FreeRTOS-中断管理

Created time	@November 16, 2022 7:58 AM
Last edited time	@November 18, 2022 5:50 PM

Cortex-M中断管理：

- 中断简介

  中断由硬件产生，Cortex-M内核的MCU提供了一个中断管理的嵌套向量中断控制器（NVIC）。Cortex-M3的NVIC最多支持240个中断请求（IRQ），1个不可屏蔽中断（NMI），1个滴答定时器（Systick）和多个系统异常。

编号	类型	优先级	简介
0	N/A	N/A	没有异常在运行
1	复位	-3（最高）	复位
2	NMI	-2	不可屏蔽中断（来自外部 NMI 输入脚）
3	硬(hard)fault	-1	所有被除能的 fault，都将“上访”(escalation)成硬 fault。只要 FAULTMASK 没有置位，硬 fault 服务例程就被强制执行。Fault 被除能的原因包括被禁用，或者 FAULTMASK 被置位。
4	MemManage fault	可编程	存储器管理 fault，MPU 访问犯规以及访问非法位置均可引发。企图在“非执行区”取指也会引发此 fault
5	总线 fault	可编程	从总线系统收到了错误响应，原因可以是预取流产（Abort）或数据流产，或者企图访问协处理器
6	用法(usage) Fault	可编程	由于程序错误导致的异常。通常是使用了一条无效指令，或者是非法的状态转换，例如尝试切换到 ARM 状态
7-10	保留	N/A	N/A
11	SVCall	可编程	执行系统服务调用指令（SVC）引发的异常
12	调试监视器	可编程	调试监视器（断点，数据观察点，或者是外部调试请求
13	保留	N/A	N/A
14	PendSV	可编程	为系统设备而设的“可悬挂请求”（pendable request）
15	SysTick	可编程	系统滴答定时器（也就是周期性溢出的时基定时器——译注）

- 中断优先级

  Cortex-M处理器有些中断的优先级是固定的，比如复位、NMI、HardFault，这些中断的优先级都是负数，优先级最高。Cortex-M处理器有3个固定优先级和256个可编程的优先级，但实际的优先级数量是芯片厂商决定。STM32只有16级优先级，利用高四位作为优先级。

  STM32用了4位作为优先级分组，因此最多有5组优先级分组设置。NVIC_Priority_Group_4表示4位全部都是抢占优先级，没有亚优先级。移植FreeRTOS的时候配置的就是组4.

```
#define NVIC_PriorityGroup_0         ((uint32_t)0x700) /*!< 0 bits for pre-emption priority
                                                            4 bits for subpriority */
#define NVIC_PriorityGroup_1         ((uint32_t)0x600) /*!< 1 bits for pre-emption priority
                                                            3 bits for subpriority */
#define NVIC_PriorityGroup_2         ((uint32_t)0x500) /*!< 2 bits for pre-emption priority
                                                            2 bits for subpriority */
#define NVIC_PriorityGroup_3         ((uint32_t)0x400) /*!< 3 bits for pre-emption priority
                                                            1 bits for subpriority */
#define NVIC_PriorityGroup_4         ((uint32_t)0x300) /*!< 4 bits for pre-emption priority
                                                            0 bits for subpriority */
```

- 优先级设置

  FreeRTOS在设置PendSV和Systick的中断优先级的时候都是直接操作地址0xE000_ED20。

  用于中断屏蔽的特殊寄存器：

  1. PRIMASK寄存器

     用于禁止除NMI和HardFault外的所有异常和中断。直接利用汇编语言改值。

```
CPSIE   I;   //清除 PRIMASK(使能中断)
CPSID   I;   //设置 PRIMASK(禁止中断)
```

     PRIMASK 寄存器还可以通过 MRS 和 MSR 指令访问，如下：

```
MOVS    R0, #1
MSR     PRIMASK,    R0   ;//将 1 写入 PRIMASK 禁止所有中断
```

     以及：

```
MOVS    R0, #0
MSR     PRIMASK,    R0   ;//将 0 写入 PRIMASK 以使能中断
```

  2. FAULTMASK寄存器

     可以屏蔽HardFault。使用方法与PRIMASK类似。

  3. BASEPRI寄存器

     FreeRTOS的开关中断就是操作该寄存器实现。可以关闭低于某个阈值的中断，高于这个阈值的中断就不会被关闭。

FreeRTOS中断配置

- 配置宏参数

  1. configPRIO_BITS：设置MCU使用几位优先级。STM32用的是4位，因此该值为4。
  2. configLIBRARY_LOWEST_INTERRUPT_PRIORITY：设置最低优先级。STM32优先级使用了4位，而且NVIC_Priority_Group_4，即4位抢占优先级，因此优先级数是16个，最低优先级是15，所以改值设置为15.

     P.S：不同MCU，此宏的值不同，具体根据MCU架构来定。
  3. configKERNEL_INTERRUPT_PRIORITY：设置内核中断优先级。而PendSV和滴答定时器的中断优先级也是根据该宏来确定。如下图所示。

     从定义可以看出，该宏的值是configLIBRARY_LOWEST_INTERRUPT_PRIORITY左移4位，因为STM32使用了4位优先级，而这4位是高4位，因此需要左移4位才是设置真正的优先级。

     PendSV和Systick的优先级寄存器对应此32位数据的最高8位和次高8位，因此左移16位和24位。

```
#define configKERNEL_INTERRUPT_PRIORITY ( configLIBRARY_LOWEST_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )
#define portNVIC_PENDSV_PRI ( ( ( uint32_t ) configKERNEL_INTERRUPT_PRIORITY ) << 16UL )
#define portNVIC_SYSTICK_PRI ( ( ( uint32_t ) configKERNEL_INTERRUPT_PRIORITY ) << 24UL )
```

  4. configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY：设置FreeRTOS系统可管理的最大优先级，STM32中一般设置为5。即高于5的优先级（0-4优先级）不归FreeRTOS管理。
  5. configMAX_SYSCALL_INTERRUPT_PRIORITY：此宏设置好后，低于此优先级的中断可以安全的调用FreeRTOS的API函数。

```
#define configMAX_SYSCALL_INTERRUPT_PRIORITY   ( configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )
```

FreeRTOS开关中断

```
#define portDISABLE_INTERRUPTS() vPortRaiseBASEPRI() //关中断
#define portENABLE_INTERRUPTS() vPortSetBASEPRI( 0 ) //开中断
```

- 临界段代码

  指那些必须完整运行，不能被打断的代码段，因此FreeRTOS在进入临界段代码的时候需要关闭中断。FreeRTOS系统本身有很多临界段代码，这些代码都需要加临界段代码保护。

  在中断使能之前需要进入临界区，只有所有的临界段代码都退出以后才会使能中断。具体实现如下段程序所示。

```
void start_task(void *pvParameters)
{
    taskENTER_CRITICAL();           //进入临界区
    //创建中断测试任务
    xTaskCreate((TaskFunction_t )interrupt_task,
                (const char*    )"interrupt_task",
                (uint16_t       )INTERRUPT_STK_SIZE,
                (void*          )NULL,
                (UBaseType_t    )INTERRUPT_TASK_PRIO,
                (TaskHandle_t*  )&INTERRUPTTask_Handler);
  vTaskDelete(StartTask_Handler); //退出初始化任务
    taskEXIT_CRITICAL();            //退出临界区
}
```