

# USING THE FREERTOS REAL TIME KERNEL

## A Practical Guide

Richard Barry



# **Xilinx Zynq FPGA, TI DSP, MCU 프로그래밍 및 회로 설계 전문가 과정**

**Translation of Using FreeRTOS Doc v-0.0.1**

**강사 – Innova Lee(이상훈)**

**gcccompil3r@gmail.com**

# CHAPTER1

# TASK MANAGEMENT

## 1. CHAPTER INTRODUCTION AND SCOPE

[부록은 FreeRTOS 소스코드 사용에 관한 실제 정보도 제공한다.]

### An Introduction to Multi Tasking in Small Embedded Systems

다른 멀티 태스킹 시스템은 다른 목표를 가지고 있다. 워크 스테이션 및 데스크탑을 예로 들면 다음과 같다

- '옛날' 프로세서는 비쌌으며 많은 사용자가 단일 프로세서에 액세스할 수 있도록 멀티태스킹이 사용되었다. 이러한 유형의 시스템에 사용된 스케줄링 알고리즘은 각 사용자가 처리 시간을 '공정한 분배'할 수 있도록 하기 위해 고안되었다.
- 최근에는 처리 능력이 떨어지면서 각 사용자가 하나 이상의 프로세서에 독점적으로 액세스할 수 있게 되었다. 이러한 시스템 유형의 스케줄링 알고리즘은 사용자가 컴퓨터가 응답하지 않아도 동시에 여러 응용 프로그램을 실행할 수 있도록 설계되었다. 예를 들어 사용자는 워드 프로세서, 스프레드 시트, 이메일 클라이언트 및 웹 브라우저를 동시에 실행할 수 있으며 모든 응용 프로그램이 항상 입력에 적절하게 응답할 것으로 기대한다.

데스크톱 컴퓨터의 입력 처리는 '소프트 리얼 타임'으로 분류할 수 있다. 최고의 사용자 경험을 보장하기 위해 컴퓨터는 원하는 시간 내에 각 입력에 응답해야 하지만 이 제한을 벗어난 응답은 컴퓨터를 쓸모 없게 만들지 않는다. 예를 들어, 키 누름은 키를 누른 특정 시간 내에 눈으로 확인할 수 있어야 한다. 이 시간 외에 키 누름을 등록하면 시스템이 응답하지 않는 것처럼 보이지만 사용할 수 없게 될 수 있다.

실시간 임베디드 시스템에서의 멀티 태스킹은 개념적으로 데스크톱 시스템의 멀티 태스킹과 유사하여 단일 프로세서를 사용하는 여러 실행 스레드를 설명한다. 그러나 실시간 임베디드 시스템의 목표는 특히 임베디드 시스템이 '하드 실시간' 동작을 제공할 것으로 예상되는 경우 데스크톱의 목표와 상당히 다를 수 있다.

하드 실시간 기능은 지정된 시간 제한 내에 완료해야 한다. 그렇게 하지 않으면 시스템의 절대적인 오류가 발생한다. 자동차의 에어백 트리거링 메커니즘은 하드 실시간 기능의 한 예이다. 에어백은 주어진 시간 제한 내에서 전개되어야 한다. 이 시간 제한을 벗어난 응답은 운전자가 피해야 할 부상을 입을 수 있다.

대부분의 임베디드 시스템은 하드 및 소프트 실시간 요구 사항을 모두 구현한다.

## A Note About Terminology

FreeRTOS 에서 각 실행 스레드는 '작업'이라고 한다. 임베디드 커뮤니티 내의 용어에 대한 합의 된 합의 는 절대적으로 없지만 이전 경험에 따라 스레드가 보다 구체적인 의미를 가질 수 있으므로 '스레드'에 '작업'을 선호한다.

## Scope

이 장은 독자들에게 다음에 대한 좋은 이해를 제공하고자 한다 :

- FreeRTOS 가 응용 프로그램 내의 각 작업에 처리 시간을 할당하는 방법.
- FreeRTOS 가 주어진 시간에 어떤 작업을 실행할지 선택하는 방법.
- 각 작업의 상대적 우선 순위가 시스템 동작에 미치는 영향.
- 작업이 존재할 수 있는 상태.

또한 독자는 다음을 잘 이해할 수 있다.

작업을 구현하는 방법.

하나 이상의 작업 인스턴스를 만드는 방법.

task 매개 변수를 사용하는 방법.

이미 생성 된 작업의 우선 순위를 변경하는 방법.

작업을 삭제하는 방법.

주기적 처리를 구현하는 방법.

유휴 작업이 실행되고 어떻게 사용될 수 있는지.

이 장에서 제시된 개념은 FreeRTOS 를 사용하는 방법과 FreeRTOS 응용 프로그램이 작동하는 방법을 이해하는 데 있어 기본이다. 따라서 이 책에서 가장 자세한 장이 된다.

## 1.2 TASK FUNCTIONS

작업은 C 함수로 구현된다. 특수한 점은 void 를 반환하고 void 포인터 매개 변수를 가져와야 하는 프로 토 타입뿐이다. 프로 토 타입은 Listing 1 에 나와 있다.

```
void ATaskFunction( void *pvParameters );
```

#### Listing 1 The task function prototype

각 과제는 그 자체로 작은 프로그램이다. 엔트리 포인트가 있으며 무한 루프 내에서 영원히 계속 실행되며 종료되지 않는다. 일반적인 태스크의 구조는 Listing 2 와 같다.

FreeRTOS 작업은 어떤 식으로든 구현 함수에서 복귀할 수 없도록 해야 한다. 'return'문을 포함해서는 안되며 함수의 끝을 지나서 실행할 수 없도록 해야 한다. 더 이상 필요하지 않은 작업은 명시적으로 삭제해야 한다. 이것은 Listing 2 에도 나와있다.

단일 태스크 함수 정의를 사용하여 원하는 수의 태스크를 작성할 수 있다. 생성된 각 태스크는 자체 스택이 있는 개별 실행 인스턴스이고 태스크 자체 내에 정의된 자동 (스택) 변수의 자체 복사본이다.

```
void ATaskFunction( void *pvParameters )
{
    /* 변수는일반함수처럼선언할수있다. 이함수를사용하여생성된작업의각인스턴스에는고유한
    iVariableExample 변수사본이있다. 변수가정적으로선언된경우에는
    참이되지않는다.이경우변수의복사본이하나만존재하며이복사본은생성된각작업인스턴스에서공유된다. */
    int iVariableExample = 0;

    /* 태스크는일반적으로무한루프처럼구현될것이다. */
    for( ;; )
    {
        /* 작업기능을구현하는코드는여기에있다. */
    }

    /* 작업구현이위의루프에서벗어나야합니까?
    이기능이끝나기전에작업을삭제해야한다. vTaskDelete () 함수에전달된 NULL 매개 변수는삭제할작업이호출 (this)
    작업임을나타낸다. */
    vTaskDelete( NULL );
}
```

#### Listing 2 The structure of a typical task function

## 1.3 TOP LEVEL TASK STATES

응용 프로그램은 많은 작업으로 구성될 수 있다. 애플리케이션을 실행하는 마이크로 컨트롤러가 단일 코어만을 포함한다면 주어진 시간에 실제로 하나의 작업만 실행될 수 있다. 이는 실행중인

상태와 실행 중이지 않은 상태 중 하나에 작업이 존재할 수 있음을 의미한다. 이 단순화 된 모델을 먼저 고려할 것이다. 그러나 이것은 나중에 실행되지 않는 상태에 실제로 많은 수의 하위 상태가 포함되어 있음을 알기 때문에 지나치게 단순화되었음을 명심하시오.

작업이 실행 중 상태에 있으면 프로세서는 실제로 코드를 실행하고 있다. 작업이 실행되지 않은 상태에 있는 경우 작업은 휴면 상태이며 다음에 스케줄러가 실행 상태로 전환할 것을 결정할 때 해당 상태가 실행을 다시 시작할 준비가 되었다. 태스크가 실행을 다시 시작하면 실행 상태를 마지막으로 떠나기 전에 실행하려고 했던 명령에서 수행한다.

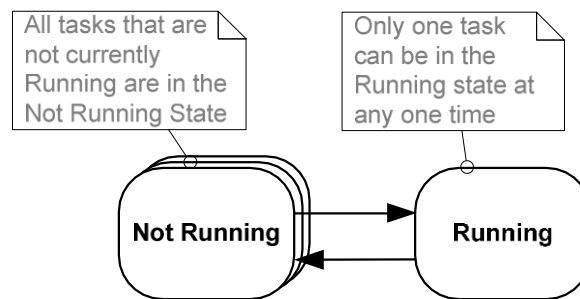


Figure 1 Top level task states and transitions.

실행되지 않은 상태에서 실행 중 상태로 전환 된 작업은 "전환 된" 또는 "스왑 된" 상태라고 한다. 반대로 실행 중 상태에서 실행 중 상태로 전환 된 작업은 "스위치 아웃" 또는 "스왑 아웃" 상태라고 한다. FreeRTOS 스케줄러는 태스크를 안팎으로 전환 할 수 있는 유일한 엔티티이다.

## 1.4 CREATING TASKS

### xTaskCreate() API Function

작업은 FreeRTOS xTaskCreate () API 함수를 사용하여 만든다. 이것은 아마도 모든 API 함수 중에서 가장 복잡하기 때문에 처음에는 불행한 일이지만, 작업은 멀티 태스킹 시스템의 가장 기본적인 구성 요소이므로 먼저 마스터해야 한다. 이 책과 함께 제공되는 모든 예제는 xTaskCreate () 함수를 사용하므로 많은 예제를 참조 할 수 있다. 부록 5 : 사용 된 데이터 유형과 명명 규칙에 대해 설명한다.

```
portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode, const signed
portCHAR * const pcName, unsigned portSHORT usStackDepth, void
*pvParameters,
```

```
unsigned portBASE_TYPE uxPriority, xTaskHandle
                                *pxCreatedTask );
```

**Listing 3 The xTaskCreate() API function prototype**

Table 1 xTaskCreate() parameters and return value

Parameter Name/Returned Value	Description
pvTaskCode	작업은 절대로 종료하지 않는 C 함수이며 일반적으로 무한 루프로 구현된다. pvTaskCode 매개 변수는 단순히 태스크를 구현하는 함수 (사실상 함수 이름)에 대한 포인터이다.
pcName	작업을 설명하는 이름이다. 이것은 FreeRTOS 에서 어떤 식 으로든 사용하지 않는다. 이것은 디버깅을 돕는 목적으로 만 사용된다. 사람이 읽을 수 있는 이름으로 작업을 식별하는 것은 핸들에서 동일한 작업을 시도하는 것보다 훨씬 간단하다 응용 프로그램 정의 상수 configMAX_TASK_NAME_LEN 은 작업 이름이 취할 수 있는 최대 길이 (NULL 종결 자 포함)를 정의한다. 이 최대 값보다 긴 문자열을 제공하면 단순히 문자열이 자동으로 잘린다.

Table 1 xTaskCreate() parameters and return value

Parameter Name/Returned Value	Description
-------------------------------	-------------



**usStackDepth**      각 작업에는 작업이 생성 될 때 커널이 작업에 할당하는 고유 한 상태가 있다.  
usStackDepth 값은 스택을 만드는 데 필요한 커널의 크기를 알려준다.

이 값은 스택이 보유 할 수 있는 단어 수를 지정하며 바이트 수는 지정하지 않는다.

예를 들어, 스택이 32 비트 폭이고 usStackDepth 가 100 으로 전달되면 스택 공간의 400 바이트가 할당된다 (100 \* 4 바이트).

스택 깊이에 스택 폭을 곱한 값은 size\_t 유형의 변수에 포함될 수 있는 최대 값을 초과하지 않아야 한다.

유휴작업에 사용되는 스택의 크기는 응용 프로그램 정의 상수

configMINIMAL\_STACK\_SIZE 에 의해 정의된다. 사용되는 마이크로 컨트롤러 아키텍처에 대한 FreeRTOS 데모 애플리케이션에서 이 상수에 할당 된 값은 모든 작업에 대해 최소한 권장된다. 태스크가 많은 스택 공간을 사용한다면 더 큰 값을 할당해야 한다.

작업에 필요한 스택 공간을 쉽게 결정할 수 있는 방법은 없다. 계산할 수도 있지만 대부분의 사용자는 합리적인 값이라고 생각하는 것을 할당 한 다음 FreeRTOS 에 서 제공하는 기능을 사용하여 할당 된 공간이 실제로 충분한 지, RAM 이 불필요하게 낭비되지 않는지 확인한다. 6 장에는 작업에서 사용 중인 스택 공간을 쿼리하는 방법에 대한 정보가 들어 있다.

**pvParameters**      태스크 함수는 void (void \*)에 대한 포인터 유형의 매개 변수를 승인한다.

pvParameters 에 할당 된 값은 태스크에 전달되는 값이다. 이 문서의 일부 예제는 매개 변수의 사용 방법을 보여준다.

작업을 실행할 우선 순위를 정의한다. 우선 순위는 가장 낮은 우선 순위 인

**uxPriority**      0에서

가장 높은 우선 순위 인 (configMAX\_PRIORITIES - 1)까지 할당 할 수 있다.

configMAX\_PRIORITIES 는 사용자가 정의한 상수이다. 사용할 수 있는 우선 순위 수에 대한 상한선은 없다 (사용되는 데이터 유형의 한계와 마이크로 컨트롤러에서 사용 가능한 RAM 제외). 그러나 RAM 낭비를 피하기 위해 실제로 필요한 최소 우선 순위수를 사용해야 한다 .

위의 uxPriority 값 (configMAX\_PRIORITIES - 1)을 전달하면 작업에 할당 된 실제 우선 순위가 자동으로 최대 합법적 인 값으로 제한된다.

Table 1 xTaskCreate() parameters and return value

Parameter Name/Returned Value	Description
pxCreatedTask	<p>pxCreatedTask 는 생성되는 작업에 대한 핸들을 전달하는 데 사용할 수 있다. 그런 다음이 핸들을 사용하여 예를 들어 태스크 우선 순위를 변경하거나 태스크를 삭제 하는 API 호출 내에서 태스크를 참조 할 수 있다.</p> <p>응용 프로그램에서 작업 핸들을 사용하지 않으면 pxCreatedTask 를 NULL 로 설정 할 수 있다.</p>
Returnedvalue	<p>가능한 두 가지 반환 값이 있다. :</p> <ol style="list-style-type: none"><li>1. pdTRUE</li></ol> <p>이것은 작업이 성공적으로 생성되었음을 나타낸다.</p> <ol style="list-style-type: none"><li>2. errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY</li></ol> <p>이것은 FreeRTOS 가 태스크 데이터 구조와 스택을 보유하기에 충분한 RAM 을 할당하기에 부족한 힙 메모리가 있었기 때문에 태스크를 작성할 수 없음을 나타낸다.</p> <p>5 장에서는 메모리 관리에 대한 자세한 정보를 제공한다.</p>

### Example 1. Creating Tasks

부록 1 : 예제 프로젝트를 빌드하는 데 필요한 도구에 대한 정보가 들어 있다.

이 예제는 두 개의 간단한 태스크를 작성한 후 실행중인 태스크를 시작하는 데 필요한단계를보여준다. 작업은주기적으로널 (null) 루프를사용하여주기 지연을 작성하는 문자열을 정기적으로 인쇄한다. 두 작업 모두 동일한 우선 순위로 만들어지며 인쇄 된 문자열과 동일하다.

- 각각의 구현에 대해서는 Listing 4 및 Listing 5 를 참조하시오..

```
void vTask1( void *pvParameters )
{
    const char *pcTaskName = "Task 1 is running\r\n";    volatile unsigned long ul;

    /* 대부분의작업에따라이작업은무한루프로구현된다. */ for( ;; )
    {
        /*이작업의이름을출력하시오. */
        vPrintString( pcTaskName );

        /* 일정기간지연. */
    }
```

```
for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
{
    /*이루프는매우조잡한지연구현이다. 여기에는할일이없다. 이후의예제들은이조잡한 루프를적절한지연 /
    슬립기능으로대체할것이다. * /
}
}
}
```

**Listing 4 Implementation of the first task used in Example 1**

```
void vTask2( void *pvParameters )
{
    const char *pcTaskName = "Task 2 is running\r\n"; volatile unsigned long ul;

    /* 대부분의작업에따라이작업은무한루프로구현된다. * /
    for( ;; )
    {
        /*이작업의이름을출력하시오. * /
        vPrintString( pcTaskName );

        /* 일정기간지연. * /
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /*이루프는매우조잡한지연구현이다. 여기에는할일이없다. 이후의예제들은이조잡한루프를적절한지연
            / 슬립기능으로대체할것이다. * /
        }
    }
}
```

**Listing 5 Implementation of the second task used in Example 1**

main () 함수는 단순히 스케줄러를 시작하기 전에 태스크를 생성한다. 구현을 위해 Listing 6 을 보자.

```
int main( void )
{
    /* 두작업중하나를만든다. 실제응용프로그램은 xTaskCreate () 호출의반환값을확인
    하여작업이성공적으로만들어졌는지확인해야한다. * /

    xTaskCreate(      vTask1, /* 태스크를구현하는함수의포인터. * /
                    "Task 1", /* 작업의텍스트이름이다. 이는디버깅을용이하게하기위한것이다.
                    * /
                    1000,      /* 스택깊이 - 대부분의소형마이크로컨트롤러는이보다훨씬적은
                               스택을사용한다. * /
                    NULL, /* 우리는 task 매개변수를사용하지않는다. * /
```

```
1,          / *이작업은우선순위 1 에서실행된다. * /
NULL ); / * 우리는태스크핸들을사용하지않을것이다. * /

/ * 똑같은방법으로동일한우선/* Create 순위로다른작업을만든다. * /
xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

/ * 작업이실행되기시작하도록스케줄러를시작하시오. * /
vTaskStartScheduler();

/ * 모든것이잘되면스케줄러가이제태스크를실행할때 main () 이어기에도달하지않는다.
main () 이어기에도달하면, 유틸작업을생성하는데사용할수있는힙메모리가부족하다.
5 장에서는메모리관리에대한자세한정보를제공한다. * /
for( ;; ); }
```

#### Listing 6 Starting the Example 1 tasks

Executing the example produces the output shown in Figure 2.

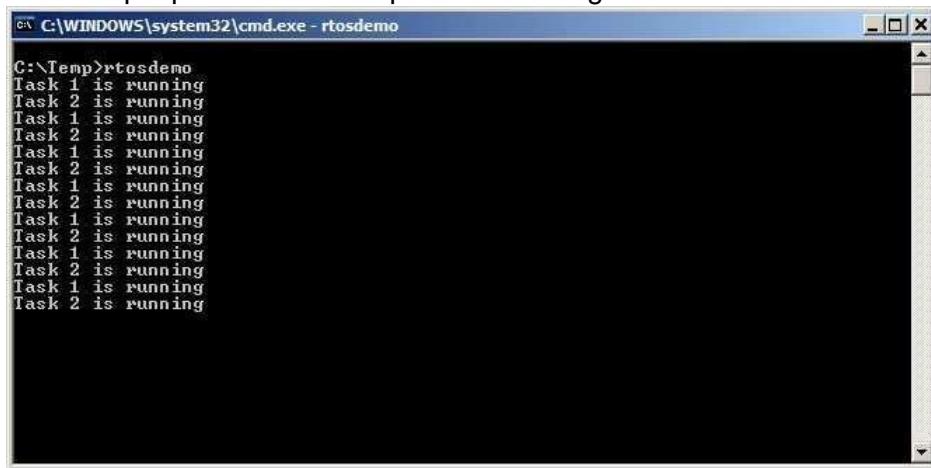
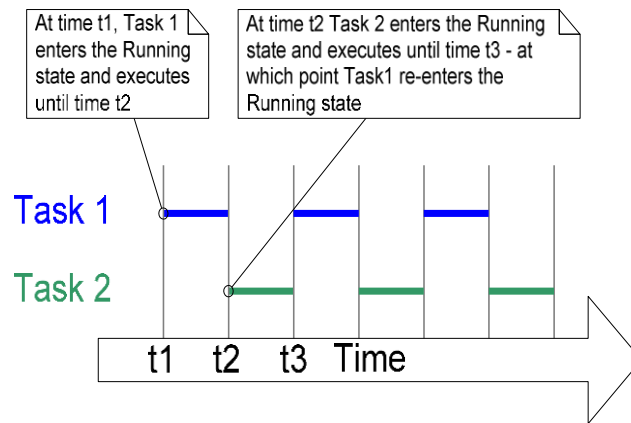


Figure 2 The output produced when Example 1 is executed

그림 2 는 동시에 실행되는 두 작업을 보여준다. 그러나 두 작업 모두 동일한 프로세서에서 실행되므로 실제로는 그렇지 않다. 현실적으로 두 작업 모두 실행 상태에 빠르게 들어가고 종료한다. 두 작업 모두 동일한 우선 순위로 실행되므로 단일 프로세서에서 시간을 공유하시오. 실제 실행 패턴은 그림 3 과 같다. 그림 3 하단의 화살표는 시간 t1 이후의 시간 경과를 보여준다. 색이 있는 선은 각 시점에서 실행중인 작업을 보여준다. 예를 들어 Task1 이 t1 과 t2 사이에서 실행 중이다.

하나의 작업 만 실행 상태에 존재할 수 있으므로 한 작업이 실행 상태 (작업이 전환 됨)로 들어가고 다른 작업이 실행되지 않음 상태 (작업이 전환 됨)가 된다.



**Figure 3 The actual execution pattern of the two Example 1 tasks**

예제 1 은 스케줄러를 시작하기 전에 main () 내에서 두 태스크를 작성했다. 다른 작업 내에서 작업을 만들 수도 있다. main ()에서 Task1 을 만든 다음 Task1 에서 Task2 를 만들 수 있다. 이 작업을 수행했으면 Task1 함수가 Listing 7 과 같이 변경 된다. Task2 는 스케줄러가 시작될 때까지 작성되지 않지만 예제로 생성 된 출력 은 동일하다.

```
void vTask1( void *pvParameters )
{
    const char *pcTaskName = "Task 1 is running\r\n";    volatile unsigned long ul;

    /*이작업코드가실행중이면스케줄러가이미시작되어있어야한다. 무한루프에들어가기전에다른작업을만 든다. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    for( ;; )
    {
        /*이작업의이름을출력하시오. */
        vPrintString( pcTaskName );

        /* 일정기간지연. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /*이루프는매우조잡한지연구현이다. 여기에는할일이없다. 이후의예제들은이조잡한루프를적절한지연
            / 슬립기능으로대체할것이다. */
        }
    }
}
```

**Listing 7 Creating a task from within another task – after the scheduler has started**

## Example 2. Using the Task Parameter

예제 1 에서 생성 된 두 가지 작업은 거의 동일했지만 그 차이점은 인쇄 한 텍스트 문자열뿐이었다. 이 복 제는 대신 단일 태스크 구현의 두 인스턴스를 작성하여 제거 할 수 있다. 그런 다음 task 매개 변수를 사 용하여 해당 인스턴스가 인쇄해야 하는 문자열을 각 작업에 전달할 수 있다.

Listing 8 에는 예제 2 에서 사용한 단일 태스크 함수 (vTaskFunction)의 코드가있다.이 단일 함수는 예제 1 에서 사용 된 두 태스크 함수 (vTask1 및 vTask2)를 대체한다.

태스크가 출력해야 하는 문자열을 얻으려면 태스크 매개 변수가 char \*로 어떻게 캐스팅되는지 주목하십시오.

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    volatile unsigned long
    ul;

    / * 인쇄할문자열은매개변수를통해전달된다. 이것을캐릭터포인터에캐스트하십시오. * / pcTaskName =
    ( char * ) pvParameters;

    / * 대부분의작업에따라이작업은무한루프로구현된다. * / for( ;; )
    {
        / *이작업의이름을출력하십시오. * /
        vPrintString( pcTaskName );

        / * 일정기간지연. * /
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            / *이루프는매우조잡한지연구현이다. 여기에는할일이없다. 나중에연습문제는이지연루프를적절한지연 /
            수면기능으로대체할것이다. * /
        }
    }
}
```

**Listing 8** The single task function used to create two tasks in Example 2

이제는 하나의 작업 구현 (vTaskFunction)이 있더라도 정의 된 작업의 둘 이상의 인스턴스를 만들 수 있다. 생성 된 각 인스턴스는 FreeRTOS 스케줄러의 제어 하에 독립적으로 실행된다.

xTaskCreate () 함수의 pvParameters 매개 변수는 텍스트 문자열을 전달하는 데 사용된다 (Listing 9 참조).

```
/ * 작업매개변수로전달될문자열을정의하십시오. 이것들은태스크가실행중일때유효하게유지
되도록스택이아니라 const 로정의된다. * /
```

```
static const char *pcTextForTask1 = "Task 1 is running\r\n";
static const char *pcTextForTask2 = "Task 2 is running\t\n";

int main( void )
{
    /* 두작업중하나를만든다. */
    xTaskCreate( on,          vTaskFunction, /* * 태스크를구현하는함수의포인터. */
                "Task 1",    /* * 작업의텍스트이름이다. 이것은 ~이다.
                               디버깅만촉진한다. */
                1000,         /* * 스택깊이 - 대부분의소형마이크로컨트롤러는
                               이보다훨씬적은스택을사용한다. */
                (void*)pcTextForTask1, /* * 인쇄할텍스트를작업에전달한다. task
                               매개변수를사용하십시오. */
                1,            /* *이작업은우선순위 1 에서실행된다. */
                NULL );      /* * 우리는작업핸들을사용하지않는다. */

    /* 똑같은방식으로다른작업을만든다. 이번에는 SAME 작업구현 (vTaskFunction)에서여러
    개의작업이만들어지고있음을유의하십시오. 매개변수에서전달된값만다르다. 동일한작업의두 인스턴스가생성된다. */
    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 1, NULL );

    /* 우리의작업이실행되기시작하도록스케줄러를시작하십시오. */
    vTaskStartScheduler();

    /* 모든것이잘되면스케줄러가이제태스크를실행할때 main ()이여기에도달하지않는다. main
    () 이여기에도달하면유틸리티작업을생성하는데사용할수있는힙메모리가부족할수있다.
    5 장에서는메모리관리에대한자세한정보를제공한다. */
    for(;;);
}
```

Listing 9 The main() function for Example 2.

예제 2의 결과는 그림 2의 예제 1과 동일하다.

## 1.5 TASK PRIORITIES

xTaskCreate () API 함수의 uxPriority 매개 변수는 생성되는 작업에 초기 우선 순위를 할당한다. 우선 순위는 vTaskPrioritySet () API 함수를 사용하여 스케줄러가 시작된 후에 변경 될 수 있다.

사 용 가 능 한 최 대 우 선 순 위 수 는 FreeRTOSConfig.h 내 의 응 용 프 로 그 램 정 의 configMAX\_PRIORITIES 컴파일 시간 구성 상수에 의해 설정된다. FreeRTOS 자체는 이 상수가 취할

수 있는 최대 값을 제한하지 않지만 configMAX\_PRIORITIES 값이 높을수록 커널이 소비하는 RAM 이 많아 지므로 필요한 최소값으로 설정하는 것이 좋다.

FreeRTOS 는 작업에 우선 순위를 할당하는 방법에 대한 제한을 두지 않는다. 어떤 수의 작업도 동일한 우선 순위를 공유 할 수 있으므로 최대한의 설계 유연성을 보장한다. 원하는 경우 모든 작업에 고유 한 우선 순위를 할당 할 수 있지만 (일부 일정 기능 알고리즘에서 요구하는 대로)이 제한은 어떤 식 으로든 적용되지 않는다.

낮은 숫자 우선 순위 값은 낮은 우선 순위 작업을 나타내며 우선 순위 0 이 가능한 가장 낮은 우선 순위 이다. 따라서 사용 가능한 우선 순위 범위는 0 ~ (configMAX\_PRIORITIES - 1)이다.

스케줄러는 항상 실행할 수 있는 가장 우선 순위가 높은 태스크가 실행 상태로 들어가기 위해 선택된 태스크인지 확인한다. 동일한 우선 순위의 태스크가 두 개 이상 실행될 수 있는 경우 스케줄러는 각 태 스크를 실행 상태로 전환한다. 이것은 두 테스트 작업이 동일한 우선 순위에서 만들어지고 둘 다 항상 실행될 수 있었던 지금까지의 예제에서 관찰 된 동작이다. 이러한 각 태스크는 "타임 슬라이스"에 대해 실행되고, 타임슬라이스의시작부분에서실행상태로들어가고타임슬라이스의끝에서실행상태를 종료한다. 그림 3 에서 t1 과 t2 사이의 시간은 단일 시간 슬라이스와 같다.

스케줄러 자체를 실행할 다음 태스크를 선택할 수 있으려면 각 시간 조각의 끝에서 실행해야 한다. 틱 인터럽트라고하는 주기적인 인터럽트가이 용도로 사용된다. 시간 조각의 길이는 FreeRTOSConfig.h 의 configTICK\_RATE\_HZ 컴파일 시간 구성 상수에 의해 구성된 틱 인터럽트 빈도에 의해 효과적으로 설정된다. 예를 들어, configTICK\_RATE\_HZ 가 100 (Hz)로 설정된 경우 시간 조각은 10ms 가 된다. 그림 3 은 실행 순서에서 스케줄러 자체의 실행을 보여주기 위해 확장 될 수 있다. 이것은 그림 4 에 나와 있다.

FreeRTOS API 호출은 항상 틱 인터럽트 (일반적으로 '틱'이라고 함)에서 시간을 지정한다. portTICK\_RATE\_MS 상수는 시간 지연이 틱 인터럽트 수에서 밀리 초로 변환 될 수 있도록 제공된다. 사용 가능한 해상도는 틱 주파수에 따라 다르다.

'tick count'값은 스케줄러가 시작된 이후 발생한 틱 인터럽트의 총 수이다. 틱 수가 오버 플로우되지 않았다고 가정한다. 사용자 응용 프로그램은 커널이 내부적으로 시간 일관성을 관리 할 때 지연 기간을 지정할 때 오버플로를 고려할 필요가 없다.



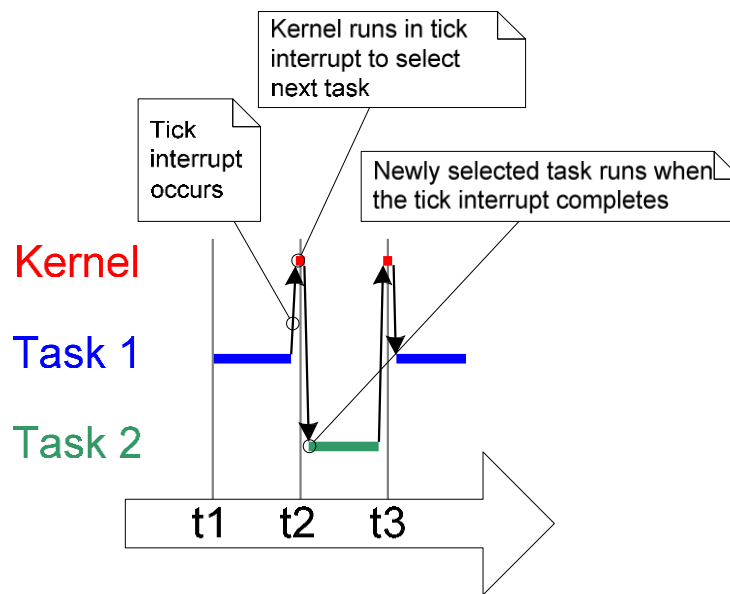


Figure 4 The execution sequence expanded to show the tick interrupt executing.

그림 4 에서 빨간색 선은 커널 자체가 실행 중일 때를 나타낸다. 검은 색 화살표는 작업에 서 인터럽트로, 그런 다음 인터럽트에서 다른 작업으로 실행 순서를 보여준다.

### Example 3. Experimenting with priorities

스케줄러는 항상 실행할 수 있는 가장 우선 순위가 높은 태스크가 실행 상태로 들어가기 위해 선택된 태스크인지 확인한다. 지금까지는 두 개의 태스크가 동일한 우선 순위로 생성되었으므로 둘 다 실행 상태로 들어가고 나왔다. 이 예제에서는 예제 2에서 생성된 두 태스크 중 하나의 우선 순위를 변경할 때 어떤 일이 발생하는지 살펴본다. 이번에는 첫 번째 태스크가 우선 순위 1에서 생성되고 두 번째 태스크가 우선 순위 2에서 생성된다. 태스크를 생성하는 코드는 Listing 10. 두 태스크를 구현하는 단일 함수는 변경되지 않았지만 지연을 생성하기 위해 널 루프를 사용하여 문자열을 주기적으로 인쇄한다.

```
/* 작업 매개변수로 전달될 문자열을 정의하십시오. 이것들은 태스크가 실행 중일 때 유효하게 유지되도록 스택이 아니라
const 로 정의됩니다. */
static const char *pcTextForTask1 = "Task 1 is running\r\n";
static const char *pcTextForTask2 = "Task 2 is running\t\n";

int main( void )
{
    /* 우선 순위 1 의 첫 번째 작업을 만든다. 우선 순위는 두 번째 매개변수에서 마지막 매개변수이다. */
    xTaskCreate( vTaskFunction, "Task 1", 1000, (void*)pcTextForTask1, 1, NULL );

    /* 우선 순위 2 에서 두 번째 작업을 만든다. */
}
```

```
xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 2, NULL );

/ * 작업이실행되기시작하도록스케줄러를시작하시오. * /
vTaskStartScheduler();

return
0; }
```

#### Listing 10 Creating two tasks at different priorities

예제 3에 의해 생성된 출력은 그림 5에 나와 있다.

스케줄러는 항상 실행할 수 있는 가장 우선 순위가 높은 작업을 선택한다. 작업 2는 작업 1보다 우선 순위가 높으며 항상 실행할 수 있다. 따라서 작업 2는 실행 상태로 전환하는 유일한 작업입니다. 작업 1은 실행 상태가 되지 않으므로 문자열을 인쇄하지 않는다. 작업 1은 작업 2에 의해 처리 시간이 '굵어 죽었다'고 한다.

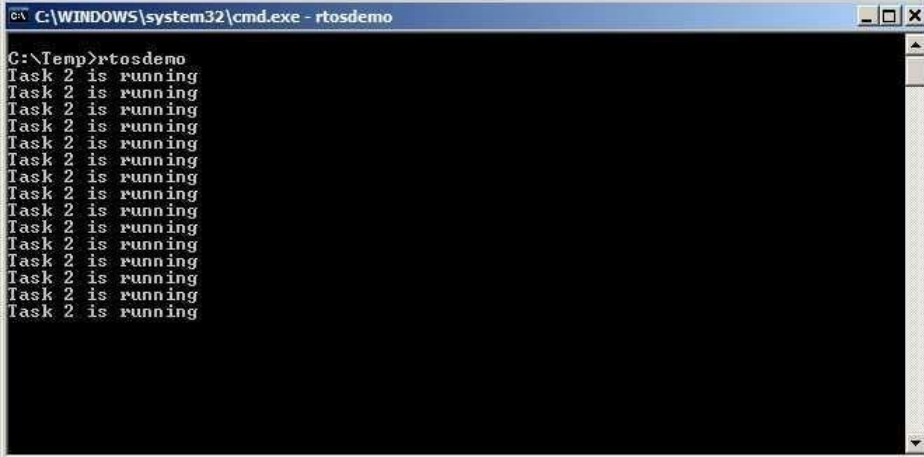


Figure 5 Running both test tasks at different priorities

작업 2는 아무 것도 기다릴 필요가 없기 때문에 항상 실행될 수 있습니다. 이는 널 루프를 돌거나 터미널에 인쇄하는 것이다. 그림 6은 예제 3의 실행 순서를 보여준다.

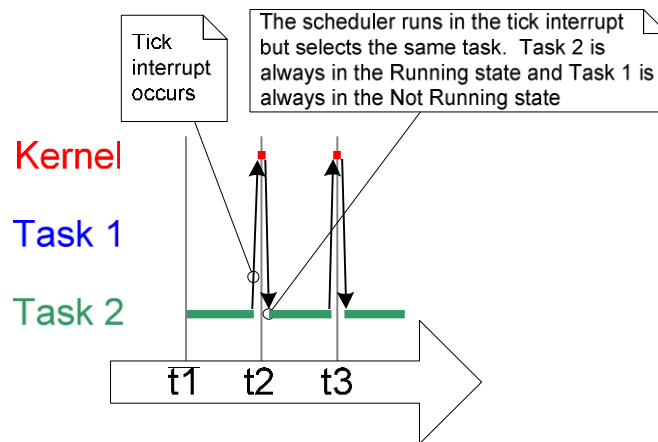


그림 6 한태스크가 다른태스크보다높은우선순위를가는경우의실행패턴

## 6. EXPANDING THE 'NOT RUNNING' STATE

지금까지 제시된 예제에서 제시된 각 예제는 항상 수행하기를 원했고 아무 것도 기다릴 필요가 전혀 없는 처리를 처리했다. 항상 실행 상태가 될 수 있는 것을 기다릴 필요가 없었다. 이러한 유형의 '연속 처리' 작업은 매우 낮은 우선 순위에서만 생성 될 수 있기 때문에 유용성이 제한적이다. 다른 우선 순위에서 실행되는 경우 우선 순위가 낮은 작업은 전혀 실행되지 않는다.

우리의 임무를 실제로 유용하게 만들기 위해서는 우리가해야 할 일이 이벤트 중심이 될 수 있는 방법이 필요하다. 이벤트 구동 태스크는 이벤트를 발생시키는 이벤트 발생 후 수행 할 작업 (처리) 만 가지며 이 이벤트가 발생하기 전에 실행 중 상태에 들어갈 수 없다. 스케줄러는 항상 실행할 수 있는 가장 높은 우선 순위의 태스크를 선택한다. 우선 순위가 높은 작업을 실행할 수 없다는 것은 스케줄러가 선택할 수 없음을 의미하며 대신 실행할 수 있는 우선 순위가 낮은 작업을 선택해야 한다. 따라서 이벤트 중심 작업을 사용하면 우선 순위가 가장 높은 작업이 처리 시간의 우선 순위가 낮은 모든 작업을 굶주리게 하지 않고 도 여러 가지 우선 순위에서 작업을 만들 수 있다.

### The Blocked State

이벤트를 기다리는 작업은 실행되지 않음 상태의 하위 상태 인 '차단됨' 상태라고 한다.

작업은 다음 두 가지 유형의 이벤트를 기다리려면 차단된 상태로 들어갈 수 있다.

1. 시간 (시간 관련) 이벤트 - 지연 기간이 만료되거나 절대 시간에 도달하는 이벤트이다. 예를 들어 작업이 10 밀리초가 경과하기를 기다리는 Blocked 상태로 들어갈 수 있다.
2. 동기화 이벤트 - 이벤트가 다른 작업 또는 인터럽트에서 비롯된 것이다. 예를 들어, 작업이 대기열에 데이터가 도착하기를 기다리는 Blocked (차단됨) 상태로 들어갈 수 있다. 동기화 이벤트는 광범위한 이벤트 유형을 포함한다.

FreeRTOS 큐, 바이너리 세마포어, 카운팅 세마포어, 재귀 세마포어 및 뮤텍스를 모두 사용하여 동기화 이벤트를 만들 수 있다. 2 장과 3 장에서 이에 대해 자세히 다룬다.

태스크가 타임 아웃으로 동기화 이벤트를 차단하여 두 이벤트 유형을 동시에 효과적으로 차단할 수 있다. 예를 들어, 작업은 대기열에 데이터가 도착할 때까지 최대 10 밀리 초 동안 대기하도록 선택할 수 있다.

다. 데이터가 10 밀리 초 이내에 도착하거나 데이터가 도착하지 않고 10 밀리 초가 지나면 작업은 차단된 상태를 유지한다.

## The Suspended State

'Suspended' 는 또한 Not Running 의 하위 상태이다. Suspended 상태의 작업은 스케줄러에서 사용할 수 없다. Suspended 상태로 가는 유일한 방법은 vTaskSuspend () API 함수를 호출하고 vTaskResume () 또는 xTaskResumeFromISR () API 함수를 호출하는 유일한 방법이다. 대부분의 응용 프로그램은 Suspended 상태를 사용하지 않는다.

## The Ready State

실행 중이 아니지만 차단 또는 일시 중단되지 않은 작업은 준비 상태라고 한다. 그들은 실행할 수 있고 따라서 실행 준비가 되어 있지만 현재 실행 상태가 아니다.

## Completing the State Transition Diagram

그림 7 은 이전 섹션의 단순화 된 상태 다이어그램을 확장하여이 섹션에서 설명하는 모든 실행되지 않음 하위상태를포함한다. 지금까지예제에서생성된작업은 Blocked 또는 Suspended 상태를사용하지않 았으므로 Ready 상태와 Running 상태 사이에서만 전환되었다 (그림 7 의 굵은 선으로 강조 표시).

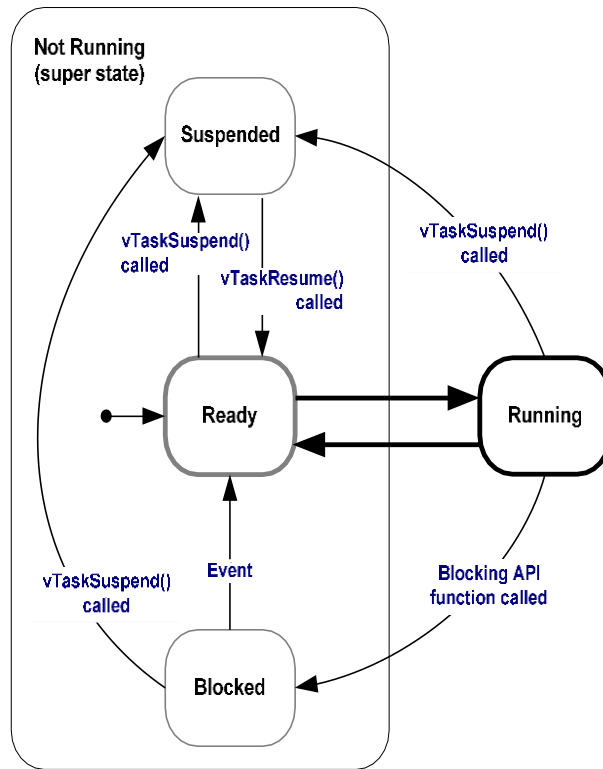


Figure 7 Full task state machine

#### Example 4. Using the Blocked state to create a delay

지금까지 제시된 예제에서 생성된 모든 작업은 '주기적'이었다. 즉, 한동안 지연되고, 문자열을 인쇄한 다음, 다시 한 번 연기하는 등의 작업을 수행했다. 지연은 널 루프를 사용하여 매우 정교하게 생성되었다. 태스크는 고정된 값에 도달할 때까지 점증하는 루프 카운터를 효과적으로 폴링한다. 예 3은 이 방법의 단점을 분명히 보여주었다. null

루프를 실행하는 동안 작업은 준비 상태로 남아 있었고 처리 시간의 다른 작업은 'starving.' '굶주렸다.' 어떤 형태의 폴링에도 다른 몇 가지 단점이 있으며, 그 중 비효율적인 것은 아니다. 작업을 폴링하는 작업은 실제로 할 일이 없지만 최대 처리 시간은 여전히 사용한다.

따라서 프로세서 사이클이 낭비된다. 예제 4는 폴링 널 루프를 `vTaskDelay()` API 함수에 대한 호출로 대체하여 이 동작을 수정한다 (Listing 11의 프로토타입은 Listing 11을 참조하십시오). 새 태스크 정의는 Listing 12에 표시되어 있다.

vTaskDelay ()는 일정 수의 틱 인터럽트에 대해 호출하는 태스크를 Blocked 상태로 둡니다. Blocked 상태에서는 작업이 전혀 처리 시간을 사용하지 않으므로 처리 시간은 실제로 수행해야 할 작업이있을 때 만 소모된다.

```
void vTaskDelay( portTickType xTicksToDelay );
```

#### Listing 11 The vTaskDelay() API function prototype

Table 2 vTaskDelay() parameters

Parameter Name	Description
xTicksToDelay	호출중인 작업이 준비 됨 상태로 다시 전환되기 전에 차단 된 상태로 유지되어야하는 틱 인터럽트의 수이다.  예를 들어 틱 수가 10,000 인 동안 vTaskDelay (100)라는 태스크가 즉시 차단 상태 가되어 틱 수가 10,100 에 도달 할 때까지 그 상태를 유지한다.  상수 portTICK_RATE_MS 를 사용하여 밀리 초를 틱으로 변환 할 수 있다.

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;

    /* 인쇄할문자열은매개변수를통해전달된다. 이것을캐릭터포인트어캐스트하시오. */ pcTaskName =
    ( char * ) pvParameters;

    /* 대부분의작업에따라이작업은무한루프로구현된다. */
    for( ;; ) {

        /*이작업의이름을출력하시오. */
        vPrintString( pcTaskName );
        vPrintString( pcTaskName );

        /* 일정기간지연. 이번에는 vTaskDelay () 호출이사용되어지연기간이만료될때까지작업을차단됨상태 로만든다.
        지연기간은 '틱'으로지정되지만상수 portTICK_RATE_MS 를사용하여이값을밀리초단위로보다
        사용자에게친숙한값으로변환할수있다. 이경우 250 밀리초의기간이지정된다. */ vTaskDelay( 250 /
        portTICK_RATE_MS );
    }
}
```

#### Listing 12 The source code for the example task after the null loop delay has been replaced by a call to vTaskDelay()





두 가지 작업의 구현 만 변경되었으며 기능은 변경되지 않았다. 그림 9와 그림 4를 비교하면 이 기능이 훨씬 더 효율적인 방식으로 실현되고 있음을 분명히 알 수 있다.

그림 4는 태스크가 널 루프를 사용하여 지연을 생성 할 때 실행 패턴을 보여 주므로 결과적으로 항상 많은 프로세서 시간을 실행하고 사용할 수 있었다. 그림 9는 작업이 지연 기간 전체에 대해 차단된 상태로 들어가기 때문에 실제로 수행해야하는 작업이 있는 경우에만 프로세서 시간을 활용한다 (이 경우 메시지는 단순히 인쇄된다).

그림 9 시나리오에서 작업이 차단된 상태를 벗어날 때마다 차단 된 상태로 다시 들어가기 전에 일정 기간 동안 만 실행된다. 대부분의 경우 실행할 수 있는 응용 프로그램 태스크가 없으므로 (준비 상태에서는 응용 프로그램 태스크가 없음) 실행중인 상태로 들어가기 위해 선택할 수 있는 응용 프로그램 태스크가 없다. 이 경우 유휴 작업이 실행된다. 유휴 작업이 처리하는 시간은 시스템에서 예비 처리 용량을 측정 한 것이다.

그림 10의 굵은 선은 예제 4의 작업에 의해 수행 된 전환을 보여 주며 각 전환은 준비 된 상태로 되돌아 가기 전에 차단된 상태로 전된다.

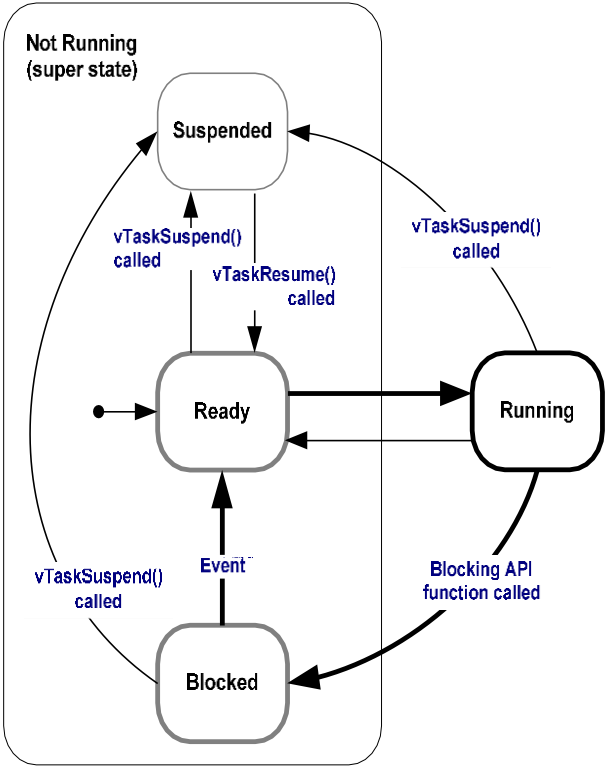


Figure 10 Bold lines indicate the state transitions performed by the tasks in Example 4

## vTaskDelayUntil() API function

vTaskDelayUntil ()은 vTaskDelay ()와 유사하다. 방금 설명한 바와 같이 vTaskDelay () 매개 변수는 vTaskDelay ()를 호출하는 작업과 차단 된 상태에서 다시 한 번 같은 작업으로 전환 할 때 발생해야 하는 눈금 중단 수를 지정한다. 작업이 차단 된 상태로 유지되는 시간은 vTaskDelay () 매개 변수에 의해 지정 되지만 작업이 차단 된 상태를 벗어나는 실제 시간은 vTaskDelay ()가 호출 된 시간을 기준으로한다. vTaskDelayUntil ()의 매개 변수 대신 호출하는 작업을 차단된 상태에서 준비 된 상태로 이동해야하는 정 확한 눈금 수 값을 지정한다. vTaskDelayUntil ()은 호출 된 태스크가 차단 해제 된 시간이 절대 시간이 아닌 절대 값일 때 고정 된 실행 기간 (고정 주파수로 태스크를 주기적으로 실행하려는 경우)이 필요할 때 사용해야하는 API 함수이다. 함수가 호출되었다 (vTaskDelay ()의 경우처럼).

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

### Listing 13 vTaskDelayUntil() API function prototype

Table 3 vTaskDelayUntil() parameters

Parameter Name	Description
pxPreviousWakeTime	<p>이 매개 변수의 이름은 vTaskDelayUntil ()이 주기적으로 고정 된 빈도로 실행 되는 작업을 구현하는 데 사용되는 것으로 가정한다. 이 경우 pxPreviousWakeTime 은 작업이 마지막으로 차단된 상태 ( '깨어나 기'상태)를 유지 한 시간을 유지한다. 이 시간은 작업이 다음에 차단된 상태를 벗어나는 시간을 계산하기위한 참조 점으로 사용된다.</p> <p>pxPreviousWakeTime 이 가리키는 변수는 vTaskDelayUntil () 함수 내에서 자동으로 업데이트되며 일반적으로 다음과 같이 수정되지 않는다. 변수가 처음 초기화 될 때가 아닌 응용 프로그램 코드. Listing 14 는 초기화가 수행되는 방법을 보여준다.</p>
xTimeIncrement	<p>이 매개 변수는 vTaskDelayUntil ()이 xTimeIncrement 값에 의해 설정된 빈도 로 고정 주파수로 주기적으로 실행되는 작업을 구현하는 데 사용된다는 가정 하에명명된다.</p> <p>xTimeIncrement 는 '틱'으로 지정된다. 상수 portTICK_RATE_MS 를 사용하여 밀리 초를 틱으로 변환 할 수 있다.</p>

#### Example 5. Converting the example tasks to use vTaskDelayUntil()

예제 4 에서 생성 된 두 가지 작업은 주기적인 작업이지만 vTaskDelay ()를 사용한다고 해서 vTaskDelay ()를 호출 할 때와 비교하여 작업이 Blocked 상태를 벗어나는 시간으로 고정되어 있다고 보장 할 수는 없 다. vTaskDelay () 대신 vTaskDelayUntil ()을 사용하도록 태스크를 변환하면이 잠재적인 문제점을 해결 할 수 있다.

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;    portTickType xLastWakeTime;

    /* 인쇄할 문자열은 매개 변수를 통해 전달된다. 이것을 캐릭터 포인터에 캐스트 하시오. */
    pcTaskName = ( char * ) pvParameters;

    /* xLastWakeTime 변수는 현재 틱 수로 초기화 되어야 한다. 이것은 변수가 명시적으로 기록되는 유일한 시간이다. 이후
    xLastWakeTime 은 vTaskDelayUntil () 내에서 자동으로 내부적으로 업데이트된다. */
    xLastWakeTime = xTaskGetTickCount();

    /* 대부분의 작업에 따라 작업은 무한 루프로 구현된다. */ for( ;; )
    {
        /* 이 작업의 이름을 출력 하시오. */
        vPrintString( pcTaskName );

        /* 이 작업은 정확히 250 밀리초마다 실행되어야 한다. vTaskDelay () 함수에 따라 시간은 틱 단위로 측정되고
        portTICK_RATE_MS 상수는 밀리초를 틱으로 변환하는데 사용된다.
        xLastWakeTime 은 vTaskDelayUntil () 내에서 자동으로 업데이트되므로 작업에의 해명 시적으로 업데이트되지 않는다.
        */
        vTaskDelayUntil( &xLastWakeTime, ( 250 / portTICK_RATE_MS ) );
    }
}
```

#### Listing 14 The implementation of the example task using vTaskDelayUntil()

실시 예 4 에 대한 도 8 에 도시 된 것과 동일한 경우, 실시 예 5 에 의해 산출 된 출력. Example 6.

#### 차단 및 비 차단 작업 결합

이전 예제에서는 폴링 및 차단 작업의 동작을 개별적으로 검사했다. 이 예에서는 다음과 같이 두 스키마가 결합 될 때 실행 순서를 보여줌으로써 명시된 예상 시스템 동작을 다시 시행한다.

우선 순위 1 에서 두 개의 작업이 생성된다. 이는 문자열을 연속적으로 출력하는 것 외에는 아무 것도 하지 않는다.

이러한 작업은 Blocked 상태가 될 수 있는 API 함수 호출을 하지 않으므로 항상 Ready 또는 Running 상태가 된다. 이 성질의 일은 언제나 할 일이있는 것처럼 '지속적인 처리'일컬어지며,이 경우 다소 사소한 일이긴하지만. 연속 처리 태스크의 소스는 Listing 15 와 같다.

그런 다음 세 번째 작업이 우선 순위 2 에서 만들어 지므로 다른 두 작업의 우선 순위보다 높다. 세 번째 작업은 문자열 만 출력하지만 이번에는 주기적으로 vTaskDelayUntil () API 호출을 사용하여 각 인쇄 반 복 사이에 Blocked 상태로 만든다.

주기적 태스크의 소스는 Listing 16 과 같다.

```
void vContinuousProcessingTask( void *pvParameters )
{
    char *pcTaskName;

    / * 인쇄할문자열은매개변수를통해전달된다. 이것을캐릭터포인터에캐스트하시오. * / pcTaskName =
    ( char * ) pvParameters;

    / * 대부분의작업에따라이작업은무한루프로구현된다. * /
    for( ;; ) {
        / *이작업의이름을출력하시오. 이작업은차단하거나지연하지않고반복적으로이작업을수행한 다. * /
        vPrintString( pcTaskName );
    }
}
```

**Listing 15** The continuous processing task used in Example 6.

```
void vPeriodicTask( void *pvParameters )
{
    portTickType xLastWakeTime;

    / * xLastWakeTime 변수는현재틱수로초기화되어야한다. 이것은변수가명시적으로기록되는유일한시간이
    다. 이후 xLastWakeTime 은 vTaskDelayUntil () API 함수에의해자동으로관리된다. * /
    xLastWakeTime = xTaskGetTickCount();

    / * 대부분의작업에따라이작업은무한루프로구현된다. * / for( ;; )
    {
        / *이작업의이름을출력하시오. * /
        vPrintString( "Periodic task is running\r\n" );

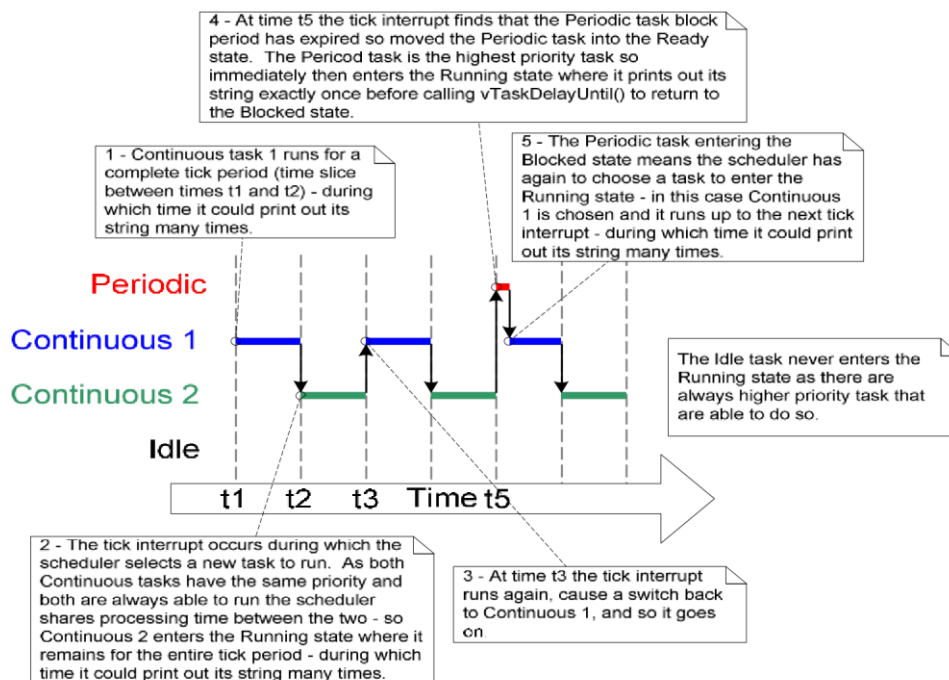
        / * 작업은정확히 10 milliseconds 마다실행되어야한다. * /
        vTaskDelayUntil( &xLastWakeTime, ( 10 / portTICK_RATE_MS ) );
    }
}
```

**Listing 16 The periodic task used in Example 6.**

그림 11 은 그림 12 에 표시된 실행 순서에 따라 관찰 된 동작에 대한 설명과 함께 예 6 에서 생성 된 출력 을 보여준다.

```
DOSBox 0.72, Cpu Cycles: 3000, Frameskip 0, Program: RTOSDEMO
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Periodic task is running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Continuous task 1 running
Periodic task is running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Continuous task 2 running
Periodic task is running
Continuous task 1 running
```

**Figure 11 The output produced when Example 6 is executed<sup>1</sup>.**



**Figure 12 The execution pattern of Example 6**

<sup>1</sup> 이 출력은 DOSBox 시뮬레이터를 사용하여 실행을 한 화면 캡처에서 전체 동작을 관찰할 수 있는 수준까지 그려게 만들었다.

## 7. THE IDLE TASK AND THE IDLE TASK HOOK

예제 4 에서 생성된 작업은 대부분의 시간을 차단된 상태로 유지한다. 이 상태에서는 실행할 수 없으며 스케줄러에서 선택할 수 없다.

프로세서는 항상 실행할 대상이 필요하다. 실행 상태가 될 수 있는 작업이 적어도 하나 이상 있어야 한다.

이 경우 `vTaskStartScheduler ()`가 호출되면 유휴 작업이 스케줄러에 의해 자동으로 생성된다. 유휴 작업은 루프에 앉아있는 것만 큼 쉽지 않으므로 원래 예제의 작업과 마찬가지로 항상 실행할 수 있다.

유휴 작업은 우선 순위가 가장 낮은 우선 순위 (우선 순위 0)를 가지므로 높은 우선 순위의 응용 프로그램 작업이 실행 중 상태가 되는 것을 결코 방지 할 수 없다. 원할 경우 응용 프로그램 디자이너가 작업을 생성하여 공유하기 때문에 유휴 작업 우선 순위를 막을 수는 없다.

최하위 우선 순위로 실행하면 우선 순위가 높은 작업이 준비 상태가 되면 즉시 유휴 작업이 실행 상태에서 벗어나게 된다. 이것은 그림 9의 시간  $t_n$ 에서 볼 수 있는데, 유휴 작업이 즉시 교체되어 작업 2가 차단된 상태에서 즉시 실행될 수 있다. 작업 2는 유휴 작업을 선택했다고 한다. 선택은 사전 선택된 작업에 대한 지식 없이 자동으로 수행된다.

### Idle Task Hook Functions

유휴 작업 루프의 반복마다 한 번 유휴 작업에 의해 자동으로 호출되는 유휴 후크 (또는 콜백) 기능을 사용하여 응용 프로그램 특정 기능을 유휴 작업에 직접 추가 할 수 있다.

유휴 작업 후크의 일반적인 용도는 다음과 같다.

- 낮은 우선 순위, 백그라운드 또는 연속 처리를 실행한다.
- 예비 처리 용량 측정 (유휴 작업은 다른 모든 작업에 수행 할 작업이 없을 때만 실행되므로 유휴 작업에 할당된 처리 시간을 측정하면 처리 시간이 여유분을 명확히 알 수 있다).
- 프로세서를 저전력 모드로 설정 - 수행 할 응용 프로그램 처리가 없을 때마다 전원을 절약하는 자동 방법 제공.

### Limitations on the Implementation of Idle Task Hook Functions

유휴 작업 후크 기능은 다음 규칙을 따라야 한다.

1. 그들은 결코 차단하거나 정지 시키면 안 된다. 유틸 작업은 다른 작업이 그렇게 할 수 없을 때만 실행 된다 (응용 프로그램 작업이 유틸 우선 순위를 공유하지 않는 한). 어떤 방식으로든 유틸 작업을 차단하면 실행 가능한 상태로 전환 할 수 없는 작업이 발생할 수 있다!

2 응용 프로그램에서 vTaskDelete () API 함수를 사용하는 경우 유틸 작업 후크는 적절한 시간 내에 항상 호출자에게 반환되어야 한다. 유틸 작업은 작업이 삭제 된 후 커널 리소스를 정리해야 하기 때 문이다. 유틸 작업이 유틸 혹 기능에 영구적으로 남아 있으면 이 정리 작업은 발생할 수 없다.

유틸 태스크 혹 함수는 Listing 17 에 표시된 이름과 프로토 타입을 가져야 한다.

```
void vApplicationIdleHook( void );
```

**Listing 17 The idle task hook function name and prototype.**

### **Example 7. Defining an Idle Task Hook Function**

예제 4 에서 vTaskDelay () API 호출을 사용하면 두 응용 프로그램 작업이 모두 차단된 상태이므로 Idle 작업이 실행될 때 많은 유틸 시간이 발생했다. 이 예제는 Idle hook 함수를 추가하여 이 유틸 시간을 사용한다 (Listing 18 의 소스 참조)..

```
/* 혹함수에의해증가될변수를선언하시오. */
unsigned long ulIdleCycleCount = 0UL;

/* Idle hook 함수는 반드시 vApplicationIdleHook () 이라고 불리며, 매개변수를 취하지 않고 void 를 반환해야 한다. */
void vApplicationIdleHook( void )
{
/* 이 후크 함수는 카운터를 증가시키는 것 외에는 아무것도 하지 않는다. */
ulIdleCycleCount++; }

```

**Listing 18 A very simple Idle hook function**

유틸 후크 기능이 호출되려면 configUSE\_IDLE\_HOOK 가 FreeRTOSConfig.h 에서 1 로 설정되어야 한다.

생성 된 태스크를 구현하는 함수는 ulIdleCycleCount 값을 출력하기 위해 약간 수정된다 (Listing 19 참조).

```
void vTaskFunction( void *pvParameters )
```

FreeRTOS

Designed For Microcontrollers;

© 2009 Richard Barry. Distribution or publication in any form is strictly prohibited.

```
{
char *pcTaskName;

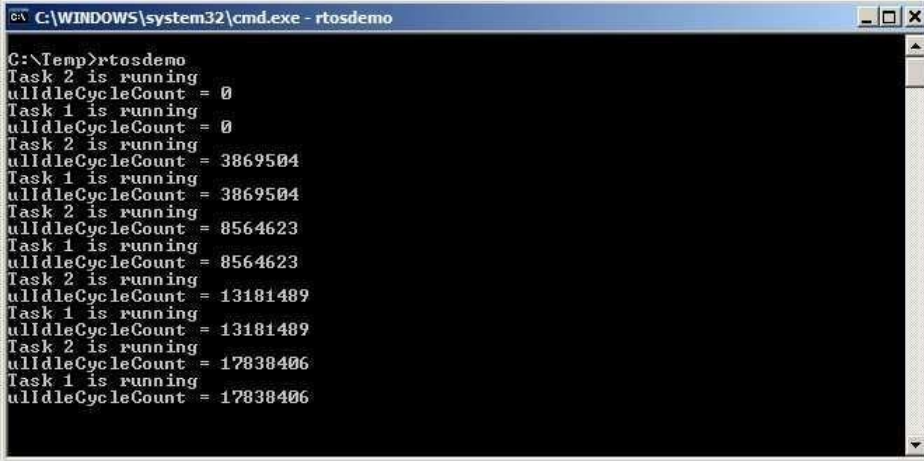
/ * 인쇄할문자열은매개변수를통해전달된다. 이것을캐릭터포인터에캐스트하시오. * / pcTaskName =
( char * ) pvParameters;

/ * 대부분의작업에따라이작업은무한루프로구현된다. * /
for( ;; )
{
/ *이작업의이름과 ulIdleCycleCount 가증가된횟수를출력한다. * /
vPrintStringAndNumber( pcTaskName, ulIdleCycleCount );

/ * 250 밀리초동안지연. * /
vTaskDelay( 250 / portTICK_RATE_MS );
}
}
```

Listing 19 The source code for the example task now prints out the ulIdleCycleCount value

예제 7 에 의해 생성 된 결과는 그림 13 에 나와 있으며 (내 컴퓨터에서) 유틸리티 태스크 혹은 기능은 애플리케이션 작업의 반복마다 약 450 만 번 (매우) 호출된다. nction 은 응용 프로그램 작업을 반복 할 때마다 약 450 만 번 (매우) 호출된다.



```
C:\WINDOWS\system32\cmd.exe - rtsdemo
C:\Temp>rtsdemo
Task 2 is running
ulIdleCycleCount = 0
Task 1 is running
ulIdleCycleCount = 0
Task 2 is running
ulIdleCycleCount = 3869504
Task 1 is running
ulIdleCycleCount = 3869504
Task 2 is running
ulIdleCycleCount = 8564623
Task 1 is running
ulIdleCycleCount = 8564623
Task 2 is running
ulIdleCycleCount = 13181489
Task 1 is running
ulIdleCycleCount = 13181489
Task 2 is running
ulIdleCycleCount = 17838406
Task 1 is running
ulIdleCycleCount = 17838406
```

Figure 13 The output produced when Example 7 is executed

## 1.8 CHANGING THE PRIORITY OF A TASK

### vTaskPrioritySet() API function

vTaskPrioritySet () API 함수는 스케줄러가 시작된 후 모든 태스크의 우선 순위를 변경하는 데 사용될 수 있다.



```
void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority );
```

**Listing 20 The vTaskPrioritySet() API function prototype**

Table 4 vTaskPrioritySet() parameters

Parameter Name	Description
pxTask	우선 순위가 수정되는 작업 (대상 작업)의 핸들 - 작업 처리에 대한 정보는 xTaskCreate () API 함수의 pxCreatedTask 매개 변수를 참조하십시오.  태스크는 유효한 태스크 핸들 대신에 NULL 을 전달하여 자신의 우선 순위를 변경할 수 있다.
uxNewPriority	주 작업이 설정 될 우선 순위이다. 이것은 가능한 최대 우선 순위 (configMAX_PRIORITIES - 1)로 자동으로 제한된다. 여기서 configMAX_PRIORITIES 는 FreeRTOSConfig.h 헤더파일에 설정된 컴파일 시간 옵션이다.

**uxTaskPriorityGet() API function**

uxTaskPriorityGet () API 함수를 사용하여 작업의 우선 순위를 쿼리 할 수 있다..

```
unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );
```

**Listing 21 The uxTaskPriorityGet() API function prototype**

Table 5 uxTaskPriorityGet() parameters and return value

Parameter Name/ReturnValue	Description
pxTask	우선 순위가 질의되는 태스크 (대상 태스크)의 핸들 - 태스크 핸들을 얻는 방법에 대한 정보는 xTaskCreate () API 함수의 pxCreatedTask 매개 변수를 참조하십시오.

태스크는 유효한 태스크 핸들 대신에 NULL 을 전달하여 자신의 우선 순위를 질의 할 수 있다.

Returnedvalue      쿼리중인 작업에 현재 할당 된 우선 순위이다.

### Example 8. Changing task priorities

스케줄러는 항상 가장 높은 준비 상태 태스크를 실행 상태로 들어가기 위한 태스크로 선택한다. 예제 8에서는 vTaskPrioritySet () API 함수를 사용하여 두 태스크의 우선 순위를 서로 다르게 변경하여 이를 보여준다.

두가지우선순위에서두가지작업이생성된다. 어느작업도차단상태가될수있는 API 함수호출을 하지 않으므로 둘 다 항상 준비 상태 또는 실행 상태에 있다. 따라서 상대 우선 순위가 가장 높은 작업은 항상 스케줄러가 실행 상태에 있어야 한다. 예제 8은 다음과 같이 동작한다.

ask1 (Listing 22)이 가장 높은 우선 순위로 생성되므로 먼저 실행하는 것이 보장된다. Task1 은 Task2 (우선 순위 23)의 우선 순위를 우선 순위보다 높게하기 전에 두 개의 문자열을 출력한다.

Task2는 상대적 우선 순위가 높아 지 자마자 실행을 시작한다 (실행 중 상태가 됩니다). 한 번에 하나의 작업 만 실행 중 상태가 될 수 있으므로 Task2가 실행 중 상태 인 Task1은 준비 상태이다.

Task2는 자신의 우선 순위를 Task1 보다 낮은 우선 순위로 설정하기 전에 메시지를 인쇄한다.

Task2가 우선 순위를 낮게 설정하면 Task1이 다시 한번 가장 우선 순위가 높은 작업이므로 다시 Task1이 실행 중 상태가되어 Task2가 다시 준비 상태가된다.

```
void vTask1( void *pvParameters )
{
    unsigned portBASE_TYPE uxPriority;

    /*이작업은우선순위가더높은 Task2가 생성되기전에항상실행된다. Task1 또는 Task2는항상차단되므로둘다항상실행또는준비상태가된다.

    이태스크가실행되는우선순위를쿼리하시오. NULL을전달하면 "내우선순위를반환합니다"를의미한다. */
    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )
    {
        /*이작업의이름을출력하시오. */
        vPrintString( "Task1 is running\r\n" );

        /* Task1 우선순위보다 Task2 우선순위를높게설정하면 Task2가즉시실행을시작하게된다
        (Task2는
```

생성된두작업의우선순위). 작업핸들사용에주목하십시오.

```
2 (xTask2Handle) vTaskPrioritySet () 호출합니다. Listing 24 는핸들을얻은방법을보여준 다. *  
/  
vPrintString( "About to raise the Task2 priority\r\n" );  
vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );  
  
/ * Task1 은 Task2 보다우선순위가높은경우에만실행된다. 따라서이작업이이시점에도달하 려면  
Task2 가이미실행되어이작업의우선순위보다낮은우선순위로다시설정되어있어야한다.  
* /  
}  
}
```

#### Listing 22 The implementation of Task1 in Example 8

```
void vTask2( void *pvParameters )  
{  
    unsigned portBASE_TYPE uxPriority;  
  
    / * Task1 은 우선 순위가 더 높은 Task1 이 생성 될 때마다 항상 이 작업 전에 실행된다. Task1 또는  
    Task2 가항상차단하지는않으므로항상 Running 또는 Ready 상태가된다.  
  
    이태스크가실행되는우선순위를쿼리하십시오. NULL 을전달하면 "내우선순위를반환합니다" 를의미한다. * /  
    uxPriority = uxTaskPriorityGet( NULL );  
  
    for( ;; )  
    {  
        / *이작업이이시점에도달하려면 Task1 이이미실행되어있어야하며이작업의우선순위를 자신보다높게설정해야한다.  
  
        이작업의이름을인쇄하십시오. * /  
        vPrintString( "Task2 is running\r\n" );  
  
        / * 우선순위를원래값으로되돌린다. 작업처리로 NULL 을전달하는것은 "우선순위변경" 을의미한다. 설정  
        Task1 보다우선순위가낮으면 Task1 이바로시작된다.  
        다시실행 -이작업을선택하십시오. * /  
        vPrintString( "About to lower the Task2 priority\r\n" );  
        vTaskPrioritySet( NULL, ( uxPriority - 2 ) );  
    }  
}
```

#### Listing 23 The implementation of Task2 in Example 8

각작업은유효한작업핸들을사용하지않고자체우선순위를쿼리하고설정할수있다. 대신 NULL 을 그대로 사용한다. 작업 핸들은 작업이 다른 작업을 참조하려고 할 때만 필요하다.

Task1 이 Task2 의 우선 순위를 변경할 때와 동일하다. Task1 이이를 수행하도록 허용하기 위해 Task2 핸들을 가져와서 Task2 가 Listing 24 의 주석에 강조 표시되어 생성 될 때 저장된다.

```
/ * Task2 의 핸들을 유지하는데 사용되는 변수를 선언하시오. * /
xTaskHandle xTask2Handle;

int main( void )
{
/ * 우선순위 2 의 첫 번째 작업을 만든다. 작업 매개 변수가 사용되지 않는다. NULL 로 설정하십시오.
태스크 핸들도 사용되지 않으므로 NULL 로 설정된다. * /
xTaskCreate( vTask1, "Task 1", 1000, NULL, 2, NULL );
/ * 작업은 우선순위 2 ^에서 생성된다.^ * /

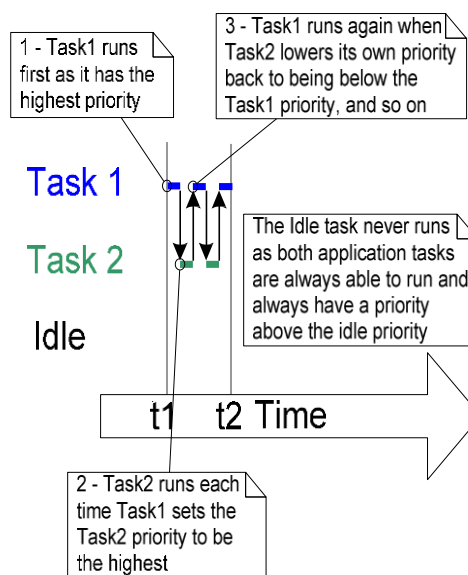
/ * Task1 에 주어진 우선순위보다 낮은 우선순위 1 에서 두 번째 작업을 만든다. 다시 태스크 매개 변수가 사용 되지 않으므로
NULL 로 설정된다. 하지만 이번에는 xTask2Handle 의 주소가 마지막 매개 변수에 전달되도록 태스크 핸들이 필요하다.
*/ xTaskCreate( vTask2, "Task 2", 1000, NULL, 1,
&xTask2Handle );
/ * 작업 핸들은 마지막 매개 변수이다 ^^^^^^^^^^^^^^^^^ * /

/ * 작업이 실행되기 시작하도록 스케줄러를 시작하십시오. * /
vTaskStartScheduler();

/ * 모든 것이 잘 되면 스케줄러가 이제 태스크를 실행할 때 main () 이여기에도달하지 않는다. main () 이여기 에도달하면,
유 휴 작업을 생성하는데 사용할 수 있는 힙 메모리가 부족하다.
5 장에서는 메모리 관리에 대한 자세한 정보를 제공한다. * /
for(;;); }
```

#### Listing 24 The implementation of main() for Example 8

그림 14 는 예제 8 태스크가 실행되는 순서를 보여 주며 결과 출력은 그림 15 와 같다.



**Figure 14 The sequence of task execution when running Example 8**

```
DOSBox 0.72, Cpu Cycles: 3000, Frameskip 0, Program: RTOSDEMO
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
```

**Figure 15 The output produced when Example 8 is executed**

## 1.9 DELETING A TASK

### vTaskDelete() API function

태스크는 vTaskDelete () API 함수를 사용하여 자체 또는 다른 태스크를 삭제할 수 있다.

삭제 된 작업은 더 이상 존재하지 않으며 실행 상태로 다시 들어갈 수 없다.

유휴 작업의 책임은 이후에 삭제 된 작업에 할당 된 메모리를 해제하는 것이다. 따라서 vTaskDelete () API 함수를 사용하는 응용 프로그램이 모든 처리 시간의 유휴 작업을 완전히 굶지 않게 하는 것이 중요 하다.

태스크가 삭제되면 커널 자체에 의해 태스크에 할당 된 메모리 만 자동으로 해제된다. 태스크의 구현이 자신을 할당하는 메모리 또는 기타 자원은 명시 적으로 해제해야 한다.

```
void vTaskDelete( xTaskHandle pxTaskToDelete );
```

**Listing 25 The vTaskDelete() API function prototype**

**Table 6 vTaskDelete() parameters**

Parameter Name/ReturnValue	Description
pxTaskToDelete	삭제할 작업의 핸들 (본 태스크) - 태스크 핸들을 얻는 방법에 대한 정보는 xTaskCreate () API 함수의 pxCreatedTask 매개 변수를 참조하십시오.

작업은 유효한 작업 핸들 대신 NULL 을 전달하여 자체를 삭제할 수 있다.

### Example 9. Deleting tasks

이것은 다음과 같이 동작하는 아주 간단한 예이다.

작업 1 은 우선 순위 1 로 main ()에 의해 생성된다. 실행될 때 우선 순위 2 로 작업 2 를 만든다. 작업 2 가 이제 가장 우선 순위가 높은 작업이므로 즉시 실행되기 시작한다. main ()의 소스는 Listing 26 에, Task 1 은 Listing 27 에있다.

작업 2 는 자체 삭제 만 수행한다. `vTaskDelete ()` 에 `NULL` 을 전달하여 자체를 삭제할 수 있지만 데모 목 적으로만 자체 태스 크 핸들 을 사용한다. `Task2` 의 소스 는 Listing 28 과 같다.

작업 2 가 삭제 되면 작업 1 이 다시 최 우선 순위의 작업 이 되므로 실행 을 계속 한다. 이 시점에서 `vTaskDelay ()` 를 호출 하여 잠시 차단 한다.

유 휴 작업 은 작업 1 이 차단 된 상태에서 실행 되고 현재 삭제 된 작업 2 에 할당 된 메모리 를 확보 한다.

작업 1 이 차단 된 상태 를 벗어나면 다시 최고 우선 순위의 준비 상태 작업 이 되므로 유 휴 작업 을 우선 적 용 한다. 실행 상태 가 되면 작업 2 가 다시 생성 되므로 계속 실행 된다.

```
int main( void )
{
    /* 우선 순위 1 에서 첫 번째 작업을 만든다. 작업 매개 변수가 사용되지 않으므로 NULL 로 설정된다.
    작업 핸들 도 사용되지 않으므로 마찬가지로 NULL 이 설정된다. */
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 1, NULL );

    /* 작업 은 우선 순위 1 ^에서 생성된다. */

    /* 작업 이 실행 되기 시작 하도록 스케줄러 를 시작 하시오. */
    vTaskStartScheduler();

    /* 스케줄러 가 시작 되면 main () 이 여기 에 도달 해서는 안 된다. */
    for( ;; );
}
```

Listing 26 The implementation of `main()` for Example 9

```
void vTask1( void *pvParameters )
{
    const portTickType xDelay100ms = 100 / portTICK_RATE_MS;

    for( ;; )
    {
        /* 이 작업 의 이름 을 출력 하시오. */
        vPrintString( "Task1 is running\r\n" );

        /* 작업 2 를 더 높은 우선 순위로 만든다. 다시 태스 크 매개 변수가 사용되지 않으므로 NULL 로 설정된
        다. 그러나 이번에는 xTask2Handle 의 주소가 마지막 매개 변수로 전달되도록 태스 크 핸들 이 필요하다. */
        xTaskCreate( vTask2, "Task 2", 1000, NULL, 2, &xTask2Handle );

        /* 작업 핸들 은 마지막 매개 변수 qustnd; 다 ^^^^^^^^^^^^^^^^^ * /

        * Task2 의 우선 순위 가 더 높다. 따라서 Task1 이 여기 에 도달 하려면 Task2 가 이미 실행 되어 삭제 되어야 한다. 100
        밀리 초 동안 지연. */
    }
}
```

```
vTaskDelay( xDelay100ms );  
}  
}
```

#### Listing 27 The implementation of Task 1 for Example 9

```
void vTask2( void *pvParameters )  
{
```

```
/* Task2 는 자체를 삭제하지만 아무것도 하지 않는다. 이렇게 하려면 매개변수로 NULL 을 사용하여 vTaskDelete  
( ) 를 호출할 수 있지만 데모용으로 대신 자체 태스크 핸들을 전달하는 vTaskDelete ( ) 를 호출한다. */  
vPrintString( "Task2 is running and about to delete  
itself\r\n" ); vTaskDelete( xTask2Handle ); }
```

#### Listing 28 The implementation of Task 2 for Example 9

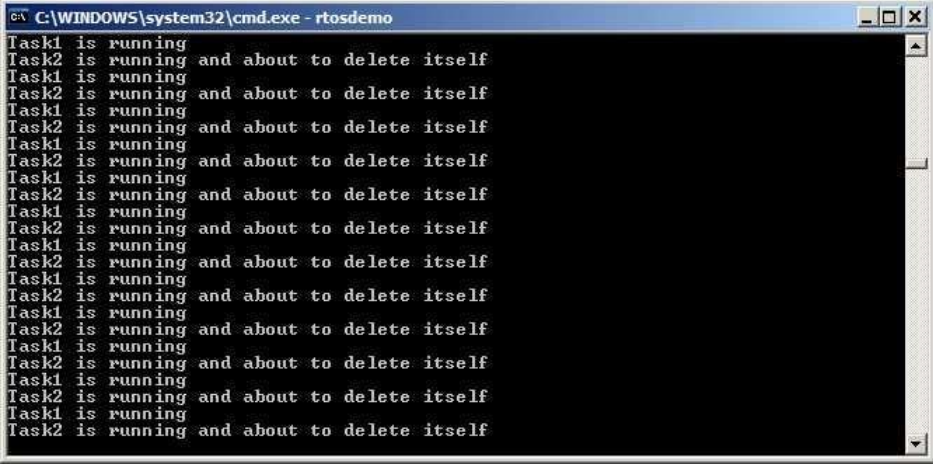
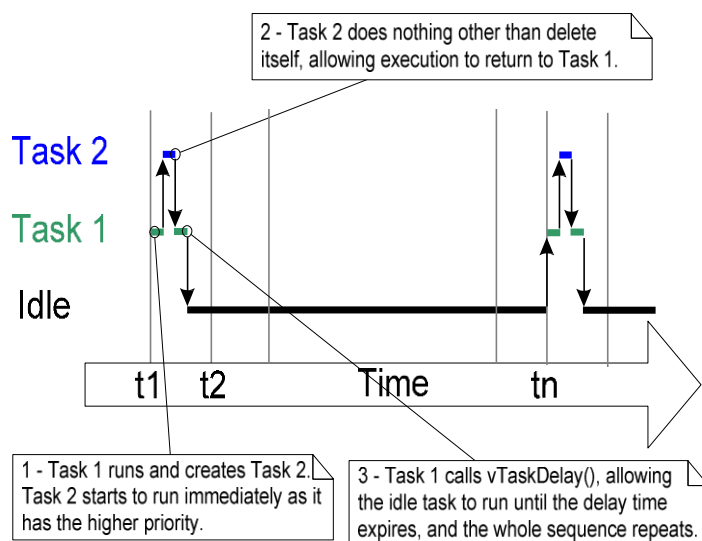


Figure 16 The output produced when Example 9 is executed





**Figure 17 The execution sequence for example 9**

## 10. THE SCHEDULING ALGORITHM – A SUMMARY

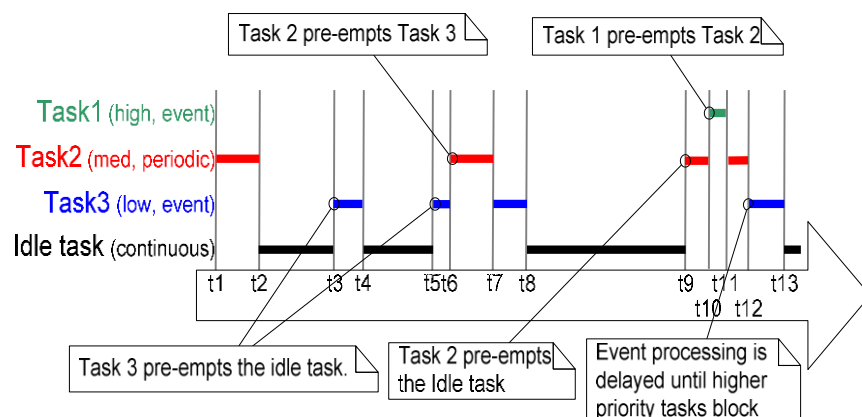
### Prioritized Preemptive Scheduling

이 장의 예에서는 FreeRTOS 가 실행 상태에 있어야하는 작업을 선택하는 방법과시기를 보여준다.

- 각 작업에는 우선 순위가 지정된다.
- 각작업은여러상태중하나에존재할수있다.
- 한 번에 하나의 작업 만 실행 중 상태로 존재할 수 있다.
- 스케줄러는 항상 가장 높은 우선 순위의 준비 상태 태스크를 선택하여 실행 상태로 전환한다

이러한 유형의 구성표를 '고정 우선 선점 예약 (Fixed Priority Preemptive Scheduling)'이라고 한다. 'Fixed Priority (고정 우선 순위)'는 각 태스크에 커널 자체가 변경하지 않는 우선 순위가 할당되기 때문에 (태스크 만이 우선 순위를 변경할 수 있음). 실행 상태 작업의 우선 순위가 낮으면 준비 상태로 들어가거나 우선 순위가 변경된 작업이 항상 실행 상태 작업을 우선하기 때문에 '선점 형'이다.

작업은 이벤트에 대해 차단된 상태에서 대기 할 수 있으며 이벤트가 발생할 때 자동으로 준비 됨 상태로 다시이동된다. 일시적인이벤트는특정시간에발생한다 (예 : 블록시간이만료된경우). 일반적으로 주기적 또는 시간 초과 동작을 구현하는 데 사용된다. 동기화 이벤트는 태스크 또는 인터럽트 서비스 루틴이 대기열 또는 많은 유형의 세마포어 중 하나에 정보를 보낼 때 발생한다. 이들은 일반적으로 주변 장 치에 도착하는 데이터와 같은 비동기 활동을 신호하는 데 사용된다. 그림 18은 가상 응용 프로그램의 실행 패턴을 보여줌으로써 이러한 모든 동작을 보여준다.



**Figure 18 Execution pattern with pre-emption points highlighted**

Referring to Figure 18:

#### 1.Idle Task

유틸리티 작업은 가장 낮은 우선 순위에서 실행되므로 높은 우선 순위의 작업이 준비 상태 (예 : 시간 t3, t5 및 t9)로 들어갈 때마다 선점된다.

#### 2.Task 3

작업 3 은 상대적으로 낮은 우선 순위이지만 유틸리티 작업 우선 순위보다 높은 우선 순위로 실행되는 이벤트 기반 작업이다. 대부분의 시간을 Blocked 상태에서 관심있는 이벤트를 기다리고, 이벤트가 발생할 때마다 Blocked 상태에서 Ready 상태로 전환한다. 모든 FreeRTOS 태스크 간 통신 메커니즘 (대기열, 세마 포어 등)을 사용하여 이벤트에 신호를 보내고 이러한 방식으로 작업을 차단 해제 할 수 있다.

이벤트는 시간 t3, t5 및 t9 와 t12 사이의 어딘가에 발생한다. 시간 t3 및 t5 에 발생하는 이벤트는이 시간에 즉시 처리된다. 작업 3 은 실행할 수있는 가장 우선 순위가 높은 작업이다. 시간 t9 와 t12 사이에 발생하는 이벤트는 t12 때까지 처리되지 않는다. 그때까지는 우선 순위가 높은 작업 인 Task 1 과 Task 2 가 아직 실행 중이기 때문이다. 작업 1 과 작업 2 가 모두 차단 상태에있는 것은 시간 t12에서만 작업 3 을 최우선 순 위의 준비 상태 작업으로 만든다.

#### 3.Task 2

작업 2 는 작업 3 의 우선 순위보다 우선하지만 작업 1 의 우선 순위보다 낮은 우선 순위로 실행되는주기적 인 작업이다. 기간 간격은 작업 2 가 시간 t1, t6 및 t9 에서 실행하려고한다는 것을 의미한다.

시간 t6 에서 작업 3 은 실행 중 상태이지만 작업 2 는 상대적 우선 순위가 높으므로 작업 3 을 선점하고 즉시 실행되기 시작한다. 작업 2 는 처리를 완료하고 t7 시점에 Blocked (차단됨) 상태로 다시 들어간다. 이 시점에서 작업 3 은 실행 중 상태를 다시 입력하여 처리를 완료 할 수 있다. 작업 3 자체가 시간 t8 에서 차단된다.

#### 4.Task 1

작업 1 은이벤트기반작업이기도하다. 가장우선순위가높은것으로실행되므로시스템의다른작업을 선점 할 수 있다. 표시된 유일한 작업 1 이벤트는 시간 t10 에 발생한다. 작업 1 은 작업 2 를 선점한다. 작업 2 는 작업 1 이 시간 t11 에 다시 입력 된 후에 처리를 완료 할 수 있다.

## Selecting Task Priorities

그림 18은 애플리케이션의 작동 방식에 대한 기본 우선 순위 지정 방법을 보여준다.

일반적으로 하드 실시간 기능을 구현하는 작업에는 소프트 실시간 기능을 구현하는 작업보다 우선 순위가 할당된다. 그러나 실행 시간 및 프로세서 사용률과 같은 다른 특성도 고려해야 하므로 전체 응용 프로그램이 어려운 실시간 마감 시간을 놓치지 않는다.

Rate Monotonic Scheduling (RMS)은 주기적인 실행 속도 작업에 따라 각 작업에 고유한 우선 순위를 지정하는 일반적인 우선 순위 할당 기술이다. 가장 높은 우선 순위는 주기적 실행 빈도가 가장 높은 작업에 할당된다. 가장 낮은 우선 순위는 주기 실행 빈도가 가장 낮은 태스크에 할당된다. 이러한 방식으로 우선 순위를 지정하는 것은 전체 애플리케이션의 '스케줄 가능성'을 극대화하는 것으로 나타났다. 나쁜 타이밍 변형과 모든 작업이 주기적이지 않은 사실로 절대 계산을 복잡한 프로세스로 만든다.

## Co-operative Scheduling

이 책에서는 선점 예약에 중점을 둡니다. 스케줄링 FreeRTOS는또한선택적으로협동조합을사용할 수 있다.

순수한 협동 작업 스케줄러가 사용되는 경우 실행 상태 태스크가 차단된 상태로 들어가거나 실행 상태 태스크가 명시적으로 `taskYIELD()`를 호출하는 경우에만 컨텍스트 전환이 발생한다. 작업은 선점되지 않으며 동일한 우선 순위의 작업은 자동으로 처리 시간을 공유하지 않는다. 이러한 방식의 협동적 스케줄링은 더 간단하지만 반응이 느린 시스템이 될 수 있다.

또한 인터럽트 서비스 루틴을 사용하여 명시적으로 컨텍스트 전환을 발생시키는 하이브리드 체계가 가능하다. 이렇게 하면 동기화 이벤트가 선점을 유발할 수 있지만 일시적인 이벤트는 발생하지 않는다. 결과는 타이밍 슬리핑이 없는 선점 시스템이다. 이것은 효율성 향상 때문에 바람직할 수 있으며 일반적인 스케줄러 구성이다.

# CHAPTER2

## QUEUE MANAGEMENT

## 1. CHAPTER INTRODUCTION AND SCOPE

FreeRTOS 를 사용하는 응용 프로그램은 독립적인 작업 세트로 구성되어 있다. 각 작업은 사실상 자체 적으로 미니 프로그램이다. 이 자율적인 작업 컬렉션은 유용한 시스템 기능을 제공 할 수 있도록 서로 통신해야 한다. 대기열은 모든 FreeRTOS 통신 및 동기화 메커니즘에서 사용되는 기본 프리미티브이다.

### Scope

이 장은 독자들에게 다음에 대한 좋은 이해를 제공하고자 한다 :

- 대기열을 만드는 방법.
- 큐가 포함하는 데이터를 관리하는 방법.
- 대기열로 데이터를 보내는 방법.
- 대기열에서 데이터를 받는 방법.
- 대기열에서 차단하는 것이 의미하는 것.
- 효과 작업 우선 순위는 대기열에 쓰거나 대기열에서 읽을 때 있다.

태스크 간 통신 만이 장에서 다룹니다. 태스크 통신에 대한 인터럽트 및 인터럽트 태스크는 제 3 장에서 다룬다.

## 2.2 CHARACTERISTICS OF A QUEUE

### Data Storage

대기열은 유한 크기의 고정 크기 데이터 항목을 보유 할 수 있다. 대기열에 보관할 수 있는 최대 항목 수를 '길이'라고 한다. 각 데이터 항목의 길이와 크기는 대기열이 만들어 질 때 설정된다.

일반적으로 대기열은 FIFO (First In First Out) 버퍼로 사용되며 데이터는 대기열의 끝 (꼬리)에 쓰여지고 대기열의 앞면 (머리)에서 제거됩니다. 대기열의 앞에 쓸 수도 있다.

대기열에 데이터를 쓰면 데이터의 바이트 복사를 위한 바이트가 대기열 자체에 저장된다. 대기열에서 데이터를 읽으면 데이터 사본이 대기열에서 제거된다. 그림 19 는 대기열에 쓰여지고 읽혀지는 데이터 와 각 작업의 대기열에 저장된 데이터에 대한 효과를 보여준다.

### Access by Multiple Tasks

대기열은 소유권이 없거나 특정 작업에 할당되지 않은 공유 한 개체이다. 임의의 수의 작업이 동일한 대기열에 쓸 수 있으며 여러 작업이 동일한 대기열에서 읽을 수 있다. 여러 개의 작성자가 있는 대기열 은 매우 일반적이지만 여러 개의 독자가 있는 대기열은 매우 드물다.

### Blocking on Queue Reads

태스크가 큐에서 읽기를 시도 할 때 선택적으로 '블록'시간을 지정할 수 있다. 대기열이 이미 비어있는 경우 대기열에서 데이터를 사용할 수 있을 때까지 기다릴 작업이 차단된 상태로 유지되어야 하는 시간 이다. 대기열에서 데이터를 사용할 수 있게 대기하는 차단된 상태에 있는 작업은 다른 작업 또는 인터럽트가 데이터를 대기열에 넣을 때 자동으로 준비 된 상태로 이동된다. 데이터가 사용 가능하게 되기 전에 지정된 블록 시간이 만료되면 작업이 차단된 상태에서 준비 된 상태로 자동 이동된다.

대기열에는 여러 개의 독자가 있을 수 있으므로 하나의 대기열에서 둘 이상의 작업이 데이터 대기를 차단할 수 있다. 이 경우 데이터가 사용 가능 해지면 하나의 작업 만 차단 해제된다. 차단 해제 된 작업은 항상 데이터를 기다리는 최우선 순위 작업이다. 차단 된 작업이 동등한 우선 순위를 갖는 경우 차단되 지 않은 데이터를 가장 많이 기다린 작업이 된다.

### Blocking on Queue Writes

대기열에서 읽을 때와 마찬가지로 작업은 선택적으로 대기열에 쓸 때 블록 시간을 지정할 수 있다. 이 경우 블록 시간은 대기열이 이미 가득 차면 대기열에서 공간을 사용할 수 있을 때까지 대기 할 작업이 차단된 상태로 유지되어야 하는 최대 시간이다.

대기열에는 여러 작성자가 있을 수 있으므로 전체 대기열에서 보내기 작업을 완료하기 위해 대기중인 둘 이상의 작업이 차단 될 수 있다. 이 경우 하나의 작업 만 수행된다. 대기열의 공간을 사용할 수 있게 되면 차단 해제된다. 차단 해제 된 작업은 항상 공간을 기다리는 최우선 순위 작업이다. 차단 된 작업이 동등한 우선 순위를 갖는다면 차단되지 않은 가장 긴 공간을 기다리고 있 는 작업이 된다.

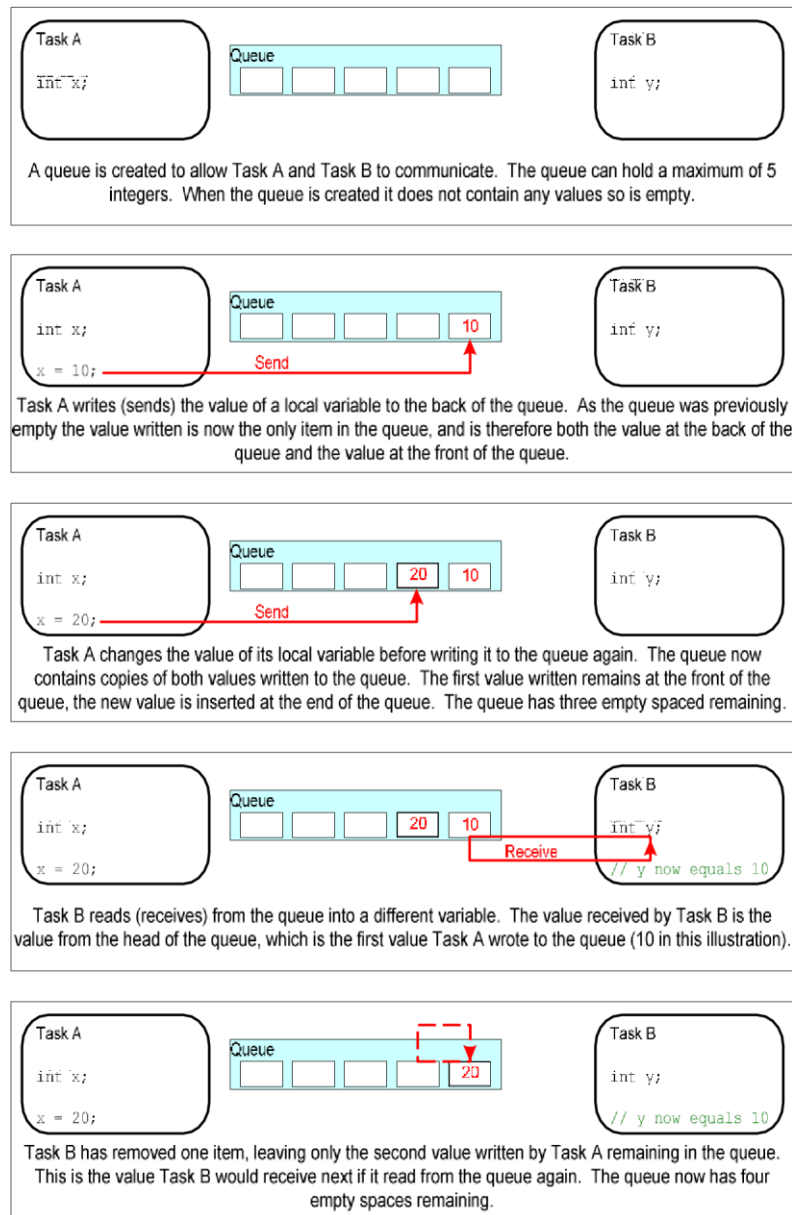


Figure 19 An example sequence of writes and reads to/from a queue

## 2.3 USING A QUEUE

### xQueueCreate() API Function

큐는 사용하기 전에 명시 적으로 작성되어야 한다.

큐는 `xQueueHandle` 유형의 변수를 사용하여 참조된다. `xQueueCreate ()`는 큐를 생성하는데 사용되며 `xQueueHandle` 을 리턴하여 생성 된 큐를 참조한다.



FreeRTOS 는 큐가 생성 될 때 FreeRTOS 힙에서 RAM 을 할당한다. RAM 은 큐 데이터 구조와 큐에 포함 된 항목을 모두 보유하는 데 사용된다. xQueueCreate ()는 큐 생성에 사용할 수있는 힙 RAM 이 충분하지 않은 경우 NULL 을 반환한다. 5 장에서는 힙 메모리 관리에 대해 자세히 설명한다.

```
xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength,
                           unsigned portBASE_TYPE uxItemSize
                           );
```

#### Listing 29 The xQueueCreate() API function prototype

Table 7 xQueueCreate() parameters and return value

Parameter Name	Description
uxQueueLength	생성되는큐가한번에보유할수있는최대항목수이다.
uxItemSize	대기열에 저장할 수 있는 각 데이터 항목의 크기 (바이트).
ReturnValue	NULL 이 리턴되면 FreeRTOS 가 큐 데이터 구조와 저장 영역을 할당하기에 사용할 수있는 힙 메모리가 충분하지 않아서 큐를 작성할 수 없다. 리턴되는 NULL 이 아닌 값은 큐가 성공적으로 작성되었음을 나타낸다. 반환 된 값은 생성 된 큐에 대한 핸들로 저장되어야 한다.

#### xQueueSendToBack() and xQueueSendToFront() API Functions

예상대로 xQueueSendToBack ()은 큐의 뒤 (꼬리)로 데이터를 보내고, xQueueSendToFront ()는 큐의 앞 (머리)으로 데이터를 보내는 데 사용된다 xQueueSend ()는 xQueueSendToBack ()과 동일하고 정확히 동일하다.

인터럽트 서비스 루틴에서 xQueueSendToFront () 또는 xQueueSendToBack ()을 호출하지 마시오. 그 대신 인터럽트 안전 버전 인 xQueueSendToFrontFromISR ()과 xQueueSendToBackFromISR ()을 사용 해야한다. 이것들은 제 3 장에서 설명한다.

```
portBASE_TYPE xQueueSendToFront( xQueueHandle xQueue,
                                  const void * pvItemToQueue, portTickType
                                  xTicksToWait );
```

**Listing 30 The xQueueSendToFront() API function prototype**

```
portBASE_TYPE xQueueSendToFront( xQueueHandle xQueue,  
const void * pvItemToQueue, portTickType  
xTicksToWait );
```

**Listing 31 The xQueueSendToBack() API function prototype**

Table 8 xQueueSendToFront() and xQueueSendToBack() function parameters and return value

Parameter Name/Returned Value	Description
xQueue	데이터가 보내지는 (쓰여지는) 큐의 핸들. 큐 핸들은 큐를 생성하는 데 사용 된 xQueueCreate ()에 대한 호출에서 반환된다.
pvItemToQueue	대기열에 복사 될 데이터의 포인터.  대기열이 생성 할 수 있는 각 항목의 크기는 대기열이 생성 될 때 설정되므로 이 많은 바이트는 pvItemToQueue 에서 대기열 저장소 영역으로 복사된다.

Table 8 xQueueSendToFront() and xQueueSendToBack() function parameters and return value

Parameter Name/Returned Value	Description
xTicksToWait	대기열이 이미 가득차면 대기열에서 공간을 사용할 수 있을 때까지 대기 할 작업이 차단됨 상태로 유지되어야 하는 최대 시간이다.  xTicksToWait 가 0 이고 큐가 이미 가득 찬 경우 xQueueSendToFront () 및 xQueueSendToBack ()이 즉시 반환된다.  블록 시간은 틱 주기로 지정되므로 절대 시간은 틱 주파수에 따라 달라진다. 상수 portTICK_RATE_MS 를 사용하여 밀리 초 단위로 지정된 시간을 틱으로 지정된 시간으로 변환할 수 있다.  xTicksToWait 을 portMAX_DELAY 로 설정하면 FreeRTOSConfig.h 에서 INCLUDE_vTaskSuspend 가 1 로 설정되어 태스크가 무기한 대기한다 (시간

초과 없이).

Returnedvalue 가능한 두 가지 반환 값이 있다. :

1. pdPASS

pdPASS 는 데이터가 큐에 성공적으로 전송 된 경우에만 리턴된다.

블록 시간이 지정된 경우 (xTicksToWait 이 0 이 아님) 호출 한 작업이 차단 된 상태로 설정되어 함수가 반환되기 전에 공간이 대기열에서 사용 가능할 때까지 대기 할 수 있다. 이전에 데이터가 대기열에 성공적으로 기록되었다. 블록 시간이 만료되었다.

2. errQUEUE\_FULL

대기열이 이미 가득 차서 데이터를 대기열에 쓸 수 없으면 errQUEUE\_FULL 이 반환된다.

블록 시간이 지정되면 (xTicksToWait 이 0 이 아님) 호출 작업은 Blocked 상태로 전환되어 다른 작업이나 인터럽트가 대기열에 들어가기까지 기다리지 만 지정된블록시간이만료되기전에만료된다.

### **xQueueReceive() and xQueuePeek() API Functions**

xQueueReceive ()는 큐에서 항목을 수신 (읽기)하는 데 사용된다. 수신 된 항목이 대기열에서 제거된다.

xQueuePeek ()는 항목이 대기열에서 제거되지 않고 대기열에서 항목을 수신하는 데 사용된다. xQueuePeek ()는 큐에 저장된 데이터 나 큐에 데이터가 저장되는 순서를 수정하지 않고 큐의 헤드에서 항목을받는다.

인터럽트 서비스 루틴에서 xQueueReceive () 또는 xQueuePeek ()를 호출하지 마시오. 인터럽트 안전 xQueueReceiveFromISR () API 함수는 3 장에서 설명한다.

```
portBASE_TYPE xQueueReceive( xQueueHandle xQueue,  const void *  
pvBuffer,  portTickType xTicksToWait );
```

**Figure 20 The xQueueReceive() API function prototype**

```
portBASE_TYPE xQueuePeek( xQueueHandle xQueue,  const void * pvBuffer,  
portTickType xTicksToWait );
```

**Listing 32 The xQueuePeek() API function prototype**

**Table 9 xQueueReceive() and xQueuePeek() function parameters and return values**

Parameter Name/Returned value	Description
xQueue	데이터가 수신되는 큐의 핸들 (읽기). 큐 핸들은 큐를 생성하는 데 사용된 xQueueCreate ()에 대한 호출에서 반환된다.
pvBuffer	수신 된 데이터가 복사 될 메모리의 포인터. 큐가 보유하는 각 데이터 항목의 크기는 큐가 작성 될 때 설정된다. pvBuffer 가 가리키는 메모리는 적어도 그 많은 바이트를 저장할만큼 커야 한다.

**Table 9 xQueueReceive() and xQueuePeek() function parameters and return values**

Parameter Name/Returned value	Description
-------------------------------	-------------

**xTicksToWait** 대기열이 이미 비어있는 경우 데이터가 대기열에서 사용 가능할 때까지 대기하는 작업이 차단된 상태로 유지되어야 하는 최대 시간이다.

xTicksToWait 가 0 이면 큐가 이미 비어있는 경우 xQueueReceive ()와 xQueuePeek ()이 즉시 반환된다.

블록 시간은 틱 주기로 지정되므로 절대 시간은 틱 주파수에 따라 달라진다. 상 수 portTICK\_RATE\_MS 를 사용하여 밀리 초 단위로 지정된 시간을 틱으로 지정 된시간으로변환할수있다.

xTicksToWait 을 portMAX\_DELAY 로 설정하면 FreeRTOSConfig.h 에서 INCLUDE\_vTaskSuspend 가 1 로 설정되어 태스크가 무기한 대기한다 (시간 초 과없이)

**Returnedvalue** 두 가지 가능한 반환 값이 있다.

1. pdPASS

pdPASS 는 데이터가 큐에서 성공적으로 읽 t 진 경우에만 리턴된다.

블록시간이지정된경우 (xTicksToWait 이 0 이아님) 호출하는작업이차단상 태에 놓여 데이터가 대기열에서 사용 가능할 때까지 대기 할 수 있지만 블 록 시간이 만료되기 전에 데이터가 대기열에서 성공적으로 읽힐 가능성이 있다 .

2. errQUEUE\_EMPTY

큐가 이미 비었기 때문에 큐에서 데이터를 읽을 수 없을 때 errQUEUE\_EMPTY 가 리턴된다.

블록 시간이 지정되면 (xTicksToWait 이 0 이 아님) 호출 태스크는 Blocked 상 태로 전환되어 다른 태스크를 기다리거나 인터럽트가 대기열로 데이터를 보내지 만 블록 시간이 만료되기 전에 만료된다.

**uxQueueMessagesWaiting() API Function**

uxQueueMessagesWaiting ()은 현재 대기열에있는 항목 수를 쿼리하는 데 사 용된다.

인터럽트 서비스 루틴에서 uxQueueMessagesWaiting ()을 호출하지 마십시

오. 인터럽트 안전 `uxQueueMessagesWaitingFromISR()` 을 대신 사용해야 한다.

```
unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle xQueue );
```

**Listing 33 The `uxQueueMessagesWaiting()` API function prototype**

**Table 10 `uxQueueMessagesWaiting()` function parameters and return value**

Parameter Name/Returned Value	Description
<code>xQueue</code>	질의되는 큐의 핸들. 큐 핸들은 큐를 생성하는 데 사용된 <code>xQueueCreate()</code> 에 대한 호출에서 반환된다.
Returned value	쿼리중인 큐가 현재 보유하고 있는 항목 수이다. 0 이 리턴되면 큐는 비어 있다.

#### **Example 10. Blocking When Receiving From a Queue**

이 예에서는 작성중인 대기열, 여러 태스크에서 대기열로 보내지는 데이터 및 대기열에서 수신중인 데이터를 보여준다. 대기열은 `long` 유형의 데이터 항목을 보유하기 위해 작성된다. 대기열로 보내는 작업은 차단 시간을 지정하지 않지만 대기열에서받는 작업은 지정하지 않는다.

대기열로 보내는 작업의 우선 순위는 대기열에서받는 작업의 우선 순위보다 낮다. 즉, 대기열에 데이터를 보내자마자 수신 작업이 차단을 해제하고 전송 작업을 선점하고 데이터를 제거하여 대기열을 다시 한 번 비울 수 있기 때문에 대기열에 항목이 두 개 이상 포함되어서는 안 된다.

Listing 34 는 큐에 쓰는 태스크의 구현을 보여준다. 이 작업의 두 인스턴스가 만들어지며 값 100 을 큐에 지속적으로 쓰고 다른 인스턴스는 값 200 을 동일한 큐에 계속 쓴다. `task` 매개 변수는 이 값을 각 태스크 인스턴스에 전달하는 데 사용된다.

```
static void vSenderTask( void *pvParameters )
{
    long lValueToSend;    portBASE_TYPE xStatus;

    /* 이작업의두인스턴스가만들어지므로대기열로보내지는값이작업매개변수를통해전달된다.
    이렇게하면각인스턴스가다른값을사용할수있다. 대기열은 long 유형의값을보유하기위해 작성되었으며,
    매개변수를필수유형으로변환하시오. */
    lValueToSend = ( long ) pvParameters;

    /* 대부분의작업에따라이작업은무한루프내에서구현된다. */ for( ;; )
    {
        /* 대기열에값을보낸다.
```

첫번째매개변수는데이터를보낼큐이다. 큐는스케줄러가시작되기전에작성되었으므로이태스크가실행되기전에시작된다. 두번째매개변수는전송할데이터의주소이며, 이경우 lValueToSend 의주소이다.

세번째매개변수는차단시간 - 대기열이이미가득차면대기열에서공간을사용할수있을때까지 대기할작업이차단됨상태로유지되어야하는시간이다. 이경우대기열에둘 이상의항목이없어야 하므로블록시간이지정되지않으므로절대채워지지않는다. \* /

```
xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );
```

```
if( xStatus != pdPASS )
```

```
{
```

```
/* 대기열이가득차서보내기작업을완료할수없다. 대기열에둘 이상의항목이없어야하므로 오류가발생해야한다! * /
```

```
vPrintString( "Could not send to the  
queue.\r\n" ); }
```

```
/* 다른보낸사람작업을실행하도록허용한다. taskYIELD () 는현재시간조각이끝날때까지이
```

```
작업을실행상태로유지하는대신다른작업으로의전환이지금발생해야한다는것을스케줄러에게 알린다. * /
```

```
taskYIELD();
```

```
}
```

```
}
```

#### Listing 34 Implementation of the sending task used in Example 10.

Listing 35 는 큐에서 데이터를받는 태스크의 구현을 보여준다. 수신 태스크는 100 밀리 초의 블록 시간을 지정하므로 데이터가 사용 가능하게 될 때까지 대기 할 Blocked 상태가 됩니다. 대기열에서 데이터를 사용할 수 있거나 데이터를 사용할 수 없으면 100 밀리 초가 지나면 차단된 상태가 유지된다. 이 예에서 대기열에 연 속적으로쓰는두가지작업이있으므로 100 밀리초제한시간이만료되지않아 야 한다.

```
static void vReceiverTask( void *pvParameters )
```

```
{
```

```
/* 대기열에서받은값을보유할변수를선언하시오. * /
```

```
long lReceivedValue; portBASE_TYPE xStatus; const  
portTickType xTicksToWait = 100 / portTICK_RATE_MS;
```

```
/*이작업은또한무한루프내에서정의된다. * /
```

```
for( ;; ) {
```

```
/*이호출은대기열에기록된데이터를즉시제거하므로이대기열은항상대기열을찾는다. * /
```

```
if( uxQueueMessagesWaiting( xQueue ) != 0 )
```

```
{
```

FreeRTOS

Designed For Microcontrollers;

© 2009 Richard Barry. Distribution or publication in any form is strictly prohibited.

```
vPrintString( "Queue should have been  
empty!\r\n" ); }
```

/ \* 대기열에서데이터를수신한다.

첫번째매개변수는데이터를받을큐이다. 대기열은스케줄러가시작되기전에작성되므로이태스크가 처음실행되기전에작성된다.

두번째매개변수는수신된데이터가놓일버퍼이다. 이경우버퍼는단순히수신된데이터를보유하  
는데필요한크기를갖는변수의주소이다.

마지막매개변수는블록시간 - 대기열이이미비어있는경우데이터가사용가능할때까지대기할작  
업이차단됨상태로유지되어야하는최대시간이다. 이경우 constant portTICK\_RATE\_MS 는 100

100 milliseconds 를틱으로지정된시간으로변환하는데사용된다. \* /

```
xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );
```

```
if( xStatus == pdPASS )
```

```
{
```

/ \* 대기열에서데이터를성공적으로수신하여수신된값을출력한다. \* /

```
vPrintStringAndNumber( "Received = ", lReceivedValue );
```

```
}
```

```
else
```

```
{
```

/ \* 100ms 를기다린후에도대기열에서데이터를받지못했다. 전송작업이자유롭게실행되고지속적  
으로대기열에기록되므로오류가발생해야한다. \* /

```
vPrintString( "Could not receive from the queue.\r\n" );
```

```
}
```

```
}
```

```
}
```

**Listing 35 Implementation of the receiver task for Example 10.**

Listing 36 에는 main () 함수의 정의가 들어있다. 이렇게 하면 스케줄러를 시작하기 전에 대기열과 세  
가 지 태스크가 생성된다. 작업의 우선 순위가 설정되어 있어도 대기열이 실제로 한 번에 둘 이상의  
항목을 포함하지 않을지라도 대기열은 최대 5 개의 긴 값을 보유하도록 작성된다.

/ \* xQueueHandle 유형의변수를선언하시오. 이것은세가지모든작업에서액세스하는대기열에대한참조를  
저장하는데사용된다. \* /

```
xQueueHandle xQueue;
```

```
int main( void )
```

```
{
```



```
/* 대기열은 최대 5 개의 값을 보유하기 위해 생성되며, 각 값은 long 유형의 변수를 보유할 수 있을 만큼 충분 히크다. */
xQueue = xQueueCreate( 5, sizeof( long ) );

if( xQueue != NULL )
{
    /* 대기열로 보낼 작업의 두 인스턴스를 만든다. task 매개 변수는 태스크가 큐에 쓸 값을 전달하는데 사용되며
    로 다른 태스크가 지속적으로 대기열에 200 을 쓰는 동안 한 태스크는 대기열에 100 을 계속 기록한다. 두 작업 모두 우선 순위
    1 로 생성된다. */
    xTaskCreate( vSenderTask, "Sender1", 1000, ( void * ) 100, 1, NULL );
    xTaskCreate( vSenderTask, "Sender2", 1000, ( void * ) 200, 1, NULL );

    /* 대기열에서 읽을 작업을 만든다. 작업은 우선 순위 2 로 생성되므로 보낸 사람 작업의 우선 순위보다 높다. */
    /* xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2,
    NULL ); */

    /* 생성된 작업이 실행되기 시작하도록 스케줄러를 시작하십시오. */ vTaskStartScheduler();
}
else
{
    /* 대기열을 만들 수 없다. */
}

/* 모든 것이 잘 되면 스케줄러가 이제 태스크를 실행할 때 main () 이여기에도 달하지 않는다. main () 이여기 에도 달하면,
유류 작업을 생성하는데 사용할 수 있는 힙 메모리가 부족하다.
5 장에서는 메모리 관리에 대한 자세한 정보를 제공한다. */
for( ;; ); }
```

#### Listing 36 The implementation of main() Example 10

대기열로 보내는 태스크는 무한 루프의 각 반복에서 taskYIELD ()를 호출한다.

현재 시간 조각의 끝까지 실행 상태의 작업을 실행하는 대신 다른 작업으로의 전환이 지금 발생해야 한다는 것을 스케줄러에게 알린다. taskYIELD ()가 실제로 Running 상태에서 제거되도록 자원 하고 있다. 대기열로 보내는 두 작업은 모두 taskYIELD 를 호출 할 때마다 동일한 우선 순위를 가지므로 다른 작업은 실행을 시작한다. taskYIELD ()를 호출하는 작업은 다른 작업을 대기 상태로 이동 하면서 준비 상태로 이동한다.

실행 상태. 이렇게 하면 두 개의 송신 작업이 차례대로 데이터를 대기열로 보낸다. 예제 10 에 의해 생성 된 출력이 그림 21 에 나와 있다.

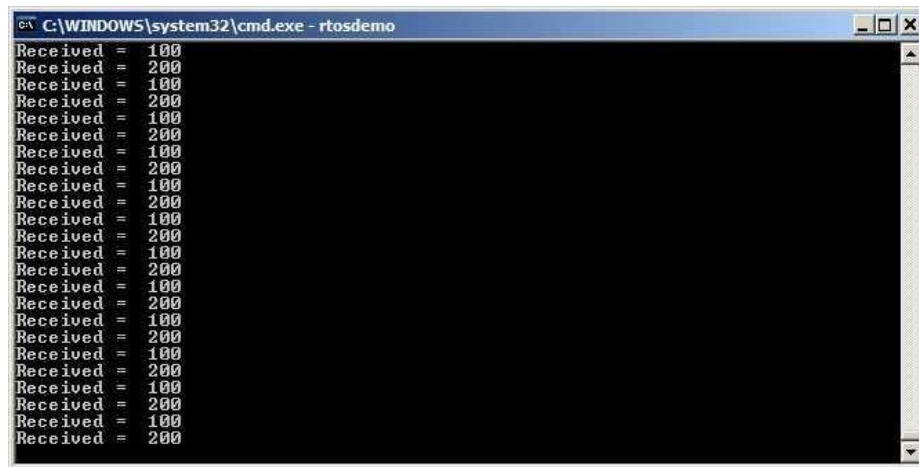


Figure 21 The output produced when Example 10 is executed Figure 22

demonstrate the sequence of execution.

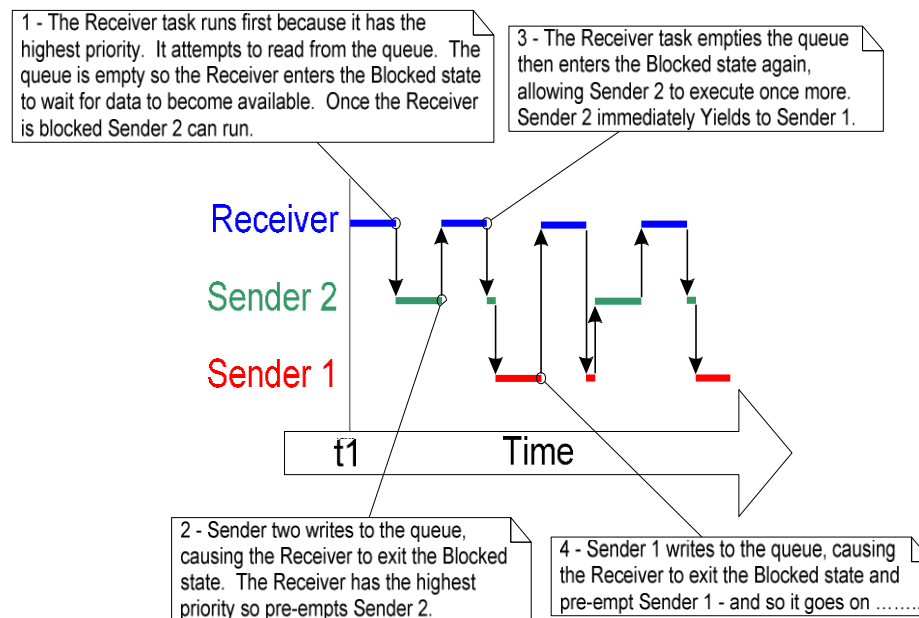
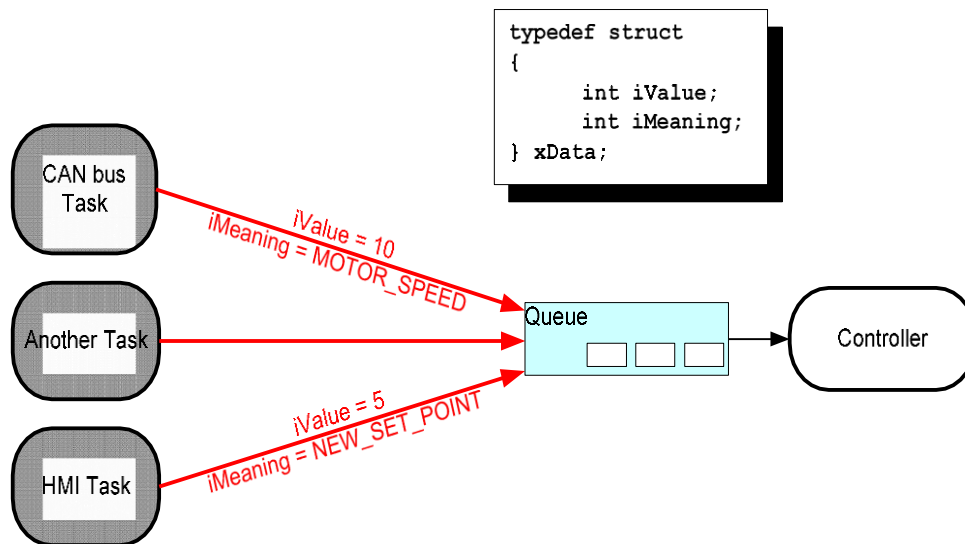


Figure 22 The sequence of execution produced by Example 10

## Using Queues to Transfer Compound Types

하나의 큐에서 여러 소스의 데이터를 받는 것이 일반적이다. 종종 데이터의 수신자는 데이터의 출처를 알아야 처리 방법을 결정할 수 있다. 이를 달성하는 한 단한 방법은 큐 값을 사용하여 데이터 값과 데이터 소스가 모두 구조 필드에 포함 된 구조를 전송하는 것이다. 이 구성표는 그림 23에 나와 있다.



**Figure 23 An example scenario where structures are sent on a queue**

Referring to Figure 23:

xData 유형의 구조를 보유하는 대기열이 작성된다. 구조체 멤버는 데이터 값과 하나의 메시지에서 큐로 보내지는 데이터의 의미를 나타내는 코드를 모두 허용한다.

중앙 제어기 태스크는 기본 시스템 기능을 수행하는 데 사용된다. 이것은 큐에 전달된 시스템 상태에 대한 입력 및 변경 사항에 대응해야 한다.

CAN 버스 작업은 CAN 버스 인터페이스 기능을 캡슐화하는 데 사용된다. CAN 버스 태스크가 메시지를 수신하고 디코딩하면 이미 디코딩된 메시지를 xData 구조의 컨트롤러 태스크로 보낸다. 전달된 구조체의 iMeaning 멤버는 제어 작업에 데이터가 무엇인지 알리는 데 사용된다. 설명된 경우에는 모터 속도이다. 이전된 구조의 iValue 멤버는 제어 작업에 실제 모터 속도 값을 알리는 데 사용된다.

휴먼 머신 인터페이스 (HMI) 태스크는 모든 HMI 기능을 캡슐화하는 데 사용된다. 기계 운영자는 HMI 작업 내에서 감지되고 해석되어야 하는 여러 가지 방법으로 명령을 입력하고 값을 쿼리할 수 있다. 새로운 명령이 입력되면 HMI 태스크는 명령을 xData 구조의 컨트롤러 태스크로 보낸다. 전달된 구조체의 iMeaning 멤버는 제어 작업에 데이터가 무엇인지 알리는 데 사용된다. 설명된 경우 새 설정 값이다. 이전된 구조의 iValue 멤버를 사용하여 Controlling 태스크가 실제 설정 값을 알 수 있다.

## Example 11. Blocking When Sending to a Queue / Sending Structures on a Queue

예제 11 은 예제 10 과 비슷하지만 태스크 우선 순위가 반대로되어 수신 태스크의 우선 순위가 송신 태스크보다 낮다. 또한 대기열은 단순한 긴 정수가 아닌 작업간에 구조를 전달하는 데 사용된다.

Listing 37 은 예제 11 에 사용 된 구조의 정의를 보여준다.

```
/ * 대기열에전달될구조체유형을정의하시오. * /
typedef struct
{
    unsigned char ucValue;  unsigned char
    ucSource; } xData;

/ * 대기열에전달될 xData 유형의두변수를선언한다. * /
static const xData xStructsToSend[ 2 ] =
{
    { 100, mainSENDER_1 }, /* Sender1 에서사용한다. */
    { 200, mainSENDER_2 } /* Sender2 에서사용한다. */
};
```

**Listing 37** The definition of the structure that is to be passed on a queue, plus the declaration of two variables for use by the example.

(대기열에전달할구조의정의와예제에서사용할두변수의선언.)

예 10 에서 수신 작업은 가장 높은 우선 순위를 가지므로 대기열에는 두 개 이상의 항목이 포함되지 않았다. 이는 수신 작업이 데이터가 대기열에 배치되는 즉시 전송 작업을 선점했기 때문이다. 예 11 에서는 보내는 작업의 우선 순위가 높아지므로 대기열은 일반적으로 가득 찰 것이다. 이는 수신 작업이 대기열에서 항목을 제거하자마자 전송 작업 중 하나가 대기열을 채우는 즉시 대기열을 다시 채울 것이기 때문이다. 그런 다음 전송 작업은 대기열에서 공간을 다시 사용할 수 있게 될 때까지 대기하기 위해 차단된 상태로 다시 들어간다.

Listing 38 은 전송 태스크 구현을 보여준다. 보내는 작업은 100 milliseconds 의 블록 시간을 지정하므로 대기열이 가득 찰 때마다 공간이 사용 가능하게 될 때까지 대기하기 위해 차단된 상태가 된다. 큐에서 공간을 사용할 수 있거나 100 milliseconds 가 지나면 사용할 수 없게 차단 상태가 된다. 이 예제에서받는 작업은 대기열에서 항목을 제거하여 공간을 계속 만들고 있으므로 100 milliseconds 시간 초과가 절대로 없어야 한다..

```
static void vSenderTask( void *pvParameters )
{
    portBASE_TYPE xStatus;
    const portTickType xTicksToWait = 100 / portTICK_RATE_MS;
```

```

/ * 대부분의작업에따라이작업은무한루프내에서구현된다. * / for( ;; )
{
/ * 대기열로보낸다.

두번째매개변수는전송할구조의주소이다. 주소는 task 매개변수로전달되므로 pvParameters 가직접사용된 다.

세번째매개변수는차단시간 - 대기열이이미꽂차면대기열에서공간을사용할수있을때까지대기할작업
이차단됨상태로유지되어야하는시간입니다. 큐를가득채울것으로예상되는송신타스크가수신타스크보다우
선순위가높으므로블록시간이지정됩니다. 수신타스크는두개의전송타스크가모두차단상태일때큐에서
항목을실행하고제거합니다. * /
xStatus = xQueueSendToBack( xQueue, pvParameters, xTicksToWait );

if( xStatus != pdPASS )
{
/ * 100ms 를기다린후에도보내기작업을완료할수없다. 두가지보내는작업이차단됨상태가되면수신작
업에서대기열에공간을만들어야하므로이오류가발생해야한다. * / vPrintString( "Could not send to the
queue.\r\n" );
}

/ * 다른보낸사람작업을실행하도록허용한다. * / taskYIELD();
}
}
```

#### Listing 38 The implementation of the sending task for Example 11.

수신 작업은 가장 낮은 우선 순위를 가지므로 두 송신 작업이 차단된 상태 일 때만 실행된다. 전송 작업 은 대기열이 가득 차면 차단된 상태로 전환되므로 수신 대기열 작업은 대기열이 이미 가득 차면 실행된 다. 따라서 항상 블록 시간을 지정하지 않아도 데이터를 수신 할 것으로 예상된다.

수신 태스크의 구현은 Listing 39 와 같다.

```

static void vReceiverTask( void *pvParameters )
{
/ * 대기열에서받은값을보유할구조체를선언하시오. * / xData
xReceivedStructure; portBASE_TYPE xStatus;

/ *이작업은또한무한루프내에서정의된다. * /
for( ;; ) {
/ * 가장낮은우선순위를가지기때문에이작업은보내는작업이차단됨상태일때만실행된다. 보내는작업은
대기열이가득차면차단됨상태로전환되므로이작업에서는대기열의항목수가이경우대기열길이 - 3 과항 상같을것으로예상한다.
* /
if( uxQueueMessagesWaiting( xQueue ) != 3 )
{
vPrintString( "Queue should have been
full!\r\n" ); }

/ * 대기열에서수신한다.
```

두번째매개변수는수신된데이터가놓일버퍼이다. 이경우버퍼는단순히수신된구조를유지하는데필요한 크기를갖는변수의주소이다.

마지막매개변수는차단시간 - 데이터가사용가능할때까지대기할작업이 Blocked 상태로유지되는최대시간 대기열이이미비어있는경우이경우대기열이가득찼을때만이작업이실행되므로차단시간은필요하지않다.

```
* /
xStatus = xQueueReceive( xQueue, &xReceivedStructure, 0 );

if( xStatus == pdPASS )
{
    /* 대기열에서데이터를성공적으로수신하여수신된값과값의출처를출력한다. * /
    if( xReceivedStructure.ucSource == mainSENDER_1 )
    {
        vPrintStringAndNumber( "From Sender 1 = ", xReceivedStructure.ucValue );
    }
    else
    {
        vPrintStringAndNumber( "From Sender 2 = ", xReceivedStructure.ucValue );
    }
}
else {
    /* 대기열에서아무것도수신되지않았다. 이작업은대기열이가득찼을때만실행되어야하므로오류여야한다.
    * / vPrintString( "Could not receive from the
    queue.\r\n ); }
}
}
```

#### Listing 39 The definition of the receiving task for Example 11

main ()은 이전 예제에서 약간만 변경된다. 큐는 세 개의 xData 구조를 보유하기 위해 작성되며 송신 및 수신 태스크의 우선 순위가 반대로 된다. main ()의 구현은 Listing 40 과 같다.

```
int main( void )
{
    /* 큐는 xData 유형의구조를최대 3 개보유하기위해작성된다. * /
    xQueue = xQueueCreate( 3, sizeof( xData ) );

    if( xQueue != NULL )
    {
        /* 대기열에של작업의두인스턴스를만든다. 이매개변수는태스크가큐에של구조체를전달하는데사용되므
        로다른태스크가지속적으로 xStructsToSend [1]을보내는동안한태스크가 xStructsToSend [0]을큐에게 속전송한다.
        양자모두
        태스크는수신자의우선순위보다높은우선순위 2 에서작성된다. * /
        xTaskCreate( vSenderTask, "Sender1", 1000, &(amp;xStructsToSend[ 0 ] ), 2, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, &(amp;xStructsToSend[ 1 ] ), 2, NULL );

        /* 대기열에서읽을작업을만든다. 작업은우선순위 1 로생성되므로보낸사람작업의우선순위보다낮다. *
        / xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 1,
        NULL );
    }
}
```

```
/* * 생성된작업이실행되기시작하도록스케줄러를시작하시오. * / vTaskStartScheduler();
}
else
{
/* * 대기열을만들수없다. * /
}

/* * 모든것이잘되면스케줄러가이제태스크를실행할때 main ()이여기에도달하지않는다. main ()이여기 에도달하면,
유휴작업을생성하는데사용할수있는힙메모리가부족하다.
5 장에서는메모리관리에대한자세한정보를제공한다. * /
for( ;; ); }
```

#### Listing 40 The implementation of main() for Example 11

예제 10 에서와 같이 큐에 보내는 태스크는 무한 루프의 반복마다 생성되므로 차례대로 데이터를 대기열 로 보낸다. 예제 11 에 의해 생성 된 출력이 그림 24 에 나와 있다.

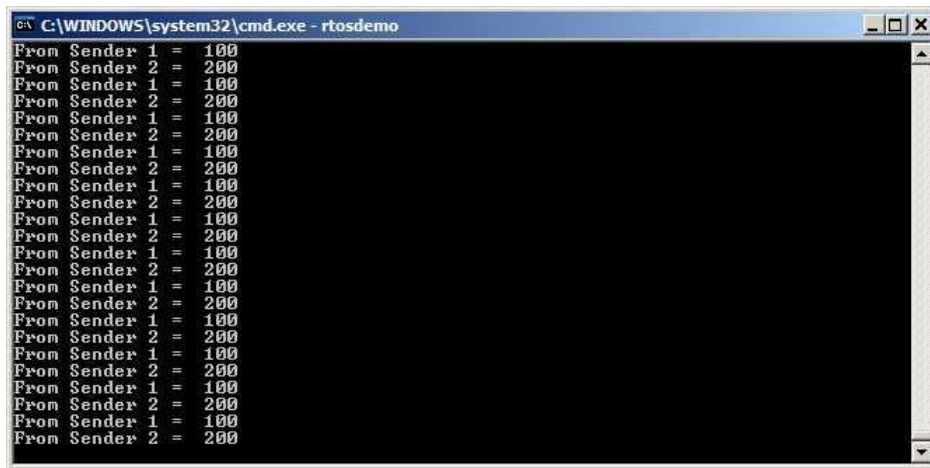


Figure 24 The output produced by Example 11

그림 25 는 수신 태스크보다 전송 태스크의 우선 순위가 높기 때문에 발생하는 실행 순서를 보여준다. 그림 25 에 대한 자세한 설명이 표 12 에 나와 있다.

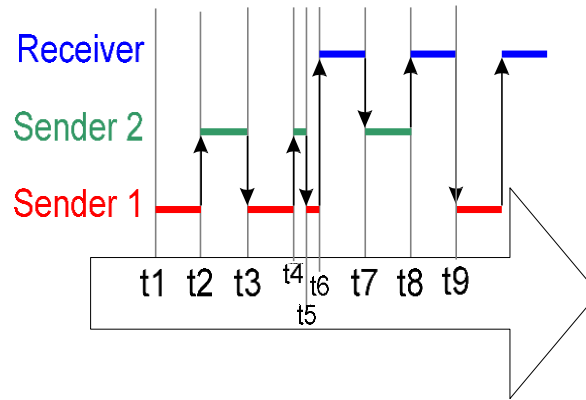


Figure 25 The sequence of execution produced by Example 11

Table 11 Key to Figure 25

Time	Description
t1	작업보낸사람 1 은데이터를실행하여큐에보낸다. t2 보낸 사람 1 은 보낸 사람 2 에게 양보합니다. 보낸 사람 2 는 데이터를 대기열에 쓴다.
t3	보낸 사람 2 는 보낸 사람 1 에게 되돌아옵니다. 보낸 사람 1 은 큐에 데이터를 기록하여 큐를 가득 채 운다.
t4	보낸 사람 1 은 보낸 사람 2 에게 양보한다.
t5	보낸 사람 2 가 대기열에 데이터를 쓰려고 시도한다. 큐가 이미 가득 차 있기 때문에 Sender 2 는 Blocked 상태가 되어 공간을 사용할 수 있게 될 때까지 기다리므로 Sender 1 이 다시 한 번 실행할 수 있 다.
t6	보낸사람 1 이데이터를대기열에쓰려고시도한다. 큐가이미가득차있기때문에 Sender 1 은 Blocked 상태로 전환되어 공간을 사용할 수 있을 때까지 기다린다. 이제 Sender 1 과 Sender 2 가 모두 차단된 상태이므로 낮은 우선 순위의받는 사람 작업을 실행할 수 있다.
t7	수신자 작업은 대기열에서 항목을 제거한다. 대기열에 공간이 있으면 Sender 2 는 Blocked 상태를 유 지하고 우선 순위가 높은 작업으로 Receiver 작업을 선제한다. 보낸 사람 2 는 대기열에 씁니다. 수신자 작업에 의해 생성 된 공간을 채 운다. 대기열이 다시 가득 찼다. 보낸 사람 2 는 taskYIELD ()를 호출하지 만 보낸 사람 1 은 여전히 차단된 상태이므로 보낸 사람 2 는 실행 중 상태 작업으로 다시 선택되고 계속 실행된다.
t8	보낸 사람 2 가 대기열에 쓰려고 시도한다. 큐가 이미 가득 차서 보낸 사람 2 가 차단된 상태가된다. 다 시 한 번 Sender 1 과 Sender 2 가 모두 차단된 상태이므로 수신자 작업을 실행할 수 있다.



t9 수취인 작업은 항목을 대기열에서 제거한다. 대기열에 공간이 있으면 Sender 1 은 Blocked 상태를 유지하고 우선 순위가 높은 작업으로 Receiver 작업을 선점한다. 보낸 사람 1 은 대기열에 쓴다. 수신자 작업에 의해 생성된 공간을 채운다. 대기열이 다시 가득 찼습니다. 보낸 사람 1 은 taskYIELD ()를 호출하지만 보낸 사람 2 는 여전히 차단된 상태이므로 보낸 사람 1 은 실행 중 상태 작업으로 다시 선택되고 계속 실행된다. 보낸 사람 1 이 대기열에 쓰기를 시도하지만 대기열이 가득 차서 보낸 사람 1 이 차단된 상태가 된다.

Sender 1 과 Sender 2 는 다시 Blocked 상태에 있으므로 우선 순위가 낮은 Receiver 작업을 실행할 수 있다.

## 4. WORKING WITH LARGE DATA

대기열에 저장된 데이터의 크기가 큰 경우 데이터 자체를 대기열 안팎으로 바이트 단위로 복사하는 대신 대기열을 사용하여 데이터에 포인터를 전송하는 것이 좋다. 포인터를 전송하는 것은 처리 시간과 대기열을 만드는 데 필요한 RAM 양 모두에서 더 효율적이다. 그러나 대기열에 있는 포인터를 사용할 때 다음 사항을 확인하는 데 극도의 주의를 기울여야 한다.

1. The owner of the RAM being pointed to is clearly defined.  
(가리키고 있는 RAM 의 소유자가 명확하게 정의되어 있다.)

포인터를 통해 작업간에 메모리를 공유할 때 두 작업이 동시에 메모리 내용을 수정하지 않거나 메모리 내용이 유효하지 않거나 일관되지 않을 수 있는 다른 작업을 수행하는 것이 중요하다. 메모리로의 포인터가 대기 행렬에 올 때까지 송신 작업 만이 메모리에 액세스 할 수 있어야 하며, 포인터가 대기열에서 수신된 후에는 수신 태스크 만 메모리에 액세스 할 수 있어야 한다.

2. The RAM being pointed to remains valid  
(가리키고 있는 RAM 이 유효한 상태이다)

가리키는 메모리가 동적으로 할당된 경우 정확하게 하나의 작업이 메모리를 해제된 후에는 작업에 메모리에 액세스하려고 시도하지 않아야 한다.

포인터는 작업 스택에 할당된 데이터에 액세스하는 데 사용해서는 안 된다. 스택 프레임이 변경된 후에는 데이터가 유효하지 않다.

# CHAPTER 3

# INTERRUPT MANAGEMENT

## 3.1 CHAPTER INTRODUCTION AND SCOPE

### Events

임베디드 실시간 시스템은 환경에서 발생하는 이벤트에 대응하여 조치를 취해야 한다. 예를 들어, 이더 넷 주변 장치 (이벤트)에 도착하는 패킷은 처리 (작업)를 위해 TCP / IP 스택으로 전달해야 할 수 있다. 중요하지 않은 시스템은 여러 소스에서 발생한 이벤트를 처리해야 하며, 모든 프로세스 오버 헤드와 응답 시간 요구 사항이 다르다. 각각의 경우 최상의 이벤트 처리 구현 전략에 대한 판단을 내려야 한다.

1. 이벤트가 어떻게 탐지되어야 할까? 인터럽트는 일반적으로 사용되지만 입력도 폴링 할 수 있다.
2. 인터럽트가 사용될 때, 인터럽트 서비스 루틴 (ISR) 내에서 수행되어야 하는 처리량과 외부 처리량은 얼마나 될까? 일반적으로 각 ISR 은 가능한 한 짧게 유지하는 것이 바람직하다.
3. 어떻게 이벤트를 메인 (비 -ISR) 코드로 전달할 수 있으며 잠재적으로 비동기 발생의 처리를 가장 잘 수용 할 수 있도록 이 코드를 구성 할 수 있을까?

FreeRTOS 는 응용 프로그램 설계자에게 특정 이벤트 처리 전략을 부과하지 않지만 선택한 전략을 간단하고 유지 보수가 용이 한 방식으로 구현할 수있는 기능을 제공한다.

'FromISR'또는 'FROM\_ISR'로 끝나는 API 함수 및 매크로 만 인터럽트 서비스 루틴 내에서 사용되어야 한다.

### Scope

이 장은 독자들에게 다음에 대한 좋은 이해를 제공하고자 한다 :

인터럽트 서비스 루틴 내에서 사용할 수있는 FreeRTOS API 함수

자연 인터럽트 방식을 구현하는 방법.

- 바이너리 세마포어 및 카운팅 세마포어를 생성하고 사용하는 방법.
- 바이너리와 카운팅 세마포어의 차이점.
- 큐를 사용하여 인터럽트 서비스 루틴 안팎으로 데이터를 전달하는 방법.

- 일부 FreeRTOS 포트에서 사용할 수 있는 인터럽트 중첩 모델이다.

## 3.2 DEFERRED INTERRUPT PROCESSING

### Binary Semaphores used for Synchronization

A Binary Semaphore 는 특정 인터럽트가 발생할 때마다 작업을 차단 해제하여 효과적으로 작업을 인터럽트와 동기화하는 데 사용할 수 있다. 이를 통해 대부분의 인터럽트 이벤트 처리를 동기화 된 작업 내에서 구현할 수 있으며 매우 빠르고 짧은 부분 만 ISR 에 직접 남아 있다. 인터럽트 처리는 '처리기'작업 에 '지연'되었다고 한다.

인터럽트 처리가 특히 시간이 중요한 경우 처리기 작업 우선 순위를 설정하여 처리기 작업이 항상 시스템의 다른 작업보다 먼저 수행되도록 할 수 있다. 그러면 ISR 자체가 실행을 완료 할 때 ISR 이 반환하는 작업이된다. 전체 이벤트 처리가 ISR 내에서 모두 구현 된 것처럼 시간에 따라 연속적으로 실행되도록 하는 효과가 있다. 이 구성표는 그림 26 에 나와 있다.

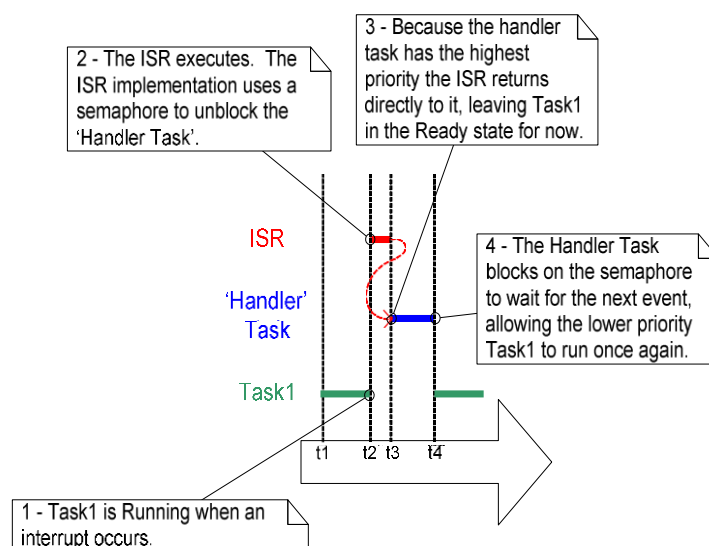


Figure 26 The interrupt interrupts one task, but returns to another.

핸들러 태스크는 이벤트가 발생할 때까지 기다리는 Blocked 상태로 들어가는 수단으로 세마포어에 대한 'take'호출을 사용한다. 이벤트가 발생하면 ISR 은 동일한 이벤트 처리를 진행할 수 있도록 동일한 세마포어에서 '부여'작업을 사용하여 작업을 차단 해제한다.

세마포어를 '받는'과 '주는'은 사용 시나리오에 따라 여러 가지 의미가있는 개념이다. 클래식 세마포어 용어에서 세마포어를 가져 오는 것은 P () 연산과 동일하며 세 마포어 부여는 V () 연산과 같다.

이 인터럽트 동기화 시나리오에서 세마포어는 개념적으로 길이가 1 인 큐로 간주 될 수 있다. 대기열은 언제든지 최대 하나의 항목을 포함할 수 있으므로 항상 비어 있거나 가득 차 있다 (따라서 2 진수). `xSemaphoreTake ()` 를 호출하면 핸들러 태 스 크는 블록 시간과 함께 큐에서 효과적으로 읽기를 시도하여 큐가 비어있는 경우 태 스 크 가 `Blocked` 상 태 가 된 다 . 이 벤 트 가 발 생 하 면 ISR 은 단 순 히 `Microcontrollers;xSemaphoreGiveFromISR ()` 함수를 사용하여 대기열에 토큰 (세마포어)을 배치 <sup>69</sup> 하여 대기열을 가득 채 운다.

이렇게 하면 처리기 작업이 `Blocked` 상태를 벗어나서 토큰을 제거하고 큐를 다시 비워 둔다. 일단 처리 기 작업이 처리를 완료하면 다시 대기열에서 읽으려고 시도하고 빈 대기열을 찾으면 차단 된 상태로 다 시 들어가서 다음 이벤트를 기다린다. 이 시퀀스는 그림 27 에 나와 있다.

그림 27 은 세마포어를 처음 '가져간 것이 아니지만' 세마포어를 '가져 오는' 작업을 하지만 결코 돌려주지 않은 경우에도 세마포어를 '알리는' 인터럽트를 보여준다. 이것이 시나리오가 개념적으로 대기열에 쓰고 대기열에서 읽는 것과 비슷한 개념으로 설명된다. 제 3 장에서 설명한 시나리오처럼 세마포어를 사용하 는 작업이 항상 이를 돌려 주어야 하는 다른 세마포어 사용 시나리오와 동일한 규칙을 따르지 않으므로 종종 혼란을 야기한다.

## **vSemaphoreCreateBinary() API Function**

다양한 유형의 FreeRTOS 세마포어에 대한 핸들은 `xSemaphoreHandle` 유형의 변수에 저장된다.

세마포어를 실제로 사용하려면 먼저 세마포어를 만들어야 한다. 바이너리 세마포어를 생성하려면 `vSemaphoreCreateBinary ()` API 함수를 사용하시오<sup>2</sup>.

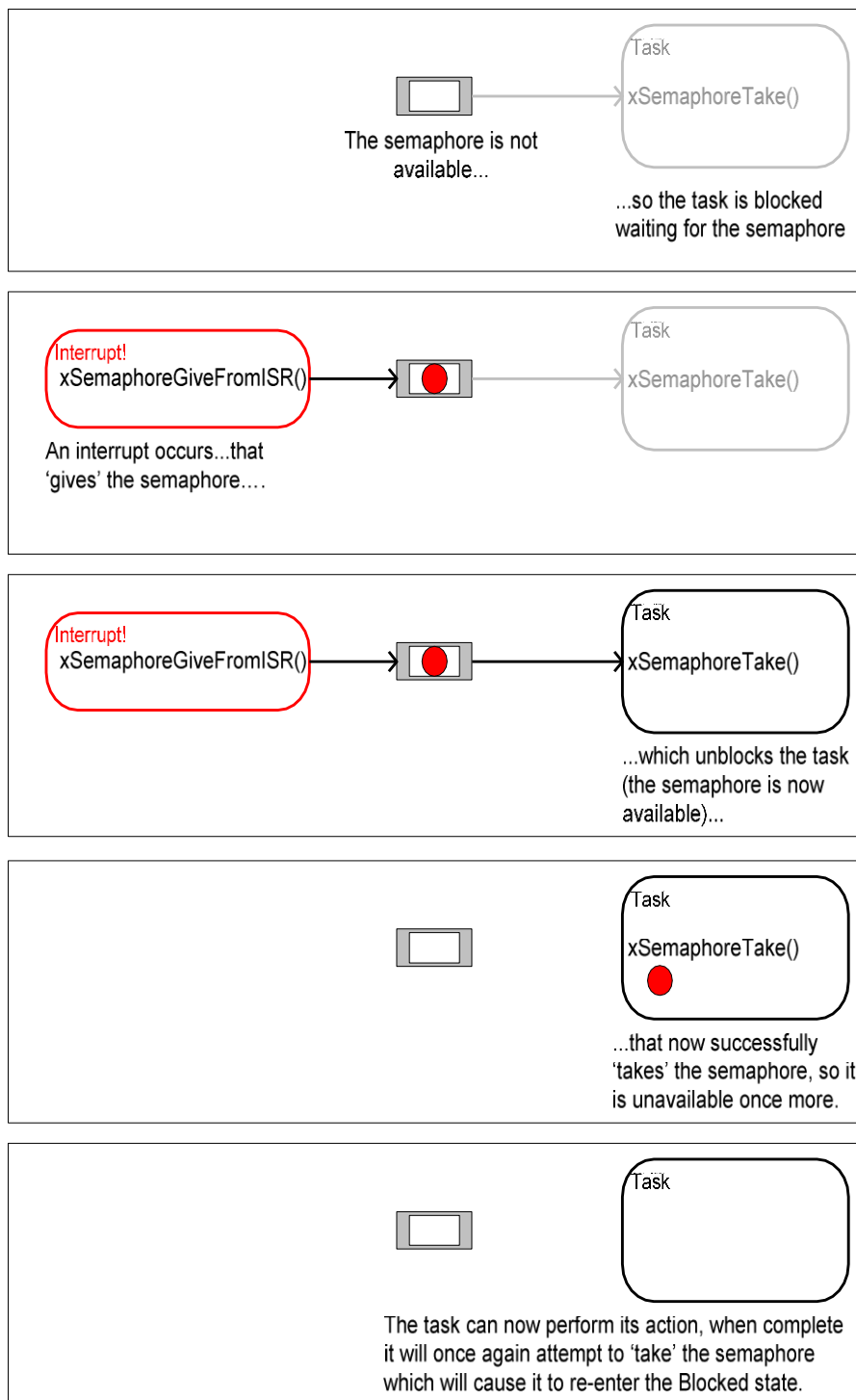
```
void vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore );
```

### **Listing 41 The vSemaphoreCreateBinary() API function prototype**

Table 12 vSemaphoreCreateBinary() parameters

<sup>2</sup> 세마포어 API 는 실제로 함수가 아닌 매크로 집합으로 구현됩니다. 이 책 전체에서 그들은 단순화를 위해 함수로 언급된다.

Parameter Name	Description
xSemaphore	생성되는 세마포어.  vSemaphoreCreateBinary ()는 실제로 매크로로 구현되므로 세마포어 변수는 참조가 아닌 직접 전달되어야 한다. 그만큼 이 장의 예제에는 참조로 사용하고 복사할 수 있는 vSemaphoreCreateBinary ()에 대한 호출이 포함된다.



**Figure 27 Using a binary semaphore to synchronize a task with an interrupt  
xSemaphoreTake() API Function**

세마포어를 '가져 오는 것'은 세마포어를 '얻거나' '받는'것을 의미한다. 세마포어를 사용할 수 있는 경우에만 가져올 수 있다. 클래식 세마포어 용어에서 xSemaphoreTake ()는 P () 연산과 동일하다.

재귀 적 세마포어를 제외한 다양한 유형의 FreeRTOS 세마포어는 xSemaphoreTake () 함수를 사용하여 가져올 수 있다.

xSemaphoreTake ()는 인터럽트 서비스 루틴에서 사용되어서는 안된다.

```
portBASE_TYPE xSemaphoreTake( xSemaphoreHandle xSemaphore, portTickType xTicksToWait );
```

**Listing 42 The xSemaphoreTake() API function prototype**



Table 13 xSemaphoreTake() parameters and return value

Parameter Name / Return Value	Description
Semaphore	세마포어가 '찍혔습니다'.
Wait	<p>세마포어는 xSemaphoreHandle 유형의 변수에 의해 참조되며 사용되기 전에 명시 적으로 작성되어 한다.</p> <p>세마포어가 아직 사용 가능하지 않은 경우 작업이 차단 상태로 유지되어야 하는 최대 시간이다.</p> <p>xTicksToWait 이 0 이면 세마포어를 사용할 수 없는 경우 xSemaphoreTake ()가 즉시 반환된다.</p> <p>블록 시간은 틱 주기로 지정되므로 절대 시간은 틱 주파수에 따라 달라진다. 상수 portTICK_RATE_MS 를 사용하여 밀리 초 단위로 지정된 시간을 틱으로 지정된 시간으로 변환 할 수 있다.</p> <p>xTicksToWait 을 portMAX_DELAY 로 설정하면 FreeRTOSConfig.h 에서 INCLUDE_vTaskSuspend 가 1 로 설정된 경우 태스크가 무기한 대기한다 (시간 초과 없이).</p> <p>가능한 두 가지 반환 값이 있다. :</p> <ol style="list-style-type: none"><li>1. pdPASS<p>pdPASS 는 xSemaphoreTake ()에 대한 호출이 세마포를 획득하는 데 성공한 경우에만 리턴된다.</p><p>블록 시간이 지정 되었으면 (xTicksToWait 이 0 이 아님) 호출 할 수 있는 태스크 가 즉시 사용 가능하지 않은 경우 세마포어를 기다리기 위해 Blocked 상태에 있 지만 세마포어가 만료되기 전에 사용할 수 있게 되었을 가능성이 있다.</p></li><li>2. pdFALSE<p>세마포를 사용할 수 없다.</p><p>블록 시간이 지정 되었으면 (xTicksToWait 이 0 이 아님) 호출 작업은 세마포어 를 사용할 수 있게 되기를 기다리려면 Blocked 상태로 설정되었지만 이 전에는 블록 시간이 만료되었다.</p></li></ol>

## **xSemaphoreGiveFromISR() API Function**

재귀 적 세마포어를 제외한 다양한 유형의 FreeRTOS 세마포어는 xSemaphoreGiveFromISR () 함수를 사용하여 '지정'할 수 있다.

xSemaphoreGiveFromISR ()은 xSemaphoreGive ()의 특수한 형태로 특별히 인터럽트 서비스 루틴 내에서 사용된다.

```
portBASE_TYPE xSemaphoreGiveFromISR( xSemaphoreHandle xSemaphore,  
portBASE_TYPE *pxHigherPriorityTaskWoken );
```

### **Listing 43 The xSemaphoreGiveFromISR() API function prototype**

Table 14 xSemaphoreGiveFromISR() parameters and return value

Parameter Name / Returned Value	Description
xSemaphore	<p>세마포어가 주어진다.‘.</p> <p>세마포어는 xSemaphoreHandle 유형의 변수에 의해 참조되며 사용되기 전에 명시 적으로 작성되어야한다.</p>
pxHigherPriorityTaskWoken	<p>단일 세마포어가 세마포어를 사용할 수있게되기를 기다리면서 차단된 하나 이상의 작업을 가질 수 있다. xSemaphoreGiveFromISR ()을 호출하면 세마포어를 사용할 수 있게 되므로 이러한 작업이 차단된 상태가 된다. xSemaphoreGiveFromISR ()을 호출하면 작업이 Blocked 상태를 벗어나고 차단되지 않은 작업의 우선순위가 현재 실행 중인 작업 (중단 된 작업)보다 높으면 xSemaphoreGiveFromISR ()이 내부적으로 * pxHigherPriorityTaskWoken 을 pdTRUE 로 설정한다.</p> <p>xSemaphoreGiveFromISR ()이 값 을 pdTRUE 로 설정하면 인터럽트가 종료되기 전에 컨텍스트 전환이 수행되어야 한다. 이렇게 하면 인터럽트가 가장 우선 순위가 높은 준비 상태 태스크로 직접 반환된다.</p>
Returnedvalue	<p>가능한 두 가지 반환 값이 있다. :</p> <p>3. pdPASS</p> <p>pdPASS 는 xSemaphoreGiveFromISR ()에 대한 호출이 성공한 경우에만 리턴된다.</p> <p>4. pdFAIL</p> <p>세마포어를 이미 사용할 수 있는 경우 이를 지정할 수 없으며 xSemaphoreGiveFromISR ()은 pdFAIL 을 반환한다.</p> <p><b>Example 12. Using a Binary Semaphore to Synchronize a Task with an Interrupt</b></p> <p>이 예제는 바이너리 세마포어를 사용하여 인터럽트 서비스 루틴 내에서 태스크를 차단 해제하여 효과적으로 태스크를 인터럽트와 동기화한다.</p>

간단한 주기적 작업은 500 밀리 초마다 소프트웨어 인터럽트를 생성하는 데 사용된다. 소프트웨어 인터럽트는 시뮬레이트 된 DOS 환경에서 실제 IRQ 에 연결하는 것이 어렵 기 때문에 편의상 사용된다. Listing 44 는 주기적 태스크의 구현을 보여준다. 인터럽트가 발생하<sub>75</sub>기 전과 후에 문자열을 출력한다.

이는 예제가 실행될 때 생성 된 출력에서 명시적인 실행 순서를 허용하기위한 것이다.

```
static void vPeriodicTask( void *pvParameters )
{
    for( ;; )
    {
        /*이작업은 500ms 마다소프트웨어인터럽트를생성하여인터럽트를 '시뮬레이트'하는데사용된다. */
        vTaskDelay( 500 / portTICK_RATE_MS );

        /* 인터럽트를생성하여전후에메시지를출력하므로예제가실행될때생성된출력에서실행순서가분명하다. */
        vPrintString( "Periodic task - About to generate an interrupt.\r\n" );
        asm( int 0x82 ); /*이줄은인터럽트를생성한다. */
        vPrintString( "Periodic task - Interrupt generated.\r\n\r\n\r\n" );
    }
}
```

**Listing 44 Implementation of the task that periodically generates a software interrupt in Example 12**

Listing 45 는 바이너리 세마포어를 사용하여 소프트웨어 인터럽트와 동기화되는 태스크 인 핸들러 태스 크의 구현을 보여준다. 다시 한 번 메시지가 작업의 각 반복에 인쇄되어 작업 및 인터럽트가 실행되는 순 서가예제가실행될때생성된출력에서분명한다.

```
static void vHandlerTask( void *pvParameters )
{
    /* 대부분의작업에따라이작업은무한루프내에서구현된다. */
    for( ;; ) {
        /* 세마포어를사용하여이벤트를기다린다. 세마포어는이작업이처음실행되기전에스케줄러가시작되기전에 만들어졌다.
        작업은무기한으로차단되므로세마포어를성공적으로가져온경우에만함수호출이반환된다. 따라서
        함수반환값을확인할필요가없다. */
        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );

        /* 여기에도착하려면이벤트가발생했어야한다. 이벤트를처리하십시오. 이경우처리는단순히메시지를인쇄
        하는문제이다. */
        vPrintString( "Handler task - Processing event.\r\n" );
    }
}
```

}

**Listing 45 The implementation of the handler task (the task that synchronizes with the interrupt) in Example 12**

Listing 46 은 실제 인터럽트 핸들러를 보여준다. 이것은 핸들러 태스크를 차단 해제하기 위해 세마포어를 '주지' 않는다. `pxHigherPriorityTaskWoken` 매개 변수가 사용된 방법에 유의하십시오. `xSemaphoreGiveFromISR()`을 호출하기 전에 `pdFALSE` 로 설정되고, `pdTRUE` 와 동일한 것으로 발견되면 컨텍스트 전환이 수행된다.

인터럽트 서비스 루틴 선언의 구문과 컨텍스트 전환을 강제로 호출하는 매크로는 모두 Open Watcom DOS 포트에만 적용되며 다른 포트에서는 다르다. 필요한 실제 구문을 찾기 위해 사용되는 포트에 대한 데모 응용 프로그램에 포함 된 예제를 참조하십시오.

```
static void interrupt far vExampleInterruptHandler( void )
{
    static portBASE_TYPE xHigherPriorityTaskWoken;

    xHigherPriorityTaskWoken = pdFALSE;

    /* 작업을 차단 해제하기 위한 세마포어를 '줘'. */
    xSemaphoreGiveFromISR( xBinarySemaphore, &xHigherPriorityTaskWoken );

    if( xHigherPriorityTaskWoken == pdTRUE )
    {
        /* 세마포어에 작업을 차단 해제하고 차단 해제된 작업의 우선 순위가 현재 실행 중인 작업보다 높다. 인터럽트가 차단되지 않은
        (높은 우선 순위의) 작업으로 직접 반환되도록 강제 전환한다.

참고 : ISR 에서 컨텍스트 전환을 강제하는데 사용할 실제 매크로는 포트에 따라 다르다. 이것은 Open Watcom DOS
포트에 대한 올바른 매크로입니다. 다른 포트에는 다른 구문이 필요할 수 있다.
필요한 구문을 판별하는데 사용된 포트에 제공된 예제를 참조하십시오. */
        portSWITCH_CONTEXT();
    }
}
```

**Listing 46 The software interrupt handler used in Example 12**

`main()` 함수는 바이너리 세마포어와 태스크를 생성하고, 인터럽트 핸들러를 설치하고, 스케줄러를 시작한다. 구현은 Listing 47 과 같다..

```
int main( void )
{
    /* 세마포어를 사용하기 전에 명시적으로 만들어야 한다. 이 예제에서는 바이너리 세마포어가 생성된다. */
    vSemaphoreCreateBinary( xBinarySemaphore );
```

```
/* * 인터럽트핸들러를설치하시오. * */
_dos_setvect( 0x82, vExampleInterruptHandler );

/* * 세마포어가성공적으로생성되었는지확인하시오. * */
if( xBinarySemaphore != NULL )
{
    /* * 'handler'태스크를만든다. 이것은인터럽트와동기화될작업이다. 핸들러태스크는인터럽트가종료된직
    후에실행되도록높은우선순위로작성된다. 이경우 3 의우선순위가선택된다. * */
    xTaskCreate( vHandlerTask, "Handler", 1000, NULL, 3, NULL );

    /* * 주기적으로소프트웨어인터럽트를생성하는작업을만든다. 이것은핸들러태스크아래에서우선순위로작성
    되어핸들러태스크가 Blocked 상태를벗어날때마다선제공격을받을수있도록한다. * */
    xTaskCreate( vPeriodicTask, "Periodic", 1000, NULL, 1, NULL );

    /* * 생성된작업이실행되기시작하도록스케줄러를시작하시오. * */
    vTaskStartScheduler(); }

/* * 모든것이잘되면스케줄러가이제태스크를실행할때 main () 이여기에도달하지않는다. main () 이여기
에도달하면유휴작업을생성하는데사용할수있는힙메모리가부족할수있다.
5 장에서는메모리관리에대한자세한정보를제공한다. * */
for( ;; ); }
```

#### Listing 47 The implementation of main() for Example 12

예제 12 는 그림 28 에 나와있는 출력을 생성한다. 예상대로 핸들러 태스크는 인터럽트가 생성 된 즉시 실행되므로 핸들러 태스크의 출력은 주기 태스크가 생성 한 출력을 분할합니다. 자세한 설명은 그림 29 에 나와 있다.

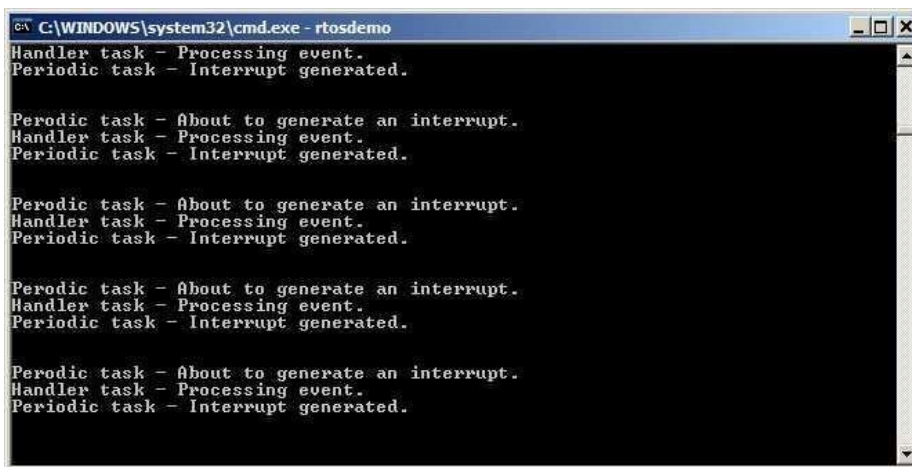


Figure 28 The output produced when Example 12 is executed

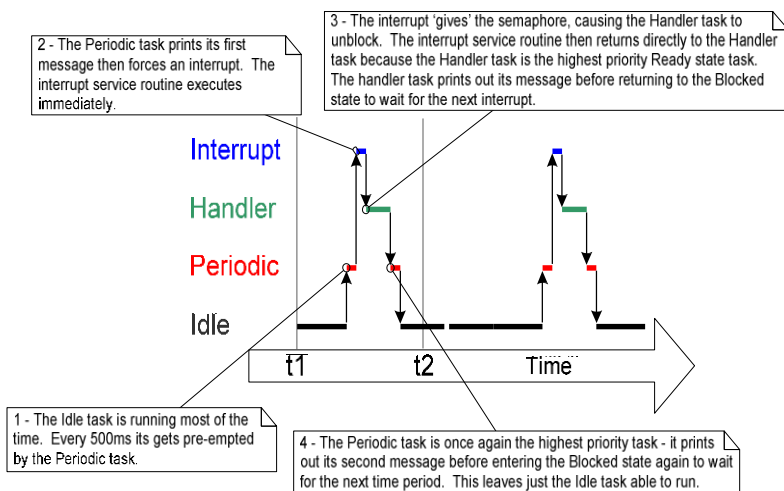


Figure 29 The sequence of execution when Example 12 is executed

### 3.3 COUNTING SEMAPHORES

예제 12에서는 태스크를 인터럽트와 동기화하는 데 사용되는 바이너리 세마포어를 보여 주었습니다. 실행 순서는 다음과 같다.

1. 인터럽트가 발생했다.
2. 인터럽트 서비스 루틴이 실행되어 '처리' 작업을 차단 해제하기 위해 세마포를 부여한다'.
3. 핸들러태스크는 인터럽트가 완료되자마자 실행된다. 처리작업이 수행한 첫 번째 작업은 세마포어를 '가져 오는' 작업이었다.
4. Handler 태스크는 세마포어를 다시 '가져 오기' 전에 이벤트 처리를 수행했다.
5. 세마포어를 즉시 사용할 수 없는 경우 차단된 상태로 전환한다.

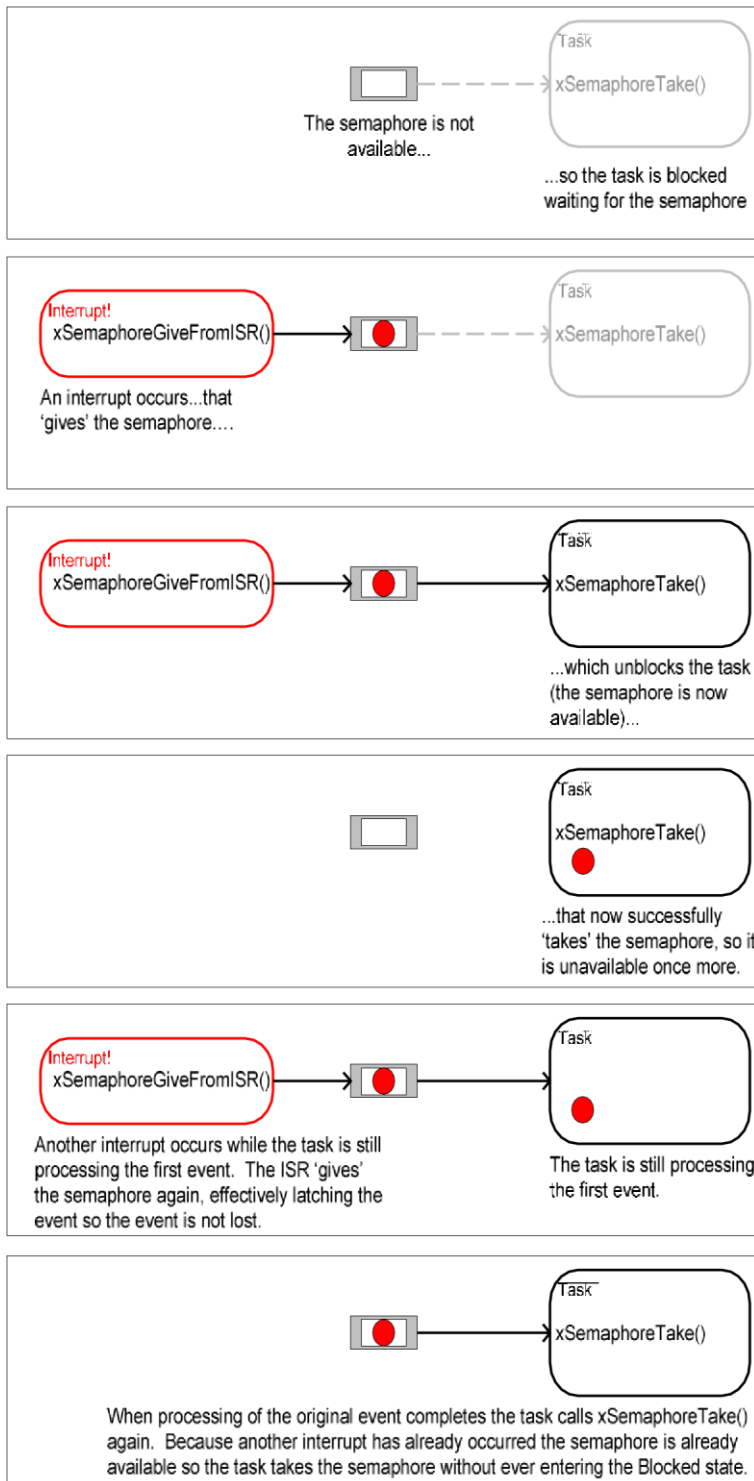
인터럽트가 상대적으로 낮은 빈도에서만 발생할 수 있는 경우 이 시퀀스가 완벽하게 적합하다. 처리기 태스크가 첫 번째 처리를 완료하기 전에 다른 인터럽트가 발생하면 2진 세마포어가 효과적으로 이벤트를 래치하여 처리기 태스크가 원래 이벤트 처리를 완료한 직후에 새 이벤트를 처리할 수 있게 한다. 핸들러 태스크는 `xSemaphoreTake()`가 호출될 때 즉시 래치된 세마포어를 사용할 수 있으므로 두 이벤트를 처리하는 사이 Blocked 상태로 들어 가지 않는다. 이 시나리오는 그림 30과 같다.

그림 30은 바이너리 세마포어가 최대 하나의 인터럽트 이벤트를 래치할 수 있음을 보여준다. 래치된 이벤트가 처리되기 전에 발생하는 모든 후속 이벤트는 손실된다. 이 시나리오는 바이너리 세마포어 대신 카운팅 세마포어를 사용하여 피할 수 있다.

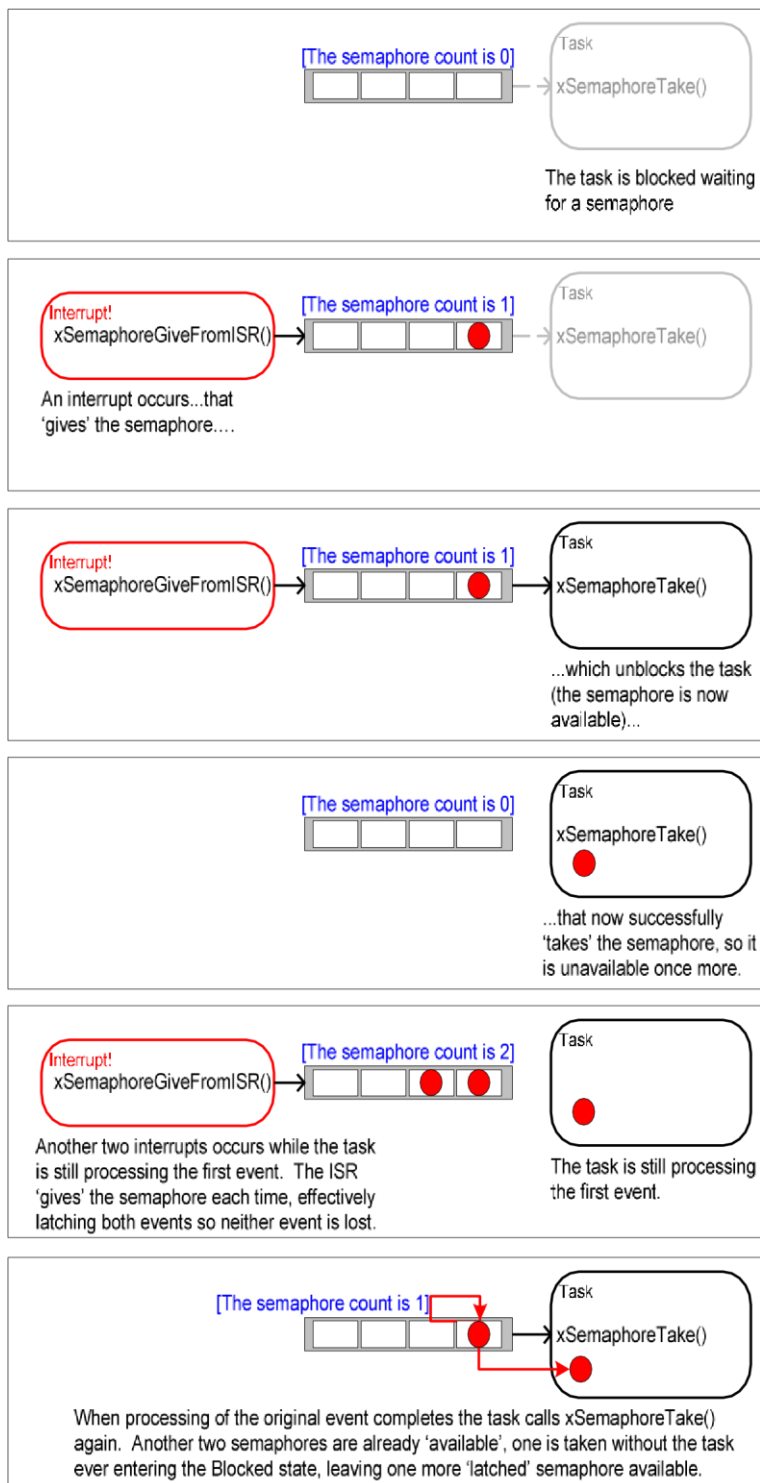
바이너리 세마포어는 개념적으로 길이가 1 인 큐로 간주 될 수 있기 때문에 카운팅 세마포는 하나 이상의 길이를 갖는 큐로 간주 될 수 있다. 작업은 대기열에 저장된 데이터에 관심이 없습니다. 단지 대기열이 비어 있는지 여부이다.

카운팅 세마포어가 '지정'될 때마다 대기열의 다른 공간이 사용된다. 큐에 있는 항목의 수는 세마포어 'count'값이다.





**Figure 30 A binary semaphore can latch at most one event**



**Figure 31 Using a counting semaphore to 'count' events**

카운팅 세마포어는 일반적으로 두 가지 용도로 사용된다.

## 1.Counting events.

FreeRTOS

Designed For Microcontrollers;

© 2009 Richard Barry. Distribution or publication in any form is strictly prohibited.

이 사용 시나리오에서 이벤트 핸들러는 이벤트가 발생할 때마다 세마포어를 '주겠다'- 세마포어 카운트 값을 각각 증가시 키도록한다. 핸들러 태스크는 이벤트를 처리 할 때마다 세마포어를 '취합니다'- 각 테 이크에서세마포어카운트값을감소시킨다. 카운트값은발생한이벤트수와처리된숫자의차이입니 다. 이 메커니즘은 그림 31 에 나와 있다. 이벤트 카운트에 사용되는 카운팅 세마포어는 초기 카운트 값이 0 으로 생성된다.

## 2.Resource management.

이 사용 시나리오에서 카운트 값은 사용 가능한 리소스의 수를 나타낸다. 자원의 제어권을 얻으려면 먼 저 세마포어를 얻어야 한다. 세마포어 카운트 값을 감소시킨다. 카운트 값이 0 에 도달하면 사용 가능한 자원이 없다. 자원으로 작업이 끝나면 세마포어를 다시 가져옵니다 - 세마포어 카운트 값을 증가시킨다.

리소스를 관리하는 데 사용되는 카운팅 세마포가 만들어져 초기 카운트 값이 사용 가능한 리소스 수와 같아진다. 4 장에서는 리소스를 관리하기 위해 세마포어를 사용하는 방법을 다룬다.

### xSemaphoreCreateCounting() API Function

다양한 유형의 FreeRTOS 세마포어에 대한 핸들은 xSemaphoreHandle 유형의 변수에 저장된다.

세마포어를 실제로 사용하려면 먼저 세마포어를 만들어야한다. 카운팅 세마포를 생성하려면 xSemaphoreCreateCounting () API 함수를 사용하시오.

```
xSemaphoreHandle xSemaphoreCreateCounting( unsigned portBASE_TYPE uxMaxCount, unsigned
                                           portBASE_TYPE uxInitialCount );
```

#### Listing 48 The xSemaphoreCreateCounting() API function prototype

Table 15 xSemaphoreCreateCounting() parameters and return value

Parameter Name / ReturnedValue	Description
--------------------------------	-------------

uxMaxCount	<p>세마포어의 최대 값. 대기열 유추를 계속하려면 uxMaxCount 값이 효과적으로 대기열의 길이이다.</p> <p>세마포어를 사용하여 이벤트를 카운트하거나 래치 할 때 uxMaxCount 는 래치 될 수 있는 최대 이벤트 수이다.</p> <p>리소스 컬렉션에 대한 액세스를 관리하기 위해 세마포를 사용할 경우 uxMaxCount 를 사용 가능한 총 리소스 수로 설정해야 한다.</p>
uxInitialCount	<p>생성 된 후의 세마포어의 초기 카운트 값.</p> <p>세마포어를 사용하여 이벤트를 카운트하거나 래치 할 때 uxInitialCount 를 0 으로 설정해야 한다. 세마포가 생성 될 때 이벤트가 아직 발생하지 않았기 때문일 수 있다.</p> <p>세마포어가 리소스 컬렉션에 대한 액세스를 관리하는 데 사용될 때 uxMaxCount 와 함께 uxInitialCount 를 설정해야 한다. 세마포어를 만들 때 모든 리소스를 사용 할 수 있는 것으로 추정된다.</p>
Returnedvalue	<p>NULL 이 반환되면 FreeRTOS 가 세마포어 데이터 구조를 할당하기에 부족한 힙 메모리가있기때문에세마포를만들수없다. 5 장은메모리관리에대한더많은 정보를 제공한다..</p> <p>반환되는 NULL 이 아닌 값은 세마포가 성공적으로 생성되었음을 나타낸다. 반환 된 값은 생성 된 세마포어에 대한 핸들로 저장되어야 한다.</p>

### **Example 13. Using a Counting Semaphore to Synchronize a Task with an Interrupt**

#### **(카운팅세마포를사용하여태스크를인터럽트와동기화)**

예제 13 은 바이너리 세마포어 대신 카운팅 세마포어를 사용하여 예제 12 구현을 향상 시 킨 다 . main () 이 vSemaphoreCreateBinary () 에 대 한 호 출 대 신 xSemaphoreCreateCounting ()에 대한 호출을 포함하도록 변경되었다. 새로운 API 호출은 Listing 49 와 같다.

```
/ * 세마포어를사용하기전에명시적으로만들어야한다. 이예에서는카운팅세마포가생성됩니다.  
세마포어는최대카운트값이 10 이고초기카운트값이 0 이되도록생성된다. * /
```

```
xCountingSemaphore = xSemaphoreCreateCounting( 10, 0 );
```

**Listing 49 Using xSemaphoreCreateCounting() to create a counting semaphore**

높은 빈도로 발생하는 여러 이벤트를 시뮬레이트하기 위해 인터럽트 서비스 루틴이 인터럽트 당 두 번 이상 세마포어를 '제공'하도록 변경된다. 각 이벤트는 세마포어 카운트 값으로 래치된다. 수정 된 인터럽트 서비스 루틴은 Listing 50 과 같다.

```
static void interrupt far vExampleInterruptHandler( void )
{
    static portBASE_TYPE xHigherPriorityTaskWoken;

    xHigherPriorityTaskWoken = pdFALSE;

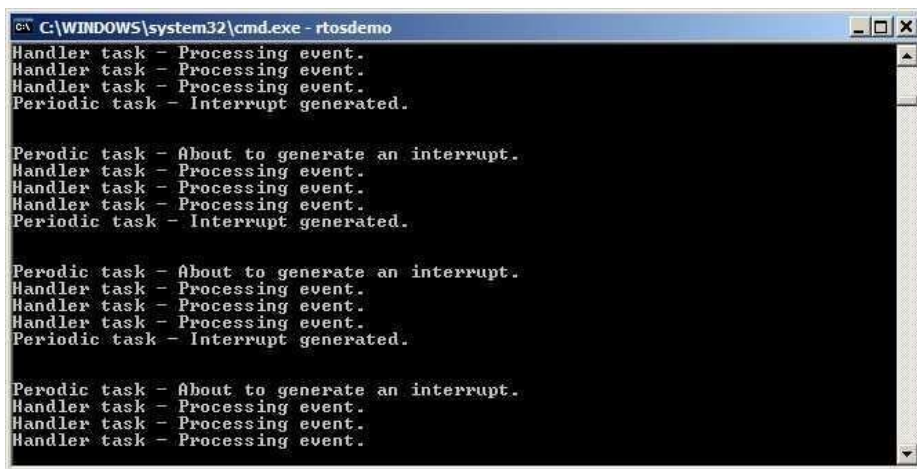
    / * 세마포어를여러번 '줘'. 첫번째는핸들러태스크를차단해제한다. 다음은 '세마포어'가이벤트를래치하
    여행들러태스크가이벤트를읽지않고순서대로처리할수있음을보여준다. 이경우프로세서가단일인터럽트
    발생내에서시뮬레이션된경우에도다중인터럽트를시뮬레이트한다. * /
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );

    if( xHigherPriorityTaskWoken == pdTRUE )
    {
        / * 세마포어에작업을차단해제하고차단해제된작업의우선순위가현재실행중인작업보다높다. 인터럽트가 차단되지않은
        (높은우선순위의) 작업으로직접반환되도록강제전환한다.
        참고 : ISR 에서컨텍스트전환을강제하는데사용할실제매크로는포트에따라다르다. 이것은 Open Watcom DOS
        포트에대한올바른매크로입이다. 다른포트에는다른구문이필요할수있다.
        필요한구문을판별하는데사용된포트에제공된예제를참조하십시오. * / portSWITCH_CONTEXT();
    }
}
```

#### Listing 50 The implementation of the interrupt service routine used by Example 13

다른 모든 기능은 예 12 에서 사용 된 기능에서 수정되지 않은 상태로 유지된다.

예제 13 이 실행될 때 생성 된 출력이 그림 32 에 나와 있다. 알 수 있듯이, Handler 태스크는 인터럽트가 생성 될 때마다 세 개의 [시뮬레이션] 이벤트를 모두 처리한다. 이벤트는 세마포어의 카운트 값에 래치되 어 핸들러 태스크가 세마포어를 순서대로 처리 할 수 있게 한다.



```
C:\WINDOWS\system32\cmd.exe - rtosdemo
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
```

Figure 32 The output produced when Example 13 is executed

## 3.4 USING QUEUES WITHIN AN INTERRUPT SERVICE ROUTINE

`xQueueSendToFrontISM ()`, `xQueueSendToBackFromISR ()` 및 `xQueueReceiveFromISR ()`은 각각 인터럽트 서비스 루틴 내에서 사용하기에 안전한 `xQueueSendToFront ()`, `xQueueSendToBack ()` 및 `xQueueReceive ()`의 버전입니다.

세마포어는 이벤트를 전달하는 데 사용된다. 대기열은 이벤트를 전달하고 데이터를 전송하는 데 사용된다.

### **`xQueueSendToFrontFromISR()` and `xQueueSendToBackFromISR()` API Functions**

`xQueueSendFromISR ()`은 `xQueueSendToBackFromISR ()`과 동일하고 완전히 동일하다.

```
portBASE_TYPE xQueueSendToFrontFromISR( xQueueHandle xQueue,  
void *pvItemToQueue  
portBASE_TYPE *pxHigherPriorityTaskWoken  
);
```

**Listing 51 The `xQueueSendToFrontFromISR()` API function prototype**

```
portBASE_TYPE xQueueSendToBackFromISR( xQueueHandle xQueue,  
void *pvItemToQueue  
portBASE_TYPE *pxHigherPriorityTaskWoken  
);
```

**Listing 52 The `xQueueSendToBackFromISR()` API function prototype**

**Table 16 `xQueueSendToFrontFromISR()` and `xQueueSendToBackFromISR()` parameters and return values**

Parameter Name / Returned Value	Description
<code>xQueue</code>	데이터가 보내지는 (쓰여지는) 큐의 핸들. 큐 핸들은 큐를 생성하는 데 사용된 <code>xQueueCreate ()</code> 에 대한 호출에서 반환된다.
<code>pvItemToQueue</code>	대기열에 복사 될 데이터에 대한 포인터. 대기열이 생성 할 수있는 각 항목의 크기는 대기열이 생성 될 때 설정되

므로 이 많은 바이트는 `pvlItemToQueue` 에서 대기열 저장소 영역으로 복 사된다.

`pxHigherPriorityTaskWoken`

단일 대기열에 하나 이상의 작업이 차단 될 수 있다 데이터를 사용할 수 있을 때까지 기다린다.

`xQueueSendToFrontFromISR ()` 또는 `xQueueSendToBackFromISR ()` 을 호출하면 데이터를 사용할 수있게되어 그러한 작업이 `Blocked` 상태를 벗어날 수 있다. API 함수를 호출하면 작업이 `Blocked` 상태를 벗어나고 차단되지 않은 작업의 우선 순위가 현재 실행중인 작업 (중단 된 작업)

보다높으면 API 함수는내부적으로 \* `pxHigherPriorityTaskWoken` 을 `pdTRUE` 로 설정한다.

`xQueueSendToFrontFromISR ()` 또는 `xQueueSendToBackFromISR ()` 이 값을 `pdTRUE` 로 설정하면 인터럽트가 종료되기 전에 컨텍스트 전환 이 수행되어야 한다. 이렇게 하면 인터럽트가 최상위 우선 순위의 준비 상태 태스크로 직접 반환된다.

`Returnedvalue`

가능한 두 가지 반환 값이 있다. :

1.`pdPASS`

`pdPASS` 는 데이터가 큐에 성공적으로 전송 된 경우에만 리턴된다.

2.`errQUEUE_FULL`

대기열이 이미 가득 차 있기 때문에 데이터를 대기열로 보낼 수 없는 경우 `errQUEUE_FULL` 이 반환된다.

### Efficient Queue Usage

**FreeRTOS** 다운로드에포함된대부분의데모응용프로그램에는대기열을 사용하여 문자를 전송 인터럽트 처리기로 전달하고 수신 인터럽트 처리기에서 문자를 전달하는 간단한 UART 드라이버가 포함되어 있다. 전송되거나 수신되는 모든 문자는 개별적으로 대기열을 통과한다.



UART 드라이버는 인터럽트 내에서 사용되는 대기열을 보여주는 편리한 방법으로 순전히 이 방식으로 구현된다. 대기열을 통해 개별 문자를 전달하는 것은 매우 비효율적이므로 (특히 높은 보드 속도에서) 프 로덕션 코드에 권장되지 않는다. 보다 효율적인 기술은 다음과 같다.

수신 된 각 문자를 간단한 RAM 버퍼에 넣은 다음, 완료 메시지가 수신 된 후에 버퍼를 처리하기 위해 세마포어를 사용하여 차단을 해제하거나 전송 중단이 감지되었다.

인터럽트 서비스 루틴 내에서 수신 된 문자를 직접 해석 한 다음 대기열을 사용하여 인터럽트 및 디코딩 된 명령을 처리를 위해 작업에 보낸다 (그림 23 과 유사한 방식으로). 이 기술은 데이터 스트림 해석이 인터럽트 내에서 전체적으로 수행되기에 충분히 빠르면 적합하다.

#### **Example 14. Sending and Receiving on a Queue from Within an Interrupt**

이 예 제 는 동 일 한 인 터 럽 트 내 에 서 사 용 되 는 `xQueueSendToBackFromISR ()` 및 `xQueueReceiveFromISR ()`을 보여준다. 이전과 마찬가지로 소프트웨어 인터럽트가 편의상 사용된다.

200 밀리 초마다 대기열에 5 개의 숫자를 보내는 주기적 작업이 생성된다. 다섯 개의 값이 모두 전송 된 후에 만 소프트웨어 인터럽트를 생성한다. 태스크 구현은 Listing 53 과 같다.

```
static void vIntegerGenerator( void *pvParameters )
{
    portTickType xLastExecutionTime;  unsigned portLONG ulValueToSend = 0;  int i;

    /* vTaskDelayUntil () 호출에사용된변수를초기화한다. * /
    xLastExecutionTime = xTaskGetTickCount();

    for( ;; )
    {
        /* 이것은정기적인작업이다. 다시실행할시간이될때까지차단하시오. 작업은매
        200ms 마다실행된다. * /
        vTaskDelayUntil( &xLastExecutionTime, 200 / portTICK_RATE_MS );

        /* 증가하는숫자를큐에다섯번보낸다. 값은인터럽트서비스루틴에의해큐에서읽혀진다. 인터럽트서비스
        루틴은항상대기열을비우기때문에이작업은 5 개의값을모두쓸수있으므로블록시간이필요하지않다. *
        for( i = 0; i < 5;
        i++ )
        {
            xQueueSendToBack( xIntegerQueue, &ulValueToSend, 0 );
            ulValueToSend++; }

        /* 인터럽트서비스루틴이큐에서값을읽을수있도록인터럽트를강제실행한다. * /
        vPrintString( "Generator task - About to generate an interrupt.\r\n" ); __
        asm( int 0x82 ) / *이줄은인터럽트를생성한다. * /
        vPrintString( "Generator task - Interrupt generated.\r\n\r\n\r\n" );
    }
}
```

#### Listing 53 The implementation of the task that writes to the queue in Example 14

인터럽트 서비스 루틴은 주기적 태스크에 의해 큐에 기록 된 모든 값이 제거되고 큐가 비어있을 때까지 xQueueReceiveFromISR ()을 반복적으로 호출한다. 각 수신 된 값의 마지막 두 비트는 문자열 배열에 대한 인덱스로 사용되며 xQueueSendFromISR ()에 대한 호출을 사용하여 해당 인덱스 위치에있는 문자열에 대한 포인터가 다른 대기열로 전송된다. 인터럽트 서비스 루틴의 구현은 Listing 54 와 같다.

```
static void interrupt far vExampleInterruptHandler( void )
{
    static portBASE_TYPE xHigherPriorityTaskWoken;  static unsigned long ulReceivedNumber;

    /* 문자열은정적 const 로선언되어 ISR 스택에할당되지않고 ISR 이실행되지않을때에도존재한다. * / static const
    char *pcStrings[] =
    {
        "String 0\r\n", "String 1\r\n", "String 2\r\n", "String 3\r\n"
    }; xHigherPriorityTaskWoken =

    pdFALSE;
```

```

/ * 큐가비게될때까지반복하시오. * /
while( xQueueReceiveFromISR( xIntegerQueue,
                             &ulReceivedNumber,
                             &xHigherPriorityTaskWoken ) != errQUEUE_EMPTY )
{
/ * 수신된값을마지막두비트 (0 ~ 3 inc 값) 로절단한다음잘린값에해당하는문자열에대한포인터를 다른대기열로보낸다.
* /
ulReceivedNumber &= 0x03;
xQueueSendToBackFromISR( xStringQueue,
                          &pcStrings[ ulReceivedNumber ],
                          &xHigherPriorityTaskWoken ); }

/ * 대기열에서수신중이거나대기열에서보내는것은현재실행중인작업보다우선순위가높은작업의차단을
해제할까? 그렇다면여기서컨텍스트를강제로전환하시오. * /
if( xHigherPriorityTaskWoken == pdTRUE )
{
/ * 참고 : ISR 에서컨텍스트스위치를강제실행하는데사용하는실제매크로는포트에따라다르다. 이것은 Open Watcom
DOS 포트에대한올바른매크로이다. 다른포트에는다른구문이필요할수있다. 필요한구문을판
별하는데사용된포트에제공된예제를참조하시오. * / portSWITCH_CONTEXT();
}
}
```

**Listing 54** The implementation of the interrupt service routine used by Example 14

인터럽트 서비스 루틴에서 문자 포인터를 받는 작업은 메시지가 도착할 때까지 대기열을 차단하여 각 문자열을 받은대로 인쇄한다. 구현은 Listing 55 와 같다.

```

static void vStringPrinter( void *pvParameters )
{
char *pcString;

for( ;; )
{
/ * 데이터가도착할때까지대기할큐를차단하시오. * /
xQueueReceive( xStringQueue, &pcString, portMAX_DELAY );

/ *받은문자열을출력한다. * /
vPrintString( pcString );
}
}
```

**Listing 55** The task that prints out the strings received from the interrupt service routine in Example 14

정상적으로 main ()은 스케줄러를 시작하기 전에 필요한 대기열과 작업을 생성한다. 구현은 Listing 56 과 같다.

```
int main( void ) {
    /* 대기열을사용하려면먼저대기열을작성해야한다. 이에제에서사용된두대기열을모두작성
    하시오. 하나의큐는 unsigned long 타입의변수를가질수있고, 다른큐는 char * 타입의변수를
    가질수있다. 두대기열모두최대 10 개의항목을보유할수있다. 실제응용프로그램은대기열
    이성공적으로작성되었는지확인하기위해리턴값을점검해야한다. */ xIntegerQueue =
    xQueueCreate( 10, sizeof( unsigned long ) ); xStringQueue = xQueueCreate( 10,
    sizeof( char * ) );

    /* 인터럽트핸들러를설치하십시오. */
    _dos_setvect( 0x82, vExampleInterruptHandler );

    /* 대기열을사용하여인터럽트서비스루틴에정수를전달하는작업을만든다. 작업은우선순위 1 에서생성된다. */
    xTaskCreate( vIntegerGenerator, "IntGen", 1000, NULL, 1, NULL );

    /* 인터럽트서비스루틴에서전송된문자열을출력하는작업을만든다. 이작업은 2 의높은우선순위 에서생성된다. */
    xTaskCreate( vStringPrinter, "String", 1000, NULL, 2, NULL );

    /* 생성된작업이실행되기시작하도록스케줄러를시작하십시오. */
    vTaskStartScheduler();

    /* 모든것이잘되면스케줄러가이제태스크를실행할때 main () 이어기에도달하지않는다. main
    () 이어기에도달하면, 유틸작업을생성하는데사용할수있는힙메모리가부족하다.
    5 장에서는메모리관리에대한자세한정보를제공한다 */
    for( ;; ); }
```

#### Listing 56 The main() function for Example 14

알 수 있듯이 인터럽트는 5 개의 정수를 모두 받고 5 개의 문자열을 응답으로 생성한다. 더 많은 설명이 그림 34 에 나와 있다.

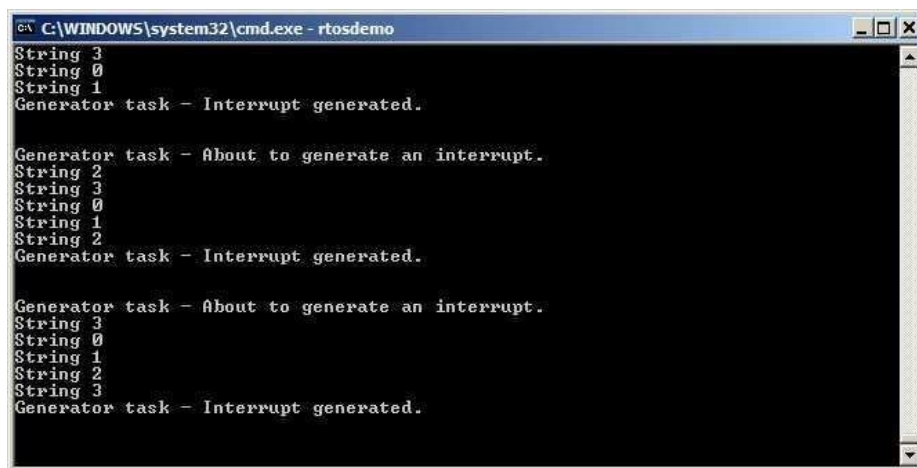


Figure 33 The output produced when Example 14 is executed

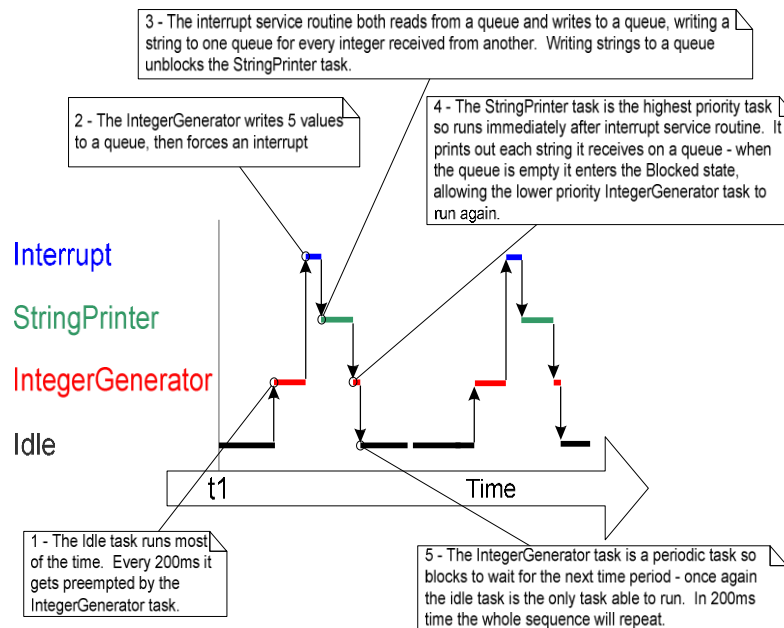


Figure 34 The sequence of execution produced by Example 14

## 3.5 INTERRUPT NESTING

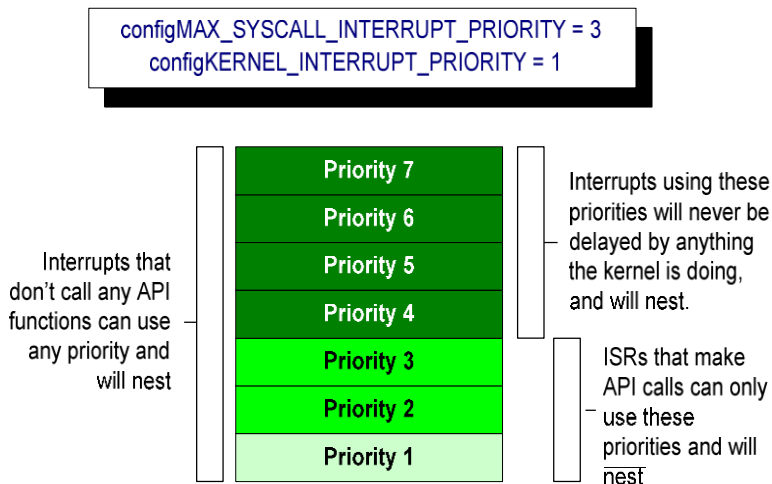
가장 최근의 FreeRTOS 포트는 인터럽트가 중첩되도록 한다. 이 포트는 표 17에 설명된 상수 중 하나 또는 둘 모두가 FreeRTOSConfig.h 내에 정의되어야 한다.

Table 17 Constants that control interrupt nesting

Constant	Description
configKERNEL_INTERRUPT_PRIORITY	틱 인터럽트가 사용하는 인터럽트 우선 순위를 설정한다.  포트가 configMAX_SYSCALL_INTERRUPT_PRIORITY 상수를 사용하지 않으면 인터럽트 안전 FreeRTOS API 함수를 사용하는 모든 인터럽트도 이 우선 순위로 실행되어야 한다.
configMAX_SYSCALL_INTERRUPT_PRIORITY	인터럽트 안전 FreeRTOS API 함수를 호출 할 수 있는 가장 높은 인터럽트 우선 순위를 설정 한다.

전 체 인 터 럽 트 중 첩 모 델 은 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 를 configKERNEL\_INTERRUPT\_PRIORITY 보 다 높 은 우 선 순 위 로 설 정 하 여 생 성 된 다 . 이 는 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 가 3 으 로 설 정 되 고 configKERNEL\_INTERRUPT\_PRIORITY 가 1 로 설정된 가설 시나리오를 보여주는 그림 35 에서 설명된 다. 7 개 의 다른 인터럽트 우선 순위 레벨을 가진 동일하게 가상의 마이크로 컨트롤러가 표시된다. 값 7 은 이 가상의 예에 대한 임의의 숫자이며 특정 마이크로 컨트롤러 아키텍처를 대표하지는 않는다.

작업 우선 순위와 우선 순위 인터럽트 사이에 혼동이 생기는 경우가 일반적이다. 그림 35 는 마이크로 컨 트롤러 아키텍처에 정의 된 인터럽트 우선 순위를 보여준다. 인터럽트 서비스 루틴이 서로 상대적으로 실행되는 하드웨어 제어 우선 순위이다. 태스크는 인터럽트 서비스 루틴에서 실행되지 않으므로 태스크 에 할당 된 소프트웨어 우선 순위는 인터럽트 소스에 할당 된 하드웨어 우선 순위와 아무런 관련이 없다.



**Figure 35 Constants affecting interrupt nesting behavior** Referring to Figure 35:

우선 순위 1 에서 3 까지의 우선 순위를 사용하는 인터럽트는 커널 또는 응용 프로그램이 중요한 섹션 내에 있는 동안 실행되지 못하지만 인터럽트 안전 FreeRTOS API 함수를 사용할 수 있다.

우선 순위 4 이상을 사용하는 인터럽트는 임계 영역의 영향을 받지 않으므로 커널이 수행하는 인터럽트가 마이크로 컨트롤러 자체의 제한 내에서 즉시 실행되는 것을 방해하지 않는다. 일반적으로 매우 엄격한 타이밍 정확도 (예 : 모터 제어)가 필요한 기능은 스케줄러가 인터럽트 응답 시간에 지터를 도입 하지 않도록 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 보다 높은 우선 순위를 사용한다.

FreeRTOS API 호출을 하지 않는 인터럽트는 임의의 우선 순위를 자유롭게 사용할수있다.

## A Note to ARM Cortex M3 Users

Cortex M3 은 논리적으로 우선 순위가 높은 인터럽트를 나타 내기 위해 숫자 가 낮은 우선 순위 번호를 사용한다. 이것은 반 직관적으로 보일 수 있으며 잊 기 쉽습니다! 인터럽트를 낮은 우선 순위 에 할당하려면 높은 숫자 값을 할당 해야 한다. 우선 순위를 0 (또는 다른 낮은 숫자 값)으로 지정하지 마십시오. 실제로 인터럽트가 시스템에서 가장 높은 우선 순위를 가질 수 있으므로 우선 순위가 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 보다 높으면 잠재적 으로 시스템이 중단 될 수 있다.

Cortex M3 코어의 최하위 우선 순위는 255 이지만 다른 Cortex M3 공급 업체 는다른우선순위비트수를구현하고우선순위를다른방식으로지정할수 있는 라이브러리 기능을 제공한다. 예를 들어, STM32 에서 ST 드라이버 라이 브러리 호출에서 지정할 수 있는 가장 낮은 우선 순위는 15 이고 지정할 수있 는 가장 높은 우선 순위는 0 이다.

# CHAPTER 4

# RESOURCE MANAGEMENT

## 4.1 CHAPTER INTRODUCTION AND SCOPE

하나의 작업이 자원에 액세스하기 시작하지만 실행 상태에서 벗어나기 전에 액세스를 완료하지 않으면 멀티 태스킹 시스템에서 충돌이 발생할 수 있다. 작업으로 인해 리소스가 일관성 없는 상태로 남아 있으면 다른 작업이나 인터럽트로 동일한 리소스에 액세스하면 데이터가 손상되거나 다른 유사한 오류가 발생할 수 있다. 다음은 몇 가지 예이다.

### 1.Accessing Peripherals(주변장치액세스)

두 작업이 LCD 에 쓰기를 시도하는 다음 시나리오를 고려하시오. :

- 작업 A 가 실행되고 문자열 "Hello world"를 LCD 에 작성하기 시작한다.
- 작업 A 는 문자열 "Hello w"의 시작 부분 만 출력 한 후 작업 B 에 의해 선점된다.
- 작업 B 는 차단 상태로 들어가기 전에 LCD 에 "Abort, Retry, Fail?"라고 쓴다.
- 작업 A 는 선점 된 시점부터 계속 진행하고 나머지 문자 "orld"를 출력한다.

LCD 에 손상된 문자열 "Hello Abort, Retry, Fail? world"가 표시된다.

### 2.Read, Modify, Write Operations(읽기, 수정, 쓰기 작업)

Listing 57 은 C 코드 라인과 그 결과 인 [ARM7] 어셈블리 출력을 보여준다. PORTA 의 값은 먼저 메모리 에서 레지스터로 읽혀지고 레지스터 내에서 수정 된 다음 다시 메모리에 기록된다는 것을 알 수 있다. 이를 읽기, 수정, 쓰기 작업이라고 한다.

```
/ * 컴파일되고있는 c 코드. * /155:
    PORTA |= 0x01;

/ * 생성된어셈블리코드. * /
0x00000264 481C LDR    R0,[PC,#0x0070] ; PORTA 의주소얻기
0x00000266 6801 LDR    R1,[R0,#0x00] ; PORTA 의값을 R1 로읽는다.
0x00000268 2201 MOV     R2,#0x01 ; 절대상수 1 을 R2 로이동하시오.
0x0000026A 4311 ORR     R1,R2 ; 또는 R1 (PORTA)와 R2 (상수 1)
0x0000026C 6001 STR     R1,[R0,#0x00] ; 새값을다시 PORTA 에저장하시오.
```

Listing 57 An example read, modify, write sequence



완료하기 위해 하나 이상의 명령어가 필요하고 인터럽트 될 수 있기 때문에 이것은 '비 원자'연산이다. 두 작업이 PORTA 라는 메모리 매핑 된 레지스터를 업데이트하려고 시도하는 다음 시나리오를 고려하십시오.

작업 A 는 PORTA 의 값을 레지스터 (작업의 읽기 부분)에 로드한다.

작업 A 는 동일한 작업의 수정 및 쓰기 부분을 완료하기 전에 작업 B 에 의해 선점된다.

B 태스크는 PORTA 값을 업데이트 한 다음 Blocked 상태로 들어간다.

작업 A 는 선점 된 시점부터 계속된다. 업데이트 된 값을 다시 PORTA 에 쓰기 전에 레지스터에 이미 보유하고있는 PORTA 값의 복사본을 수정한다

Task A 가 PORTA 의 최신 값을 업데이트하고 썼다. 태스크 B 는 태스크 A 와 PORTA 값의 사본을 취한 태스크 A 와 수정 된 값을 PORTA 레지스터에 쓰는 태스크 A 사이에서 PORTA 를 수정했다. 태스크 A 가 PORTA 에 쓸 때 태스크 B 가 이미 수행 한 수정을 덮어 씌으로써 PORTA 레지스터 값을 효과적으로 손상시킨다.

이 예에서는 주변 장치 레지스터를 사용하지만 전역 변수에서 읽기, 수정, 쓰기 작업을 수행 할 때 동일한 주체가 적용된다.

### 3.Non-atomic Access to Variables(변수에 대한 비 원자 액세스)

구조의 여러 멤버를 업데이트하거나 구조의 자연어 크기보다 큰 변수를 업데이트하는 경우 (예 : 16 비트 시스템에서 32 비트 변수 업데이트)는 둘 다 비 원자 연산의 예이다. 방해를 받으면 데이터가 손실되거나 손상 될 수 있다.

### 4.Function Reentrancy(기능 재진입 성)

함수는 둘 이상의 태스크 또는 태스크와 인터럽트 모두에서 함수를 호출하는 것이 안전하면 재진입 가능하다.

각 태스크는 자체 스택과 자체 코어 레지스터 값 집합을 유지 관리한다. 함수가 스택에 할당되거나 레지스터에 저장된 데이터 이외의 다른 데이터에 액세스하지 않으면 함수는 재진입 함수이다.

Listing 58 은 재진입 함수의 예이다. Listing 59 는 재진입이 아닌 함수의 예제이다.

```
/* 매개변수가함수에전달된다. 이것은스택이나 CPU 레지스터에전달된다. 어느  
쪽의방법이라도각작업이자체스택과자체레지스터값세트를유지하므로안전하  
다. */ long lAddOneHundered( long lVar1 )  
{
```

```
/*이함수범위변수는컴파일러와최적화수준에따라스택이나레지스터에도할당 된다.
이함수를호출하는각태스크또는인터럽트에는 lVar2 의자체복사본이있다.
* / long lVar2; lVar2 = lVar1 + 100;

/* 반환값은 CPU 레지스터에배치되지만스택에배치될가능성이높다. * / return lVar2; }
```

#### Listing 58 An example of a reentrant function

```
/*이경우 lVar1 은전역변수이므로호출하는모든작업 함수는변수의동일한단일사본에액세스한다. * /
long lVar1;

long lNonsenseFunction( void )
{
/*이변수는정적이므로스택에할당되지않는다. 함수를호출하는각태스크는변수의 동일한단일복사본에액세스한다. * /
static long lState = 0; long lReturn;

switch( lState )
{ case 0 : lReturn = lVar1 + 10; lState = 1; break;

case 1 : lReturn = lVar1 + 20; lState = 0; break;
}
}
```

#### Listing 59 An example of a function that is not reentrant

### Mutual Exclusion

작업간에 또는 작업과 인터럽트간에 공유되는 리소스에 대한 액세스는 데이터 일관성이 항상 유지되도록 '상호 배제'기술을 사용하여 관리해야 한다. 목표는 일단 태스크가 공유 자원에 액세스하기 시작하면 동일한 태스크가 자원이 일관성 있는 상태로 리턴 될 때까지 독점적 액세스를 보장하는 것이다.

FreeRTOS 는 상호 배제를 구현하는 데 사용할 수있는 몇 가지 기능을 제공하지만 최상의 상호 배제 방법은 리소스가 공유되지 않고 각 리소스가 단일 태스크에서만 액세스되는 방식으로 (가능한 경우) 애플리케이션을 설계하는 것이다.

### Scope

이 장에서는 독자의 좋은 이해를 제공하는 것을 목표로 :

자원 관리 및 통제가 필요한시기와 이유.

중요한 부분은 무엇입니까.

상호 배제가 의미하는 것.

스케줄러를 일시 중단하는 것이 무엇을 의미하나.

뮤텍스사용방법.

게이트 키퍼 작업을 생성하고 사용하는 방법.

우선 순위 반전은 무엇이며 우선 순위 상속이 해당 영향을 줄이거나 제거하지는 못한다.

## 4.2 CRITICAL SECTIONS AND SUSPENDING THE SCHEDULER

### Basic Critical Sections

기본 임계 섹션은 코드 50 의 영역으로, `taskENTER_CRITICAL ()` 및 `taskEXIT_CRITICAL ()` 매크로를 호출하여 각각 둘러싸여 있다 (목록 60 참조). 중요 섹션은 중요 영역.

```
/* PORTA 레지스터에 대한 액세스가 중요한 섹션 내에 배치되어 중단되지 않도록 하십시오. 중요 섹션을 입력 하십시오. */
taskENTER_CRITICAL();

/* 다른 작업으로의 전환은 taskENTER_CRITICAL () 호출과 taskEXIT_CRITICAL () 호출 사이에서 발생
할 수 없다. 인터럽트는 여전히 인터럽트 중첩을 허용하는 FreeRTOS 포트에서 실행될 수 있지만 우선 순위가
configMAX_SYSCALL_INTERRUPT_PRIORITY 상수에 지정된 값보다 높은 인터럽트만 가능하며 이러한 인터럽트는
FreeRTOS API 함수를 호출할 수 없다. */
PORTA |= 0x01;

/* 우리는 PORTA 에 접근했으므로 위험 부분을 안전하게 떠날 수 있다. */
taskEXIT_CRITICAL();
```

#### Listing 60 Using a critical section to guard access to a register

이 책과 함께 제공되는 예제 프로젝트에서는 `vPrintString ()`이라는 함수를 사용하여 표준 Out 에 문자열을 작성한다. 이것은 Open Watcom DOS 실행 파일의 터미널 창입니다. `vPrintString ()`은 많은 다른 태스크에서 호출되므로 이론적으로 구현시 Critical 섹션을 사용하여 표준 출력에 대한 액세스를 보호 할 수 있다 (Listing 61 참조).

```
void vPrintString( const portCHAR *pcString )
{
/* 중요한 섹션을 상호 배타의 조잡한 방법으로 사용하여 문자열을 stdout 에 쓴다. */ taskENTER_CRITICAL();
```

```
{  
printf( "%s", pcString );  fflush( stdout );  
} taskEXIT_CRITICAL();
```

\* 아무키나 실행 중인 응용 프로그램을 멈추게 할 수 있다. 실제로 키 값을 사용하는 실제 응용 프로그램은 키보드 입력에 대한 액세스도 보호해야 한다. \* / if( kbhit() )

```
{  
vTaskEndScheduler();  
}  
}
```

#### Listing 61 A possible implementation of vPrintString()

이런 식으로 구현된 중요 섹션은 상호 제외를 제공하는 매우 조잡한 방법이다. 인터럽트를 완전히 비활성화하거나 사용 중인 FreeRTOS 포트에 따라 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY에 설정된 인터럽트 우선 순위까지 간단하게 작동한다. 선점형 컨텍스트 전환은 인터럽트 내에서만 발생할 수 있으므로 인터럽트가 비활성화된 상태로 유지되는 한 taskENTER\_CRITICAL()을 호출한 태스크는 중요 섹션이 종료될 때까지 실행 중 상태로 유지된다.

중요한 섹션은 매우 짧게 유지해야 한다. 그렇지 않으면 인터럽트 응답 시간에 부정적인 영향을 미친다. taskENTER\_CRITICAL()에 대한 모든 호출은 taskEXIT\_CRITICAL()에 대한 호출과 밀접하게 연결되어야 한다. 이러한 이유 때문에 터미널에 쓰는 것이 상대적으로 길기 때문에 Critical 섹션을 사용하여 표준 출력을 보호해서는 안 된다 (Listing 61 참조). 또한 도스 에뮬레이터와 Open Watcom이 터미널 출력을 처리하는 방식은 라이브러리 호출이 인터럽트를 활성화된 채로 두기 때문에 이 형태의 상호 배제와 호환되지 않는다. 이 장의 예제는 대체 솔루션을 탐색한다.

커널이 중첩 깊이의 수를 유지하기 때문에 중요 섹션이 중첩되는 것이 안전하다. 임계 구역은 중첩 깊이가 0으로 돌아올 때만 - taskEXIT\_CRITICAL()에 대한 하나의 호출이 taskENTER\_CRITICAL()에 대한 모든 이전 호출에 대해 실행된 경우에만 종료된다.

#### Suspending (or Locking) the Scheduler

중요 섹션은 스케줄러를 일시 중단하여 작성할 수도 있다. 스케줄러 일시 중단은 때때로 스케줄러를 '잠금'하는 것으로도 알려져 있다.

기본 크리티컬 섹션은 다른 태스크 및 인터럽트를 통해 코드 영역을 액세스로부터 보호한다. 스케줄러를 일시 중단하여 구현된 중요 섹션은 인터럽트가 사용 가능하게 유지되기 때문에 다른 작업에 의한 코드 영역 액세스만 보호한다.

인터럽트를 단순히 비활성화하여 구현하기에는 너무 긴 임계 구역은 대신 스케줄러를 일시 정지하여 구현할 수 있지만 스케줄러를 다시 시작 (또는 '일시 중단 해제')하면 상대적으로 긴 작업이 될

가능성이 있으므로 고려할 사항은 다음과 같다. 사용하는 가장 좋은 방법은 각각의 경우에 만들어야 한다. **vTaskSuspendAll() API Function** `void vTaskSuspendAll( void );`

**Listing 62 The vTaskSuspendAll() API function prototype**

`vTaskSuspendAll()`을 호출하여 스케줄러가 일시 중단되었다. 스케줄러를 일시 중단하면 컨텍스트 전환이 발생하지 않지만 인터럽트는 사용 가능하게 유지된다. 인터럽트가 스케줄러가 일시 중단된 동안 컨텍스트 전환을 요청하면 요청은 보류 중으로 유지되고 스케줄러가 다시 시작될 때 (일시 중단된 경우)에 만 수행된다.

FreeRTOS API 함수는 스케줄러가 일시 중단된 동안 호출되어서는 안 된다.

### **xTaskResumeAll() API Function**

```
portBASE_TYPE xTaskResumeAll( void );
```

**Listing 63 The xTaskResumeAll() API function prototype** The

scheduler is resumed (un-suspended) by calling `xTaskResumeAll()`.

Table 18 `xTaskResumeAll()` return value

Returned Value	Description
Returned value	스케줄러가 일시 중단된 동안 요청된 컨텍스트 스위치는 보류 상태로 유지되고 스케줄러가 다시 시작될 때만 수행된다. <code>xTaskResumeAll()</code> 이 리턴하기 전에 이전에 보류중인 A 텍스트 전환이 수행되면 <code>pdTRUE</code> 가 리턴된다. 다른 모든 경우 <code>xTaskResumeAll()</code> 은 <code>pdFALSE</code> 를 반환한다.

`vTaskSuspendAll()` 및 `xTaskResumeAll()`에 대한 호출이 중첩되기 때문에 안전하다. 커널 중첩 깊이를 유지한다. 중첩 깊이가 0으로 돌아갈 때만 - 즉 `vTaskSuspendAll()`에 대한 모든 이전 호출에 대해 `xTaskResumeAll()`에 대한 하나의 호출이 실행된 경우에만 스케줄러가 다시 시작된다.

Listing 64에서는 터미널 출력에 대한 액세스를 보호하기 위해 스케줄러를 일시 중단하는 `vPrintString()`의 실제 구현을 보여 준다.

```
void vPrintString( const portCHAR *pcString ) {  
    /* 문자열을 stdout 에쓰고, 스케줄러를상호배제의방법으로정지시킨다. */ vTaskSuspendScheduler();  
    {  
        printf( "%s", pcString );    fflush( stdout );  
    }  
}
```

```
xTaskResumeScheduler() ;

/ * 아무키나실행중인응용프로그램을멈추게할수있다. 실제로키값을사용하는실제응용프로
그램은키보드입력에대한엑세스도보호해야한다. * / if( kbhit() )
{
vTaskEndScheduler() ;
}
}
```

**Listing 64 The implementation of vPrintString()**

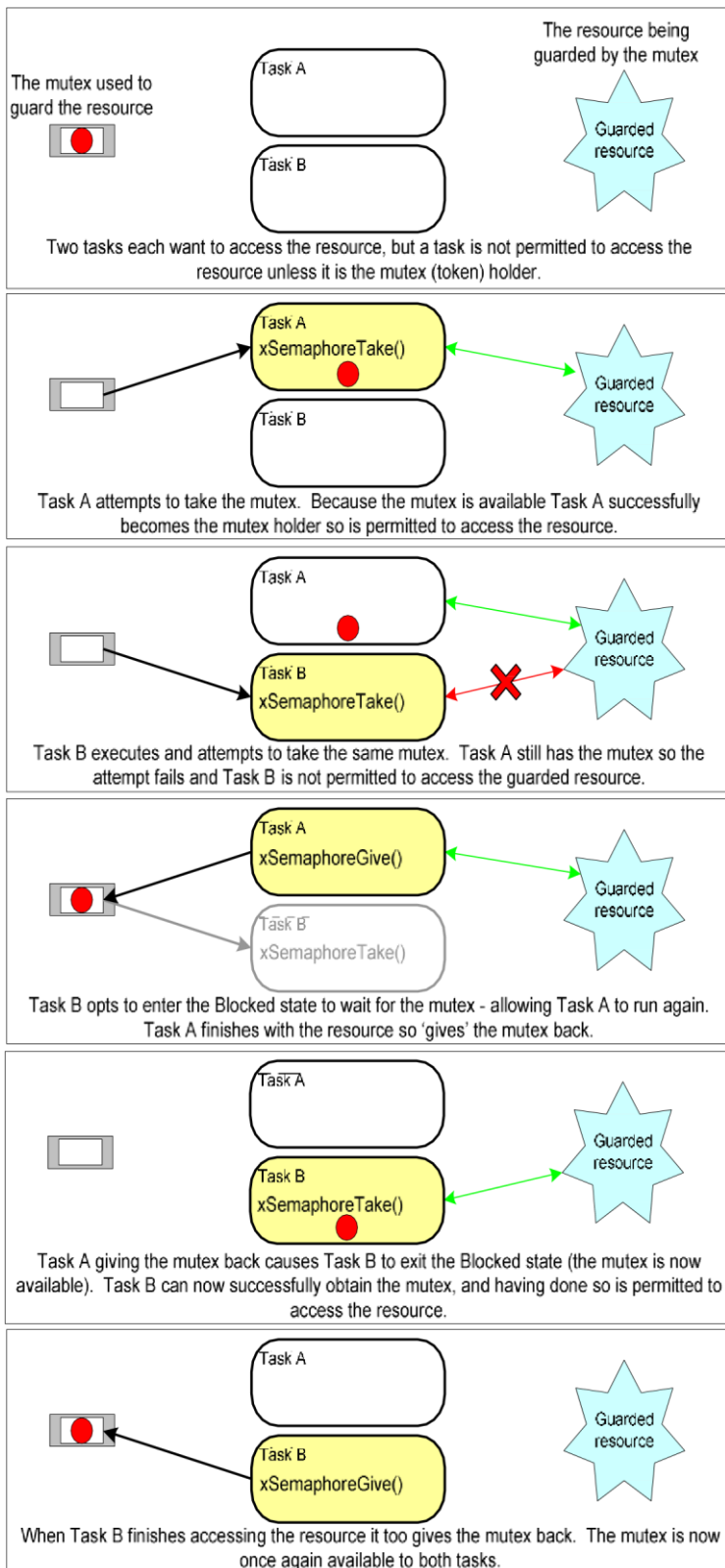
### 3. MUTEXES (AND BINARY SEMAPHORES)

뮤텍스는 두 개 이상의 작업간에 공유되는 리소스에 대한 액세스를 제어하는 데 사용되는 특수 유형의 바이너리 세마포이다. 단어 MUTEX 은 "MUTual EXclusion"에서 비롯된다.

상호 배제 시나리오에서 뮤텍스를 사용하면 개념적으로 공유되는 리소스와 연관된 토큰으로 생각할 수 있다. 리소스에 합법적으로 액세스하는 작업의 경우 먼저 토큰을 성공적으로 가져와야 한다 (토큰 소유자가 되어야 함). 토큰 소유자가 자원으로 끝나면 토큰을 다시 '반환'해야 한다. 토큰이 반환된 경우에만 다른 작업이 토큰을 성공적으로 가져 와서 동일한 공유 리소스에 안전하게 액세스 할 수 있다. 태스크가 토큰을보유하고있지않으면태스크가공유자원에액세스할수없습니다. 이메커니즘은그림 36 에나 와 있다.

뮤텍스와 바이너리 세마포어가 많은 특성을 공유하지만 그림 36 (뮤텍스가 상호 배제에 사용되는 시나리오)은 그림 30 (바이너리 세마포어가 동기화에 사용되는 경우)과 완전히 다르다. 가장 큰 차이점은 세마포어를 얻은 후 일어나는 일이다.

- 상호 배제에 사용되는 세마포어는 항상 반환되어야 한다.
- 동기화에 사용되는 세마포는 일반적으로 무시되고 반환되지 않는다.





**Figure 36 Mutual exclusion implemented using a mutex**

이 메커니즘은 순수하게 응용 프로그램 작성자의 규율을 통해 작동한다. 작업이 언제든지 자원에 액세스 할 수 있는 이유는 없지만 처음에는 뮤텝스 소유자가 될 수 없다면 각 작업이 "동의"한다.

### **xSemaphoreCreateMutex() API Function**

뮤텝스는 일종의 세마포어이다. 다양한 유형의 FreeRTOS 세마포어에 대한 핸들은 xSemaphoreHandle 유형의 변수에 저장된다.

뮤텝스를 실제로 사용하려면 먼저 만들어야 한다. 뮤텝스 유형 세마포어를 만들려면 xSemaphoreCreateMutex () API 함수를 사용하십시오.

```
xSemaphoreHandle xSemaphoreCreateMutex( void );
```

**Listing 65 The xSemaphoreCreateMutex() API function prototype**

**Table 19 xSemaphoreCreateMutex() return value**

ParameterName / Returned Value	Description
Returnedvalue	NULL 이 리턴되면 FreeRTOS 가 뮤텝스 데이터 구조를 할당하기에 부족한 힙 메모리가 있었기 때문에 뮤텝스를 작성할 수 없다. 5 장에서는 메모리 관리에 대한 자세한 정보를 제공한다.  NULL 이 아닌 반환 값은 뮤텝스가 성공적으로 만들어 졌음을 나타낸다. 반환 된 값은 생성 된 뮤텝스의 핸들로 저장되어야 한다.

### **Example 15. Rewriting vPrintString() to Use a Semaphore**

이 예에서는 prvNewPrintString ()이라는 vPrintString ()의 새 버전을 만든 다음 여러 작업에서 새 함수를 호출한다. prvNewPrintString ()은 vPrintString ()과 동일한 기능을하지만 기본 임계 영역 대신 표준 출력에 대한 액세스를 제어하기 위해 뮤텝스를 사용한다. prvNewPrintString ()의 구현은 Listing 66 과 같다.

```
static void prvNewPrintString( const portCHAR *pcString )  
{
```

```
/ * mutex 는 스케줄러가 시작되기 전에 생성되므로 이 태스크가 처음 실행될 때까지 이미 존재한다.

mutex를 사용할 수 없는 경우 mutex를 무기한 차단하여 mutex를 기다린다. xSemaphoreTake ()에 대한 호출은
mutex가 성공적으로 획득되었을 때만 리턴되므로 함수 리턴값을 확인할 필요가 없다. 다른 지연 시간이 사용된
경우 코드는 xSemaphoreTake ()가 공유 리소스 (이 경우에는 표준 출력)에 액세스하기 전에 pdTRUE를 반환하는
지 확인해야 한다. * / xSemaphoreTake( xMutex, portMAX_DELAY ); {
/ * mutex가 성공적으로 얻어지면 다음 라인만 실행된다. 한번에 하나의 작업만 mutex를 가질 수 있으므로 표
준 출력에 자유롭게 액세스할 수 있다. * /
printf( "%s", pcString );
fflush( stdout );

/ * mutex는 반드시 돌려줘야 한다! * /
}
xSemaphoreGive( xMutex );

/ * 아무 키나 실행 중인 응용 프로그램을 멈추게 할 수 있다. 실제로 키값을 사용하는 실제 응용 프로그램은 키보
드에 대한 액세스도 보호해야 한다. 실제 응용 프로그램에서는 키 누르기를 처리하는 둘 이상의 작업이 있을 가능성 이 거의 없다. * /
if( kbhit() )
}
}
```

#### Listing 66 The implementation of prvNewPrintString()

prvNewPrintString ()은 prvPrintTask ()라는 태스크의 두 인스턴스에 의해 반복적으로 호출된다. 임의의 지연 시간이 각 호출 사이에 사용된다. task 매개 변수는 태스크의 각 인스턴스에 고유 한 문자열을 전달 하는 데 사용된다. prvPrintTask ()의 구현은 Listing 67 과 같다.

```
static void prvPrintTask( void *pvParameters )
{
    char *pcStringToPrint;

    / * 이 작업의 두 인스턴스가 만들어지므로 작업이 prvNewPrintString ()에 보낼 문자열이 작업 매개 변수를 사
    용하여 작업으로 전달된다.
    이것을 필요한 유형으로 전송하십시오. * /
    pcStringToPrint = ( char * ) pvParameters;

    for( ;; )
    {
        / * 새롭게 정의된 함수를 사용하여 문자열을 출력한다. * /
        prvNewPrintString( pcStringToPrint );

        / * 의사 임의 시간 대기. rand ()는 반드시 재진입 가능하지는 않지만 이 경우 코드가 반환되는 값을 신경 쓰지 않아도 상관 없다.
        보다 안전한 응용 프로그램에서는 재진입 가능하다고 알려진 버전의 rand ()를 사용해야 한다.
        또는 중요한 섹션을 사용하여 rand ()에 대한 호출을 보호해야 한다. * /
        vTaskDelay( ( rand() & 0x1FF ) );
    }
}
```

```
}  
}
```

#### Listing 67 The implementation of prvPrintTask() for Example 15

정상적으로 main ()은 단순히 뮤텍스를 생성하고, 작업을 생성 한 다음 스케줄러를 시작한다. 구현은 Listing 68 과 같다.

prvPrintTask ()의 두 인스턴스는 다른 우선 순위에서 만들어 지므로 우선 순위가 낮은 작업이 우선 순위 가 높은 작업에 의해 선점되는 경우가 있다. 뮤텍스는 한 태스크가 다른 태스크에 의해 선점 될 때에도 각태스크가터미널에대해상호배타적인엑세스를보장하기위해사용되므로표시되는문자열은정확하고 결코 손상되지 않는다. 선행 빈도는 작업이 Blocked 상태에서 소비하는 최대 시간을 줄이면 증가 할 수 있으며 기본값은 0x1ff tick 이다.

```
int main( void ) {  
    /* 세마포어를사용하기전에명시적으로만들어야한다. 이에제에서뮤텍스타입세마포어가생성된다. */ xMutex =  
    xSemaphoreCreateMutex();  
  
    /* 태스크는의사난수지연을사용하고, 난수생성기에시드한다. */  
    srand( 567 );  
  
    /* 작업을만들기전에세마포가성공적으로만들어졌는지확인하시오. */  
    if( xMutex != NULL ) {  
        /* stdout 에쓰는태스크의두인스턴스를만든다. 쓰는문자열은 task 매개변수로전달된다. 작업은다른우  
선순위에서만들어지므로일부선매가발생한다. */ xTaskCreate( prvPrintTask, "Print1", 1000,  
"Task 1 *****\r\n", 1, NULL );  
  
xTaskCreate( prvPrintTask, "Print2", 1000,  
"Task 2 -----\r\n", 2, NULL );  
  
        /* 생성된작업이실행되기시작하도록스케줄러를시작하시오. */  
        vTaskStartScheduler(); }  
  
    /* 모든것이잘되면스케줄러가이제태스크를실행할때 main ()이여기에도달하지않는다. main ()이여기  
에도달하면유휴작업을생성하는데사용할수있는힙메모리가부족할수있다.  
5 장에서는메모리관리에대한자세한정보를제공한다. */  
    for( ;; ); }
```

#### Listing 68 The implementation of main() for Example 15

예 15 가 실행될 때 생성 된 출력이 그림 37 에 표시된다. 가능한 실행 순서가 그림 38 에 설명되어 있다.

```

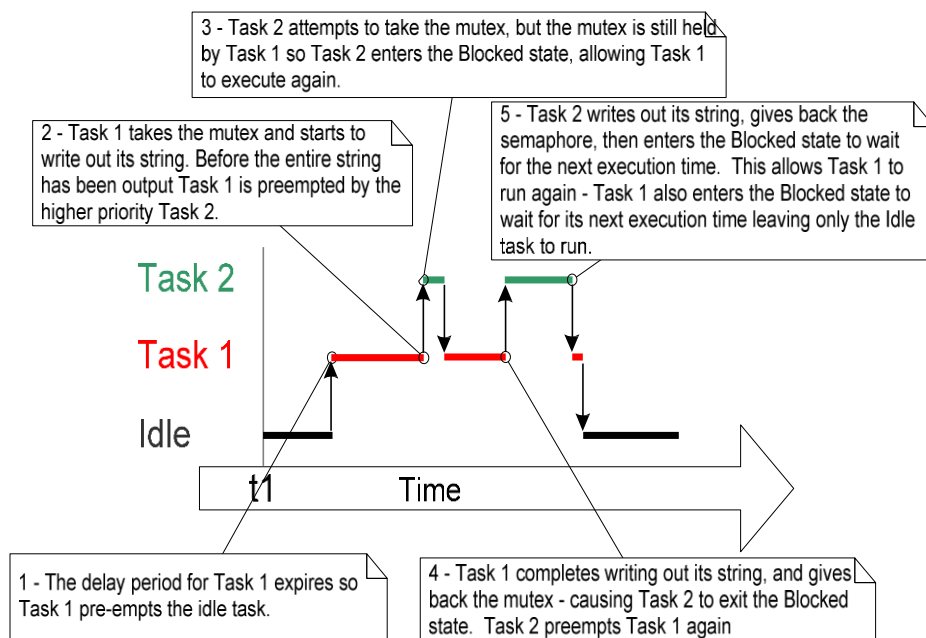
C:\WINDOWS\system32\cmd.exe - rtsdemo

Task 2
Task 1 *****
Task 1 *****
Task 2 -----
Task 2 -----
Task 1 *****
Task 2 -----
Task 2 -----
Task 1 *****
Task 1 *****
Task 2 -----
Task 1 *****
Task 1 *****
Task 2 -----
Task 2 -----
Task 1 *****
Task 2 -----
Task 1 *****
Task 1 *****
Task 2 -----
Task 2 -----
Task 1 *****
Task 2 -----
Task 1 *****
Task 1 *****
Task 2 -----
Task 2 -----

```

**Figure 37 The output produced when Example 15 is executed**

그림 37에서는 예상대로 터미널에 표시되는 문자열에 손상이 없음을 보여준다. 임의 순서는 작업에서 사용되는 임의 지연 기간의 결과이다.



**Figure 38 A possible sequence of execution for Example 15**

## Priority Inversion

그림 38은 뮤텍스를 사용하여 상호 배제를 제공 할 때 발생할 수 있는 잠재적인 함정 중 하나를 보여준다. 묘사 가능한 실행 순서는 더 낮은 우선 순위의 태

스크 1 이 뮅텍스의 제어를 포기하기를 기다려야 하는 더 높은 우선 순위의 태스크 2 를 도시한다. 이런 방식으로 우선 순위가 낮은 작업이 지연되는 것을 우선순위역전이라고한다. 높은우선순위의작업이세마포어를기다리는동안 중간 우선 순위의 작업이 실행되기 시작하면 낮은 우선 순위의 작업을 실행하지 않고도 낮은 우선 순위의 작업을 기다리는 높은 우선 순위의 작업이 실행되 면 이 바람직하지 않은 동작이 더 과장 될 것이다 ! 이 최악의 시나리오는 그림 39 에 나와 있다.

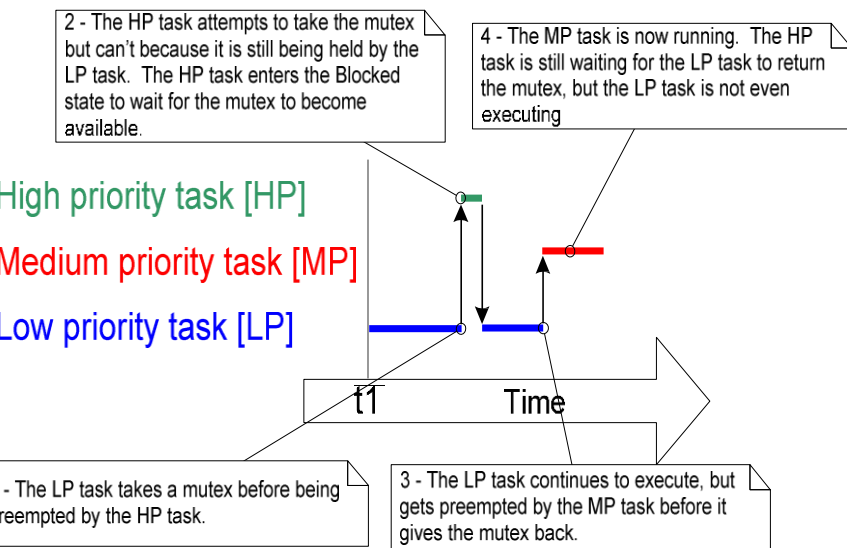


Figure 39 A worst case priority inversion scenario

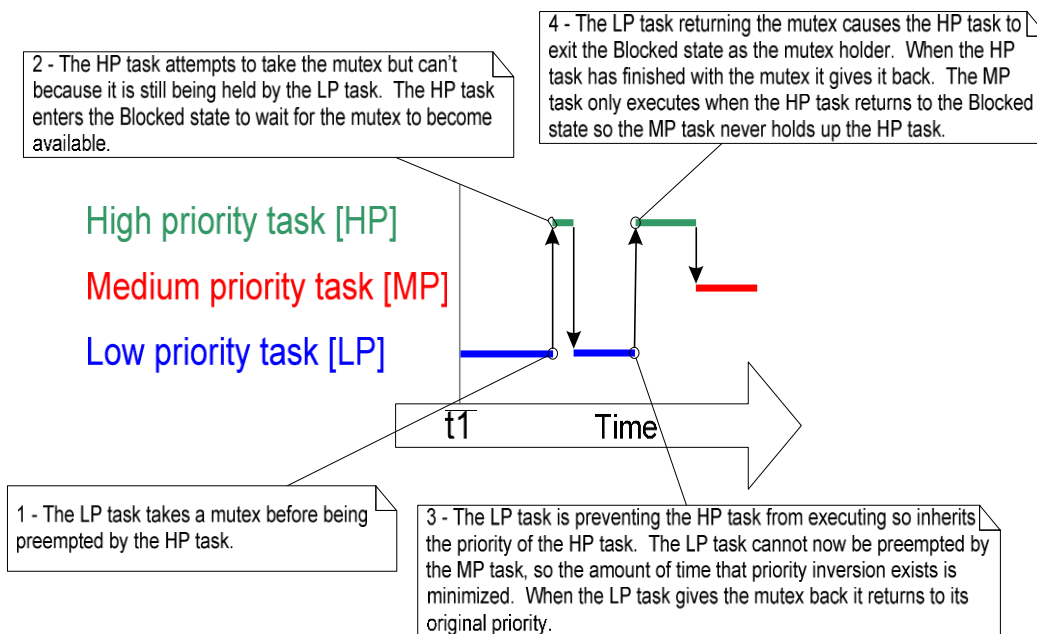
우선 순위 반전은 중대한 문제가 될 수 있지만 작은 임베디드 시스템에서 는 시스템 설계시 자원에 액세스하는 방법을 고려하여 피해야 한다.

## Priority Inheritance

FreeRTOS 뮅텍스와 바이너리 세마포어는 매우 유사함 하다. 뮅텍스가 유 일한 차이점은 자동적으로 기본적인 '우선 순위 상속'메커니즘을 제공한다 는 것이다. 우선 순위 상속은 우선 순위 반전의 부정적인 영향을 최소화하 는 방식이다. 우선 순위 반전을 '수정'하지 않고 단지 영향을 줄인다. 우선 순위 상속은 시스템 동작의 수학적 분석을 더 복잡한 연습으로 만든다. 따 라서 피할 수 있는 경우 우선 순위 상속에 의존하지 않는 것이 좋다.

우선 순위 상속은 뮅텍스 홀더의 우선 순위를 일시적으로 높이고 동일한 뮅텍스를 얻으려고 하는 우선 순위가 높은 작업의 우선 순위를 높이는 방 식으로 작동한다. 뮅텍스를 유지하는 우선 순위가 낮은

작업은 뮉텍스를 대기하는 작업의 우선 순위를 '상속합니다'. 이것은 그림 40 에 나와 있다. 뮉텍스 홀더의 우선 순위는 뮉텍스를 다시 돌려 주면 원래 값으로 자동 재 설정된다.



**Figure 40 Priority inheritance minimizing the effect of priority inversion**

선호도는 우선순위역전을 피하기 위한 것이고, FreeRTOS 는 메모리가 제한된 마이크로 컨트롤러를 대상으로 하기 때문에 뮤텁스에 의해 구현되는 우선 순위 상속 메커니즘은 태스크가 한 번에 단일 뮤텁스 만 보유한다고 가정하는 기본적인 형태 일뿐이다.

### Deadlock (or Deadly Embrace)

'교착 상태'는 뮤텁스를 상호 배제에 사용하는 또 다른 잠재적인 함정이다. 교착 상태는 때로는 '치명적인 포옹'이라는 더 극적인 이름으로도 알려져 있다.

교착 상태는 두 태스크가 다른 태스크가 보유하는 자원을 대기 중이므로 두 태스크가 진행될 수 없을 때 발생한다. 동작을 수행하기 위해 작업 A와 작업 B가 모두 뮤텁스 X와 뮤텁스 Y를 획득해야 하는 다음 시나리오를 고려하십시오.

1. 작업 A가 실행되고 뮤텁스 X를 성공적으로 가져온다.
2. 작업 A는 작업 B에 의해 선점된다.
3. 태스크 B는 뮤텁스 X를 취하기 전에 뮤텁스 Y를 성공적으로 사용하지만 태스크 A는 뮤텁스 X를 보유하므로 태스크 B는 사용할 수 없다. 태스크 B는 뮤텁스 X가 해제 될 때까지 대기 할 Blocked 상태로 들어간다.
4. 작업 A는 계속 실행된다. 뮤텁스 Y를 취하려고 시도하지만 뮤텁스 Y는 작업 B에 의해 유지되므로 작업 A에는 사용할 수 없다. 작업 A는 뮤텁스 Y가 해제 될 때까지 대기 할 Blocked 상태로 들어간다.

이시나리오의끝에서작업 A 는작업 B 가보유한뮤텍스를기다리고있으며작업 B 는 작업 A 가 보유한 뮤텍스를 기다리고 있다. 작업이 더 이상 진행될 수 없기 때문에 교착 상태가 발생했다.

우선 순위 반전과 마찬가지로 교착 상태를 피하는 가장 좋은 방법은 디자인 타임에 잠재력을 고려하고 시스템이 단순히 발생하지 않도록 설계하는 것이다. 실제로 시스템 설계자는 전체 애플리케이션을 잘 이해하고 발생할 수 있는 영역을 식별하고 제거 할 수 있기 때문에 소형 임베디드 시스템의 경우 교착 상태가 큰 문제는 아니다.



## 4. GATEKEEPER TASKS

게이트 키퍼 태스크는 우선 순위 반전 또는 교착 상태에 대한 걱정 없이 상호 배제를 구현하는 깔끔한 방법을 제공한다.

게이트 키퍼 태스크는 리소스의 단독 소유권을 갖는 태스크이다. 게이트 키퍼 작업 만 리소스에 직접 액세스 할 수 있다. 리소스에 액세스해야 하는 다른 작업은 게이트 키퍼의 서비스를 사용하여 간접적으로 수행 할 수 있다.

### Example 16. Re-writing vPrintString() to Use a Gatekeeper Task

예제 16 은 vPrintString ()에 대한 대안 구현을 제공한다. 이번에는 게이트 키퍼 태스크가 표준 출력에 대한 액세스를 관리하는 데 사용된다. 작업이 터미널에 메시지를 쓰고 싶을 때 직접 인쇄 기능을 호출하지 않고 메시지를 게이트 키퍼에 보낸다.

게이트 키퍼 태스크는 FreeRTOS 큐를 사용하여 터미널에 대한 액세스를 직렬화한다. 태스크의 내부 구현은 터미널에 직접 액세스 할 수 있는 유일한 태스크이기 때문에 상호 배제를 고려할 필요가 없다.

게이트 키퍼 작업은 메시지가 대기열에 도착하기를 기다리는 차단 상태에서 대부분의 시간을 소비한다. 메시지가 도착하면 게이트 키퍼는 차단 된 상태로 돌아가 다음 메시지를 기다리기 전에 메시지를 표준 출력에 쓴다. 게이트 키퍼 태스크의 구현은 Listing 70 과 같다.

인터럽트는 대기열로 보낼 수 있으므로 인터럽트 서비스 루틴은 게이트 키퍼의 서비스를 사용하여 안전하게 메시지를 터미널에 쓸 수 있다. 이 예에서 틱 후크 (tick hook) 기능은 매 200 틱마다 메시지를 작성하는 데 사용된다. 틱 후크 (또는 콜백)는 각 틱 인터럽트 동안 커널에 의해 호출되는 함수이다. 틱 후크 기능을 사용하려면 :

- FreeRTOSConfig.h 에서 configUSE\_TICK\_HOOK 를 1 로 설정하시오.
- Listing 69 의 정확한 함수 이름과 프로토 타입을 사용하여 hook 함수의 구현을 제공한다.

```
void vApplicationTickHook( void );
```

**Listing 69 The name and prototype for a tick hook function**

틱 후크 함수는 틱 인터럽트의 컨텍스트 내에서 실행되므로 매우 짧게 유지해야 하며 중간 크기의 스택공간만 사용하고 'FromISR ()'로 끝나지 않는 FreeRTOS API 함수는 호출하지 않아야 한다.

```
static void prvStdioGatekeeperTask( void *pvParameters )
{
    char *pcMessageToPrint;

    /* 이것은 터미널 출력에 쓸 수 있는 유일한 작업이다. 문자열을 출력에 쓰고자 하는 다른 작업은 터미널에 직접
    액세스하지 않고 이 작업으로 문자열을 보낸다. 이 태스크만 표준에 액세스하므로 태스크 자체의 구현 내에서 고려
    해야 할 상호배제 또는 직렬화 문제는 없다. */
    for( ;; )
    {
        /* 메시지가 도착할 때까지 기다린다. 무기한 블록 시간이 지정된다.
        따라서 반환 값을 확인할 필요가 없다. 메시지가 성공적으로 수신된 경우에만 함수가 반환된다.
        */ xQueueReceive( xPrintQueue, &pcMessageToPrint, portMAX_DELAY );

        /* 수신된 문자열을 출력한다. */ printf( "%s", pcMessageToPrint ); fflush( stdout );

        /* 이제 간단히 다음 메시지를 기다리시오. */
    }
}
```

#### Listing 70 The gatekeeper task

메시지를 인쇄하는 작업은 예제 15 에서 사용된 것과 비슷하다. 단, 이 시간은 문자열이 대기열에서 게이트키퍼 작업으로 직접 보내지는 것이다. 구현은 Listing 71 에 표시되어 있다. 이전처럼 두 개의 개별 인스턴스가 작성되었으며, 각 인스턴스는 task 매개변수를 통해 전달된 고유한 문자열을 출력한다.

```
static void prvPrintTask( void *pvParameters )
{
    int iIndexToString;

    /* 이 작업의 두 인스턴스가 생성된다. 작업 매개변수는 전달하는데 사용된다.
    태스크에의 캐릭터라인 배열에 대한 인덱스. 이것을 필요한 유형으로 전송하시오. */ iIndexToString =
    ( int ) pvParameters;

    for( ;; )
    {
        /* 문자열을 직접 출력하는 것이 아니라 문자열을 가리키는 포인터를 대기열을 통해 게이트키퍼 작업에 전달하여
        문자열을 출력한다. 큐가 생성되기 전에 스케줄러가 시작되므로 이 태스크가 처음 실행될 때까지 이미 존재한다.
        블록 시간은 큐에 항상 공백이 있어야 하므로 로지정되지 않는다. */
        xQueueSendToBack( xPrintQueue, &( pcStringsToPrint[ iIndexToString ] ), 0 );

        /* 의사 임의 시간 대기. rand () 는 반드시 재진입 가능하지는 않지만 이 경우 코드가 반환되는 값을 신경쓰지 않아도 상관없다.
        보다 안전한 응용 프로그램에서는 재진입 가능하다고 알려진 버전의 rand () 를 사용해야 한다.
        또는 중요한 섹션을 사용하여 rand () 에 대한 호출을 보호해야 한다. */
        vTaskDelay( ( rand() & 0xFF ) );
    }
}
```

```
}
```

#### Listing 71 The print task implementation for Example 16

틱 후크 기능은 호출 횟수를 단순히 계산하여 카운트가 200 에 도달 할 때마다 메시지를 게이트 키퍼 태 스크로보낸다. 데모용으로 틱후크는 큐의 맨앞에쓰고 인쇄작업은뒤쪽에쓴다 대기열의 tick 후크 구현 은 Listing 72 와 같다.

```
void vApplicationTickHook( void )
{
    static int iCount = 0;
    portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

    / * 200 틱마다 메시지를 출력하시오. 이 메시지는 직접 작성되지는 않지만 게이트 키퍼 작업으로 전송된다. * / iCount++;
    if( iCount >= 200 )
    {
        / * 이 경우 마지막 매개변수 (xHigherPriorityTaskWoken) 는 실제로 사용되지 않지만 계속 제공되어야 한다. *
        /
        xQueueSendToFrontFromISR( xPrintQueue,
        &( pcStringsToPrint[ 2 ] ), &xHigherPriorityTaskWoken );

        / * 200 틱시간에 문자열을 다시 인쇄할 수 있도록 카운트를 재설정한다. * / iCount = 0;
    }
}
```

#### Listing 72 The tick hook implementation

정상적으로 main ()은 예제를 실행하는 데 필요한 대기열과 작업을 생성 한 다음 스케줄러를 시작한다. main ()의 구현은 Listing 73 과 같다.

```

/ * 작업과인터럽트가게이트키퍼를통해출력할문자열을정의하시오. * /
static char *pcStringsToPrint[] =
{
"Task 1 *****\r\n",
"Task 2 ----- \r\n",
"Message printed from the tick hook interrupt #####\r\n" };

/ * ----- * /

/ * xQueueHandle 유형의변수를선언하시오. 프린트작업및틱인터럽트에서게이트키퍼작업으
로메시지를보내는데사용된다. * /
xQueueHandle xPrintQueue;

/*-----*/

int main( void )
{
/ * 대기열을사용하기전에명시적으로만들어야한다. 대기행렬은최대 5 개의문자포인터를보 유하도록작성된다. * /
xPrintQueue = xQueueCreate( 5, sizeof( char * ) );

/ * 태스크는의사난수지연을사용하고, 난수생성기에시드한다. * / srand( 567 );

/ * 큐가성공적으로생성되었는지확인하시오. * /
if( xPrintQueue != NULL )
{
/ * 게이트키퍼에게메시지를보내는두개의인스턴스를생성한다. 작업에서사용하는문자열에대 한인텍스는작업매개변수
(xTaskCreate () 의네번째매개변수) 를통해작업에전달된다. 작업
은다른우선순위에서만들어지므로우선순위가높은작업이우선순위가낮은작업보다우선적으 로선택된다. * /
xTaskCreate( prvPrintTask, "Print1", 1000, ( void * ) 0, 1, NULL );
xTaskCreate( prvPrintTask, "Print2", 1000, ( void * ) 1, 2, NULL );

/ * 게이트키퍼작업을만든다. 이것은표준출력에직접엑세스할수있는유일한작업이다. * /
xTaskCreate( prvStdioGatekeeperTask, "Gatekeeper", 1000, NULL, 0, NULL );

/ * 생성된작업이실행되기시작하도록스케줄러를시작하시오. * /
vTaskStartScheduler(); }

/ * 모든것이잘되면스케줄러가이제태스크를실행할때 main () 이여기에도달하지않는다. main
() 이여기에도달하면유훁작업을생성하는데사용할수있는힙메모리가부족할수있다.
5 장에서는메모리관리에대한자세한정보를제공한다. * / for( ;; ); }
```

#### Listing 73 The implementation of main() for Example 16

예제 16 이 실행될 때 생성 된 출력이 그림 41 에 나와 있다. 태스크에서 발생한 문자열과 인터럽트에서 비롯된 문자열은 모두 손상 없이 올바르게 인쇄된다.

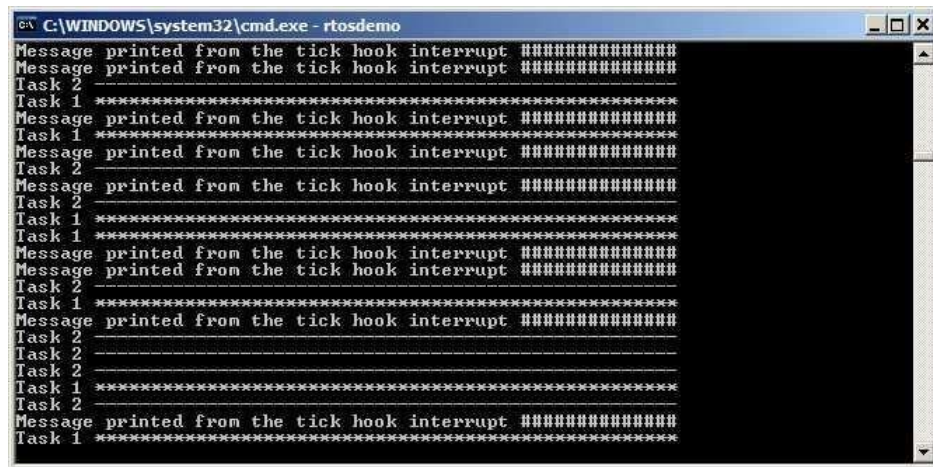


Figure 41 The output produced when Example 16 is executed

게이트 키퍼 작업은 인쇄 작업보다 우선 순위가 낮게 할당되었으므로 게이트 키퍼에 전송된 메시지는 인쇄 작업이 차단된 상태가 될 때까지 대기열에 남아 있었다. 경우에 따라 게이트 키퍼에 우선 순위를 지정하여 메시지가 더 빨리 처리되도록 하는 것이 적절할 수 있다

- 이렇게 하면 보호된 리소스에 대한 액세스가 완료 될 때까지 우선 순위가 낮은 작업이 지연되는 게이트 키퍼가 발생한다.

## CHAPTER 5

# MEMORY MANAGEMENT

## 1. CHAPTER INTRODUCTION AND SCOPE

커널은 태스크, 큐 또는 세마포어가 생성 될 때마다 동적으로 RAM 을 할당해야한다. 표준 malloc () 및 free () 라이브러리 함수를 사용할 수 있지만 다음 문제 중 하나 이상이 발생할 수 있다.

1. 그들은 작은 임베디드 시스템에서 항상 사용할 수 있는 것은 아니다.
2. 구현이 상대적으로 클 수 있으므로 중요한 코드 공간을 차지한다.
3. 스레드는 거의 안전하지 않다.
4. 그것들은 결정적이지 않다. 기능을 실행하는 데 걸리는 시간은 호출마다 다르다.
5. 메모리 조각화로 어려움을 겪을 수 있다.
6. 링커 구성이 복잡해질 수 있다.

임베디드 시스템마다 다양한 RAM 할당 및 타이밍 요구 사항이 있으므로 단일 RAM 할당 알고리즘 만 응용 프로그램의 하위 집합에 적합하다. 따라서 FreeRTOS 는 메모리 할당을 (코어 코드베이스의 일부가 아닌) 휴대용 계층의 일부로 취급한다. 따라서 개별 응용 프로그램이 적절한 경우 고유 한 특정 구현을 제공 할 수 있다.

커널이 malloc ()을 직접 호출하는 대신 RAM 을 필요로 할 때 pvPortMalloc ()을 대신 호출한다. RAM 이 해제 될 때 free ()를 직접 호출하는 대신 커널이 대신 vPortFree ()를 호출한다. pvPortMalloc ()은 malloc ()과 동일한 프로토 타입을 갖고 vPortFree ()는 free ()와 동일한 프로토 타입을 갖는다.

FreeRTOS 에는 pvPortMalloc ()과 vPortFree ()의 세 가지 구현 예가 있다.이 모든 것이 장에서 설명한 다. FreeRTOS 사용자는 예제 구현 중 하나를 사용하거나 자체 구현을 제공 할 수 있다.

세 가지 예제는 각각 heap\_1.c, heap\_2.c 및 heap\_3.c 파일에 정의되어 있으며 모두 FreeRTOS \ Source \ Portable \ MemMang 디렉토리에 있습니다. FreeRTOS 의 초기 버전에서 사용 된 원래의 메모리 풀과 블록 할당 체계는 블록과 풀의 크기를 결정하는 데 필요한 노력과 이해 때문에 제거되었다.

작은 임베디드 시스템에서는 스케줄러가 시작되기 전에 태스크, 큐 및 세마포어 만 생성하는 것이 일반적이다. 이 경우 응용 프로그램이 실제 실시간 기능을 수행하기 전에 메모리가 동적으로 할당되고 할당

된 메모리는 다시 해제되지 않는다. 이는 선택된 할당 스키마가 결정론 및 조각화와 같은 더 복잡한 문제를 고려할 필요가 없으며 대신 코드 크기 및 단순성과 같은 특성만 고려할 수 있음을 의미한다.

## Scope

이 장은 독자들에게 다음에 대한 좋은 이해를 제공하고자 한다 :

- FreeRTOS 가 RAM 을 할당 할 때.
- FreeRTOS 와 함께 제공되는 세 가지 예제 메모리 할당 체계.

## 2. EXAMPLE MEMORY ALLOCATION SCHEMES

### Heap\_1.c

Heap\_1.c 는 pvPortMalloc ()의 아주 기본적인 버전을 구현하고 vPortFree ()를 구현하지 않는다. 작업, 대기열 또는 세마포어를 삭제하지 않는 응용 프로그램은 heap\_1 을 사용할 가능성이 있다. Heap\_1 은 항상 결정적이다.

할당스키마는 pvPortMalloc ()에대한호출이만들어지면서간단한배열을작은블록으로세분한다. 배열은 FreeRTOS 힙이다.

어레이의 전체 크기 (바이트)는 FreeRTOSConfig.h 내의 configTOTAL\_HEAP\_SIZE 정의에 의해 설정된다. 이 방법으로 큰 배열을 정의하면 배열이 실제로 할당되기 전에도 응용 프로그램이 많은 RAM 을 소비하는 것처럼 보일 수 있다.

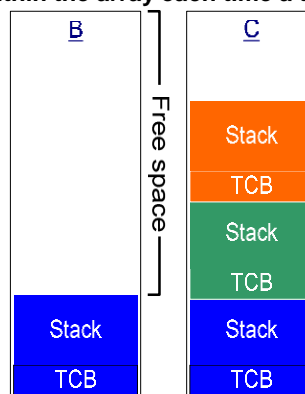
생성된 각 태스크에는 TCB (Task Control Block)와 스택이 힙에서 할당되어야 한다. 그림 42 는 heap\_1 이 작업이 생성될 때 간단한 배열을 세분화하는 방법을 보여준다. 그림 42 를 참조하면 :

- A 는 작업이 생성되기 전에 배열을 보여 주며, 전체 배열은 자유이다.
- B 는 하나의 작업이 생성된 후 배열을 보여준다.
- C 는 세 개의 작업이 만들어진 후 배열을 보여준다.

Figure 42 RAM being



allocated within the array each time a task is



created



## Heap\_2.c

Heap\_2.c 는 또한 configTOTAL\_HEAP\_SIZE 에 의해 차원이 지정된 간단한 배열을 사용한다. 메모리를 할당하는 데 가장 적합한 알고리즘을 사용하고 heap\_1 과 달리 메모리를 해제할 수 있다. 다시 배열은 정적으로 선언되므로 배열이 실제로 할당되기 전에도 응용 프로그램이 많은 RAM 을 소비하는 것처럼 보인다.

가장 적합한 알고리즘은 pvPortMalloc ()이 요청한 바이트 수에 가장 가까운 크기의 메모리 블록을 사용하도록 보장한다. 예를 들어, :

1. 힙에는 각각 5 바이트, 25 바이트 및 100 바이트의 사용 가능한 메모리 블록이 있다.
2. pvPortMalloc ()은 20 바이트의 RAM 을 요청하기 위해 호출된다.

요청 된 바이트 수에 맞는 RAM 의 가장 작은 빈 블록은 25 바이트 블록이므로 pvPortMalloc ()은 25 바이트 블록을 20 바이트의 한 블록과 5 bytes<sup>3</sup>의 한 블록으로 분할 한 다음 20 바이트의 포인터를 반환한다 블록. 새로운 5 바이트 블록은 향후 pvPortMalloc () 호출에 사용할 수 있다.

Heap\_2.c 는 인접한 빈 블록을 하나의 큰 블록으로 결합하지 않으므로 조각화가 발생할 수 있다. 그러나 할당되고 해제 된 블록이 항상 동일한 크기이면 조각화는 문제가 되지 않는다. Heap\_2.c 는 작성된 타스 크에 할당 된 스택의 크기가 변경되지 않으면 태스 크를 반복적으로 작성하고 삭제하는 응용 프로그램에 적합하다.

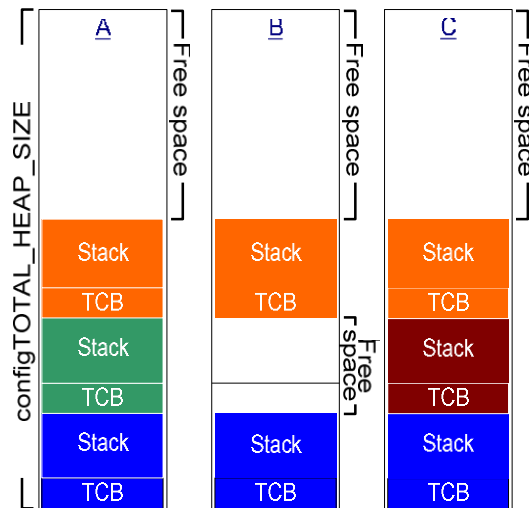


Figure 43 RAM being allocated from the array as tasks are created and deleted

<sup>3</sup> heap\_2 는 힙 영역 내의 블록 크기에 대한 정보를 저장하므로 두 개의 분할된 블록의 합이 실제로 25 보다 작기때문에 단순화된 것이다.

그림 43 은 작업이 생성, 삭제 및 다시 생성 될 때 최적 알고리즘이 어떻게 작동 하는지를 보여준다. 그림 43 을 참조하시오.

A 는 세 가지 작업이 만들어진 후 배열을 보여준다. 어레이의 맨 위에 큰 빈 블록이 남아 있다..

- B 는 작업 중 하나가 삭제 된 후 배열을 보여줍니다. 배열 상단의 큰 빈 블록이 남아 있다. 이전에는 TCB 와 삭제 된 태스크의 스택에 이전에 할당 된 2 개의 작은 빈 블록이 있다.
- C 는 다른 작업이 만들어진 후 상황을 보여준다. 태스크를 생성하면 pvPortMalloc () 을 두 번 호출 하고 하나는 새 TCB 를 할당하고 하나는 태스크 스택을 할당한다 (pvPortMalloc () 호출은 xTaskCreate () API 함수내에서 내부적으로 발생함).
- 모든 TCB 는 정확히 같은 크기이므로 가장 적합한 알고리즘은 이전에 삭제 된 작업의 TCB 에 할당되었던 RAM 블록이 새 작업의 TCB 를 할당하는 데 다시 사용되도록 보장한다.
- 새로 생성 된 작업에 할당 된 스택의 크기는 이전에 삭제 된 작업에 할당 된 크기와 동일하므로 가장 적합한 알고리즘을 사용하면 이전에 삭제 된 작업의 스택에 할당 된 RAM 블록을 다시 사용할 수 있다. 새 작업의 스택을 할당하시오.
- 배열의 맨 위에 있는 더 큰 할당되지 않은 블록은 그대로 유지된다.

Heap\_2.c 는 결정적이지 않지만 malloc () 및 free () 의 대부분의 표준 라이브러리 구현보다 더 효율적이다.

### Heap\_3.c

Heap\_3.c 는 단순히 표준 라이브러리 malloc () 및 free () 함수를 사용하지만 스케줄러를 일시적으로 일시 중단하여 호출을 안전하게 만든다. 구현은 Listing 74 와 같다.

힙 크기는 configTOTAL\_HEAP\_SIZE 의 영향을받지 않으며 대신 링커 구성에 의해 정의된다

```
.
void *pvPortMalloc( size_t xWantedSize )
{ void *pvReturn;

vTaskSuspendAll(); {
pvReturn = malloc( xWantedSize );
} xTaskResumeAll();

return pvReturn; }

void vPortFree( void *pv )
{ if( pv != NULL ) { vTaskSuspendAll();

{ free( pv );

} xTaskResumeAll();
}
}
```

**Listing 74 The heap\_3.c implementation**

# CHAPTER6

## TROUBLE SHOOTING

## 6.1 CHAPTER INTRODUCTION AND SCOPE

이 장은 FreeRTOS 를 처음 사용하는 사용자가 겪는 가장 일반적인 문제를 강조하는 것을 목표로 한다. 스택 문제는 수년 동안 가장 자주 요청되는 소스로 입증 되었기 때문에 주로 스택 오버플로 및 스택 오버플로 감지에 중점을 둔다. 그런 다음 간단히 FAQ 스타일로 다른 일반적인 오류, 가능한 원인 및 해결 방법에 대해 설명한다.

### **printf-stdarg.c**

스택 사용은 표준 C 라이브러리 함수, 특히 `sprintf ()`와 같은 IO 및 문자열 처리 함수가 사용될 때 특히 높게 나타날 수 있다. FreeRTOS 다운로드에는 표준 라이브러리 버전 대신 사용할 수 있는 최소한의 스택 효율적인 `sprintf ()` 버전이 들어있는 `printfstdarg.c` 라는 파일이 포함되어 있다. 대부분의 경우 `sprintf ()` 및 관련 함수를 호출하는 각 작업에 더 작은 스택을 할당 할 수 있다.

`Printf-stdarg.c` 는 오픈 소스이지만 타사 소유이므로 FreeRTOS 와 별도로 라이선스가 부여된다. 라이선스 조항은 소스 파일의 맨 위에 있다.

## 6.2 STACK OVERFLOW

FreeRTOS 는 스택 관련 문제를 트래핑하고 디버깅하는 데 도움이 되는 몇 가지 기능을 제공한다<sup>4</sup>.

### uxTaskGetStackHighWaterMark() API Function

각 작업은 자체 스택을 유지 관리하며 작업의 생성 시 전체 크기가 지정된다. uxTaskGetStackHighWaterMark ()는 태스크에 할당된 스택 공간을 오버플로우시키는 방법을 쿼리하는데 사용됩니다. 이 값을 스택 최고점이라고 한다.

```
unsigned portBASE_TYPE uxTaskGetStackHighWaterMark( xTaskHandle xTask );
```

Listing 75 The uxTaskGetStackHighWaterMark() API function prototype

Table 20 uxTaskGetStackHighWaterMark() parameters and return value

Parameter Name/Returned Value	Description
xTask	스택 최고 수위가 조회되는 태스크 (대상 태스크)의 핸들 - 태스크 핸들을 얻는 방법에 대한 정보는 xTaskCreate () API 함수의 pxCreatedTask 매개 변수를 참조하십시오.  태스크는 유효한 태스크 핸들 대신 NULL 을 전달하여 고유 한 스택 최고 워터 마크를 쿼리 할 수 있다.
Returned value	작업이 실제로 사용하는 스택의 양은 작업이 실행되고 인터럽트가 처리 될 때 늘어나고 줄어 든다. uxTaskGetStackHighWaterMark () 작업이 시작된 이후에 사용 가능한 최소 스택 공간의 양을 반환한다. 이것은 스택 사용량이 가장 큰 (가장 깊은) 값일 때 사용되지 않은 채로 남아있는 스택의 양이다. 상위 워터 마크가 0 에 가까울수록 스택 오버플로가 발생한다.

<sup>4</sup> DOS 는 분할된 메모리를 사용하기 때문에 불행히도 이러한 기능은 시뮬레이션된 DOS 환경에서 사용할 수 없다. 따라서 Open Watcom 예제를 사용하여 그 사용법을 보여줄 수는 없다.

## Run Time Stack Checking - Overview

FreeRTOS 는 2 개의 선택적 런타임 스택 검사 메커니즘을 포함한다. 이들은 FreeRTOSConfig.h 내의 configCHECK\_FOR\_STACK\_OVERFLOW 컴파일 시간 설정 상수에 의해 제어된다. 두 방법 모두 컨텍스트 전환을 수행하는 데 걸리는 시간이 길어진다.

스택 오버 플로우 후크 (또는 콜백)은 스택 오버 플로우를 감지했을 때 커널이 호출하는 함수이다. 스택 오버플로 후크 기능을 사용하려면 다음과 같이하십시오.

FreeRTOSConfig.h 에서 configCHECK\_FOR\_STACK\_OVERFLOW 를 1 또는 2 로 설정하십시오.

Listing 76 의 정확한 함수 이름과 프로토 타입을 사용하여 hook 함수의 구현을 제공한다.

```
void vApplicationStackOverflowHook( xTaskHandle *pxTask, signed portCHAR *pcTaskName );
```

### Listing 76 The stack overflow hook function prototype

스택 오버플로 후크는 스택 오류를 쉽게 트래킹하고 디버깅하도록 하기 위해 제공되지만 스택 오버플로 가 발생하면 실제로 복구 할 방법이 없다. 오버플로가 작업 이름을 손상시킬 수는 있지만 매개 변수는 해당 스택에서 오버플로 된 작업의 이름과 핸들 이름을 후크 기능에 전달한다. 스택 오버플로 후크는 인터럽트의 컨텍스트에서 호출 될 수 있다.

일부 마이크로 컨트롤러는 잘못된 메모리 액세스를 감지 할 때 오류 예외를 생성하며 커널이 오버플로 후크 기능을 호출 할 수 있는 기회를 갖기 전에 오류가 트리거 될 수 있다.

## Run Time Stack Checking - Method 1 configCHECK\_FOR\_STACK\_OVERFLOW 가

1 로 설정되면 방법 1 이 선택된다.

전체 실행 컨텍스트의 작업은 스왑 아웃 될 때마다 스택에 저장된다. 스택 사용량이 최고점에 도달하는 시간이될가능성이크다. configCHECK\_FOR\_STACK\_OVERFLOW 가 1 로 설정되면 커널은 컨텍스트 를 저장 한 후에 스택 포인터가 유효한 스택 공간 내에 있는지 확인한다. 스택 포인터가 유효한 범위를 벗어난 것으로 발견되면 스택 오버플로 후크가 호출된다. 방법 1 은 빠르게 실행되지만 컨텍스트 저장간에 발생하는 스택 오버플로를 놓칠 수 있다.

## Run Time Stack Checking - Method 2

방 법 2 는 방 법 1 에 대 해 이 미 설 명 한 사 항 에 대 해 추 가 점 검 을 수 행 한 다 . 이 는 configCHECK\_FOR\_STACK\_OVERFLOW 가 2 로 설정된 경우 선택된다.

작업이 생성되면 해당 스택은 알려진 패턴으로 채워진다. 방법 2 는이 패턴이 겹쳐 쓰여지지 않았는지 확인하기 위해 작업 스택 공간의 마지막 유효한 20 바이트를 탐색한다. 스택 오버플로 후크 함수는 20 바이트 중 하나가 예상 값에서 변경된 경우 호출된다.

방법 2 는 방법 1 처럼 빨리 실행되지 않지만 20 바이트 만 테스트되므로 상대적으로 빠르다. 모든 스택 오버플로우를잡을가능성은매우높지만, 일부는여전히누락될수있다.



## 6.3 OTHER COMMON SOURCES OF ERROR

### **Symptom: Adding a Simple Task to a Demo Causes the Demo to Crash**

작업을 만들려면 힙에서 메모리를 가져와야 한다. 데모 응용 프로그램 프로젝트 중 상당수는 데모 작업을 생성 할 수 있을 정도로 힙의 크기를 정확하게 나타낸다. 따라서 작업을 만든 후에 추가 작업, 대기열 또는 세마포를 추가하기에 충분한 힙이 남아있게 된다.

유틸리티 작업은 `vTaskStartScheduler ()`가 호출 될 때 자동으로 생성된다. `vTaskStartScheduler ()`는 유틸리티 작업을 작성하기에 충분한 힙 메모리가 남아 있지 않은 경우에만 리턴한다. 널 루프 포함 `[for (;;) ; ]` `vTaskStartScheduler ()` 호출 후 이 오류를 쉽게 디버깅 할 수 있다.

더 많은 작업을 추가하려면 힙 크기를 늘리거나 기존 데모 작업 중 일부를 제거하시오..

### **Symptom: Using an API Function Within an Interrupt Causes the Application to Crash**

API 함수의 이름이 "FromISR ()"로 끝나지 않으면 인터럽트 서비스 루틴 내에서 API 함수를 사용하지 마시오.

### **Symptom: Sometimes the Application Crashes within an Interrupt Service Routine**

가장 먼저 확인해야 할 것은 인터럽트가 스택 오버플로를 유발하지 않는다는 것이다.

방법 인터럽트가 정의와 포트 사이 컴파일러 사이에 다르다 사용되는 - 그래서 확인하는 두 번째 것은 데모에 대한 설명서 페이지에 설명 된 대로 인터럽트 서비스 루틴에서 사용되는 구문, 매크로 및 호출 규칙을 정확히 있다는 것이다, 데모에서 다른 인터럽트 서비스 루틴과 똑같이 보여준다.

응용 프로그램이 Cortex M3 에서 실행중인 경우 논리적으로 우선 순위가 높은 인터럽트를 나타내는 데 숫자 낮은 우선 순위 번호가 사용된다는 점을 고려하여 각 인터럽트에 할당 된 우선 순위를 고려해야 한다. 이것은 반 직관적으로 보일 수 있다. FreeRTOS API 함수를 사용하는 인터럽트를 `configMAX_SYSCALL_INTERRUPT_PRIORITY`에 정의 된 우선 순위보다 우연히 할당하는 것은 일반적인 실수이다.

### **Symptom: The Scheduler Crashes When Attempting to Start the First Task**

ARM7 마이크로 컨트롤러를 사용하는 경우 `vTaskStartScheduler ()`를 호출하기 전에 프로세서가 감독자 모드에 있는지 확인하시오. 이를 달성하는 가장 쉬운 방법은 `main ()`이 호출되기 전에 C 시작 코드

내에서 프로세서를 관리자 모드로 전환하는 것이다. 이것이 ARM7 데모 애플리케이션이 구성되는 방법이다.

프로세서가 감독자 모드에 있지 않으면 스케줄러가 시작되지 않는다.

### **Symptom: Critical Sections Do Not Nest Correctly**

taskENTER\_CRITICAL () 및 taskEXIT\_CRITICAL ()에 대한 호출 이외의 방법을 사용하여 마이크로 컨트롤러 인터럽트 활성화 비트 또는 우선 순위 플래그를 변경하지 마시오. 이러한 매크로는 호출 중첩 깊이를 계산하여 호출 중첩이 완전히 0으로 풀린 경우에만 인터럽트가 다시 활성화되도록 한다.

### **Symptom: The Application Crashes Even Before the Scheduler is Started**

컨텍스트 전환을 유발할 수 있는 인터럽트 서비스 루틴은 스케줄러가 시작되기 전에 실행되지 않아야 한다. 큐또는세마포어와주고받기를시도하는인터럽트서비스루틴에대해서도마찬가지이다. 컨텍스트 전환은 스케줄러가 시작될 때까지 발생할 수 없다.

많은 API 함수는 스케줄러가 시작되기 전에 호출 될 수 없다. vTaskStartScheduler ()가 호출 될 때까지는 API 사용을 작업, 대기열 및 세마포의 생성으로 제한하는 것이 가장 좋다.

### **Symptom: Calling API Functions While the Scheduler is Suspended Causes the Application to Crash**

스케줄러는 vTaskSuspendAll ()을 호출하여 일시 중지되고 xTaskResumeAll ()을 호출하여 재개 (일시 중단 해제)된다.

스케줄러가 일시 중단 된 동안 API 함수를 호출하지 마시오.

### **Symptom: The Prototype For pxPortInitialiseStack() Causes Compilation to Fail**

모든 포트에는 올바른 커널 헤더 파일이 빌드에 포함되도록 매크로를 정의해야 한다. pxPortInitialiseStack () 프로토 타입을 컴파일 할 때 발생하는 오류는 거의 확실하게 이 매크로가 사용되는 포트에 대해 잘못 설정되었다는 증상이다. 자세한 내용은 부록 4 :를 참조하십시오.

제공된 데모 프로젝트에서 사용중인 포트와 관련된 새로운 응용 프로그램을 기반으로 한다. 이렇게 하면 올바른 파일이 포함되고 올바른 컴파일러 옵션이 설정된다.

## **APPENDIX 1: BUILDING THE EXAMPLES**

이 책은 수많은 예제를 제공한다. 소스 코드는 함께 제공된다.

무료 Open Watcom IDE 에서 열 수있는 프로젝트 파일과 함께 .zip 파일을 생성 할 수 있다. 결과 실행 파 일은 Windows 명령 터미널 내에서 또는 무료 DOSBox DOS 에뮬레이터에서 실행될 수 있다. 도구 다운 로드 에 대해서는 <http://www.openwatcom.org> 및 <http://www.dosbox.com> 을 참조하시오.

**Ensure to include the 16bit DOS target options when installing the Open Watcom compiler!**

Open Watcom 프로젝트 파일은 모두 RTOSDemo.wpj 라고 하며 Examples \ Example0nn 디렉토리에 있다. 여기서 'nn'은 예제 번호이다.

DOS 는 FreeRTOS 에 대한 이상적인 목표와 거리가 멀고 예제 응용 프로그램은 진정한 실시간 특성으로 실행되지 않는다. DOS 는 사용자가 전문 하드웨어 또는 도구에 우선적으로 투자하지 않고도 예제를 실험 할 수 있기 때문에 간단하게 사용된다.

Open Watcom 디버거는 중요 섹션 내에 있는 코드를 단계별로 실행하는 경우에도 단계 작업간에 인터럽트를 실행할 수 있음을 참고하시오. 이것은 불행히도 컨텍스트 전환 프로세스를 단계적으로 수행하는 것을 불가능하게 만든다. 생성 된 실행 파일은 Open Watcom IDE 가 아닌 명령 프롬프트에서 실행하는 것이 가장 좋다.

## APPENDIX 2: THE DEMO APPLICATIONS

공식 FreeRTOS 포트에는 오류나 경고가 발생하지 않고 빌드해야 하는 데모 응용 프로그램이 함께 제공된다<sup>5</sup>. 데모 애플리케이션에는 여러 가지 용도가 있다.

1. 올바른 파일이 포함되고 올바른 컴파일러 옵션이 설정된 작업 및 사전 구성된 프로젝트의 예를 제공한다.
2. 최소한의 셋업이나 사전 지식으로 '즉시 사용 가능한' 실험을 가능하게 한다.
3. FreeRTOS API 를 시연한다.
4. 실제 응용 프로그램을 만들 수 있는 기반.

각 데모 프로젝트는 데모 디렉토리의 고유 한 디렉토리에 있다 (부록 3 참조). 디렉토리 이름은 데모 프로젝트와 관련된 포트를 나타낸다.

모든 데모 응용 프로그램에는 FreeRTOS.org 웹 사이트에서 호스팅되는 설명서 페이지도 함께 제공된다. 문서 페이지에는 FreeRTOS 디렉토리 구조에서 개별 데모 응용 프로그램을 찾는 데 필요한 정보가 들어 있다.

모든 데모 프로젝트는 Demo \ Common 디렉토리 트리에 있는 소스 파일에 정의된 태스크를 작성한다. 대부분은 Demo \ Common \ Minimal 디렉토리의 파일을 사용한다.

main.c 라는 파일이 각 프로젝트에 포함되어 있다. 여기에는 모든 데모 애플리케이션 작업이 생성되는 main () 함수가 포함되어 있다. 특정 데모 응용 프로그램의 기능에 대한 자세한 내용은 개별 main.c 파일의 주석을 참조하십시오.

---

<sup>5</sup> 이는 이상적인 시나리오이며 일반적으로 해당되지만 데모를 작성하는 데 사용되는 컴파일러의 버전에 따라 다르다. 업그레이드된 컴파일러는 전임자가 하지 않은 경우 경고를 생성할 수 있다.

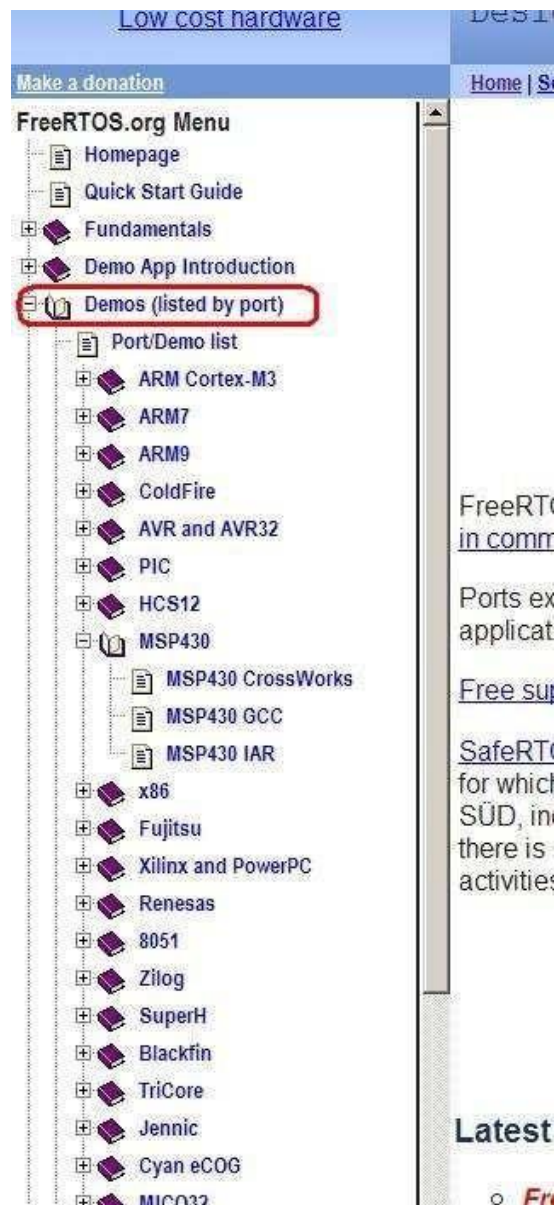


Figure 44 Locating the demo application documentation in the menu frame of the FreeRTOS.org WEB site

## APPENDIX 3: FREERTOS FILES AND DIRECTORIES

이 부록에서 설명하는 디렉토리 구조는 FreeRTOS.org 웹 사이트에서 다운로드 할 수 있는 .zip 파일에만 관련된다. 이 책과 함께 제공되는 예제는 약간 다른 조직을 사용한다.

FreeRTOS는 다음을 포함하는 단일 .zip 파일로 다운로드 된다.

핵심 FreeRTOS 소스 코드. 이것은 모든 포트에 공통적인 코드이다.

지원되는 각 마이크로 컨트롤러 및 컴파일러 조합에 대한 포트 계층.

지원되는 각 마이크로 컨트롤러와 컴파일러 조합에 대한 데모 응용 프로그램을 빌드하는 프로젝트 파일 또는 메이크 파일.

각 데모 응용 프로그램에 공통적인 데모 작업 세트. 이 데모 작업은 포트 관련 데모 프로젝트에서 참조된다.

zip 파일에는 Source 라는 소스와 Demo 라는 두 개의 최상위 디렉토리가 있다. 소스 디렉토리 트리 는 전체 FreeRTOS 커널 구현 - 공통 구성 요소와 포트 특정 구성 요소 모두 포함한다. 데모 디렉토리 트리에는 데모 애플리케이션 프로젝트 파일과 데모 작업을 정의하는 소스 파일 만 포함되어 있다.

```
FreeRTOS
|
|--Demo           데모응용프로그램소스및프로젝트가들어있다..
|
|--Source         실시간커널의구현을포함한다.
```

**Figure 45 The top level directories – Source and Demo**

핵심 FreeRTOS 소스 코드는 모든 마이크로 컨트롤러 포트에 공통적인 3 개의 C 파일에 포함되어 있다. 이것들은 queue.c, tasks.c 및 list.c 라고 불리며 Source 디렉토리 바로 아래에 위치 할 수 있다. 포트 특정 파일은 이동식 디렉토리 트리에 있으며 Source 디렉토리에 직접 위치한다.

croutine.c 라는 네 번째 옵션 소스 파일은 FreeRTOS 코 루틴 기능을 구현한다. 공동 루틴이 실제로 사용될 경우 빌드에 포함될 필요가 있다.

```
FreeRTOS
|
|--Demo           데모응용프로그램소스및프로젝트가들어있다..
|
|--Source         실시간커널의구현을포함한다.
|               |--tasks.c       세 가지 핵심 커널 파일 중 하나입니다.. 세
|--tasks.c       |--queue.c     가지 핵심 커널 파일 중 하나입니다..
|--queue.c       |--list.c      세 가지 핵심 커널 파일 중 하나입니다..
|--list.c
|--portable      모든 포트 특정 파일을 포함하는 하위 디렉토리.
```

**Figure 46 The three core files that implement the FreeRTOS kernel**

## Removing Unused Files

주 FreeRTOS .zip 파일에는 모든 포트 및 모든 데모 응용 프로그램 용 파일이 포함되어 있으므로 한 포트를 사용하는 데 필요한 것보다 더 많은 파일이 포함되어 있다. 데모 응용 프로그램 프로젝트 또는 사용되는 포트와 함께 제공되는 메이크 파일은 파일이 필요하고 삭제할 수 있는 참조로 사용할 수 있다.

'portable layer'는 FreeRTOS 커널을 특정 컴파일러와 아키텍처에 맞추는 코드이다. 포터블 레이어 소스 파일은 FreeRTOS \ Source \ Portable \ [compiler] \ [architecture] 디렉토리에 있다. 여기서 [compiler]는 사용되는 툴 체인이고 [architecture]는 사용되는 마이크로 컨트롤러 변형이다.

사용중인 도구 체인과 관련이 없는 FreeRTOS \ Source \ Portable 아래의 모든 하위 디렉터리는 FreeRTOS \ Source \ Portable \ MemMang 디렉터를 제외하고는 삭제할 수 있다.

FreeRTOS \ Source \ Portable \ [compiler] 아래에 있는 사용중인 마이크로 컨트롤러 변형과 관련이 없는 모든 하위 디렉터리는 삭제할 수 있다.

사용중인 데모 응용 프로그램과 관련이 없는 FreeRTOS \ Demo 아래의 모든 하위 디렉터리는 모든 데모 응용 프로그램에서 참조하는 파일이 들어있는 FreeRTOS \ Demo \ Common 디렉터를 제외하고는 삭제할 수 있다.

FreeRTOS \ Demo \ Common에는 하나의 데모 응용 프로그램에서 참조된 것보다 많은 파일이 포함되어 있으므로 원하는 경우 디렉토리를 thin 아낼 수도 있다.

## APPENDIX 4: CREATING A FREERTOS PROJECT

### Adapting One of the Supplied Demo Projects

모든 공식 FreeRTOS 포트에는 오류나 경고없이 빌드해야 하는 미리 구성된 데모 응용 프로그램이 함께 제공된다 (부록 2 참조). 기존 프로젝트 중 하나를 적용하여 새로운 프로젝트를 생성하는 것이 좋다. 이 렇게하면 프로젝트에 올바른 파일이 포함되고 올바른 컴파일러 옵션이 설정된다.

기존 데모 프로젝트에서 새 응용 프로그램을 시작하려면 :

1. 제공된 데모 프로젝트를 열고 예상대로 빌드하고 실행하는지 확인하십시오.
2. 데모 작업을 정의하는 소스 파일을 제거하십시오. Demo \ Common 디렉토리 트리에 있는 파일은 프로젝트 파일이나 makefile에서 제거할 수 있다.

3.prvSetupHardware () 이외의 main.c 에있는모든함수를삭제하시오.

4.FreeRTOSConfig.h 에서 configUSE\_IDLE\_HOOK, configUSE\_TICK\_HOOK 및 configCHECK\_FOR\_STACK\_OVERFLOW 가 모두 0 으로 설정되어 있는지 확인하시오. 이렇게 하면 링 커가 후크 기능을 찾지 못하게 됩니다. 필요한 경우 후크 기능을 나중에 추가 할 수 있다.

5.Listing 77 의 템플릿에서 새 main () 함수를 만든다.

6.프로젝트가 여전히 빌드되는지 확인하시오.

이 단계를 따르면 FreeRTOS 소스 파일을 포함하지만 기능을 정의하지 않는 프로젝트를 제공한다.

```
int main( void )
{
    /* 필요한하드웨어설정을수행하시오. */
    prvSetupHardware();

    /* --- 응용프로그램작업은여기에서만들수있다. */

    /* 생성된작업을시작하시오. */
    vTaskStartScheduler();

    /* 스케줄러를시작할충분한힘이없다면실행은여기에도달할것이다. */
    for(;;); return 0; }
```

**Listing 77 The template for a new main() function**

## Creating a New Project from Scratch

방금 언급했듯이 기존 프로젝트에서 새 프로젝트를 만드는 것이 좋다. 어떤 이유로 이것이 바람직하지 않은 경우 다음 단계를 사용하여 새 프로젝트를 만들 수 있다.

1.선택한도구체인을사용하여새로운빈프로젝트파일이나 makefile 을만든다.

2.표 21 에 설명 된 파일을 새로 작성한 프로젝트 또는 makefile 에 추가하시오.

3.기존 FreeRTOSConfig.h 파일을 프로젝트 디렉터리에 복사하시오.

4.프로젝트가 헤더 파일을 찾기 위해 검색 할 경로에 프로젝트 디렉토리와 FreeRTOS \ Source \ include 를 모두 추가하시오.

5.컴파일러 설정을 관련 데모 프로젝트 또는 makefile 에서 복사하시오. 특히 모든 포트에는 올바른 커널 헤더 파일이 빌드에 포함되도록 매크로를 설정해야 한다. 예를 들어 IAR 컴파일러를 사용하여 MegaAVR 을 대상으로하는 빌드는 IAR\_MEGA\_AVR 을 정의해야하며 GCC 컴파일러를 사용하여 ARM



Cortex M3 를 대상으로 하는 빌드는 GCC\_ARMCM3 을 정의해야한다. 정의는 FreeRTOS \ Source \ include \ portable.h 에서사용된다. 어떤정의를필요한지확실하지않으면검사할수있다.

Table 21 FreeRTOS source files to include in the project

File Location

tasks.c	FreeRTOS\Source	queue.c	FreeRTOS\Source	list.c
FreeRTOS\Source				

port.c FreeRTOS \ Source \ portable \ [compiler] \ [architecture] 여기서 [compiler]는 사용되는 컴파일러이고 [architecture]는 사용되는 마이크로 컨트롤러 변형이다.

port.x 일부 포트에서는 프로젝트에 어셈블리 파일을 포함해야 한다. 파일은 port.c 와 같은 디렉토리에 있다. 파일 이름 확장명은 사용중인 도구 체인에 따라 달라진다. x 는 실제 파일 이름 확장명으로바뀌야한다.

## Header Files

FreeRTOS API 를 사용하는 소스 파일은 "FreeRTOS.h"를 포함해야하며 "task.h", "queue.h"또는 "semphr.h"중 하나 인 API 함수의 프로토 타입이 포함 된 헤더 파일을 포함해야 한다.

## APPENDIX 5: DATA TYPES AND CODING STYLE GUIDE

### Data Types

FreeRTOS의 각 포트에는 고유한 portable.h 헤더 파일이 있으며 이 헤더 파일에는 사용되는 데이터 유형을 상세하게 설명하는 매크로 세트가 정의되어 있다. 모든 FreeRTOS 소스 코드 및 데모 응용 프로그램은 기본 C 데이터 유형을 직접 사용하지 않고 이러한 매크로 정의를 사용한다. 그러나 FreeRTOS를 사용하는 응용 프로그램이 동일한 작업을 수행해야 하는 이유는 절대적으로 없다. 응용 프로그램 작성자는 표 22에 정의된 각 매크로에 대해 실제 데이터 유형을 대체할 수 있다.

Table 22 Data types used by FreeRTOS

Macro or typedef used	Actual type
portCHAR	char
portSHORT	short
portLONG	long
portTickType	틱 수를 저장하고 블록 시간을 지정하는 데 사용된다.  portTickType은 FreeRTOSConfig.h 내의 configUSE_16_BIT_TICKS 설정에 따라 부호없는 16 비트 유형 또는 부호없는 32 비트 유형이 될 수 있다.  16 비트 유형을 사용하면 8 및 16 비트 아키텍처의 효율성을 크게 향상시킬 수 있지만 지정된 블록 시간 범위를 심각하게 제한한다. 32 비트 아키텍처에서 16 비트 유형을 사용하는 것은 이치에 맞지 않는다.
portBASE_TYPE	이것은 아키텍처에 가장 효율적인 유형으로 정의 될 것이다. 일반적으로 이것은 32 비트 아키텍처에서는 32 비트 유형, 16 비트 아키텍처에서는 16 비트 유형, 8 비트 아키텍처에서는 8 비트 유형이다.  portBASE_TYPE은 일반적으로 매우 제한된 범위의 값만 사용할 수 있는 반환 유형 및 부울에 사용된다.  일부 컴파일러는 모든 규정되지 않은 char 변수를 부호없는 변수로 만들고, 다른 컴파일러에서는 부호를 만든다. 이런 이유 때문에 FreeRTOS 소스 코드는

portCHAR 의 모든 사용을 부호가 있거나 부호없는 것으로 명시 적으로 규정한다.  
int 타입은 결코 사용되지 않는다 - 길고 짧음.

## Variable Names

변수는 해당 유형으로 미리 고정되어 있다. 'c'는 char, short 는 'l', long 은 'l'이고 다른 유형 (구조체, 작업 핸들, 대기열 핸들 등)은 'x'이다.

변수에 부호가 없으면 'u'접두사가 붙는다. 변수가 포인터 인 경우 접두사 'p'가 붙습니다. 그러므로 unsigned char 타입의 변수는 'uc'접두사가 붙고 char 타입 포인터 변수는 'pc'접두사가 붙는다.

## Function Names

함수는 접미사가 반환하는 유형과 그 안에 정의 된 파일을 접두어로 사용한다. 예 :

vTaskPrioritySet ()은 void 를 리턴하고 task.c 내에 정의된다.

xQueueReceive ()는 portBASE\_TYPE 유형의 변수를 반환하며 queue.c 내에 정의된다.

vSemaphoreCreateBinary ()는 void 를 반환하고 semphr.h 내에 정의됩니다. 파일 범위 (private) 함수의 접두어는 'prv'이다.

## Formatting

1 탭은항상 4 칸으로설정된다.

## Macro Names

대부분의 매크로는 대문자로 쓰여지고 매크로가 정의 된 위치를 나타내는 소문자로 시작된다. 표 23 은 접두어 목록을 제공한다..

Table 23	Macro
prefixes	
Location of macro definition	port (for example, portMAX_DELAY)
Prefix	portable.h
task (for example, taskENTER_CRITICAL())	task.h
pd (for example, pdTRUE)	projdefs.h
config (for example, configUSE_PREEMPTION)	FreeRTOSConfig.h
err (for example, errQUEUE_FULL)	projdefs.h

세마포어 API 는 거의 모든 매크로 집합으로 작성되지만 매크로 명명 규칙보다는 함수 명명 규칙을 따른 다.

표 24 에 정의 된 매크로는 FreeRTOS 소스에서 사용된다.

Table 24 Common macro

definitions

Macro		Value
pdTRUE	1	pdFALSE
0	pdPASS	1
pdFAIL	0	

### Rationale for Excessive Type Casting

FreeRTOS 소스 코드는 다양한 컴파일러로 컴파일 할 수 있다.이 컴파일러는 경고를 생성하는 방법과시기에 따라 서로 다른 단점이 있다. 특 히다른컴파일러는다른방법으로캐스팅을사용하기를원한다. 결과적으로 FreeRTOS 소스 코드에는 일반적으로 보장되는 것보다 더 많은 유형 캐스팅이 포함된다.

## APPENDIX 6: LICENSING INFORMATION

FreeRTOS 는 GNU 일반 공중 사용 허가서 (GPL)의 수정 된 버전에 따라 사용이 허가되며 해당 사용 허가서에 따라 상용 응용 프로그램에서 사용할 수 있다. 다음과 같은 경우 대체 및 선택적 상용 라이선스를 사용할 수 있다.

표 25 의 "오픈 소스 수정 된 GPL 라이선스"열에 명시된 요구 사항을 충족시킬 수 없다.

직접적인 기술 지원을 받기를 원한다.

개발에 대한 도움을 받고 싶다.

귀하는 보장과 배상을 요구한다.

Table 25 Open Source Vs Commercial License Comparison

	Open source modified GPL license	Commercial license
입니까?	Yes	No
응용 프로그램에서 사용할 수 있습니까?	Yes	Yes

은 로열티가 없습니까?	Yes	Yes
용 프로그램 코드를 오픈 소스 화해야 합니까?	No	No
RTOS 커널에 대한 변경 사항을 오픈 소스에 공개 합니까?	Yes	No
에서 FreeRTOS 를 사용하고 있음을 문 해야 합니까?	Yes	No
용 프로그램 사용자에게 FreeRTOS 소스 코드를 해야 합니까?	예 (보통 FreeRTOS.org 사 이트에 대한 WEB 링크로 충분합니다)	No
적으로 지원을받을 수 있습니까?	No	Yes
보증이 제공됩니까?	No	Yes

### Open Source License Details

FreeRTOS 소스코드는 GNU General Public License (GPL) 버전 2 에서예외적으로사용이허가된다. GPL 전문은 <http://www.freertos.org/license.txt> 에서볼수있습니다. 예외텍스트는아래에제공된다.

이에외는 FreeRTOS.org 웹사이트에게시된 API 를통해서만 FreeRTOS 를사용하는응용프로그램의소스코드가 폐쇄소스로유지되도록허용하므로전체응용프로그램을공개하지않아도상용응용프로그램에서 FreeRTOS 를 사용할수있다. 예외는 FreeRTOS 와독점제품을결합하려는경우에만사용할수있으며예외조항에명시된조건을준수해야한다.

### GPL Exception Text

예외 텍스트는 변경 될 수 있다. 가장 최신 버전은 FreeRTOS.org 웹 사이트를 참조하십시오.

#### Clause 1

FreeRTOS 를다른모듈과정적또는동적으로연결하는것은 FreeRTOS 를기반으로한결합된작업을만들고있다. 따라서 GNU 일반공중사용허가서의조건은전체적인조합을포괄한다.

예외적으로, FreeRTOS 의저작권자는 FreeRTOS 를 FreeRTOS API 인터페이스를통해서만 FreeRTOS 와통신하는독립모듈 과링크할수있는권한을제공하며, 이러한독립모듈의사용권조항에관계없이복사할수있다  
귀하가선택한조항에따라결과로생기는결합된저작물을배포할수있다:

1.결합된저작물의모든사본에는수신자에게사용된 FreeRTOS 버전과수신자가요청할경우 FreeRTOS 소스코드를제공 하라는제안서가상세히기재된서면진술서가첨부된다.

2. 결합된 작업 자체는 *RTOS*, 스케줄러, 커널 또는 관련 제품이 아닙니다.

3. 결합된 저작물 자체는 다른 소프트웨어 응용 프로그램과 링크하기 위한 라이브러리가 아닙니다.

*FreeRTOS* 소스코드는 *GNU General Public License* 및 이에 외 조항에 따라 귀하가 직접 배포할 수 있다. 독립 모듈은 *FreeRTOS* 에서 파생되거나 *FreeRTOS* 를 기반으로 하지 않는 모듈이다.

## **Clause 2**

*FreeRTOS.org* 는 *Richard Barry* 의 명시적인 허가 없이 런타임이나 컴파일 시간의 형태를 포함하여 어떤 경쟁 또는 비교 목적으로도 사용될 수 없습니다(업계의 표준이며 정보의 정확성을 보장하기 위한 것이다).