FreeRTOS - Interrupt

FreeRTOS only handles 3 interrupts which are SVC, PendSV, and SysTick. All remaining interrupts are handled by application, and they are not blocked by tasks. However, there are some rules have to be followed in order to not mess up the task stacks and task priority in scheduler.

#arm #stm32 #rtos #interrupt

Last update: 2021-08-10 16:50:49

Table of Content

1. Interrupts

SVC interrupt

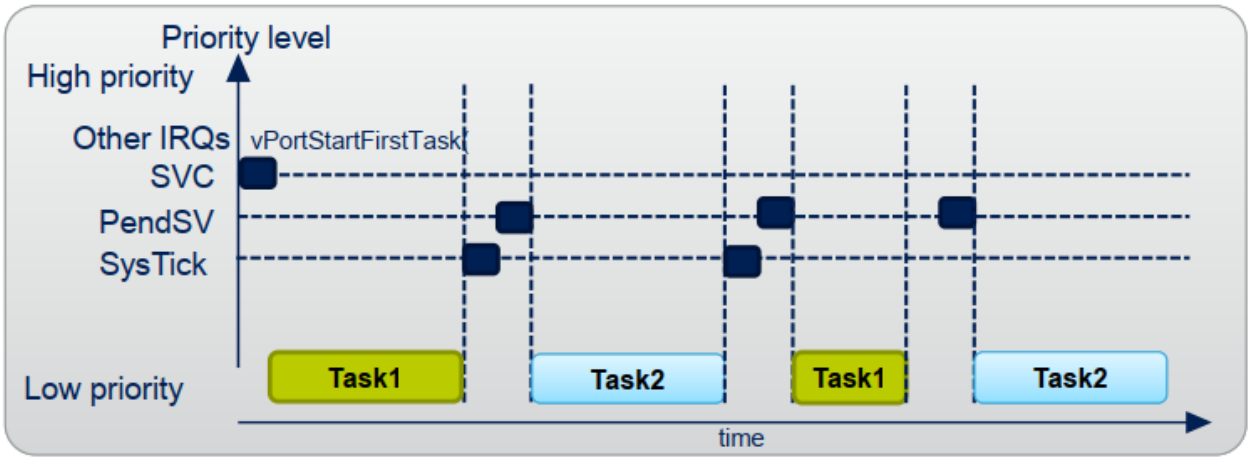
- Interrupt risen by SVC instruction
- SVC 0 call used only once, to start the scheduler (within `vPortStartFirstTask()` which is used to start the kernel)

SysTick timer

- Lowest NVIC interrupt priority
- Used for task switching on `configTICK_RATE_HZ` regular time base
- Set PendSV if context switch is necessary

PendSV interrupt

- Used for task switching before tick rate
- Lowest NVIC interrupt priority
- Not triggered by any peripheral



Kernel interrupts

- FreeRTOS kernel and its IRQ procedures (`PendSV` , `SysTick`) have the lowest possible interrupt priority (255) set in `FreeRTOSConfig.h` by the declaration `configKERNEL_INTERRUPT_PRIORITY` .
- There is a group of interrupts which can cooperate with FreeRTOS API by calling its functions. Maximum level for those peripherals (based on the position in vector table) is set in `configMAX_SYSCALL_INTERRUPT_PRIORITY` .
- It is possible to use nested interrupts.

ISR is independent of all tasks

The code of the ISR does not belong to any part of the task code of FreeRTOS. Each ISR is a function, but it is not a task, and it will not be called by any task.

- ISR uses the stack in different way of a task. FreeRTOS allocates independent stack space for each task, which is used to save local variables of functions and so on. When an interrupt occurs, certain registers of the CPU will be saved to the current stack (instead of specifying a task stack), and then the ISR program will be executed.
  - If the code of a certain task is currently being executed, it will occupy the stack of the task
  - If the code of another ISR is currently being executed and interrupt nesting occurs, then the stack of the task that was interrupted earlier may continue to be used
- ISR does not change the status of the current task. Although the execution of the currently running task is suspended after the IRQ occurs, and the CPU executes the code of the ISR, the state of the current task is still Running, and it does not change to another state - this is obviously different from the task being preempted.

  Even if the FreeRTOS API is called in the ISR, other tasks with higher priority than the current task are awakened (turned to the Ready state), and the task switching operation will be performed after the ISR returns, and the task to run will be reselected. In fact, the ISR does not know what the currently running task is, and it does not make sense to actively change the current task status.

2. API functions in Interrupts

In the FreeRTOS documentation, it has been emphasized that the API functions whose names end with `FromISR` must be called in the ISR, instead of the conventional API. This is because the execution environment and tasks of the ISR are different.

The API called in the ISR requires a quick return and no waiting is allowed. Some APIs cannot be used in ISR because they have blocking functions, or they can change their functions, including parameter passing requirements.

For example, there are two APIs for using Semaphore:

```
xSemaphoreGiveFromISR(semaphore, *pxHigherPriorityTaskWoken)
```

and

```
xSemaphoreGiveFromISR(semaphore)
```

The only difference for the programmer is additional argument `*pxHigherPriorityTaskWoken` which is used to determine whether a higher priority task is awakened, and the ISR itself decides whether to switch tasks. If this parameter is `pdTRUE`, context switch (PendSV IRQ) should be requested by `portYIELD_FROM_ISR()` in kernel before the interrupt exits.

When using CMSIS API, this process is automatically handled by the library (by checking `IPSR` content) and is transparent.

3. Signal from ISR to Task

In order to support real-time response to hardware events, the interrupt service routine (ISR) must be executed as soon as possible. Because the system may have a variety of interrupts, the ISR needs to be programmed as short as possible, and return after critical operations are performed to allow other interrupts to be processed.

It's preferred to post a message from an ISR using `FromISR` API and then handle that message in a task space. However, this may cause latency in processing the request if the target task is not scheduled to run soon or in the worst case, it is blocked.

Only if the requirement is to have very low latency of processing, ISR can process the data.

One solution to process data immediately after the ISR is post a message to the font of the queue, and request schedule to switch to a task.

```
void vBufferISR( void )
{
char cIn;
BaseType_t xHigherPriorityTaskWoken;

    /* We have not woken a task at the start of the ISR. */
    xHigherPriorityTaskWoken = pdFALSE;

    /* Obtain a byte from the buffer. */
    cIn = portINPUT_BYTE( RX_REGISTER_ADDRESS );

    if( cIn == EMERGENCY_MESSAGE )
    {
        /* Post the byte to the front of the queue. */
        xQueueSendToFrontFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );
    }
    else
    {
        /* Post the byte to the back of the queue. */
        xQueueSendToBackFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );
    }

    /* Did sending to the queue unblock a higher priority task? */
    /* Actual macro used here is port specific. */
```

```
    portYIELD_FROM_ISR (xHigherPriorityTaskWoken);
}
```

4. Interrupt status

There maybe a code which can be call either in a task or in an ISR. However, it should be implemented to be run in both cases without impact the execution.

The `IPSR` (Interrupt Program Status Register) is read to determine whether there is an interrupt is being processed.

```
register uint32_t __regIPSR    __ASM("ipsr");
```

An example of checking IPSR:

```
uint32_t osGetSysTick(void) {

    /* read out IPSR register*/
    register uint32_t __regIPSR     __ASM("ipsr");

    /* regIPSR == 0 if controller is in thread mode */
    if (__regIPSR == 0) {
        return xTaskGetTickCount();
    } else {
        return xTaskGetTickCountFromISR();
    }
}
```