

# FreeRTOS

## Interrupt & Critical Section

- 인터럽트 priority 와 태스크 priority 는 별개로 설정한다
- 낮은 priority 인터럽트라도 높은 priority 의 태스크를 pre-empt 할수 있지만 태스크는 인터럽트를 pre-empt 할수 없다
- FreeRTOS 에서 낮은 priority 인터럽트는 낮은 숫자 (예, 0)로, 높은 priority 인터럽트는 높은 숫자 (예, 15)로 표시한다
- Cortex-M4 는 이와 반대로 0 이 높은 priority, 15 는 낮은 priority 인터럽트를 의미한다
- FreeRTOS 는 기능은 같고 끝에 FromISR 이름이 붙는 별도의 함수를 제공하는데 인터럽트 핸들러에서는 해당 함수를 사용해야 한다 (예를 들어 xQueueSend() 함수 대신에 xQueueSendFromISR() 함수를 사용한다)
- CMSIS RTOS 는 인터럽트 핸들러 전용 함수가 나뉘어 있지 않다 (함수 내부에 if 문으로 인터럽트 상태인지 아닌지 구분하므로 똑같은 함수 예를 들어 osMessagePut() 를 사용한다)

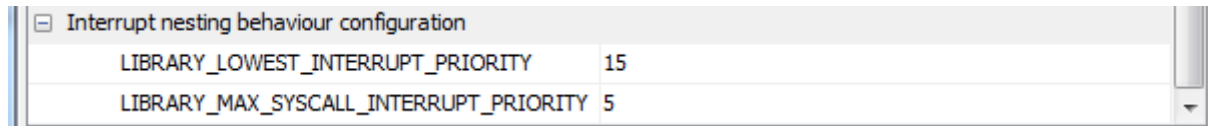
- STM32L4 (Cortex-M4) 는 아래와 같은 5개의 인터럽트 priority 그룹 정책중에 하나를 선택할 수 있다 (SCB->AIRCRR)
  - 0 bit for pre-empt priority and 4 bits for sub-priority
  - 1 bit for pre-empt priority and 3 bits for sub-priority
  - 2 bit for pre-empt priority and 2 bits for sub-priority
  - 3 bit for pre-empt priority and 1 bits for sub-priority
  - 4 bit for pre-empt priority and 0 bits for sub-priority
- 위에 설정한 priority 정책에 맞게 각각의 인터럽트에 pre-empt 와 sub-priority 숫자를 설정한다 (NVIC->IPR[x])
- Sub-priority 와 무관하게 높은 pre-empt 의 인터럽트가 낮은 pre-empt 인터럽트를 pre-empt 할 수 있다 (예를 들어 인터럽트 priority 를 낮게 설정한 UART 인터럽트 핸들러 수행 도중에 priority 를 높게 설정한 USB 인터럽트 핸들러로 pre-empt 할 수 있다)
- Pre-empt 는 동일하고 sub-priority 가 다른 인터럽트끼리는 pre-empt 하지 않지만 sub-priority는 pending 상태에 있는 exception들의 순서는 결정한다.

- Cortex-M4 의 Reset, NMI, Hard fault 인터럽트 등은 priority 가 고정되어 있는데 IRQ 부터는 칩 제조사와 종류에 따라 달라지며 priority 는 사용자가 0 부터 15 까지 가변하게 설정할 수 있다

Table 17. Properties of the different exception types

Exception number <sup>(1)</sup>	IRQ number <sup>(1)</sup>	Exception type	Priority	Vector address or offset <sup>(2)</sup>	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	Hard fault	-1	0x0000000C	-
4	-12	Memory management fault	Configurable <sup>(3)</sup>	0x00000010	Synchronous
5	-11	Bus fault	Configurable <sup>(3)</sup>	0x00000014	Synchronous when precise Asynchronous when imprecise
6	-10	Usage fault	Configurable <sup>(3)</sup>	0x00000018	Synchronous
7-10	-	-	-	Reserved	-
11	-5	SVCall	Configurable <sup>(3)</sup>	0x0000002C	Synchronous
12-13	-	-	-	Reserved	-
14	-2	PendSV	Configurable <sup>(3)</sup>	0x00000038	Asynchronous
15	-1	SysTick	Configurable <sup>(3)</sup>	0x0000003C	Asynchronous
16 and above	0 and above	Interrupt (IRQ)	Configurable <sup>(4)</sup>	0x00000040 and above <sup>(5)</sup>	Asynchronous

- FreeRTOS 는 인터럽트 priority 와 관련해서 두가지 상수 설정을 할 수 있으며 CubeMX 에서는 아래 설정에 해당한다

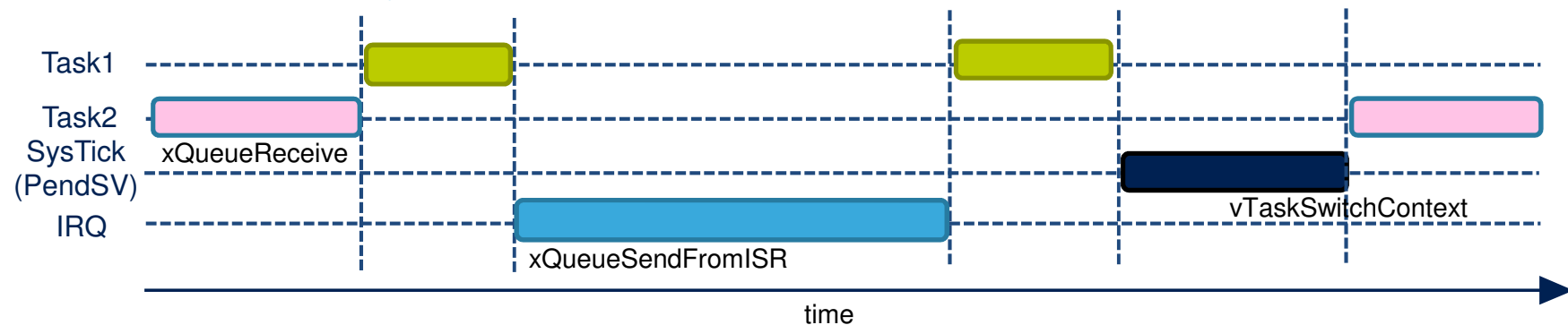


Interrupt nesting behaviour configuration	
LIBRARY_LOWEST_INTERRUPT_PRIORITY	15
LIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY	5

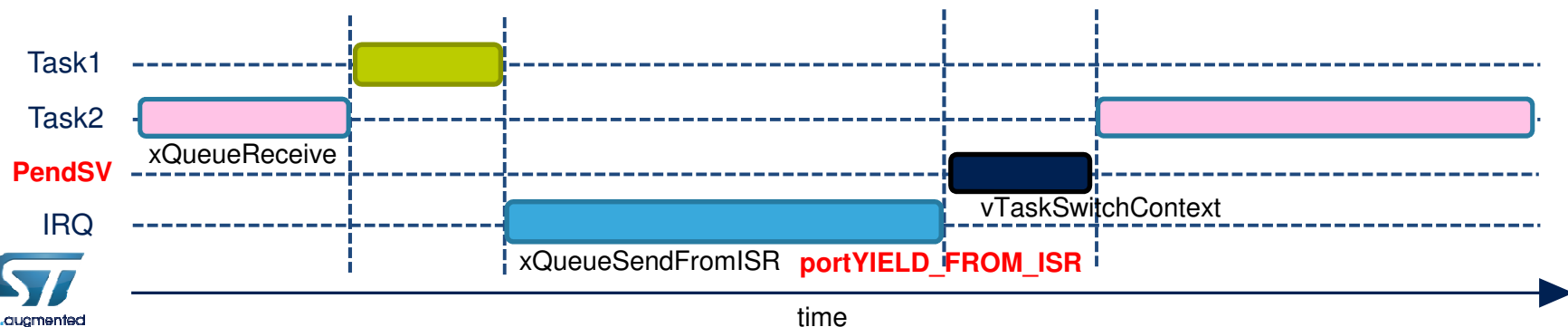
- configKERNEL\_INTERRUPT\_PRIORITY
  - CubeMX 에서 LIBRARY\_LOWEST\_INTERRUPT\_PRIORITY 의 값 (디폴트 15)
  - FreeRTOS 커널 자체의 인터럽트 priority 로서 SysTick 타이머 인터럽트와 PendSV 인터럽트의 priority 값으로 설정된다 (SysTick 타이머 주기마다 FreeRTOS 의 핵심 기능인 태스크 스케줄러 함수 vTaskSwitchContext() 가 호출된다)
- configMAX\_SYSCALL\_INTERRUPT\_PRIORITY
  - CubeMX 에서 LIBRARY\_MAX\_SYSCALL\_INTERRUPT\_PRIORITY 의 값 (디폴트 5)
  - FreeRTOS 의 API 함수를 호출하지 않는 인터럽트는 해당 상수값과 무관하게 어떠한 priority 숫자를 설정할 수 있다
  - FreeRTOS 의 API 함수를 호출하는 인터럽트는 해당 상수값보다 같거나 낮은 priority 숫자를 설정해야 한다 (Cortex-M4 는 priority 숫자가 반대이기 때문에 예를 들어 priority 가 낮은 UART 인터럽트를 디폴트 5보다 큰 6 으로 설정)
  - 해당 상수보다 낮은 priority 인터럽트는 critical section 에서 disable (masking) 된다

- portYIELD\_FROM\_ISR (xHigherPriorityTaskWoken)

- 인터럽트 핸들러 처리 이후 태스크 스케줄링을 바로 해야되는 경우 해당 함수를 호출한다
- 예를 들어 인터럽트 핸들러에서 portYIELD\_FROM\_ISR 을 사용하지 않을 경우, Task2 가 xQueueReceive 로 blocked 상태로 되고나서 Task1 이 처리되는 도중 IRQ 인터럽트가 발생하고, 인터럽트 핸들러 내부에서 xQueueSendFromISR 로 Task2 를 unblock 시키는 경우 Task1 로 복귀후 다음번 SysTick 으로 인한 태스크 스위칭시 Task2 로 전환된다



- 인터럽트 핸들러에서 portYIELD\_FROM\_ISR 를 사용하면 아래와 같이 Task2 로 처리순서가 바로 변경된다



- 인터럽트 핸들러는 최대한 짧은 시간 동안 peripheral 과 time critical 한 사용자 처리만 하고 실제 데이터 전후 처리는 태스크로 위임하는게 바람직하다

```
QueueHandle_t xBinarySem;
```

```
void one_sec_interrupt(void)
{
    BaseType_t higher_task_woken=pdFALSE;

    xSemaphoreGiveFromISR(
        xBinarySem,
        &higher_task_woken);

    portYIELD_FROM_ISR(higher_task_woken);
}
```

```
void Task1(void const * argument)
{
    for(;;){
        if(xSemaphoreTake(xBinarySem,2000)==pdPASS){
            //do something user logic
        }
        vTaskDelay(1000);
    }
}

void Task2(void const * argument)
{
    for(;;){
        vTaskDelay(100);
    }
}

void main(void)
{
    ...
    xBinarySem = xSemaphoreCreateBinary();
    xTaskCreate(Task1,"Task1", 1000, NULL, 1, NULL);
    xTaskCreate(Task2,"Task2", 1000, NULL, 2, NULL);
    ...
}
```

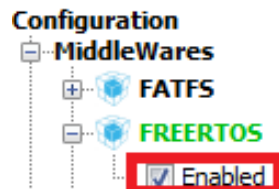
- FreeRTOS 는 전역 변수나 코드와 같은 공유 자원을 보호하기 위해 2가지 방법을 사용한다
- 인터럽트 끄기
  - taskENTER\_CRITICAL() 함수와 taskEXIT\_CRITICAL() 함수 사이에 공유 자원을 위치 시킨다
  - STM32L4 (Cortex-M4) 포팅은 taskENTER\_CRITICAL() 함수의 구현을 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 보다 priority 가 낮은 인터럽트를 BASEPRI 레지스터를 통해서 masking 한다
  - configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 보다 priority 가 높은 인터럽트로부터는 공유 자원을 보호할수 없다
- 스케줄러 끄기
  - vTaskSuspendAll() 함수와 xTaskResumeAll() 함수 사이에 공유 자원을 위치 시킨다
  - 다른 task 에서 해당 task 를 pre-empt 하는 것을 막을수는 있으나 인터럽트에서 pre-empt 하는 것은 막을수 없다



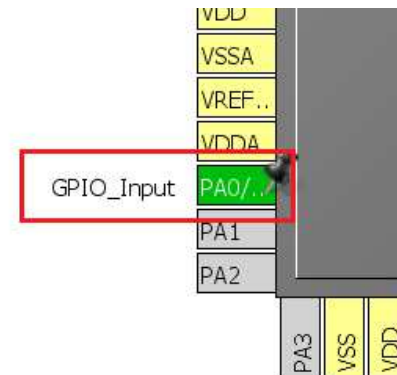
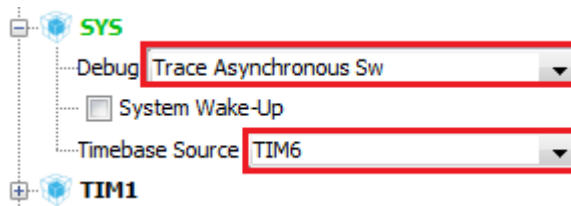
# Interrupt & Critical Section 실습

9

- STM32F429I-DISCO 보드에 SWO 핀과 PB3 핀이 케이블로 연결되어 있는지 확인
- CubeMX 를 실행하고 새로운 프로젝트를 생성한다
- New Project -> STM32F429ZI
- Pinout -> Middlewares -> FREERTOS 활성화



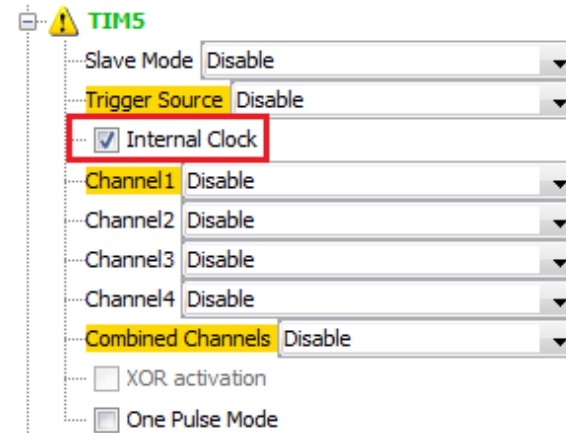
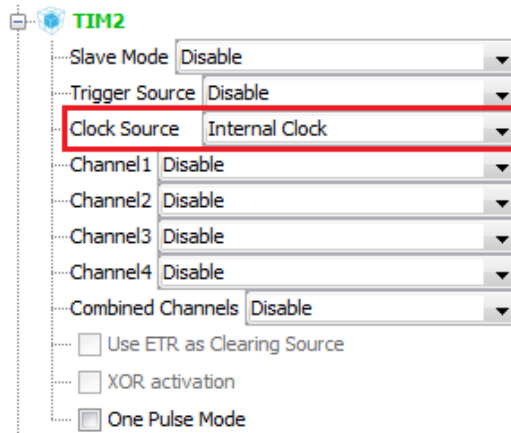
- Pinout -> Peripherals -> SYS 에서 Debug 와 Timebase Source 설정
  - Timebase Source 설정 시 interrupt timer로 사용할 TIM2, TIM5 보다 우선 순위가 낮은 Timer로 설정.
- PA0 번핀 (User Button) 을 입력핀으로 설정



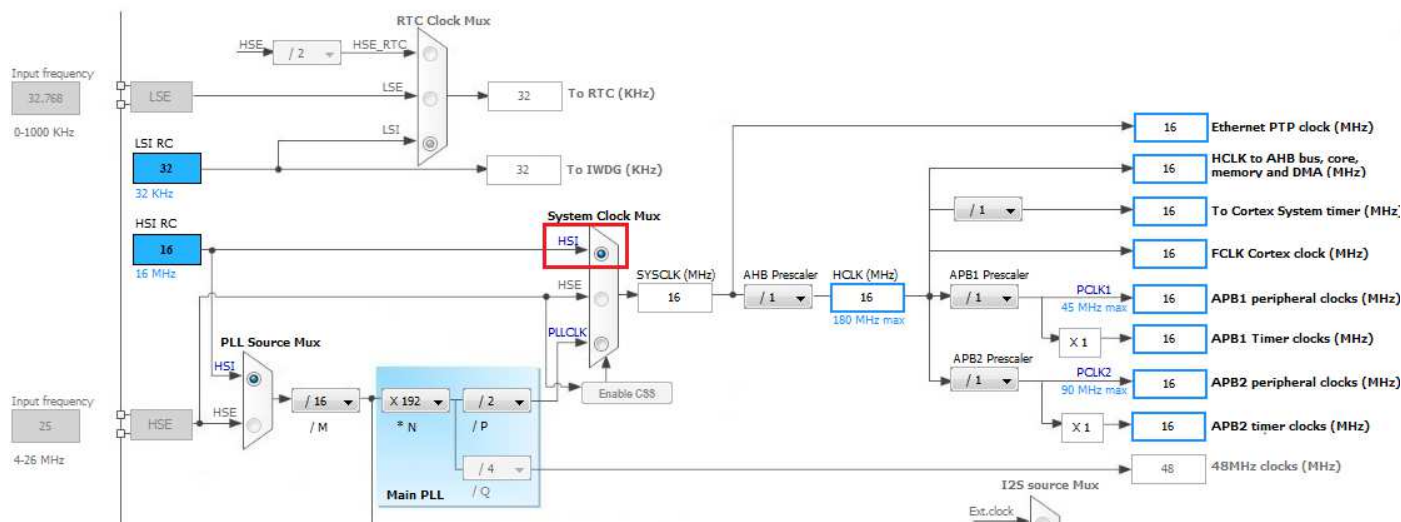
# Interrupt & Critical Section 실습

10

- Pinout -> Peripherals -> TIM2, TIM5 (32bit 타이머) 활성화



- Clock configuration -> HCLK 를 16 MHz 로 설정



# Interrupt & Critical Section 실습

11

- Configuration -> TIM2 와 TIM5 를 1초 인터럽트 주기로 설정
- TIM2 와 TIM5 은 APB1 (PCLK1) 을 클럭 소스로 사용하므로 앞에서 설정한 16 MHz 를 16 분주 (Prescaler+1) 해서 1 MHz 를 만든후 up 또는 down 카운터로 reload 값 1,000,000 을 설정한다

TIM2 Configuration

Parameter Settings User Constants NVIC Settings DMA Settings

Configure the below parameters :

Search : Search (Ctrl+F)

Counter Settings

Prescaler (PSC - 16 bits value)	15
Counter Mode	Up
Counter Period (AutoReload Register - 32 bits value)	1000000
Internal Clock Division (CKD)	No Division

Trigger Output (TRGO) Parameters

Master/Slave Mode	Disable (no output)
Trigger Event Selection TRGO	Reset (UG bit)

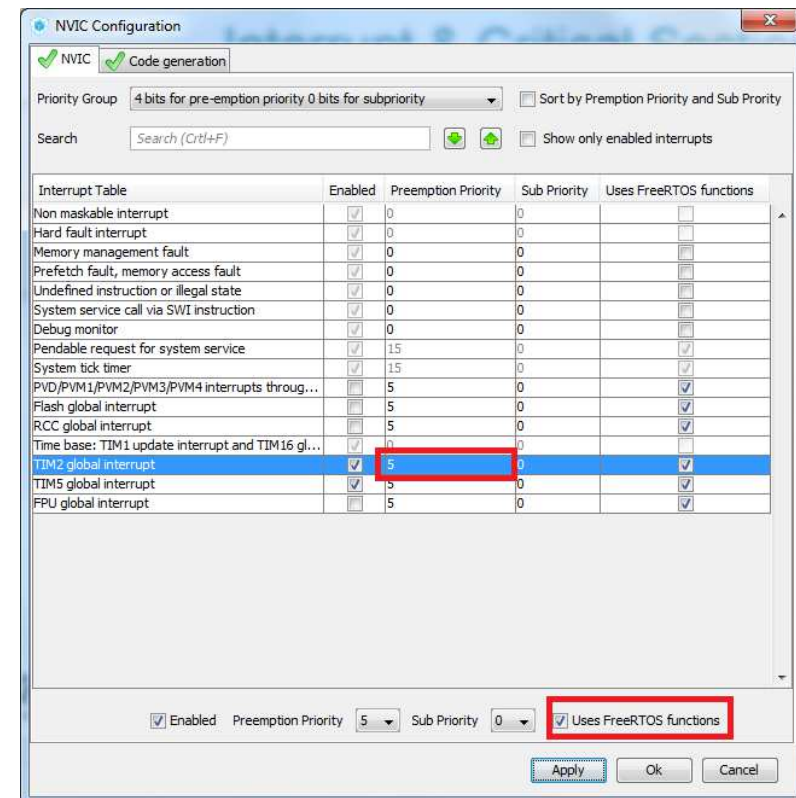
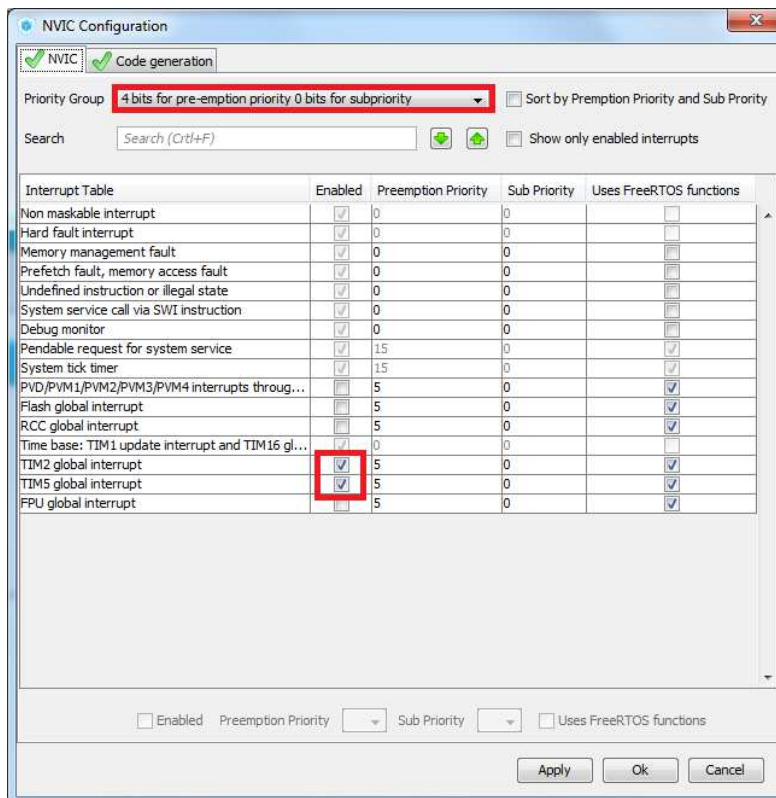
Restore Default

0x4000 3C00 - 0x4000 3FFF	SPI3 / I2S3	APB1	Section 28.5.10: SPI register map on page 918
0x4000 3800 - 0x4000 3BFF	SPI2 / I2S2		
0x4000 3400 - 0x4000 37FF	I2S2ext	APB1	
0x4000 3000 - 0x4000 33FF	IWDG		Section 21.4.5: IWDG register map on page 703
0x4000 2C00 - 0x4000 2FFF	WWDG		Section 22.6.4: WWDG register map on page 710
0x4000 2800 - 0x4000 2BFF	RTC & BKP Registers		Section 26.6.21: RTC register map on page 826
0x4000 2000 - 0x4000 23FF	TIM14		Section 19.5.12: TIM10/11/13/14 register map on page 686
0x4000 1C00 - 0x4000 1FFF	TIM13		Section 19.4.13: TIM9/12 register map on page 675
0x4000 1800 - 0x4000 1BFF	TIM12		Section 20.4.9: TIM6&TIM7 register map on page 698
0x4000 1400 - 0x4000 17FF	TIM7		
0x4000 1000 - 0x4000 13FF	TIM6		
0x4000 0C00 - 0x4000 0FFF	TIM5		Section 18.4.21: TIMx register map on page 639
0x4000 0800 - 0x4000 0BFF	TIM4		
0x4000 0400 - 0x4000 07FF	TIM3		
0x4000 0000 - 0x4000 03FF	TIM2		

# Interrupt & Critical Section 실습

12

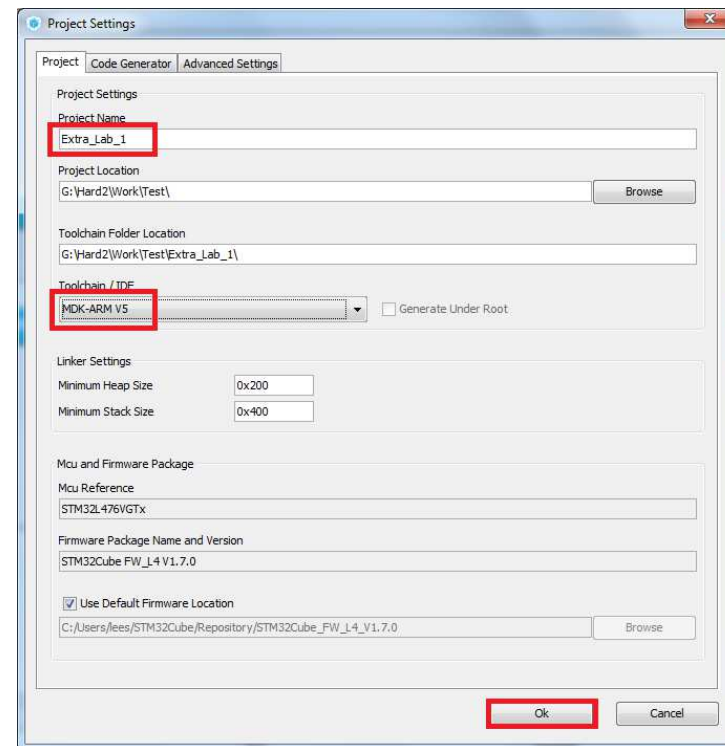
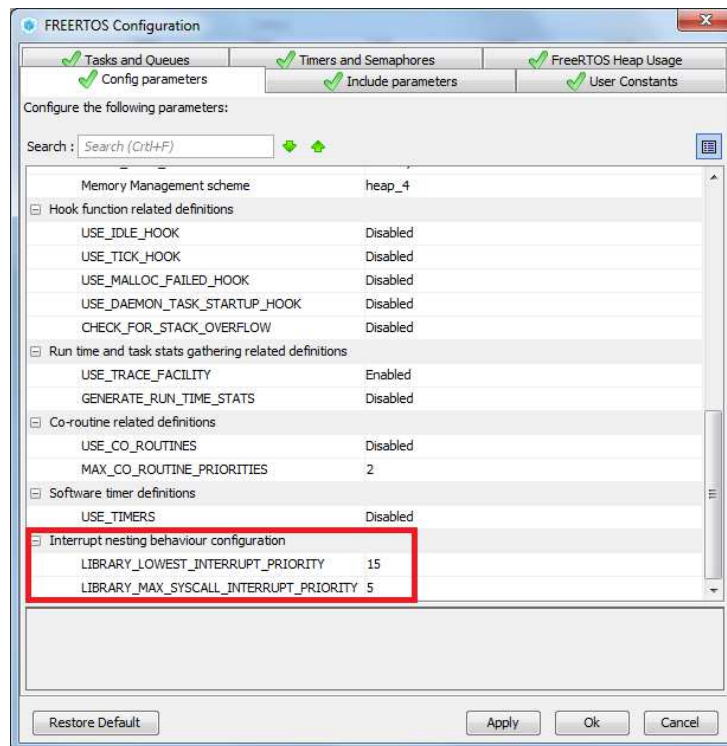
- Configuration -> NVIC 에서 TIM2 와 TIM5 의 인터럽트를 enable 한다
- CubeMX 툴에서는 FreeRTOS 를 enable 하면 pre-emptive priority 를 최대로 사용하는 4 bit for pre-emptive priority, 0 bit for subpriority 로 고정된다
- Uses FreeRTOS functions 체크박스에 따라서 TIM2, TIM5 에 설정할수 있는 Preemptive Priority 범위가 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 이상 또는 미만으로 바뀌는 것을 확인할 수 있다
- TIM2 와 TIM5 를 Uses FreeRTOS functions 를 체크하고 Priority 를 5로 설정한다



# Interrupt & Critical Section 실습

13

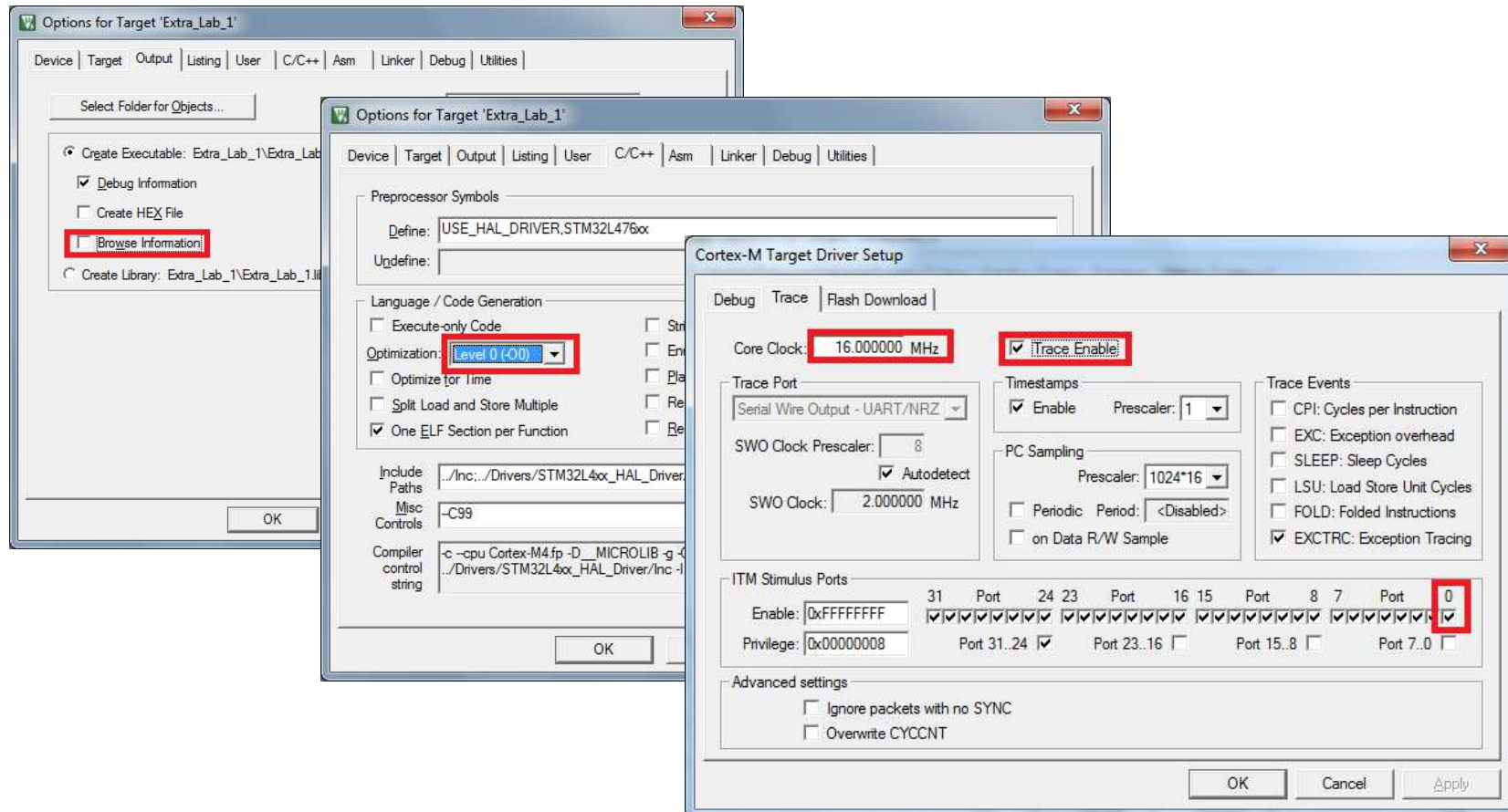
- Configuration -> FREERTOS 에서 lowest priority 와 max syscall priority 를 확인할수 있다
- Project -> Settings 에서 project 이름과 toolchain 을 선택한다
- Project -> Generate Code 를 선택하고 프로젝트를 연다



# Interrupt & Critical Section 실습

14

- 빠른 빌드를 위해 Project -> Option for Target -> Output -> Browse Information 을 해제 한다
- 디버깅 편의를 위해서 Project -> Option for Target -> C/C++ -> Optimization 을 0 으로 낮춘다
- printf 출력을 SWO 로 보기 위해서 Project -> Option for Target -> Debug -> ST-Link Debugger -> Settings -> Trace 에서 Core clock 속도를 맞춰주고 Trace enable 옵션과 ITM 0번 포트가 체크되어 있는지 확인한다





# Interrupt & Critical Section 실습

15

- 앞장에서 사용한 디버그 프린트문 코드를 추가한다

```
/* USER CODE BEGIN Includes */
#include <stdio.h>
struct __FILE { int handle; /* Add whatever is needed */ };
int fputc(int ch, FILE *f) {
    for(uint32_t delay=0;delay<100;delay++){
        ITM_SendChar(ch); //send method for SWV
    }
    return(ch);
}
/* USER CODE END Includes */
```

- CubeMX 는 타이머 설정에 대한 코드를 자동 생성해 주지만 실제 시작 함수는 사용자가 원하는 시점과 장소에 위치 시켜야 한다

```
/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start_IT(&htim2);
HAL_TIM_Base_Start_IT(&htim5);
/* USER CODE END 2 */
```

# Interrupt & Critical Section 실습

16

- 인터럽트 핸들러 내부에서 printf 출력은 절대로 권장하지 않지만 쉽고 빠른 실습을 위해 타이머2, 타이머5의 인터럽트가 수행되는 1초마다 printf 문을 출력하는 코드를 추가한다

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    /* USER CODE BEGIN Callback 0 */

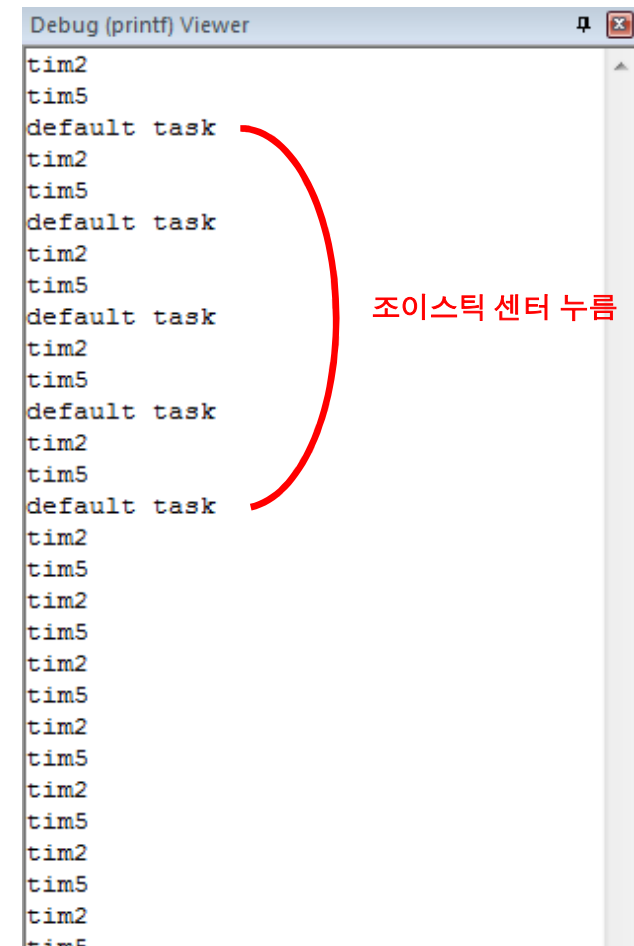
    /* USER CODE END Callback 0 */
    if (htim->Instance == TIM1) {
        HAL_IncTick();
    }
    /* USER CODE BEGIN Callback 1 */
    if (htim->Instance == TIM2) {
        printf("tim2\n");
    }

    if (htim->Instance == TIM5) {
        printf("tim5\n");
    }
    /* USER CODE END Callback 1 */
}
```



## 17

- ```
void StartDefaultTask(void const * argument)
{
    /* USER CODE BEGIN 5 */
    /* Infinite loop */
    for(;;)
    {
        //taskENTER_CRITICAL();
        while(HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0)==SET){};
        //taskEXIT_CRITICAL();
        printf("default task\n");
        osDelay(1000);
    }
    /* USER CODE END 5 */
}
```

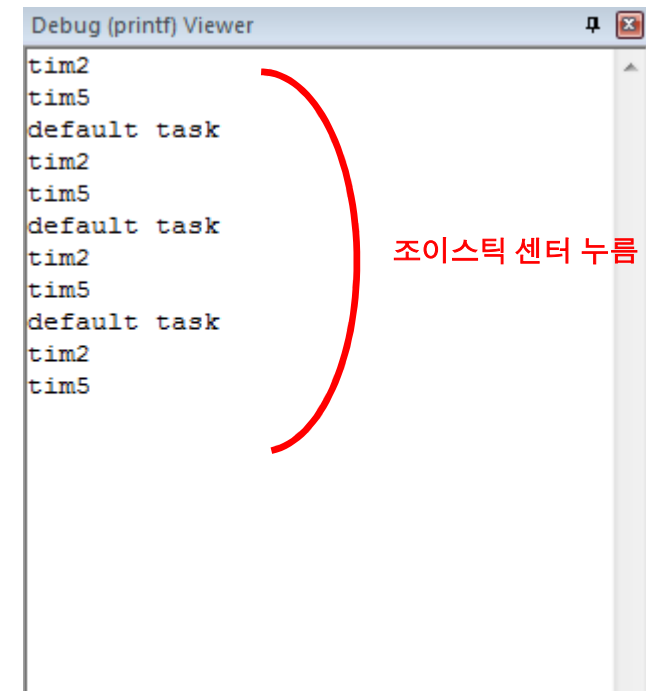


# Interrupt & Critical Section 실습

18

- 아래 코드에서 taskENTER\_CRITICAL() 과 taskEXIT\_CRITICAL() 의 주석을 해제하고 PA0 (조이스틱 센터 버튼) 을 누르면 View -> Serial Windows -> Debug Viewer 의 출력이 어떻게 바뀌는지 확인한다

```
void StartDefaultTask(void const * argument)
{
    /* USER CODE BEGIN 5 */
    /* Infinite loop */
    for(;;)
    {
        taskENTER_CRITICAL();
        while(HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0)==SET){};
        taskEXIT_CRITICAL();
        printf("default task\n");
        osDelay(1000);
    }
    /* USER CODE END 5 */
}
```



# Interrupt & Critical Section 실습

19

- HAL\_TIM\_Base\_MspInit 함수에서 타이머 5 의 인터럽트 priority 를 하나 작은 값 (높은 인터럽트 priority) 으로 변경하고 위의 테스트를 다시 수행했을때 PA0 (조이스틱 센터 버튼) 을 누르면 View -> Serial Windows -> Debug Viewer 의 출력이 어떻게 바뀌는지 확인한다

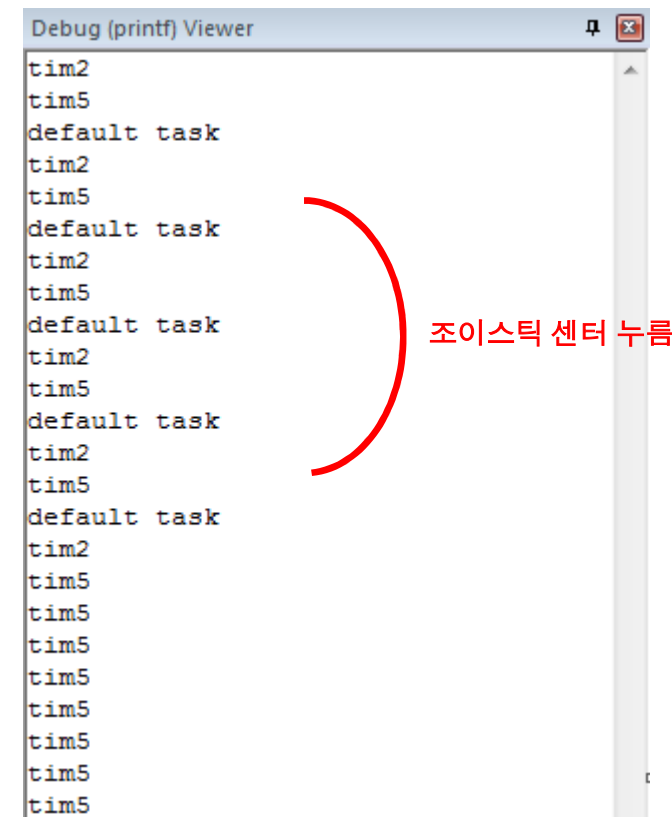
```
else if(htim_base->Instance==TIM5)
{
    /* USER CODE BEGIN TIM5_MspInit 0 */

    /* USER CODE END TIM5_MspInit 0 */
    /* Peripheral clock enable */
    __HAL_RCC_TIM5_CLK_ENABLE();
    /* Peripheral interrupt init */

    //HAL_NVIC_SetPriority(TIM5_IRQn, 5, 0);
    HAL_NVIC_SetPriority(TIM5_IRQn, 4, 0);

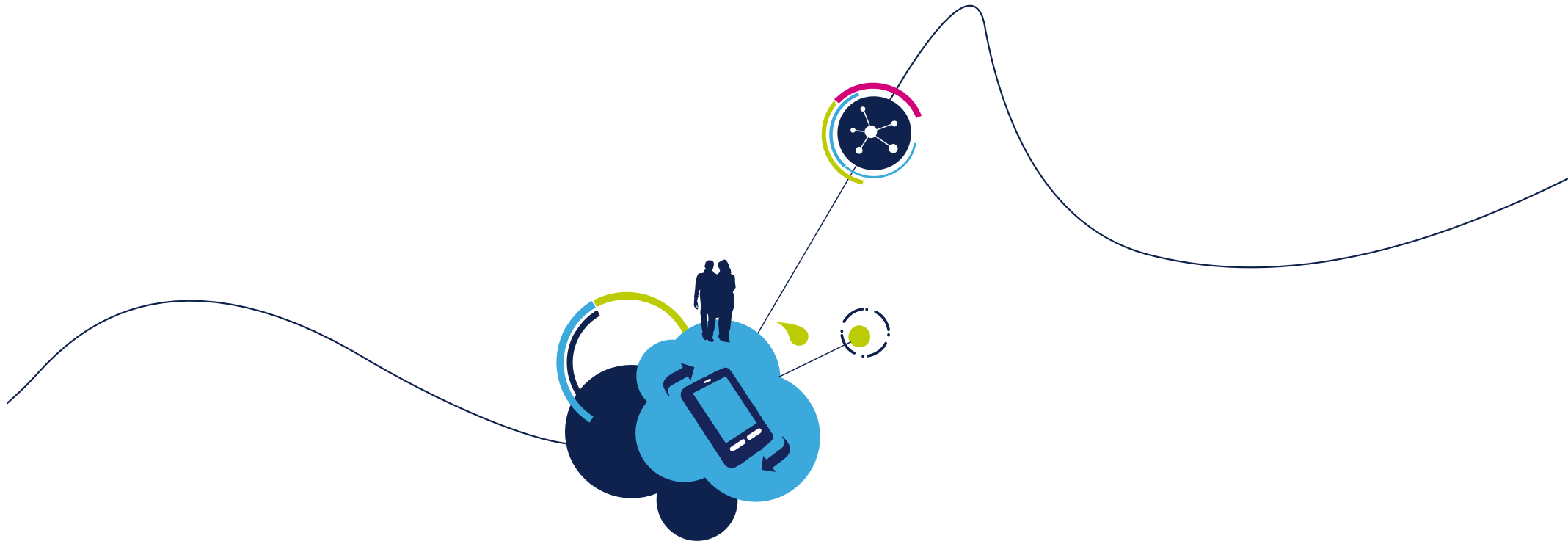
    HAL_NVIC_EnableIRQ(TIM5_IRQn);
    /* USER CODE BEGIN TIM5_MspInit 1 */

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



```
Debug (printf) Viewer
tim2
tim5
default task
tim2
tim5
default task
tim2
tim5
default task
tim2
tim5
default task
tim2
tim5
tim5
tim5
tim5
tim5
tim5
tim5
tim5
```

조이스틱 센터 누름



# FreeRTOS Event Group

- Semaphore 나 Queue 와 다르게 하나 또는 여러개의 이벤트를 조합해서 전달하고 기다리는 동기화 방법으로 사용된다
- configUSE\_16\_BIT\_TICKS 가 1 이면 Event Group (EventGroupHandle\_t) 은 16 비트 크기의 데이터형이며 하위 8 비트만 Event Flag 로 사용된다
- configUSE\_16\_BIT\_TICKS 가 0 이면 Event Group (EventGroupHandle\_t) 은 32 비트 크기의 데이터형이며 하위 24 비트만 Event Flag 로 사용된다
- Event Group 내의 한 비트를 Event Flag 라고 부르며 1 또는 0 으로 이벤트 발생 여부를 나타낸다
- 인터럽트 핸들러에서 사용하는 xEventGroupSetBitsFromISR 와 xEventGroupClearBitsFromISR 함수는 Set 또는 Clear 할 Bit (List\_t) 의 수가 가변이기 때문에 인터럽트 핸들러에서 non-deterministic 딜레이가 발생하지 않도록 flag 세팅만 하고 실제 동작은 FreeRTOS 커널 태스크로 위임하도록 구현되어 있다. 해당 API 를 사용하려면 configUSE\_TIMERS, configUSE\_TRACE\_FACILITY , INCLUDE\_xTimerPendFunctionCall 도 같이 세팅되어야 한다

- 제공되는 API 는 아래와 같다

## Event Groups and Event Bits API Functions

- **xEventGroupCreate**
- **xEventGroupCreateStatic**
- **xEventGroupWaitBits**
- **xEventGroupSetBits**
- **xEventGroupSetBitsFromISR**
- **xEventGroupClearBits**
- **xEventGroupClearBitsFromISR**
- **xEventGroupGetBits**
- **xEventGroupGetBitsFromISR**
- **xEventGroupSync**
- **vEventGroupDelete**

- 아래는 Task1, Task2 에서 Event Flag 1 과 2를 각각 세팅하고 Task3 은 Event Flag 1 과 2 가 모두 함께 세팅되는 것을 2초 동안 기다리는 예제이다

```
EventGroupHandle_t evtGrp;
uint32_t evt_flag_1 = 0x01; //bit 0
uint32_t evt_flag_2 = 0x04; //bit 2

void Task1(void const * argument)
{
    for(;;){
        xEventGroupSetBits(evtGrp, evt_flag_1);
        vTaskDelay(1000);
    }
}

void Task2(void const * argument)
{
    for(;;){
        xEventGroupSetBits(evtGrp, evt_flag_2);
        vTaskDelay(1000);
    }
}
```

```
void Task3(void const * argument)
{
    uint32_t result;
    for(;;){
        result = xEventGroupWaitBits(
            evtGrp,
            (evt_flag_1 | evt_flag_2),
            pdTRUE, //xClearOnExit
            pdTRUE, //xWaitForAllBits
            2000); //xTicksToWait
        if(result & (evt_flag_1 | evt_flag_2) ==
            (evt_flag_1 | evt_flag_2)){
            //flag_1 and flag_2 are all set
        }
        else{
            if(result & evt_flag_1){//flag_1 is set}
            else if(result & evt_flag_2){//flag_2 is set}
            else {//none set}
        }
    }
}

void main(void)
{
    evtGrp = xEventGroupCreate();
}
```

# Event Group 실습

24

- 앞장에서 사용한 예제를 다음과 같이 타이머 5 시작만 주석처리 한다

```
/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start_IT(&htim2);
//HAL_TIM_Base_Start_IT(&htim5);
/* USER CODE END 2 */
```

- Event Group, Event Flag 그리고 2개의 태스크 핸들을 선언하고

```
/* USER CODE BEGIN PV */
/* Private variables -----*/
EventGroupHandle_t evtGrp;
uint32_t evt_flag_1 = 0x01; //bit 0
uint32_t evt_flag_2 = 0x02; //bit 1
static TaskHandle_t xTask1_Handle = NULL;
static TaskHandle_t xTask2_Handle = NULL;
#define configTASKS_PRIORITY (tskIDLE_PRIORITY)
/* USER CODE END PV */
```



- main 함수에서 Event Group 을 생성하고 태스크 2개를 생성한다

```
osThreadDef(defaultTask, StartDefaultTask, osPriorityNormal, 0, 128);  
defaultTaskHandle = osThreadCreate(osThread(defaultTask), NULL);
```

```
/* USER CODE BEGIN RTOS_THREADS */
```

```
evtGrp = xEventGroupCreate();
```

```
BaseType_t xReturn;
```

```
xReturn = xTaskCreate(Task1, "Task1", configMINIMAL_STACK_SIZE, NULL,  
                    ( UBaseType_t ) configTASKS_PRIORITY,  
                    &xTask1_Handle );
```

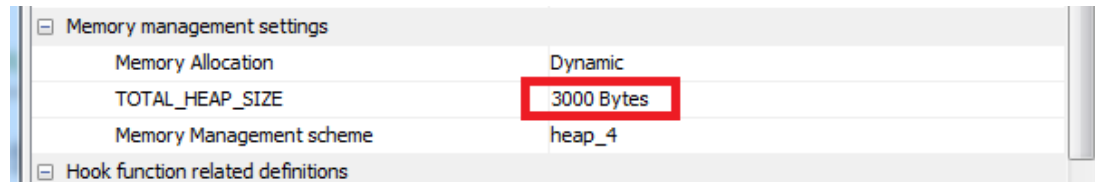
```
xReturn = xTaskCreate(Task2, "Task2", configMINIMAL_STACK_SIZE, NULL,  
                    ( UBaseType_t ) configTASKS_PRIORITY,  
                    &xTask2_Handle );
```

```
/* USER CODE END RTOS_THREADS */
```

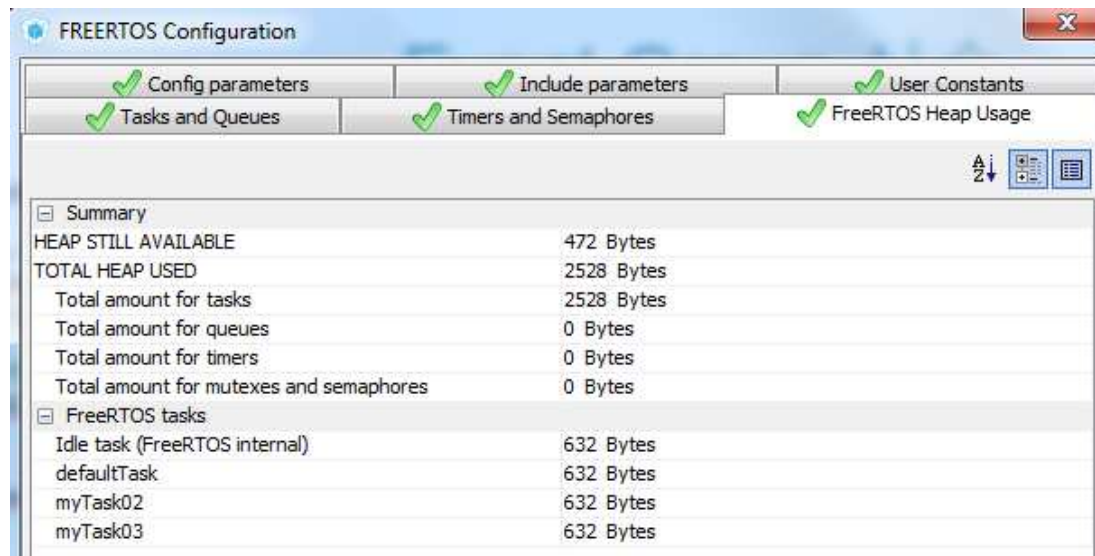
# Event Group 실습

26

- 태스크를 여러개 생성하는 경우 HEAP 사이즈가 충분한지 유의한다



- CubeMX 를 사용해서 태스크를 생성할 경우 위의 TOTAL\_HEAP\_SIZE 대비 현재 사용중인 HEAP 사이즈와 남아있는 HEAP 사이즈를 볼수 있다



# Event Group 실습

27

- 태스크1 은 1초 마다 Event Group 의 Flag 1 을 세팅하도록 코드를 추가하고
- 태스크2 에 3초 마다 Event Group 의 Flag 2 를 세팅하도록 코드를 추가한다

```
/* USER CODE BEGIN 0 */
static void Task1( void *pvParameters )
{
    for(;;){
        xEventGroupSetBits(evtGrp, evt_flag_1);
        vTaskDelay(1000);
    }
}

static void Task2( void *pvParameters )
{
    for(;;){
        xEventGroupSetBits(evtGrp, evt_flag_2);
        vTaskDelay(3000);
    }
}
/* USER CODE END 0 */
```

# Event Group 실습

28

- default 태스크는 Event Group 의 Flag 1 과 2가 동시에 세팅되는것을 5초 동안 기다리고 결과를 printf로 출력 한다 (USER CODE BEGIN 사이 코드 위치)

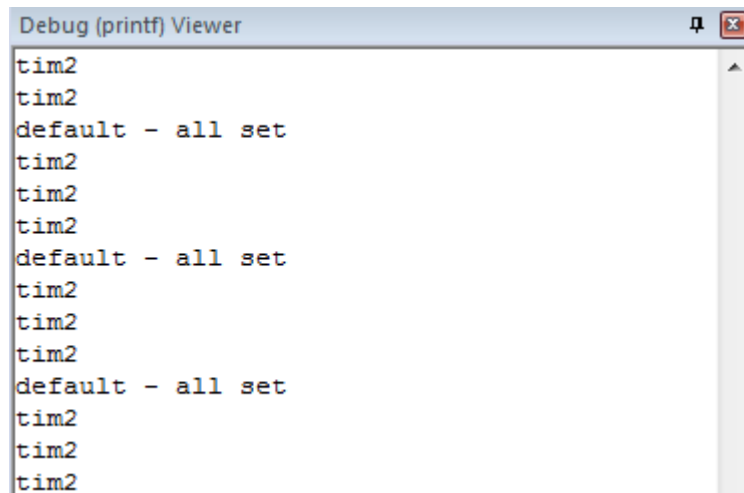
```
void StartDefaultTask(void const * argument)
{
    /* USER CODE BEGIN 5 */
    /* Infinite loop */
    uint32_t result;

    for(;;){
        result = xEventGroupWaitBits(
            evtGrp,
            (evt_flag_1 | evt_flag_2),
            pdTRUE, //xClearOnExit
            pdTRUE, //xWaitForAllBits
            5000); //xTicksToWait
        if((result&(evt_flag_1 | evt_flag_2)) == (evt_flag_1 | evt_flag_2)){
            printf("default - all set\n");
        }
        else{
            if(result & evt_flag_1){printf("default - flag1 set\n");}
            else if(result & evt_flag_2){printf("default - flag2 set\n");}
            else {printf("default - none set\n");}
        }
        vTaskDelay(500);
    }
    /* USER CODE END 5 */
}
```

# Event Group 실습

29

- 즉, 태스크1 은 1초 간격으로 Flag 1 을 세팅하고
- 태스크2는 3 초 간격으로 Flag 2 를 세팅하도록 테스트 했을때
- 앞장에서 실습했던 타이머2는 1초 간격으로 printf 출력이 이루어지고 있으며 default 태스크는 Flag 1 과 2 가 모두 세팅되는 3초마다 printf 출력이 되는것을 확인할 수 있다



```
Debug (printf) Viewer
tim2
tim2
default - all set
tim2
tim2
tim2
default - all set
tim2
tim2
tim2
default - all set
tim2
tim2
tim2
```

- xEventGroupWaitBits 함수의 xClearOnExit 또는 xWaitForAllBits 인자를 pdFALSE 로 바꾸면 결과가 어떻게 바뀌는지 실습해본다

# Event Group 실습

30

- xEventGroupWaitBits 함수의 xClearOnExit 또는 xWaitForAllBits 인자를 모두 pdTRUE 로 복구하고 타이머 2 를 아래와 같이 수정해서 다시 실습해본다

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    /* USER CODE BEGIN Callback 0 */

    /* USER CODE END Callback 0 */
    if (htim->Instance == TIM1) {
        HAL_IncTick();
    }
    /* USER CODE BEGIN Callback 1 */
    BaseType_t higher_task_woken=pdFALSE;

    if (htim->Instance == TIM2) {
        if (evtGrp!=NULL){
            xEventGroupSetBitsFromISR(evtGrp, evt_flag_2,&higher_task_woken);
            portYIELD_FROM_ISR(higher_task_woken);
        }
        printf("tim2\n");
    }

    if (htim->Instance == TIM5) {
        printf("tim5\n");
    }
    /* USER CODE END Callback 1 */
}
```

# Event Group 실습

31

- xEventGroupSetBitsFromISR 을 사용하려면 CubeMX 에서 타이머와

|                            |           |
|----------------------------|-----------|
| Software timer definitions |           |
| USE_TIMERS                 | Enabled   |
| TIMER_TASK_PRIORITY        | 2         |
| TIMER_QUEUE_LENGTH         | 10        |
| TIMER_TASK_STACK_DEPTH     | 256 Words |

- 아래 두가지를 enable 시켜주고

|                             |          |
|-----------------------------|----------|
| uxTaskGetStackHighWaterMark | Disabled |
| xTaskGetCurrentTaskHandle   | Disabled |
| eTaskGetState               | Disabled |
| xEventGroupSetBitFromISR    | Enabled  |
| xTimerPendFunctionCall      | Enabled  |
| xTaskAbortDelay             | Disabled |
| xTaskGetHandle              | Disabled |

- HEAP 사이즈를 4000 Byte 로 늘려주고 다시 Generate Code 를 실행 한다

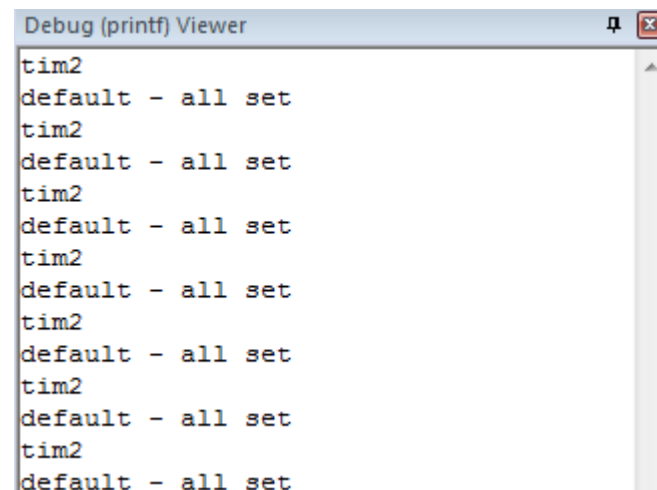
|                            |            |
|----------------------------|------------|
| Memory management settings |            |
| Memory Allocation          | Dynamic    |
| TOTAL_HEAP_SIZE            | 4000 Bytes |
| Memory Management scheme   | heap_4     |

- 14 페이지의 프로젝트 설정을 다시 해준다

# Event Group 실습

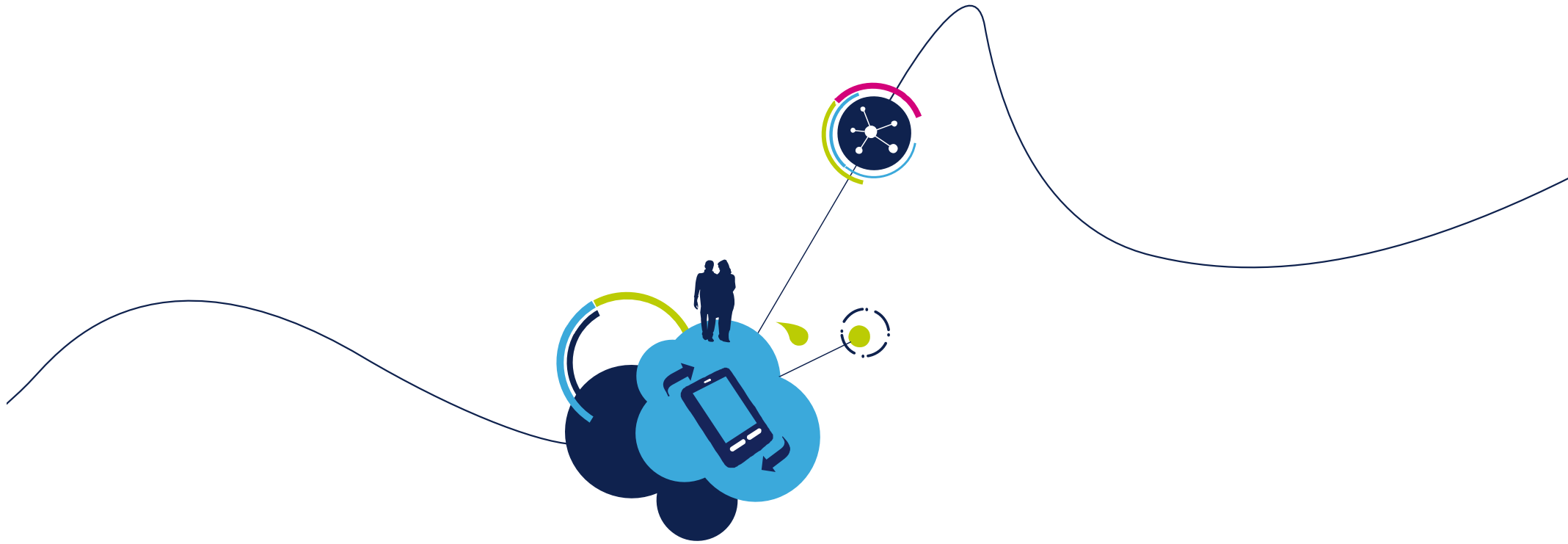
32

- 즉, 태스크1 은 1초 간격으로 Flag 1 을 세팅하고
- 태스크2는 3 초 간격으로 Flag 2 를 세팅하고
- 타이머2 는 1초 간격으로 Flag 2 를 세팅하도록 테스트 했을때
- default 태스크는 Flag 1 과 2 가 모두 세팅되는 1초마다 printf 출력이 되는것을 확인할 수 있다



```
Debug (printf) Viewer
tim2
default - all set
tim2
default - all set
tim2
default - all set
tim2
default - all set
tim2
default - all set
tim2
default - all set
tim2
default - all set
tim2
default - all set
```





# FreeRTOS Co-Routine

- 태스크는 태스크별로 stack 을 할당해서 사용하는 반면 Co-Routine 들은 하나의 stack 을 공유해서 사용한다
- 램 용량이 많이 부족한 시스템에서 태스크 대신에 Co-Routine 이 사용되며 태스크의 TCB (Task Control Block)보다 작은 사이즈의 CRCB (Co-Routine Control Block) 가 사용된다
- 태스크는 SysTick (PendSV) 을 통해서 스케줄링이 이루어지는 반면 Co-Routine 은 Idle 태스크 hook 에서 vCoRoutineSchedule() 를 호출하는 코드를 넣어 줌으로서 스케줄링이 이루어진다
- Idle 태스크는 Running(Ready) 상태의 태스크가 없을때만 수행되기 때문에 태스크와 Co-Routine 을 동시에 함께 사용할수 있지만 태스크의 우선순위가 항상 Co-Routine 보다 높다

- Co-Routine 의 priority 는 0 부터 configMAX\_CO\_ROUTINE\_PRIORITIES-1 값으로 설정하며 태스크 priority와 무관한 Co-Routine 들 사이의 priority 값으로 사용된다
- Co-Routine 은 함수 시작부분에 crSTART(), 마지막 부분에 crEND() 함수를 호출해야 하며 함수가 리턴 되면 안되기 때문에 무한 루프로 처리한다
- Co-Routine 용 API 함수를 사용해야 한다 (예를 들어, crDELAY(), crQUEUE\_SEND() 등)
- Stack 을 공유해서 사용하기 때문에 Co-Routine 함수 내부에는 전역변수 또는 static 변수만 사용해야 하며 Co-Routine 용 API 함수는 분기된 다른 함수에서 호출되면 안되고 Co-Routine 내부에서 직접 호출되어야 한다
- Switch case 문 안에서 crDELAY() 와 같은 blocking 함수를 사용하면 안된다
- Co-Routine 은 사용법에 제약사항을 유의해야 하므로 램 용량이 부족하지 않다면 사용을 권장하지 않는다

- 제공되는 API 는 아래와 같다

## Co-routine specific

[API]

### Modules

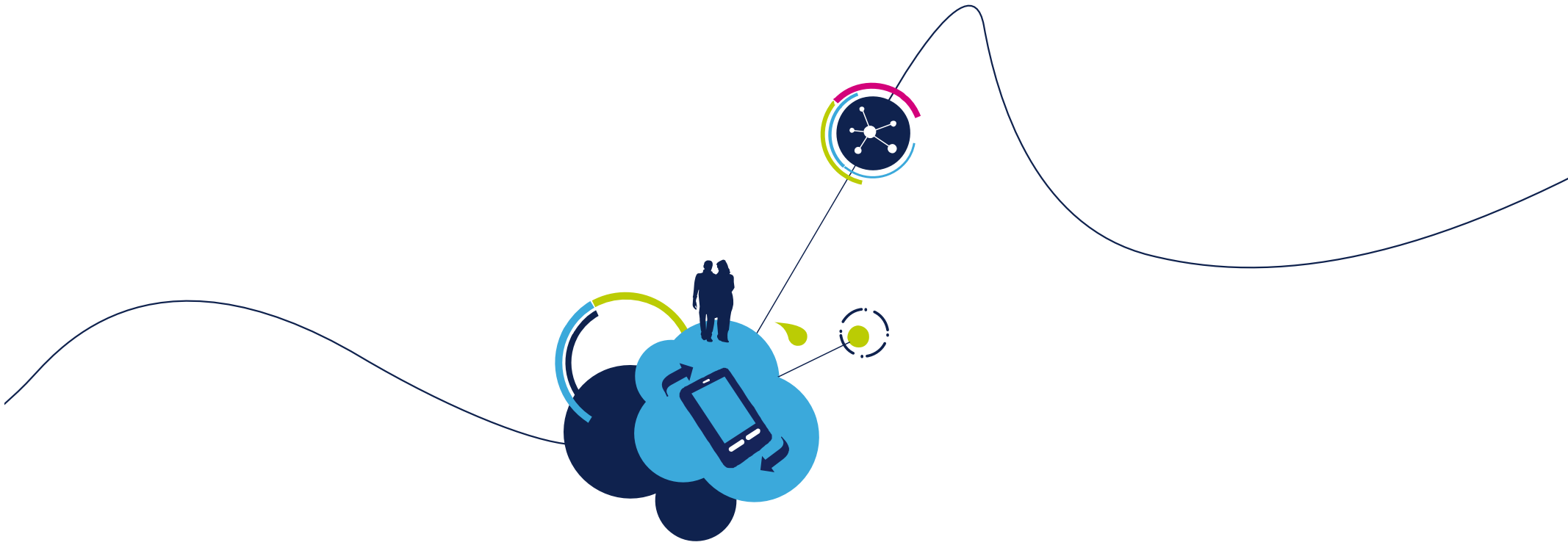
- CoRoutineHandle\_t
- xCoRoutineCreate
- crDELAY
- crQUEUE\_SEND
- crQUEUE\_RECEIVE
- crQUEUE\_SEND\_FROM\_ISR
- crQUEUE\_RECEIVE\_FROM\_ISR
- vCoRoutineSchedule

- Co-Routine 의 사용 예제는 아래와 같다

```
void CoRoutine1(xCoRoutineHandle xHandle, unsigned portBASE_TYPE uxIndex)
{
    static uint32_t no_local_variable = 0x11;
    crSTART( xHandle );
    for( ;; ){
        no_local_variable++;
        crDELAY( xHandle, 10 );
    }
    crEND();
}

void CoRoutine2(xCoRoutineHandle xHandle, unsigned portBASE_TYPE uxIndex)
{
    crSTART( xHandle );
    for( ;; ){
        crDELAY( xHandle, 20 );
    }
    crEND();
}

void main(void)
{
    ...
    xCoRoutineCreate( CoRoutine1, PRIORITY_0, 0 );
    xCoRoutineCreate( CoRoutine2, PRIORITY_1, 0 );
}
```



# FreeRTOS

## Task Notification

- 태스크 생성시 32bit 의 notification value 가 태스크마다 하나씩 할당된다. Notification value 를 통해서 태스크들 및 인터럽트 핸들러들 사이에 동기화 신호를 주거나 및 32bit 데이터 전송을 할수 있는 용도로 사용된다
- Queue, semaphore, event group 과 같이 송수신 중간에 객체를 사용해 전달하는 방법이 아닌 태스크로의 직접 전달 방법이기 때문에 binary semaphore 보다 속도와 램 사용량에 이점이 있다
- Queue, semaphore, event group 은 핸들을 가지고 있는 여러개의 태스크에서 수신을 할수 있지만 Task Notification 은 하나의 수신 태스크만 사용 가능하다
- Task Notification 은 인터럽트 (또는 태스크)에서 송신하고 태스크에서 수신이 가능하다
- Task Notification 은 태스크에서 송신하고 인터럽트에서 수신은 불가능하다
- Queue, semaphore, event group 의 송신은 blocked 상태에서 송신 완료를 기다릴수 있지만 Task Notification 의 송신은 단순히 overwrite 된다

- 제공되는 API 는 아래와 같다

## RTOS Task Notifications

[API]

### RTOS task notification API functions:

- `xTaskNotifyGive()`
- `vTaskNotifyGiveFromISR()`
- `ulTaskNotifyTake()`
- `xTaskNotify()`
- `xTaskNotifyAndQuery()`
- `xTaskNotifyAndQueryFromISR()`
- `xTaskNotifyFromISR()`
- `xTaskNotifyWait()`
- `xTaskNotifyStateClear()`



- Task Notification 을 binary semaphore 처럼 사용하는 예제는 아래와 같다

```
static TaskHandle_t task1_handle = NULL;
void Task1(void const * argument)
{
    uint32_t noti_value;
    for( ;; ){
        noti_value = ulTaskNotifyTake( pdTRUE, 3000 ); //xClearCountOnExit, xTicksToWait
        if(noti_value != 0){/*Something received*/}
        else {/*Nothing received*/}
    }
}

void one_sec_interrupt(void)
{
    BaseType_t higher_task_woken=pdFALSE;
    if(task1_handle != NULL){
        vTaskNotifyGiveFromISR( task1_handle, &higher_task_woken );
    }

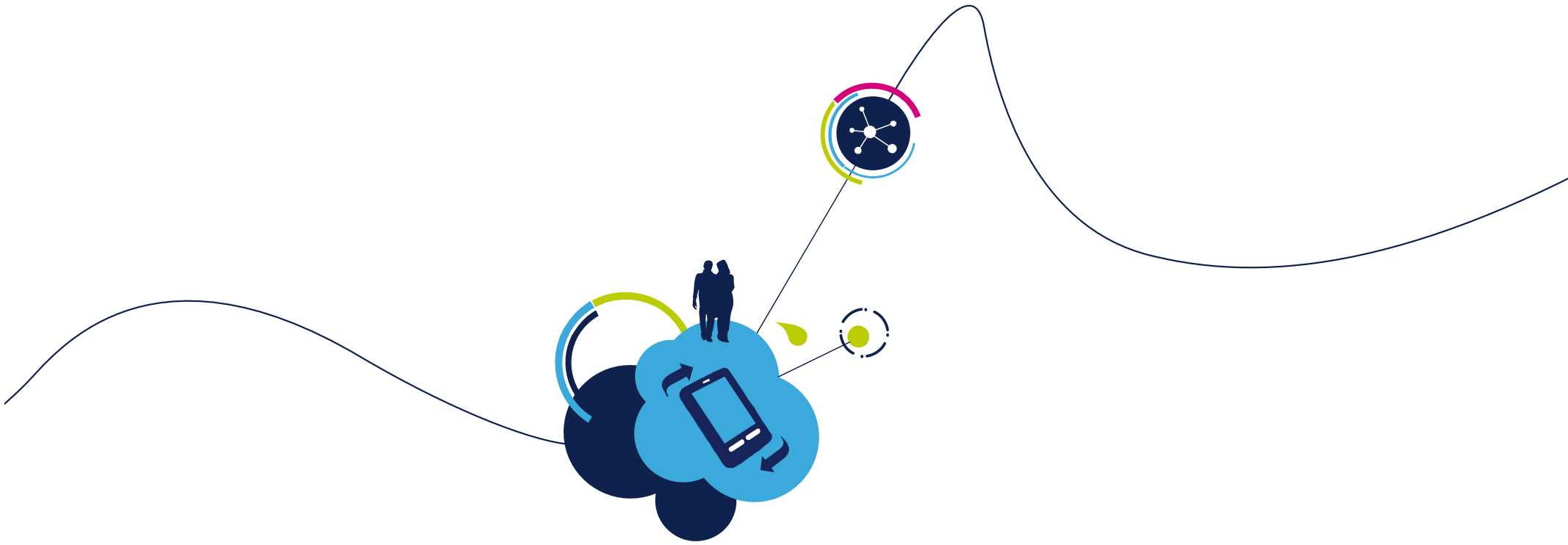
    portYIELD_FROM_ISR(higher_task_woken);
}

void main(void)
{
    ...
    task1_handle = xTaskCreate(Task1,"Task1", 1000, NULL, 1, NULL);
}
```

- Task Notification 을 동기화 및 데이터 전송으로 사용하는 예제는 아래와 같다

```
static TaskHandle_t task1_handle = NULL;
void Task1(void const * argument)
{
    uint32_t noti_value;
    for( ;; ){
        xTaskNotifyWait( 0x00,          /* Don't clear any notification bits on entry. */
                        0xffffffff, /* Reset the notification value to 0 on exit. */
                        &noti_value, /* Notified value pass out in noti_value. */
                        portMAX_DELAY ); /* Block indefinitely. */
        if(noti_value & 0x01){/*Bit0 is set*/}
        else {/*Something else*/}
    }
}

static uint32_t tick_count;
void one_sec_interrupt(void)
{
    BaseType_t higher_task_woken=pdFALSE;
    ++tick_count;
    if(task1_handle != NULL){
        xTaskNotifyFromISR( task1_handle, tick_count, eSetBits, &higher_task_woken );
    }
    xTaskNotify( task2_handle, 0, eNoAction );
    xTaskNotify( task3_handle, 0x50, eSetValueWithOverwrite );
    portYIELD_FROM_ISR(higher_task_woken);
}
```



# FreeRTOS

## Run Time Stack Checking

# Run Time Stack Checking

44

- 태스크에 할당된 stack 의 overflow 체크를 위해 아래 상수를 1 또는 2로 설정한다 (FreeRTOSConfig.h)
  - configCHECK\_FOR\_STACK\_OVERFLOW 가 1 인 경우, 스케줄러 vTaskSwitchContext() 함수가 호출될 때마다 TCB 에 있는 stack top 주소 변수와 현재 stack 주소 변수를 비교해서 stack overflow 를 검사한다
  - configCHECK\_FOR\_STACK\_OVERFLOW 가 2 인 경우, 태스크가 생성될때 stack 전체를 특정 패턴의 데이터 (0xa5) 로 채우고 스케줄러 vTaskSwitchContext() 함수가 호출될 때마다 현재 stack 주소에서 20 byte 를 읽어서 특정패턴(0xa5) 이 다른 데이터로 overwrite 되지 않았는지 검사한다
- Stack overflow 후킹 함수는 위의 상수가 1 또는 2로 설정된 경우, stack overflow 검사시 발생한 태스크의 핸들과 이름을 인자로 리턴한다. Stack overflow 가 심할 경우 잘못된 태스크 이름이 전달될 수도 있고 후킹 함수가 호출 조차 안될수 있기 때문에 신뢰도는 제한적이다
  - vApplicationStackOverflowHook( xTaskHandle \*pxTask, signed portCHAR \*pcTaskName )
- 태스크 생성 이후부터 기록된 남아 있는 최소 stack 크기 조회를 매뉴얼 하게 하려면 INCLUDE\_uxTaskGetStackHighWaterMark 를 enable 하고 아래 함수를 호출한다. 리턴값 1 은 1 word 를 나타낸다.

# Task Notification & Run Time Stack Checking 실습

45

- 앞장에서 사용한 예제의 타이머2 시작도 주석처리 한다

```
/* USER CODE BEGIN 2 */
//HAL_TIM_Base_Start_IT(&htim2);
//HAL_TIM_Base_Start_IT(&htim5);
/* USER CODE END 2 */
```

- FreeRTOSConfig.h 파일에 INCLUDE\_uxTaskGetStackHighWaterMark 를 1로 선언해 준다

```
116 /* Set the following definitions to 1 to include the API function, or zero
117 to exclude the API function. */
118 #define INCLUDE_vTaskPrioritySet 1
119 #define INCLUDE_uxTaskPriorityGet 1
120 #define INCLUDE_vTaskDelete 1
121 #define INCLUDE_vTaskCleanUpResources 0
122 #define INCLUDE_vTaskSuspend 1
123 #define INCLUDE_vTaskDelayUntil 0
124 #define INCLUDE_vTaskDelay 1
125 #define INCLUDE_xTaskGetSchedulerState 1
126 #define INCLUDE_uxTaskGetStackHighWaterMark 1
```

# Task Notification & Run Time Stack Checking 실습

46

- 태스크1 에서 default 태스크로 1초 간격으로 tick\_count 를 notification value 로 보내도록 코드를 아래와 같이 수정한다

```
void Task1(void const * argument)
{
    static uint32_t tick_count;
    for(;;){
        ++tick_count;
        if(defaultTaskHandle != NULL){
            xTaskNotify( defaultTaskHandle, tick_count, eSetValueWithOverwrite );
        }
        vTaskDelay(1000);
    }
}
```

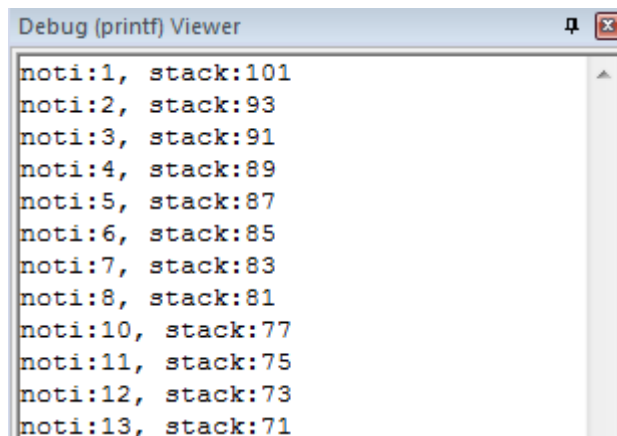
# Task Notification & Run Time Stack Checking 실습

47

- default 태스크를 아래와 같이 수정한다

```
void StartDefaultTask(void const * argument)
{
    uint32_t noti_value;
    for(;;){
        xTaskNotifyWait( 0x00,          /* Don't clear any notification bits on entry. */
                        0xffffffff, /* Reset the notification value to 0 on exit. */
                        &noti_value, /* Notified value pass out in noti_value. */
                        portMAX_DELAY ); /* Block indefinitely. */
        printf("noti:%d, stack:%d\n", noti_value, uxTaskGetStackHighWaterMark(defaultTaskHandle));
        StartDefaultTask(NULL);      //스택 overflow 를 만들기 위한 Recursive call
    }
}
```

- Recursive 함수와 지역변수에 의해 stack 이 줄어드는것을 확인할 수 있다



```
Debug (printf) Viewer
noti:1, stack:101
noti:2, stack:93
noti:3, stack:91
noti:4, stack:89
noti:5, stack:87
noti:6, stack:85
noti:7, stack:83
noti:8, stack:81
noti:10, stack:77
noti:11, stack:75
noti:12, stack:73
noti:13, stack:71
```

# Task Notification & Run Time Stack Checking 실습

48

- FreeRTOSConfig.h 파일에 configCHECK\_FOR\_STACK\_OVERFLOW 를 2로 선언해 준다

```
106 #define configMAX_TASK_NAME_LEN          ( 16 )
107 #define configUSE_TRACE_FACILITY          1
108 #define configUSE_16_BIT_TICKS            0
109 #define configUSE_MUTEXES                  1
110 #define configQUEUE_REGISTRY_SIZE          8
111 #define configCHECK_FOR_STACK_OVERFLOW     2
```

- Stack overflow 후킹 함수를 아래와 같이 추가해 준다

```
/* USER CODE BEGIN 0 */
void vApplicationStackOverflowHook( TaskHandle_t xTask, char *pcTaskName )
{
    printf("overflow: %s\n", pcTaskName);
}
```



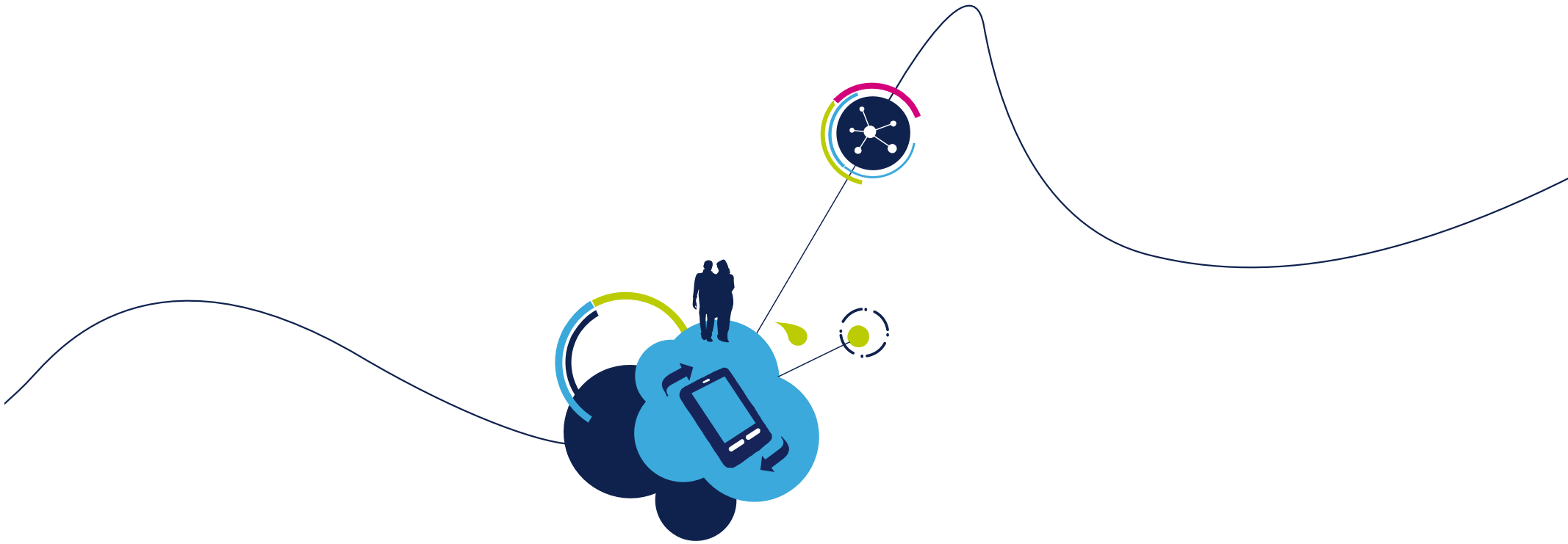
# Task Notification & Run Time Stack Checking 실습

49

- 아래와 같이 4 word 크기의 stack 이 남은 지점부터 특정패턴(0xa5) 20 byte 검사에서 stack overflow 가 감지되면서 후킹 함수의 printf 문이 호출되는 것을 확인 할 수 있다

```
noti:34, stack:18  
noti:35, stack:18  
noti:36, stack:16  
noti:37, stack:14  
noti:38, stack:12  
noti:39, stack:10  
noti:40, stack:8  
noti:41, stack:8  
noti:42, stack:4  
overflow: defaultTask  
noti:43, stack:2  
overflow: defaultTask  
noti:44, stack:2  
overflow: defaultTask  
noti:45, stack:0  
overflow: defaultTask  
noti:46, stack:1  
overflow: defaultTask  
noti:48, stack:0  
overflow: defaultTask  
noti:49, stack:0  
overflow: defaultTask
```





Thank you