TI DSP, MCU 및

Xilinx Zynq FPGA 프로그램 전문가 과정

zero_bird Portfolio

최초 작성일: 2018. 06. 13

최종 작성일: 2018. 06. 00

작성자: 은태영(zero_bird@naver.com)

Library

1 FreeRTOS (V10.0.0)

1.1 API 사용 시 주의사항

FreeRTOSAPI 사용 시 다음 규칙이 적용된다.

1. “FromISR”로 끝나지 않는 API 기능은 ISR(인터럽트 서비스 루틴)에서 사용해서는 안된다.

일부 FreeRTOS 포트는 “FromISR”로 끝나는 API 함수도 configMAX_SYSCALL_INTERRUPT_PRIORITY(포트별로 명칭이 다르다. configMAX_API_CALL_INTERRUPT_PRIORITY)보다 우선순위가 높은 인터럽트 서비스 루틴을 사용할 수 없도록 제한한다.

인터럽트가 설정에 의해 결정된 값보다 더 높은 우선순위를 갖게 될 경우, freeRTOS 는 configMAX_SYSCALL_INTERRUPT_PRIORITY 에 영향을 줄 수 없다.

2. 스케줄러가 일시 중단된 상태에서 컨텍스트 스위치를 유발할 수 있는 API 기능을 호출하면 안된다.

3. 컨텍스트 스위치를 유발할 수 있는 API 기능은 크리티컬 섹션 내에서 호출하면 안된다.

1.2 Task 및 스케줄러 API

1.2.1 task 생성 및 제거

1.2.1.1 xTaskCreate() : 메모리 동적 할당 task 생성

```
#include “FreeRTOS.h”
#include “task.h”

BaseType_t xTaskCreate ( TaskFunction_t pvTaskCode, const char * const pcName, unsigned short usStackDepth,
                         void *pvParameters, UBaseType_t uxPriority, TaskHandle_t *pxCreatedTask );
```

새로운 task 를 생성한다.

각 task 는 task 상태를 유지하는데 사용되는 RAM(task control block, TCB)이 필요하며 해당 작업은 스택으로 사용한다. xTaskCreate()를 사용하여 task 를 만들면 필요한 RAM 이 FreeRTOS 힙에서 자동 할당된다. 새로 생성된 task 는 처음 준비 상태에서, 높은 우선순위의 task 가 없을 경우 즉시 실행 상태가 된다. 스케줄러가 시작되기 전과 후에 task 를 생성할 수 있다.

이 함수를 사용하려면 configSUPPORT_DYNAMIC_ALLOCATION 에 있는 FreeRTOSConfig.h 을 1 로 설정하거나 설정되지 않은 상태로 두어야한다.

1.2.1.1.1 매개 변수

pvTaskCode : task 는 일반적으로 무한 루프로 구현되어 있는 c 함수다. 해당 매개 변수는 단순히 task 를 구현하는 함수(사실상 함수의 이름)에 대한 포인터다.

pcName : 작업을 설명하는 이름이다. 이건 주로 디버깅을 쉽게 하기 위해 사용되지만 xTaskGetHandle()을 이용하여 작업 핸들을 얻을 수도 있다. 해당 이름의 최대 길이는 NULL 종결자를 포함해서 configMAX_TASK_NAME_LEN 을 이용해 설정한다. 이 문자열보다 길 경우 자동으로 문자열이 잘린다.

usStackDepth : task 는 생성될 때, 커널이 task 에 할당한 고유 스택이 있다. usStackDepth 값은 커널에 사용하고자 하는 스택의 크기를 알려준다. 이 값은 스택이 보유할 수 있는 문자의 수를 지정하며 바이트 수는 지정하지 않는다. 예를 들어 스택 폭이 4 바이트인 아키텍처에서 usStackDepth 가 100 으로 전달되면 400 바이트의 스택 공간이 할당된다. (100 * 4 바이트)

스택 폭에 스택 깊이를 곱한 값은 size_t 유형의 변수에 포함될 수 있는 최대 값을 초과하지 않는다. idle task 에 사용되는 스택의 크기는 configMINIMAL_STACK_SIZE 에 의해 설정된다. 선택한 마이크로 컨트롤러 아키텍처를 위해 데모 어플리케이션이 제공된다. 이 애플리케이션에서 할당된 값은 해당 아키텍처에 관한 모든 task 들에 대해서 권장되는 최소값이다. 작업에서 많은 스택 공간을 사용하는 경우, 더 큰 값을 할당해야 한다.

pvParameters : task 함수는 ‘void *’유형의 매개변수를 허용한다. pvParameters 에 할당된 값은 task 에 전달되는 값이다. 이 매개변수에는 task 매개 변수를 효과적으로 허용하고 캐스팅을 통하여 간접적으로 모든 유형의 매개변수를 받을 수 있도록 ‘void *’유형으로 되어 있다.

예를 들어 정수 유형은 task 가 작성된 지점에서 정수를 void *로 캐스팅한 다음, void *매개변수를 task 함수가 보유하고 있는 매개 변수로 다시 캐스팅하여 task 함수에 전달된다.

uxPriority : task 의 우선순위를 정의한다. 가장 낮은 우선순위 0(config)부터 가장 높은 우선순위 (configMAX_PRIORITIES -1)까지 지정할 수 있다. configMAX_PRIORITIES 는 사용자가 지정할 수 있다.

configUSE_PORT_OPTIMISED_TASK_SELECTION 을 0 으로 설정할 경우 사용 가능한 우선순위 숫자의 상한 값이 없어진다. (사용되는 데이터 유형의 한계와 마이크로 컨트롤러에서 사용하는 RAM 제외한다.) 그러나, RAM 낭비를 방지하기 위해 필요한 우선순위 수를 적게 사용하는것이 좋다.

uxPriority 값을 가장 높은 우선순위(configMAX_PRIORITIES - 1)이상으로 할 경우, task 의 우선순위는 최대 사용 가능한 우선순위로 자동으로 제한된다.

pxCreatedTask : pxCreatedTask 는 생성중인 task 에 대한 핸들을 전달하는데 사용된다. 해당 핸들은 예를 들어 task 우선순위를 변경하거나 task 를 삭제하는 API 를 호출하는 등 task 를 참조할 수 있다. 어플리케이션에서 task 핸들을 사용하지 않으면 pxCreatedTask 를 NULL 로 설정할 수 있다.

1.2.1.1.2 반환 값

errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY : 작업이 성공적으로 만들어 졌는지를 나타낸다.

FreeRTOS 가 task 데이터 구조와 스택을 할당하는데 있어서 사용할 수 있는 힙 메모리가 충분하지 않을 경우 task 를 생성할 수 없음을 나타낸다. heap_1.c, heap_2.c 또는 heap_4.c 가 프로젝트에 포함되어 있으면 사용 가능한 힙의 총량은 FreeRTOSConfig.h 의 configTOTAL_HEAP_SIZE 에 의해 정해지며, 메모리 할당 실패는 vApplicationMallocFailedHook()함수(callback 또는 hook)를 사용하고, 남아있는 힙 메모리 양은 xPortGetFreeHeapSize()함수를 사용해서 확인할 수 있다.

heap_3.c 가 프로젝트에 포함되면 최대 힙 크기는 링커 구성에 의해 정해진다.

1.2.1.1.3 기타

```
// xStruct 구조체와 xStruct 타입의 변수를 정의한다. 이것은 task 함수로 전달되는 매개변수를 위하여 사용된다.
typedef struct A_STRUCT {
        char cStructMember1;
        char cStructMember2;
} xStruct;

// task 매개변수로 전달할 xStruct 을 세팅한다.
xStruct xParameter = { 1, 2 };

// 생성할 task 를 정의한다. task 를 구현하는 함수의 이름은 xTaskCreate()함수의 첫번째 매개변수이다.
void vTaskCode( void * pvParameters ) {
        xStruct *pxParameters;

        // void *매개변수를 다시 필요한 유형으로 캐스팅한다.
        pxParameters = ( xStruct * ) pvParameters;

        // 매개변수 값에 따라 처리한다.
        if( pxParameters->cStructMember1 != 1 ) {
        }

        // 무한루프를 이용하여 기능을 구현한다.
        for( ;; ) {
        }
}

// task 를 생성하는 함수를 정의한다. 스케줄러가 시작되기 전이나 후에 호출할 수 있다.
void vAnotherFunction( void ) {
        TaskHandle_t xHandle;

        /*task 를 생성한다. (task 의 함수 포인터, "작업 이름", 스택의 크기(바이트가 아니라 단어로 정의된다.),
                        매개변수(경고 방지를 위해 void *로 변환), task 우선순위, task 핸들);    이다. */
        if( xTaskCreate( vTaskCode, "Demo task", STACK_SIZE, (void*) &xParameter, TASK_PRIORITY, &xHandle) != pdPASS ) {
            /* 힙 메모리가 부족해서 task 생성이 불가능하다. heap_1.c, heap_2.c 또는 heap_4.c 가 프로젝트에 포함되어 있으면
            vApplicationMallocFailedHook()(callback 또는 hook)함수를 사용하여 이 상황을 trapped 할 수 있으며 할당되지 않은
            FreeRTOS 힙 메모리의 양을 xPortGetFreeHeapSize()함수를 사용하여 확인할 수 있다. */
        } else {
            // task 가 성공적으로 생성되었다. 핸들을 이용하여 다른 API 함수를 사용할 수 있다.(ex. 우선순위 변경 등)
            vTaskPrioritySet( xHandle, 2 );
        }
}
```

1.2.1.2 xTaskCreateStatic() : 메모리 정적 할당 task 생성

```
#include “FreeRTOS.h”
#include “task.h”

BaseType_t xTaskCreateStatic ( TaskFunction_t pvTaskCode, const char * const pcName,
                               unsigned short usStackDepth, void *pvParameters, UBaseType_t uxPriority,
                               StackType_t * const puxStackBuffer, StaticTask_t * const pxTaskBuffer );
```

새로운 task 를 생성한다.

각 task 는 task 상태를 유지하는데 사용되는 RAM(task control block, TCB)이 필요하며 해당 작업은 스택으로 사용한다. xTaskCreateStatic()을 사용해서 task 를 만들 경우 **작성자에 의해 RAM 이 제공되며, 이를 위해 두 개의 추가 매개변수가 필요로 한다.** 이 방식은 **컴파일 시 RAM 을 정적으로 할당할 수 있다.** 새로 생성된 task 는 처음 준비 상태에서, 높은 우선순위의 task 가 없을 경우 즉시 실행 상태가 된다. 스케줄러가 시작되기 전과 후에 task 를 생성할 수 있다.

이 함수를 사용하려면 configSUPPORT_DYNAMIC_ALLOCATION 에 있는 FreeRTOSConfig.h 을 1 로 설정하거나 설정되지 않은 상태로 두어야한다.

1.2.1.2.1 매개 변수

pvTaskCode : task 는 일반적으로 무한 루프로 구현되어 있는 c 함수다. 해당 매개 변수는 단순히 task 를 구현하는 함수(사실상 함수의 이름)에 대한 포인터다.

pcName : 작업을 설명하는 이름이다. 이건 주로 디버깅을 쉽게 하기 위해 사용되지만 xTaskGetHandle()을 이용하여 작업 핸들을 얻을 수도 있다. 해당 이름의 최대 길이는 NULL 종결자를 포함해서 configMAX_TASK_NAME_LEN 을 이용해 설정한다. 이 문자열보다 길 경우 자동으로 문자열이 잘린다.

ulStackDepth : puxStackBuffer 매개변수는 StackType_t 변수의 배열을 xTaskCreateStatic()에 전달하는데 사용한다. ulStackDepth 는 배열의 인덱스 수를 설정한다.

pvParameters : task 함수는 ‘void *’유형의 매개변수를 허용한다. pvParameters 에 할당된 값은 task 에 전달되는 값이다. 이 매개변수에는 task 매개 변수를 효과적으로 허용하고 캐스팅을 통하여 간접적으로 모든 유형의 매개변수를 받을 수 있도록 ‘void *’유형으로 되어 있다.

예를 들어 정수 유형은 task 가 작성된 지점에서 정수를 void *로 캐스팅한 다음, void *매개변수를 task 함수가 보유하고 있는 매개 변수로 다시 캐스팅하여 task 함수에 전달된다.

uxPriority : task 의 우선순위를 정의한다. 가장 낮은 우선순위 0(config)부터 가장 높은 우선순위 (configMAX_PRIORITIES -1)까지 지정할 수 있다. configMAX_PRIORITIES 는 사용자가 지정할 수 있다.

configUSE_PORT_OPTIMISED_TASK_SELECTION 을 0 으로 설정할 경우 사용 가능한 우선순위 숫자의 상한 값이 없어진다. (사용되는 데이터 유형의 한계와 마이크로 컨트롤러에서 사용하는 RAM 제외한다.) 그러나, RAM 낭비를 방지하기 위해 필요한 우선순위 수를 적게 사용하는것이 좋다.

uxPriority 값을 가장 높은 우선순위(configMAX_PRIORITIES - 1)이상으로 할 경우, task 의 우선순위는 최대 사용 가능한 우선순위로 자동으로 제한된다.

puxStackBuffer : ulStackDepth 인덱스를 가진 StackType_t 변수 배열을 가리킨다. (위의 ulStackDepth 참조.) 배열은 생성된 task 의 스택으로 사용되므로 영구적이어야 한다. (함수로 만든 스택 프레임이나 어플리케이션이 실행될 때 다른 메모리에서 덮어쓸 수 있는 메모리는 사용하지 않는다.)

pxTaskBuffer : StaticTask_t 유형의 변수를 가리킨다. 변수는 생성된 task 의 데이터 구조(TCB)를 유지하는데 사용되므로 영구적이어야 한다. (함수 또는 어플리케이션이 실행될 때 다른 메모리에서 덮어쓸 수 있는 메모리는 사용하지 않는다.)

1.2.1.2.2 반환 값

NULL : puxStackBuffer 또는 pxTaskBuffer 가 NULL 이기 때문에 task 를 만들 수 없다.

다른 값 : NULL 이 아닌 값이 리턴되면 task 가 만들어지고 리턴 값은 작성된 task 의 핸들이다.

1.2.1.2.3 기타

```
/* 생성할 task 가 스택으로 사용할 버퍼의 크기를 지정한다. 이것은 스택이 차지할 단어 수이며 바이트 수가 아니다.
예를 들어, 각 스택 항목이 32bit 고 값이 100 으로 설정되면 400byte(100 * 32bit)가 할당된다. */
#define STACK_SIZE 200

// 생성되는 task 의 TCB 를 보유한다.
StaticTask_t xTaskBuffer;

// 생성되는 task 가 스택으로 사용할 버퍼다. 이것은 StackType_t 변수의 배열이다. StackType_t 크기는 RTOS 포트마다 다르다.
StackType_t xStack[ STACK_SIZE ];

// 생성되어 동작하는 task 함수이다.
void vTaskCode( void * pvParameters ) {

        // xTaskCreateStatic()호출 시 pvParameters 매개 변수에 1 이 전달되기 때문에 1 로 비교한다.
        configASSERT( ( uint32_t ) pvParameters == 1UL );

        for( ;; ) {
            // 작업 코드가 위치한다.
        }
}

// task 를 생성하는 함수이다.
void vFunction( void ) {

        TaskHandle_t xHandle = NULL;

        /* 동적 메모리 할당을 사용하지 않고 task 를 생성한다. task 함수, 이름, xStack 배열의 인덱스 수,
        task 의 매개변수, task 우선순위, task 가 스택으로 사용할 배열, task 데이터 구조를 유지하는 변수.

        puxStackBuffer 와 pxTaskBuffer 가 NULL 이 아니므로 작업이 생성되고 xHandle 이 task 의 핸들이 된다. */
        xHandle = xTaskCreateStatic ( vTaskCode, "NAME", STACK_SIZE,
                                      ( void * ) 1, tskIDLE_PRIORITY, xStack, &xTaskBuffer );

        // 핸들을 사용하여 작업을 일시 중단 한다.
        vTaskSuspend ( xHandle );
}
```

1.2.1.3 xTaskCreateRestricted() : MPU 보호 task 생성

```
#include “FreeRTOS.h”
#include “task.h”

BaseType_t xTaskCreateRestricted ( TaskParameters_t *pxTaskDefinition, TaskHandle_t *pxCreatedTask );
```

이 기능은 FreeRTOS MPU 포트(메모리 보호 장치를 사용하는 FreeRTOS 포트)에 관련되어 있다.

MPU(메모리 보호 장치)로 제한된 task 를 만든다.

생성된 task 는 처음에 준비상태로 설정되지만, 실행할 수 있는 더 높은 우선순위 작업이 없다면 실행 상태가 된다.

스케줄러가 시작되기 전후에 task 를 생성할 수 있다.

1.2.1.3.1 매개 변수

pxTaskDefinition : task 를 정의하는 구조체를 가리키는 포인터이다.

```
typedef struct xTASK_PARAMTERS {
        TaskFunction_t pvTaskCode;
        const signed char * const pcName;
        unsigned short usStackDepth;
        void *pvParameters;
        UBaseType_t uxPriority;
        portSTACK_TYPE *puxStackBuffer;
        MemoryRegion_t xRegions[ portNUM_CONFIGURABLE_REGIONS ];
} TaskParameters_t;

typedef struct xMEMORY_REGION {
        void *pvBaseAddress;
        unsigned long ulLengthInBytes;
        unsigned long ulParameters;
} MemoryRegion_t;
```

rvTaskCode ~ uxPriority : 이러한 구조체 맴버는 xTaskCreate()API 함수의 매개변수와 같다. 표준 FreeRTOS task 와 달리 보호된 task 는 사용자(권한이 없는) 또는 관리자(권한이 있는) 모드에서 만들 수 있으며, uxPriority 구조체 맴버는 이 옵션을 제어하는데 사용된다.

사용자 모드에서 작업을 생성하기 위해선 uxPriority 를 생성되는 task 우선순위와 같게 설정한다.

관리자 모드에서는 task 를 생성하기 위해 uxPriority 를 만들 task 의 우선순위와 동일하게 설정하고, 가장 중요한 bit 를 설정한다. 이를 위하여 portPRIVILEGE_BIT 매크로가 제공된다.

예를 들어 우선순위 3 에서 사용자 모드 작업을 만든다면 uxPriority 를 3 으로 설정한다.

우선순위 3 에서 관리자 모드로 task 를 만들려면 uxPriority 를 동일하게(3 | portPRIVILEGE_BIT)로 설정한다.

puxStackBuffer : xTaskCreate()API 함수는 생성되는 task 에서 사용할 스택을 자동으로 할당한다. MPU 를 사용하여 부과된 제한사항은 xTaskCreateRestricted()함수가 동일하게 작업을 수행할 수 없다는 것을 의미하며, puxStackBuffer 매개변수를 이용하여 task 에서 사용하는 스택을 정적으로 할당하여 사용할 수 있다.

제한된 작업이 전환될 때마다(실행 상태로 전환) MPU 는 동적으로 재구성되어 자체 스택에 대한 task 읽기 및 쓰기 엑세스를 제공하는 MPU 영역을 정의한다. 따라서 정적으로 할당된 task 스택은 MPU 에 의해 정의된 크기 및 정렬 제한 사항을 준수해야 한다. 즉, 각각의 영역 크기와 정렬 두 값이 동일해야 한다.

스택 버퍼를 정적으로 선언하면 컴파일러 확장을 사용하여 정렬을 관리할 수 있으며, 링커가 스택 배치를 관리할 수 있기 때문에 효율적으로 작업을 수행할 수 있다. 예를 들어 GCC 를 사용하는 경우 다음 구문을 사용하여 스택을 선언하고 올바르게 정렬할 수 있다.

```
char cTaskStack [ 1024 ] __attribute__ ((align(1024)));
```

MemoryRegion_t 도 구조체이다.

MemoryRegion_t 구조체는 생성되는 task 에서 사용할 단일 MPU 메모리 영역을 정의한다.

Cortex-M3 RTOS-MPU 포트는 portNUM_CONFIGURABLE_REGIONS 를 3 으로 정의한다. 이는 task 가 생성되면 3 개의 영역을 정의할 수 있다는 것이다. region 은 vTaskAllocateMPURegions()함수를 사용하여 런타임을 다시 정의할 수 있다.

pvBaseAddress 와 ulLengthInBytes 맴버는 메모리 영역의 시작과 길이이다.

ulParameters 는 task 가 정의되는 메모리 영역에 엑세스하는 방법을 정의하며 다음 값의 bit OR 을 통해 얻을 수 있다.

- portMPU_REGION_READ_WRITE

- portMPU_REGION_PRIVILEGED_READ_ONLY

- portMPU_REGION_READ_ONLY

- portMPU_REGION_PRIVILEGED_READ_WRITE

- portMPU_REGION_CACHEABLE_BUFFERABLE

- portMPU_REGION_EXECUTE_NEVER

pxCreatedTask : pxCreatedTask 는 생성중인 task 에 대한 핸들을 전달하는데 사용한다. 핸들을 사용할 경우 예를 들어 task 의 우선순위를 변경하거나 task 를 삭제하는 API 호출에서 task 를 참조할 수 있다.

어플리케이션에서 작업 핸들을 사용하지 않으면, pxCreatedTask 를 NULL 로 설정할 수 있다.

1.2.1.3.2 반환 값

pdPASS : task 가 성공적으로 만들어 졌다.

다른 값 : task 데이터 구조를 할당하는데 사용되는 FreeRTOS 힙 메모리가 부족하기 때문에 task 를 설정한 대로 작성하지 못한 것을 나타낸다. heap_1.c, heap_2.c 또는 heap_4.c 가 프로젝트에 포함되어 있으면 사용 가능한 힙의 총량은 FreeRTOSConfig.h 의 configTOTAL_HEAP_SIZE 에 의해 정해지며, 메모리 할당 실패는 vApplicationMallocFailedHook()함수(callback 또는 hook)를 사용하고, 남아있는 힙 메모리 양은 xPortGetFreeHeapSize()함수를 사용해서 확인할 수 있다.

heap_3.c 가 프로젝트에 포함되면 최대 힙 크기는 링커 구성에 의해 정해진다.

1.2.1.3.3 기타

```
/* 보호 작업에 사용할 스택을 선언한다. 스택 정렬은 크기와 일치해야 하며 2 의 제곱으로 한다.
128 단어가 스택에 예약되어 있다면, 128 * 4byte 로 정렬되어야 한다. 이 예제는 GCC 구문을 사용한다 */
static portSTACK_TYPE xTaskStack[ 128 ] __attribute__((aligned(128*4)));

// 생성되는 보호된 task 에 의해 액세스 될 배열을 선언한다. task 는 배열에서 읽기만 가능하고 쓰기는 할 수 없어야 한다.
char cReadOnlyArray[ 512 ] __attribute__((aligned(512)));

// task 를 정의하기 위해 TaskParameters_t 구조체를 채운다. xTaskCreateRestricted()함수에 전달될 구조체다.
static const TaskParameters_t xTaskDefinition = {
        // pvTaskCode, pcName, usStackDepth, pvParameters, uxPriority(1 은 사용자 모드에서 시작한다.), puxStackBuffer
        vTaskFunction, "A task", 128, NULL, 1, xTaskStack, {
```

```
            // xRegions(세 개의 사용자 정의 가능 영역 중 하나만 사용한다) 매개변수는 읽기 전용 영역을 설정하는데 사용한다.
            // 기본 주소의 길이 매개변수이다.
            { cReadOnlyArray, 512, portMPU_REGION_READ_ONLY }, { 0, 0, 0 }, { 0, 0, 0 },
        }
};

void main( void )
{
        // xTaskDefinition 에 정의된 작업을 만든다. 작업 핸들을 사용하지 않으므로 NULL 을 두번째 매개변수로 사용한다.
        xTaskCreateRestricted( &xTaskDefinition, NULL );

        // 스케줄러를 시작한다.
        vTaskStartScheduler();

        // 여기에 도달하면 안된다.
}
```

1.2.1.4 vTaskDelete() : task 제거

```
#include “FreeRTOS.h”
#include “task.h”

void vTaskDelete ( TaskHandle_t pxTask );
```

xTaskCreate() 또는 xTaskCreateStatic()으로 생성된 task 를 삭제한다. 삭제된 task 는 더이상 존재하지 않으므로 실행 중 상태가 될 수 없다. 삭제 된 후, 해당 task 를 참조하는 핸들을 사용하지 않도록 한다.

task 가 삭제되면 해당 task 의 스택 및 데이터 구조를 idle task 가 해제한다. 따라서 어플리케이션에서 vTaskDelete() API 함수를 사용하는 경우 idle task 의 작업 처리 시간이 지연되지 않도록 해야 한다. (idle task 에는 실행 상태의 시간이 할당되어야 한다.)

task 가 삭제되면 커널에 의해 task 에 할당된 메모리만 자동으로 해제된다. 어플리케이션 작업에 할당하는 메모리 또는 기타 리소스는 task 가 삭제될 때 명시적으로 해제해야 한다.

1.2.1.4.1 매개 변수

pxTask : 삭제하는 task 의 핸들이다. task 의 핸들을 얻으려면 xTaskCreate() 일 경우 pxCreatedTask 매개 변수를 사용하거나, xTaskCreateStatic()을 사용하여 작업을 만들고 반환된 값을 이용한다. 아니면 xTaskGetHandle()을 이용한다.

task 는 유효한 task 핸들 대신 NULL 을 전달하여 자체를 삭제할 수 있다.

1.2.1.4.2 기타

```
void vAnotherFunction( void ) {
        // 생성할 task 에서 사용할 핸들을 만든다.
        TaskHandle_t xHandle;

        if( xTaskCreate( vTaskCode, "Demo task", STACK_SIZE, NULL, PRIORITY, &xHandle ) != pdPASS ) {
            // task 데이터 구조와 스택을 할당할 수 있는 FreeRTOS 힙 메모리가 충분하지 않아서 task 생성이 불가능하다.
        } else {
            // 방금 생성한 작업을 삭제한다.
            vTaskDelete( xHandle );
        }
        // 함수를 호출한 task 를 삭제한다.
        vTaskDelete( NULL );
}
```

1.2.2 MPU 설정

1.2.2.1 portSWITCH_TO_USER_MODE() : MPU 권한 변경

```
#include “FreeRTOS.h”
#include “task.h”

void protSWITCH_TO_USER_MODE ( void );
```

이 기능은 FreeRTOSMPU 포트에 관련되어 있다. (메모리 보호 장치를 사용하는 FreeRTOS 포트)

MPU 의 제한된 작업을 생성할 때, xTaskCreateRestricted()를 이용한다. xTaskCreateRestricted()에 제공되는 매개 변수는 작성중인 task 가 사용자(비공유) 모드인지, 관리자 모드 task 인지 지정한다. 관리자 모드의 task 는 해당 함수를 호출하여 사용자 모드 task 로 변환할 수 있다.

1.2.2.1.1 기타

task 가 portSWITCH_TO_USER_MODE()함수를 통하여 유저 모드로 전환을 하게 되면, 관리자 모드와 동등한 권한이 없다.

1.2.2.2 vTaskAllocateMPURegions() : MPU 메모리 영역 설정

```
#include “FreeRTOS.h”
#include “task.h”

void vTaskAllocateMPURegions ( TaskHandle_t xTaskToModify, const MemoryRegion_t * const xRegions );
```

이 기능은 FreeRTOSMPU 포트에 관련되어 있다. (메모리 보호 장치를 사용하는 FreeRTOS 포트)

MPU 의 제한된 작업에서 사용할 MPU(메모리 보호 장치)영역 집합을 정의한다.

MPU 에서 제어하는 메모리 영역은 작업이 생성될 때 xTastSreattSestricted()함수를 사용하여 MPU 에서 제한된 작업에 할당할 수 있다.

그 뒤 VtaskAllocateMPURegions()함수를 사용하여 런타임에 다시 정의하거나 재할당할 수 있다.

1.2.2.2.1 매개 변수

xTaskToModify : 제한된 작업의 핸들이 수정된다. (xRegions 매개변수로 정의된 메모리 영역에 대해서 엑세스가 제공되는 작업) task 의 핸들은 xTaskCreateRestricted()API 함수의 pxCreatedTask 매개 변수를 가져온다. task 는 유효한 핸들 대신 NULL 을 전달하여 자체 메모리 영역 엑세스 정의를 수정할 수 있다.

xRegions : MemoryRegion_t 구조체의 배열이다. 배열의 형태는 포트 NUM_CONFIGURABLE_REGIONS 에 의하여 정의된다.

배열의 각 MemoryRegion_t 구조체는 xTaskToModify 매개 변수가 참조하는 작업에서 사용할 단일 MPU 메모리 영역을 정의한다.

1.2.2.2.2 기타

```
typedef struct xMEMORY_REGION
{
    void *pvBaseAddress;
    unsigned long ulLengthInBytes;
    unsigned long ulParameters;
} MemoryRegion_t;
```

pvBaseAddress 및 ulLengthInBytes 멤버는 메모리 영역의 시작과 메모리 영역의 길이다. 이들은 MPU 에 의해 부과된 크기 및 정렬 제한을 준수해야 하며, 특히 각 영역의 크기와 정렬은 정해진 값과 같아야 한다.

ulParameters 는 태스크가 정의되는 메모리 영역에 액세스하는 방법을 정의하며 다음 값의 비트를 OR 한다.

- portMPU_REGION_READ_WRITE

- portMPU_REGION_PRIVILEGED_READ_ONLY

- portMPU_REGION_READ_ONLY

- portMPU_REGION_PRIVILEGED_READ_WRITE

- portMPU_REGION_CACHEABLE_BUFFERABLE

- portMPU_REGION_EXECUTE_NEVER

1.2.3 Tag

1.2.3.1 vTaskSetApplicationTaskTag : task 에 태그 값 할당

```
#include “FreeRTOS.h”
#include “task.h”

void vTaskSetApplicationTaskTag ( TaskHandle_t xTask, TaskHookFunction_t pxTagValue );
```

vTaskSetApplicationTaskTag() API 함수를 이용하여 태그 값을 task 에 할당할 수 있다. 태그 값의 의미와 사용은 응용 프로그램 작성자가 정의한다. 커널은 일반적으로 태그 값을 액세스 하지 않는다.

태그 값은 함수 포인터를 보유하는데 사용될 수 있다. 이 작업이 완료되면 xTaskCallApplicationTaskHook() 함수를 사용하여 태그 값에 할당된 함수를 호출할 수 있다. 이 기술은 실제 콜백 함수를 task 에 할당한다. 이러한 콜백은 traceTASK_SWITCHED_IN() 매크로와 함께 사용되어 실행 추적 기능을 구현하는 것이 일반적이다.

vTaskSetApplicationTaskTag()를 사용하려면 FreeRTOSConfig.h 에서 configRequest_APPLICATION_TASK_TAG 를 1 로 설정해야 한다.

1.2.3.1.1 매개 변수

xTask : 태그 값을 할당할 task 의 핸들이다. NULL 을 사용하여 호출한 task 의 태그 값을 할당할 수 있다.

pxTagValue : task 의 태그 값으로 할당되는 값이다. 이것은 태그에 포인터를 할당할 수 있도록 허용하는 TaskHookFunction_t 유형이지만, 간접적으로 태그 값은 모든 유형이 될 수 있다.

1.2.3.1.2 기타

```
// 이 예에서는 task 태그 값으로 정수가 설정된다.
void vATask( void * pvParameters ) {
    // 현재 실행중인 task 에 태그 값 1 을 할당한다. (void *)는 컴파일러 경고를 방지하는데 사용한다.
    vTaskSetApplicationTaskTag( NULL, ( void * ) 1 );

    for( ;; ) {
      // 나머지 코드를 입력한다.
    }
}
// 콜백 함수가 task 태그로 지정하며, 콜백 함수를 정의한다. 이 예에서는 TaskHookFunction_t 유형을 사용한다.
static BaseType_t prvExampleTaskHook( void * pvParameter ) {
    // 일부 작업을 수행한다. 값 기록, 작업 상태 업데이트, 값 출력 등의 작업일 수 있다.
    return 0;
}
// prvExampleTaskHook()을 hook / 태그 값으로 설정하는 작업을 정의한다. 이것이 실제 task 콜백 함수를 등록한다.
void vAnotherTask( void * pvParameters ) {
    // 현재 실행중인(호출중인) task 를 위한 콜백 함수를 등록한다.
    vTaskSetApplicationTaskTag( NULL, prvExampleTaskHook );

    for( ;; ) {
      // 작업 코드를 입력한다.
    }
}

/* hook(콜백) 사용의 예. hookTASK_SWITCHED_OUT()매크로를 정의하여 hook 함수를 호출한다. 그 후, 커널은 task 가 전환될
때마다 자동으로 task hook 를 호출한다. 이 기술을 사용하여 실행 추적을 생성할 수 있다. pxCurrentTCB 는 현재 실행중인 task 를
참조한다. */
#define traceTASK_SWITCHED_OUT() xTaskCallApplicationTaskHook( pxCurrentTCB, 0 )
```

1.2.3.2 xTaskCallApplicationTaskHook() : task 의 태그 사용

```
#include “FreeRTOS.h”
#include “task.h”

BaseType_t xTaskCallApplicationTaskHook ( TaskHandle_t xTask, void *pvParameters );
```

태그 값을 사용하여 task hook(또는 callback) 함수를 task 에 연결할 수 있다. 이 작업이 완료되면 xTaskCallApplicationTaskHook()함수를 사용하여 hook 함수가 호출된다. task hook 기능은 목적에 상관없이 사용할 수 있다.

태그 값은 vTaskSetApplicationTaskTag()함수를 사용하여 작업을 할당할 수 있다. 태그 값의 의미와 사용은 작성자가 정의하도록 한다. 일반적으로 커널 자체는 태그 값에 액세스하지 않는다.

task hook 함수는 다음과 같은 프로토 타입을 가져야 한다.

```
BaseType_t xAnExampleTaskHookFunction (void *pvParameters );
```

xTaskCallApplicationTaskHook()함수는 FreeRTOSConfig.h 에서 configUSE_APPLICATION_TASK_TAG 가 1 로 설정된 경우에만 사용이 가능하다.

1.2.3.2.1 매개 변수

xTask : 관련 후크 기능이 호출되는 task 핸들이다. 핸들을 만들기 위해서는 xTaskCreate()를 사용하거나, pxCreatedTask 매개 변수를 사용하거나 xTaskCreateStatic()을 사용하여 작업을 만들고 반환된 값을 저장하거나 xTaskGetHandle()를 호출할 때 작업의 이름을 사용한다.

task 는 유효한 task 핸들 대신 NULL 을 전달하여 자체 hook 기능을 호출할 수 있다.

pvParameters : task hook 함수 자체에 대한 매개 변수로 사용되는 값이다. 이 매개 변수에는 작업 hook 함수 매개 변수를 허용하고, 유형에 관계없도록 하기 위하여 void *를 사용하고 있다.

예를 들어 정수 타입은 hook 함수가 호출된 곳에서 void *로 정수를 받은 뒤, 그것을 hook 함수에 전달한다.

1.2.3.2.2 반환 값

사용자가 지정한다.

1.2.3.2.3 기타

예제는 디버그 추적 정보를 출력하는데 사용되는 task hook 을 보여준다.

```
// task hook 함수의 프로토 타입을 이용하여 hook(callback)함수를 정의한다.
static BaseType_t prvExampleTaskHook ( void * pvParameter ) {

	// 사용자가 원하는 동작을 작성한다. 해당 예제에서는 디버그 추적을 예시로 하고 있다.
	vWriteTrace( pxCurrentTCB );

	// 해당 예제에서는 hook 리턴 값을 사용하지 않으므로 0 을 리턴 하도록 한다.
	return 0;
}
```

```
// 태그 값을 사용하는 예제 task 이다.
void vAnotherTask ( void *pvParameters ) {

        // vTaskSetApplicationTaskTag()는 task 와 연관된 태그 값을 설정한다.
        // task 핸들 대신 NULL 을 설정하여 호출하는 task 의 태그 값을 나타낸다.
        vTaskSetApplicationTaskTag ( NULL, prvExampleTaskHook );

        for( ;; ) {   // 작업할 코드의 위치이다.
        }
}

// traceTASK_SWITCHED_OUT()함수를 매크로로 정의하여 스위치 아웃 된 각 task 의 hook 기능을 호출한다.
// pxCurrentTCB 는 현재의 핸들을 나타낸다.
#define traceTASK_SWITCHED_OUT() xTaskCallApplicationTaskHook( pxCurrentTCB, 0 )
```

1.2.3.3 xTaskGetApplicationTaskTag : 태그 값 출력

```
#include “FreeRTOS.h”
#include “task.h”

TaskHookFunction_t xTaskGetApplicationTaskTag ( TaskHandle_t xTask );
```

task 와 관련된 태그 값을 반환한다.

태그 값의 의미와 사용은 응용 프로그램 작성자가 정의한다. 일반적으로 커널 자체는 태그 값에 액세스 하지 않는다.

태그 값은 함수 포인터를 보유하는데 사용할 수 있다. 이 작업이 완료되면 xTaskCallApplicationTaskHook() API 함수를 사용하여 태그 값에 할당된 함수를 호출할 수 있다. 이 기술은 실제 콜백 함수를 task 에 할당하고 있으며, 이러한 콜백을 traceTASK_SWITCHED_IN()매크로와 함께 사용하여 구현하는것이 일반적이다.

xTaskGetApplicationTaskTag()를 사용하려면 configRes0_Config.h 에 있는 configUSE_APPLICATION_TASK_TAG 를 1 로 설정해야 한다.

1.2.3.3.1 매개 변수

xTask : 사용하고자 하는 task 의 핸들이다. task 에 핸들 대신 NULL 을 사용하여 자체 태그 값을 얻을 수도 있다.

1.2.3.3.2 반환 값

사용하고자 하는 task 의 태그 값이다.

1.2.3.3.3 기타

```
// 해당 예제는 작업 태그 값을 정수로 설정한다.
void vATask( void *pvParameters ) {
        // 현재 실행중인 작업에 태그 값으로 1 을 할당한다. (void *)는 컴파일러 경고를 방지하는데 사용된다.
        vTaskSetApplicationTaskTag( NULL, ( void * ) 1 );

        for( ;; ) {
            // 나머지 동작을 추가한다.
        }
}
```

```
void vAFunction( void ) {
        TaskHandle_t xHandle;
        long lReturnedTaskHandle;

        // 예제에 사용하고자 하는 task 를 생성한다.
        // task 를 만든다. ( 동작 함수 포인터, 작업 이름, task 스택 크기, task 의 매개 변수, task 우선순위, task 핸들 )
        if( xTaskCreate( vATask, "Demo task", STACK_SIZE, NULL, TASK_PRIORITY, &xHandle ) == pdPASS ) {
            // 작업이 성공적으로 생성되었다. 작업이 실행되도록 잠시 지연하도록 한다.
            vTaskDelay( 100 );

            // task 에 할당된 태그 값을 받는다.
            // 반환된 태그 값은 정수로 저장되었기 때문에 컴파일러 경고를 방지하기 위해 정수로 받는다.
            lReturnedTaskHandle = ( long ) xTaskGetApplicationTaskTag( xHandle );
        }
}
```

1.2.4 task 핸들 및 이름 출력

1.2.4.1 xTaskGetCurrentTaskHandle : 실행 상태의 task 핸들 출력

```
#include “FreeRTOS.h”
#include “task.h”

TaskHandle_t xTaskGetCurrentTaskHandle ( void );
```

실행 상태인 task 의 핸들을 출력한다. 이것은 xTaskGetCurrentTaskHandle()를 호출한 task 의 핸들이다.

xTaskGetCurrentTaskHandle()를 사용하려면 FreeRTOSConfig.h 의 INCLUDE_xTaskGetCurrentTaskHandle 을 1 로 설정해야 한다.

1.2.4.1.1 반환 값

xTaskGetCurrentTaskHandle()를 호출한 task 의 핸들이다.

1.2.4.2 xTaskGetIdleTaskHandle : Idle task 의 핸들 출력

```
#include “FreeRTOS.h”
#include “task.h”

TaskHandle_t xTaskGetIdleTaskHandle ( void );
```

idle task 와 연결된 task 의 핸들을 반환한다. idle task 는 스케줄러가 시작될 때 자동으로 작성된다.

xTaskGetCurrentTaskHandle()를 사용하려면 FreeRTOSConfig.h 의 INCLUDE_xTaskGetIdleTaskHandle 을 1 로 설정해야 한다.

1.2.4.2.1 반환 값

idle task 의 핸들이다.

1.2.4.3 xTaskGetHandle : task 이름을 통한 핸들 출력

```
#include “FreeRTOS.h”
#include “task.h”

TaskHandle_t xTaskGetHandle ( const char * pcNameToQuery );
```

task 생성 시 pcName 이라는 매개 변수를 가지고 있다. xTaskGetHandle()은 task 의 이 이름을 보고 task 핸들을 반환한다.

xTaskGetHandle()을 완료하는데 비교적 오랜 시간이 걸릴수도 있다. 따라서 xTaskGetHandle()은 각 task 이름에 한번만 사용하는 것이 좋다. xTaskGetHandle()에 의해 반환된 task 핸들은 나중에 다시 사용할 수 있도록 저장할 수 있다.

동일한 이름을 가진 둘 이상의 task 가 있을 경우 xTaskGetHandle()이 동작하지 않는다.

xTaskGetHandle()을 사용하려면 FreeRTOSConfig.h 에서 INCLUDE_xTaskGetHaandle 을 1 로 설정해야 한다.

1.2.4.3.1 매개 변수

pcNameToQuery : task 의 이름이다. 이 이름은 표준 NULL 로 끝나는 C 문자열로 지정되어 있다.

1.2.4.3.2 반환 값

task 가 pcNameToQuery 매개 변수로 지정된 것과 정확히 같은 이름을 보유할 경우 task 핸들이 리턴 된다. 지정된 이름을 가진 task 가 없을 경우 NULL 을 리턴한다.

1.2.4.3.3 기타

```
void vATask( void * pvParameters ) {
        const char * pcNameToLookup = "MyTask";
        TaskHandle_t xHandle;

        // 이름이 MyTask 인 task 핸들을 찾아 반환된 핸들을 나중에 다시 사용할 수 있도록 로컬에 저장한다.
        xHandle = xTaskGetHandle( pcNameToLookup );

        if( xHandle != NULL ) {
            // 작업 핸들이 발견될 경우, TaskHandle_t 매개 변수를 사용하는 다른 FreeRTOS API 함수들을 사용할 수 있다.
        }

        for( ;; ) {
            // 나머지 task 코드들은 여기에 위치한다.
        }
}
```

1.2.4.4 pcTaskGetName : task 텍스트 이름 출력

```
#include “FreeRTOS.h”
#include “task.h”

char * pcTaskGetName ( TaskHandle_t xTaskToQuery );
```

사람이 읽을 수 있는 task 의 텍스트 이름을 출력한다. 텍스트 이름은 task 생성시 pcName 매개 변수를 통해 task 에 지정된다.

1.2.4.4.1 매개 변수

xTaskToQuery : 출력하고자 하는 task 의 핸들이다. 핸들 대신 NULL 입력 시, 호출한 해당 task 의 텍스트 이름을 출력한다.

1.2.4.4.2 반환 값

task 이름은 표준 NULL 로 종료되는 C 문자열이다. 반환된 값은 task 이름에 대한 포인터이다.

1.2.5 vTaskSetThreadLocalStoragePointer : TLS 에 데이터 입력

```
#include “FreeRTOS.h”
#include “task.h”

void vTaskSetThreadLocalStoragePointer ( TaskHandle_t xTaskToSet, BaseType_t xIndex, void * pvValue );
```

스레드 로컬 저장소(TLS)를 사용하여 응용 프로그램 작성자가 작업 제어 블록 내에 값을 저장하고, task 자체에 고유한 값을 만들어 각 task 가 고유한 값을 가질 수 있다. 각 task 에는 스레드 로컬 저장소로 사용할 수 있는 자체 포인터 배열이 있다.

배열의 인덱스 수는 FreeRTOSConfig.h 의 configNUM_THREAD_LOCAL)STORAGE_POINTERS 컴파일 시간 구성 상수에 의해 설정된다. vTaskSetThreadLocalStoragePointer()는 배열의 인덱스 값을 설정하여 효과적으로 스레드 로컬 값을 저장한다.

1.2.5.1 매개 변수

xTaskToSet : 스레드 로컬 데이터가 쓰여지는 task 의 핸들이다.

xIndex : 데이터가 입력되는 스레드 로컬 저장소 배열에 대한 인덱스다.

pvValue : xIndex 로 지정된 인덱스에 값을 입력한다.

1.2.5.2 기타

```
uint32_t ulVariable;

/* 스레드 로컬 저장소 배열의 인덱스 1 에 32 비트 0x12345678 값을 입력한다. task 핸들이 NULL 이기 때문에 호출하는 task 의 스레드 로컬 저장소 배열에 쓰여진다. */
vTaskSetThreadLocalStoragePointer( NULL, 1, ( void * )0x12345678 );

// 32 비트 변수 ulVariable 의 값을 호출 task 의 스레드 로컬 스토리지 배열 인덱스 0 에 저장한다.
ulVariable = ERROR_CODE;

vTaskSetThreadLocalStoragePointer( NULL, 0, ( void * )&ulVariable );
```

1.2.6 task 상태

1.2.6.1 eTaskGetState : task 상태 출력

```
#include “FreeRTOS.h”
#include “task.h”

eTaskState eTaskGetState ( TaskHandle_t pxTask );
```

eTaskGetState()가 호출되었을 때 task 의 상태를 열거형으로 반환한다.

eTaskGetState()를 사용하려면 FreeRTOSConfig.h 의 INCLUDE_eTaskGetState 를 1 로 설정해야 한다.

1.2.6.1.1 매개 변수

pxTask : 확인하고자 하는 task 의 핸들이다.

1.2.6.1.2 반환 값

eRunning : 실행 상태

eReady : 준비 상태

eBlocked : 차단 상태

eSuspended : Suspended 상태

eDeleted : 삭제 대기 상태 (task 구조 정리 대기 중)

1.2.6.2 vTaskDelay() : tick 인터럽트 기반 차단 상태

```
#include “FreeRTOS.h”
#include “task.h”

void vTaskDelay ( TickType_t xTicksToDelay );
```

vTaskDelay()를 호출하는 task 에 대해 정해진 값 동안, tick 인터럽트가 발생할 때까지 차단 상태로 설정한다.

tick 의 지연시간을 0 으로하면 호출한 task 가 차단상태가 되지 않고, 이 task 와 우선순위를 공유하는 모든 것들은 준비 상태로 변경한다. 이때 vTaskDelay(0)은 taskYIELD()를 호출하는 것과 같다.

1.2.6.2.1 매개 변수

xTicksToDelay : 호출하는 task 를 준비 상태로 다시 전환하기 전에 차단 상태로 유지하는 tick 인터럽트의 수다. 예를 들어 tick 이 10,000 일 경우, vTaskDelay(100)을 호출하면 차단 상태로 변경되고 tick 계수가 10,100 에 도달할 때까지 차단 상태를 유지한다. vTaskDelay()가 호출되면 tick 인터럽트를 발생시켜서, tick 의 최대 값이 될 때까지 계산한다. 즉, 딜레이 시간을 지정할 때 적용할 수 있는 최대 시간 분해능은 1 개의 tick 인터럽트발생 시간과 같다. 매크로 pdMS_to_TICKS()를 사용하여 밀리초로 전환할 수 있다.

vTaskDelay() API 함수를 사용하려면 INCLUDE_vTaskDelay 를 FreeRTOSConfig.h 의 값을 1 로 설정해야 한다.

1.2.6.2.2 기타

```
void vAnotherTask( void * pvParameters ) {
        for( ;; ) {

            // 처리 과정을 수행한다.

            // 20 개의 tick 인터럽트 발생동안 차단 상태로 들어간다. 실제 소요 시간은 tick 주파수에 의존한다.
            vTaskDelay( 20 );

            // 차단 상태를 20 밀리초 동안 유지한다.
            // pdMS_TO_TICKS() 매크로를 사용하면 tick 빈도가 차단 상태에서 소비된 시간에 영향을 주지 않고 변경된다.
            vTaskDelay( pdMS_TO_TICKS( 20 ) );
        }
}
```

1.2.6.3 vTaskDelayUntil() : 일정한 주기간 차단 상태

```
#include “FreeRTOS.h”
#include “task.h”

void vTaskDelayUntil ( TickType_t * pxPreviousWakeTime, TickType_t xTimeIncrement );
```

정해진 시간에 도달할 때까지 vTaskDelayUntil()을 호출하는 task 를 차단 상태로 만든다. 주기적인 task 는 vTaskDelayUntil()을 사용하여 실행 빈도를 일정하게 유지할 수 있다.

vTaskDelay()와 vTaskDelayUntil()의 차이점

vTaskDelay()는 호출한 시간부터 지정된 tick 의 수만큼 호출 task 를 차단 상태로 변경한다. 해당 task 가 차단 상태를 종료하는 시간은 해당 함수가 호출된 시점과 관련된다.

vTaskDelayUntil()은 호출 task 가 절대 시간에 도달할 때까지 차단 상태로 변경한다. 해당 task 는 호출된 시간이 아닌 특정 시간에 차단 상태를 정확히 종료한다.

vTaskDelay()API 함수를 사용하려면 FreeRTOSConfig.h 에서 INCLUDE_vTaskUntil 을 1 로 설정해야 한다.

1.2.6.3.1 매개 변수

pxPreviousWakeTime : 해당 매개 변수는 vTaskDelayUntil()가 일정한 빈도로 실행되는 작업을 구현하는데 사용된다. 이 경우 pxPreviousWakeTime 은 작업이 마지막으로 차단 상태에서 벗어난 시간을 유지한다. 이 시간은 task 가 다음 차단 상태에서 벗어나는 시간을 계산하기 위한 참조점으로 사용된다.

pxPreviousWakeTime 이 가리키는 변수는 해당 함수 내에서 자동으로 업데이트 된다. 변수가 처음 초기화 될 때를 제외하고 일반적으로 어플리케이션 코드에 의해 수정되지 않는다.

xTimeIncrement : 이 매개 변수는 빈도를 나타내며, 고정 주파수로 주기적으로 실행되는 task 를 구현하는데 사용된다. xTimeIncrement 는 tick 으로 지정된다. pdMS_TO_TICKS() 매크로는 밀리초를 tick 으로 변환하는데 사용할 수 있다.

1.2.6.3.2 기타

```
// 해당 예제는 50 밀리 초마다 작동하는 task 이다.
void vCyclicTaskFunction( void * pvParameters ) {
        TickType_t xLastWakeTime;
        const TickType_t xPeriod = pdMS_TO_TICKS( 50 );

        // xLastWakeTime 변수는 현재 tick 수로 초기화 해야한다. 이것은 변수가 명시적으로 쓰여지는 유일한 시간이다.
        // 이 할당 후, xLastWakeTime 은 vTaskDelayUntil() 내부에서 자동적으로 업데이트된다.
        xLastWakeTime = xTaskGetTickCount();

        // 함수를 동작을 정의하는 루프를 입력한다.
        for( ;; ) {
        // 해당 작업은 50 밀리 초마다 실행된다. 이때 시간은 tick 단위로 측정된다.
        // pdMS_TO_TICKS 매크로를 통해 밀리 초를 tick 으로 변환하는데 사용되었다.
        vTaskDelayUntil( &xLastWakeTime, xPeriod );

        // 수행 작업을 작성한다.
        }
}
```

1.2.6.4 xTaskCheckForTimeOut() : 특정 시간까지 차단 상태로 이벤트 대기

```
#include “FreeRTOS.h”
#include “task.h”

BaseType_t xTaskCheckForTimeOut ( TimeOut_t * const pxTimeOut, TickType_t * const pxTicksToWait );
```

task 는 차단 상태로 전환하여 이벤트를 대기할 수 있다.

일반적으로 task 차단 상태를 만드는데 있어서 무기한 대기하지 않고 시간 초과를 지정한다. task 가 차단 상태에서 이벤트가 발생하기 전에 제한시간이 만료하면 task 는 차단 상태에서 해제된다. task 는 이벤트가 발생하기를 기다리는 동안 task 가 차단 상태로 두 번 이상 들어가고 나오는 경우, task 는 차단 상태로 전환될 때마다 사용된 시간 초과를 조정해야 한다. 이때 차단 상태에서 소요된 시간의 합계가 원래 지정된 시간 초과를 하지 않아야 한다. (차단이 해제될 경우 타이머가 다시 처음부터 작동하기 때문이다.)

xTaskCheckForTimeOut()를 사용함에 있어서 tick 카운트 오버 플로우와 같은 상황을 고려하여 조정을 해야 한다. 그렇지 않으면 수동 조정으로 인해 오류가 발생할 수 있다.

xTaskCheckForTimeOut()는 vTaskSetTimeOutState()와 함께 사용된다.

vTaskSetTimeOutState()를 호출하여 초기 조건을 설정한 후, xTaskCheckForTimeOut()을 시간 초과 상태를 확인하며, 남아있는 블록 시간이 시간초과가 발생하지 않도록 조정한다.

1.2.6.4.1 매개 변수

pxTimeOut : 시간 초과가 발생했는지를 결정하는데 필요한 정보를 담고있는 구조체 포인터이다. pxTimeOut 은 vTaskSetTimeOutState()를 사용하여 초기화 할 수 있다.

pxTicksToWait : 조정된 블록 시간을 전달하는데 사용되며, 이미 차단 상태에서 보낸 시간을 고려한 후 남은 블록시간이다.

1.2.6.4.2 반환 값

pdTRUE 가 리턴 되면 블록시간이 남아있지 않고 시간 초과가 발생한 상태이다.

pdFALSE 가 반환되면 블록시간이 남아있으므로 시간 초과가 발생하지 않은 상태이다.

1.2.6.4.3 기타

```
/* UART 인터럽트로 채워진 Rx 버퍼에서 uxWantedBytes 를 수신하는데 사용되는 라이브러리 드라이버 함수이다. Rx 버퍼에 바이트가
충분하지 않으면, 데이터양이 더 저장될 때까지 task 는 차단 상태가 된다. 그래도 데이터가 충분하지 않을 경우 task 차단 상태에서
다시 시도를 하고, xTaskCheckForTimeOut()를 이용하여 차단 시간을 MAX_TIME_TO_WAIT 을 초과하지 않도록 다시 계산하여
사용한다. 버퍼에 적어도 uxWantedBytes 가 포함되어 있거나 차단 상태에서 소비된 총 시간이 MAX_TIME_TO_WAIT 에 도달할
때까지 작업을 계속한다. 이때 작업은 최소 uxWantedBytes 이상이 될 때까지 가능한 많은 바이트를 읽은다. */
size_t xUART_Receive ( uint8_t *pucBuffer, size_t uxWantedBytes )
{
        size_t uxReceived = 0;
        TickType_t xTicksToWait = MAX_TIME_TO_WAIT;
        TimeOut_t xTimeOut;

        // xTimeOut 을 초기화 한다. 이것은 입력한 시간을 기록한다.
        vTaskSetTimeOutState ( &xTimeOut );

        // 버퍼가 원하는 수의 바이트를 포함할 때까지 반복하거나, 시간 초과가 발생한다.
        while( UART_bytes_in_rx_buffer ( pxUARTInstance ) < uxWantedBytes ) {

            /* 버퍼에 충분한 데이터가 없으면 작업이 차단 상태가 된다. 블록 상태에서 소비한 총 시간이 MAX_TIME_TO_WAIT 를
             초과하지 않도록 함수 내에서 블록 상태로 소비된 시간을 고려하여 xTicksToWait 을 조정한다. */
            if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) != pdFALSE ) {
                // 원하는 바이트 수가 사용 가능하기 전에 시간 초과하였다. 루프를 종료한다.
                break;
            }
            // 더 많은 데이터가 버퍼 안에 존재할 때까지 최대 xTicksToWait tick 을 기다린 뒤, 이것을 수신 인터럽트가 알려준다.
            ulTaskNotifyTake ( pdTRUE, xTicksToWait );
        }
        // uxWantedBytes 를 수신 버퍼에서 pucBuffer 로 읽는다.
        // 읽은 실제 바이트 수(uxWantedBytes 보다 작을 수 있다.)가 리턴된다.
        uxReceived = UART_read_from_receive_buffer ( pxUARTInstance, pucBuffer, uxWantedBytes );
        return uxReceived;
}
```

1.2.6.5 xTaskAbortDelay() : 차단 상태 해제

```
#include “FreeRTOS.h”
#include “task.h”

BaseType_t xTaskAbortDelay ( TaskHandle_t xTask );
```

시간 초과 매개 변수가 포함된 API 함수를 호출하면 호출 작업이 차단 상태로 전환될 수 있다.

차단 상태에 있는 작업은 특정 시간까지 대기하거나 이벤트가 발생할 때까지 시간 초과로 대기한 후, 작업은 자동으로 준비 상태로 변한다.

이러한 동작에 대한 예는 다음과 같다.

- task 가 vTaskDelay()를 호출하면 함수의 매개 변수로 지정된 시간 제한이 경과할 때까지 차단 상태가 된다. 시간 제한 이후 task 는 자동으로 차단 상태에서 준비 상태가 된다.

- task 는 ulTaskNotifyTake()의 값이 0 일 때나 함수의 매개 변수 중 하나에 의해 지정된 시간을 초과하기 전까지 차단 상태로 존재한다. 시간이 지난 후 task 는 자동으로 차단 상태에서 준비 상태가 된다.

작업이 차단 상태일 경우, 스케줄러에서 사용할 수 없기 때문에 처리 시간을 소비하지 않는다.

xTaskAbortDelay()를 사용할 수 있으려면 FreeRTOSConfig.h 에서 INCLUDE_xTaskAbortDelay 를 1 로 설정해야 한다.

1.2.6.5.1 매개 변수

xTask : 차단 상태에서 변경할 task 의 핸들이다. 핸들을 만들기 위해서는 xTaskCreate()를 사용하거나, pxCreatedTask 매개 변수를 사용하거나 xTaskCreateStatic()을 사용하여 작업을 만들고 반환된 값을 저장하거나 xTaskGetHandle()를 호출할 때 작업의 이름을 사용한다.

1.2.6.5.2 반환 값

xTask 에 의해 참조된 task 가 차단 상태에서 해제가 되면 pdPASS 가 리턴된다. xTask 가 참조하는 작업이 차단상태가 아니기 때문에 차단 상태에서 제거되지 않는 경우 pdFAIL 이 반환된다.

1.2.6.5.3 기타

```
void vAfunction ( TaskHandle_t xTask )
{
    // 차단 상태에 따라 코드가 동작하도록 한다.
    if ( xTaskAbortDelay ( xTask ) == pdFAIL )
    {

    } else {

    }
}
```

1.2.6.6 vTaskSuspend : task 를 일시 중지 상태로 설정

```
#include “FreeRTOS.h”
#include “task.h”

void vTaskSuspend ( TaskHandle_t pxTaskToSuspend );
```

task 를 Suspended 상태로 설정한다. Suspended 상태의 task 는 절대 종료되지 않는다.

Suspended 상태에서 task 를 제거하는 유일한 방법은 vTaskResume()호출의 대상으로 지정하는 것이다.

FreeRTOS 6.1.0 이상을 사용하면 vTaskStartScheduler()가 호출되기 전에 vTaskSuspend()를 호출하여 스케줄러가 시작되기 전에 task 를 Suspended 상태로 설정할 수 있다. 그러면 Suspended 상태에서 task 가 실행된다.

1.2.6.6.1 매개 변수

pxTaskToSuspend : 일시 중단하고자 하는 task 의 핸들이다. NULL 입력 시 호출하는 task 가 일시 중단된다.

1.2.6.6.2 기타

```
void vAFunction( void ) {
    TaskHandle_t xHandle;

    // 생성된 task 의 핸들을 xHandle 에 저장하고 task 를 만든다.
    if( xTaskCreate( vTaskCode, "Demo task", STACK_SIZE, NULL, PRIORITY, &xHandle ) != pdPASS ) {
      // task 가 생성 실패되었다.
    } else {
      // 생성된 task 의 핸들을 사용하여 task 를 Suspended 상태로 만든다.
      // FreeRTOS 6.1.0 부터 스케줄러가 시작되기 전에 이 작업을 수행할 수 있다.
      vTaskSuspend( xHandle );
```

```
        // 다른 task 가 vTaskResume( xHandle )을 호출하지 않는 한 생성된 task 는 이 기간동안 실행되지 않는다.
        // NULL 매개 변수를 사용하여 호출하는 task 를 일시 중단한다.
        vTaskSuspend( NULL );

        // 이 task 는 다른 task(중단되지 않은)을 다시 시작하는 경우에만 vTaskResume()호출을 사용하여 실행할 수 있다.
    }
}
```

1.2.6.7 vTaskResume : Suspended 상태를 Ready 상태로 전환

```
#include “FreeRTOS.h”
#include “task.h”

void vTaskResume ( TaskHandle_t pxTaskToResume );
```

Suspended 상태에서 Ready 상태로 task 를 전환한다. vTaskSuspend()에 대한 호출이며, Suspended 상태에 있어야 한다.

task 는 시간 초과를 지정하여 대기열 이벤트를 기다리는 task 를 차단할 수 있다. 이러한 차단된 task 를 vTaskSuspend()에 대한 호출을 사용하여 Suspended 상태로 이동한 다음 vTaskResume()호출을 사용하여 Suspended 상태를 벗어나 준비상태로 진행되는 것이 정상적인 동작 방법이다. 이 시나리오에 따라 task 가 다음에 실행 상태가 되면 해당 제한 시간이 만료되었는지 여부를 확인한다. 시간 종료 기간이 만료되지 않은 경우 task 는 원래 지정된 시간 초과 기간의 나머지 시간 동안 대기열 이벤트를 대기하는 차단 상태가 된다.

vTaskDelay() 또는 vTaskDelayUntil()API 함수를 사용하여 일시적인 이벤트를 기다리는 task 도 차단할 수 있다. 이러한 차단된 task 도 vTaskSuspend()에 대한 호출을 사용하여 Suspended 상태로 이동한 다음 vTaskResume()을 이용하여 준비상태로 이동시킬 수 있다. 이 시나리오에 따라 다음 task 가 실행 상태가 되면 다음과 같이 지정된 지연 기간이 만료된 것처럼 vTaskDelay() 또는 vTaskDelayUntil()함수를 종료한다. 실제로는 그렇지 않더라도 종료된다.

vTaskResume()은 실행중인 task 에서 호출되어야 하므로 스케줄러 초기화 상태(스케줄러가 시작되기 전)에는 호출하면 안된다.

1.2.6.7.1 매개 변수

pxTaskToResume : 준비 상태로 전환하고자 하는 task 의 핸들이다.

1.2.6.7.2 기타

```
void vAFunction( void ) {
        TaskHandle_t xHandle;

        // task 를 생성하고, 핸들을 저장한다.
        if( xTaskCreate( vTaskCode, "Demo task", STACK_SIZE, NULL, PRIORITY, &xHandle ) != pdPASS ) {
            // task 생성에 실패했다.
        } else {
            // 핸들을 사용하여 task 를 일시 중지 상태로 전환한다.
            vTaskSuspend( xHandle );

            // 다른 task 가 vTaskResume(xHandle)을 호출하지 않는 한 일시 중지된 task 는 실행되지 않는다.
            // 일시 중단된 task 를 다시 시작한다.
            vTaskResume( xHandle );

            // 생성된 task 를 다시 스케줄러에서 사용할 수 있으며 Running 상태로 들어갈 수 있다.
        }
}
```

1.2.6.8 xTaskResumeFromISR : ISR 에서의 vTaskResume()

```
#include “FreeRTOS.h”
#include “task.h”

BaseType_t xTaskResumeFromISR ( TaskHandle_t pxTaskToResume );
```

ISR 에서 호출할 수 있는 vTaskResume()이다.

xTaskResumeFromISR()는 인터럽트에서 호출할 수 있지만 vTaskResume()는 인터럽트에서 호출 할 수 없다. xTaskResumeFromISR()는 task 의 인터럽트와 동기화 하는데 사용되면 안된다. 이렇게 할 경우 인터럽트 이벤트가 연관된 task 레벨 처리 기능의 실행보다 빠르게 발생하면서 인터럽트 이벤트가 누락된다.

세마포어가 이벤트를 잠그기 때문에 이진 또는 카운터 세마포어를 사용하여 task 및 인터럽트 동기화를 안전하게 할 수 있다.

1.2.6.8.1 매개 변수

pxTaskToResume : 다시 시작하고자 하는 task 의 핸들이다.

1.2.6.8.2 반환 값

pdTRUE : 다시 시작 하는(블록 해제 중인) task 우선순위가 현재 실행중인 task(인터럽트 된 task)와 같거나 높을 경우 반환한다. 이는 인터럽트를 종료하기 전에 컨텍스트 스위치가 수행되어야 함을 의미한다.

pdFALSE : 다시 시작하는 task 의 우선순위가 현재 실행중인 task(인터럽트 된 task)보다 낮으면 반환한다. 이는 인터럽트를 종료하기 전에 컨텍스트 스위치를 할 필요가 없음을 나타낸다.

1.2.6.8.3 기타

```
TaskHandle_t xHandle;

void vAFunction( void ) {
    // task 를 생성하고 핸들을 xHandle 에 저장한다.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

    // 나머지 코드를 입력한다.
}

void vTaskCode( void * pvParameters ) {
    // task 를 일시 중지하고 다시 시작한다.
    for( ;; ) {
      // 여기서 기능을 수행한다. task 는 NULL 을 이용하여 해당 task 자체를 일시 중단한다.
      vTaskSuspend( NULL );

      // task 가 일시 중단되었으므로 ISR 이 다시 시작(일시 중단 해제)할 때 까지 여기에 도달하지 않는다.
    }
}

void vAnExampleISR( void ) {
    BaseType_t xYieldRequired;

    // 일시 중단된 작업을 다시 시작한다.
    xYieldRequired = xTaskResumeFromISR( xHandle );
```

```
    if( xYieldRequired == pdTRUE ) {
      /* ISR 이 다시 시작된 task 로 돌아가도록 컨텍스트 스위치가 수행되야 한다. 이는 다시 시작된 task 의 우선순위가 현재
      실행중인 task 보다 높거나 같기 때문이다. ISR 에서 컨텍스트 스위칭을 수행하는데 필요한 구문은 포트마다 다르다.
      이 if()문은 xYieldRequired 를 매크로 매개 변수로 사용하여 portYIELD_FROM_ISR() 또는 portEND_SWITCHING_ISR()에 대한
      단일 호출로 대체 될 수 있다. portYIELD_FROM_ISR (xYieldRequired); */
      portYIELD_FROM_ISR();
    }
}
```

1.2.6.9 xTaskResumeAll : 모든 task 의 Suspended 상태를 Ready 상태로 전환

```
#include "FreeRTOS.h"
#include "task.h"

BaseType_t xTaskResumeAll ( void );
```

vTaskSuspendAll()에 대한 이전 호출은, Suspended 상태에서 활성 상태로 스케줄러를 전환하여 스케줄러 활동을 재개한다.

vTaskSuspendAll()를 호출하여 스케줄러를 일시 중단할 수 있다. 스케줄러가 일시 중단되면 인터럽트는 사용 가능한 상태로 유지되지만, 컨텍스트 스위치는 발생하지 않는다. 스케줄러가 일시 중단된 상태에서 컨텍스트 스위치 요청된 경우, 스케줄러가 다시 시작될 때까지 요청이 보류된다. (일시 중단되지 않는다.)

vTaskSuspendAll()에 대한 호출이 중첩될 수 있다. 스케줄러가 Suspended 상태를 떠나서 Active 상태로 다시 들어가기 전에 vTaskSuspendAll()에 대해 이전에 생성된 수만큼, xTaskResumeAll()에 동일한 수의 호출이 이루어져야 한다.

xTaskResumeAll()은 실행중인 task 에 호출되어야 하므로 스케줄러가 초기화 상태(스케줄러가 시작되기 전)에 있는 동안 호출하면 안된다. 다른 FreeRTOS API 함수는 스케줄러가 일시 중단된 동안 호출되면 안된다.

1.2.6.9.1 반환 값

pdTRUE : 스케줄러가 활성화 상태로 전환되었다. 전환으로 인해 보류중인 컨텍스트 스위칭이 발생하였다.

pdFALSE : 스케줄러가 활성화 상태가 전환되었으나 전환으로 컨텍스트 스위칭이 발생하지 않았거나, vTaskSuspendAll()에 대한 중첩 호출로 인해 스케줄러가 일시 중단 상태로 남아 있다.

1.2.6.9.2 기타

```
// 일시 중단 후 다음 스케줄러를 다시 시작하는 함수이다.
void vDemoFunction( void ) {
    // 이 함수는 스케줄러를 일시 중단한다. vTask1 에서 호출될 때 스케줄러는 이미 일시 중단되어 있으므로 중첩 깊이를 2 로 만든다.
    vTaskSuspendAll();

    /* 여기서 task 를 수행한다. vTaskSuspendAll()에 대한 호출이 중첩되었기 때문에
    여기서 스케줄러를 다시 시작해도 스케줄러가 활성화 상태로 다시 들어가지 않는다. */
    xTaskResumeAll();
}

void vTask1( void * pvParameters ) {
    for( ;; ) {
      /* 여기서 task 를 수행한다. task 는 swapped out 을 원하지 않는 작업을 수행하거나,
      다른 task(인터럽트가 아닌)에서 액세스하고있는 데이터에 대하여 액세스 하려고 한다.
      연산의 길이로 인해 인터럽트가 누락될 수 있어서 taskENTER_CRITICAL()이나 taskEXIT_CRITICAL()을 사용할 수 없다. */
      // 스케줄러가 컨텍스트 스위치를 수행하지 못하도록 한다.
      vTaskSuspendAll();
```

```
        // 여기서 작업을 수행한다. vTaskSuspendAll()에 대한 호출은 중첩될 수 있으므로 비 API 함수를 호출하는 것이 안전하다.
        // vTaskSuspendAll()를 호출하기 때문에 API 함수는 스케줄러가 일시 중단된 동안 호출되면 안된다.
        vDemoFunction();

        // task 가 완료된 후, 스케줄러를 다시 활성화 상태로 설정한다.
        if( xTaskResumeAll() == pdTRUE ) {
          // 컨텍스트 스위치가 xTaskResumeAll()내에서 발생했다.
        } else {
          // 컨텍스트 스위치가 xTaskResumeAll()내에서 발생하지 않았다.
        }
    }
}
```

1.2.7 task 인터럽트 설정

1.2.7.1 taskDISABLE_INTERRUPTS() : 인터럽트 비활성화

```
#include “FreeRTOS.h”
#include “task.h”

void taskDISABLE_INTERRUPTS ( void );
```

사용중인 FreeRTOS 포트가 configMAX_SYSCALL_INTERRUPT_PRIORITY(혹은 configMAX_API_CALL_INTERRUPT_PRIORITY) 커널 구성 상수를 사용하지 않으면, taskDISABLE_INTERRUPTS()을 호출 시 인터럽트가 전역 적으로 비활성화 된다.

반대로 사용하고 있을 경우, 함수 호출 시 configMAX_SYSCALL_INTERRUPT_PRIORITY 가 설정한 인터럽트 우선순위와 그 이하의 인터럽트가 해제되고 그보다 높은 우선순위가 전부 활성화된다. configMAX_SYSCALL_INTERRUPT_PRIORITY 은 일반적으로 FreeRTOSConfig.h 에서 정의된다.

taskDISABLE_INTERRUPTS()와 taskENABLE_INTERRUPTS()호출은 중첩되도록 설계되지 않았다. 즉, taskDISABLE_INTERRUPTS()호출을 두 번 해도, taskENABLE_INTERRUPTS()을 한번만 호출하면 인터럽트가 활성화된다. 중첩이 필요할 경우, taskENTER_CRITICAL()과 taskEXIT_CRITICAL()을 사용한다.

일부 FreeRTOS API 함수는 크리티컬 섹션의 중첩 수가 0 일 경우 인터럽트를 다시 활성화한다. 심지어 API 함수가 호출되기 전 taskDISABLE_INTERRUPTS()을 통해 인터럽트가 비활성화 된 경우에도 마찬가지다. 따라서, 인터럽트가 이미 비활성화 된 경우 FreeRTOS API 함수를 호출하는 것은 권장하지 않는다.

1.2.7.2 taskENABLE_INTERRUPTS() : 인터럽트 활성화

```
#include “FreeRTOS.h”
#include “task.h”

void taskENABLE_INTERRUPTS ( void );
```

taskENABLE_INTERRUPTS()을 호출하면 모든 인터럽트 우선 순위가 활성화된다.

1.2.7.3 taskENTER_CRITICAL() : 중첩을 통한 인터럽트 비활성화

```
#include “FreeRTOS.h”
#include “task.h”

void taskENTER_CRITICAL ( void );
```

taskENTER_CRITICAL() 및 taskEXIT_CRITICAL()은 호출이 중첩되도록 설계되었다. 따라서 크리티컬 섹션 발생 시, taskENTER_CRITICAL()에 대한 taskEXIT_CRITICAL()의 모든 후속 호출이 완료되어야 종료된다. 크리티컬 섹션은 매우 짧게 유지해야 하며, 아닐 경우 인터럽트 응답 시간에 부정적인 영향을 준다.

FreeRTOS API 함수는 크리티컬 섹션 내에서 호출되면 안된다.

크리티컬 섹션이 발생 시, taskENTER_CRITICAL()을 호출하여 입력한 다음 taskEXIT_CRITICAL()을 호출하여 종료한다. (다중 작업으로 인한 인터럽트의 중복이 발생하거나, 인터럽트 발생보다 더 중요한 작업이 있을 수 있기 때문에 사용된다.) taskENTER_CRITICAL()은 인터럽트 서비스 루틴에서 호출되면 안된다. 인터럽트 서비스 루틴에서 사용할 경우 ITTER_CRITICAL_FROM_ISR()을 참조한다.

taskENTER_CRITICAL() 과 taskEXIT_CRITICAL()매크로 함수는 인터럽트를 전역적으로 혹은, 특정 인터럽트 우선순위까지 단순 비활성화하는 기본 크리티컬 섹션을 구현한다. 인터럽트를 비활성화하지 않고 중요 섹션을 만드는 방법은 vTaskSuspendAll() API 를 참조한다.

FreeRTOS 포트가 사용되고 있다면, configMAX_SYSCALL_INTERRUPT_PRIORITY 을 사용한다. (우선순위를 설정한다. 선점형 컨텍스트 스위칭의 경우 인터럽트 내부에서 발생하기 때문에 비활성화 된 경우 발생하지 않는다. 따라서 taskENTER_CRITICAL() 사용시 task 는 명시적으로 차단 상태로 만들거나 시도하지 않는 한 크리티컬 섹션이 종료될 때까지 실행 상태로 유지된다. (크리티컬 섹션 내부에서 수행하면 안된다.))

1.2.7.3.1 기타

```
// 크리티컬 섹션을 사용하는 함수이다.
void vDemoFunction( void ) {
        // 이 예제는 크리티컬 섹션의 중첩이 2 가 된다.
        taskENTER_CRITICAL();

        // 여기서 크리티컬 섹션에 의해 보호되는 동작을 수행한다.

        // 크리티컬 섹션을 종료한다. 이 예제에서는 중첩 수를 1 만 감소시키므로 인터럽트를 활성화하지 않는다.
        taskEXIT_CRITICAL();
}

// 크리티컬 섹션을 호출하는 task 이다.
void vTask1( void * pvParameters ) {
        for( ;; ) {
            // 원하는 동작을 입력한다.

            // 크리티컬 섹션을 생성하기 위해 해당 함수를 호출한다.
            taskENTER_CRITICAL();

            // 크리티컬 섹션을 필요로 하는 코드를 입력한다.

            // 크리티컬 섹션에 대해 중첩될 수 있다.
            // 하지만, 함수 내에서 자체적으로 taskENTER_CRITICAL()와 taskEXIT_CRITICAL()를 포함하는 것이 안전하다.
            vDemoFunction();

            // 중첩 상태가 0 이기 때문에 인터럽트가 다시 활성화 된다.
            taskEXIT_CRITICAL();
        }
}
```

1.2.7.4 taskEXIT_CRITICAL() : 중첩을 통한 인터럽트 비활성화 해제

```
#include “FreeRTOS.h”
#include “task.h”

void TaskEXIT_CRITICAL ( void );
```

크리티컬 섹션은 taskENTER_CRITICAL()을 호출하여 입력한 후, taskEXIT_CRITICAL()을 호출하여 종료한다.

taskEXIT_CRITICAL()은 인터럽트 서비스 루틴에서 호출되면 안된다.

1.2.7.5 taskENTER_CRITICAL_FROM_ISR() : 인터럽트 서비스 루틴에서의 비활성화

```
#include “FreeRTOS.h”
#include “task.h”

UBaseType_t taskENTER_CRITICAL_FROM_ISR ( void );
```

인터럽트 서비스 루틴에서 사용할 수 있는 taskENTER_CRITICAL() 버전이다.

taskENTER_CRITICAL_FROM_ISR()와 taskEXIT_CRITICAL_FROM_ISR()로 동작한다. 이때, 사용중인 FreeRTOS 포트가 인터럽트 중첩을 지원하지 않으면 아무런 효과도 없다.

taskENTER_CRITICAL_FROM_ISR()와 taskEXIT_CRITICAL_FROM_ISR()에 대한 호출은 중첩되도록 설계되었지만 매크로가 사용되는 방식은 taskENTER_CRITICAL()와 taskEXIT_CRITICAL()의 방식과 다르다.

1.2.7.5.1 반환 값

ITTER_CRITICAL_FROM_ISR()이 호출된 시점의 인터럽트 마스크 상태이다.

반환 값은 taskEXIT_CRITICAL_FROM_ISR()에 사용된다.

1.2.7.5.2 기타

```
// 인터럽트 서비스 루틴이 호출될 함수이다.
void vDemoFunction( void ) {
        UBaseType_t uxSavedInterruptStatus;
        // 이 예제에서 이 함수는 크리티컬 섹션 내에서 호출되므로 중첩이 2 가 된다.
        // askENTER_CRITICAL_FROM_ISR()의 반환 값을 로컬 스택 변수에 저장하여,
        // askEXIT_CRITICAL_FROM_ISR()에 전달할 수 있다.
        uxSavedInterruptStatus = taskENTER_CRITICAL_FROM_ISR();

        // 크리티컬 섹션에 보호되는 동작을 한다.

        // 크리티컬 섹션을 종료한다. 하지만, 중첩이 남아있기 때문에 비활성화 상태이다..
        taskEXIT_CRITICAL_FROM_ISR( uxSavedInterruptStatus );
}

void vDemoISR( void ) {
        UBaseType_t uxSavedInterruptStatus;

        uxSavedInterruptStatus = taskENTER_CRITICAL_FROM_ISR();

        vDemoFunction();

        // 크리티컬 섹션이 필요로 하는 작업을 완료하고 종료한다.
        taskEXIT_CRITICAL_FROM_ISR( uxSavedInterruptStatus );
}
```

1.2.7.6 taskEXIT_CRITICAL_FROM_ISR() : 인터럽트 비활성화 해제

```
#include “FreeRTOS.h”
#include “task.h”

void taskENTER_CRITICAL_FROM_ISR ( UBaseType_t uxSavedInterruptStatus );
```

taskENTER_CRITICAL_FROM_ISR()을 호출하여 입력 된 크리티컬 섹션을 종료한다

1.2.7.6.1 매개 변수

uxSavedInterruptStatus : taskENTER_CRITICAL_FROM_ISR()에서 반환된 값을 사용해야 한다.

1.2.8 Notification

1.2.8.1 xTaskNotify : task 알림 및 notification 값 업데이트

```
#include “FreeRTOS.h”
#include “task.h”

BaseType_t xTaskNotify ( TaskHandle_t xTaskToNotify,
                         uint32_t ulValue,
                         eNotifyAction eAction );
```

각 task 에는 task 생성 시, 0 으로 초기화 되는 32 비트 notification 값이 있다. xTaskNotify()는 직접 이벤트를 보내거나, task 차단을 해제하는데 사용되며 다음 방법 중 하나를 이용하여 수신 작업의 notification 값을 업데이트한다.

- notification 값에 32 비트의 숫자를 쓴다.

- notification 값 하나를 추가(증가)한다.

- notification 값에 하나 이상의 비트를 설정한다.

- notification 값을 변경하지 않고 그대로 둔다.

task 의 notification 값은 가볍지만, 이진 또는 카운트 세마포어로 사용하고자 한다면 notification 대신 더 간단한 xTaskNotifyGive() API 함수를 사용하면 된다.

RTOS task 는 notification 기능을 기본적으로 사용 가능하며, FreeRTOSConfig.h 의 configUSE_TASK_NOTIFICATIONS 를 0 으로 설정하여 빌드에서 제외(task 당 8 바이트로 저장된다.)할 수 있다.

1.2.8.1.1 매개 변수

xTaskToNotify : 제어하고자 하는 RTOS task 의 핸들이다.

ulValue : task 의 notification 값을 갱신하는데 사용한다. 이 값은 eAction 의 매개 변수 값에 따라 동작이 다르다.

eAction : task 의 notification 에 수행할 작업이다. eAction 은 열겨형이며 다음 중 하나를 사용할 수 있다.

- eNoAction : task 에 알림을 주지만 notification 값은 변경되지 않는다. 이 경우 ulValue 는 사용되지 않는다.

- eSetBits : task 에 notification 값을 ulValue 와 OR 비트연산 한다. 예를 들어 ulValue 가 0x01 일 경우 0 번 비트의 notification 값이 설정되고, 0x04 일 경우 2 번 비트가 notification 에 설정된다. eSetBits 는 작업 알림을 빠르고 가볍게 주기 때문에 이벤트 그룹에 대한 대안으로 사용할 수 있다.

- eIncrement : task 의 notification 값을 1 씩 증가시킨다. 이 경우 ulValue 는 사용하지 않는다.

- eSetValueWithOverwrite : task 의 notification 값은 xTaskNotify()가 호출될 때 task 에 보류중인 알림이 있어도 무조건 ulValue 값으로 설정한다.

- eSetValueWithoutOverwrite : task 에 보류중인 알림이 있는 경우 notification 값이 변경되지 않고 xTaskNotify()가 pdFAIL 을 반환한다. 작업에 보류중인 알림이 없는 경우 notification 값은 ulValue 로 설정된다.

1.2.8.1.2 반환 값

eAction 매개 변수가 eSetValueWithoutOverwrite 로 설정되고, notification 값이 업데이트 되지 않으면 pdFAIL 이 반환된다.

그게 아닐 경우, pdPASS 가 리턴 된다.

1.2.8.1.3 기타

```
// xTask1Handle 가 참조하는 task 의 notification 값에 8 비트를 설정한다.
xTaskNotify( xTask1Handle, ( 1UL << 8UL ), eSetBits );

// xTask2Handle 이 참조하는 task 에 알림을 보내 task 를 차단 상태에서 제거하지만 task 의 notification 값은 업데이트 하지 않는다.
xTaskNotify( xTask2Handle, 0, eNoAction );

// task 가 이전 notification 값을 읽지 않았어도 xTaskHandle 이 참조하는 task 의 notification 값을 0x50 으로 설정한다.
xTaskNotify( xTask3Handle, 0x50, eSetValueWithOverwrite );

/* xTask4Handle 이 참조하는 task 의 notification 값을 0xfff 로 설정한다. task 가 수행하기 전에 xTaskNotifyWait()나
ulTaskNotifyTake()를 호출하여 task 의 기존 notification 값을 덮어 쓰지 않는 경우에만 notification 값을 0xfff 로 설정한다. */
if( xTaskNotify( xTask4Handle, 0xfff, eSetValueWithoutOverwrite ) == pdPASS ) {
    // notification 값이 업데이트 되었다.
} else {
    // notification 값이 업데이트 되지 않았다.
}
```

1.2.8.2 xTaskNotifyFromISR : ISR 에서 task 알림 및 notification 값 업데이트

```
#include “FreeRTOS.h”
#include “task.h”

BaseType_t xTaskNotifyFromISR ( TaskHandle_t xTaskToNotify, uint32_t ulValue,
                                eNotifyAction eAction, BaseType_t * pxHigherPriorityTaskWoken );
```

ISR 에서 사용할 수 있는 xTaskNotify()버전이다.

task 의 notification 값은 가볍지만, 이진 또는 카운트 세마포어로 사용하고자 한다면 notification 대신 더 간단한 vTaskNotifyGiveFromISR() API 함수를 사용하면 된다.

1.2.8.2.1 매개 변수

xTaskToNotify : 제어하고자 하는 RTOS task 의 핸들이다.

ulValue : task 의 notification 값을 갱신하는데 사용한다. 이 값은 eAction 의 매개 변수 값에 따라 동작이 다르다.

eAction : task 의 notification 에 수행할 작업이다. eAction 은 열겨형이며 다음 중 하나를 사용할 수 있다.

- eNoAction : task 에 알림을 주지만 notification 값은 변경되지 않는다. 이 경우 ulValue 는 사용되지 않는다.

- eSetBits : task 에 notification 값을 ulValue 와 OR 비트연산 한다. 예를 들어 ulValue 가 0x01 일 경우 0 번 비트의 notification 값이 설정되고, 0x04 일 경우 2 번 비트가 notification 에 설정된다. eSetBits 는 작업 알림을 빠르고 가볍게 주기 때문에 이벤트 그룹에 대한 대안으로 사용할 수 있다.

- eIncrement : task 의 notification 값을 1 씩 증가시킨다. 이 경우 ulValue 는 사용하지 않는다.

- eSetValueWithOverwrite : task 의 notification 값은 xTaskNotify()가 호출될 때 task 에 보류중인 알림이 있어도 무조건 ulValue 값으로 설정한다.

- eSetValueWithoutOverwrite : task 에 보류중인 알림이 있는 경우 notification 값이 변경되지 않고 xTaskNotify()가 pdFAIL 을 반환한다. 작업에 보류중인 알림이 없는 경우 notification 값은 ulValue 로 설정된다.

pxHigherPriorityTaskWoken : * pxHigherPriorityTaskWoken 는 pdFALSE 로 초기화되어야 한다.

xTaskNotifyFromISR()은 차단 상태를 벗어나도록 알림을 보내고, 알림을 받는 작업이 현재 실행중인 작업보다 우선순위가 높은 경우 * pxHigherPriorityTaskWoken 를 pdTRUE 로 설정한다. xTaskNotifyFromISR ()의 이 값이 pdTRUE 로 설정하면 인터럽트가 종료되기 전에 컨텍스트 스위칭을 요청해야 한다.

pxHigherPriorityTaskWoken 는 선택적 매개 변수이며 NULL 로 설정할 수 있다.

1.2.8.2.2 반환 값

eAction 매개 변수가 eSetValueWithoutOverwrite 로 설정되고, notification 값이 업데이트 되지 않으면 pdFAIL 이 반환된다.

그게 아닐 경우, pdPASS 가 리턴 된다.

1.2.8.2.3 기타

```
// 첫번째 비트는 각 인터럽트 소스를 나타내기 위해 정의된다.
#define TX_BIT 0x01
#define RX_BIT 0x02

// 인터럽트로 알림을 받을 task 의 핸들이다.
static TaskHandle_t xHandlingTask;

// 송신 인터럽트 서비스 루틴(ISR)이다.
void vTxISR( void ) {
	BaseType_t xHigherPriorityTaskWoken = pdFALSE;

	// 인터럽트 소스를 지운다.
	prvClearInterrupt();

	// task 의 notification 값에 TX_BIT 를 설정하여 전송이 완료되었음을 알린다.
	xTaskNotifyFromISR( xHandlingTask, TX_BIT, eSetBits, &xHigherPriorityTaskWoken );

	/* xHigherPriorityTaskWoken 가 pdTRUE 로 설정된 경우 인터럽트가 가장 높은 우선순위의 작업으로 직접 반환되도록
	 컨텍스트 스위치를 수행한다. 이 용도로 사용되는 매크로는 포트에 따라 다르며, END_SWITCHING_ISR ()등이 있다. */
	portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}

// 수신 인터럽트 서비스 루틴의 구현은, 수신 task 의 notification 값에 설정된 비트를 제외하고 동일하다.
void vRxISR( void ) {
	BaseType_t xHigherPriorityTaskWoken = pdFALSE;

	// 인터럽트를 지운다.
	prvClearInterrupt();

	// task 의 notification 값에 RX_BIT 를 설정하여 수신이 완료되었음을 알린다.
	xTaskNotifyFromISR( xHandlingTask, RX_BIT, eSetBits, &xHigherPriorityTaskWoken );

	/* xHigherPriorityTaskWoken 가 pdTRUE 로 설정된 경우 인터럽트가 가장 높은 우선순위의 작업으로 직접 반환되도록
	 컨텍스트 스위칭을 수행한다. 이 용도로 사용되는 매크로는 포트에 따라 다르며, END_SWITCHING_ISR ()등이 있다. */
	portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

```
// 인터럽트 서비스 루틴에 의해 제어되는 task 다.
static void prvHandlingTask( void * pvParameter ) {
	const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 500 );
	BaseType_t xResult;
```

```
        for( ;; ) {
            // 인터럽트에 대한 알림을 기다린다.
            // 입력 시 비트를 지우지 않는다, 종료 시 모든 비트를 지운다, notification 값을 저장한다, 최대 시간이다.
            xResult = xTaskNotifyWait( pdFALSE, ULONG_MAX, &ulNotifiedValue, xMaxBlockTime );

            if( xResult == pdPASS ) {
                // 알림을 받는다. 어떤 비트가 설정되었는지 확인한다.
                if( ( ulNotifiedValue & TX_BIT ) != 0 ) {
                    // TX_ISR 이 설정되었다.
                    prvProcessTx();
                }
                if( ( ulNotifiedValue & RX_BIT ) != 0 ) {
                    // RX_ISR 이 설정되었다.
                    prvProcessRx();
                }
            } else {
                // 시간 내에 알림을 받지 못했다.
                prvCheckForErrors();
            }
        }
}
```

1.2.8.3 xTaskNotifyGive : xTaskNotify()의 eIncrement 매크로 함수

```
#include "FreeRTOS.h"
#include "task.h"

BaseType_t xTaskNotifyGive ( TaskHandle_t xTaskToNotify );
```

각 task 에는 task 생성 시, 0 으로 초기화 되는 32 비트 notification 값이 있다.

task 의 notification 은 수신 task 의 차단을 해제하고 선택적으로 수신 task 의 notification 값을 업데이트 할 수 있는, task 로 직접 전송하는 이벤트다.

xTaskNotifyGive()은, 이진, 세마포어, 카운터 세마포어 등의 대안으로 사용되는 매크로로서, 가볍고 빠르다.

FreeRTOS 의 세마포어는 xSemaphoreGive()API 함수를 사용하여 제공되며, xTaskNotifyGive()는 세마포어 객체 대신에 수신 작업의 notification 값을 사용하는것과 동일하다. 이때 작업 notification 값은 바이너리 또는 카운터 세마포어로 사용될 때, 알림을 받은 task 는 xTaskNotifyWait()대신 간단한 xTaskNotifyTake()API 함수를 사용하여 알림을 기다려야한다.

RTOS task 의 알림 기능은 기본적으로 사용 가능하며, FreeRTOSConfig.h 에서 configUSE_TASK_NOTIFICATIONS 를 0 으로 설정하여 빌드에서 제외(task 당 8 바이트 저장한다.)할 수 있다.

1.2.8.3.1 매개 변수

xTaskToNotify : 알림을 사용하고자 하는 task 의 핸들이다.

1.2.8.3.2 반환 값

xTaskNotifyGive()는 eAction 매개 변수를 eIncrement 로 설정한 xTaskNotify()를 호출하는 매크로다. 따라서 리턴 값은 pdPASS 이다.

1.2.8.3.3 기타

```
// main()에 의해 생성된 두 task의 프로토 타입이다.
static void prvTask1( void *pvParameters );
static void prvTask2( void *pvParameters );

// main()에 의해 생성된 task의 핸들러다.
static TaskHandle_t xTask1 = NULL, xTask2 = NULL;

// 서로 다른 알림을 보내는 두개의 task를 만든 다음 RTOS 스케줄러를 실행한다.
void main( void ) {
        xTaskCreate( prvTask1, "Task1", 200, NULL, tskIDLE_PRIORITY, &xTask1 );
        xTaskCreate( prvTask2, "Task2", 200, NULL, tskIDLE_PRIORITY, &xTask2 );
        vTaskStartScheduler();
}

static void prvTask1( void *pvParameters ) {
        for( ;; ) {
            // prvTask2()에 알림을 보내 블록 상태를 벗어난다.
            xTaskNotifyGive( xTask2 );

            // prvTask2()에 알림을 기다리는 블록이다.
            ulTaskNotifyTake( pdTRUE, portMAX_DELAY );
        }
}

static void prvTask2( void *pvParameters ) {
        for( ;; ) {
            // prvTask1()에 알림을 기다리는 블록이다.
            ulTaskNotifyTake( pdTRUE, portMAX_DELAY );

            // prvTask1 에 알림을 보내서 블록 상태를 벗어난다.
            xTaskNotifyGive( xTask1 );
        }
}
```

1.2.8.4 vTaskNotifyGiveFromISR : ISR 에서의 xTaskNotifyGive()

```
#include “FreeRTOS.h”
#include “task.h”

void vTaskNotifyGiveFromISR ( TaskHandle_t xTaskToNotify, BaseType_t * pxHigherPriorityTaskWoken );
```

인터럽트 서비스 루틴에서 호출할 수 있는 xTaskNotifyGive()이다.

FreeRTOS 세마포어는 xSemaphoreGiveFromISR()API 함수를 제공하며 vTaskNotifyGiveFromISR()은 세마포어 개체 대신 수신 작업의 notification 값을 사용하는 것과 동일하다.

task notification 값으로 바이너리 또는 카운터 세마포어로 사용될 때 알림을 받는 task는 xTaskNotifyWait()대신 ulTaskNotifyTake() 함수를 사용하여 알림을 기다려야한다.

RTOS task의 알림 기능은 기본적으로 사용 가능하며, FreeRTOSConfig.h 에서 configUSE_TASK_NOTIFICATIONS를 0으로 설정하여 빌드에서 제외(task 당 8 바이트 저장한다.)할 수 있다.

1.2.8.4.1 매개 변수

xTaskToNotify : 알림을 사용하고자 하는 task 의 핸들이다.

pxHigherPriorityTaskWoken : * pxHigherPriorityTaskWoken 는 pdFALSE 로 초기화되어야 한다.

vTaskNotifyGiveFromISR()은 차단 상태를 벗어나도록 알림을 보내고, 알림을 받는 작업이 현재 실행중인 작업보다 우선순위가 높은 경우 * pxHigherPriorityTaskWoken 를 pdTRUE 로 설정한다. vTaskNotifyGiveFromISR()의 이 값이 pdTRUE 로 설정하면 인터럽트가 종료되기 전에 컨텍스트 스위칭을 요청해야 한다. pxHigherPriorityTaskWoken 는 선택적 매개 변수이며 NULL 로 설정할 수 있다.

1.2.8.4.2 기타

아래 예제는 일반적인 주변 장치 드라이버의 전송 기능의 한 예이다. task 는 전송 기능을 호출한 다음 전송 완료 알림이 올 때까지 차단 상태(CPU 시간을 사용하지 않는다.)에서 대기한다. 전송은 DMA 에 의해 수행되고 DMA 종료 인터럽트는 task 에 알리기 위해 사용된다.

```
static TaskHandle_t xTaskToNotify = NULL;

// 주변 장치 드라이버의 전송 기능이다.
void StartTransmission( uint8_t *pcData, size_t xDataLength ) {
	// 이 시점에서는 진행중인 전송이 없기 때문에 xTaskToNotify 가 NULL 이어야 한다.
	// 필요한 경우 뮤텍스를 사용하여 주변 장치에 대한 접근을 보호할 수 있다.
	configASSERT( xTaskToNotify == NULL );

	// 호출한 task 의 핸들을 저장한다.
	xTaskToNotify = xTaskGetCurrentTaskHandle();

	// 전송을 시작한다. 전송이 완료되면 인터럽트가 발생한다.
	vStartTransmit( pcData, xDataLength );
}

// 전송 종료 인터럽트이다.
void vTransmitEndISR( void ) {
	BaseType_t xHigherPriorityTaskWoken = pdFALSE;

	// 이 시점에서 전송이 진행 중일때, xTaskToNotify 는 NULL 이 아니여야 한다.
	configASSERT( xTaskToNotify != NULL );

	// 전송이 완료되었음을 task 에 알린다.
	vTaskNotifyGiveFromISR( xTaskToNotify, &xHigherPriorityTaskWoken );

	// 진행중인 전송이 없기 때문에 알림을 할 task 가 없다.
	xTaskToNotify = NULL;

	/* xHigherPriorityTaskWoken 이 pdTRUE 로 설정된 경우, 인터럽트가 가장 높은 우선순위의 task 로 직접 반환되도록
	컨텍스트 스위칭을 수행한다. 이 용도로 사용되는 매크로는 포트별로 다르며 END_SWITCHING_ISR()등이 있다. */
	portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}

// 전송을 시작한 task 는 차단 상태가 되기 때문에 CPU 시간을 소비하지 않고 완료될 때까지 기다린다.
void vAFunctionCalledFromATask( uint8_t ucDataToTransmit, size_t xDataLength ) {
	uint32_t ulNotificationValue;
	const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 200 );

	// 함수를 호출하여 전송을 시작한다.
	StartTransmission( ucDataToTransmit, xDataLength );

	// 전송이 완료될 때까지 기다린다.
	ulNotificationValue = ulTaskNotifyTake( pdFALSE, xMaxBlockTime );
```

```
        if( ulNotificationValue == 1 ) {
            // 전송이 정상적으로 완료되었다.
        } else {
            // ulTaskNotifyTake()에 대한 호출 시간이 초과하였다.
        }
}
```

1.2.8.5 xTaskNotifyAndQuery : 기존 notification 값을 반환

```
#include "FreeRTOS.h"
#include "task.h"

BaseType_t xTaskNotifyAndQuery ( TaskHandle_t xTaskToNotify, uint32_t ulValue,
                                 eNotifyAction eAction, uint32_t * pulPreviousNotifyValue );
```

xTaskNotifyAndQuery()는 xTaskNotify()와 유사하지만 task 의 이전 알림 값이 반환되는 추가 매개 변수를 포함한다.

1.2.8.5.1 매개 변수

xTaskToNotify : 제어하고자 하는 RTOS task 의 핸들이다.

ulValue : task 의 notification 값을 갱신하는데 사용한다. 이 값은 eAction 의 매개 변수 값에 따라 동작이 다르다.

eAction : task 의 notification 에 수행할 작업이다. eAction 은 열겨형이며 다음 중 하나를 사용할 수 있다.

- eNoAction : task 에 알림을 주지만 notification 값은 변경되지 않는다. 이 경우 ulValue 는 사용되지 않는다.

- eSetBits : task 에 notification 값을 ulValue 와 OR 비트연산 한다. 예를 들어 ulValue 가 0x01 일 경우 0 번 비트의 notification 값이 설정되고, 0x04 일 경우 2 번 비트가 notification 에 설정된다. eSetBits 는 작업 알림을 빠르고 가볍게 주기 때문에 이벤트 그룹에 대한 대안으로 사용할 수 있다.

- eIncrement : task 의 notification 값을 1 씩 증가시킨다. 이 경우 ulValue 는 사용하지 않는다.

- eSetValueWithOverwrite : task 의 notification 값은 xTaskNotify()가 호출될 때 task 에 보류중인 알림이 있어도 무조건 ulValue 값으로 설정한다.

- eSetValueWithoutOverwrite : task 에 보류중인 알림이 있는 경우 notification 값이 변경되지 않고 xTaskNotify()가 pdFAIL 을 반환한다. 작업에 보류중인 알림이 없는 경우 notification 값은 ulValue 로 설정된다.

pulPreviousNotifyValue : xTaskNotifyAndQuery()의 동작에 의해 임의이 비트가 수정되기 전, 대상 task 의 notification 값을 전달하는데 사용된다. pulPreviousNotifyValue 는 선택적 매개 변수이기 때문에 필요하지 않을 경우 NULL 로 설정할 수 있다. 하지만 이것을 사용하지 않는다면 xTaskNotify()를 사용하는것이 더 좋다.

1.2.8.5.2 반환 값

eAction 매개 변수가 eSetValueWithoutOverwrite 로 설정되고, notification 값이 업데이트 되지 않으면 pdFAIL 이 반환된다.

그게 아닐 경우, pdPASS 가 리턴 된다.

1.2.8.5.3 기타

```
uint32_t ulPreviousValue;

xTaskNotifyAndQuery( xTask1Handle, ( 1UL << 8UL ), eSetBits, NULL );

xTaskNotifyAndQuery( xTask2Handle, 0, eNoAction, &ulPreviousValue );

xTaskNotifyAndQuery( xTask3Handle, 0x50, eSetValueWithOverwrite, &ulPreviousValue );

if( xTaskNotifyAndQuery( xTask4Handle, 0xfff, eSetValueWithoutOverwrite, &ulPreviousValue ) == pdPASS ) {

} else {

}
```

1.2.8.6 xTaskNotifyAndQueryFromISR : ISR 에서 기존 notification 값을 반환

```
#include “FreeRTOS.h”
#include “task.h”

BaseType_t xTaskNotifyAndQueryFromISR ( TaskHandle_t xTaskToNotify, uint32_t ulValue,
                                        eNotifyAction eAction, uint32_t * pulPreviousNotifyValue,
                                        BaseType_t * pxHigherPriorityTaskWoken );
```

ISR 에서 사용할 수 있는 xTaskNotifyAndQuery()버전이다.

1.2.8.6.1 매개 변수

xTaskToNotify : 제어하고자 하는 RTOS task 의 핸들이다.

ulValue : task 의 notification 값을 갱신하는데 사용한다. 이 값은 eAction 의 매개 변수 값에 따라 동작이 다르다.

eAction : task 의 notification 에 수행할 작업이다. eAction 은 열겨형이며 다음 중 하나를 사용할 수 있다.

- eNoAction : task 에 알림을 주지만 notification 값은 변경되지 않는다. 이 경우 ulValue 는 사용되지 않는다.

- eSetBits : task 에 notification 값을 ulValue 와 OR 비트연산 한다. 예를 들어 ulValue 가 0x01 일 경우 0 번 비트의 notification 값이 설정되고, 0x04 일 경우 2 번 비트가 notification 에 설정된다. eSetBits 는 작업 알림을 빠르고 가볍게 주기 때문에 이벤트 그룹에 대한 대안으로 사용할 수 있다.

- eIncrement : task 의 notification 값을 1 씩 증가시킨다. 이 경우 ulValue 는 사용하지 않는다.

- eSetValueWithOverwrite : task 의 notification 값은 xTaskNotify()가 호출될 때 task 에 보류중인 알림이 있어도 무조건 ulValue 값으로 설정한다.

- eSetValueWithoutOverwrite : task 에 보류중인 알림이 있는 경우 notification 값이 변경되지 않고 xTaskNotify()가 pdFAIL 을 반환한다. 작업에 보류중인 알림이 없는 경우 notification 값은 ulValue 로 설정된다.

pulPreviousNotifyValue : xTaskNotifyAndQuery()의 동작에 의해 임의이 비트가 수정되기 전, 대상 task 의 notification 값을 전달하는데 사용된다. pulPreviousNotifyValue 는 선택적 매개 변수이기 때문에 필요하지 않을 경우 NULL 로 설정할 수 있다. 하지만 이것을 사용하지 않는다면 xTaskNotifyFromISR()를 사용하는 것이 더 좋다.

pxHigherPriorityTaskWoken : * pxHigherPriorityTaskWoken 는 pdFALSE 로 초기화되어야 한다.

xTaskNotifyAndQueryFromISR()은 차단 상태를 벗어나도록 알림을 보내고, 알림을 받는 작업이 현재 실행중인 작업보다 우선순위가 높은 경우 * pxHigherPriorityTaskWoken 를 pdTRUE 로 설정한다. xTaskNotifyAndQueryFromISR()의 이 값이 pdTRUE 로 설정하면 인터럽트가 종료되기 전에 컨텍스트 스위칭을 요청해야 한다.

pxHigherPriorityTaskWoken 는 선택적 매개 변수이며 NULL 로 설정할 수 있다.

1.2.8.6.2 반환 값

eAction 매개 변수가 eSetValueWithoutOverwrite 로 설정되고, notification 값이 업데이트 되지 않으면 pdFAIL 이 반환된다.

그게 아닐 경우, pdPASS 가 리턴 된다.

1.2.8.6.3 기타

```
uint32_t ulPreviousValue;

// xHigherPriorityTaskWoken 는 pdFALSE 로 설정해야 인터럽트 내에서 호출된 함수가 pdTRUE 로 설정된 경우 감지할 수 있다.
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

// xTask1Handle 이 참조하는 task 의 notification 값에 8 비트를 설정한다. 작업 이전 알림 값은 필요하지 않으므로,
// pulPreviousNotifyValue 매개 변수는 NULL 로 설정된다.
xTaskNotifyAndQueryFromISR( xTask1Handle, ( 1UL << 8UL ), eSetBits, NULL, &xHigherPriorityTaskWoken );

// xTask2Handle 이 참조하는 task 에 알림을 보내 차단 상태에서 제거하지만 task 의 notification 값은 업데이트 하지 않는다.
// task 의 현재 notification 값은 ulPreviousValue 에 저장한다.
xTaskNotifyAndQueryFromISR( xTask2Handle, 0, eNoAction, &ulPreviousValue, &xHigherPriorityTaskWoken );

/* xHigherPriorityTaskWoken 이 pdTRUE 로 설정된 경우, 인터럽트가 가장 높은 우선순위의 task 로 직접 반환되도록 컨텍스트
스위칭을 수행해야 한다. 이 용도로 사용되는 매크로는 사용중인 포트에 따라 다르며, END_SWITCHING_ISR() 등이 있다. */
portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
```

1.2.8.7 xTaskNotifyStateClear : 보류중인 notification 값 제거

```
#include “FreeRTOS.h”
#include “task.h”

BaseType_t xTaskNotifyStateClear ( TaskHandle_t xTask );
```

각 task 는 0 으로 초기화되는 32 비트 notification 값이 있다. task 알림은 수신 task 를 차단 해제하고, 선택적으로 수신 task 의 notification 값을 업데이트 할 수 있는 task 로 직접 전송되는 이벤트다. task 는 알림이 도착할 때, 알림을 대기하도록 차단 상태에 있었다면 차단 상태를 종료하며 알림은 보류 상태로 남아있지 않는다. 알림이 도착할 때 task 가 알림을 기다리지 않으면 알림은 다음 중 하나가 될 때까지 보류 상태를 유지한다.

- 송신 task 가 알림 값을 읽는다.

- 송신 task 가 xTaskNotifyStateClear()호출을 한 주 task 이다.

xTaskNotifyStateClear()는 대기중인 알림을 지우지만, notification 값은 변경하지 않는다.

1.2.8.7.1 매개 변수

xTask : 보류중인 알림이 지워지는 task 의 핸들이다. NULL 로 설정 시, xTaskNotifyStateClear()를 호출한 task 의 보류 알림이 지워진다.

1.2.8.7.2 반환 값

xTask 가 참조하는 task 에 보류중인 알림이 있으면 pdPASS 가 리턴 된다. 알림이 없을 경우 pdFAIL 이 리턴 된다.

1.2.8.7.3 기타

```
/* 예제는 UART 수신 함수이다. 이 기능은 UART 수신을 시작한 다음 수신이 완료되었음을 통보 받을 때 까지 대기한다. 수신 완료
알림은 UART 인터럽트에 전송된다. 호출 task 의 통지 상태는 수신이 시작되기 전에 삭제되어 task 가 통지 상태에서 차단을 시도하기
전에 이미 보류 중이 아닌지 확인한다. */
void vSerialPutString( const signed char * const pcStringToSend, uint16_t usStringLength ) {
        const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 5000 );

        /* xSendingTask 는 송신이 완료되기를 기다리는 task 의 핸들을 갖고 있다. 해당 값이 NULL 일 경우, 수신이 진행되지 않는다.
        이전 문자열의 전송이 완료되지 않았을 경우, 새 문자열을 보내지 않도록 한다. */
        if( ( xSendingTask == NULL ) && ( usStringLength > 0 ) ) {

            // 호출 task 의 알람 상태가 아직 보류 상태가 아닌지 확인한다.
            xTaskNotifyStateClear( NULL );

            // 송신 task 의 핸들을 저장한다. 전송이 완료되면 task 차단을 해제하는데 사용한다.
            xSendingTask = xTaskGetCurrentTaskHandle();

            // 문자열 전송을 시작한다. 전송은 인터럽트에 의해 제어된다.
            UARTSendString( pcStringToSend, usStringLength );

            // xSendingTask 에 UART ISR 을 통해(차단 해제) 송신이 완료될 때까지
            // 차단 상태(CPU 시간을 사용하지 않는다.)로 기다린다.
            ulTaskNotifyTake( pdTRUE, xMaxBlockTime );
        }
}
```

1.2.8.8 ulTaskNotifyTake : notification 값 감소 및 초기화

```
#include “FreeRTOS.h”
#include “task.h”

uint32_t ulTaskNotifyTake ( BaseType_t xClearCountOnExit, TickType_t xTicksToWait );
```

각 task 는 0 으로 초기화되는 32 비트 notification 값이 있다. task 알림은 수신 task 를 차단 해제하고, 선택적으로 수신 task 의 notification 값을 업데이트 할 수 있는 task 로 직접 전송되는 이벤트다.

ulTaskNotifyTake()는 task 의 알림이 바이너리 세마포어 또는 카운터 세마포어 대신 사용하기 위한 것으로, 가볍고 빠르다. FreeRTOS 세마포어는 xSemaphoreTake()API 함수를 사용하여 가져오며, ulTaskNotifyTake()는 별도의 세마포어 객체 대신 task notification 값을 사용한다.

알림이 있을 때까지 xTaskNotifyWait()가 반환되며, ulTaskNotifyTake()는 task 의 notification 값이 0 이 아닌 경우 반환하고, 반환하기 전에 task 의 notification 값을 감소시킨다.

task 는 선택적으로 ulTaskNotifyTake()를 사용하여 task notification 값이 0 이 아닐 때까지 대기할 수 있다. task 가 차단 상태일 때 CPU 시간은 전혀 소비하지 않는다.

ulTaskNotifyTake()는 종료 시 task 의 notification 값을 0 으로 지울 수 있다. 이 경우 notification 값은 바이너리 세마포어 처럼 작동하거나 task 의 notification 값을 감소시킨다. 이 때, notification 값은 카운터 세마포어 처럼 동작한다.

task 가 자신의 notification 값을 바이너리 또는 카운터 세마포어로 사용하는 경우 다른 task 및 인털럽트는 함수의 eAction 매개 변수가 eIncrement 로 설정된 xTaskNotifyGive()매크로 또는 xTaskNotify()함수로 알람을 보내야 한다. (둘 다 동일하다.)

RTOS task 알림 기능은 기본적으로 사용 가능하며 FreeRTOSConfig.h 에서 configUSE_TASK_NOTIFICATIONS 를 0 으로 설정하여 빌드에서 제외(task 당 8 바이트를 사용한다.)할 수 있다.

1.2.8.8.1 매개 변수

xClearCountOnExit : xClearCountOnExit 을 pdFALSE 로 설정하면 ulTaskNotifyTake()가 종료되기 전에 task 의 notification 값이 감소한다. 이는 xSemaphoreTake()에 대한 성공적인 호출로 카운터 세마포어 값이 감소하는것과 같다. xClearCountOnExit 를 pdTRUE 로 설정하면 ulTaskNotifyTake()가 종료되기 전에 task 알림 값을 0 으로 재설정 한다. 이것은 xSemaphoreTake()에 대한 호출이 성공한 후 0 이 남아있는 바이너리 세마포어 값과 같다.

xTicksToWait : ulTaskNotifyTake()가 호출될 때 알림이 아직 보류 중이 아닌 경우, 알림을 수신하기 위해 차단된 상태에서 대기할 수 있는 최대 시간이다. RTOS task 는 차단 상태일 때 CPU 시간을 소비하지 않는다.

시간은 RTOS tick 으로 지정된다. pdMS_TO_TICKS()매크로는 밀리 초 단위로 지정된 시간을 tick 으로 변환하는데 사용된다.

1.2.8.8.2 반환 값

task 의 notification 값이 감소되거나, 0 으로 초기화 되기 전의 task notification 값이다.

1.2.8.8.3 기타

```
/* 인터럽트를 생성한 이벤트보다 처리되는 우선 순위가 높은 task 의 차단 해제하는 인터럽트 처리기다.
task 우선순위가 충분히 높으면 인터럽트는 task 로 다시 되돌아 간다. (하던 작업을 인터럽트 하여, 다른 task 로 돌아간다.)
따라서 모든 처리가 완료된 것처럼 처리가 연속적으로 발생한다. 인터럽트 처리기 자체에서 이를 수행한다. */
void vANInterruptHandler( void ) {
        BaseType_t xHigherPriorityTaskWoken = pdFALSE;

        // 인터럽트를 지운다.
        prvClearInterruptSource();

        // task 가 인터럽트에 의해 필요한 모든 프로세싱을 수행할 수 있도록 핸들링 task 의 차단을 해제한다.
        // xHandlingTask 는 task 생성 시 얻는 작업의 핸들이다.
        vTaskNotifyGiveFromISR( xHandlingTask, &xHigherPriorityTaskWoken );

        // xHigherPriorityTaskWoken 이 pdTRUE 로 설정되면 컨텍스트 스위치가 강제 실행한다.
        // 이렇게 하는데 사용되는 매크로는 포트에 따라 다르며 END_SWITCHING_ISR()등이 있다.
        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}

// 주변 장치에 대한 서비스가 필요하다는 알림을 받기 위해 차단 상태인 task 이다.
void vHandlingTask( void *pvParameters ) {
        BaseType_t xEvent;
        for( ;; ) {
            // 알림을 대기하려면 시간제한 없이 차단한다. (시간 초과 없이, 함수 반환 값을 확인할 필요 없도록 한다.)
            여기서 RTOS task 알림은 바이너리 세마포어로 사용되므로 종료 시 notification 값이 0 으로 지워진다.
            실제 응용 프로그램 사용 시 무기한으로 차단하지 말고, 인터럽트가 더 이상 알림을 보내지 못할 수 있는
            오류 조건을 처리하기 위하여 가끔 시간을 끌어야 한다. */
            // 종료 시 notification 값을 지운다, 무한정 차단한다.
            ulTaskNotifyTake( pdTRUE, portMAX_DELAY );
            // RTOS task 알림은 바이너리(카운터와 반대) 세마포어로 사용되므로,
            // 주변 장치에서 보류중인 모든 이벤트가 처리되었을 때만 알림을 대기한다.
            do {
                xEvent = xQueryPeripheral();
                if( xEvent != NO_MORE_EVENTS ) {
                    vProcessPeripheralEvent( xEvent );
                }
            }
            while( xEvent != NO_MORE_EVENTS );
        }
}
```

1.2.8.9 xTaskNotifyWait : 대기 시간을 보유한 notification 값 제어

```
#include “FreeRTOS.h”
#include “task.h”

BaseType_t xTaskNotifyWait ( uint32_t ulBitsToClearOnEntry, uint32_t ulBitsToClearOnExit,
                             uint32_t *pulNotificationValue, TickType_t xTickToWait );
```

xTaskNotifyWait()은 호출한 task 가, 지정한 시간까지 task 알림에 대해 수신 대기한다.

수신 task 가 차단 상태일 때 알림 수신을 받을 경우, 차단 상태에서 해제되며 알림이 지워진다.

task 알림을 이용하여 이진 또는 카운터 세마포어 동작을 구현하는 경우, xTaskNotifyWait() 대신 간단한 ulTaskNotifyTake()를 사용한다.

RTOS task 알림 기능은 기본적으로 사용 가능하며 FreeRTOSConfig.h 에서 configUSE_TASK_NOTIFICATIONS 를 0 으로 설정하여 빌드에서 제외(task 당 8 바이트를 사용한다.)할 수 있다.

1.2.8.9.1 매개 변수

ulBitsToClearOnEntry : ulBitsToClearOnEntry 에 설정된 비트는 xTaskNotifyWait()호출 시 알림이 보류 상태가 아닌 경우(task 가 새로운 알림을 대기하기 전에), 호출한 task 의 notification 값을 지운다.

예를 들어, ulBitsToClearOnEntry 가 0x01 이면 task notification 값의 비트 0 은 해당 기능이 입력될 때 지워진다.
ulBitsToClearOnEntry 를 0xffffffff(ULONG_MAX)로 설정하면 task notification 값의 모든 비트가 지워지고 0 이 된다.

ulBitsToClearOnExit : ulBitsToClearOnExit 에 설정된 비트는 xTaskNotifyWait()함수가 알림을 받으면 종료되기 전에 호출 task 의 notification 값을 지운다. 또한, task 의 notification 값이 *pulNotificationValue 에 저장되면 비트가 지워진다.

예를 들어, ulBitsToClearOnExit 가 0x03 이면 task notification 값의 비트 0 과 1 은 함수가 종료되기 전에 지워진다.

ulBitsToClearOnExit 를 0xffffffff(ULONG_MAX)로 설정하면 task notification 값의 모든 비트가 0 으로 지워진다

pulNotificationValue : task 의 notification 값을 전달하는데 사용된다. 해당 값에 복사된 값은 task 의 notification 값으로 ulBitsToClearOnExit 설정으로 제거되기 전의 비트 값이다.

해당 매개 변수는 선택적 매개 변수이며 사용하지 않을 경우 NULL 로 설정할 수 있다.

xTicksToWait : xTaskNotifyWait()가 호출될 때 알림이 아직 보류 상태가 아닌 경우, 알림 수신을 위해 차단 상태에서 대기할 수 있는 최대 시간이다. task 가 차단 상태일 때 CPU 의 시간은 소비되지 않다. 시간은 RTOS tick 으로 지정된다.

pdMS_TO_TICKS()매크로는 tick 을 밀리 초 단위로 지정된 시간을 변환하는데 사용된다.

1.2.8.9.2 반환 값

알림을 받거나 xTaskNotifyWait()가 호출되었을 때, 알림이 이미 보류 상태면 pdTRUE 가 리턴 된다.

xTaskNotifyWait()에 대한 호출이 알림을 수신하기 전에 시간 초과될 경우 pdFALSE 가 리턴 된다.

1.2.8.9.3 기타

```
// 이 task 는 이벤트 그룹의 플래그와 동일한 목적으로 사용된다. 이 task 는 notification 값의 비트를 통하여 이벤트를 표시한다.
void vAnEventProcessingTask( void *pvParameters ) {
        uint32_t ulNotifiedValue;

        for( ;; ) {

            /* 알림을 무기한으로 기다리며 차단 상태로 존재한다. (시간 초과가 없기 때문에 반환 값 확인이 필요 없다.)
            이 RTOS task 의 notification 값 비트는 알림 task 및 인터럽트에 의해 설정되어, 어떤 이벤트가 발생했는지 표시한다. */
            /* 입력 시 notification 비트를 유지한다, 종료 시 notification 비트를 0 으로 한다,
            notification 값을 ulNotifiedValue 에 저장한다, 무한정 차단한다. */
            xTaskNotifyWait( 0x00, ULONG_MAX, &ulNotifiedValue, portMAX_DELAY );

            // notification 값으로 된 이벤트를 처리한다.
            if( ( ulNotifiedValue & 0x01 ) != 0 ) {

                // 0 비트 이벤트를 처리한다.
                prvProcessBit0Event();

            } if( ( ulNotifiedValue & 0x02 ) != 0 ) {

                // 1 비트 이벤트를 처리한다.
                prvProcessBit1Event();

            } if( ( ulNotifiedValue & 0x04 ) != 0 ) {

                // 2 비트 이벤트를 처리한다.
                prvProcessBit2Event();
            }
        // 기타 동작을 처리한다.
        }
}
```

1.2.9 스케줄러

1.2.9.1 xTaskGetTickCount : 현재 tick 카운트 값을 출력

```
#include “FreeRTOS.h”
#include “task.h”

TickType_t xTaskGetTickCount ( void );
```

tick 카운트는 스케줄러가 시작된 이후 발생한 tick 인터럽트의 총 수다. xTaskGetTickCount()는 현재 tick 카운트 값을 반환한다.

tick 구간이 실제로 표시되는 시간은 FreeRTOSConfig.h 내의 configTICK_RATE_HZ 에 할당된 값을 따른다. pdMS_TO_TICKS()매크로는 밀리 초 단위의 시간을 tick 시간 단위로 변환하는데 사용할 수 있다. 일반적으로 tick 이 오버 플로우 될 경우 0 으로 돌아가지만, 이것은 커널의 내부 동작에 영향을 주지 않는다. 예를 들어 task 가 차단 상태일 때, tick 수가 오버 플로우 되더라도 지정된 시간동안 task 는 차단 상태로 존재한다. 하지만 응용 프로그램이 tick 의 값을 직접 사용하는 경우 오버 플로우에 대해 고려해야 한다. tick 이 오버 플로우 되는 빈도는 tick 의 빈도와 카운트 값을 유지하는데 사용되는 데이터 유형에 따라 변한다.

configUSE_16_BIT_TICKS 가 1 로 설정되면 tick 카운트는 16 비트 변수에서 유지되며, 0 으로 설정되면 32 비트 변수로 유지된다.

1.2.9.1.1 반환 값

xTaskGetTickCount()는 해당 API 가 호출될 때의 tick 값을 반환한다.

1.2.9.1.2 기타

```
void vAFunction( void ) {
        TickType_t xTime1, xTime2, xExecutionTime;

        // 함수가 시작된 시간을 저장한다.
        xTime1 = xTaskGetTickCount();

        // 일부 작업을 수행하고, 작업 실행 후 시간을 저장한다.
        xTime2 = xTaskGetTickCount();

        // 해당 작업의 완료까지 걸린 시간을 나타낸다.
        xExecutionTime = xTime2 - xTime1;
}
```

1.2.9.2 xTaskGetTickCountFromISR : ISR 내부에서 tick 카운트 값을 출력

```
#include “FreeRTOS.h”
#include “task.h”

TickType_t xTaskGetTickCountFromISR ( void );
```

ISR 에서 호출할 수 있는 xTaskGetTickCount()버전이다. 내용은 동일하다.

1.2.9.2.1 반환 값

xTaskGetTickCountFromISR()은 해당 API 가 호출될 때의 tick 값을 반환한다.

1.2.9.2.2 기타

```
void vAnISR( void ) {
        static TickType_t xTimeISRLastExecuted = 0;
        TickType_t xTimeNow, xTimeBetweenInterrupts;

        // 인터럽트가 입력된 시간을 저장한다.
        xTimeNow = xTaskGetTickCountFromISR();

        // 일부 작업을 진행하고, 이 인터럽트와 이전 인터럽트 사이에 tick 차이를 구한다.
        xTimeBetweenInterrupts = xTimeISRLastExecuted - xTimeNow;

        // 인터럽트 사이에 tick 이 일정 이상 차이가 발생 시 특정 동작을 하도록 한다.
        if( xTimeBetweenInterrupts > 200 ) {
            // 취할 동작을 입력한다.
        }

        // 이 인터럽트가 입력된 시간을 기억한다.
        xTimeISRLastExecuted = xTimeNow;
}
```

1.2.9.3 vTaskSetTimeOutState : TimeOut 시간 초기화

```
#include “FreeRTOS.h”
#include “task.h”

void vTaskSetTimeOutState ( TimeOut_t * const pxTimeOut );
```

task 는 차단 상태로 전환하여 이벤트를 기다릴 수 있다. 일반적으로 task 는 차단 상태로 무기한 대기하지 않고, 제한 시간이 지정된다. task 가 대기중인 이벤트가 발생하기 전에 제한 시간이 만료되면 task 가 차단 상태에서 제거된다. task 가 발생하기를 기다리는 동안 차단 상태로 두 번 이상 들어가고 나가면 작업이 차단 상태로 전환될 때마다 사용된 시간 제한을 조정하여 차단 상태에서 소요된 시간의 합계를 조정해야 한다. 이 조정은 통해 원래 지정된 시간을 초과하지 않아야 한다.

xTaskCheckForTimeOut()은 tick 카운터 오버 플로우같은 비 정기적인 상황을 고려하여 조정을 수행해야 한다. 그렇지 않으면 수동 조정으로 인한 오류가 발생할 수 있다.

vTaskSetTimeOutState()는 xTaskCheckForTimeOut()와 같이 사용된다. vTaskSetTimeOutState()를 호출하여 초기 조건을 설정한 후, xTaskCheckForTimeOut()를 호출하여 시간 초과 조건을 확인 후, 시간 초과가 발생하지 않은 경우 나머지 블록 시간을 조정할 수 있다.

1.2.9.3.1 매개 변수

pxTimeOut : 타임 아웃 발생 여부를 결정하는데 필요한 정보를 저장하기 위해 필요한, 초기화 될 구조체에 대한 포인터이다.

1.2.9.3.2 기타

```
/* UART 인터럽트로 채워진 RX 버퍼에서 uxWantedBytes 를 수신하는데 사용되는 드라이버 라이브러리 함수다. Rx 버퍼에 충분한
공간이 없으면 task 는 버퍼에 공간이 생길 때까지 차단 상태가 된다. 만약 데이터 공간이 충분하지 않은 상태로 차단이 해제될 경우,
다시 차단 상태로 다시 들어가고, xTaskCheckForTimeOut()은 차단 시간동안 소비된 총 시간이 MAX_TIME_TO_WAIT 를 초과하지
않도록 차단 시간을 다시 계산한다. 버퍼에 uxWantedBytes 바이트가 포함되거나 차단 상태에서 소비된 총 시간이
MAX_TIME_TO_WAIT 에 도달할 때까지 이 작업이 계속된다. 이때 task 는 최대 uxWantedBytes 까지 사용 가능한 많은 바이트를
읽는다. */
size_t xUART_Receive( uint8_t *pucBuffer, size_t uxWantedBytes ) {
    size_t uxReceived = 0;
    TickType_t xTicksToWait = MAX_TIME_TO_WAIT;
    TimeOut_t xTimeOut;
```

```
    // xTimeOut 을 초기화 한다. 이것은 입력된 시간을 기록한다.
    vTaskSetTimeOutState( &xTimeOut );

    // 버퍼가 원하는 바이트 수를 포함하거나 제한 시간이 될 때까지 반복한다.
    while( UART_bytes_in_rx_buffer( pxUARTInstance ) < uxWantedBytes ) {

      /* 버퍼에 충분한 데이터가 없으므로 task 가 차단 상태가 된다. 차단 상태에서 소비된 총 시간은 MAX_TIME_TO_WAIT 를
      초과하지 않도록 차단 상태로 소비된 시간을 고려하여 xTicksToWait 를 조정한다. */
      if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) != pdFALSE ) {

        // 요구하는 바이트 수를 사용 가능해지기 전에 시간초과를 했다. 루프를 종료한다.
        break;
      }
      // 수신 인터럽트가 버퍼 공간에 더 많은 데이터를 배치했다는 통지를 받으려면 최대 xTicksToWait tick 을 기다린다.
      ulTaskNotifyTake( pdTRUE, xTicksToWait );
    }
    // uxWantedBytes 를 수신 버퍼에서 pucBuffer 로 읽는다. 읽은 실제 바이트 수(uxWantedBytes 보다 작을 수 있다.)가 반환된다.
    uxReceived = UART_read_from_receive_buffer( pxUARTInstance, pucBuffer, uxWantedBytes );

    return uxReceived;
}
```

1.2.9.4 vTaskStepTick : tick 값 재설정

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskStepTick ( TickType_t xTicksToJump );
```

RTOS 가 tick Idle 기능을 사용하도록 구성된 경우 tick 인터럽트가 중지되고 마이크로 컨트롤러는 Idle task 만 실행할 수 있는 저전력 상태가 된다. 저전력 상태를 빠져나오면 tick 카운터 값은 그것이 정지된 동안의 시간을 고려하여 수정되야 한다.

FreeRTOS 포트가 기본적으로 portSUPPRESS_TICKS_AND_SLEEP()의 구현을 포함하면 vTaskStepTick()이 내부적으로 사용되어 올바른 tick 계수 값이 유지된다.

vTaskStepTick()은 기본적으로 portSUPPRESS_TICKS_AND_SLEEP()의 구현을 재정의 하고, 사용중인 포트가 기본 값을 제공하지 않을 경우, portSUPPRESS_TICKS_AND_SLEEP()을 제공할 수 있도록 하는 공용 API 함수이다.

vTaskStepTick()을 사용하려면 configRootConfig.h 의 configUSE_TICKLESS_IDLE 값을 1 로 설정해야 한다.

1.2.9.4.1 매개 변수

xTicksToJump : tick 인터럽트가 중지되고 다시 시작될 때까지 경과한 RTOS tick 주기(tick 인터럽트가 억제된 시간)의 수다. 올바른 조작을 위해서는 매개 변수가 portSUPPRESS_TICKS_AND_SLEEP() 매개 변수보다 작거나 같아야 한다.

1.2.9.4.2 기타

```
/* 이것은 portSUPPRESS_TICKS_AND_SLEEP()가 응용 프로그램 작성자에 의해 구현되는 방법의 예이다.
이 기본 구현은 일정 시간과 관련하여 커널이 유지 관리하는 시간을 추적하는데 있어서 부정확성을 초래한다.
vTaskStepTick()만이 FreeRTOS API 의 일부이다. 다른 함수는 데모 용이다. */

// portSUPPRESS_TICKS_AND_SLEEP() 매크로를 정의한다. 매개 변수는 다음 커널을 실행할 때까지의 tick 이다.
#define portSUPPRESS_TICKS_AND_SLEEP( xIdleTime ) vApplicationSleep( xIdleTime )
```

```
// portSUPPRESS_TICKS_AND_SLEEP()에 의해 호출되는 함수를 정의한다.
void vApplicationSleep( TickType_t xExpectedIdleTime ) {
    unsigned long ulLowPowerTimeBeforeSleep, ulLowPowerTimeAfterSleep;

    // 마이크로 컨트롤러가 저전력 상태일 때 작동 상태를 유지하는 시간 소스에서 현재 시간을 읽는다.
    ulLowPowerTimeBeforeSleep = ulGetExternalTime();

    // tick 인터럽트를 발생시키는 타이머를 중지한다.
    prvStopTickInterruptTimer();

    // 커널이 다음에 실행될 필요가 있을 때 마이크로 컨트롤러를 저전력 상태에서 빠져나오도록 인터럽트 설정을 한다.
    vSetWakeTimeInterrupt( xExpectedIdleTime );

    // 저전력 상태로 들어간다.
    prvSleep();

    /* 마이크로 컨트롤러가 vSetWakeTimeInterrupt()호출로 구성된 인터럽트가 아닌 다른 인터럽트로 인해 저전력 모드에서
    해제된 경우, 실제 저전력 상태에 있었던 시간을 결정한다. 이 값은 xExpectedIdleTime 보다 작다.
    portSUPPRESS_TICKS_AND_SLEEP() 호출되기 전에 스케줄러가 일시 중단되고, portSUPPRESS_TICKS_AND_SLEEP()이
    반환될 때 다시 시작된다. 따라서 이 함수가 완료될 때까지 다른 task 는 실행되지 않는다. */
    ulLowPowerTimeAfterSleep = ulGetExternalTime();

    // 마이크로 컨트롤러가 저전력 상태에서 소비한 시간을 설명하기 위해 커널 tick 계수를 수정한다.
    vTaskStepTick( ulLowPowerTimeAfterSleep - ulLowPowerTimeBeforeSleep );

    // tick 인터럽트를 생성하는 타이머를 다시 시작한다.
    prvStartTickInterruptTimer();
}
```

1.2.9.5 uxTaskPriorityGet : 우선 순위 출력

```
#include “FreeRTOS.h”
#include “task.h”

UBaseType_t uxTaskPriorityGet ( TaskHandle_t pxTask );
```

uxTaskPriorityGet()이 호출될 때 task 에 할당된 우선순위를 출력한다.

1.2.9.5.1 매개 변수

pxTask : 할당 우선순위를 알고자 하는 task 의 핸들이다. NULL 을 전달하여 자신의 우선 순위를 확인할 수 있다.

1.2.9.5.2 반환 값

uxTaskPriorityGet()이 호출될 때 질문하는 task 의 우선순위이다.

1.2.9.5.3 기타

```
void vAFunction( void ) {
        TaskHandle_t xHandle;
        UBaseType_t uxCreatedPriority, uxOurPriority;

        // task 를 생성하고 핸들을 저장한다.
        if( xTaskCreate( vTaskCode, "Demo task", STACK_SIZE, NULL, PRIORITY, &xHandle ) != pdPASS ) {
```

```
            // 생성 실패하였다.
        } else {
            // 핸들을 사용하여 생성된 task 의 우선 순위를 출력한다.
            uxCreatedPriority = uxTaskPriorityGet( xHandle );

            // NULL 을 사용하여 호출하는 task 의 우선순위를 출력한다.
            uxOurPriority = uxTaskPriorityGet( NULL );

            // 두 task 의 우선 순위를 비교한다.
            if( uxOurPriority > uxCreatedPriority ) {
                // 더 높다.
            }
        }
}
```

1.2.9.6 vTaskPrioritySet : task 우선순위 변경

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskPrioritySet ( TaskHandle_t pxTask, UBaseType_t uxNewPriority );
```

task 의 우선 순위를 변경한다.

vTaskPrioritySet()은 실행중인 task 에서만 호출되어야 하므로 초기화 상태(스케줄러 시작 전)에서는 호출하면 안된다.

큐 또는 세마포어 이벤트를 기다리는 동안 차단된 task 집합을 가질 수 있다. 이러한 작업은 우선 순위에 따라 정렬된다.

예를 들어, 첫번째 이벤트는 이벤트를 기다리고 있던 가장 높은 task 를 차단을 해제하고, 두번째 이벤트는 원래 이벤트를 기다리고 있던 두번째 우선순위 task 의 차단을 해제한다.

vTaskPrioritySet()를 사용하여 차단된 task 의 우선순위를 변경해도 위에 정렬된 task 의 순서가 변경되지 않는다.

1.2.9.6.1 매개 변수

pxTask : 수정하고자 하는 task 의 핸들이다. NULL 입력시 호출한 task 의 우선순위를 변경한다.

uxNewPriority : 변경할 task 의 우선순위이다. 이 우선순위는 가장 낮은 0 부터 가장 높은 우선순위(configMAX_PRIORITIES - 1)까지 할당할 수 있다.

configMAX_PRIORITIES 는 FreeRTOSConfig.h 에 정의되어 있다. 설정된 값 보다 위의 값을 전달하면 task 에 할당할 수 있는 가장 높은 값으로 제한된다.

1.2.9.6.2 기타

```
void vAFunction( void ) {
    TaskHandle_t xHandle;

    // task 를 생성하고, 핸들을 저장한다.
    if( xTaskCreate( vTaskCode, "Demo task", STACK_SIZE, NULL, PRIORITY, &xHandle ) != pdPASS ) {
        // task 생성에 실패했다.
    } else {
        // 핸들을 사용하여 생성된 task 의 우선순위를 올린다.
        vTaskPrioritySet( xHandle, PRIORITY + 1 );
```

```
            // 유효한 task 핸들 대신 NULL 을 사용하여 호출하는 task 의 우선순위를 1 로 설정한다.
            vTaskPrioritySet( NULL, 1 );
        }
}
```

1.2.9.7 xTaskGetSchedulerState : 스케줄러 상태 값 리턴

```
#include "FreeRTOS.h"
#include "task.h"

BaseType_t xTaskGetSchedulerstate ( void );
```

xTaskGetSchedulerstate()가 호출될 때 스케줄러의 상태를 나타내는 값을 반환한다.

xTaskGetSchedulerstate()를 사용하기 위해 FreeRTOSConfig.h 의 INCLUDE_xTaskGetSchedulerState 를 1 로 설정해야 한다.

1.2.9.7.1 반환 값

taskSCHEDULER_NOT_STARTED : 이 값은 vTaskStartScheduler()가 호출되기 전에 xTaskGetSchedulerState()를 호출하면 리턴 한다.

taskSCHEDULER_RUNNING : 스케줄러가 Suspended 상태가 아닐때, xTaskGetSchedulerState()를 호출하면 리턴 한다.

taskSCHEDULER_SUSPENDED : vTaskSuspendAll()이 호출되어 스케줄러가 중단된 상태일때 리턴 한다.

1.2.9.8 vTaskStartScheduler : 스케줄러 시작

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskStartScheduler ( void );
```

실행중인 FreeRTOS 스케줄러를 시작한다. 일반적으로 스케줄러가 시작되기 전에 main()(또는 main()이 호출하는 함수)이 실행된다.

스케줄러가 시작된 후에는 task 와 인터럽트만 실행된다. 스케줄러를 시작하면 스케줄러가 초기화 상태에 있는 동안 작성된 우선순위 task 가 실행 상태가 된다.

ARM7 및 ARM9 마이크로 컨트롤러에서 실행되는 포트는 vTaskStartScheduler()가 호출되기 전에 프로세서가 관리자 모드여야 한다.

1.2.9.8.1 반환 값

Idle task 는 스케줄러가 시작될 때 자동으로 설정된다. vTaskStartScheduler()는 Idle task 가 만들 수 있는 FreeRTOS 힙 메모리가 충분하지 않을 경우에만 반환한다.

1.2.9.8.2 기타

```
TaskHandle_t xHandle;

// task 함수를 정의한다.
void vATask( void ) {
    for( ;; ) {
      // 작업 코드를 작성한다.
```

```
    }
}

void main( void ) {
    // 적어도 하나의 task 를 생성한다. 이 예제는 위에 정의된 task 를 생성한다.
    // task 를 생성하기 전에 vTaskStartScheduler() 호출을 하면 Idle task 가 실행 상태가 된다.
    xTaskCreate( vTaskCode, "task name", STACK_SIZE, NULL, TASK_PRIORITY, NULL );

    // 스케줄러를 시작한다.
    vTaskStartScheduler();

    // 이 코드는 vTaskStartScheduler() 내에서 Idle task 를 만들 수 없는 경우에만 전달된다.
    // 무한 루프는 이 시나리오가 main()을 종료하지 않도록 하여 디버깅을 돕는데 사용된다.
    for( ;; );
}
```

1.2.9.9 vTaskSuspendAll : 스케줄러 일시 중지

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskSuspendAll ( void );
```

스케줄러를 일시 중지한다. 스케줄러를 일시 중지하면 컨텍스트 스위칭이 발생하지 않지만, 인터럽트는 사용이 가능하다. 인터럽트는 스케줄러가 일시 중단된 동안 컨텍스트 스위치를 요청할 경우, 요청은 보류 중으로 유지되고 스케줄러가 다시 시작될 때(일시 중지 해제)에만 수행된다.

vTaskSuspendAll() 호출은 xTaskResumeAll() 호출로 일시 중지 상태에서 해제되며, 컨텍승트 스위치가 가능하다. vTaskSuspendAll()호출은 중첩이 가능하다. 스케줄러가 일시 중지 상태를 떠나서 동작 상태로 돌아가려면 vTaskSuspendAll()에 의해 중첩된 수만큼 xTaskResumeAll()의 수를 요구한다.

xTaskResumeAll()은 실행중인 task 에서만 호출되어야 하므로 스케줄러가 초기화 상태(스케줄러가 시작되기 전)에서는 호출하면 안된다.

다른 FreeRTOS API 함수는 스케줄러가 일시 중단된 동안 호출되면 안된다.

1.2.9.9.1 기타

```
// 일시 중지 후 스케줄러를 다시 시작하는 함수이다.
void vDemoFunction( void ) {
    // 이 함수는 스케줄러를 일시 중단한다. vTask1 에서 호출될 때 스케줄러는 이미 일시 중지되어 있으므로, 호출의 중첩은 2 다.
    vTaskSuspendAll();

    // 여기서 작업을 수행한다.
    // vTaskSuspendAll()이 중첩되어 있으므로, 스케줄러가 활성화 상태로 돌아가지 않는다.
    xTaskResumeAll();
}

void vTask1( void * pvParameters ) {
    for( ;; ) {
      // 여기서 작업을 수행한다.
      /* 어떤 시점에서 task 는 swapped out 을 원하지 않는 task 를 수행하려고 하거나,
      다른 task(인터럽트가 아닌)에서 엑세스 하는 데이터를 엑세스 하려고 한다.
      연산 길이로 인해 누락될 수 있으므로 taskENTER_CRITICAL() / taskEXIT_CRITICAL()을 사용할 수 없다.  */
      // 스케줄러가 컨텍스트 스위치를 수행하지 못하도록 한다.
      vTaskSuspendAll();
```

```
        /* 여기서 작업을 수행한다. task 가 인터럽트 서비스 루틴에 의해 사용된 것 이외의 모든 처리 시간을 가지므로
        크리티컬 섹션을 사용할 필요가 없다. vTaskSuspendAll()에 대한 호출은 중첩될 수 있으므로, vTaskSuspendAll()에 대한
        호출을 포함하는 함수(비 API)를 호출하는 것이 안전하다. API 함수는 스케줄러가 일시 중단된 동안 호출되어서는 안된다. */
        vDemoFunction();

        // 작업이 완료된 후, 스케줄러를 다시 활성화 상태로 설정한다.
        if( xTaskResumeAll() == pdTRUE ) {
          // 컨텍스트 스위치가 xTaskResumeAll()내에서 발생했다.
        } else {
          // 컨텍스트 스위치가 xTaskResumeAll()내에서 발생하지 않았다.
        }
    }
}
```

1.2.9.10 taskYIELD : 동등한 우선순위의 다른 task 실행

```
#include "FreeRTOS.h"
#include "task.h"

void taskYIELD ( void );
```

우선순위가 동일한 다른 task 를 수행한다.

이 task 는 실행 상태에서 유연하게 다른 task 로 지원할 수 있으며, 선점형 스케줄링을 통해, 타임 슬라이스를 사용할 수 있다.

taskYIELD()는 실행중인 task 에서만 호출되므로 스케줄러가 초기화 상태(스케줄러가 시작되기 전)에서는 호출하면 안된다. task 가 taskYIELD()를 호출하면 스케줄러는 동등한 우선순위의 다른 task 를 선택하여 실행한다. 우선순위가 다른 task 가 없는 경우, taskYIELD()를 호출한 task 가 곧바로 실행 상태로 전환된다. 스케줄러는 taskYIELD()를 호출한 task 와 동일한 우선순위 task 만을 선택한다. 왜냐하면 준비 상태에 있는 보다 높은 우선순위의 task 가 있다면 taskYIELD()를 호출하기 전에 실행되기 때문이다.

1.2.9.10.1 기타

```
void vATask( void * pvParameters) {
    for( ;; ) {
      /* 작업을 수행한다. 이 task 와 동일한 우선순위의 task 가 준비 상태에 있으면 실행한다.
      이 작업은 time slice 를 사용하지 않은 경우에도 마찬가지다. */

      taskYIELD();
      // 준비 상태에서 task 와 동일한 우선순위의 task 가 있는 경우, task 가 여기에 도달하기 전에 실행된다.
    }
}
```

1.2.10 디버그

1.2.10.1 uxTaskGetNumberOfTasks : 현재 존재하는 task 의 수 출력

```
#include “FreeRTOS.h”
#include “task.h”

UBaseType_t uxTaskGetNumberOfTasks ( void );
```

uxTaskGetNumberOfTasks()가 호출될 때 존재하는 모든 task 의 수를 반환한다.

1.2.10.1.1 반환 값

반환된 값은 uxTaskGetNumberOfTasks()가 호출될 때 FreeRTOS 커널이 제어하고 있는 task 의 수다. Suspended 상태의 task 수와 차단 상태의 task 수, 준비 상태의 task 수, Idle 상태의 task 수, 실행 상태의 task 수이다.

1.2.10.2 uxTaskGetStackHighWaterMark : task 의 남아있는 스택 공간 출력

```
#include “FreeRTOS.h”
#include “task.h”

UBaseType_t uxTaskGetStackHighWaterMark ( TaskHandle_t xTask );
```

각 task 는 자신의 스택을 유지 관리하며, 이 스택의 총 크기는 task 생성시 지정된다.

uxTaskGetStackHighWaterMark()는 task 에 할당된 스택 공간이 오버 플로우 하기까지 얼마나 남았는지를 리턴 한다.

이 값을 스택의 ‘high water mark’라고 한다.

uxTaskGetStackHighWaterMark()는 실행하는데 있어 비교적 오랜 시간이 걸리기 때문에 빌드를 테스트하는 디버그 용도로만 사용하는 것이 좋다.

uxTaskGetStackHighWaterMark()를 사용하려면 FreeRTOSConfig.h 의 INCLUDE_uxTaskGetStackHighWaterMark 를 1 로 설정해야 한다.

1.2.10.2.1 매개 변수

xTask : high water mark 를 조회하고자 하는 task 의 핸들이다. NULL 을 전달 시 해당 task 의 high water mark 를 확인할 수 있다.

1.2.10.2.2 반환 값

task 에 사용되는 스택의 양은 task 가 실행된 후 인터럽트 처리에 따라 커지고 줄어든다.

uxTaskGetStackHighWaterMark()는 task 실행이 시작된 이후 사용 가능한 가장 작은 스택의 공간을 반환한다. 이것은 사용량이 가장 큰 값일 때 남아있는 스택의 양이다. high water mark 가 0 에 가까울수록 오버 플로우에 근접한 상태이다.

1.2.10.2.3 기타

```
void vTask1( void * pvParameters ) {
        UBaseType_t uxHighWaterMark;
        // task 실행 전 high water mark 를 검사한다.
        uxHighWaterMark = uxTaskGetStackHighWaterMark ( NULL );
```

```
        for( ;; ) {
            // 함수를 호출한다.
            vTaskDelay( 1000 );

            // 함수를 호출하면 스택 공간이 사용되기 때문에 uxTaskGetStackHighWaterMark()는 처음보다 더 낮은 값을 리턴 한다.
            uxHighWaterMark = uxTaskGetStackHighWaterMark( NULL );
        }
}
```

1.2.10.3 pvTaskGetThreadLocalStoragePointer : TLS 인덱스 값의 결과 출력

```
#include “FreeRTOS.h”
#include “task.h”

void * pvTaskGetThreadLocalStoragePointer ( TaskHandle_t xTaskToQuery, BaseType_t xIndex );
```

스레드 로컬 저장소(TLS)를 사용하면 응용 프로그램 작성자가 task 제어 블록 내에 값을 저장하고, 각 task 에 고유한 값을 만들고, 고유한 값을 가질 수 있다. 각 task 에는 스레드 로컬 저장소로 사용할 수 있는 자체 포인터 배열이 있다. 배열의 인덱스 수는 FreeRTOSConfig.h 의 configNUM_THREAD_LOCAL_STORAGE_POINTERS 컴파일 시간 구성 상수에 의해 정의된다.

pvTaskGetThreadLocalStoragePointer()는 배열의 인덱스에서 값을 읽어 효과적으로 스레드 로컬 값을 검색한다.

스레드 로컬 저장소(TLS)는 시스템 내에서 논리적으로 동일한 변수를 스레드마다 별도의 메모리 공간에 저장할 수 있다.

(스레드 전용 스택으로, 별도의 스택으로 이루어져 있다.)

1.2.10.3.1 매개 변수

xTaskToQuery : 스레드 로컬 데이터를 읽을 task 핸들이다. 매개 변수에 NULL 을 입력 시 해당 task 의 스레드 로컬 데이터를 읽은다.

xIndex : 데이터가 읽히는 스레드 로컬 저장소 배열에 대한 인덱스이다.

1.2.10.3.2 반환 값

task 의 스레드 로컬 저장소 배열로부터 인덱스 xIndex 값을 읽어 낸 값이다.

1.2.10.3.3 기타

```
uint32_t ulVariable;

// 호출한 task 의 스레드 로컬 저장소 배열 인덱스 값 5 에 저장된 것을 ulVariable 로 읽어온다.
ulVariable = ( uint32_t ) pvTaskGetThreadLocalStoragePointer( NULL, 5 );
```

1.2.10.4 vTaskGetRunTimeStats : 실행 시간 통계 테이블 출력

```
#include “FreeRTOS.h”
#include “task.h”

void vTaskGetRunTimeState ( char * pcWriteBuffer );
```

FreeRTOS 는 task 실행시간의 통계를 수집할 수 있다. task 실행시간 통계는 task 가 수신한 처리 시간에 대한 정보를 제공한다.

수치는 전체 응용 프로그램 실행 시간의 절대 시간과 백분율로 제공된다.

vTaskGetRunTimeStats() API 함수는 수집된 실행 시간 통계를 사람이 읽을 수 있는 테이블로 형식화하여 제공한다.

task 이름과 할당된 절대 시간, 해당 task 에 할당된 전체 응용 프로그램 실행시간의 백분율에 대한 열이 생성된다.

Idle task 를 포함하여 시스템의 각 task 에 대해 행이 형성되는데 출력의 예는 아래와 같다.

```
Task            Abs Time      % Time
****************************************


uIP             12050         <1%
IDLE            587724        24%
QProdB2         2172          <1%
QProdB3         10002         <1%
QProdB5         11504         <1%
QConsB6         11671         <1%
PolSEM1         60033         2%
PolSEM2         59957         2%
IntMath         349246        14%
MuLow           36619         1%
```

vTaskGetRunTimeStats()는 편의상 제공되는 유틸리티 함수이다. 커널의 일부로 간주되지 않는다.

vTaskGetRunTimeStats()는 xTaskGetSystemState() API 함수를 사용하여 원시 데이터를 가져온다.

vTaskGetRunTimeStats()함수를 사용하려면 FreeRTOSConfig.h 에서 configUSE_STATS_FORMATTING_FUNCTIONS 와 configGENERATE_RUN_TIME_STATS 를 1 로 설정해야 한다.

configGENERATE_RUN_TIME_STATS 를 설정하려면 응용프로그램에서 다음 매크로를 정의해야 한다.

매크로는 FreeRTOSConfig.h 에서 정의한다.

portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() : 이 매크로는 time base 기반을 생성하는데 사용되는 주변장치를 초기화 하기 위해 제공된다. 실행 시간의 통계에 사용되는 기준은 tick 인터럽트보다 높은 해상도를 가져야한다. 그렇게 하지 않으면 수집된 통계가 부정확하기 때문에 실제로 사용할 수 없다. tick 인터럽트보다 10~20 배 정도 빠른 time base 를 만드는 것이 좋다.

아래 2 개의 매크로 함수는 현재 시간 축을 기준으로 값을 반환한다. 즉, 선택한 시간 축으로 응용프로그램이 실행된 총 시간을 나타낸다.

portGET_RUN_TIME_COUNTER_VALUE() : 응용 프로그램이 실행된 총 시간이다.

portALT_GET_RUN_TIME_COUNTER_VALUE(Time) : Time 매개 변수를 현재 시간의 기준 값으로 설정하여 계산한다.

1.2.10.4.1 매개 변수

pcWriteBuffer : 사람이 읽을 수 있는 형식화된 테이블이 들어가 있는 문자 버퍼에 대한 포인터이다. Boundary Checking 가 수행되지 않으므로 전체 테이블을 보유할 만큼 버퍼가 커야 한다.

1.2.10.4.2 기타

```
/* LM3Sxxxx Eclipse 데모 애플리케이션에는 이미 20KHz 타이머 인터럽트가 포함되어 있다.
해당 인터럽트는 실행될 때마다 ulHighFrequencyTimerTicks 이라는 변수를 단순 증가시키는 작업을 한다.
portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()로 변수를 0 으로 설정하고, portGET_RUN_TIME_COUNTER_VALUE()로 값을
반환한다. 이 동작을 위하여 FreeRTOSConfig.h 에 아래 내용이 추가되어 있다. */
extern volatile unsigned long ulHighFrequencyTimerTicks;

// ulHighFrequencyTimerTicks 이 이미 20KHz 로 증가중이기 때문에 그 값을 0 으로 초기화한다.
#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() ( ulHighFrequencyTimerTicks = 0UL )

// 주파수의 카운터 값을 반환한다.
#define portGET_RUN_TIME_COUNTER_VALUE() ulHighFrequencyTimerTicks
```

```
/* LPC17xx 데모 애플리케이션에는 고주파 인터럽트가 포함되어 있지 않으므로,
portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()은 주변 장치의 time base 의 타이머가 0 을 구성하는데 사용된다.
portGET_RUN_TIME_COUNTER_VALUE()는 현재 타이머인 카운터 값을 반환한다.
이것들은 매크로를 통해 구현되어 있다. */

// main.c 에 정의되어 있다.
void vConfigureTimerForRunTimeStats ( void ) {
	const unsigned long TCR_COUNT_RESET = 2, CTCR_CTM_TIMER = 0x00, TCR_COUNT_ENABLE = 0x01;

	// 전원을 켜고 타이머에 클럭을 넘긴다.
	PCONP |= 0x02UL;
	PCLKSEL0 = (PCLKSEL0 & (~(0x3 << 2))) | (0x01 << 2);

	// 타이머 리셋
	T0TCR = TCR_COUNT_RESET;

	// 카운트 업
	T0CTCR = CTCR_CTM_TIMER;

	// 높은 해상도를 얻을 수록 좋지만, 오버 플로우가 발생하지 않도록 주파수를 정한다.
	T0PR = ( configCPU_CLOCK_HZ / 10000UL ) - 1UL;

	// 카운터를 시작한다.
	T0TCR = TCR_COUNT_ENABLE;
}

// FreeRTOSConfig.h 에 정의되어 있다.
extern void vConfigureTimerForRunTimeStats ( void );
#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() vConfigureTimerForRunTimeStats()
#define portGET_RUN_TIME_COUNTER_VALUE() T0TC
```

```
void vAFunction( void ) {
	// 생성된 테이블을 담을 수 있는 충분한 양의 버퍼를 정의한다.
	// 대부분의 경우 버퍼가 너무 커서 스택에 할당할 수 없으므로 이 예제에서는 정적으로 선언한다.
	static char cBuffer[ BUFFER_SIZE ];

	// 버퍼를 vTaskGetRunTimeStats()에 전달하여 데이터 테이블을 생성한다.
	vTaskGetRunTimeStats ( cBuffer );

	// 여기에 생성된 정보를 저장하거나 볼 수 있다.
}
```

1.2.10.5 vTaskList : task 의 상태를 출력

```
#include “FreeRTOS.h”
#include “task.h”

void vTaskList ( char * pcWriteBuffer );
```

vTaskList()가 호출될 때 각 task 의 상태를 사람이 읽을 수 있도록 테이블을 문자 버퍼로 작성한다.

```
Name          State   Priority  Stack   Num
**********************************************
Print           R       4       331     29
Math7           R       0       417     7
Math8           R       0       407     8
QConsB2         R       0       53      14
QProdB5         R       0       52      17
QConsB4         R       0       53      16
SEM1            R       0       50      27
SEM1            R       0       50      28
IDLE            R       0       64      0
Math1           R       0       436     1
Math2           R       0       436     2
```

이 버퍼에는 다음과 같은 정보가 존재한다.

Name : task 생성 시 주어진 텍스트 이름이다.

State : vTaskList()호출 당시 task 들의 상태를 나타낸다.

- X : 실행 상태이다.(vTaskList()를 호출한 task)

- B : 차단 상태이다.

- R : 준비 상태이다.

- S : task 가 일시 중지 상태이거나 제한시간 없이 차단 상태이다.

- D : task 가 삭제되었지만 idle task 가 task 데이터 구조와 스택을 보유하고 사용중인 메모리를 아직 해제하지 않은 상태이다.

Priority : vTaskList()호출 시 task 에 할당된 우선순위 이다.

Stack : task 스택의 high water mark 를 표시한다.

Num : 이것은 각 작업에 할당된 고유 번호이다. 하나 이상의 task 에 동일한 이름이 지정되었을 때 task 를 식별하는 것을 돕는 것 외에는 아무런 기능도 없다.

vTaskList()는 편의상 제공되는 유틸리티 함수이다. 커널의 일부로 간주되지 않는다. vTaskList()는 xTaskGetSystemState()를 사용하여 원시 데이터를 가져온다. vTaskList()는 실행 기간동안 인터럽트를 비활성화 한다. 그렇기 때문에 실시간 기능이 포함된 응용 프로그램에서는 적합하지 않을 수 있다.

vTaskList()를 사용하려면 FreeRTOSConfig.h 에서 configUSE_TRACE_FACILITY 및 configUSE_STATS_FORMATTING_FUNCTIONS 를 1 로 설정해야 한다.

기본적으로 vTaskList()는 표준 라이브러리 sprintf()함수를 사용한다. 이로 인해 컴파일 된 이미지 크기와 스택 사용량이 크게 증가할 수 있다. FreeRTOS 다운로드에는 printf-stdarg.c 라는 파일에 sprintf()의 오픈 소스 버전이 포함되어 있다. 표준 라이브러리 sprintf()대신 이 코드를 사용하여 코드 크기의 영향을 최소화 할 수 있다. printf-stdarg.c 는 FreeRTOS 에 대해 별도로 라이센스가 부여된다. 라이센스 조항은 파일 자체에 포함되어 있다.

1.2.10.5.1 매개 변수

pcWriteBuffer : 테이블 텍스트가 쓰여지는 버퍼이다. 이는 경계 검사가 수행되지 않으므로 전체 테이블을 보유할 만큼 버퍼가 충분히 커야한다.

1.2.10.5.2 기타

```
void vAFunction( void ) {
        // 생성된 테이블을 담을 만큼 충분히 큰 버퍼를 정의한다.
        // 대부분의 경우 버퍼가 너무 커서 스택을 할당할 수 없으므로 예제에서는 정적으로 선언하였다.
        static char cBuffer[ BUFFER_SIZE ];

        // 버퍼를 vTaskList()에 전달하여 정보 테이블을 생성한다.
        vTaskList( cBuffer );

        // 버퍼를 이용하여 생성된 정보를 볼 수 있다.
}
```

1.2.10.6 vTaskGetTaskInfo : 단일 task 정보 출력

```
#include “FreeRTOS.h”
#include “task.h”

UBaseType_t vTaskGetTaskInfo ( TaskHandle_t xTask,
                               TaskStatus_t * pxTaskStatus,
                               BaseType_t xGetFreeStackSpace,
                               eTaskState eState );
```

vTaskGetTaskInfo()는 단일 task 에 대한 taskStatus_t 구조체를 채운다.

taskStatus_t 구조체는 task 의 핸들, 이름, 우선순위, 상태 및 소비된 총 실행 시간을 포함한다.

이 함수는 스케줄러가 오랜 시간 정지된 채 남아있기 때문에 디버깅 용도로만 사용한다.

uxTaskGetSystemState()를 사용하기 위해선 FreeRTOSConfig.h 의 configUSE_TRACE_FACILITY 를 1 로 설정해야 한다.

1.2.10.6.1 매개 변수

xTask : 확인하고자 하는 task 의 핸들이다.

pxTaskStatus : taskStatus_t 유형의 변수를 가리키며, 확인하고자 하는 정보로 채워진다.

xGetFreeStackSpace : TaskStatus_t 구조체에는 확인하고자 하는 task 의 스택의 남은 공간을 알려주는 멤버가 포함되어 있다. 이 값은 0 에 가까울수록 오버 플로우에 가깝다. 스택의 남은 공간을 계산함에 있어서 비교적 오랜 시간이 걸리기 때문에 시스템이 일시적으로 응답하지 않을 수 있다. 따라서 xGetFreeStackSpace 매개 변수를 제공하여 high water mark 확인을 건너 뛸 수 있다. high water mark 는 xGetFreeStackSpace 가 pdFALSE 로 설정되지 않은 경우 TaskStatus_t 구조체에만 기록된다.

eState : TaskStatus_t 구조체에 task 의 상태를 보고하는 멤버가 포함되어 있다. task 상태를 얻는 것은 빠르게 할 수 없기 때문에, eState 매개 변수를 통하여 TaskStatus_t 구조체에 상태 정보를 생략할 수 있도록 제공한다. 상태정보를 얻으려면 eState 를 eInvalid 로 설정한다. 그렇지 않으면 eState 에 전달된 값이 TaskStatus_t 구조체의 task 상태로 보고된다.

1.2.10.6.2 기타

```
void vAFunction( void ) {
        TaskHandle_t xHandle;
        TaskStatus_t xTaskDetails;

        // task 의 핸들을 얻는다.
        xHandle = xTaskGetHandle( "Task_Name" );

        // 핸들이 NULL 이 아닌지 확인한다.
        configASSERT( xHandle );

        // 핸들을 사용하여 작업에 대한 정보를 얻는다.
        // 핸들, TaskStatus_t 구조체, 구조체에 high water mark 를 포함시킨다, 구조체에 task 상태 표시를 한다.
        vTaskGetTaskInfo( xHandle, &xTaskDetails, pdTRUE, eInvalid );
}
```

1.2.10.7 uxTaskGetSystemState : 시스템 전체 task 정보 출력

```
#include “FreeRTOS.h”
#include “task.h”

UBaseType_t uxTaskGetSystemState ( TaskStatus_t * const pxTaskStatusArray,
                                   const UBaseType_t uxArraySize,
                                   unsigned long * const pulTotalRunTime );
```

uxTaskGetSystemState()는 시스템의 각 task 에 대한 taskStatus_t 구조체를 채운다.

taskStatus_t 구조체는 task 의 핸들, 이름, 우선순위, 상태 및 사용된 총 실행 시간을 포함한다.

이 함수는 스케줄러가 오랜 시간 정지된 채 남아있기 때문에 디버깅 용도로만 사용한다.

시스템의 모든 작업 대신 단일 작업에 대한 정보를 얻으려면 vTaskGetTaskInfo()를 사용한다.

uxTaskGetSystemState()를 사용하기 위해선 FreeRTOSConfig.h 의 configUSE_TRACE_FACILITY 를 1 로 설정해야 한다.

1.2.10.7.1 매개 변수

pxTaskStatusArray : taskStatus_t 구조체의 배열에 대한 포인터다. 이 구조체는 RTOS 가 제어하는 각 task 에 대해 하나 이상의 taskStatus_t 가 포함되어 있어야 한다. RTOS 의 제어 하에 있는 task 의 수는 uxTaskGetNumberOfTasks()를 이용하여 결정할 수 있다.

uxArraySize : pxTaskStatusArray 매개 변수가 가리키는 배열의 크기이다. 크기는 배열의 바이트 수가 아닌 배열의 인덱스 수로 지정된다.

pulTotalRunTime : configGENERATE_RUN_TIME_STATS 를 1 로 설정하면 pulTotalRunTime 은 대상이 부팅된 이후 uxTaskGetSystemState()에 의해 정의된 총 실행 시간으로 설정된다.

1.2.10.7.2 반환 값

uxTaskGetSystemState()에 의해 채워진 taskStatus_t 구조체의 숫자이다.

이 값은 uxTaskGetNumberOfTasks()함수에서 반환하는 숫자와 같아야 하지만, uxArraySize 매개 변수에 전달된 값이 너무 작으면 0 을 반환한다.

1.2.10.7.3 기타

```
/* 해당 예제는 uxTaskGetSystemState()에서 제공하는 원시 데이터에서 사람이 읽을 수 있는 런타임 통계 정보 테이블로 생성되는
방법을 보여준다. 사람이 읽을 수 있는 테이블은 pcWriteBuffer 에 저장한다. (실제 작업 수행은 vTaskList()를 참조한다.) */
void vTaskGetRunTimeStats( signed char *pcWriteBuffer ) {
        TaskStatus_t *pxTaskStatusArray;
        volatile UBaseType_t uxArraySize, x;
        unsigned long ulTotalRunTime, ulStatsAsPercentage;

        // 쓰기 버퍼에 문자열이 있는지 확인한다.
        *pcWriteBuffer = 0x00;

        // 이 함수가 실행되는 동안, 변경될 경우를 대비해 여러 task 에 대한 스냅 샷을 만든다.
        uxArraySize = uxTaskGetNumberOfTasks();

        // 각 task 에 대한 TaskStatus_t 구조체를 할당한다. 해당 배열을 정적으로 할당할 수 있다.
        pxTaskStatusArray = pvPortMalloc( uxArraySize * sizeof( TaskStatus_t ) );

        if( pxTaskStatusArray != NULL ) {
            // 각 작업에 대한 원시 정보를 생성한다.
            uxArraySize = uxTaskGetSystemState( pxTaskStatusArray, uxArraySize, &ulTotalRunTime );

            // 퍼센트 계산
            ulTotalRunTime /= 100UL;

            //  0 은 에러이다.
            if( ulTotalRunTime > 0 ) {
                // pxTaskStatusArray 배열로 채워진 원시 데이터들을 사람이 읽을 수 있는 ASCII 데이터로 포맷한다.
                for( x = 0; x < uxArraySize; x++ ) {
                    // task 가 사용된 시간을 전체 실행시간의 퍼센트로 처리한다. 이때 퍼센트는 정수로 내림 계산된다.
                    ulStatsAsPercentage = pxTaskStatusArray[ x ].ulRunTimeCounter / ulTotalRunTime;

                    if( ulStatsAsPercentage > 0UL ) {
                        sprintf( pcWriteBuffer, "%s₩t₩t%lu₩t₩t%lu%%₩r₩n",
                                pxTaskStatusArray[ x ].pcTaskName,
                                pxTaskStatusArray[ x ].ulRunTimeCounter,
                                ulStatsAsPercentage );
                    } else {
                        // 백분율이 0 이면 작업의 총 실행 시간은 1% 미만이다.
                        sprintf( pcWriteBuffer, "%s₩t₩t%lu₩t₩t<1%%₩r₩n”,
                                pxTaskStatusArray[ x ].pcTaskName,
                                pxTaskStatusArray[ x ].ulRunTimeCounter );
                    }
                    pcWriteBuffer += strlen( ( char * ) pcWriteBuffer );
                }
            }
            // 배열이 더이상 필요 없기 때문에 소비하는 메모리를 비운다.
            vPortFree( pxTaskStatusArray );
        }
}
```

```
typedef struct xTASK_STATUS {
        // 해당 구조체와 관련된 task 핸들이다.
        TaskHandle_t xHandle;

        // task 의 이름을 나타내는 포인터이다. 구조체가 채워진 이후 task 가 삭제되면 해당 값은 유효하지 않는다.
        const signed char *pcTaskName;
```

```
        // task 에 대한 고유 번호이다.
        UBaseType_t xTaskNumber;

        // 구조체가 채워 졌을 때 task 의 상태이다.
        eTaskState eCurrentState;

        // 구조체가 채워 졌을 때 task 실행의 우선순위이다.
        UBaseType_t uxCurrentPriority;

        /* 뮤텍스를 얻을 때, 우선순위 역전현상을 피하기 위해 task 가 우선순위를 상속된 경우(PIP(Priority Inheritance Protocol)),
        task 가 반환하는 우선순위이다. FreeRTOSConfig.h 의 configUSE_MUTEXES 를 1 로 설정해야 유효하다. */
        UBaseType_t uxBasePriority;

        // task 에 지금까지 할당된 총 실행 시간을 나타낸다.
        // FreeRTOSConfig.h 의 configGENERATE_RUN_TIME_STATS 가 1 일때만 유효하다.
        unsigned long ulRunTimeCounter;

        // task 스택 영역의 가장 낮은 주소를 나타낸다.
        StackType_t *pxStackBase;

        // task 생성 이후 task 내부에 남아있는 스택의 가장 작은 값이다. 0 에 가까울수록 오버 플로우에 가까워진다.
        unsigned short usStackHighWaterMark;
} TaskStatus_t;
```