIMER_TASK_STACK_DEPTH`必须定义

- `configUSE_TIMER_SLICING`：

  默认`FreeRTOS`使用抢占式调度器,这意味着调度器永远都在执行已经就绪了的最高优先级任务,优先级相同的任务会在时钟节拍中断中进行切换,为0时不会进行切换,默认为1

- `configUSE_TRACE_FACILITY`：

  为1时启用可视化跟踪调试,会增加一些结构体成员和API函数

2.cortex-m,svc和pendsv解析

svc

```
__asm void prvStartFirstTask( void )
{
    PRESERVE8

    /* Use the NVIC offset register to locate the stack. */
    ldr r0, =0xE000ED08
    ldr r0, [r0]
    ldr r0, [r0]
    /* Set the msp back to the start of the stack. */
    msr msp, r0
    /* Globally enable interrupts. */
    cpsie i
    cpsie f
    dsb     /* clear cpu data line */
    isb     /* clear cpu assembly line */
    /* Call SVC to start the first task. */
    svc 0
    nop
    nop
```

```
}
```

```
__asm void vPortSVCHandler( void )
{
    PRESERVE8

    /* Get the location of the current TCB. */
    ldr r3, =pxCurrentTCB
    ldr r1, [r3]
    ldr r0, [r1]    /* get the topStackPointer of current task */
    /* Pop the core registers. */
    /* pop the r4-r11 of topStackPointer to the r4-r11 of cpu, at the same time,
r0(topStackPointer) increase with step 4 */
    ldmia r0!, {r4-r11, r14}
    msr psp, r0 /* update the new stack pointer to psp */
    isb     /* clear cpu assembly line */
    mov r0, #0
    msr basepri, r0 /* allow all interrupt */
    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

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

主要流程如下所示:

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
    /* cpu load (psp update) r0~r3 r12 psr pc r14 to oldTask of Stack
automatically */
    mrs r0, psp /* store the psp of old taskStack */
    isb /* wait the assembly line */
    /* Get the location of the current TCB. */
    ldr r3, =pxCurrentTCB
    ldr r2, [r3]

    /* Save the core registers. */
    /* store the r4~r11 of cpu to the oldTask of Stack */
    stmdb r0!, {r4-r11, r14}

    /* Save the new top of stack into the first member of the TCB. */
    str r0, [r2] /* the r0 is the TopStackPointer of Old Task */
    /* Here , the old Context Store process is completed */
    /* push oldTaskPointer to stack because of protection , */
    stmdb sp!, {r3}
    /* shutdown the interrupt to make basepri*/
    /* enter the critical zone */
    mov r0, #configMAX_SYSCALL_INTERRUPT_PRIORITY
    cpsid i
    msr basepri, r0
    dsb
    isb
    cpsie i

    bl vTaskSwitchContext /* call taskswitch */
    mov r0, #0
    msr basepri, r0
    /* exit the critical zone */
    ldmia sp!, {r3} /* recover the r3 (msp) */

    /* The first item in pxCurrentTCB is the task top of stack. */
    ldr r1, [r3] /* load new taskTCB to r1 */
    ldr r0, [r1] /* load new stackpointer to r0 */

    /* Pop the core registers to cpu. */
    ldmia r0!, {r4-r11, r14}

    msr psp, r0 /* save the psp */
    isb /* clear the assembly line */

    bx r14  /* when exception appear , r14 save the flag of EXC_RETURN */
}
```

3.freertos中断管理

中断配置宏

- `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`：设置最低优先级,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,即直接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*/
```

开关中断

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

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

任务级代码保护

```
    #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
```

中断级临界段代码保护

```
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);
}
```

4.freertos时间管理

vTaskDelay()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。

xTaskDelayUntil()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

`vTaskDelayUntil()` 源码:

```
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，所以程序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

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 ) );
                    //检查任务是否等待某个时间,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;
}
```

5.freertos确定任务栈大小

与编写裸机应用程序完全一样，所需的堆栈量取决于以下应用程序特定参数

- 函数调用嵌套深度
- 函数作用域变量声明的数量和大小
- 函数参数个数
- 处理器架构
- 编译器
- 编译器优化级别

中断服务程序(ISR)的栈要求 —— 对于许多 RTOS 端口来说为零，
因为 RTOS 将在进入中断服务程序时切换到使用专用中断栈。

每次调度程序临时停止运行任务以运行不同的任务时，

处理器上下文都会保存到任务的栈中。

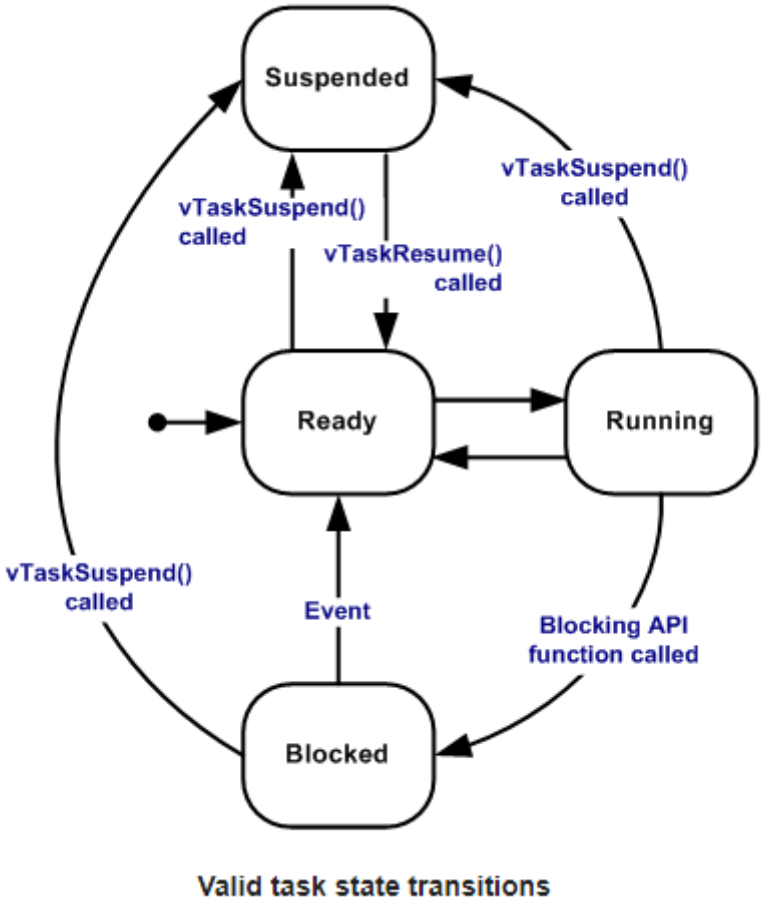
保存的处理器上下文会在下次任务运行时从任务栈中弹出。

保存处理器上下文所需的栈空间是任务的栈需求中唯一来自于 RTOS 本身的额外需求。。

6.freertos任务管理

任务状态

- 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

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

  当任务因为延时或等待事件的发生时会处于阻塞状态，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）

任务具体管理

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运行态任务

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

通过链表理解调度机制

- 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

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 */
    }
}
```

freertos任务接口

任务创建

```
portBASE_TYPE xTaskCreate(pdTASK_CODE pvTaskCode,
                          const signed char * const pcName,
                          unsigned short usStackDepth,
                          void *pvParameters,
                          unsigned portBASE_TYPE uxPriority,
                          xTaskHandle *pvCreatedTask
                         );
return -> pdPASS or errCOULE_NOT_ALLOCATE_REQUIRED_MEMORY
            //动态创建任务_堆栈由函数自行创建
//任务的接口函数一般为死循环,通常由延时,信号量,队列等引起调度并进入阻塞态。
//一般不会跳出死循环,要退出则调用vTaskDelete;
portBASE_TYPE xTaskCreateStatic(pdTASK_CODE pvTaskCode,
                          const signed char * const pcName,
                          unsigned short usStackDepth,
                          void *pvParameters,
                          unsigned portBASE_TYPE uxPriority,
                          StackType_t *puxStackBuffer, //任务堆栈数组
                          xTaskHandle *pvCreatedTask
                         );
// return NULL or xTaskHandle
portBASE_TYPE xTaskCreateRestricted(xTaskParameters *pxTaskDefinition,
                                    xTaskHandle *pxCreatedTask);
//此函数要求MCU要有MPU,用此函数创建的任务会受到MPU保护
// return pdPASS or other
```

任务删除

```
void vTaskDelete(xTaskHandle pxTask);
//如果任务的空间是由用户分配的,则需要手动释放,例如pvPortMalloc() pxPortfree()
```

任务挂起和恢复

```
void vTaskSuspend(xTaskHandle pxTaskToSuspend);
//用于将某个任务设置为挂起态,进入挂起态的任务永远不会进入运行态,退出的唯一方法就是调用恢复函数
//当参数为NULL，表示挂起自己
void vTaskSuspendAll(void);
//将调度器挂起
void vTaskResume(xTaskHandle pxTaskToResume);
//将任务从挂起状态恢复
portBASE_TYPE vTaskResumeAll(void);
portBASE_TYPE vTaskResumeFromISR(xTaskHandle pxTaskToResume);
//return
//pdTRUE  恢复运行的任务的任务优先级等于或者高于正在运行的任务(被中断打断的任务),这意味着在退出中断服务函数以后必须进行一次上下文切换
//pdFALSE 恢复运行的任务的任务优先级低于当前正在运行的任务(被中断打断的任务),这意味着在退出中断服务函数的以后不需要进行上下文切换
```

freertos任务控制块

```
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;
```

7.freertos任务调度

执行系统调用

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

内核时钟节拍中断

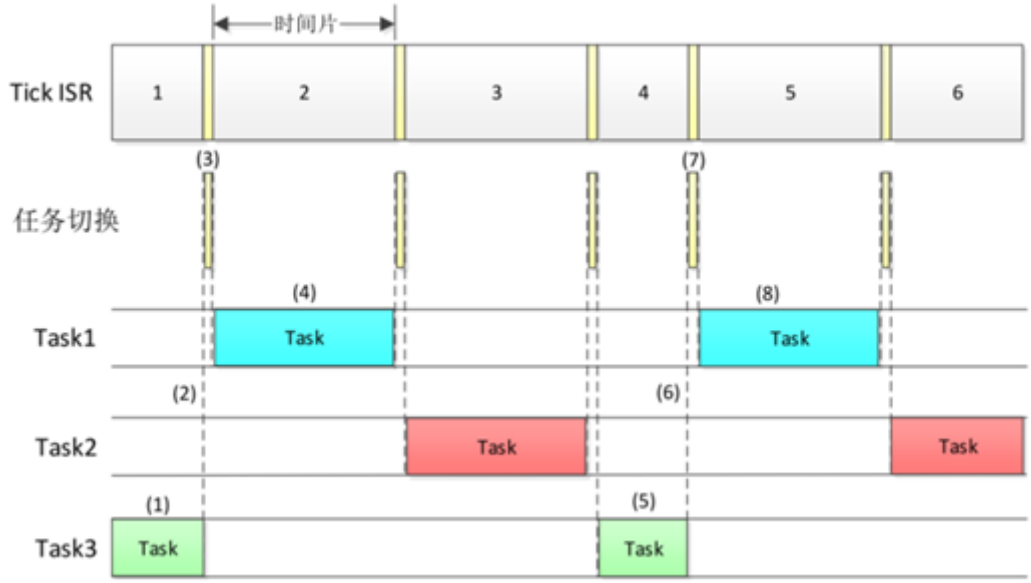
FreeRTOS中Systick的ISR也会进行任务切换

```
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
}
```

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

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

- 一种通用方法
- 一种硬件方法

使用方法通过宏`configUSE_PORT_OPTIMISED_TASK_SELECTION`来决定,为1时使用硬件方法,否则为通用方法

通用方法

就是所有处理器都可以使用的方法:

```
    #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
```

8.freertos其他API

内核接口

```
taskYIELD()                         任务切换
taskENTER_CRITICAL()                    进入临界区,任务级
taskEXIT_CRITICAL()                 退出临界区,任务级
taskENTER_CRITICAL_FROM_ISR()       进入临界区,ISR级
taskEXIT_CRITICAL_FROM_ISR()        退出临界区,ISR级
taskDISABLE_INTERRUPTS()                关闭中断
taskENABLE_INTERRUPTS()             打开中断
vTaskStartScheduler()               开启任务调度器
vTaskEndScheduler()                 关闭任务调度器
vTaskSuspendAll()                       挂起任务调度器
xTaskResumeAll()                        恢复任务调度器
vTaskStepTick()                     设置系统节拍
void vTaskEndScheduler( void )
{
    portDISABLE_INTERRUPTS();
    xSchedulerRunning = pdFALSE;
    vPortEndScheduler();
}
void vTaskSuspendAll( void )
{
    //uxSchedulerSuspended是挂起嵌套计时器
    //调度器挂起是支持嵌套的
    //使用函数xTaskResumeAll()即可恢复任务调度器
    //调用了几次vTaskSuspendAll()挂起调度器,同样也得调用几次xTaskResumeAll()才会最终恢复
调度器
    ++uxSchedulerSuspended;
}
BaseType_t xTaskResumeAll( void )
{
    TCB_t * pxTCB = NULL;
    BaseType_t xAlreadyYielded = pdFALSE;
```

```
    configASSERT( uxSchedulerSuspended );
    //进入临界区
    taskENTER_CRITICAL();
    {
        //调度器挂起嵌套计数器-1
        --uxSchedulerSuspended;
        //如果uxScheDulerSuspended为0说明所有的挂起都解除,调度器运行
        if( uxSchedulerSuspended == ( UBaseType_t ) pdFALSE )
        {
            if( uxCurrentNumberOfTasks > ( UBaseType_t ) 0U )
            {
                //循环处理列表xPendingReadyList,只要不为空,说明还有任务挂到了列表
xPendingReadyList上,这里需要将这些任务从列表xPendingReadyList上移除并添加到任务所对应的就
绪列表中
                while( listLIST_IS_EMPTY( &xPendingReadyList ) == pdFALSE )
                {
                    //遍历列表xPendingReadyList,获取挂到列表xPendingReadyList上的任务
对应的TCB
                    pxTCB = listGET_OWNER_OF_HEAD_ENTRY( ( &xPendingReadyList )
);
                    //将任务从事件列表上移除
                    listREMOVE_ITEM( &( pxTCB->xEventListItem ) );
                    portMEMORY_BARRIER();
                    //将任务从状态列表上移除
                    listREMOVE_ITEM( &( pxTCB->xStateListItem ) );
                    //将任务添加到就绪列表中
                    prvAddTaskToReadyList( pxTCB );
                    //判断任务的优先级是否高于当前的任务,如果是则需要将xYieldPending标记,
需要进行任务切换
                    if( pxTCB->uxPriority >= pxCurrentTCB->uxPriority )
                    {
                        xYieldPending = pdTRUE;
                    }
                    else
                    {
                        mtCOVERAGE_TEST_MARKER();
                    }
                }
                if( xYieldPending != pdFALSE )
                {
                    #if ( configUSE_PREEMPTION != 0 )
                        {
                            //标记在函数xTaskResumeAll()进行了任务切换,使用变量
xAlreadyYielded
                            xAlreadyYielded = pdTRUE;
                        }
                    #endif
                    //进行任务切换
                    taskYIELD_IF_USING_PREEMPTION();
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
        }
```

```
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    //退出临界区
    taskEXIT_CRITICAL();
    return xAlreadyYielded;
}
```

其他api

```
uxTaskPriorityGet()                     查询某个任务的优先级
vTaskPrioritySet()                      改变某个任务的任务优先级
uxTaskGetSystemState()                  获取系统中任务状态
vTaskGetInfo()                          获取某个任务信息
xTaskGetApplicationTaskTag()            获取某个任务的标签值
xTaskGetCurrentTaskHandle()             获取当前正在运行的任务句柄
xTaskGetHandler()                           根据任务名字查找任务句柄
xTaskGetIdleTaskHandle()                    获取空闲任务的任务句柄
uxTaskGetStackHighWaterMark()           获取任务堆栈的历史剩余最小值
eTaskGetState()                         获取某个任务的状态
pcTaskGetName()                         获取某个任务的任务名字
xTaskGetTickCount()                     获取系统时间计数器值
xTaskGetTickCountFromISR()          在中断服务函数中获取时间计数器值
xTaskGetSchedulerState()                    获取任务调度器的状态
uxTaskGetNumberOfTasks()                    获取当前系统中存在的任务数量
vTaskList()             以表格形式输出当前系统中所有任务的详细信息
vTaskGetRunTimeStats()                  获取每个任务的运行时间
vTaskSetApplicationTaskTag()            设置任务标签值
SetThreadLocalStoragePointer()          设置线程本地存储指针
GetThreadLocalStoragePointer()          获取线程本地存储指针
```