FreeRTOS

[STM32 FreeRTOS] 자료 Review#3

이지훈
2020. 3. 4. 23:30

이웃추가

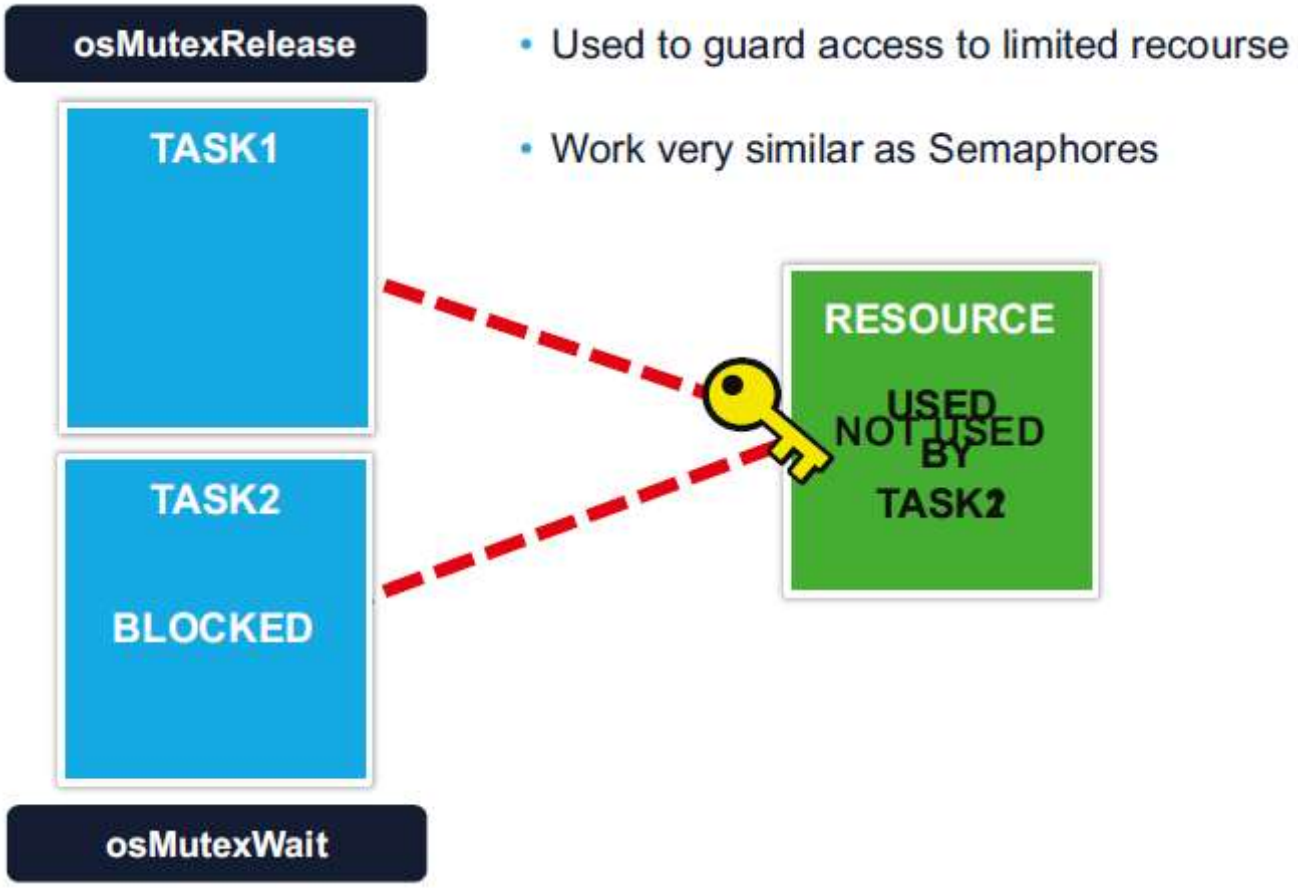
※ 내용에 오류가 있을 수 있습니다. 오류에 대해서는 Feedback 부탁 드리겠습니다.

https://github.com/eziya/STM32F4_HAL_FREERTOS_LAB

eziya/STM32F4_HAL_FREERTOS_LAB
FreeRTOS Testing for STM32F4. Contribute to eziya/STM...
github.com

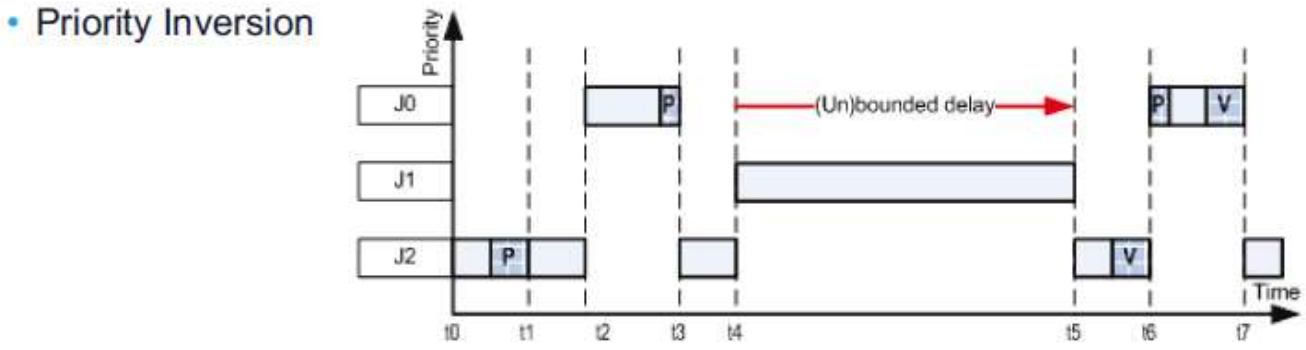
ST 에서 제공하는 자료들을 기반으로 FreeRTOS 의 기본 특성에 대해서 정리하여 보고 해당 자료의 Lab 코드들을 테스트해 보고자 합니다.

Mutex 는 Priority Inheritance 동작 지원하는 Binary 세마포어라고 생각하면 된다.
Binary 세마포어이기 때문에 하나의 Resource 에 대한 접근은 하나의 Task 만 가능하다.



Mutex 가 지원하는 Priority Inhertitance 는 Priority Inversion 이슈를 해결하기 위한 것으로 Mutex 를 소유하고 있는 task 의 priority 가 일시적으로 mutex 를 소유하고자 하는 task 의 highest priority 와 동일해지는 동작을 의미한다. mutex 소유권을 넘기면 priority 는 원상 복귀 한다.

Priority Inversion 은 아래 그림과 같은 상황을 의미한다.
그림을 보면 J0 의 우선순위가 높지만 J2 가 소유한 Mutex 를 갖고자 blocked 되면서 중간 우선 순위인 J1 이 동작하게 되면서 의도하지 않은 우선순위 동작이 발생하게 된다.



따라서 J2의 우선순위를 J0 의 우선순위와 동일하게 일시적으로 상승시키면 J1이 동작하지 않고 J2 가 빠르게 수행되고 Mutex 를 반환하면 J0 가 우선순위에 맞게 J1 보다 먼저 동작할 수 있게 된다.

[LAB#7]

프로젝트 구성

- 2개의 Task 를 생성한다.
- Task1, Task2 는 osPriorityNormal 로 우선순위를 설정한다.
- USE_MUTEXES 를 Enable 한다.
- Mutex 를 한개 생성한다.
- Task1 과 Task2 는 2초 간격으로 Mutex 를 대기한다.

동작

- 두 Task 가 동시에 Mutex 자원을 얻으려고 해도 하나의 Task 만 Mutex 를 얻을 수 있기 때문에 두 Task 의 printf 문은 번갈아 가면서 출력이 된다.

```
/* USER CODE BEGIN Header_StartTask1 */
/**
 * @brief  Function implementing the Task1 thread.
 * @param  argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartTask1 */
void StartTask1(void const *argument)
{
    /* USER CODE BEGIN StartTask1 */
    /* Infinite loop */
    for (;;)
    {
        osDelay(2000);
        if (osMutexWait(myMutex01Handle, 1000) == osOK)
        {
            printf("Task1 print\n");
            osMutexRelease(myMutex01Handle);
        }
        else
        {
            printf("Task1 mutex timeout or error\n");
        }
    }
    /* USER CODE END StartTask1 */
}

/* USER CODE BEGIN Header_StartTask2 */
/**
```

```
  * @brief Function implementing the Task2 thread.
  * @param argument: Not used
  * @retval None
  */
/* USER CODE END Header_StartTask2 */
void StartTask2(void const *argument)
{
    /* USER CODE BEGIN StartTask2 */
    /* Infinite loop */
    for (;;)
    {
        osDelay(2000);
        if (osMutexWait(myMutex01Handle, 1000) == osOK)
        {
            printf("Task2 print\n");
            osMutexRelease(myMutex01Handle);
        }
        else
        {
            printf("Task2 mutex timeout or error\n");
        }
    }
    /* USER CODE END StartTask2 */
}
```

Software Timer 는 FreeRTOS 차원에서 지원하는 타이머로 복수개의 SW 방식 타이머 제공 가능

SW 방식으로 처리하기 때문에 정확도는 높지 않지만 주기적인 동작을 처리하기에는 적합

타이머에서 지정한 시간이 흐른 뒤에는 Callback 함수가 호출됨

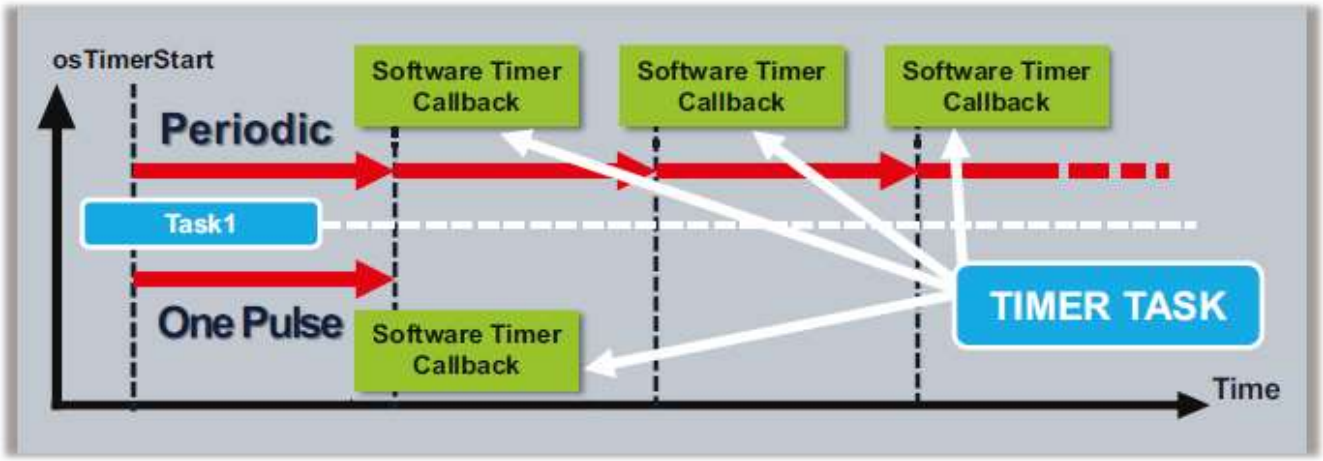
반복적으로 callback 이 호출되는 Periodic 모드와 일회성으로 호출되는 One Pulse 모드 2가지 방식 제공

SW Timer callback 에서는 osDelay 등 blocking 함수를 호출해서는 안된다.

Software Timer 는 하나의 별도 시스템 task 로 구동하는데 해당 task 는 보통 configMAX_PRIORITIES - 1 로 최대값을 적용하는 것이 타이머 오차를 줄일 수 있다.

Software Timer 의 Queue Length 는 타이머 동작을 위한 command 큐의 크기를 지정한다.

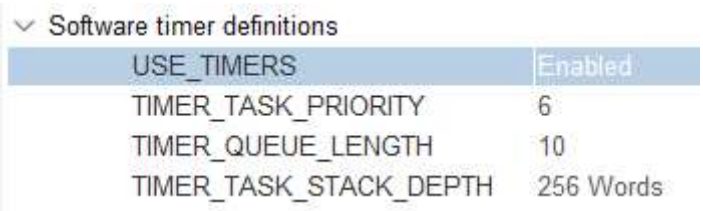
Software Timer 의 Stack Depth 는 타이머 task 의 Stack 사이즈를 지정하는 값으로 word 단위로 설정한다.

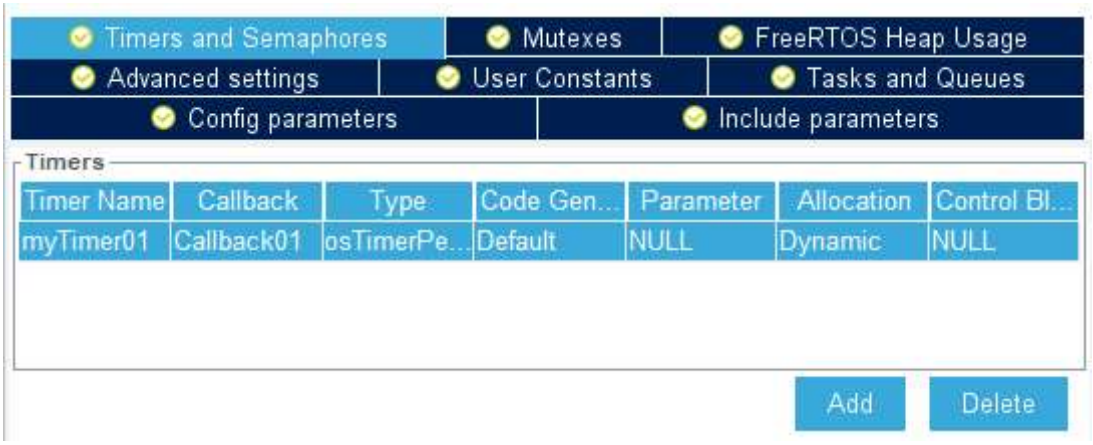


[LAB#8]

프로젝트 구성

- 1개의 Task 를 생성한다.
- Task1은 osPriorityNormal 로 우선순위를 설정한다.
- USE_TIMERS 를 Enable 한다.
- Software Timer task 의 priority 는 최대값인 6으로 설정한다.
- 타이머를 하나 생성한다.

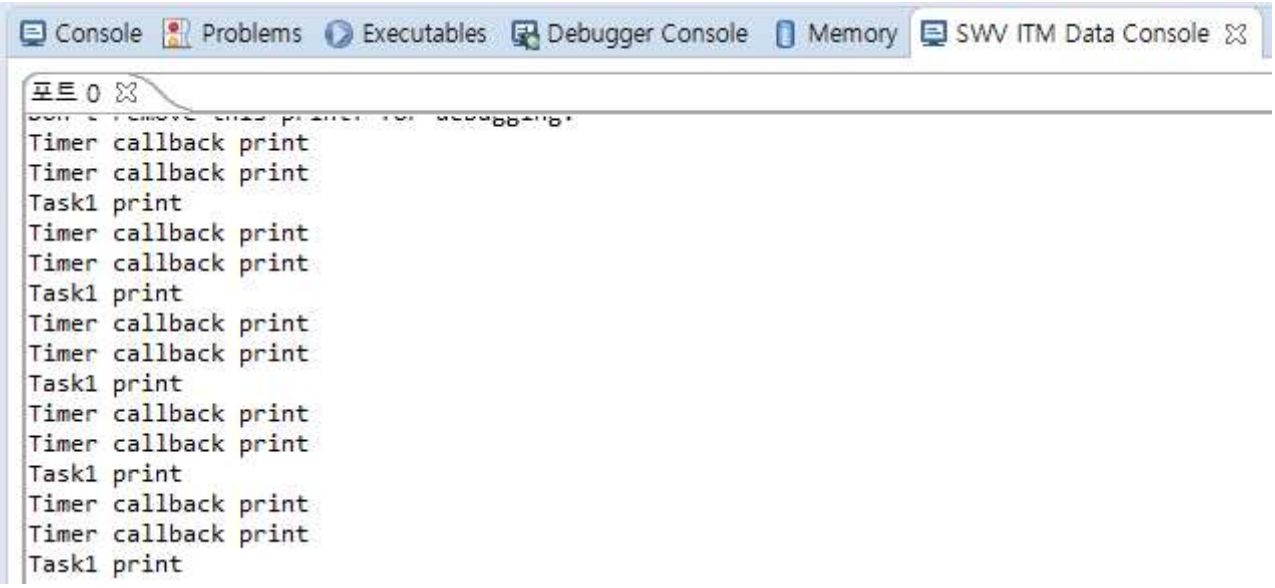




동작

- Task1 은 2초마다 printf 로 문자열을 출력한다.
- Task1 루프 시작 전에 Periodic 타이머를 시작하였기 때문에 매 1초마다 Callback01 함수가 호출되면서 printf 로 문자열을 출력한다.
- 따라서, 1초마다 "Timer callback print" 가 출력되고 2초마다 "Task1 print" 가 출력된다.

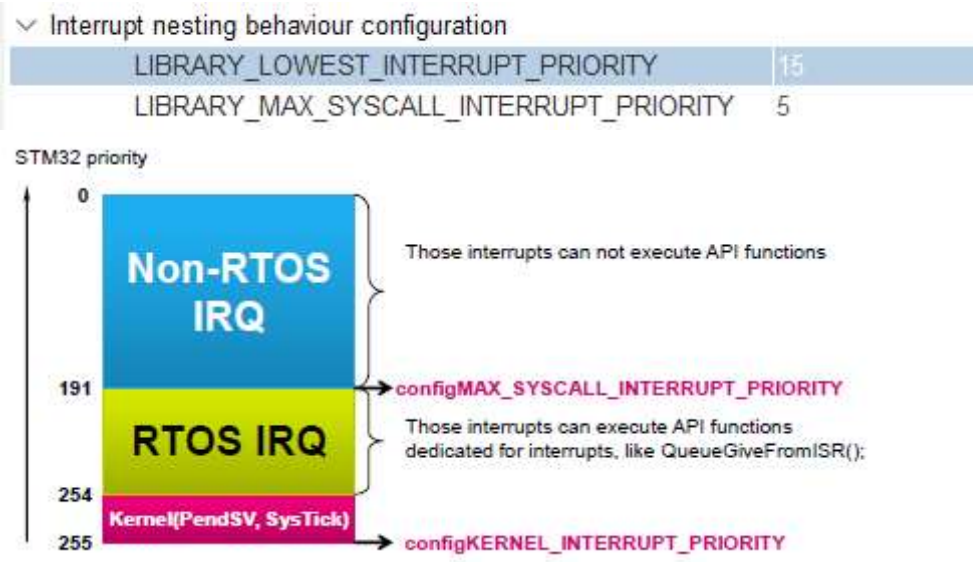
```c
/* USER CODE BEGIN Header_StartTask1 */
/**
 * @brief  Function implementing the Task1 thread.
 * @param  argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartTask1 */
void StartTask1(void const *argument)
{
    /* USER CODE BEGIN StartTask1 */
    osTimerStart(myTimer01Handle, 1000);
    /* Infinite loop */
    for (;;)
    {
        osDelay(2000);
        printf("Task1 print\n");
    }
    /* USER CODE END StartTask1 */
}

/* Callback01 function */
void Callback01(void const *argument)
{
    /* USER CODE BEGIN Callback01 */

    //no blocking function here
    printf("Timer callback print\n");
    /* USER CODE END Callback01 */
}
```



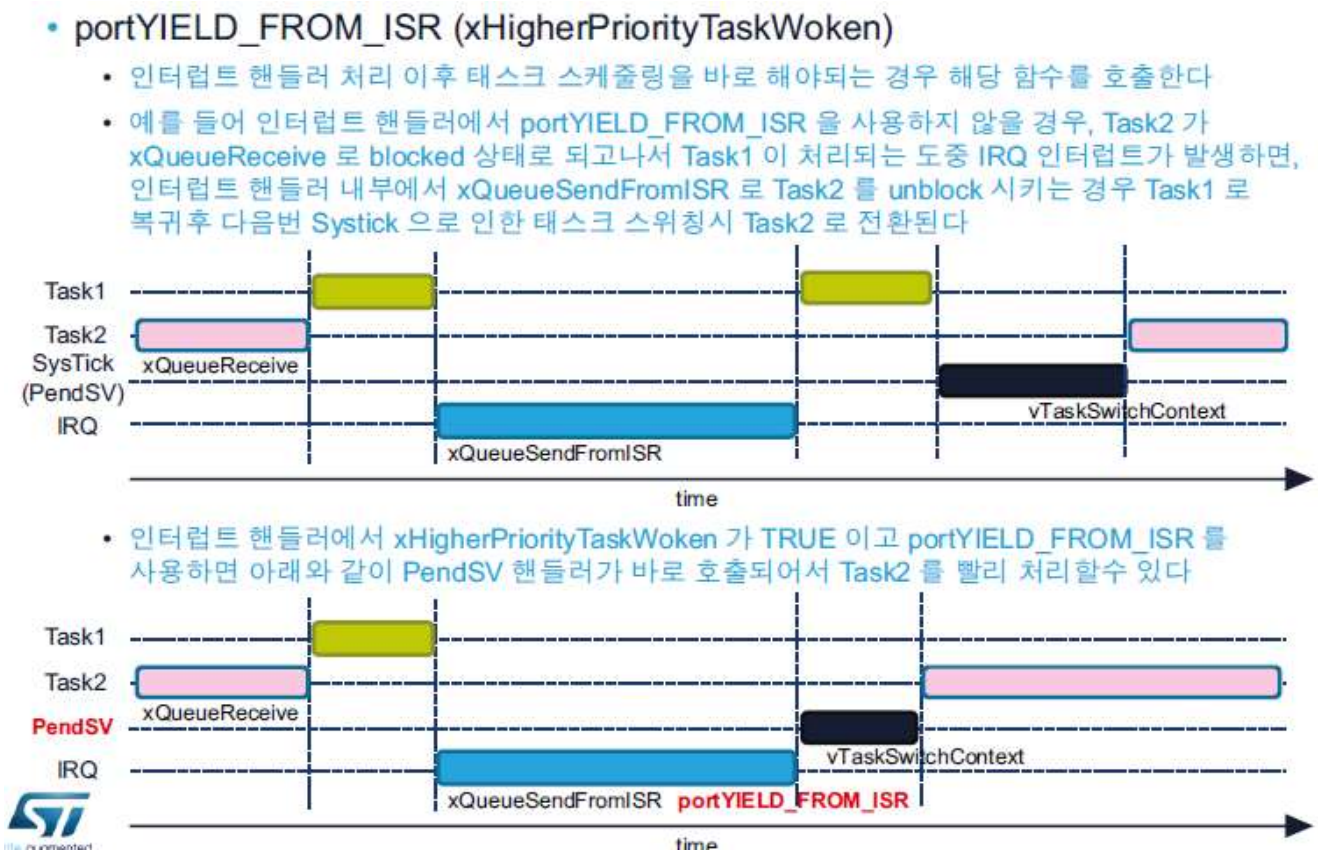
인터럽트

인터럽트의 priority 와 task 의 priority 는 별개이다. task 의 prioriy 는 task 사이의 스케쥴링 시퀀스를 결정하는데 사용될 뿐이다.
FreeRTOS 의 task priority 는 큰 숫자가 높은 우선순위를 의미한다. 반면 인터럽트의 우선순위는 작은 숫자가 높은 우선 순위를 의미한다.
FreeRTOS 는 ISR 내부에서 호출하는 함수의 경우 별도의 FromISR 함수를 사용하는 반면 CMSIS RTOS API 는 동일한 API 를 사용하고 API 내부에서 판단하여 분기하도록 되어 있다.

아래 그림을 보면 Non-RTOS IRQ 들의 우선순위가 높고 RTOS IRQ 들의 우선순위가 낮다.
가장 우선순위가 낮은 IRQ 는 Contex Switching 과 관련한 PendSV, SysTick 이다.
FreeRTOS 에서는 인터럽트 prioirty 관련 두가지 설정이 존재한다.
configKERNEL_INTERRUPT_PRIORITY 는 FreeRTOS 스케쥴러 인터럽트 우선쉰위로 PendSV 와 SysTick 인터럽트의 우선순위를 결정한다.
configMAX_SYSCALL_INTERRUPT_PRIORITY 는 FreeRTOS API 를 호출하지 않는 인터럽트 중 가장 낮은 우선순위를 결정한다. 따라서 FreeRTOS 의 API 를 호출하는 모든 인터럽트 들은 해당 인터럽트 보다 낮은 우선순위를 할당해야 한다.



인터럽트 핸들러를 처리한 후 바로 Task 스케쥴링을 위해서는 portYIELD_FROM_ISR 함수를 호출한다.
만일 인터럽트 핸들러에서 해당 Yield 함수를 호출하지 않는다면 Context switching 이 발생하지 않는다.
즉, ISR 내부에서 특정 task 를 ready 상태로 변경시키더라도 즉시 Context switching 이 수행되지 않고 Systick 에 의한 context switching 이 발생하게 된다.

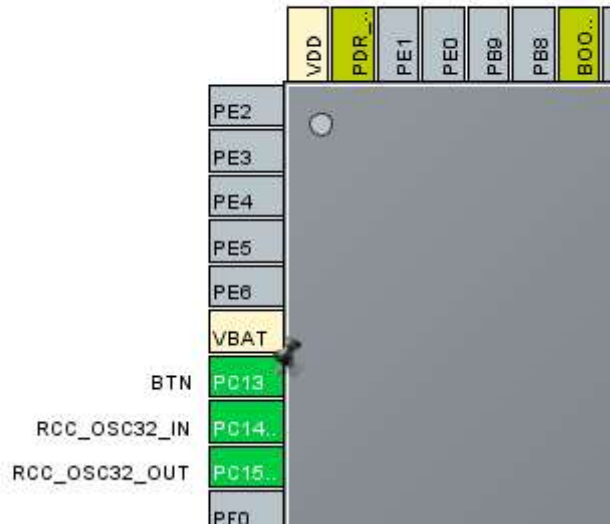


리소스를 보호하는 방법에는 Critical Section 을 사용하는 방법이나 Scheduler 를 Suspend 하는 방법이 사용될 수 있다.

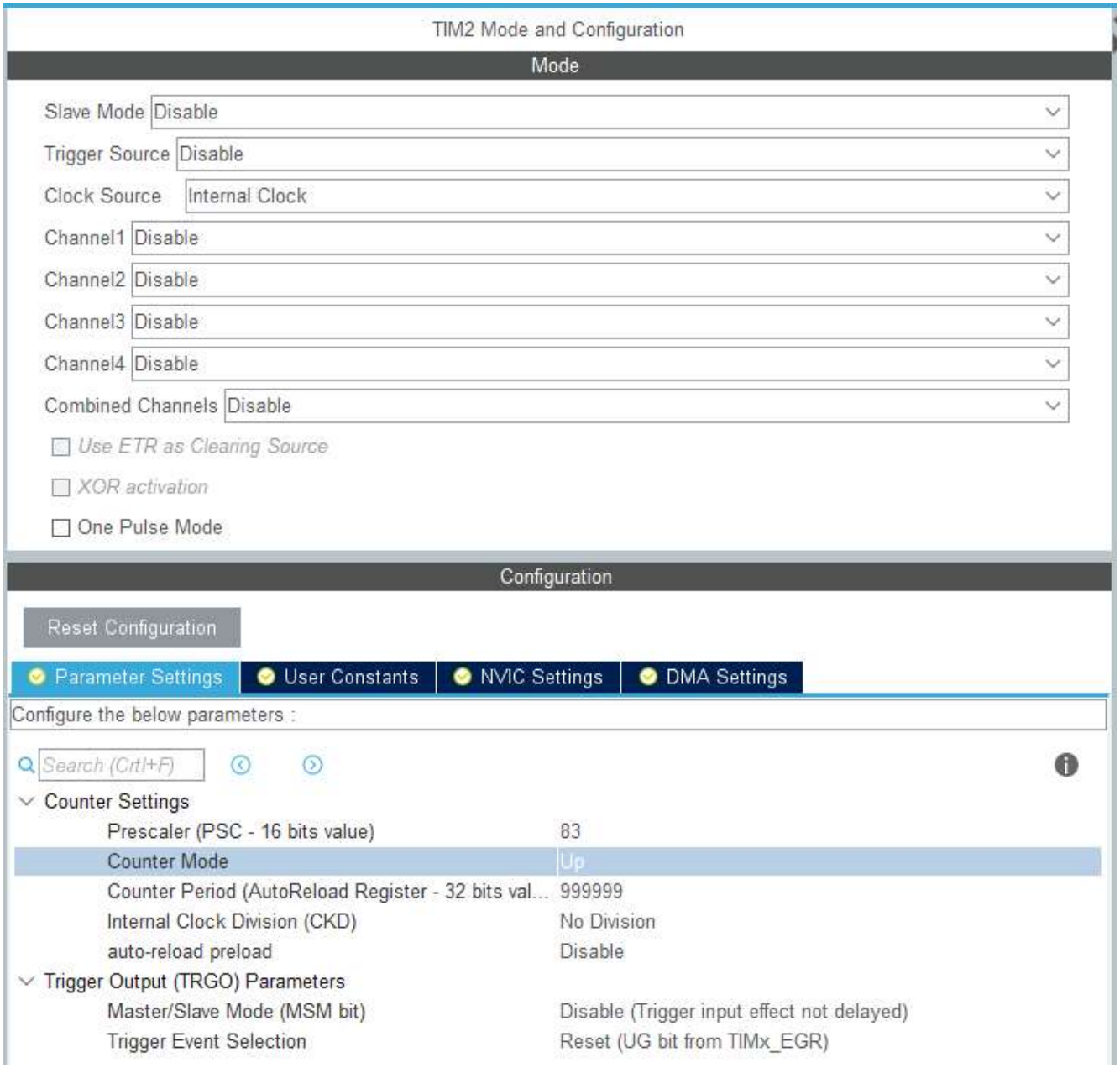
우선 Critical Section 은 전역변수 Access 시나 공유 자원 보호를 위해서 사용할 수 있다. Critical Secion 은 configMAX_SYSCALL_INTERRUPT_PRIORITY 보다 우선순위가 낮은 Interrupt 를 disable 하기 때문에 RTOS IRQ 로부터 atomic 한 operation 을 보장할 수 있다. 하지만 configMAX_SYSCALL_INTERRUPT_PRIORITY 보다 우선순위가 높은 인터럽트로부터는 보호할 수 없다.

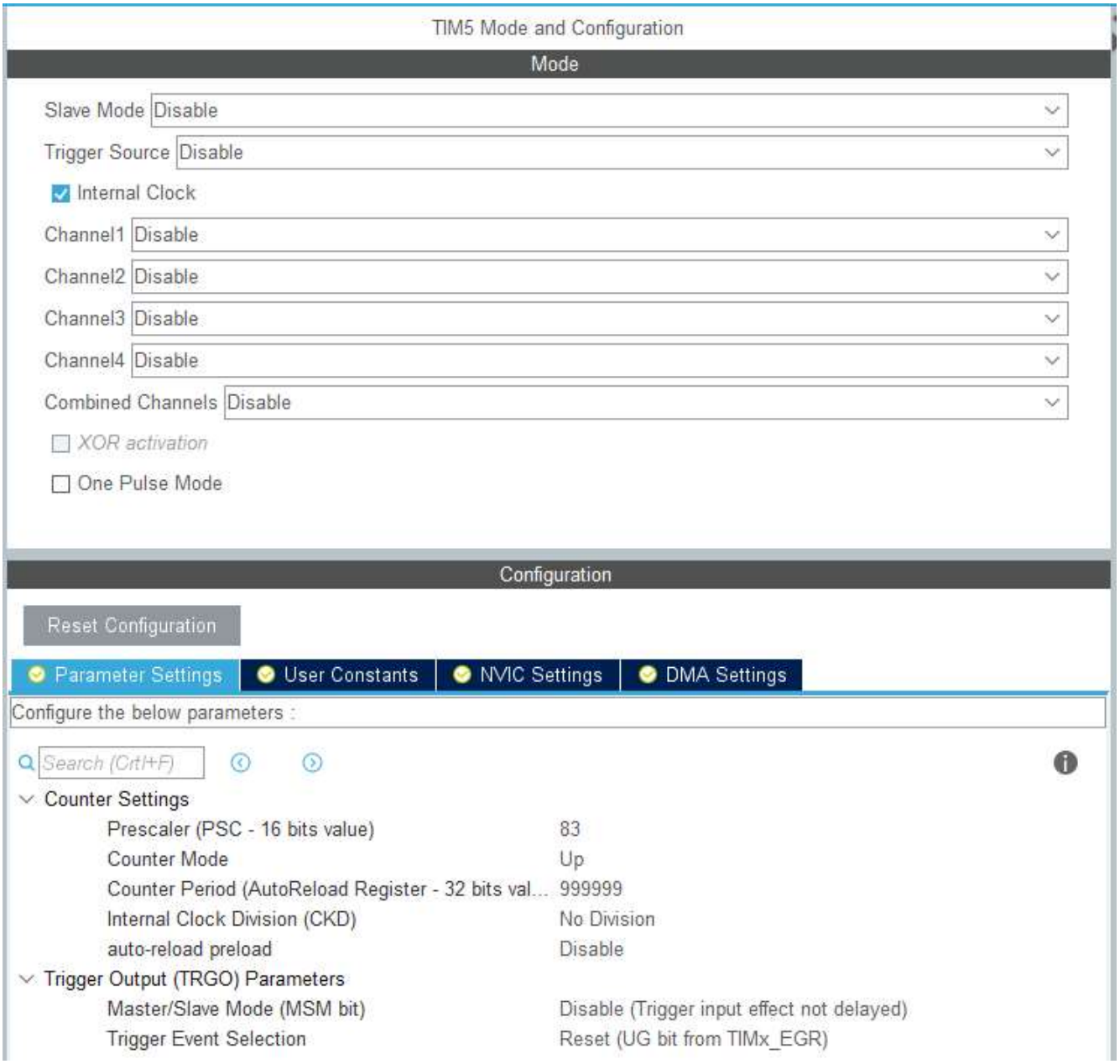
Scheduler 를 Suspend 하는 방식은 context switching 을 발생하지 않도록 하기 때문에 다른 우선 순위가 높은 task 가 선점하는 것은 막을 수 있지만 인터럽트에 의한 선점은 막을 수 없다. 개인적으로 Scheduler 를 중지하는 것보다는 Critical section 이나 Mutex 등을 이용해서 보호하는 방식이 더 올바른 방식이라 판단된다.

[LAB#9]
프로젝트 구성
- PA0 핀을 GPIO Input 으로 설정

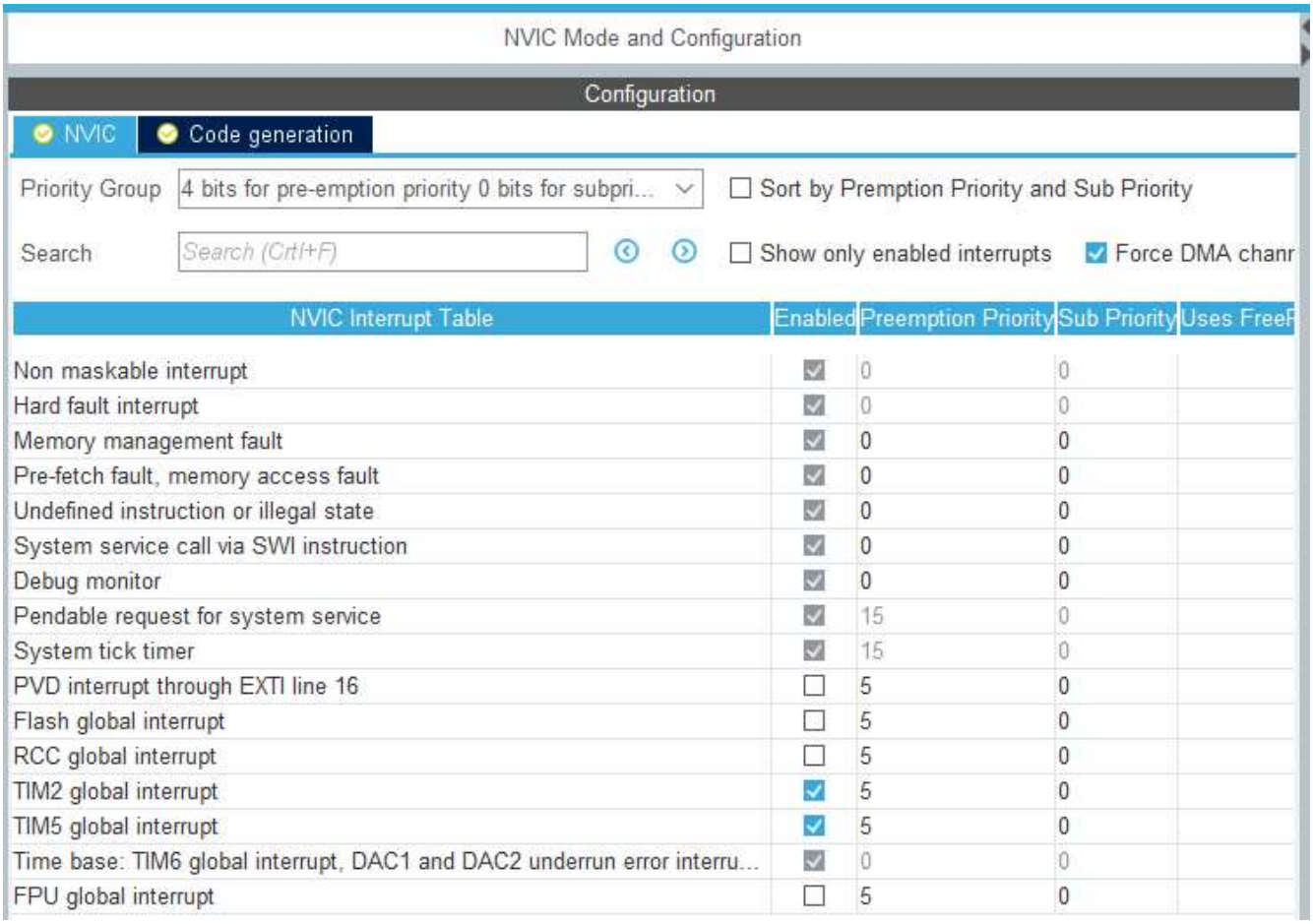


- TIM2, TIM5 를 활성화 하고 주기를 1초로 설정





- TIM2, TIM5 인터럽트 enable 하고 Use FreeRTOS functions 를 체크하면 우선순위는 LIBRARY_LOWEST_INTERRUPT_PRIORITY(15) 와 LIBRARY_MAX_SYSCAL_INTERRUPT_PRIORITY(5) 사이의 값으로 설정할 수 있다.


NVIC Mode and Configuration

Configuration

NVIC | Code generation

Priority Group: 4 bits for pre-emption priority 0 bits for subpri... | Sort by Premption Priority and Sub Priority

Search: Search (Crtl+F) | Show only enabled interrupts | Force DMA chanr

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority	Uses FreeF
Non maskable interrupt	☑	0	0	
Hard fault interrupt	☑	0	0	
Memory management fault	☑	0	0	
Pre-fetch fault, memory access fault	☑	0	0	
Undefined instruction or illegal state	☑	0	0	
System service call via SWI instruction	☑	0	0	
Debug monitor	☑	0	0	
Pendable request for system service	☑	15	0	
System tick timer	☑	15	0	
PVD interrupt through EXTI line 16	☐	5	0	
Flash global interrupt	☐	5	0	
RCC global interrupt	☐	5	0	
TIM2 global interrupt	☑	5	0	
TIM5 global interrupt	☑	5	0	
Time base: TIM6 global interrupt, DAC1 and DAC2 underrun error interru...	☑	0	0	
FPU global interrupt	☐	5	0	

- Task1 은 osPriorityNormal 로 생성한다.
- Task1 은 1초마다 PA0 핀을 상태를 체크하는데 이때 Critical section 을 사용한다.

동작

- PA0 핀이 GPIO_PIN_SET 상태인 경우에는 while 루프를 벗어나지 못해서 Critical section 에 enter 한 상태를 유지하게 되고 이때 RTOS IRQ 들은 발생하지 않는다.
- 만일 Critical section 을 사용하지 않는다면 RTOS IRQ 들이 발생하기 때문에 TIM2, TIM5 의 callback 이 호출되면서 정상적으로 메시지가 출력된다.

```
/* USER CODE BEGIN Header_StartTask1 */
/**
 * @brief  Function implementing the Task1 thread.
 * @param  argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartTask1 */
void StartTask1(void const *argument)
{
    /* USER CODE BEGIN StartTask1 */
    /* Infinite loop */
    for (;;)
    {
        taskENTER_CRITICAL();
        while (HAL_GPIO_ReadPin(BTN_GPIO_Port, BTN_Pin) == GPIO_PIN_SET)
        taskEXIT_CRITICAL();

        printf("Task1\n");
        osDelay(1000);
    }
    /* USER CODE END StartTask1 */
}
```

우선 critical section 을 사용하는 경우를 살펴보면 BTN 핀이 SET 되는 경우 무한 루프 동작을 하면서 critical section 을 벗어나지 않기 때문에 버튼이 눌려있는 동안은 TIM2, TIM5 인터럽트가 발생하지 않는다.

반대로 critical section 을 사용하지 않은 로그를 살펴보면 아래와 같이 Task1 로그 없이 TIM2, TIM5 로그가 출력되는 것을 확인할 수 있다.

```
/* USER CODE END Header_StartTask1 */
void StartTask1(void const *argument)
{
    /* USER CODE BEGIN StartTask1 */
    /* Infinite loop */
    for (;;)
    {
        //taskENTER_CRITICAL();
        while (HAL_GPIO_ReadPin(BTN_GPIO_Port, BTN_Pin) == GPIO_PIN_SET)
        //taskEXIT_CRITICAL();

        printf("Task1\n");
        osDelay(1000);
    }
    /* USER CODE END StartTask1 */
}
```

만일 TIM2 의 우선순위를 RTOS IRQ 범위보다 높게 설정하면 ciritical section 을 사용하더라도 인터럽트가 발생하게 된다.

```
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef* tim_baseHandle)
{

  if(tim_baseHandle->Instance==TIM2)
  {
  /* USER CODE BEGIN TIM2_MspInit 0 */

  /* USER CODE END TIM2_MspInit 0 */
    /* TIM2 clock enable */
    __HAL_RCC_TIM2_CLK_ENABLE();

    /* TIM2 interrupt Init */
    //HAL_NVIC_SetPriority(TIM2_IRQn, 5, 0);
    HAL_NVIC_SetPriority(TIM2_IRQn, 4, 0); //TIM2 의 우선순위를 RTOS IRQ
    HAL_NVIC_EnableIRQ(TIM2_IRQn);
  /* USER CODE BEGIN TIM2_MspInit 1 */

  /* USER CODE END TIM2_MspInit 1 */
  }
  else if(tim_baseHandle->Instance==TIM5)
  {
  /* USER CODE BEGIN TIM5_MspInit 0 */

  /* USER CODE END TIM5_MspInit 0 */
    /* TIM5 clock enable */
    __HAL_RCC_TIM5_CLK_ENABLE();

    /* TIM5 interrupt Init */
    HAL_NVIC_SetPriority(TIM5_IRQn, 5, 0);
    HAL_NVIC_EnableIRQ(TIM5_IRQn);
  /* USER CODE BEGIN TIM5_MspInit 1 */

  /* USER CODE END TIM5_MspInit 1 */
  }
}
```

#stm32 #freertos

2 0

이지훈

달릴 준비만 하는거 아냐...달려야 하는데...^^; https://github.com/eziya

이웃추가