AVR FreeRTOS : Interrupt Management

1. 이 장의 개요

Embedded Real Time 시스템은 주변 장치로부터 발생 하는 Event 에 실시간으로 응답 하여야 하는 응용 분야에 많이 이용 된다. 응용 분야에 따라서는 여러 개의 Interrupt Source로부터 발생 하는 Event를 실시간으로 처리 하여야 하고, 각각의 Interrupt 처리는 서로 다른 처리 시간과 속도를 필요로 하기 때문에 최적의 Event 처리를 위한 전략이 필요 하다

A. 효과적인 Event 처리를 위하여 고려 하여야 할 사항

i. 어떻게 Event를 Detect 할 것인가? : Interrupt 가 일반적으로 Event 발생에 사용 되지만 경우에 따라서는 Polling이 필요 할 수 있다.

ii. Interrupt를 사용 할 때 Interrupt Service Routine(ISR) 내에서 처리 하는 부분과, ISR 외에서는 처리하는 부분을 어떻게 할당 하여 처리 할 것 인가? : 가능하면 ISR 내에서 처리 하는 부분을 짧게 하는 것이 좋다.

iii. 어떻게 Event 가 Non-ISR code와 통신 하도록 할 것인가.

FreeRTOS는 특별히 Event 처리를 위한 API와 Macros를 별도로 가지고 있지는 않다. 그러나 일부 API와 Macros를 ISR 내에서 사용 할 경우에는 끝에 'FromISR' 이 포함된 API 함수와 'FROM_ISR' 이 포함된 Macros를 사용 하여야 한다.

B. 이 장의 중요 목표

i. ISR에서 사용 되는 API 함수에 대한 이해

ii. Interrupt를 사용 하는 시스템의 구현에 대한 이해

iii. Binary Semaphores(세마포어)와 Counting Semaphores를 Create 하고 사용 하는 방법에 대한 이해

iv. Binary Semaphores와 Counting Semaphores의 차이

v. Queue(큐)를 이용한 ISR과 non-ISR 사이의 Data Passing

vi. Interrupt Nesting Model

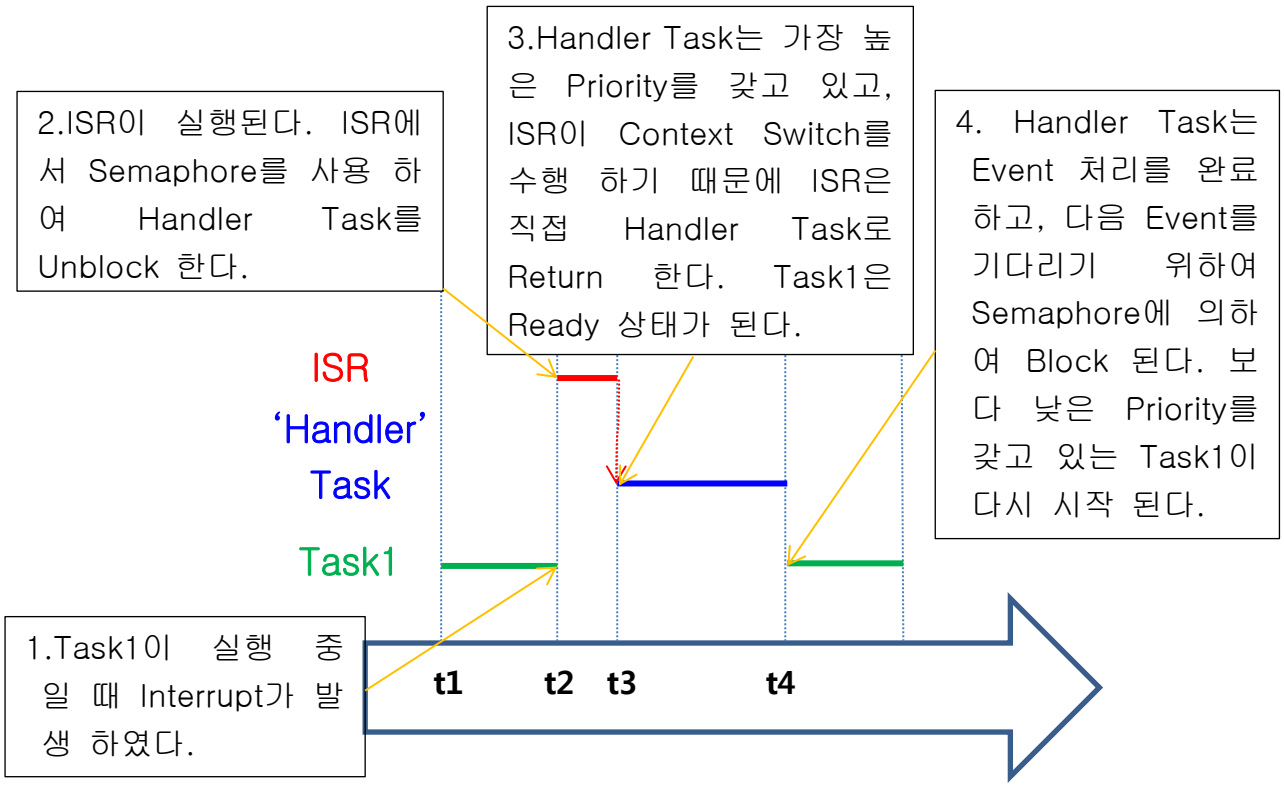
2. Deferred Interrupt Processing

A. Binary Semaphores를 이용한 동기(Synchronization)

i. Binary Semaphores는 Interrupt가 발생 하였을 때 특정한 Task를 Unblock 하는데 사용 할 수 있다. 이러한 기능은 Interrupt과 Task를 동기 시키데 효과적으로 이용 할 수 이고, 또한 ISR을 매우 빠르고 적은 량의 프로그램 코드만으로 구성 할 수 있게 한다.

ii. Time Critical 한 Interrupt 처리가 필요한 경우에는 Handler Task의 Priority를 다른 Task 보다 높게 설정 하여(Pre-empts) ISR에서 직접 Handler Task로 Return 하여 Handler Task가 바로 실행 되도록 한다.

현재 실행 중인 Task가 Interrupt되고 Handler Task에 Return 되는 예.



iii. Handler Task는 Event가 발생 하는 것을 기다리기 위하여 'Take'를 Call 하여 Blocking 상태에 있게 되고,

iv. Event가 발생 하면 ISR에서 'Give' Operation으로 Handler Task가

Unblock 상태 가 되게 한다.

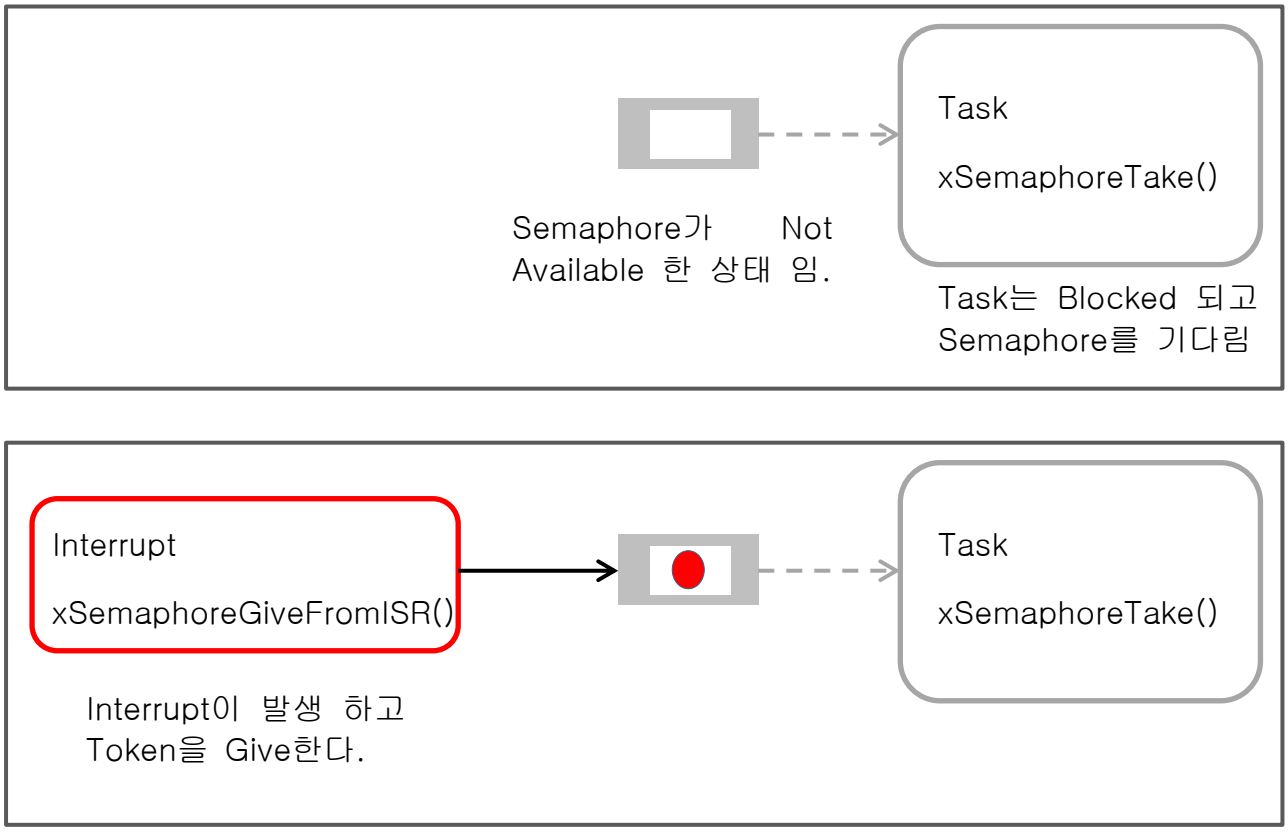
v. 위에서 설명한 Interrupt 와 Task를 동기 시키는 시나리오에서 사용한 Binary Semaphore는 길이가 1인 Queue로 생각 할 수 있다. 위 예에서 Queue는 하나의 Item만 저장 할 수 있기 때문에 Empty 상태 아니면 Full 상태(Binary) 에 있다.

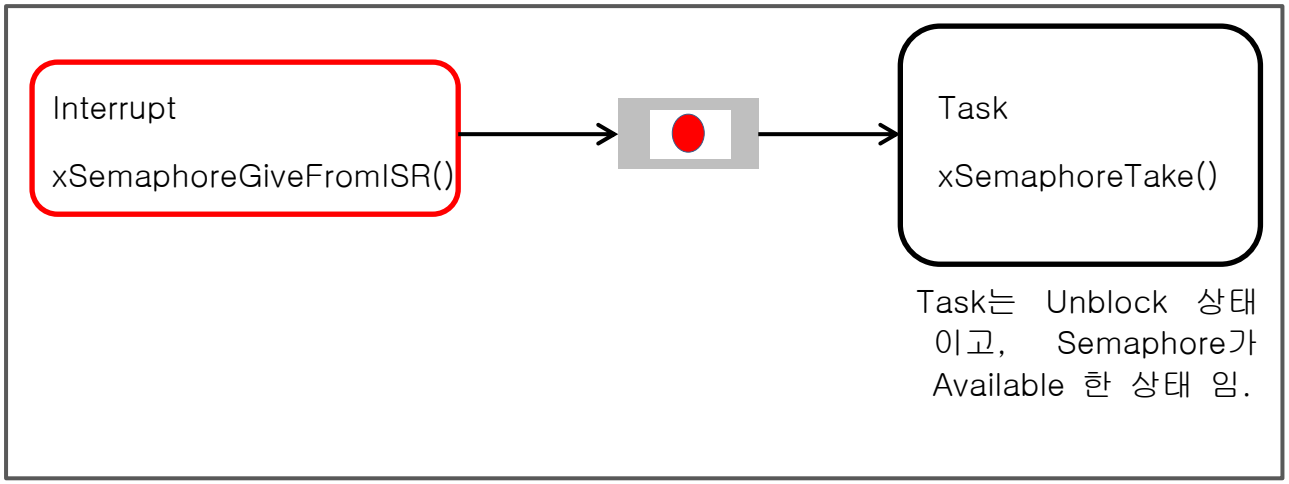
vi. Handler Task에서 xSemaphoreTake()를 Call 하였을 때 Queue가 비어 있으면 Task는 Blocked State가 된다.

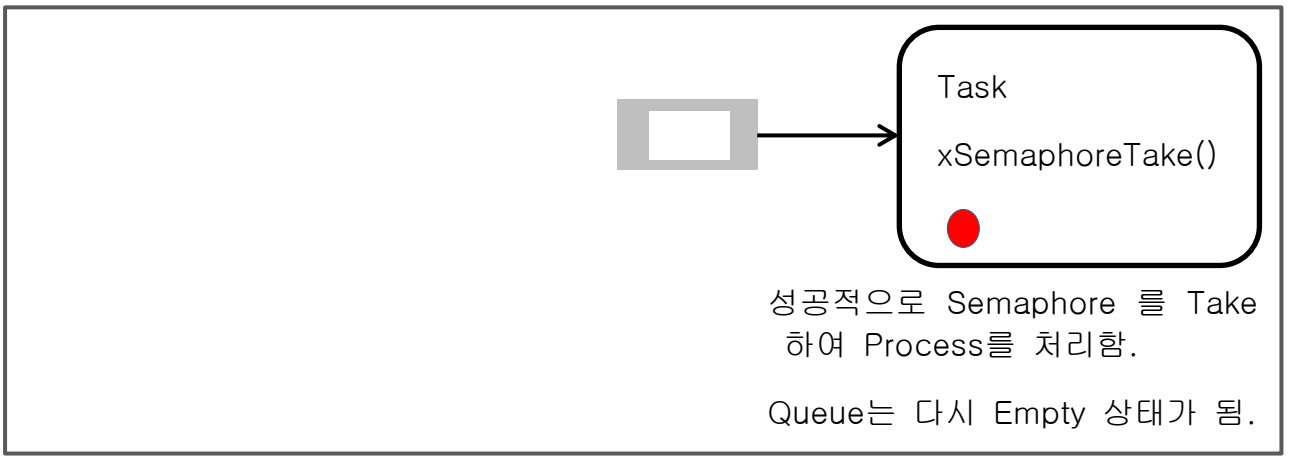
vii. 이 상태에서 Event가 발생 하면 ISR에서 xSemaphoreGiveISR() 함수를 사용 하여 Token을 Queue에 놓게 되고 Queue는 Full 상태가 된다. 그 결과 Handler Task는 Token 을 제거 함과 동시에 Blocked State를 벗어나서 Interrupt에 동기 하여 실행 하여야 할 Process를 처리 하게 된다.

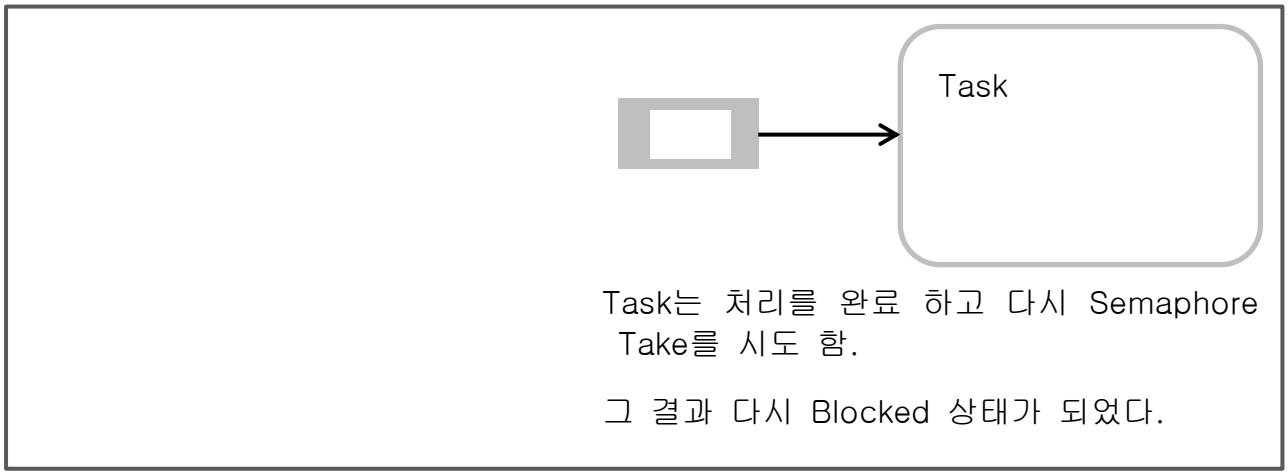
viii. 그리고, Queue는 다시 Empty 상태가 되어 다음 Event를 기다리게 된다.

Binary Semaphore를 이용 한 Interrupt와 Task 동기




Interrupt
xSemaphoreGiveFromISR()
Task
xSemaphoreTake()
Task는 Unblock 상태 이고, Semaphore가 Available 한 상태 임.


Task
xSemaphoreTake()
성공적으로 Semaphore 를 Take 하여 Process를 처리함.
Queue는 다시 Empty 상태가 됨.


Task
Task는 처리를 완료 하고 다시 Semaphore Take를 시도 함.
그 결과 다시 Blocked 상태가 되었다.

B. vSemaphoreCreateBinary() API Function

Semaphore는 사용되기 전에 Create되어야 한다.

xSemaphoreCreateBinary() API Function의 Prototype

void xSemaphoreCreateBinary (xSemaphoreHandle xSemaphore);

Parameter Name	Description
xSemaphore	Semaphore Handle

C. xSemaphoreTake() API Function

'Taking' Semaphore의 의미는 Semaphore를 'Receive' 또는 'Obtain'의 의미 이다. 는 사용되기 전에 Create되어야 한다.

xSemaphoreTake()는 Interrupt Service Routine에서 사용 할 수 없다.

xSemaphoreTake() API Function의 Prototype

portBase_TYPE xSemaphoreTake (xSemaphoreHandle xSemaphore, portTickType xTickToWait);

Parameter Name/ Returned value	Description
xSemaphore	Semaphore Handle
xTickToWait	Task가 Blocked State에서 Semaphore를 사용 할 수 있을 때 까지 기다리는 최대 Tick 수 만약 xTicksToWait가 0 이면 Semaphore를 회득 할 수 없으면 xSemaphoreTake()는 즉시 Return 된다. FreeRTOSConfig.h에서 INCLUDE_vTaskSuspend가 1로 Set되고, xTicksToWait 가 portMAX_DELAY로 설정 된 경우 Task는 Timing out 없이 무한이 기다린다.
Returned value	2가지 가능한 값을 갖는다. pdPASS: xSemaphoreTake()가 성공적으로 Semaphore를 획득 한 경우 pdFALSE: Semaphpre를 획득 할 수 없는 경우

D. xSemaphoreGiveFromISR() API 함수
xSemaphoreGiveFromISR()는 ISR에서 사용 하는 xSemaphoreGive() 함수의 특별한 형태 이다.

xSemaphoreGiveFromISR() API Function의 Prototype

portBase_TYPE xSemaphoreGiveFromISR (xSemaphoreHandle xSemaphore, portBASETYPE *pxHigherPriorityTaskWoken);

Parameter Name/ Returned value	Description
xSemaphore	Semaphore Handle
pxHigherPriorityTaskWoken	Semaphore를 사용 할 수 있을 때 까지 기다리는 Task 가 하나 또는 여러 개 일 수 있다. 만약 xSemaphoreGiveFromISR() 에 의하여 unblocked 되는 Task의 Priority 가 Interrupt에 의하여 실행이 중단된 Task 보다 높거나 같은 경우 xSemaphoreGiveFromISR() 함수는 *pxHigherPriorityTaskWoken을 pdTRUE 로 Set 한다. *pxHigherPriorityTaskWoken pdTRUE 로 Set 된 경우 Interrupt ISR 에서 직접 가장 Priority 가 높은 Task로 직접 Context Switching 할 수 있다.
Returned value	2가지 가능한 값을 갖는다. pdPASS: xSemaphoreGiveFromISR() 이 성공 한 경우 pdFALSE: Semaphpre가 이미 획득 가능한 상태이기 때문에 Semaphpre를 Give 할 수 없는 경우

E. Interrupt와 Task 동기에 Binary Semaphpre를 사용 하는 예 : RT_SW_Interrupt_binary_semaphore

```
// 실험 목표
//  Binary Semaphore를 이용 하여 Interrupt와 Task를 동기 방법에
//  대한 이해
//
// 실험 방법
```

```
//  Interrupt 입력 장치로 SW PD0를 사용 한다.
//  출력 장치로 LED를 사용 한다.
//  프로그램이 처음 시작 하면 LED 가 1회 점멸 한다.
//  SW 가 Push 되는 순간 Interrupt가 발생 하고, 이 Interrupt에 동기
하여
//  SW 가 Push 된 회수가 LED에 표시 된다.

//#define TICK_INTERRUPT_TIMER0

#include <stdlib.h>
#include <stdio.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <compat/deprecated.h>

//FreeRTOS include files
#include "FreeRTOS.h"
#include "task.h"
#include "croutine.h"
#include "semphr.h"


//User include files

#define SW_DEBOUNCE_TIME   20

static void vSW_counterHandlerTask();
static void vLED_displayTask();
static void init_port_setting(void);

// xSemaphoreHandle 변수를 선언 한다.
// Semaphore는 Task와 Interrupt의 동기에 사용 한다.
xSemaphoreHandle xBinarySemaphore = NULL;
static char sw_counter;

static void vSW_counterHandlerTask() {

    sw_counter = 0x00;
    for(;;)
    {
    // Semaphore를 획득 하지 못하면 Task는 Blocked 상태가 된다.
    // 외부 SW의 Interrupt에 의하여 Semaphore을 획득 할 때까지
    // 이 상태가 지속 된다.
```

```
            if(xSemaphoreTake(xBinarySemaphore, portMAX_DELAY )
                    == pdTRUE)
            {
                vTaskDelay(SW_DEBOUNCE_TIME);
                    // 만약 SW PD0 가 눌렸으면

                if((PIND & 0x01) == 0){
                    sw_counter++;
                    }
                EIMSK |= 0x01;    // External Interrupt 0 enable
            }
    }
}

// SW PD0의 누르면 SW가 Push 된 수를 LED에 표시 한다.
static void vLED_displayTask() {
    PORTF = 0xff;               // LED 가 1회 깜박 인다.
    vTaskDelay(500);
    PORTF = 0x00;
    vTaskDelay(500);

    for(;;)
    {
            PORTF = sw_counter;
            vTaskDelay(20);
    }
}

portSHORT main(void) {
    init_port_setting();

     // Semaphore는 사용 하기 전에 Create 되어야 한다.
    vSemaphoreCreateBinary(xBinarySemaphore );

    if(xBinarySemaphore != NULL){
            // Interrupt 와 동기를 위하여 vSW_counterHandlerTask에
            // 가장 높은 Priority를 설정 하였다.
            xTaskCreate(vSW_counterHandlerTask,
                    (signed portCHAR *)"vSW_counterHandlerTask",
                    configMINIMAL_STACK_SIZE*3, NULL,
                    tskIDLE_PRIORITY + 3, NULL );
            xTaskCreate(vLED_displayTask, (signed portCHAR
                    *)"vLED_displayTask",  configMINIMAL_STACK_SIZE,
```

```
                    NULL, tskIDLE_PRIORITY + 1, NULL );

            // RunSchedular
            vTaskStartScheduler();
    }
    for(;;) {}

    return 0;
}

static void init_port_setting(void){
    cli();                  //Disable all interrupts

     outp(0x00,DDRD);   // PORTD를 Input Port로 설정 한다.
     outp(0xff,PORTD);  // Pull up resistor 설정
     outp(0xff,DDRF);   // PORTF를 Output Port로 설정 한다.
     outp(0x00,PORTF);  // clear LED.

    EICRA = 0x02;       // External Interrupt 0, Falling Edge
                        // Asynchronously Interrupt
    EIMSK |= 0x01;      // External Interrupt 0 enable
    sei();              // Re-enable interrupts
}

// static void __interrupt __far SIG_INTERRUPT0( void )
SIGNAL (INT0_vect)
{
    static portBASE_TYPE xHigherPriorityTaskWoken;
    portBASE_TYPE xSemaStatus;

    EIMSK &= ~0x01;   // External Interrupt 0 Disable

    xHigherPriorityTaskWoken = pdFALSE;

    // 'Give' the semaphore to unblock the task.
    xSemaStatus = xSemaphoreGiveFromISR(xBinarySemaphore,
    &xHigherPriorityTaskWoken );

    if((xSemaStatus == pdTRUE) && (xHigherPriorityTaskWoken
     == pdTRUE ))
    {
            // Semaphore를 획득한 Task가 Unblocked 상태가 된다.
            // 그리고 이 Task의 Priority 가 Interrupt에 의하여 중단된
```

```
            // Task 보다 높은 경우 이 Task로 직접 Return을 한다.
            taskYIELD();
        }
        }
```

3. Counting Semaphores

A. RT_SW_Interrupt_binary_semaphore 예제 프로그램에서 Binary Semaphore가 Interrupt와 Task를 동기 시키는 데 사용 되었다. Interrupt와 Task를 동기 시키는 순서는 다음과 같다.
   i. Interrupt가 발생 한다.
   ii. Interrupt service Routine이 실행 되고, Semaphore 'Giving'에 의하여 Handler Task 가 Unblock 상태로 된다..
   iii. Interrupt 가 완료되자 마자 Handler Task 가 실행 된다. Handler Task 는 Semaphore를 'Take' 한다.
   iv. Handler Task는 Event 처리 프로그램을 실행 하고, 다시 Semaphore 'Take'를 시도 한다. 만약 Semaphore 가 즉시 유효 하지 않으면 다시 Blocked 상태가 된다.

   위의 실행 순서는 Interrupt의 발생 주기가 Event 처리 속도 보다 느린 경우에는 잘 실행 된다.

B. 만약 다른 Interrupt가 Handler Task가 먼저 발생한 Event를 처리 하는 도중에 발생 한다면 Semaphore는 Event를 Latch(Semaphore 가 다시 유효한 상태가 됨) 한 상태가 되고, Handler Task가 앞에 발생한 Event를 처리 하고 Semaphore 'Take'를 시도 하면 , Handler Task 는 Block 상태에 놓이지 않고 바로 새로 Latch된 Event를 처리 한다.

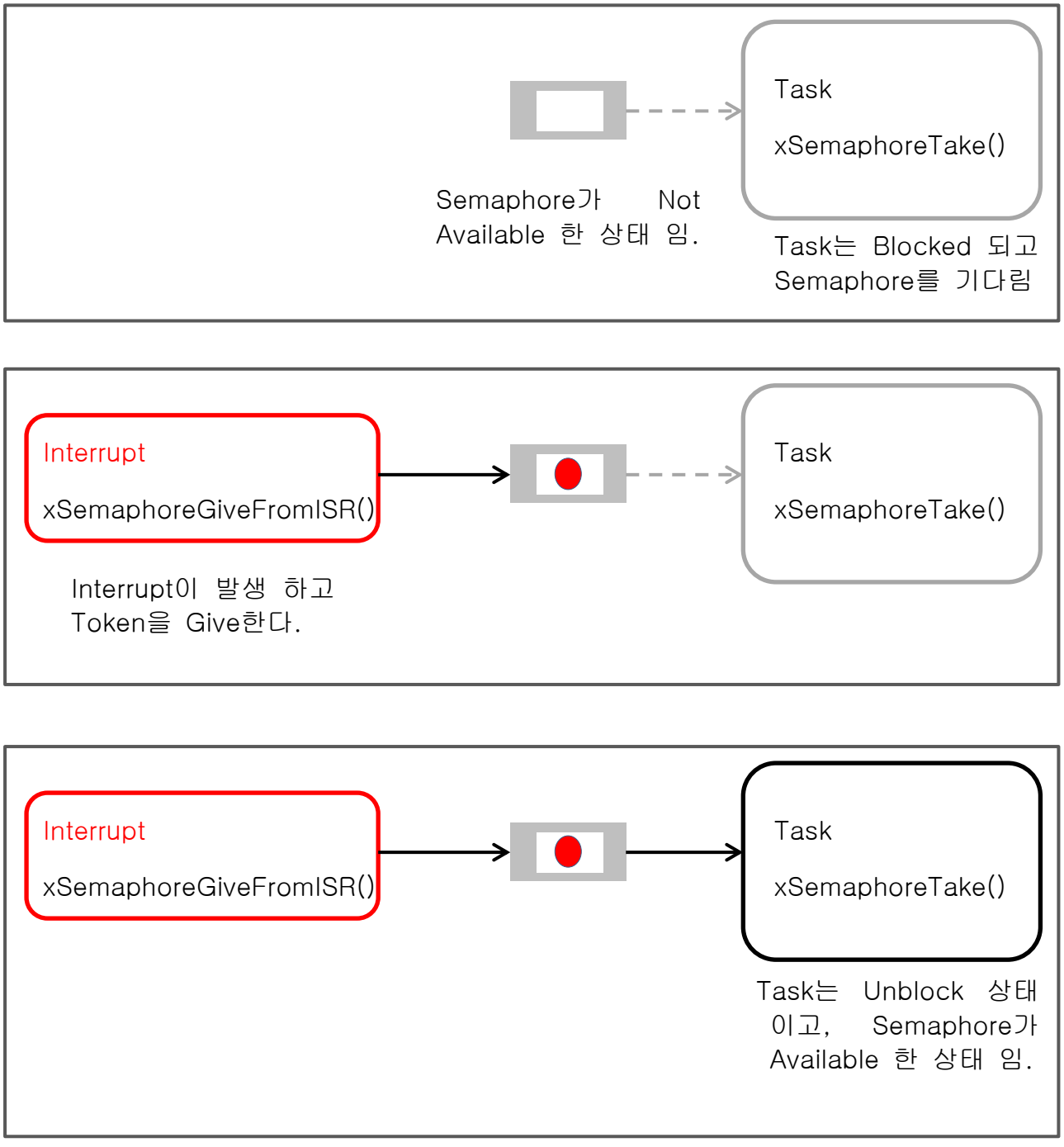
C. 그러나, 만약 , Handler Task 가 먼저 발생한 Event를 처리 하는 도중에 Semaphore에 Event 가 Latch 되고 이 것이 'Take' 되기 전에 또 새로운 Interrupt 가 발생 한다면, 이후에 발생 하는 Event 정보는 상실 된다.

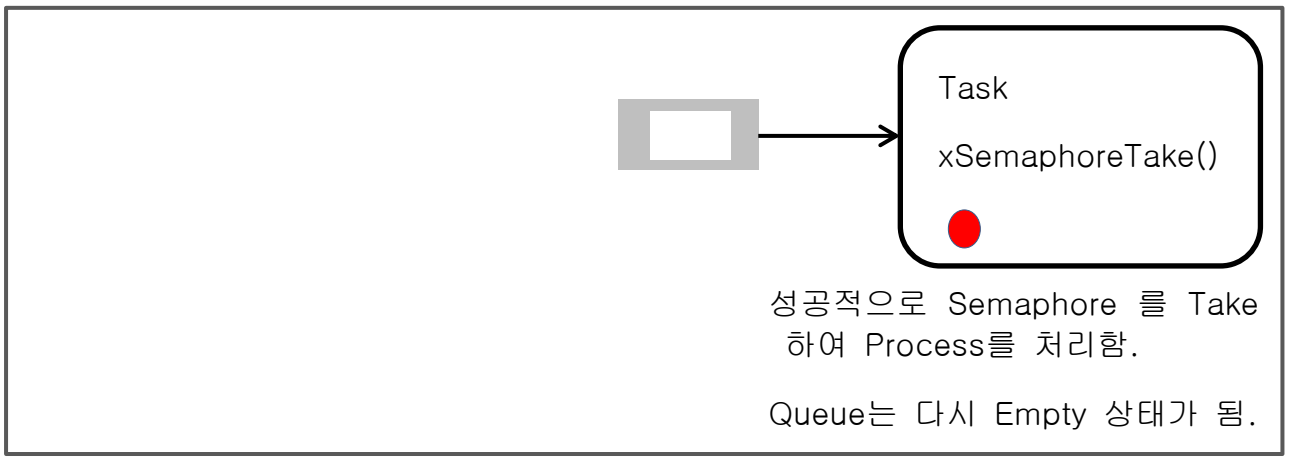
D. 이러한 문제를 해결 하기 위하여 Counting Semaphore가 필요 하다.

E. Binary Semaphores 는 길이가 1인 Queue 와 같이 생각 할 수 있고, Counting Semaphores는 1보다 큰 길이를 갖는 Queue로 생각 할 수 있다. 단 이 경우 Task는 Queue의 Data에는 관심이 없고 Queue가 Empty 인가 아닌가 에 만 관심이 있다.

F. Counting Semaphores를 사용 하는 경우에는 FreeRTOS.h 파일의 configUSE_COUNTING_SEMAPHORES 가 1로 Set 되어야 한다.

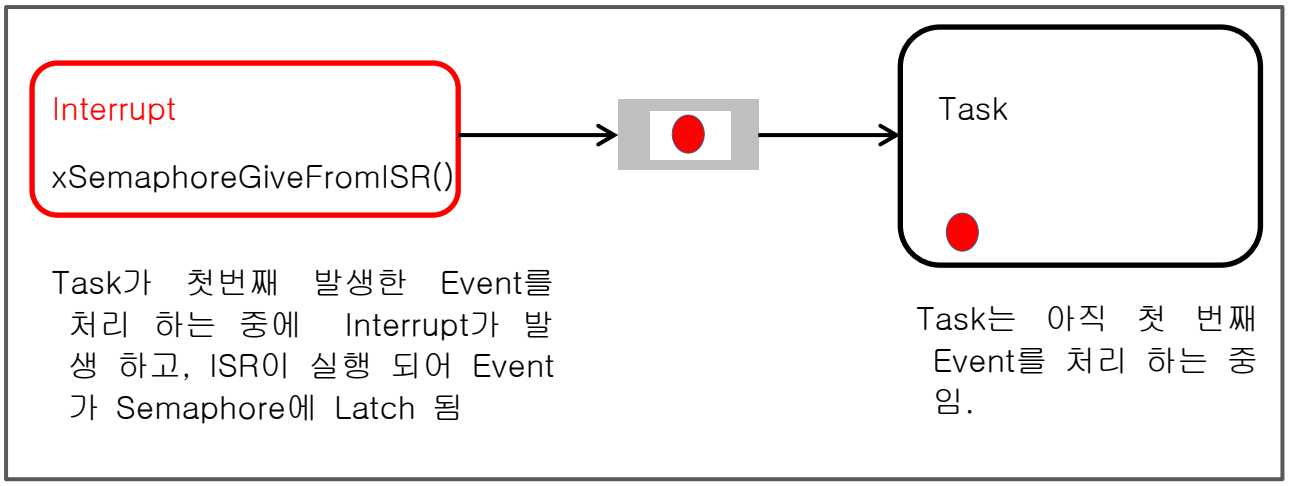
Binary Semaphore를 이용 한 Interrupt와 Task의 동기 예에서 앞서 발생한 Event를 처리 중에 새로운 Interrupt가 발생 하는 경우의 처리 순서





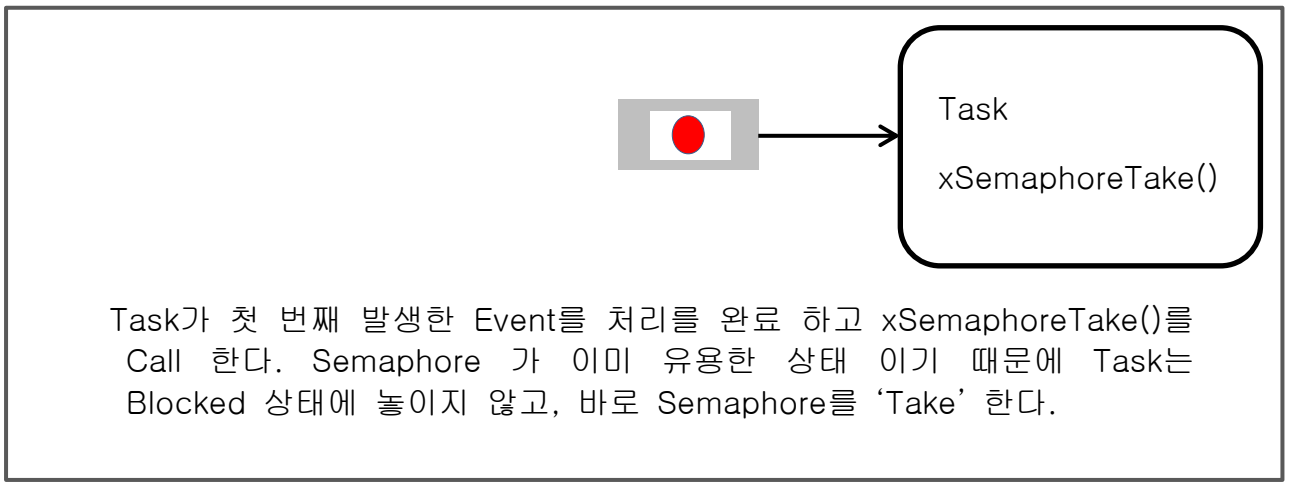
성공적으로 Semaphore 를 Take 하여 Process를 처리함.

Queue는 다시 Empty 상태가 됨.



Task가 첫번째 발생한 Event를 처리 하는 중에 Interrupt가 발생 하고, ISR이 실행 되어 Event가 Semaphore에 Latch 됨

Task는 아직 첫 번째 Event를 처리 하는 중임.

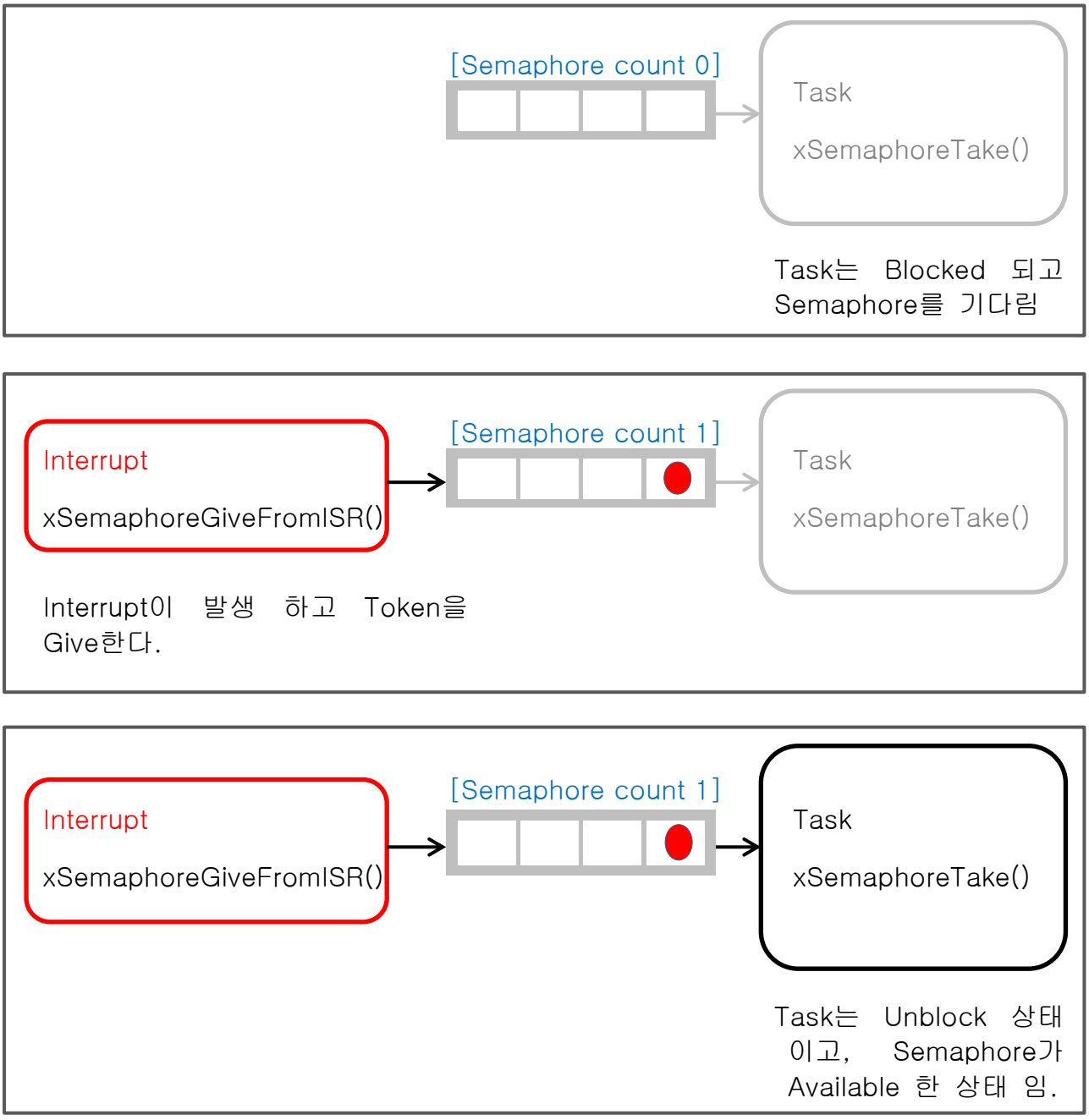


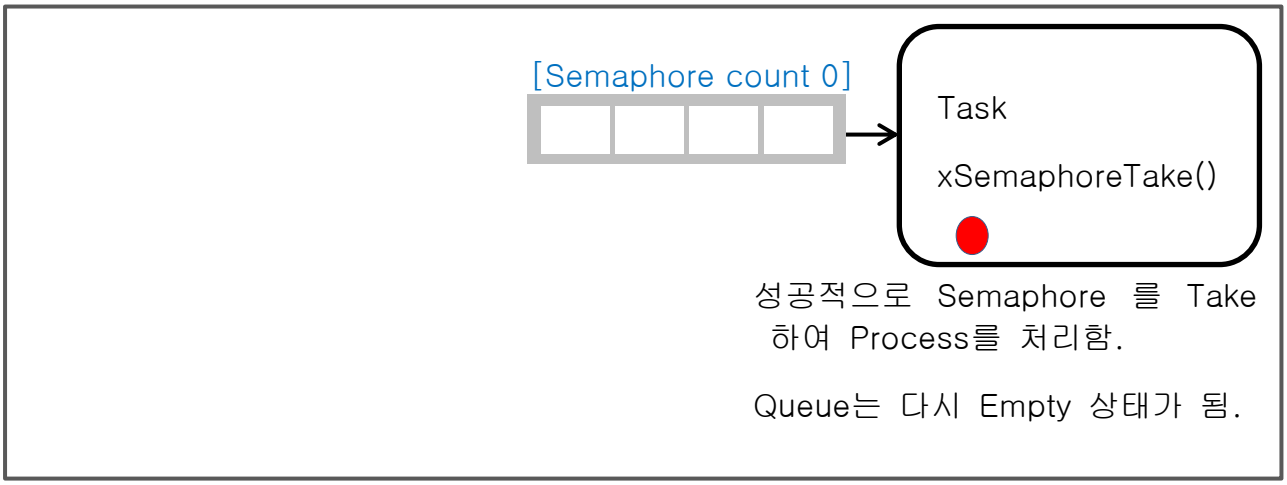
Task가 첫 번째 발생한 Event를 처리를 완료 하고 xSemaphoreTake()를 Call 한다. Semaphore 가 이미 유용한 상태 이기 때문에 Task는 Blocked 상태에 놓이지 않고, 바로 Semaphore를 'Take' 한다.

G. Counting Semaphores의 대표적인 용도는 다음과 같다.

i. Counting Semaphore를 Event Counting에 사용 하는 경우
Event Handler(ISR)은 Event가 발생 할 때 마다 Semaphore를 'Give' 한고, Semaphore의 Count는 1씩 증가 한다.
Handler Task 각 Event의 처리가 끝날 때 마다 Semaphore를 'Take' 한다. 이 때 마다 Semaphore count는 1씩 감소 한다.
Counting Semaphore를 Event Counting에 사용 하는 경우 Count Value의 초기 값을 0로 하여 Counting Semaphore를 Create 한다.

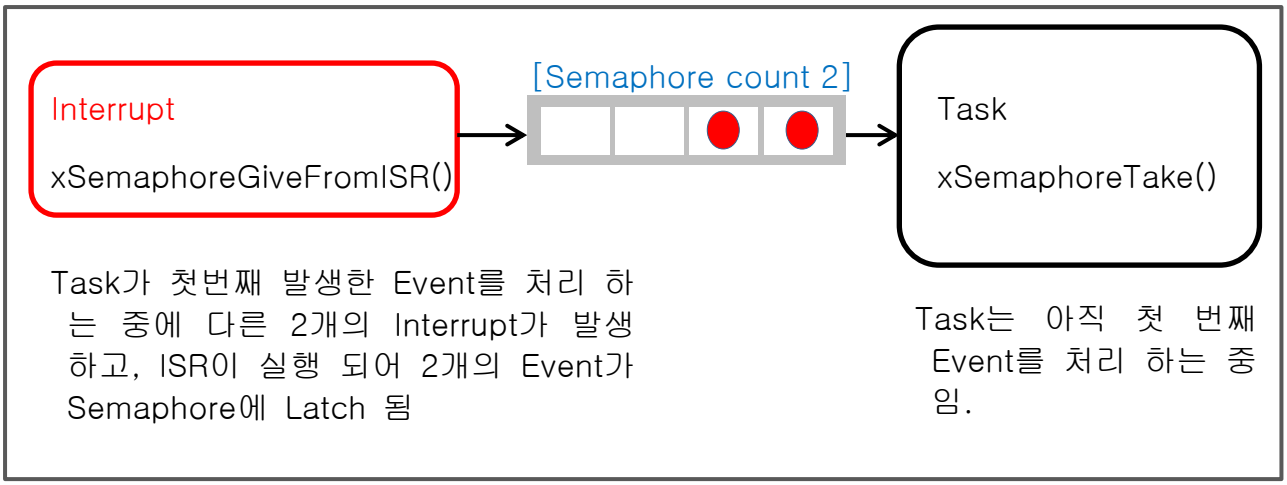
Counting Semaphore를 이용 한 Events Count의 처리 과정





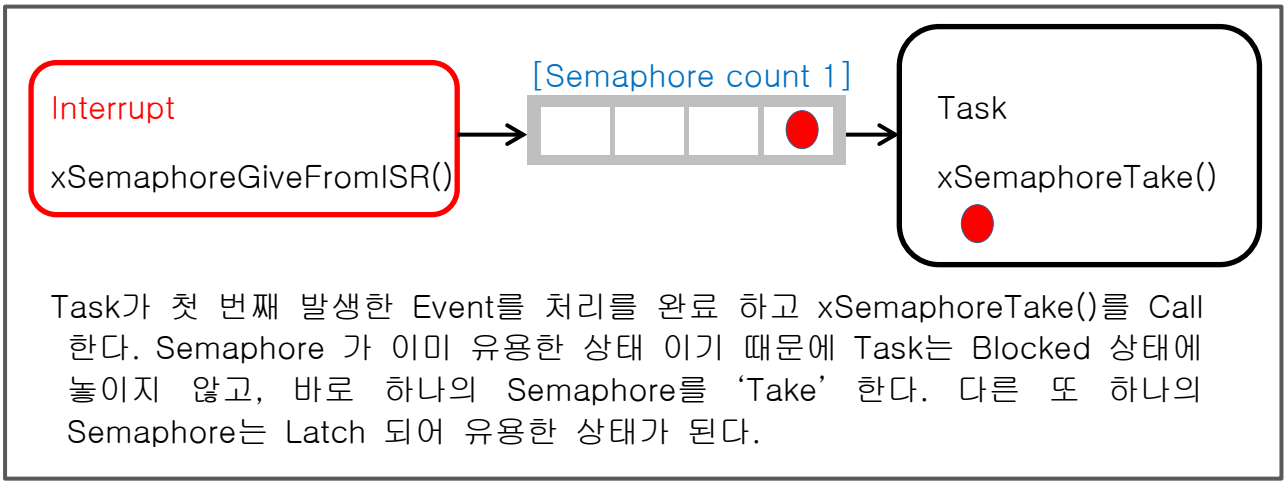
성공적으로 Semaphore 를 Take 하여 Process를 처리함.

Queue는 다시 Empty 상태가 됨.



Task가 첫번째 발생한 Event를 처리 하는 중에 다른 2개의 Interrupt가 발생 하고, ISR이 실행 되어 2개의 Event가 Semaphore에 Latch 됨

Task는 아직 첫 번째 Event를 처리 하는 중임.



Task가 첫 번째 발생한 Event를 처리를 완료 하고 xSemaphoreTake()를 Call 한다. Semaphore 가 이미 유용한 상태 이기 때문에 Task는 Blocked 상태에 놓이지 않고, 바로 하나의 Semaphore를 'Take' 한다. 다른 또 하나의 Semaphore는 Latch 되어 유용한 상태가 된다.

ii. Counting Semaphore를 Resource Management에 사용 하는 경우
이 경우 Semaphore의 Count Value는 사용 가능한 자원의 수를 표시 한다. Resource의 사용권을 획득 하기 위하여 Task는 먼저 Semaphore를 'Take' 하여야 한다. Task가 Semaphore를 'Take' 할 때 마다 Semaphore의 Count Value는 1씩 감소 하고 Count Value가 0가 되면 사용 할 수 있는 Resource가 없게 된다.
Task 의 Resource의 사용이 완료 될 경우 Task는 Semaphore를 'Give' 하고 Count Value는 1 증가 한다. 그러면 다른 Task가 자원을 하나 더 사용 할 수 있게 된다.
Counting Semaphore를 자원 관리에 사용 하는 경우 Count Value의 초기 값을 사용 가능한 자원의 개 수로 하여 Counting Semaphore를 Create 한다.

H. xSemaphoreCreateCounting() API 함수
FreeRTOS Semaphore의 Handle은 xSemaphoreHandle 형태의 변수에 저장 된다.
Semaphore는 사용 하기 전에 Create 되어야 한다. Counting Semaphore는 xSemaphoreCreateCounting() API 함수에 의하여 Create 된다.

xSemaphoreCreateCounting() API 함수의 Prototype

xSemaphoreHandle xSemaphoreCreateCounting (unsigned portBASE_TYPE uxMaxCount, unsigned portBASE_TYPE uxInitialCount)

Parameter Name/ Returned value	Description
uxMaxCount	Count 할 Semaphore의 최대 값. Semaphore가 Count 나 Events Latch로 사용 될 경우 uxMaxCount는 Latch 되는 Event의 최대 값
uxInitialCount	Semaphore 가 Create 될 때 Count Value의 초기 값 Semaphore가 Event counting에 사용 될 경우 아직 Event 가 발생 하지 않은 경우에 0로 설정 한다. Semaphore가 자원 관리에 사용 될 경우 사용 가능한 자원의 최대수(uxMaxCount)로 설정 한다.
Returned value	NULL이 Return 되는 경우에는 Semaphore가 Create 되지 못 한 경우 이다. Non-NULL이 Return 된 경우 이 값이 Create 된 Semaphore의 Handle로 저장 된다.

I. Task와 Interrupt의 동기에 Counting Semaphore를 사용 하는 예

프로그램 예 : RT_SW_Interrupt_counting_semaphore 참고 요

4. Interrupt Service Routine에서 Queue의 사용

Interrupt와 Task 간의 동기 및 Data Transfer에 Queue를 이용 한다.
ISR에서는
xQueueSendToFrontFromISR(), xQueueSendToBackFromISR(),
xQueueReceiveFromISR() API 함수를 사용 한다.

xQueueSendToFrontFromISR() API 함수의 Prototype

portBASE_TYPE xQueueSendToFrontFromISR (xQueuHandle xQueue,
void *pvItemToQueue, portBASE_TYPE *pxHigherPriorityTaskWoken);

xQueueSendToBackFromISR() API 함수의 Prototype

portBASE_TYPE xQueueSendToFrontFromISR (xQueuHandle xQueue,
void *pvItemToQueue, portBASE_TYPE *pxHigherPriorityTaskWoken);

Parameter Name/ Returned value	Description
xQueue	Data를 보낼(Written) Queue의 Handle, Queue Handle은 Queue가 생성될 때 Return 된다.
pvItemToQueue	Queue에 복사될 Data의 Pointer 각 Item의 Size는 Queue 가 생성될 때 결정 된다.
pxHigherPriorityTaskWoken	Semaphore를 사용 할 수 있을 때 까지 기다리는 Task 가 하나 또는 여러 개 일 수 있다. 만약 xQueueSendToFrontFromISR()나 xQueueSendToBackFromISR()에 의하여 unblocked 되는 Task의 Priority 가 Interrupt에 의하여 실행이 중단된 Task 보다 높거나 같은 경우 함수는 *pxHigherPriorityTaskWoken을 pdTRUE 로 Set 한다. *pxHigherPriorityTaskWoken이 pdTRUE

	로 Set 된 경우 Interrupt ISR 에서 직접 가장 Priority 가 높은 Task로 직접 Context Switching 할 수 있다.
Returned value	2가지 가능한 값을 갖는다. pdPASS: Data가 성공적으로 Queue에 보내진 경우 errQUEUE_FULL: Queue 가 Full 상태라 Data가 Queue에 보내지지 못한 경우

A. Interrupt와 Queue를 이용한 Serial 통신 프로그램 예
Serial 통신에 사용 하는 UART Driver는 Interrupt와 Queue를 이용 하는 대표적인 예이다. Serial 통신 예에서 Transmit Interrupt Handler와 Receive Interrupt Handler는 Queue를 통하여 문자를 주고 받는다.

프로그램 예 : RT_serial_comm 참고 요

Serial.c

```
// USART0 를 사용 하도록 설정 함
/* BASIC INTERRUPT DRIVEN SERIAL PORT DRIVER. */
static xQueueHandle xRxedChars;
static xQueueHandle xCharsForTx;

/*-----------------------------------------------------------*/
xComPortHandle xSerialPortInitMinimal( unsigned long ulWantedBaud,
unsigned portBASE_TYPE uxQueueLength )
{
        unsigned long ulBaudRateCounter;
        unsigned char ucByte;

        portENTER_CRITICAL();
        {
          /* Create the queues used by the com test task. */
          xRxedChars = xQueueCreate( uxQueueLength,
                ( unsigned portBASE_TYPE ) sizeof( signed char ) );
          xCharsForTx = xQueueCreate( uxQueueLength,
                ( unsigned portBASE_TYPE ) sizeof( signed char ) );

          // Calculate the baud rate register value from the equation
          //   in the data sheet.
          ulBaudRateCounter = ( configCPU_CLOCK_HZ /
```

```
            ( serBAUD_DIV_CONSTANT * ulWantedBaud ) ) -
            ( unsigned long ) 1;

            /* Set the baud rate. */
            ucByte = ( unsigned char ) ( ulBaudRateCounter &
              ( unsigned long ) 0xff );
            UBRR0L = ucByte;

            ulBaudRateCounter >>= ( unsigned long ) 8;
            ucByte = ( unsigned char ) ( ulBaudRateCounter &
              ( unsigned long ) 0xff );
            UBRR0H = ucByte;

            // Enable the Rx interrupt.   The Tx interrupt will get enabled
            // later. Also enable the Rx and Tx.
            UCSR0B = ( serRX_INT_ENABLE | serRX_ENABLE |
              serTX_ENABLE );

            /* Set the data bits to 8. */
//          UCSR0C = ( serUCSRC_SELECT | serEIGHT_DATA_BITS );
            UCSR0C = ( serEIGHT_DATA_BITS );
        }
        portEXIT_CRITICAL();

        // Unlike other ports, this serial code does not allow for more
        // than one com port.   We therefore don't return a pointer to a
        // port structure and can instead just return NULL.
        return NULL;
}
//-----------------------------------------------------
signed portBASE_TYPE xSerialGetChar( xComPortHandle pxPort,
        signed char *pcRxedChar, portTickType xBlockTime )
{
        /* Only one port is supported. */
        ( void ) pxPort;

        // Get the next character from the buffer.   Return false if no
        // characters are available, or arrive before xBlockTime expires.
        if( xQueueReceive( xRxedChars, pcRxedChar, xBlockTime ) )
        {
                return pdTRUE;
        }
        else
```

```
        {
                return pdFALSE;
        }
}
//-------------------------------------------------

signed portBASE_TYPE xSerialPutChar( xComPortHandle pxPort,
        signed char cOutChar, portTickType xBlockTime )
{
        /* Only one port is supported. */
        ( void ) pxPort;

        // Return false if after the block time there is no room on the
        // Tx queue.
        if( xQueueSend( xCharsForTx, &cOutChar, xBlockTime ) !=
          pdPASS )
        {
                return pdFAIL;
        }
        vInterruptOn();
        return pdPASS;
}

//-------------------------------------------------
SIGNAL( SIG_UART0_RECV )
{
signed char cChar;
signed portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

        // Get the character and post it on the queue of Rxed characters.
        // If the post causes a task to wake force a context switch as the
        // woken task          may have a higher priority than the task we
        // have interrupted.
        cChar = UDR0;

        xQueueSendFromISR( xRxedChars, &cChar,
        xHigherPriorityTaskWoken );

        if( xHigherPriorityTaskWoken != pdFALSE )
        {
                taskYIELD();
        }
}
```

```
/*-----------------------------------------------------------*/
SIGNAL( SIG_UART0_DATA )
{
signed char cChar, cTaskWoken;

        if( xQueueReceiveFromISR( xCharsForTx, &cChar, &cTaskWoken )
           == pdTRUE )
        {
                /* Send the next character queued for Tx. */
                UDR0 = cChar;
        }
        else
        {
                /* Queue empty, nothing to send. */
                vInterruptOff();
        }
}
```

5. Interrupt Nesting

최근의 FreeRTOS Ports 는 Interrupt Nest를 허용 한다. 이 경우 아래에서 설명 하는 하나 또는 두 개의 상수를 FreeRTOSConfig.h에서 정의 하여 주어야 한다.

configKERNEL_INTERRUPT_PRIORITY

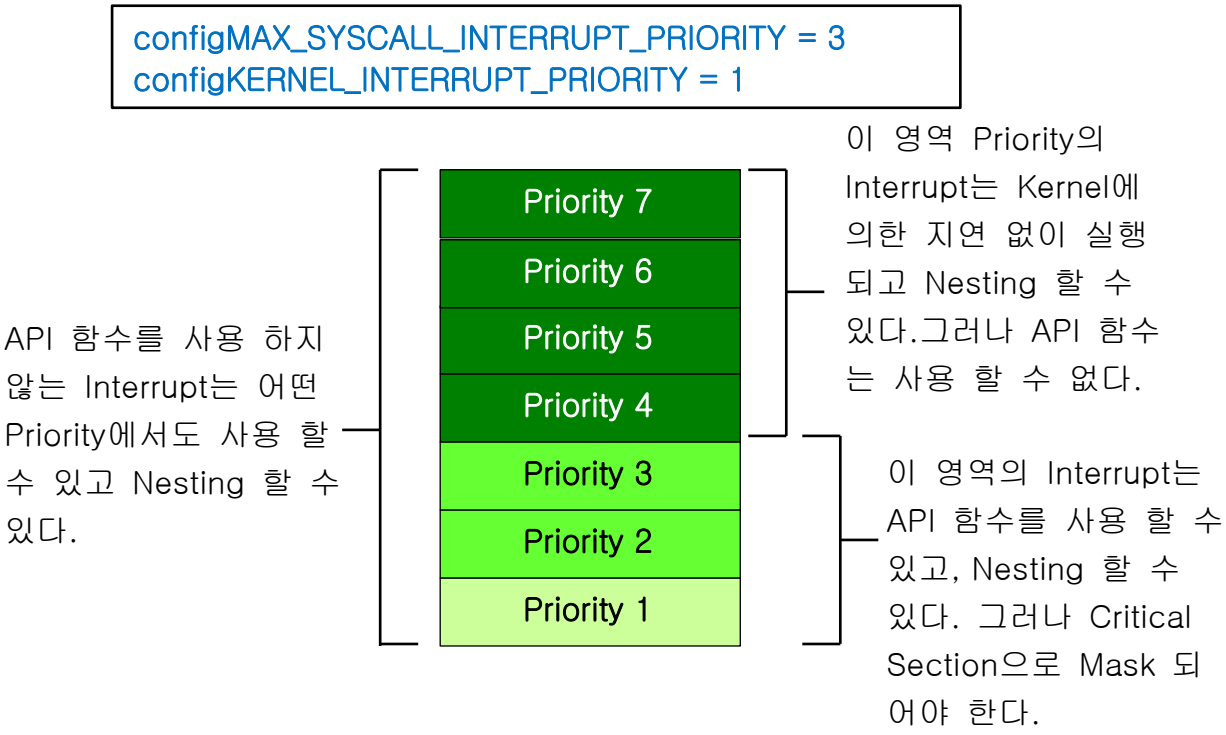
Tick Interrupt에서 사용 하는 Interrupt Priority를 설정 한다.
만약 configMAX_SYSCALL_INTERRUPT_PRIORITY 상수를 사용 하지 않는 경우에는 interrupt-safe FreeRTOS API를 사용 하는 Interrupt는 이 Priority에서 실행 된다.

configMAX_SYSCALL_INTERRUPT_PRIORITY

interrupt-safe FreeRTOS API를 Call 할 수 있는 가장 높은 Interrupt Priority를 설정 한다.

Full Interrupt Nesting Model은 configKERNEL_INTERRUPT_PRIORITY 보다 더 높은 값으로 configMAX_SYSCALL_INTERRUPT_PRIORITY를 설정 함으로서 구현 된다.

FreeRTOS에서는 일반적으로 아래의 예와 같이 Interrupt를 사용 할 수 있다. 그러나 특정한 프로세서에 적용 하는 경우 프로세서에서 제공하는 Interrupt에 대하여 이해 하고 응용 하여야 한다.


configMAX_SYSCALL_INTERRUPT_PRIORITY = 3
configKERNEL_INTERRUPT_PRIORITY = 1
Priority 7
Priority 6
Priority 5
Priority 4
Priority 3
Priority 2
Priority 1
이 영역 Priority의 Interrupt는 Kernel에 의한 지연 없이 실행 되고 Nesting 할 수 있다.그러나 API 함수는 사용 할 수 없다.
API 함수를 사용 하지 않는 Interrupt는 어떤 Priority에서도 사용 할 수 있고 Nesting 할 수 있다.
이 영역의 Interrupt는 API 함수를 사용 할 수 있고, Nesting 할 수 있다. 그러나 Critical Section으로 Mask 되어야 한다.