

---

# FreeRTOS 커널

## 개발자 안내서

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

## Table of Contents

FreeRTOS 커널 소개 .....	1
가치 제안 .....	1
용어 정의 .....	1
실시간 커널을 사용해야 하는 이유 .....	1
FreeRTOS 커널 기능 .....	3
라이선스 .....	3
함께 제공되는 소스 파일과 프로젝트 .....	3
FreeRTOS 커널 배포 .....	5
FreeRTOS 커널 배포의 이해 .....	5
FreeRTOS 커널 빌드 .....	5
FreeRTOSConfig.h .....	5
공식 FreeRTOS 커널 배포 .....	5
FreeRTOS 배포판의 상위 디렉터리 .....	6
모든 포트에 공통인 FreeRTOS 소스 파일 .....	6
포트별 FreeRTOS 소스 파일 .....	7
헤더 파일 .....	8
데모 애플리케이션 .....	8
FreeRTOS 프로젝트 생성 .....	9
처음부터 새로운 프로젝트 생성 .....	10
데이터 형식과 코딩 스타일 가이드 .....	11
변수 이름 .....	11
함수 이름 .....	11
형식 지정 .....	12
매크로 이름 .....	12
과잉 형식 변환에 대한 이론적 근거 .....	12
힙 메모리 관리 .....	13
전제 조건 .....	13
동적 메모리 할당과 FreeRTOS에 대한 적합성 .....	13
동적 메모리 할당 옵션 .....	14
메모리 할당 체계 예제 .....	14
heap_1 .....	14
heap_2 .....	15
heap_3 .....	16
heap_4 .....	17
heap_4에서 사용하는 배열의 시작 주소 설정 .....	18
heap_5 .....	18
vPortDefineHeapRegions() API 함수 .....	19
힙 관련 유틸리티 함수 .....	22
xPortGetFreeHeapSize() API 함수 .....	22
xPortGetMinimumEverFreeHeapSize() API 함수 .....	22
Malloc 실패 후크 함수 .....	23
작업 관리 .....	24
작업 함수 .....	24
최상위 작업 상태 .....	25
작업 생성 .....	26
xTaskCreate() API 함수 .....	26
작업 생성(예제 1) .....	27
작업 파라미터 사용(예제 2) .....	30
작업 우선순위 .....	32
시간 측정 및 틱 인터럽트 .....	32
우선순위 실험(예제 3) .....	34
Not Running 상태의 확장 .....	35
Blocked 상태 .....	36
Suspended 상태 .....	36

Ready 상태 .....	36
상태 전환 다이어그램의 완성 .....	36
Blocked 상태를 사용하여 지연 시간 생성(예제 4) .....	37
vTaskDelayUntil() API 함수 .....	41
vTaskDelayUntil()을 사용하기 위한 작업 예제의 변환(예제 5) .....	42
차단 작업과 비차단 작업의 결합(예제 6) .....	43
유휴 작업 및 유휴 작업 후크 .....	45
유휴 작업 후크 함수 .....	46
유휴 작업 후크 함수의 구현체에 대한 제한 사항 .....	46
유휴 작업 후크 함수의 정의(예제 7) .....	46
작업 우선순위의 변경 .....	48
vTaskPrioritySet() API 함수 .....	48
uxTaskPriorityGet() API 함수 .....	49
작업 우선순위의 변경(예제 8) .....	49
작업 삭제 .....	52
vTaskDelete() API 함수 .....	52
작업 삭제(예제 9) .....	53
스케줄링 알고리즘 .....	55
작업 상태 및 이벤트에 대한 요약 .....	55
스케줄링 알고리즘 구성 .....	56
시간 분할을 사용한 우선순위 선점형 스케줄링 .....	56
우선순위 선점형 스케줄링(시간 분할 미사용) .....	59
협력형 스케줄링 .....	60
대기열 관리 .....	63
대기열의 특성 .....	63
데이터 저장 .....	63
다수의 작업에서 액세스 .....	65
대기열 읽기에서 차단 .....	65
대기열 쓰기에서 차단 .....	65
다수의 대기열에서 차단 .....	66
대기열 사용 .....	66
xQueueCreate() API 함수 .....	66
xQueueSendToBack() 및 xQueueSendToFront() API 함수 .....	67
xQueueReceive() API 함수 .....	68
uxQueueMessagesWaiting() API 함수 .....	69
대기열에서 수신할 때 차단(예제 10) .....	70
다수의 소스에서 데이터를 수신 .....	73
대기열에 전송할 때 차단과 대기열을 통해 구조를 전송(예제 11) .....	74
대용량 또는 가변 크기의 데이터 작업 .....	79
포인터를 사용한 대기열 동작 .....	79
대기열을 사용해 여러 형식과 길이의 데이터를 전송 .....	81
다수의 대기열에서 수신 .....	84
대기열 집합 .....	84
xQueueCreateSet() API 함수 .....	85
xQueueAddToSet() API 함수 .....	86
xQueueSelectFromSet() API 함수 .....	86
대기열 집합 사용(예제 12) .....	87
더욱 현실적인 대기열 집합의 사용 사례 .....	90
대기열을 사용하여 사서함 생성 .....	92
xQueueOverwrite() API 함수 .....	93
xQueuePeek() API 함수 .....	94
소프트웨어 타이머 관리 .....	96
소프트웨어 타이머 콜백 함수 .....	96
소프트웨어 타이머의 속성과 상태 .....	96
소프트웨어 타이머의 주기 .....	96
일회성 타이머와 오토 리로드 타이머 .....	97
소프트웨어 타이머 상태 .....	97

소프트웨어 타이머의 컨텍스트 .....	98
RTOS 데몬(타이머 서비스) 작업 .....	98
타이머 명령 대기열 .....	98
데몬 작업 예약 .....	99
소프트웨어 타이머의 생성 및 시작 .....	101
xTimerCreate() API 함수 .....	101
xTimerStart() API 함수 .....	102
일회성 타이머와 오토 리로드 타이머의 생성(예제 13) .....	104
타이머 ID .....	106
vTimerSetTimerID() API 함수 .....	106
pvTimerGetTimerID() API 함수 .....	107
콜백 함수 파라미터와 소프트웨어 타이머 ID의 사용(예제 14) .....	107
타이머의 주기 변경 .....	109
xTimerChangePeriod() API 함수 .....	109
소프트웨어 타이머의 재설정 .....	112
xTimerReset() API 함수 .....	113
소프트웨어 타이머의 재설정(예제 15) .....	114
인터럽트 관리 .....	117
인터럽트 세이프 API .....	117
인터럽트 세이프 API를 별도로 사용하는 장점 .....	118
인터럽트 세이프 API를 별도로 사용하는 단점 .....	118
xHigherPriorityTaskWoken 파라미터 .....	118
portYIELD_FROM_ISR() 및 portEND_SWITCHING_ISR() 매크로 .....	120
인터럽트 처리의 위임 .....	120
동기화에 사용되는 이진 세마포어 .....	121
xSemaphoreCreateBinary() API 함수 .....	124
xSemaphoreTake() API 함수 .....	124
xSemaphoreGiveFromISR() API 함수 .....	125
이진 세마포어를 사용한 작업과 인터럽트의 동기화(예제 16) .....	126
예제 16에서 사용되는 작업 구현체 개선 .....	130
계수 세마포어 .....	134
xSemaphoreCreateCounting() API 함수 .....	136
계수 세마포어를 사용한 작업과 인터럽트의 동기화(예제 17) .....	136
작업을 RTOS 데몬 작업으로 위임 .....	138
xTimerPendFunctionCallFromISR() API 함수 .....	139
인터럽트 처리의 중앙 집중식 위임(예제 18) .....	140
인터럽트 서비스 루틴 내부에서 대기열 사용 .....	142
xQueueSendToFrontFromISR() 및 xQueueSendToBackFromISR() API 함수 .....	143
ISR에서 대기열을 사용할 때 고려해야 할 사항 .....	143
인터럽트 내부에서 대기열 전송 및 수신(예제 19) .....	144
인터럽트 중첩 .....	148
ARM Cortex-M 및 ARM GIC 사용자 .....	150
리소스 관리 .....	152
상호 배제 .....	154
임계 영역과 스케줄러의 일시 중지 .....	154
기본 임계 영역 .....	154
스케줄러의 일시 중지(잠금) .....	156
vTaskSuspendAll() API 함수 .....	156
xTaskResumeAll() API 함수 .....	157
뮤텍스(및 이진 세마포어) .....	158
xSemaphoreCreateMutex() API 함수 .....	158
세마포어를 사용하기 위한 vPrintString() 재작성(예제 20) .....	158
우선순위 역전 .....	161
우선순위 상속 .....	162
교착 상태(또는 치명적인 포용) .....	163
재귀적 뮤텍스 .....	164
뮤텍스 및 작업 스케줄링 .....	165

게이트키퍼 작업 .....	168
게이트키퍼 작업을 사용하기 위한 vPrintString() 재작성(예제 21) .....	169
이벤트 그룹 .....	173
이벤트 그룹의 특성 .....	173
이벤트 그룹, 이벤트 플래그 및 이벤트 비트 .....	173
EventBits_t 데이터 형식에 대한 기타 정보 .....	174
다수의 작업에서 액세스 .....	65
이벤트 그룹의 실제 사용 예 .....	174
이벤트 그룹을 사용한 이벤트 관리 .....	175
xEventGroupCreate() API 함수 .....	175
xEventGroupSetBits() API 함수 .....	175
xEventGroupSetBitsFromISR() API 함수 .....	176
xEventGroupWaitBits() API 함수 .....	177
이벤트 그룹 실험(예제 22) .....	180
이벤트 그룹을 사용한 작업 동기화 .....	184
xEventGroupSync() API 함수 .....	187
작업 동기화(예제 23) .....	188
작업 알림 .....	191
중재자 객체를 통한 통신 .....	191
작업 알림: 작업 직접 통신 .....	191
작업 알림의 이점과 제한 사항 .....	192
작업 알림의 제한 사항 .....	192
작업 알림 사용 .....	193
작업 알림 API 옵션 .....	193
xTaskNotifyGive() API 함수 .....	193
vTaskNotifyGiveFromISR() API 함수 .....	194
ulTaskNotifyTake() API 함수 .....	194
세마포어 대신에 작업 알림을 사용할 수 있는 방법 1(예제 24) .....	195
세마포어 대신에 작업 알림을 사용할 수 있는 방법 2(예제 25) .....	199
xTaskNotify() 및 xTaskNotifyFromISR() API 함수 .....	200
xTaskNotifyWait() API 함수 .....	202
주변 장치 드라이버에서 사용되는 작업 알림: UART 예제 .....	204
주변 장치 드라이버에서 사용되는 작업 알림: ADC 예제 .....	209
애플리케이션 내부에서 직접 사용되는 작업 알림 .....	211
개발자 지원 .....	217
configASSERT() .....	217
configASSERT() 정의 예제 .....	217
Tracealyzer .....	218
디버그 관련 후크(콜백) 함수 .....	221
런타임 및 작업 상태 정보 보기 .....	221
작업 런타임 통계 .....	221
런타임 통계 기록 .....	221
런타임 통계를 수집하도록 애플리케이션 구성 .....	222
uxTaskGetSystemState() API 함수 .....	222
vTaskList() 헬퍼 함수 .....	225
vTaskGetRunTimeStats() 헬퍼 함수 .....	226
런타임 통계 생성 및 표시, 유효 예제 .....	227
추적 후크 매크로 .....	229
사용 가능한 추적 후크 매크로 .....	229
추적 후크 매크로의 정의 .....	231
FreeRTOS 인식형 디버거 플러그인 .....	231
문제 해결 .....	233
단원 소개 및 범위 .....	233
인터럽트 우선순위 .....	233
스택 오버플로우 .....	234
uxTaskGetStackHighWaterMark() API 함수 .....	234
런타임 스택 검사에 대한 개요 .....	234

런타임 스택 검사를 위한 방법 1 .....	235
런타임 스택 검사를 위한 방법 2 .....	235
printf() 및 sprintf()의 부적절한 사용 .....	235
Printf-stdarg.c .....	236
기타 공통 오류 .....	236
증상: 단순 작업을 데모에 추가하면 데모가 충돌을 일으킵니다 .....	236
증상: 인터럽트에서 API 함수를 사용하면 애플리케이션이 충돌을 일으킵니다 .....	236
증상: 인터럽트 서비스 루틴에서 애플리케이션이 간혹 충돌을 일으킵니다 .....	236
증상: 스케줄러가 첫 번째 작업을 시작하려고 하면 충돌을 일으킵니다. ....	237
증상: 인터럽트가 갑자기 비활성화되거나, 혹은 임계 영역이 정확히 중첩되지 않습니다 .....	237
증상: 스케줄러가 시작되기 전에도 애플리케이션이 충돌을 일으킵니다. ....	237
증상: 스케줄러가 일시 중지된 상태에서, 혹은 임계 영역 내부에서 API 함수를 호출하면 애플리케이션이 충돌을 일으킵니다 .....	237

# FreeRTOS 커널 소개

FreeRTOS 커널은 Amazon의 오픈 소스 소프트웨어입니다.

FreeRTOS 커널은 마이크로 컨트롤러 또는 소형 마이크로 프로세서를 사용하는 심층 임베디드 실시간 애플리케이션에 매우 적합합니다. 이러한 유형의 애플리케이션은 대체로 엄격한(hard) 실시간 요건과 유연한(soft) 실시간 요건이 섞여 있습니다.

유연한 실시간 요건이란 기한이 명시되어 있지만 기한을 위반하더라도 시스템을 계속해서 사용할 수 있는 요건을 말합니다. 예를 들어 키 입력에 너무 느리게 반응하면 시스템이 응답하지 않을 수도 있지만 그렇다고 시스템을 실제로 사용하지 못하게 되는 것은 아닙니다.

엄격한 실시간 요건이란 기한이 명시되어 있으며, 이 기한을 위반할 경우 반드시 시스템 결함이 일어나는 요건을 말합니다. 예를 들어 운전석 에어백은 충돌 센서 입력에 너무 느리게 반응할 경우 득보다는 오히려 실이 많을 수 있습니다.

FreeRTOS 커널은 이렇게 엄격한 실시간 요건에 따라 임베디드 애플리케이션을 개발할 수 있는 실시간 커널(또는 실시간 스케줄러)입니다. 따라서 각각 실행되는 스레드를 수집하여 애플리케이션이 구성됩니다. 코어가 1개뿐인 프로세서에서는 스레드를 언제든지 한 번에 하나만 실행할 수 있습니다. 이때 커널은 애플리케이션 설계자가 각 스레드에 할당한 우선순위를 살펴보고 실행할 스레드를 결정합니다. 가장 단순한 예를 들자면, 애플리케이션 설계자는 엄격한 실시간 요건을 따르는 스레드에 높은 우선순위를, 그리고 유연한 실시간 요건을 따르는 스레드에 낮은 우선순위를 할당할 수 있습니다. 이렇게 하면 엄격한 실시간 요건을 따르는 스레드는 항상 유연한 실시간 요건을 따르는 스레드에 앞서 실행됩니다. 하지만 우선순위 할당에 대한 결정이 항상 이렇게 간단하지는 않습니다.

위에서 설명한 개념이 잘 이해되지 않는다고 해서 걱정할 필요 없습니다. 이번 안내서에서는 개념에 대해 자세하게 다루면서 특히 실시간 커널과 FreeRTOS 커널의 사용 방법을 이해할 수 있는 여러 가지 예를 들어 설명할 것입니다.

## 가치 제안

FreeRTOS 커널의 전례 없는 글로벌 성공은 매력적인 가치 제안에서 기인합니다. FreeRTOS 커널은 전문적으로 개발되어 엄격한 품질 관리와 함께 강력한 지원이 보장되며, 지적재산권의 모호성 없이 상업용 애플리케이션에서 고유의 소스 코드를 노출시키지 않고 무료로 사용할 수 있습니다. 별도의 비용 없이 FreeRTOS 커널을 사용해 제품을 출시할 수 있으며, 이미 수많은 사람들이 그렇게 하고 있습니다. 언제든지 추가 백업을 받고 싶거나, 혹은 법무 팀이 서면 보장 또는 면책을 추가로 요구할 경우에는 저렴하고 쉽게 상업용으로 업그레이드할 수 있는 경로가 있습니다. 원하는 시점에 상업용으로 업그레이드할 수 있는 경로가 있기 때문에 안심하고 사용할 수 있습니다.

## 용어 정의

FreeRTOS 커널에서는 각 실행 스레드를 작업라고 부릅니다. 임베디드 커뮤니티에서 용어에 대한 합의는 없었지만 스레드는 일부 응용 분야에서 더욱 구체적인 의미를 갖기도 합니다.

## 실시간 커널을 사용해야 하는 이유

커널을 사용하지 않고 유용한 임베디드 소프트웨어를 효과적으로 개발할 수 있는 기법들도 많습니다. 간단한 시스템을 개발한다면 이러한 기법들이 가장 적합한 솔루션이 될 수 있습니다. 더욱 복잡한 경우에는 커널을 사용하는 것이 바람직하지만 이러한 교차점이 발생하는 위치는 언제나 주관적일 수 밖에 없습니다.



작업 우선순위를 할당하면 애플리케이션이 처리 기한을 맞추는 데 효과적일 수 있지만 커널은 다음과 같이 비교적 불확실한 이점까지 실현하는 효과가 있습니다.

- 시간 정보의 추상화

커널은 실행 시간을 담당하여 시간과 관련된 API를 애플리케이션에게 제공합니다. 이로써 애플리케이션 코드의 구조가 더욱 단순화될 뿐만 아니라 전체 코드 크기도 줄어듭니다.

- 유지 관리/확장

시간 정보의 추상화는 모듈 간 상호의존성을 줄여서 제어와 예측이 가능한 방법으로 소프트웨어를 개선할 수 있습니다. 또한 커널이 시간을 담당하기 때문에 애플리케이션 성능이 기본 하드웨어의 변경에 대해 비교적 민감하게 반응하지 않습니다.

- 모듈성

작업은 독립된 모듈에서 이루어지며, 각 모듈마다 목적이 알기 쉽게 정의되어 있어야 합니다.

- 팀 개발

작업에는 알기 쉽게 정의된 인터페이스가 있어서 팀 개발이 더욱 쉽습니다.

- 손쉬운 테스트

작업이 깔끔한 인터페이스와 함께 알기 쉽게 정의된 독립 모듈이라면 작업에 대한 테스트도 따로 실행할 수 있습니다.

- 코드 재사용

뛰어난 모듈성과 낮은 상호의존성 덕분에 코드를 손쉽게 재사용할 수 있습니다.

- 효율성 향상

커널을 사용하면 소프트웨어가 완전히 이벤트 중심으로 바뀌기 때문에 발생하지 않은 이벤트까지 폴링하여 처리 시간을 낭비하지 않습니다. 처리해야 할 이벤트가 있을 때만 코드가 실행되기 때문입니다.

RTOS 틱 인터럽트를 처리하고, 작업 사이에 실행을 전환해야 한다면 효율성 향상에 배치되는 결과가 될 수도 있습니다. 하지만 RTOS를 사용하지 않는 애플리케이션들도 대체로 일정한 형태의 틱 인터럽트가 발생합니다.

- 유휴 시간

스케줄러가 시작되면 유휴 작업이 자동으로 생성됩니다. 이러한 유휴 작업은 실행할 애플리케이션 작업이 없을 때마다 실행됩니다. 유휴 작업은 예비 처리 용량을 측정하거나, 백그라운드 검사를 실행하거나, 단순히 프로세서를 저전력 모드로 전환할 때 사용할 수 있습니다.

- 전력 관리

RTOS를 사용해 효율성이 향상되면 프로세서가 더욱 오랜 시간 동안 저전력 모드를 유지할 수 있습니다.

유휴 작업이 실행될 때마다 프로세서를 저전력 모드로 전환하면 전력 소비가 크게 줄어듭니다. FreeRTOS 커널 역시 타이머 미작동 모드(tickless mode)가 있어서 프로세서가 오랜 시간 저전력 모드를 유지할 수 있습니다.

- 유연한 인터럽트 처리

인터럽트 처리를 애플리케이션 개발자가 생성한 작업 또는 FreeRTOS 데몬 작업에게 맡겨 인터럽트 핸들러를 매우 짧게 작성할 수 있습니다.

- 혼합 처리 요건

설계 패턴이 간단할 때는 애플리케이션 내에서 주기적/지속적/이벤트 중심 처리 요건을 혼합하여 해결할 수 있습니다. 또한 작업 및 인터럽트 우선순위를 적절하게 선택하여 엄격한 실시간 요건과 유연한 실시간 요건을 모두 충족하는 것도 가능합니다.

## FreeRTOS 커널 기능

FreeRTOS 커널은 다음과 같은 표준 기능을 지원합니다.

- 선제적 또는 협력적 운영
- 매우 유연한 작업 우선순위 할당
- 유연하고 빠르면서 무겁지 않은 작업 알림 메커니즘
- 대기열
- 이진 세마포어
- 계수 세마포어
- 뮤텝스
- 재귀적 뮤텝스
- 소프트웨어 타이머
- 이벤트 그룹
- 틱 후크 함수
- 유틸리티 후크 함수
- 스택 오버플로우 검사
- 추적 레코딩
- 작업 런타임 통계 수집
- 상업용 라이선스 및 지원(선택 사항)
- 최대 인터럽트 중첩 모델(일부 아키텍처에 한함)
- 저전력 애플리케이션을 위한 타이머 미작동(tickless) 기능
- 해당되는 경우 소프트웨어 관리 방식의 인터럽트 스택(RAM을 절약하는 데 효과적)

## 라이선스

FreeRTOS 커널은 MIT 라이선스 약관에 따라 사용자에게 제공됩니다.

## 함께 제공되는 소스 파일과 프로젝트

소스 코드와 사전 구성된 프로젝트 파일, 그리고 모든 예제를 위한 전체 빌드 지침이 압축 파일(zip)로 함께 제공됩니다. 문서와 함께 사본을 받지 못했다면 <http://www.FreeRTOS.org/Documentation/code>에서 압축 파일을 다운로드할 수 있습니다. 압축 파일에 포함된 FreeRTOS 커널은 최신 버전이 아닐 수도 있습니다.

이번 문서에 포함된 스크린샷은 Microsoft Windows 환경에서 FreeRTOS Windows 포트를 사용해 예제를 실행하면서 가져온 것입니다. FreeRTOS Windows 포트를 사용하는 프로젝트는 Visual Studio의 무료 Express 에디션(<https://www.visualstudio.com/vs/community/>에서 다운로드 가능)을 사용해 빌드하도록 사전 구성되어 있습니다. FreeRTOS Windows 포트가 평가, 테스트 및 개발 플랫폼으로서 편리하기는 하지만 실제로 실시간 동작을 나타내지는 않습니다.

# FreeRTOS 커널 배포

FreeRTOS 커널은 zip 파일 아카이브 1개로 배포되며, 여기에는 공식 FreeRTOS 커널 포트와 사전 구성된 데모 애플리케이션들이 저장되어 있습니다.

## FreeRTOS 커널 배포의 이해

FreeRTOS 커널은 빌드할 수 있는 컴파일러 종류가 약 20가지에 달하며, 실행 가능한 프로세서 아키텍처도 30가지가 넘습니다. 지원되는 컴파일러와 프로세서 조합을 각각 FreeRTOS 포트로 간주하면 됩니다.

## FreeRTOS 커널 빌드

FreeRTOS를 베어 메탈 애플리케이션에게 멀티태스킹 기능을 제공하는 라이브러리라고 생각할 수 있습니다.

FreeRTOS는 C 소스 파일 세트로 공급됩니다. 일부 소스 파일은 모든 포트에게 공통이지만 일부는 특정 포트 전용인 것도 있습니다. FreeRTOS API를 애플리케이션에 사용하려면 소스 파일을 빌드하여 프로젝트에 포함시키면 됩니다. 이를 위해 데모 애플리케이션마다 공식 FreeRTOS 포트가 함께 제공됩니다. 데모 애플리케이션은 소스 파일과 헤더 파일을 정확하게 빌드할 수 있도록 사전 구성되어 있습니다.

데모 애플리케이션은 사전 구성된 그대로 빌드해야 합니다. 데모가 공개된 이후 빌드 도구에서 변경할 경우 문제를 일으킬 수 있습니다. 자세한 내용은 이번 주제에서 뒷부분에 나오는 데모 애플리케이션을 참조하십시오.

## FreeRTOSConfig.h

FreeRTOS는 FreeRTOSConfig.h라고 하는 헤더 파일에서 구성됩니다.

FreeRTOSConfig.h는 특정 애플리케이션의 용도에 맞춰 FreeRTOS를 구성하는 데 사용됩니다. 예를 들어 FreeRTOSConfig.h에는 협력형 스케줄링 알고리즘과 선점형 스케줄링 알고리즘 중에서 무엇을 사용할지 정의하는 설정인 configUSEPREEMPTION 같은 상수가 포함되어 있습니다. 그 밖에 애플리케이션 고유의 정의도 포함되어 있어서 FreeRTOS 소스 코드가 저장된 디렉터리가 아니고 빌드하는 애플리케이션에 속한 디렉터리에 위치해야 합니다.

데모 애플리케이션은 모든 FreeRTOS 포트에 제공되며, 이렇게 제공되는 데모 애플리케이션마다 FreeRTOSConfig.h 파일이 포함되어 있습니다. 따라서 처음 시작할 때 FreeRTOSConfig.h 파일을 생성할 필요 없습니다. 다만, 사용 중인 FreeRTOS 포트에 제공되는 데모 애플리케이션의 FreeRTOSConfig.h 파일을 사용해 용도에 맞게 구성하는 것이 좋습니다.

## 공식 FreeRTOS 커널 배포

FreeRTOS는 zip 파일 1개로 배포됩니다. 이 zip 파일에는 모든 FreeRTOS 포트에 공통인 소스 코드와 모든 FreeRTOS 데모 애플리케이션에 공통인 프로젝트 파일이 저장되어 있습니다. 그 밖에 선택 항목으로 FreeRTOS+ 에코시스템 구성요소와 FreeRTOS+ 에코시스템 데모 애플리케이션도 있습니다.

FreeRTOS 배포판의 파일 수에 대해서는 걱정할 필요 없습니다. 어떤 애플리케이션이든 1개에 필요한 파일 수가 매우 적습니다.

## FreeRTOS 배포판의 상위 디렉터리

FreeRTOS 배포판의 상위 디렉터리와 차상위 디렉터리는 다음과 같습니다.

FreeRTOS

```
| |
| |─Source 디렉터리: FreeRTOS 소스 파일이 저장됩니다.
| |
| |─Demo 디렉터리: 포트별로 사전 구성된 FreeRTOS 데모 프로젝트가 저장됩니다.
```

FreeRTOS-Plus

```
|
|─Source 디렉터리: 일부 FreeRTOS+ 에코시스템 구성요소에 필요한 소스 코드가 저장됩니다.
|
|─Demo 디렉터리: FreeRTOS+ 에코시스템 구성요소에 필요한 데모 프로젝트가 저장됩니다.
```

zip 파일에는 FreeRTOS 소스 파일, 모든 FreeRTOS 데모 프로젝트 및 모든 FreeRTOS+ 데모 프로젝트의 복사본이 하나만 저장되어 있습니다. FreeRTOS 소스 파일은 FreeRTOS/Source 디렉터리에 있습니다. 소스 파일은 디렉터리 구조가 바뀔 경우 빌드되지 않을 수도 있습니다.

## 모든 포트에 공통인 FreeRTOS 소스 파일

주요 FreeRTOS 소스 코드는 모든 FreeRTOS 포트에 공통인 C 파일 2개에 저장되어 있습니다. 두 파일의 이름은 tasks.c와 list.c이며, FreeRTOS/Source 디렉터리에 있습니다. 다음 소스 파일도 동일한 디렉터리에 있습니다.

- queue.c

queue.c는 대기열과 세마포어 서비스를 제공하며, 거의 항상 필요합니다.

- timers.c

timers.c는 소프트웨어 타이머 기능을 제공합니다. 이 소스 파일은 소프트웨어 타이머를 사용하는 경우에 한해 빌드에 추가해야 합니다.

- eventgroups.c

eventgroups.c는 이벤트 그룹 기능을 제공합니다. 이 소스 파일은 이벤트 그룹을 사용하는 경우에 한해 빌드에 추가해야 합니다.

- croutine.c

croutine.c는 FreeRTOS 코루틴 기능을 구현합니다. 이 소스 파일은 코루틴을 사용하는 경우에 한해 빌드에 추가해야 합니다. 코루틴은 초소형 마이크로컨트롤러에서 사용할 목적으로 만들어졌습니다. 하지만 지금은 거의 사용되지 않기 때문에 유지하는 수준이 다른 FreeRTOS 기능과 동일하지는 않습니다. 이번 안내서에서는 코루틴에 대해서 설명하지 않습니다.

FreeRTOS

```
|
```

- └─Source
  - |
  - ├─tasks.c FreeRTOS 소스 파일 - 항상 필요
  - ├─list.c FreeRTOS 소스 파일 - 항상 필요
  - ├─queue.c FreeRTOS 소스 파일 - 거의 항상 필요
  - ├─timers.c FreeRTOS 소스 파일 - 선택 사항
  - ├─eventgroups.c FreeRTOS 소스 파일 - 선택 사항
  - └─croutine.c FreeRTOS 소스 파일 - 선택 사항

여러 프로젝트에 동일한 이름의 파일들이 이미 포함되어 있기 때문에 파일 이름으로 인한 네임스페이스 충돌이 일어날 수도 있습니다. 하지만 파일 이름을 변경하면 FreeRTOS를 사용하는 수많은 프로젝트를 비롯해 자동화 도구나 IDE 플러그인과의 호환성을 떨어뜨려 문제가 될 수 있습니다.

## 포트별 FreeRTOS 소스 파일

FreeRTOS 포트별 소스 파일은 FreeRTOS/Source/portable 디렉터리에 있습니다. portable 디렉터리는 컴파일러와 프로세서 아키텍처의 순으로 계층을 이루고 있습니다.

FreeRTOS를 'architecture' 아키텍처의 프로세서에서 'compiler' 컴파일러를 사용해 실행하는 경우에는 주요 FreeRTOS 소스 파일 외에도 FreeRTOS/Source/portable/[compiler]/[architecture] 디렉터리에 있는 파일들까지 빌드해야 합니다.

2장 힙 메모리 관리에서도 설명하겠지만 FreeRTOS 역시 힙 메모리 할당을 이식 계층에 포함되는 것으로 간주합니다. V9.0.0 이전 버전의 FreeRTOS를 사용하는 프로젝트에서는 힙 메모리 관리자가 필요합니다. FreeRTOS V9.0.0부터는 configSUPPORTDYNAMICALLYALLOCATION이 FreeRTOSConfig.h에서 1로 설정되거나, 혹은 configSUPPORTDYNAMICALLYALLOCATION을 정의하지 않은 경우에만 힙 메모리 관리자가 필요합니다.

FreeRTOS는 힙 할당을 위한 5가지 체계 예제를 제공합니다. 5가지 체계의 이름은 heap1 ~ heap5이며, 각 소스 파일 heap1.c부터 heap5.c를 통해 실행됩니다. 힙 할당 체계 예제는 FreeRTOS/Source/portable/MemMang 디렉터리에 있습니다. 동적 메모리 할당을 사용하도록 FreeRTOS를 구성한 경우에는 애플리케이션이 또 다른 구현 방법을 제공하지 않는 한 프로젝트의 5가지 소스 파일 중 1개를 빌드해야 합니다.

다음 그림은 FreeRTOS 디렉터리 트리의 포트별 소스 파일을 나타낸 것입니다.

FreeRTOS

- |
- └─Source
  - |
  - └─portable 디렉터리: 포트별 소스 파일이 모두 저장됩니다.
    - |
    - ├─MemMang 디렉터리: 힙 할당 소스 파일 5개가 저장됩니다.
    - |
    - └─[compiler 1] 디렉터리: 컴파일러 1 전용 포트 파일이 저장됩니다.

```
| |
| ├──[architecture 1]: 컴파일러 1 아키텍처 1 포트에 필요한 파일이 저장됩니다.
| ├──[architecture 2]: 컴파일러 1 아키텍처 2 포트에 필요한 파일이 저장됩니다.
| └──[architecture 3]: 컴파일러 1 아키텍처 3 포트에 필요한 파일이 저장됩니다.
|
| └──[compiler 2] 디렉터리: 컴파일러 2 전용 포트 파일이 저장됩니다.
|
| ├──[architecture 1]: 컴파일러 2 아키텍처 1 포트에 필요한 파일이 저장됩니다.
| ├──[architecture 2]: 컴파일러 2 아키텍처 2 포트에 필요한 파일이 저장됩니다.
| └──[etc.]
```

#### 포함 경로

FreeRTOS는 컴파일러의 포함 경로에 다음과 같이 디렉터리 3개를 추가해야 합니다.

1. 주요 FreeRTOS 헤더 파일에 대한 경로로 항상 FreeRTOS/Source/include입니다.
2. 사용 중인 FreeRTOS 포트 전용 소스 파일에 대한 경로입니다. 위에서도 언급했지만 이 파일에 대한 경로는 FreeRTOS/Source/portable/[compiler]/[architecture]입니다.
3. FreeRTOSConfig.h 헤더 파일에 대한 경로입니다.

## 헤더 파일

FreeRTOS API를 사용하는 소스 파일에는 'FreeRTOS.h'가 추가되어야 하고, 그 뒤를 이어 'task.h', 'queue.h', 'semphr.h', 'timers.h', 'eventgroups.h' 등 사용할 API 함수 프로토타입이 포함된 헤더 파일이 나와야 합니다.

## 데모 애플리케이션

FreeRTOS 포트는 각각 데모 애플리케이션이 1개 이상 함께 제공됩니다. 일부는 오래 전 데모이지만 모두 오류나 경고 없이 빌드되어야 하며, 데모 공개 이후 빌드 도구에서 변경할 경우 간혹 문제를 일으킬 수도 있습니다.

Linux 사용자를 위한 참고 사항: FreeRTOS는 Windows 호스트에서 개발 및 테스트됩니다. 데모 프로젝트가 Linux 호스트에서 빌드된 경우 빌드 오류가 발생하기도 합니다. 빌드 오류는 파일 이름을 참조할 때 사용하는 대/소문자나 파일 경로의 슬래시 문자 방향에서 기인하는 경우가 대부분입니다. 이러한 오류를 알려려면 FreeRTOS 문의 양식(<http://www.FreeRTOS.org/contact>)을 사용해 주시기 바랍니다.

데모 애플리케이션의 목적은 다음과 같습니다.

- 필요한 파일을 추가하고, 필요한 컴파일러 옵션을 설정하여 유효한 사전 구성 프로젝트 예를 제공합니다.
- 최소의 설정 또는 사전 지식으로 즉석에서 실험합니다.
- FreeRTOS API의 사용 방법을 설명합니다.
- 실제 애플리케이션을 개발할 수 있는 기반을 제공합니다.

각 데모 프로젝트는 FreeRTOS/Demo 디렉터리 아래 전용 하위 디렉터리에 있습니다. 하위 디렉터리 이름은 해당 데모 프로젝트와 관련된 포트를 가리킵니다.

모든 데모 애플리케이션은 FreeRTOS.org 웹사이트의 웹 페이지에도 설명되어 있습니다. 웹 페이지에 기록된 정보는 다음과 같습니다.

- FreeRTOS 디렉터리 구조에서 각 데모에 따른 프로젝트 파일을 찾는 방법
- 프로젝트가 사용하도록 구성된 하드웨어
- 데모 실행을 위한 하드웨어 설정 방법
- 데모 빌드 방법
- 데모의 정상적인 동작 방식

모든 데모 프로젝트는 공통적인 데모 작업의 하위 집합을 생성하며, 이렇게 구현된 파일은 FreeRTOS/Demo/Common/Minimal 디렉터리에 저장됩니다. 공통 데모 작업은 순전히 FreeRTOS API의 사용 방법을 설명하기 위한 것일 뿐, 특별히 유용한 기능을 제공하지는 않습니다.

최근 데모 프로젝트일수록 초보자를 위한 '블링키' 프로젝트를 빌드할 수 있습니다. 블링키 프로젝트는 매우 기초적입니다. 일반적으로 이 프로젝트는 작업 2개와 대기열 1개만 생성합니다.

모든 데모 프로젝트에는 main.c라고 하는 파일이 포함되어 있습니다. 이 파일에는 main() 함수가 포함되어 있으며, 데모 애플리케이션 작업이 모두 여기에서 생성됩니다. 데모별 정보는 각 main.c 파일의 주석을 참조하십시오.

다음 그림은 FreeRTOS/Demo 디렉터리 계층을 나타낸 것입니다.

FreeRTOS

|

└─ Demo 디렉터리: 데모 프로젝트가 모두 저장됩니다.

|

└─ [Demo x]: 데모 'x'를 빌드하는 프로젝트 파일이 저장됩니다.

|

└─ [Demo y]: 데모 'y'를 빌드하는 프로젝트 파일이 저장됩니다.

|

└─ [Demo z]: 데모 'z'를 빌드하는 프로젝트 파일이 저장됩니다.

|

└─ Common: 모든 데모 애플리케이션에서 빌드된 파일이 저장됩니다.

## FreeRTOS 프로젝트 생성

모든 FreeRTOS 포트는 사전 구성되어 오류나 경고 없이 빌드되어야 하는 데모 애플리케이션이 1개 이상 함께 제공됩니다. 새로운 프로젝트는 이러한 기존 프로젝트 중 하나를 용도에 맞게 조정하여 생성하는 것이 좋습니다. 그러면 프로젝트에 필요한 파일을 추가하고, 필요한 인터럽트 핸들러를 설치하고, 필요한 컴파일러 옵션을 설정할 수 있기 때문입니다.

기존 데모 프로젝트에서 새로운 애플리케이션을 시작하는 방법은 다음과 같습니다.

1. 함께 제공되는 데모 프로젝트를 열어 정상적으로 빌드 및 실행되는지 확인합니다.
2. 데모 작업을 정의하는 소스 파일을 삭제합니다. Demo/Common 디렉터리에 위치한 파일은 무엇이든 프로젝트에서 삭제할 수 있습니다.



3. 목록 1과 같이 prvSetupHardware()와 vTaskStartScheduler()를 제외하고 main()에 속한 모든 함수 호출을 삭제합니다.
4. 프로젝트가 계속해서 빌드되는지 확인합니다.

위 단계를 마치면 필요한 FreeRTOS 소스 파일이 포함된 프로젝트가 생성됩니다. 단, 이 프로젝트에는 아무런 기능도 정의되어 있지 않습니다.

```
int main( void )
{
    /* Perform any hardware setup necessary. */
    prvSetupHardware();

    /* --- APPLICATION TASKS CAN BE CREATED HERE --- */

    /* Start the created tasks running. */
    vTaskStartScheduler();

    /* Execution will only reach here if there was insufficient heap to start the scheduler. */
    for( ;; );

    return 0;
}
```

## 처음부터 새로운 프로젝트 생성

앞에서도 얘기했지만 새로운 프로젝트는 기존 데모 프로젝트에서 생성하는 것이 좋습니다. 하지만 이것의 여의치 않을 때는 다음 절차에 따라 새로운 프로젝트를 생성할 수 있습니다.

1. 선택한 도구 체인을 사용해 FreeRTOS 소스 파일이 아직 없는 프로젝트를 새롭게 생성합니다.
2. 새로운 프로젝트를 빌드한 후 대상 하드웨어에 다운로드하여 실행할 수 있는지 확인합니다.
3. 확인하여 유효한 프로젝트라고 확신하는 경우에만 표 1에 나열된 FreeRTOS 소스 파일들을 프로젝트에 추가합니다.
4. 사용 중인 포트에 제공되는 데모 프로젝트의 FreeRTOSConfig.h 헤더 파일을 프로젝트 디렉터리에 복사합니다.
5. 프로젝트가 헤더 파일을 찾기 위해 검색하는 경로에 다음 디렉터리를 추가합니다.

- FreeRTOS/Source/include
- FreeRTOS/Source/portable/[compiler]/[architecture] (여기에서  
[compiler]와 [architecture]는 선택한 포트와 맞아야 합니다)
- FreeRTOSConfig.h 헤더 파일이 저장된 디렉터리

1. 관련 데모 프로젝트에서 컴파일러 설정을 복사합니다.
2. 필요한 FreeRTOS 인터럽트 핸들러를 모두 설치합니다. 사용 중인 포트에 관한 웹 페이지와 사용 중인 포트에 제공되는 데모 프로젝트를 참조하십시오.

다음 표는 프로젝트에 추가되는 FreeRTOS 소스 파일을 나열한 것입니다.

File	위치
tasks.c	FreeRTOS/Source
queue.c	FreeRTOS/Source
list.c	FreeRTOS/Source
timers.c	FreeRTOS/Source
eventgroups.c	FreeRTOS/Source
모든 C 및 어셈블러 파일	FreeRTOS/Source/portable/[compiler]/[architecture]
heapn.c	FreeRTOS/Source/portable/MemMang, 여기에서 n은 1, 2, 3, 4 또는 5입니다. 이 파일은 FreeRTOS V9.0.0부터 선택 사항입니다.

V9.0.0 이전 버전의 FreeRTOS를 사용하는 프로젝트는 heapn.c 파일 중 1개를 빌드해야 합니다. FreeRTOS V9.0.0부터는 configSUPPORTDYNAMICALLYALLOCATION이 FreeRTOSConfig.h에서 1로 설정되거나, 혹은 configSUPPORTDYNAMICALLYALLOCATION을 정의하지 않은 경우에만 heapn.c 파일이 필요합니다. 자세한 내용은 2장 힙 메모리 관리 단원을 참조하십시오.

## 데이터 형식과 코딩 스타일 가이드

FreeRTOS의 각 포트에는 고유한 portmacro.h 헤더 파일이 있으며, 여기에는 무엇보다 포트에 따른 두 가지 데이터 형식인 TickType과 BaseType에 대한 정의가 포함되어 있습니다.

다음 표는 FreeRTOS에서 사용되는 데이터 형식을 나열한 것입니다.

일부 컴파일러는 한정되지 않은 문자 변수를 모두 부호 없음(unsigned)으로 지정하지만 부호 있음(signed)으로 지정하는 컴파일러도 있습니다. 이러한 이유 때문에 문자 변수에 ASCII 문자가 저장되거나, 혹은 문자 변수를 가리키는 포인터가 문자열을 가리키는 데 사용되는 경우를 제외하고 FreeRTOS 소스 코드는 'signed' 또는 'unsigned'를 사용해 모든 문자 변수의 사용을 명시적으로 한정합니다.

일반적인 int 형식은 절대로 사용되지 않습니다.

## 변수 이름

변수는 형식에 따라 접두사가 첨부됩니다. 예를 들어 문자일 때는 'c', int16t(short)일 때는 's', int32t(long)일 때는 'l', BaseType과 기타 비표준 형식(구조, 작업 핸들, 대기열 핸들 등)일 때는 'x'가 첨부됩니다.

변수가 부호 없음으로 지정되면 접두사 'u'가 첨부됩니다. 변수가 포인터이면 접두사 'p'가 첨부됩니다. 예를 들어 uint8t 형식의 변수는 접두사 'uc'가, 그리고 문자를 가리키는 포인터 형식의 변수는 접두사 'pc'가 첨부됩니다.

## 함수 이름

함수는 반환되는 데이터 형식과 정의되는 파일에 따라 접두사가 결정됩니다. 예를 들면 다음과 같습니다.

- vTaskPrioritySet() 함수는 보이드를 반환하고 task.c에서 정의됩니다.
- xQueueReceive() 함수는 BaseType\_t 형식의 변수를 반환하고 queue.c에서 정의됩니다.

- pvTimerGetTimerID() 함수는 보이드를 가리키는 포인터를 반환하고 timers.c에서 정의됩니다.

파일 범위(private) 함수는 접두사 'prv'가 첨부됩니다.

## 형식 지정

탭 한 번은 항상 네 칸으로 동일하게 설정됩니다.

## 매크로 이름

대부분 매크로는 대문자로 작성되며, 매크로의 정의 위치를 가리키는 접두사가 소문자로 첨부됩니다.

다음 표는 매크로 접두사를 나열한 것입니다.

접두사	매크로 정의 위치
port(예: portMAXDELAY)	portable.h 또는 portmacro.h
task(예: taskENTERCRITICAL())	task.h
pd(예: pdTRUE)	projdefs.h
config(예: configUSEPREEMPTION)	FreeRTOSConfig.h
err(예: errQUEUEFULL)	projdefs.h

세마포어 API는 거의 모두 매크로 집합으로 작성되지만 명명 규칙은 매크로보다는 함수를 따릅니다.

다음 표는 FreeRTOS 소스 코드에서 사용되는 매크로를 나열한 것입니다.

매크로	값
pdTRUE	1
pdFALSE	0
pdPASS	1
pdFAIL	0

## 과잉 형식 변환에 대한 이론적 근거

FreeRTOS 소스 코드는 여러 가지 다양한 컴파일러를 사용한 컴파일이 가능하지만 사용되는 컴파일러 모두 경고를 생성하는 방식과 시점이 다릅니다. 특히 컴파일러에 따라 형식 변환을 사용하는 방식도 여러 가지입니다. 결과적으로 FreeRTOS 소스 코드에는 일반적으로 보증되는 것보다 더욱 많은 형식 변환이 포함되어 있습니다.

# 힙 메모리 관리

V9.0.0부터는 힙 메모리를 FreeRTOS 애플리케이션에 완전히 정적으로 할당할 수 있습니다. 이 말은 힙 메모리 관리자를 추가할 필요가 없다는 것을 의미합니다.

이번 단원에서 다루는 내용은 다음과 같습니다.

- FreeRTOS의 RAM 할당 시점
- FreeRTOS와 함께 제공되는 5가지 메모리 할당 체계 예제
- 각 메모리 할당 체계의 사용 사례

## 전제 조건

FreeRTOS를 사용하려면 C 프로그래밍에 대해 잘 알고 있어야 합니다. 특히 다음 사항에 정통해야 합니다.

- 컴파일 및 링크 단계를 포함한 C 프로젝트의 빌드 방식
- 스택 및 힙에 대한 개념
- 표준 C 라이브러리 malloc() 및 free() 함수

## 동적 메모리 할당과 FreeRTOS에 대한 적합성

이번 안내서에서는 작업, 대기열, 세마포어, 이벤트 그룹 등 커널 객체에 대해 살펴보겠습니다. 이러한 커널 객체들은 FreeRTOS를 최대한 쉽게 사용할 수 있도록 컴파일 단계에서는 정적으로 할당되지 않고 런타임에서 동적으로 할당됩니다. FreeRTOS는 커널 객체가 생성될 때마다 RAM을 할당한 후 커널 객체가 삭제되면 RAM을 해제하여 여유 메모리를 확보합니다. 이러한 정책은 설계 및 계획 부담을 줄일 뿐만 아니라 API를 간소화하고 RAM이 차지하는 공간을 최소화합니다.

동적 메모리 할당은 C 프로그래밍 개념이지만 FreeRTOS 또는 멀티태스킹에만 해당되는 개념은 아닙니다. 커널 객체에 동적으로 할당된다는 점에서 FreeRTOS에 적합하지만 범용 컴파일러에서 제공되는 동적 메모리 할당 체계가 항상 실시간 애플리케이션에 적합한 것은 아닙니다.

표준 C 라이브러리 malloc() 및 free() 함수를 사용하면 메모리를 할당할 수 있지만 다음과 같은 여러 가지 이유로 적합하지 않은 경우도 있습니다.

- 소형 임베디드 시스템에서는 사용하지 못할 수도 있습니다.
- 동적 메모리 할당을 구현하면 비교적 용량이 커져 소중한 코드 공간까지 차지할 수 있습니다.
- 스레드 세이프가 거의 없습니다.
- 결정적이지 않습니다. 함수를 실행하는 데 걸리는 시간이 호출에 따라 다릅니다.
- 단편화로 인해 어려울 수 있습니다. 힙의 여유 메모리(RAM)가 작은 크기의 블록으로 서로 분리되는 경우 힙이 단편화되는 것으로 알려져 있습니다. 힙이 단편화되면 분리된 여유 힙 블록의 전체 크기가 할당할 수 없는 블록의 크기보다 몇 배 크더라도 여유 힙 블록 하나의 크기가 블록을 할당할 정도로 충분히 크지 않으면 블록을 할당하려는 시도가 실패하게 됩니다.
- 링커 구성이 복잡해질 수 있습니다.
- 힙 공간이 다른 변수에서 사용하는 메모리까지 차지할 정도로 커지면 디버그 오류의 원인이 될 수 있습니다.

## 동적 메모리 할당 옵션

초기 버전의 FreeRTOS는 메모리 풀 할당 체계를 사용했습니다. 이 체계에서는 다른 크기의 메모리 블록으로 구성된 풀을 컴파일 과정에서 사전 할당한 후 메모리 할당 함수를 통해 반환합니다. 이러한 체계가 실시간 시스템에서 흔히 사용되기는 하지만 수많은 지원 요청이 발생하였습니다. 또한 실제로 소형 임베디드 시스템에서 실행할 수 있을 만큼 RAM을 효율적으로 사용하지 못해 결국 중단되었습니다.

현재 FreeRTOS는 메모리 할당을 이식 계층에 포함시켜 처리합니다(주요 코드 베이스에 포함되지 않음). 이는 임베디드 시스템의 다양한 동적 메모리 할당 및 타이밍 요건에 대해서 잘 알고 있기 때문입니다. 애플리케이션 하위 집합인 경우에 한해 동적 메모리 할당 알고리즘 하나로 적당합니다. 또한 동적 메모리 할당을 주요 코드 베이스에서 삭제하면 애플리케이션 개발자가 가능한 경우 자신만의 특정 구현체를 입력할 수도 있습니다.

FreeRTOS에서는 RAM이 필요하면 `malloc()`이 아닌 `pvPortMalloc()`을 호출합니다. 이후 RAM을 해제할 때는 커널이 `free()`가 아닌 `vPortFree()`를 호출합니다. `pvPortMalloc()`은 표준 C 라이브러리 `malloc()` 함수와 프로토타입이 동일합니다. `vPortFree()` 역시 표준 C 라이브러리 `free()` 함수와 프로토타입이 동일합니다.

`pvPortMalloc()`과 `vPortFree()`는 퍼블릭 함수이기 때문에 애플리케이션 코드에서도 호출할 수 있습니다.

FreeRTOS는 `pvPortMalloc()`과 `vPortFree()`를 구현하는 5가지 예제를 함께 제공하며, 5가지 모두 여기에 기록되어 있습니다. FreeRTOS 애플리케이션은 이러한 구현체 예제 중 1개를 사용하거나 애플리케이션 고유의 구현체를 입력할 수도 있습니다.

5가지 예제는 FreeRTOS/Source/portable/MemMang 디렉터리에 위치한 `heap_1.c`, `heap_2.c`, `heap_3.c`, `heap_4.c` 및 `heap_5.c` 소스 파일에 정의되어 있습니다.

## 메모리 할당 체계 예제

FreeRTOS 애플리케이션에는 완전히 정적으로 할당할 수 있기 때문에 힙 메모리 관리자가 따로 필요하지 않습니다.

### heap\_1

소형 임베디드 시스템은 일반적으로 스케줄러 시작 이전에만 작업을 비롯한 기타 커널 객체를 생성할 수 있습니다. 메모리는 애플리케이션이 실시간 기능을 실행하기 전에 커널에서 동적으로 할당되고, 이후 애플리케이션 수명이 끝날 때까지 할당된 상태를 유지합니다. 이 말은 할당 체계를 선택한 후로는 결정론이나 단편화 같이 복잡한 메모리 할당 문제를 고려할 필요가 없다는 것을 의미합니다. 대신에 코드 크기나 간편성 같은 속성을 고려할 수 있습니다.

`heap_1.c`는 매우 기본적인 버전인 `pvPortMalloc()`을 구현하고, `vPortFree()`는 구현하지 않습니다. 작업 또는 기타 커널 객체를 삭제하지 않는 애플리케이션은 `heap_1`을 사용할 수 있습니다.

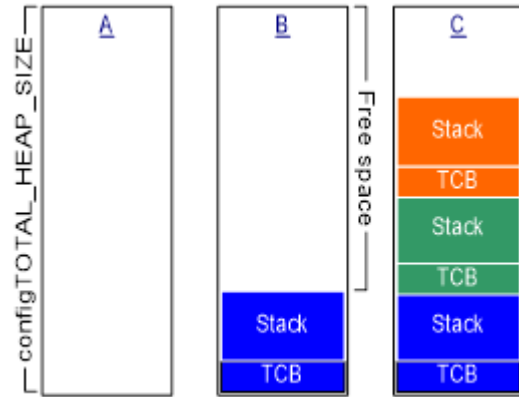
그 밖에 상업적으로 중요하거나, 안전이 필수여서 동적 메모리 할당을 제한하는 시스템 역시 `heap_1`을 사용하는 것이 가능합니다. 이러한 시스템들은 비결정론과 메모리 단편화, 그리고 잘못된 할당과 관련된 불확실성으로 인해 동적 메모리 할당을 제한하는 경우가 많지만 `heap_1`은 항상 결정적일 뿐만 아니라 메모리를 단편화할 수 없기 때문입니다.

`pvPortMalloc()`이 호출되면 `heap_1` 할당 체계가 단순 배열을 더 작은 블록으로 세분화합니다. 이러한 배열을 FreeRTOS 힙이라고 부릅니다.

배열의 총 크기(바이트)는 FreeRTOSConfig.h에서 `configTOTAL_HEAP_SIZE` 정의를 통해 설정됩니다. 이러한 방법으로 커다란 크기의 배열을 정의하면 배열에서 메모리가 할당되기 전에도 애플리케이션이 많은 용량의 RAM을 사용하는 것처럼 보일 수 있습니다.

생성되는 작업마다 작업 제어 블록(TCB)과 스택을 힙에서 할당해야 합니다.

다음 그림은 작업이 생성될 때 heap\_1이 단순 배열을 어떻게 세분화하는지 나타낸 것입니다. 작업이 생성될 때마다 RAM이 heap\_1 배열에서 할당됩니다.



- A는 작업 생성 이전의 배열을 나타냅니다. 전체 배열이 여유 RAM입니다.
- B는 작업이 1개 생성된 이후의 배열을 나타냅니다.
- C는 작업이 3개 생성된 이후의 배열을 나타냅니다.

## heap\_2

heap\_2는 역호환을 지원하기 위해 FreeRTOS 배포판에 포함되어 있습니다. 하지만 새로운 설계에 사용하는 것은 바람직하지 않습니다. 대신에 더욱 많은 기능을 제공하는 heap\_4를 사용하는 것이 좋습니다.

heap\_2.c 역시 configTOTAL\_HEAP\_SIZE에서 배열 크기를 정의한 후 이를 세분화합니다. 그런 다음 최적합 알고리즘에 따라 메모리를 할당합니다. 여기에서는 heap\_1과 달리 메모리를 해제할 수 있습니다. 다시 말하지만 배열이 정적으로 선언되기 때문에 배열에서 메모리가 할당되기 전에도 애플리케이션이 많은 용량의 RAM을 사용하는 것처럼 보일 수 있습니다.

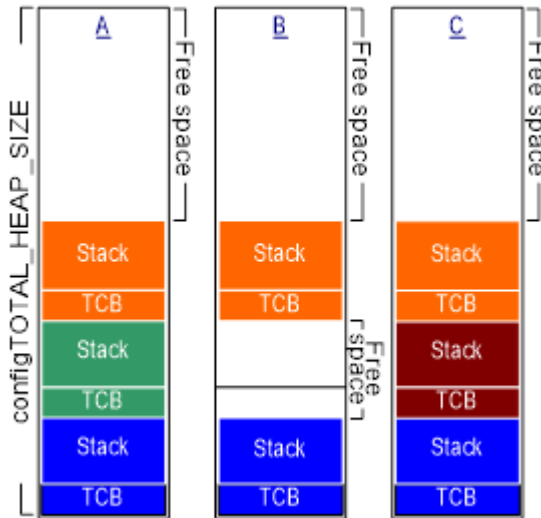
최적합 알고리즘을 따르기 때문에 pvPortMalloc()이 크기에서 요청 바이트 수와 가장 가까운 여유 메모리 블록을 사용합니다. 예를 들어 다음과 같은 시나리오를 가정하겠습니다.

- 힙에 각각 5바이트, 25바이트 및 100바이트인 여유 메모리 블록이 3개 있습니다.
- pvPortMalloc()을 호출하여 RAM 20바이트를 요청합니다.

요청한 바이트 수에 적합하면서 가장 작은 여유 RAM 블록은 25바이트 블록입니다. 따라서 pvPortMalloc()이 20바이트 블록을 가리키는 포인터를 반환하기 전에 먼저 25바이트 블록을 20바이트 블록 1개와 5바이트 블록 1개로 분할합니다. (heap\_2는 블록 크기에 대한 정보를 힙 영역에 저장하여 실제로 분할된 블록 2개의 합이 25보다 작기 때문에 이 예는 지나치게 단순화된 것입니다) 새로운 5바이트 블록은 향후 pvPortMalloc() 호출 시 사용할 수 있습니다.

heap\_2는 heap\_4와 달리 인접한 여유 블록을 더욱 큰 블록으로 결합하지 않습니다. 그렇기 때문에 단편화에 더 민감합니다. 하지만 할당되는 블록과 이후 해제되는 블록이 항상 동일한 크기라면 단편화는 문제가 되지 않습니다. heap\_2는 작업을 반복해서 생성하고 삭제하는 애플리케이션에게 적합합니다. 단, 생성된 작업에 할당되는 스택의 크기가 바뀌지 않는 경우에 한합니다.

다음 그림은 작업 생성 및 삭제 시 heap\_2 배열에서 할당되었다가 해제되는 RAM을 나타낸 것입니다.



위 그림은 작업의 생성, 삭제 및 생성이 반복될 때 최적화 알고리즘이 어떻게 이루어지는지 나타낸 것입니다.

- A는 작업이 3개 생성된 이후의 배열을 나타냅니다. 용량이 큰 여유

블록이 배열 상단을 차지합니다.

- B는 작업 중 1개가 삭제된 이후의 배열을 나타냅니다. 이

용량이 큰 여유 블록은 배열 상단에 그대로 남습니다. 또한 삭제된 작업의 TCB와 스택에 할당되었던, 작은 용량의 여유 블록 2개가 있습니다.

- C는 다른 작업이 생성된 이후의 배열을 나타냅니다. 태스크를 생성하면서

한 번은 새로운 TCB를 할당할 목적으로, 또 한 번은 작업 스택을 할당할 목적으로 `pvPortMalloc()` 을 두 번 호출했습니다. 작업은 `xTaskCreate()` API 함수를 사용해 생성되며, 자세한 내용은 [작업 생성 \(p. 26\)](#) 단원에 나와있습니다. `pvPortMalloc()` 호출은 `xTaskCreate()` 내부에서 이루어집니다.

TCB는 모두 정확하게 동일한 크기이기 때문에 최적화 알고리즘에 따라 삭제된 작업의 TCB에 할당되었던 RAM 블록을 재사용해 새로운 작업의 TCB에 할당할 수 있습니다.

새롭게 생성된 작업에 할당되는 스택의 크기도 앞에서 삭제된 작업에 할당된 크기와 동일하기 때문에 최적화 알고리즘에 따라 삭제된 작업의 스택에 할당되었던 RAM 블록을 재사용해 새로운 작업의 스택에 할당할 수 있습니다.

용량이 큰 미할당 블록은 배열 상단에 그대로 남습니다.

`heap_2`는 결정적이지 않지만 `malloc()`, `free()` 같은 대부분 표준 라이브러리 구현체보다 빠릅니다.

## heap\_3

`heap_3.c`는 표준 라이브러리 함수인 `malloc()`과 `free()`를 사용하기 때문에 링커 구성에서 힙의 크기를 정의합니다. 따라서 `configTOTAL_HEAP_SIZE` 설정은 아무런 효과가 없습니다.

`heap_3`은 FreeRTOS 스케줄러를 일시적으로 중지하여 `malloc()`과 `free()`를 스레드 세이프하게 만듭니다. 스레드 세이프와 스케줄러 일시 중지 관련 자세한 내용은 [리소스 관리 \(p. 152\)](#) 단원을 참조하십시오.

## heap\_4

heap\_1 및 heap\_2와 달리 heap\_4는 배열을 더욱 작은 블록으로 세분화합니다. 배열이 정적으로 선언되어 configTOTAL\_HEAP\_SIZE에서 크기를 정의하기 때문에 배열에서 메모리가 할당되기 전에도 애플리케이션이 많은 용량의 RAM을 사용하는 것처럼 보일 수 있습니다.

heap\_4는 최초 적합 알고리즘에 따라 메모리를 할당합니다. 또한 heap\_2와 달리 인접한 여유 메모리 블록을 더욱 큰 블록으로 결합(병합)합니다. 이를 통해 메모리 단편화의 위험을 최소화합니다.

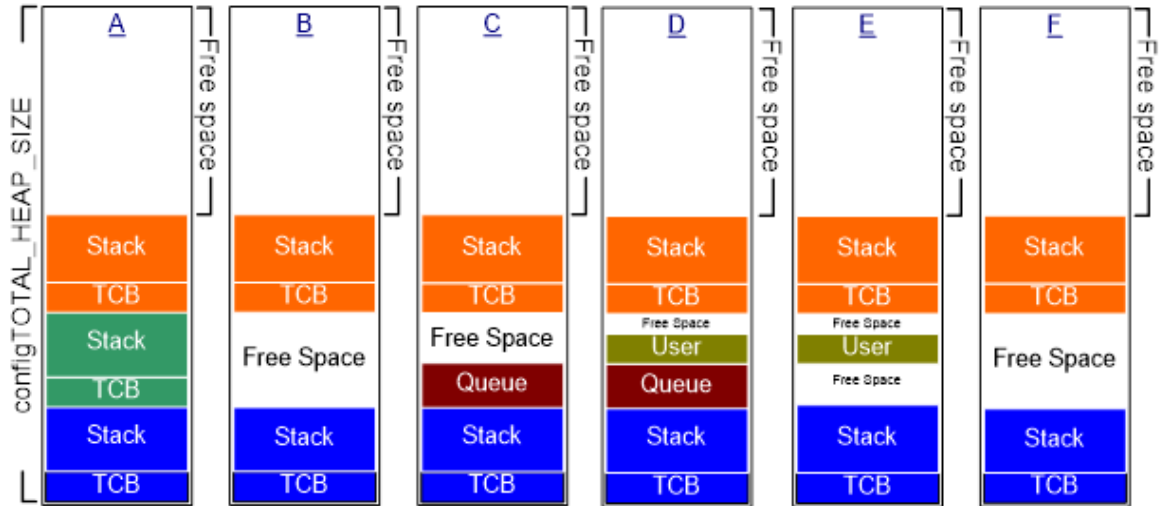
최초 적합 알고리즘을 따르기 때문에 pvPortMalloc()이 요청 바이트 수를 저장할 만큼 큰 용량의 첫 번째 여유 메모리 블록을 사용합니다. 예를 들어 다음과 같은 시나리오를 가정하겠습니다.

- 힙에 여유 메모리 블록이 3개 있습니다. 배열 순서는 5바이트, 200바이트, 100바이트입니다.
- pvPortMalloc()을 호출하여 RAM 20바이트를 요청합니다.

요청한 바이트 수에 적합한 첫 번째 여유 RAM 블록은 200바이트 블록입니다. 따라서 pvPortMalloc()이 20바이트 블록을 가리키는 포인터를 반환하기 전에 먼저 200바이트 블록을 20바이트 블록 1개와 180바이트 블록 1개로 분할합니다. (heap\_4는 블록 크기에 대한 정보를 힙 영역에 저장하여 분할된 블록 2개의 합이 200바이트보다 작기 때문에 이 예는 지나치게 단순화된 것입니다) 새로운 180바이트 블록은 향후 pvPortMalloc() 호출 시 사용할 수 있습니다.

heap\_4는 인접한 여유 메모리 블록을 더욱 큰 블록으로 결합(병합)하여 단편화의 위험을 최소화합니다. 따라서 다른 크기의 RAM 블록을 반복해서 할당 및 해제하는 애플리케이션에 적합합니다.

다음 그림은 heap\_4 배열에서 할당되었다가 해제되는 RAM을 나타낸 것입니다. 메모리 할당 및 해제 시 heap\_4 최초 적합 알고리즘이 메모리 병합과 함께 어떻게 이루어지는지 살펴볼 수 있습니다.



A는 작업이 3개 생성된 이후의 배열을 나타냅니다. 용량이 큰 여유 블록은 배열 상단을 차지합니다.

B는 작업 중 1개가 삭제된 이후의 배열을 나타냅니다. 용량이 큰 여유 블록은 배열 상단에 그대로 남습니다. 또한 삭제된 작업의 TCB 및 스택에 할당되었던 여유 블록도 1개 있습니다. TCB 삭제와 함께 해제된 메모리와 스택 삭제와 함께 해제된 메모리가 각각 별도의 여유 블록 2개로 남지 않습니다. 오히려 더욱 큰 여유 블록 1개로 결합됩니다.

C는 FreeRTOS 대기열이 생성된 이후의 배열을 나타냅니다. 대기열은 xQueueCreate() API 함수를 사용해 생성되며, 자세한 내용은 [대기열 사용 \(p. 66\)](#) 단원에 자세하게 나와있습니다. xQueueCreate()가 pvPortMalloc()을 호출하여 대기열에서 사용할 RAM도 할당합니다. heap\_4는 최초 적합 알고리즘을 따르기 때문에 pvPortMalloc()이 대기열을 저장할 만큼 큰 용량의 첫 번째 여유 RAM 블록에서 RAM을 할당합니다. 그림에서 할당되는 RAM은 작업 삭제 시 해제된 RAM입니다. 대기열이 여유 블록의 RAM을 모두 사용하지



않기 때문에 블록이 둘로 분할되어 있습니다. 미사용 RAM은 향후 pvPortMalloc() 호출 시 사용할 수 있습니다.

D는 pvPortMalloc()이 FreeRTOS API 함수를 호출하여 간접적으로 호출되지 않고 애플리케이션 코드에서 직접 호출된 이후의 배열을 나타냅니다. 사용자가 할당한 블록의 크기가 첫 번째 여유 블록, 즉 대기열에 할당된 메모리와 다음 TCB에 할당된 메모리 사이의 블록에 적합할 정도로 작습니다. 작업 삭제와 함께 해제된 메모리가 이제 별도의 블록 3개로 분할되었습니다. 첫 번째 블록에는 대기열이 저장됩니다. 두 번째 블록에서 사용자 할당 메모리가 저장됩니다. 세 번째는 여유 메모리입니다.

E는 대기열이 삭제된 이후의 배열을 나타냅니다. 이때 삭제된 대기열에 할당되었던 메모리도 자동으로 해제됩니다. 이제는 사용자 할당 블록의 양쪽에 여유 메모리가 있습니다.

F는 사용자 할당 메모리까지 해제된 이후의 배열을 나타냅니다. 사용자 할당 블록에서 사용했던 메모리가 양쪽 여유 메모리와 결합되면서 더욱 큰 용량의 여유 블록이 생성되었습니다.

heap\_4는 결정적이지 않지만 malloc(), free() 같은 대부분 표준 라이브러리 구현체보다 빠릅니다.

## heap\_4에서 사용하는 배열의 시작 주소 설정

참고: 이번 단원에는 고급 정보가 포함되어 있습니다. heap\_4를 사용하면서 이번 단원을 반드시 읽어야 할 필요는 없습니다.

애플리케이션 개발자들은 heap\_4에서 사용할 배열을 특정 메모리 주소에 배치해야 하는 경우가 간혹 있습니다. 예를 들어 FreeRTOS 작업에서 사용되는 스택은 힙에서 할당되기 때문에 힙이 느린 외부 메모리보다는 빠른 내부 메모리에 있어야 할 때가 그렇습니다.

기본적으로 heap\_4에서 사용하는 배열은 heap\_4.c 소스 파일에서 선언됩니다. 배열의 시작 주소는 링커에서 자동으로 설정됩니다. 하지만 configAPPLICATION\_ALLOCATED\_HEAP 컴파일 시간 구성 상수가 FreeRTOSConfig.h에서 1로 설정되어 있으면 배열이 애플리케이션에서 FreeRTOS를 사용해 선언되어야 합니다. 배열이 애플리케이션에서 선언되면 애플리케이션 개발자가 시작 주소를 설정할 수 있습니다.

configAPPLICATION\_ALLOCATED\_HEAP이 FreeRTOSConfig.h에서 1로 설정되어 있으면 애플리케이션의 소스 파일 중 1개에서 이름이 ucHeap이고, configTOTAL\_HEAP\_SIZE 설정에서 크기가 정의되는 uint8\_t 배열을 선언해야 합니다.

특정 메모리 주소에 변수를 배치하는 데 필요한 구문은 컴파일러에 따라 다릅니다. 자세한 내용은 사용 중인 컴파일러 설명서를 참조하십시오.

여기에서는 두 가지 컴파일러를 예로 듭니다.

다음은 GCC 컴파일러에서 배열을 선언한 후 .my\_heap이라고 하는 메모리 할당 주소에 배치하는 데 필요한 구문입니다.

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ] __attribute__ ( (section( ".my_heap" ) ) );
```

다음은 IAR 컴파일러에서 배열을 선언한 후 절대 메모리 주소인 0x20000000에 배치하는 데 필요한 구문입니다.

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ] @ 0x20000000;
```

## heap\_5

heap\_5에서 메모리를 할당하거나 해제할 때 사용하는 알고리즘은 heap\_4에서 사용하는 알고리즘과 동일합니다. 단, heap\_5는 heap\_4와 달리 정적으로 선언된 단일 배열에서 메모리를 할당해야 하는 것은 아닙니다.

heap\_5는 서로 분리된 다수의 메모리 공간에서 메모리를 할당할 수 있습니다. heap\_5는 FreeRTOS가 실행되는 시스템의 RAM이 시스템의 메모리 맵에서 연속된, 즉 공간이 없는 단일 블록으로 표시되지 않을 때 유용합니다.

heap\_5는 pvPortMalloc() 호출을 위해 유일하게 명시적으로 초기화해야 하는 메모리 할당 체계입니다. 초기화는 vPortDefineHeapRegions() API 함수를 통해 이루어집니다. heap\_5를 사용할 때 커널 객체(작업, 대기열, 세마포어 등)를 생성하려면 먼저 vPortDefineHeapRegions()를 호출해야 합니다.

## vPortDefineHeapRegions() API 함수

vPortDefineHeapRegions() 함수는 각각 독립된 메모리 영역의 시작 주소와 크기를 지정할 때 사용됩니다. 이러한 메모리 영역이 모두 모여 heap\_5에서 사용하는 총 메모리가 됩니다.

```
void vPortDefineHeapRegions( const HeapRegion_t * const pxHeapRegions );
```

독립된 메모리 영역은 각각 HeapRegion\_t 형식의 구조로 설명됩니다. 사용할 수 있는 모든 메모리 영역에 대한 설명이 HeapRegion\_t 구조 배열로 vPortDefineHeapRegions()에게 전달됩니다.

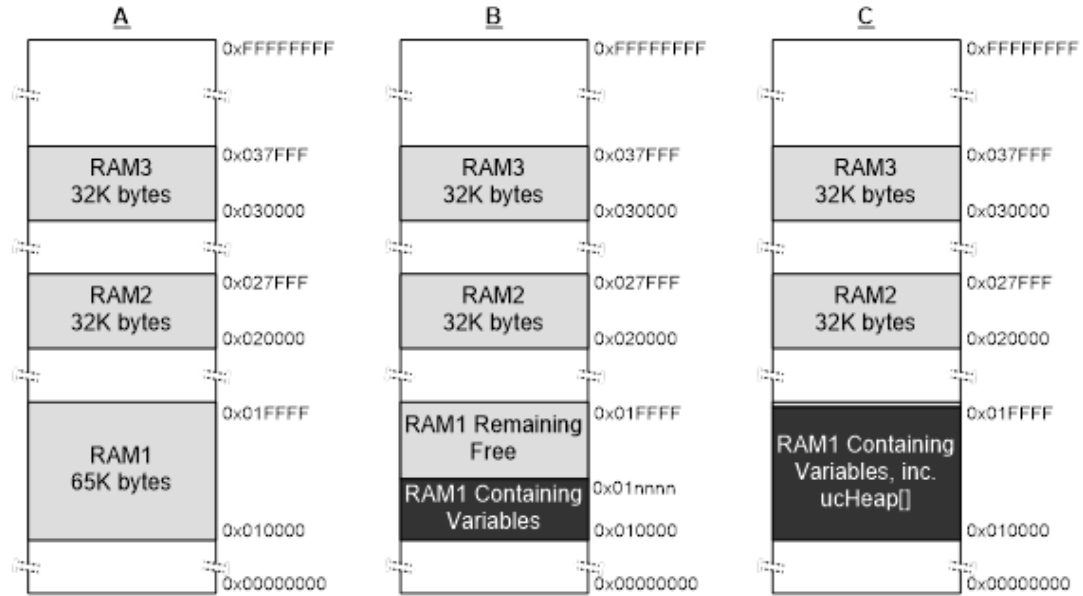
```
typedef struct HeapRegion
{
    /* The start address of a block of memory that will be part of the heap.*/
    uint8_t *pucStartAddress;

    /* The size of the block of memory in bytes. */
    size_t xSizeInBytes;
} HeapRegion_t;
```

다음 표는 vPortDefineHeapRegions() 파라미터를 나열한 것입니다.

파라미터 이름/반환 값	설명
pxHeapRegions	HeapRegion_t 구조 배열의 시작을 가리키는 포인터입니다. 배열 속 각 구조는 heap_5 사용 시 힙에 포함되는 메모리 영역의 시작 주소와 길이를 나타냅니다. 배열 속 HeapRegion_t 구조의 순서는 시작 주소를 기준으로 정렬되어야 합니다. 시작 주소가 가장 낮은 메모리 영역을 나타내는 HeapRegion_t 구조가 배열에서 첫 번째 구조가 되어야 하고, 시작 주소가 가장 높은 메모리 영역을 나타내는 HeapRegion_t 구조는 배열에서 마지막 구조가 되어야 합니다. HeapRegion_t 구조에서 pucStartAddress 멤버가 NULL로 설정되어 있으면 배열의 끝을 의미합니다.

예를 들어 다음 그림과 같이 RAM1, RAM2, RAM3 등 독립된 RAM 블록이 3개 포함된 메모리 맵을 가정하겠습니다. 실행 가능한 코드는 읽기 전용 메모리에 배치되기 때문에 그림에는 없습니다.



다음 코드는 HeapRegion\_t 구조의 배열을 나타낸 것입니다. 코드가 모두 RAM 블록 3개를 설명하고 있습니다.

```
/* Define the start address and size of the three RAM regions. */

#define RAM1_START_ADDRESS ( ( uint8_t * ) 0x00010000 )

#define RAM1_SIZE ( 65 * 1024 )

#define RAM2_START_ADDRESS ( ( uint8_t * ) 0x00020000 )

#define RAM2_SIZE ( 32 * 1024 )

#define RAM3_START_ADDRESS ( ( uint8_t * ) 0x00030000 )

#define RAM3_SIZE ( 32 * 1024 )

/* Create an array of HeapRegion_t definitions, with an index for each of the three RAM
regions, and terminating the array with a NULL address. The HeapRegion_t structures must
appear in start address order, with the structure that contains the lowest start address
appearing first. */

const HeapRegion_t xHeapRegions[] =
{
    { RAM1_START_ADDRESS, RAM1_SIZE },
    { RAM2_START_ADDRESS, RAM2_SIZE },
    { RAM3_START_ADDRESS, RAM3_SIZE },
    { NULL, 0 } /* Marks the end of the array. */
};

int main( void )
{
```

```
/* Initialize heap_5. */  
  
vPortDefineHeapRegions( xHeapRegions );  
  
/* Add application code here. */  
  
}
```

코드가 RAM을 정확하게 설명하고 있지만 모든 RAM을 힙에 할당하여 다른 변수에 사용할 여유 RAM이 없기 때문에 유용한 예제는 아닙니다.

프로젝트가 빌드되면 빌드 프로세스의 링크 단계에서 RAM 주소를 각 변수에 할당합니다. 링커에 사용할 수 있는 RAM은 일반적으로 링커 스크립트 같은 링커 구성 파일에서 설명됩니다. 위 그림에서 B는 RAM2 또는 RAM3이 아닌 RAM1에 대한 정보가 링커 스크립트에 추가되었다고 가정한 것입니다. 따라서 링커가 변수를 RAM1에 배치하면서 RAM1 상단 주소인 0x0001nnnn만 heap\_5에서 사용할 수 있게 됩니다. 0x0001nnnn의 실제 값은 링크 대상인 애플리케이션에 포함된 변수의 전체 크기에 따라 달라집니다. RAM2와 RAM3는 링커가 사용하지 않기 때문에 heap\_5에서 사용할 수 있습니다.

위의 코드를 사용한다면 heap\_5 하단 주소인 0x0001nnnn에 할당되는 RAM이 변수 저장에 사용되는 RAM과 중복됩니다. 이를 피하기 위해 xHeapRegions[] 배열에서 첫 번째 HeapRegion\_t 구조가 시작 주소로 0x00010000이 아닌 0x0001nnnn을 사용할 수 있습니다.

하지만 이러한 방법은 다음과 같은 이유로 권장할 만한 해결책이 아닙니다.

- 시작 주소를 결정하기 쉽지 않습니다.
- HeapRegion\_t 구조에 사용되는 시작 주소로 업데이트해야 할 경우 링커에 사용되는 RAM의 크기가 향후 빌드에서 바뀔 수 있습니다.
- heap\_5에서 사용되는 RAM이 중복되더라도 빌드 도구가 이를 인식하지 못하기 때문에 애플리케이션 개발자에게 경고하지 못합니다.

다음 코드는 더욱 편리하고 쉽게 관리할 수 있는 예제를 설명한 것입니다. 이 코드는 ucHeap이라고 하는 배열을 선언하고 있습니다. ucHeap은 정규 변수이기 때문에 링커에서 RAM1에 할당하는 데이터에 포함됩니다. xHeapRegions 배열에서 첫 번째 HeapRegion\_t 구조는 ucHeap의 시작 주소와 크기를 나타내기 때문에 ucHeap이 heap\_5에서 관리하는 메모리에 포함됩니다. ucHeap의 크기는 위 그림의 C와 같이 링커에 사용되는 RAM0이 RAM1을 모두 소진할 때까지 증가할 수 있습니다.

```
/* Define the start address and size of the two RAM regions not used by the linker. */  
  
#define RAM2_START_ADDRESS ( ( uint8_t * ) 0x00020000 )  
  
#define RAM2_SIZE ( 32 * 1024 )  
  
#define RAM3_START_ADDRESS ( ( uint8_t * ) 0x00030000 )  
  
#define RAM3_SIZE ( 32 * 1024 )  
  
/* Declare an array that will be part of the heap used by heap_5. The array will be placed  
   in RAM1 by the linker. */  
  
#define RAM1_HEAP_SIZE ( 30 * 1024 )  
  
static uint8_t ucHeap[ RAM1_HEAP_SIZE ];  
  
/* Create an array of HeapRegion_t definitions. Whereas in previous code listing, the first  
   entry described all of RAM1, so heap_5 will have used all of RAM1, this time the first  
   entry only describes the ucHeap array, so heap_5 will only use the part of RAM1 that  
   contains the ucHeap array. The HeapRegion_t structures must still appear in start address  
   order, with the structure that contains the lowest start address appearing first. */  
  
const HeapRegion_t xHeapRegions[] =
```

```
{
    { ucHeap, RAM1_HEAP_SIZE },
    { RAM2_START_ADDRESS, RAM2_SIZE },
    { RAM3_START_ADDRESS, RAM3_SIZE },
    { NULL, 0 } /* Marks the end of the array. */
};
```

위 코드에서 HeapRegion\_t 구조의 배열은 RAM2와 RAM3을 모두 나타내고 있지만 RAM1은 일부만 나타내고 있습니다.

여기에서 설명하는 기법의 이점은 다음과 같습니다.

- 하드 코딩된 시작 주소를 사용할 필요 없습니다.
- HeapRegion\_t 구조에서 사용되는 주소가 링커에서 자동으로 설정되기 때문에 링커에서 사용되는 RAM의 크기가 이후 빌드에서 바뀌더라도 항상 올바른 주소를 사용할 수 있습니다.
- heap\_5에 할당되는 RAM이 링커에서 RAM1에 배치되는 데이터와 중복되지 않습니다.
- ucHeap이 너무 크면 애플리케이션이 링크되지 않습니다.

## 힙 관련 유틸리티 함수

### xPortGetFreeHeapSize() API 함수

xPortGetFreeHeapSize() API 함수는 호출 시 힙의 여유 바이트 수를 반환합니다. 이렇게 반환되는 값은 힙 크기를 최적화하는 데 사용됩니다. 예를 들어 커널 객체가 모두 생성한 후 xPortGetFreeHeapSize() 함수가 2000을 반환하면 configTOTAL\_HEAP\_SIZE 값이 2000까지 줄어들 수 있습니다.

heap\_3을 사용할 때는 xPortGetFreeHeapSize() 함수가 제공되지 않습니다.

xPortGetFreeHeapSize() API 함수 프로토타입

```
size_t xPortGetFreeHeapSize( void );
```

다음 표는 xPortGetFreeHeapSize() 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
반환 값	xPortGetFreeHeapSize() 호출 시 할당되지 않고 힙에 남아있는 바이트 수를 반환합니다.

### xPortGetMinimumEverFreeHeapSize() API 함수

xPortGetMinimumEverFreeHeapSize() API 함수는 FreeRTOS 애플리케이션이 실행된 이후 지금까지 할당되지 않고 힙에 존재하는 최소 바이트 수를 반환합니다.

xPortGetMinimumEverFreeHeapSize()에서 반환되는 값은 애플리케이션의 힙 공간이 얼마나 줄어들었는지 나타내는 지표입니다. 예를 들어 xPortGetMinimumEverFreeHeapSize() 함수가 200을 반환하면 애플리케이션이 실행된 이후 임의의 시점에 힙 공간이 거의 200바이트까지 줄어들었다는 것을 의미합니다.

xPortGetMinimumEverFreeHeapSize() 함수는 heap\_4 또는 heap\_5를 사용할 때만 제공됩니다.

다음 표는 xPortGetMinimumEverFreeHeapSize() 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
반환 값	FreeRTOS 애플리케이션이 실행된 이후 지금까지 할당되지 않고 힙에 존재하는 최소 바이트 수를 반환합니다.

## Malloc 실패 후크 함수

pvPortMalloc()은 애플리케이션 코드에서 직접 호출할 수 있습니다. 그 밖에 커널 객체를 생성할 때마다 FreeRTOS 소스 파일에서도 호출 가능합니다. 작업, 대기열, 세마포어 및 이벤트 그룹이 커널 객체에 포함되며, 모두 나중에 자세하게 설명하겠습니다.

표준 라이브러리 malloc() 함수와 마찬가지로 요청 크기의 블록이 존재하지 않아서 pvPortMalloc() 함수가 RAM 블록을 반환하지 못할 경우에는 NULL을 반환합니다. 애플리케이션 개발자가 커널 객체를 생성하면서 pvPortMalloc() 함수를 실행했는데 pvPortMalloc() 호출에서 NULL이 반환되면 커널 객체도 생성되지 않습니다.

pvPortMalloc() 호출 시 NULL이 반환되면 경우 후크(또는 콜백) 함수가 호출되도록 모든 힙 할당 체계 예제를 구성할 수 있습니다.

configUSE\_MALLOC\_FAILED\_HOOK가 FreeRTOSConfig.h에서 1로 설정되면 애플리케이션이 아래와 같은 이름과 프로토타입을 갖는 malloc 실패 후크 함수를 제공해야 합니다. 함수는 애플리케이션에 적합하다면 어떤 방식으로든 구현할 수 있습니다.

```
void vApplicationMallocFailedHook( void );
```

## 작업 관리

이번 단원에서 설명하는 개념은 FreeRTOS의 사용 방법과 FreeRTOS 애플리케이션의 동작 방법을 이해하는 데 기초가 됩니다. 이번 단원에서 다루는 내용은 다음과 같습니다.

- FreeRTOS가 애플리케이션 내부에서 처리 시간을 각 작업에게 할당하는 방법
- FreeRTOS가 임의의 시간에 실행되어야 하는 작업을 선택하는 방법
- 각 작업의 상대적 우선순위가 시스템 동작에 미치는 영향
- 작업이 존재할 수 있는 상태

그 밖에도 다음과 같은 내용에 대해 설명합니다.

- 작업 구현 방법
- 작업 인스턴스를 1개 이상 생성하는 방법
- 작업 파라미터 사용 방법
- 이미 생성된 작업의 우선순위를 변경하는 방법
- 작업 삭제 방법
- 작업을 사용해 주기적 처리를 구현하는 방법 자세한 내용은 software\_tier\_management 단원을 참조하십시오.
- 유휴 작업의 실행 시점과 사용 방법

## 작업 함수

작업은 C 함수로 구현됩니다. C 함수는 프로토타입이 유일하게 특별하여 다음과 같이 보이드를 반환하는 동시에 보이드 포인터 파라미터를 사용해야 합니다.

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

각 작업은 자체적으로 작은 프로그램입니다. 진입점을 통해 무한 루프 내에서 계속해서 실행될 뿐 종료되지는 않습니다.

FreeRTOS 작업은 어떤 식으로든 구현 함수에서 복귀되어서는 안 됩니다. 따라서 작업에 'return' 문이 포함되어서는 안 될 뿐만 아니라 함수의 끝을 지나서 실행되어서도 안 됩니다. 작업이 더 이상 필요 없으면 명시적으로 삭제되어야 합니다.

단일 작업 함수 정의는 작업을 무제한으로 생성하는 데 사용될 수 있습니다. 생성된 작업이 각각 별도의 실행 인스턴스이며, 여기에 작업 자체 내에서 정의된 스택과 자동(스택) 변수의 복사본이 포함되어 있습니다.

일반적인 작업의 구조는 다음과 같습니다.

```
void ATaskFunction( void *pvParameters )
{
    /* Variables can be declared just as per a normal function. Each instance of a task created
    using this example function will have its own copy of the lVariableExample variable. This
```

```
would not be true if the variable was declared static, in which case only one copy of the
variable would exist, and this copy would be shared by each created instance of the task.
(The prefixes added to variable names are described in Data Types and Coding Style Guide.)
*/

int32_t lVariableExample = 0;

/* A task will normally be implemented as an infinite loop. */

for( ;; )

{

/* The code to implement the task functionality will go here. */

}

/* Should the task implementation ever break out of the above loop, then the task must be
deleted before reaching the end of its implementing function. The NULL parameter passed to
the vTaskDelete() API function indicates that the task to be deleted is the calling (this)
task. The convention used to name API functions is described in Data Types and Coding
Style Guide. */

vTaskDelete( NULL );

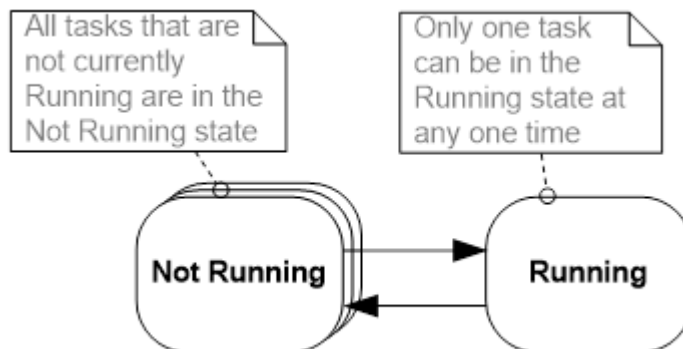
}
```

## 최상위 작업 상태

애플리케이션은 다수의 작업으로 구성될 수 있습니다. 애플리케이션을 실행하는 프로세서가 단일 코어라면 실행할 수 있는 작업은 언제나 1개로 제한됩니다. 이 말은 작업이 Running 상태와 Not Running 상태, 둘 중 하나로 존재한다는 것을 의미합니다. 하지만 이것은 지나치게 단순화된 것일 뿐입니다. 이번 단원 후반부에서 Not Running 상태가 실제로 다수의 하위 상태로 구성된다는 점을 알 수 있습니다.

작업이 Running 상태일 때는 프로세서가 작업의 코드를 실행합니다. 작업이 Not Running 상태일 때는 실행되지 않다가 스케줄러가 다음에 Running 상태로 전환해야 한다고 결정할 때 실행을 재개할 수 있도록 상태가 저장됩니다. 작업이 실행을 재개할 때는 마지막으로 Running 상태를 종료하기 전에 실행하려고 있던 명령부터 이어집니다.

다음 그림은 최상위 작업 상태와 전환을 나타낸 것입니다.



작업이 Not Running 상태에서 Running 상태로 전환되는 것을 스위치 인 또는 스왑 인이라고 합니다. 반대로 작업이 Running 상태에서 Not Running 상태로 전환되는 것을 스위치 아웃 또는 스왑 아웃이라고 합니다. FreeRTOS 스케줄러는 작업을 스위치 인 또는 스위치 아웃할 수 있는 유일한 엔티티입니다.



## 작업 생성

### xTaskCreate() API 함수

FreeRTOS V9.0.0에도 컴파일 과정에서 작업을 정적으로 생성하는 데 필요한 메모리를 할당하는 `xTaskCreateStatic()` 함수가 포함되어 있습니다. 작업은 FreeRTOS `xTaskCreate()` API 함수를 사용해 생성됩니다. 이 함수는 모든 API 함수 중에서 가장 복잡하다고 말할 수 있습니다. 작업은 멀티태스킹 시스템에서 가장 기본적인 구성 요소이기 때문에 가장 먼저 숙지해야 합니다. 이번 안내서에는 `xTaskCreate()` 함수의 예제가 다수 포함되어 있습니다.

데이터 형식 및 명명 규칙에 대한 자세한 내용은 [FreeRTOS 커널 배포 \(p. 5\)](#)에서 데이터 형식과 코딩 스타일 가이드 단원을 참조하십시오.

`xTaskCreate()` API 함수 프로토타입은 다음과 같습니다.

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode, const char * const pcName, uint16_t  
usStackSize, void *pvParameters, UBaseType_t uxPriority, TaskHandle_t *pxCreatedTask );
```

다음은 `xTaskCreate()` 파라미터와 반환 값을 나열한 것입니다.

- `pvTaskCode`: 작업은 절대로 종료되지 않는 C 함수이며, 이에 따라 일반적으로 무한 루프로 구현됩니다. `pvTaskCode` 파라미터는 작업 구현 함수를 가리키는 포인터입니다(실제 함수 이름).
- `pcName`: 작업을 나타내는 서술형 이름입니다. FreeRTOS에서는 어떠한 경우에도 이 이름을 사용하지 않습니다. 이 이름이 포함되는 이유는 디버깅을 위해서입니다. 인간이 판독할 수 있는 이름으로 작업을 식별하는 것이 핸들로 식별하는 것보다 훨씬 쉽습니다. 애플리케이션 정의 상수인 `configMAX_TASK_NAME_LEN`은 NULL 종결자를 포함하여 작업 이름의 최대 길이를 정의합니다. 최대 길이보다 긴 문자열을 입력하면 문자열이 아무런 경고 없이 잘립니다.
- `usStackSize`: 작업마다 생성될 때 커널에서 할당되는 고유 스택이 있습니다. `usStackSize` 값은 커널이 생성해야 하는 스택의 크기를 나타냅니다. 다시 말해서 이 값은 바이트 수가 아니고 스택에 저장할 수 있는 워드 수를 지정합니다. 예를 들어 스택 너비가 32비트이고, `usStackSize`가 100으로 전달된다면 스택 공간은 400바이트가 할당됩니다( $100 * 4$ 바이트). 스택 깊이와 스택 너비를 곱한 값이 `uint16_t` 형식의 변수에 저장할 수 있는 최대 값을 넘어서는 안 됩니다. 유휴 작업에서 사용할 스택 크기는 애플리케이션 정의 상수인 `configMINIMAL_STACK_SIZE`에서 정의합니다. 이는 FreeRTOS 소스 코드가 `configMINIMAL_STACK_SIZE` 설정을 유일하게 사용하는 방식입니다. 이 상수는 데모 애플리케이션 내부에서도 사용되어 데모를 여러 프로세서 아키텍처에 이식하는 데 유용합니다. 사용할 프로세서 아키텍처의 FreeRTOS 데모 애플리케이션에서 이 상수에 할당되는 값은 모든 작업에 권장되는 최소 값입니다. 작업이 스택 공간을 많이 사용한다면 더욱 높은 값을 할당해야 합니다. 작업에 필요한 스택 공간을 쉽게 결정할 수 있는 방법은 없습니다. 계산할 수는 있지만 대부분 사용자들은 타당하다고 생각되는 값을 할당한 후 FreeRTOS 기능을 사용해 할당된 공간이 실제로 적합한지, 그리고 RAM을 낭비하고 있지 않은지 확인합니다. 작업에서 사용된 최대 스택 공간을 쿼리하는 방법에 대한 자세한 내용은 문제 해결 단원에서 [스택 오버플로우 \(p. 234\)](#)를 참조하십시오.
- `pvParameters`: 작업 함수는 보이드를 가리키는 포인터(`void*`) 형식의 파라미터 사용을 허용합니다. `pvParameters`에 할당되는 값이 작업에 전달되는 값입니다. 이번 안내서에는 파라미터의 사용 방법을 나타내는 예제들이 다수 포함되어 있습니다.
- `uxPriority`: 작업의 실행 우선순위를 정의합니다. 우선순위는 가장 낮은 우선순위를 나타내는 0부터 가장 높은 우선순위인 (`configMAX_PRIORITIES - 1`)까지 할당할 수 있습니다. `configMAX_PRIORITIES`는 사용자 정의 상수이며, 자세한 내용은 [작업 우선순위 \(p. 32\)](#) 단원에 나와있습니다. `uxPriority` 값을 (`configMAX_PRIORITIES - 1`)보다 크게 전달하면 작업에 할당된 우선순위가 아무런 경고 없이 최대 허용 값으로 제한됩니다.
- `pxCreatedTask`: 이 파라미터는 핸들을 생성된 작업에게 전달하는 데 사용됩니다. 이렇게 전달된 핸들은 작업 우선순위를 변경하거나 작업을 삭제하는 등 API 호출 시 작업을 참조하는 데 사용할 수 있습니다. 애플리케이션에서 작업 핸들을 사용하지 않는다면 `pxCreatedTask`를 NULL로 설정할 수 있습니다.

반환 값은 pdPASS와 pdFAIL, 두 가지가 가능합니다. pdPASS는 작업이 성공적으로 생성되었다는 것을, 그리고 pdFAIL은 FreeRTOS가 작업 데이터 구조와 스택을 저장할 만큼 충분한 RAM을 할당하는 데 필요한 힙 메모리가 부족하여 작업이 생성되지 않았다는 것을 나타냅니다. 자세한 내용은 [힙 메모리 관리 \(p. 13\)](#) 단원을 참조하십시오.

## 작업 생성(예제 1)

이번 예제에서는 단순 작업 2개를 생성하여 실행을 시작하는 데 필요한 단계에 대해서 설명합니다. 작업은 조잡한 NULL 루프를 사용해 주기 지연 시간을 생성하여 문자열을 주기적으로 출력합니다. 두 작업은 동일한 우선순위로 생성되며 출력하는 문자열을 제외하면 동일합니다. 다음 코드는 예제 1에서 사용하는 첫 번째 작업의 구현체를 나타낸 것입니다.

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

    volatile uint32_t ul; /* volatile to ensure ul is not optimized away.*/

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation. There is nothing to do in
            here. Later examples will replace this crude loop with a proper delay/sleep function. */

        }
    }
}
```

다음 코드는 예제 1에서 사용하는 두 번째 작업의 구현체를 나타낸 것입니다.

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

    volatile uint32_t ul; /* volatile to ensure ul is not optimized away.*/

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
```

```
    /* Print out the name of this task. */  
  
    vPrintString( pcTaskName );  
  
    /* Delay for a period. */  
  
    for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )  
    {  
  
        /* This loop is just a very crude delay implementation. There is nothing to do  
in here. Later examples will replace this crude loop with a proper delay/sleep function.  
*/  
  
    }  
  
}  
  
}
```

main() 함수는 스케줄러를 시작하기 전에 작업을 생성합니다.

```
int main( void )  
{  
  
    /* Create one of the two tasks. Note that a real application should check the return  
value of the xTaskCreate() call to ensure the task was created successfully. */  
  
    xTaskCreate( vTask1, /* Pointer to the function that implements the task. */ "Task 1", /*  
* Text name for the task. This is to facilitate debugging only. */ 1000, /* Stack depth -  
small microcontrollers will use much less stack than this. */ NULL, /* This example does  
not use the task parameter. */ 1, /* This task will run at priority 1. */ NULL ); /* This  
example does not use the task handle. */  
  
    /* Create the other task in exactly the same way and at the same priority. */  
  
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );  
  
    /* Start the scheduler so the tasks start executing. */  
  
    vTaskStartScheduler();  
  
    /* If all is well then main() will never reach here as the scheduler will now be  
running the tasks. If main() does reach here then it is likely that there was insufficient  
heap memory available for the idle task to be created. For more information, see Heap  
Memory Management. */  
  
    for( ;; );  
  
}
```

출력되는 화면은 다음과 같습니다.

```
C:\WINDOWS\system32\cmd.exe - rtosdemo
C:\Temp>rtosdemo
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
```

두 작업이 동시에 실행하는 것처럼 보입니다. 하지만 두 작업이 동일한 프로세서 코어에서 실행되기 때문에 동시에 실행될 수는 없습니다. 실제로 두 작업은 Running 상태의 시작과 종료를 매우 빠르게 반복하고 있습니다. 단지 두 작업의 실행 우선순위가 같기 때문에 동일한 프로세서 코어에서 시간을 공유할 뿐입니다. 다음 그림은 실행 패턴을 나타낸 것입니다.

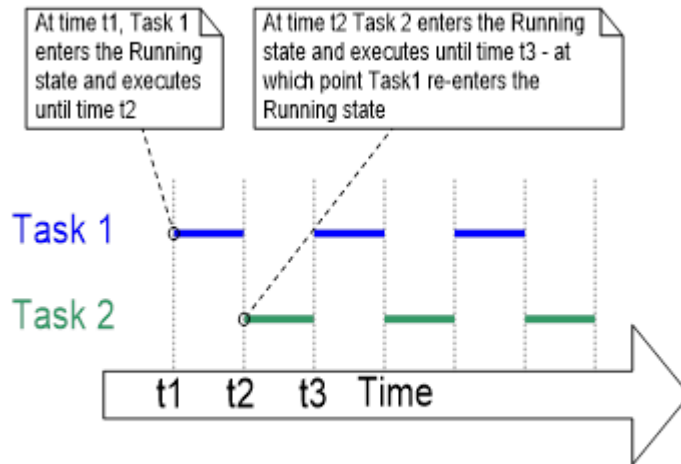


그림 하단의 화살표는 t1부터 시간의 흐름을 나타냅니다. 유색 선은 각 시점마다 실행되고 있는 작업을 나타냅니다(예를 들어 Task 1은 t1부터 t2까지 실행됩니다).

언제든지 Running 상태로 존재할 수 있는 작업은 1개뿐입니다. 따라서 작업 1개가 Running 상태로 전환되면(작업 스위치 인) 나머지 작업이 Not Running 상태로 전환됩니다(작업 스위치 아웃).

두 작업 모두 스케줄러 실행에 앞서 main()에서 생성됩니다. 다른 작업 내부에서 작업을 생성하는 것도 가능합니다. 예를 들어 Task 2는 Task 1 내부에서 생성되었을 수도 있습니다.

다음 코드는 스케줄러가 시작된 후 다른 작업에서 생성된 작업을 나타낸 것입니다.

```
void vTask1( void *pvParameters )
{
```

```
const char *pcTaskName = "Task 1 is running\r\n";

volatile uint32_t ul; /* volatile to ensure ul is not optimized away.*/

/* If this task code is executing then the scheduler must already have been started.
Create the other task before entering the infinite loop. */

xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

for( ;; )
{
    /* Print out the name of this task. */
    vPrintString( pcTaskName );

    /* Delay for a period. */
    for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
    {
        /* This loop is just a very crude delay implementation. There is nothing to do
in here. Later examples will replace this crude loop with a proper delay/sleep function.
*/
    }
}
}
```

## 작업 파라미터 사용(예제 2)

예제 1에서 생성된 작업 2개는 거의 동일합니다. 출력되는 텍스트 문자열이 유일하게 다를 뿐입니다. 이러한 중복은 단일 작업 구현체를 인스턴스 2개로 생성하면 제거할 수 있습니다. 그런 다음 작업 파라미터를 사용해 각 작업에게 출력해야 할 문자열을 전달하면 됩니다.

다음은 단일 작업 함수(vTaskFunction)의 코드입니다. 이 단일 함수는 예제 1에서 사용한 작업 함수 2개(vTask1, vTask2)를 대체하고 있습니다. 작업 파라미터는 char \*로 변환되어 작업이 출력해야 할 문자열을 가져옵니다.

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

    volatile uint32_t ul; /* volatile to ensure ul is not optimized away.*/

    /* The string to print out is passed in via the parameter. Cast this to a character
pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
```

```
/* Print out the name of this task. */
vPrintString( pcTaskName );

/* Delay for a period. */
for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
{
    /* This loop is just a very crude delay implementation. There is nothing to do
    in here. Later exercises will replace this crude loop with a delay/sleep function. */

}

}
```

이제는 작업 구현체가 1개(vTaskFunction)밖에 없지만 정의된 작업의 인스턴스를 여러 개 생성할 수 있습니다. 생성되는 인스턴스는 각각 FreeRTOS 스케줄러의 제어 하에서 독립적으로 실행됩니다.

다음 코드는 xTaskCreate() 함수에서 pvParameters 파라미터를 사용해 텍스트 문자열을 작업에게 전달하는 방법을 나타낸 것입니다.

```
/* Define the strings that will be passed in as the task parameters. These are defined
const and not on the stack to ensure they remain valid when the tasks are executing. */

static const char *pcTextForTask1 = "Task 1 is running\r\n";

static const char *pcTextForTask2 = "Task 2 is running\r\n";

int main( void )
{
    /* Create one of the two tasks. */

    xTaskCreate( vTaskFunction, /* Pointer to the function that implements the task.
    */ "Task 1", /* Text name for the task. This is to facilitate debugging only. */
    1000, /* Stack depth - small microcontrollers will use much less stack than this. */
    (void*)pcTextForTask1, /* Pass the text to be printed into the task using the task
    parameter. */ 1, /* This task will run at priority 1. */ NULL );
    /* The task handle is not used in this example. */

    /* Create the other task in exactly the same way. Note this time that multiple tasks
    are being created from the SAME task implementation (vTaskFunction). Only the value passed
    in the parameter is different. Two instances of the same task are being created. */

    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 1, NULL );

    /* Start the scheduler so the tasks start executing. */

    vTaskStartScheduler();

    /* If all is well then main() will never reach here as the scheduler will now be
    running the tasks. If main() does reach here then it is likely that there was insufficient
    heap memory available for the idle task to be created. For more information, see Heap
    Memory Management. */

    for( ;; );
}
```

## 작업 우선순위

xTaskCreate() API 함수의 uxPriority 파라미터는 생성된 작업에게 최초 우선순위를 할당합니다. 이렇게 할당된 우선순위는 스케줄러가 시작된 후에 vTaskPrioritySet() API 함수를 사용해 변경할 수 있습니다.

사용 가능한 우선순위의 최대 수치는 FreeRTOSConfig.h에서 애플리케이션 정의의 상수인 configMAX\_PRIORITIES 컴파일 시간 구성 상수로 설정합니다. 낮은 수치의 우선순위 값은 작업의 우선순위가 낮은 것을 나타내며, 값이 0일 때 우선순위가 가장 낮습니다. 따라서 사용 가능한 우선순위의 범위는 0부터 (configMAX\_PRIORITIES - 1)입니다. 동일한 우선순위를 공유할 수 있는 작업의 수에 제한이 없기 때문에 설계 유연성을 극대화할 수 있습니다.

FreeRTOS 스케줄러는 다음 두 가지 방법 중 하나를 사용해 Running 상태의 작업을 결정합니다. configMAX\_PRIORITIES에서 설정할 수 있는 최대 값은 사용하는 방법에 따라 달라집니다.

### 1. 일반적인 방법

일반적인 방법은 C로 구현되며, 모든 FreeRTOS 아키텍처 포트와 함께 사용할 수 있습니다.

일반적인 방법을 사용할 때는 FreeRTOS가 configMAX\_PRIORITIES에서 설정할 수 있는 최대 값을 제한하지 않습니다. 하지만 configMAX\_PRIORITIES 값은 필요에 따라 최소 값으로 유지하는 것이 좋습니다. 값이 높을수록 소비되는 RAM도 많을 뿐만 아니라 최악의 실행 시간이 길어질 수 있기 때문입니다.

이 방법은 FreeRTOSConfig.h에서 configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION이 0으로 설정되어 있거나, configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION이 정의되어 있지 않거나, 혹은 사용 중인 FreeRTOS 포트에서 유일하게 사용할 수 있는 방법이 일반적인 방법인 경우에 사용됩니다.

### 2. 아키텍처 최적화 방법

이 방법은 어셈블러 코드를 적게 사용하여 일반적인 방법보다 빠릅니다. configMAX\_PRIORITIES 설정은 최악의 실행 시간에 영향을 미치지 않습니다.

이 방법을 사용하는 경우에는 configMAX\_PRIORITIES 값이 32보다 클 수 없습니다. configMAX\_PRIORITIES 값은 일반적인 방법과 마찬가지로 필요에 따라 최소 값으로 유지해야 합니다. 값이 높을수록 소비되는 RAM이 많기 때문입니다.

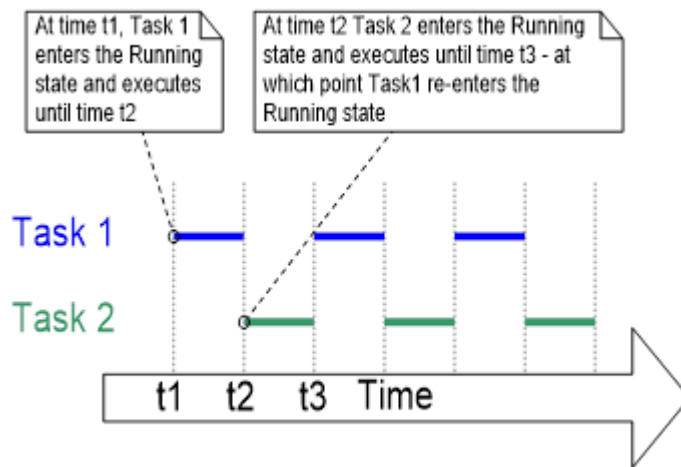
이 방법은 FreeRTOSConfig.h에서 configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION이 1로 설정된 경우에 사용됩니다.

FreeRTOS 포트라고 해서 모두 아키텍처 최적화 방법을 제공하는 것은 아닙니다.

FreeRTOS 스케줄러는 언제나 실행 가능한 작업 중에서 우선순위가 가장 높은 작업이 Running 상태로 전환할 작업으로 선택되도록 합니다. 동일한 우선순위의 작업이 다수이며, 모두 실행 가능한 상태일 때는 스케줄러가 각 작업을 차례로 Running 상태로 전환했다가 종료합니다.

## 시간 측정 및 틱 인터럽트

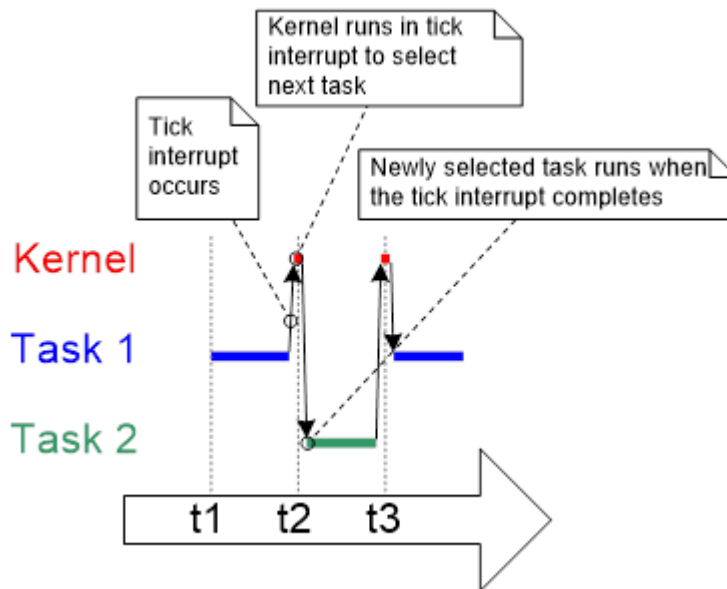
[스케줄링 알고리즘 \(p. 55\)](#) 단원에는 시간 분할이라고 하는 옵션 기능에 대한 설명이 있습니다. 시간 분할은 지금까지 소개한 예제에서도 사용되었습니다. 예제의 출력 화면에서 관찰되는 동작도 시간 분할입니다. 예제에서 작업 2개가 동일한 우선순위로 생성되었고, 항상 실행 가능한 상태였습니다. 따라서 각 작업은 시간 분할로 실행되어 시간 분할이 시작될 때 Running 상태로 전환되었다가 시간 분할이 끝날 때 Running 상태를 종료합니다. 아래 그림에서 t1과 t2 사이의 시간이 단일 시간 분할에 해당합니다.



다음에 실행할 작업을 선택하려면 스케줄러가 각 시간 분할의 끝에서 실행되어야 합니다. 하지만 스케줄러가 새롭게 실행할 작업을 선택할 수 있는 위치가 시간 분할의 끝만은 아닙니다. 스케줄러는 현재 실행 중인 작업이 Blocked 상태로 전환된 직후에, 혹은 인터럽트가 발생하여 높은 우선순위의 작업이 Ready 상태로 전환되었을 때 새롭게 실행할 작업을 선택하기도 합니다. 틱 인터럽트라는 주기적 인터럽트가 이러한 목적으로 사용됩니다. 시간 분할 길이는 틱 인터럽트 주파수를 통해 설정되며, 이 주파수는 FreeRTOSConfig.h에서 애플리케이션 정의의 상수인 configTICK\_RATE\_HZ 컴파일 시간 구성 상수로 구성됩니다. 예를 들어 configTICK\_RATE\_HZ가 100(Hz)으로 설정되면 시간 분할은 10밀리초가 됩니다. 틱 인터럽트 2개 사이의 시간은 틱 주기라고 부릅니다. 시간 분할 1개는 틱 주기 1개와 동일합니다.

다음 그림은 틱 인터럽트 실행을 설명하기 위해 실행 시퀀스를 확대하여 나타낸 것입니다. 상단 선은 스케줄러가 실행되고 있는 시간을 나타냅니다. 얇은 화살표는 작업에서 틱 인터럽트로, 다시 틱 인터럽트에서 다른 작업으로 실행되는 시퀀스를 나타냅니다.

최적의 configTICK\_RATE\_HZ 값은 개발하려는 애플리케이션에 따라 다르지만 일반적인 값으로 100이 사용됩니다.



FreeRTOS API 호출은 항상 틱 주기의 배수로 시간을 지정하며, 종종 틱으로 표현됩니다. pdMS\_TO\_TICKS() 매크로가 밀리초 단위로 지정된 시간을 틱 단위로 지정된 시간으로 변환합니다. 사용 가



능한 분해능은 정의된 틱 주파수에 따라 다릅니다. 틱 주파수가 1KHz보다 클 때는(configTICK\_RATE\_HZ가 1000보다 큰 경우) pdMS\_TO\_TICKS()를 사용하지 못합니다. 다음 코드는 pdMS\_TO\_TICKS()를 사용해 200 밀리초로 지정된 시간을 상응하는 틱 단위 시간으로 변환하는 방법을 나타낸 것입니다.

```
/* pdMS_TO_TICKS() takes a time in milliseconds as its only parameter, and evaluates to the
equivalent time in tick periods. This example shows xTimeInTicks being set to the number
of tick periods that are equivalent to 200 milliseconds. */

TickType_t xTimeInTicks = pdMS_TO_TICKS( 200 );
```

참고: 애플리케이션에서 시간을 틱 단위로 직접 지정하는 것은 좋지 않습니다. 대신에 pdMS\_TO\_TICKS() 매크로를 사용해 밀리초 단위의 시간으로 지정하십시오. 이렇게 해야만 틱 주파수를 변경하더라도 애플리케이션에서 지정한 시간이 바뀌지 않습니다.

'틱 카운트' 값은 스케줄러가 시작된 이후 발생한 틱 인터럽트의 총 수이며, 이때 틱 카운트가 오버플로우되지 않았다는 가정을 전제로 합니다. 사용자 애플리케이션에서 지연 시간을 지정할 때는 오버플로우를 고려할 필요 없습니다. FreeRTOS는 시간 일관성을 내부에서 관리하기 때문입니다.

스케줄러가 새롭게 실행할 작업을 선택하는 시간과 틱 인터럽트가 실행되는 시간에 영향을 미치는 구성 상수에 대한 자세한 내용은 [스케줄링 알고리즘 \(p. 55\)](#) 단원을 참조하십시오.

## 우선순위 실험(예제 3)

스케줄러는 언제나 실행 가능한 작업 중에서 우선순위가 가장 높은 작업이 Running 상태로 전환할 작업으로 선택되도록 합니다. 지금까지 사용된 예제에서는 작업 2개가 동일한 우선순위로 생성되었기 때문에 Running 상태로 번갈아 전환되었다 종료되었습니다. 이번 예제는 예제 2에서 생성된 작업 2개 중 하나의 우선순위가 바뀌었을 때 어떻게 되는지 나타낸 것입니다. 이번에는 첫 번째 작업이 우선순위 1로, 그리고 두 번째 작업이 우선순위 2로 생성됩니다.

작업을 다른 우선순위로 생성하는 샘플 코드는 다음과 같습니다. 두 작업을 구현하는 단일 함수는 바뀌지 않았습니다. NULL 루프를 사용해 지연 시간을 생성하여 문자열을 주기적으로 출력하는 것도 그대로입니다.

```
/* Define the strings that will be passed in as the task parameters. These are defined
const and not on the stack to ensure they remain valid when the tasks are executing. */

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

int main( void )
{
    /* Create the first task at priority 1. The priority is the second to last parameter.
    */
    xTaskCreate( vTaskFunction, "Task 1", 1000, (void*)pcTextForTask1, 1, NULL );

    /* Create the second task at priority 2, which is higher than a priority of 1. The
    priority is the second to last parameter. */
    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 2, NULL );

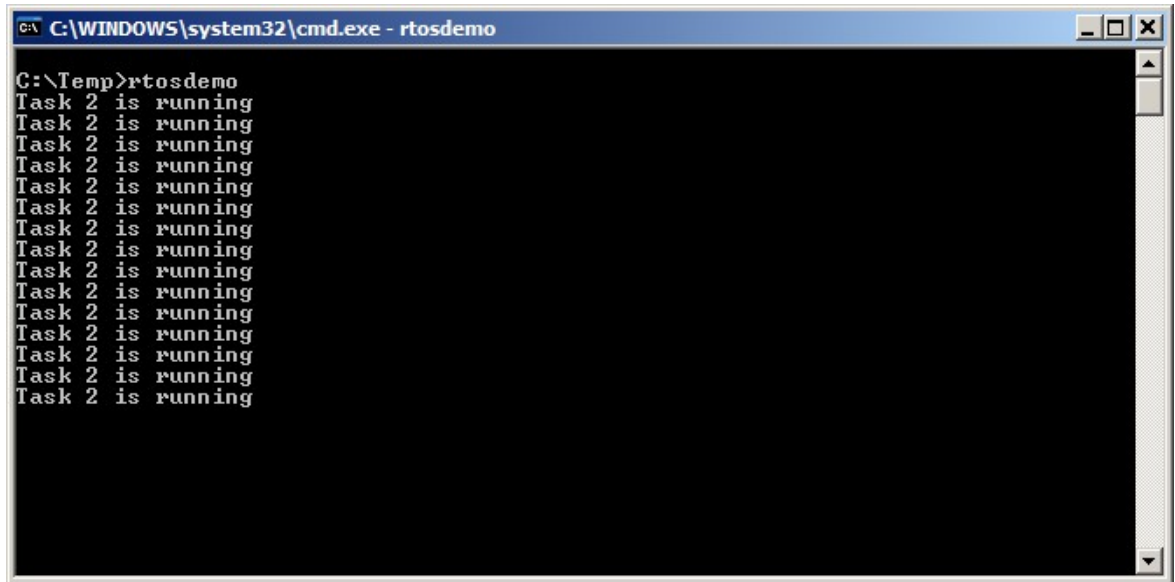
    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

    /* Will not reach here. */

    return 0;
```

```
}
```

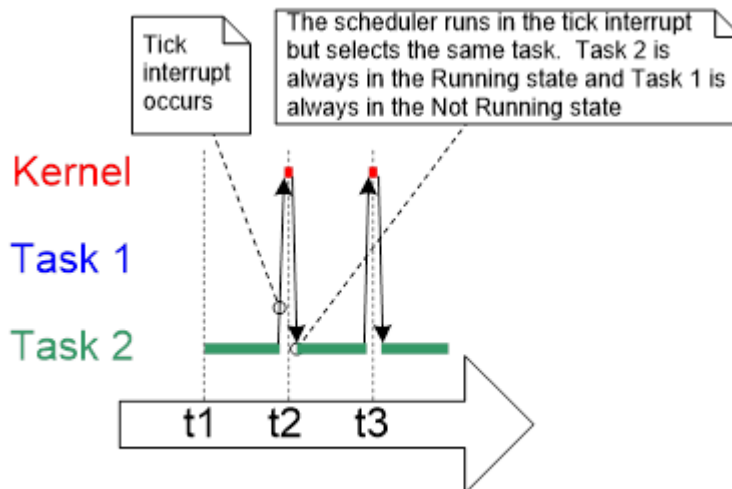
이번 예제에서 출력되는 화면은 다음과 같습니다. 스케줄러는 언제나 실행 가능한 작업 중에서 우선순위가 가장 높은 작업을 선택합니다. Task 2는 Task 1보다 우선순위가 높기 때문에 언제든지 실행 가능한 상태입니다. 따라서 Task 2가 유일하게 Running 상태로 전환할 수 있는 작업입니다. Task 1은 Running 상태로 전환되지 않아 문자열을 출력하지 않습니다. 이때 Task 1은 Task 2로 인해 처리 시간이 고갈되었다고 합니다.



```
C:\Temp>rtosdemo
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
Task 2 is running
```

Task 2는 대기하는 것이 없기 때문에 언제든지 실행 가능하여 NULL 루프를 따라 순환하거나, 혹은 터미널로 출력합니다.

다음 그림은 이전 샘플 코드의 실행 시퀀스를 나타낸 것입니다.



## Not Running 상태의 확장

지금까지 생성된 작업은 항상 처리해야 할 작업이 있어서 따로 대기해야 할 필요가 없었습니다. 따로 대기해야 할 필요가 없기 때문에 항상 Running 상태로 전환될 수 있습니다. 이러한 유형의 연속 처리 작업은 가장

낮은 우선순위로만 생성할 수 있기 때문에 유용성이 제한적입니다. 다른 우선순위에서 실행될 경우 더욱 낮은 우선순위의 작업은 전혀 실행되지 못하기 때문입니다.

작업 유용하게 사용하려면 이벤트 중심으로 다시 작성해야 합니다. 이벤트 중심의 작업은 이벤트가 발생해야만 실행할 처리 작업이 있기 때문에 이벤트가 발생하기 전에는 Running 상태로 전환되지 않습니다. 스케줄러는 언제나 실행 가능한 작업 중에서 우선순위가 가장 높은 작업을 선택합니다. 높은 우선순위의 작업을 실행하지 못하는 것은 스케줄러가 높은 우선순위의 작업을 선택하지 못하고 낮은 우선순위의 작업을 대신 선택해야 한다는 것을 의미합니다. 따라서 이벤트 중심의 작업을 사용할 경우 가장 높은 우선순위의 작업으로 인해 이보다 낮은 우선순위의 작업 처리 시간이 고갈되지 않도록 우선순위가 다르게 작업을 생성할 수 있습니다.

## Blocked 상태

이벤트가 발생할 때까지 대기하는 작업은 Not Running 상태의 하위 상태인 Blocked 상태가 되었다고 합니다.

작업은 Blocked 상태로 전환되어 다음과 같이 두 가지 유형이 이벤트가 발생할 때까지 대기할 수 있습니다.

1. 한시적(시간 관련) 이벤트: 지연 시간이 지나거나 절대 시간에 도달하여 발생하는 이벤트입니다. 예를 들어 작업은 Blocked 상태로 전환되어 10밀리초가 지날 때까지 대기할 수 있습니다.
2. 동기화 이벤트: 다른 작업 또는 인터럽트에서 발생하는 이벤트입니다. 예를 들어 작업은 Blocked 상태로 전환되어 데이터가 대기열에 수신될 때까지 대기할 수 있습니다. 동기화 이벤트는 광범위한 이벤트 유형에 적용됩니다.

FreeRTOS 대기열, 이진 세마포어, 계수 세마포어, 뮤텝스, 재귀적 뮤텝스, 이벤트 그룹, 작업 직접 알림 등은 모두 동기화 이벤트를 생성하는 데 사용 가능합니다.

작업은 제한 시간을 사용해 동기화 이벤트에서 차단하는 것도 가능하여 두 가지 유형의 이벤트에서 동시에 효과적으로 차단할 수 있습니다. 예를 들어 작업은 데이터가 대기열에 수신될 때까지 최대 10밀리초를 대기할 수 있습니다. 10밀리초 이내에 데이터가 수신되거나, 혹은 데이터가 수신되지 않더라도 10밀리초가 지나면 작업이 Blocked 상태를 종료합니다.

## Suspended 상태

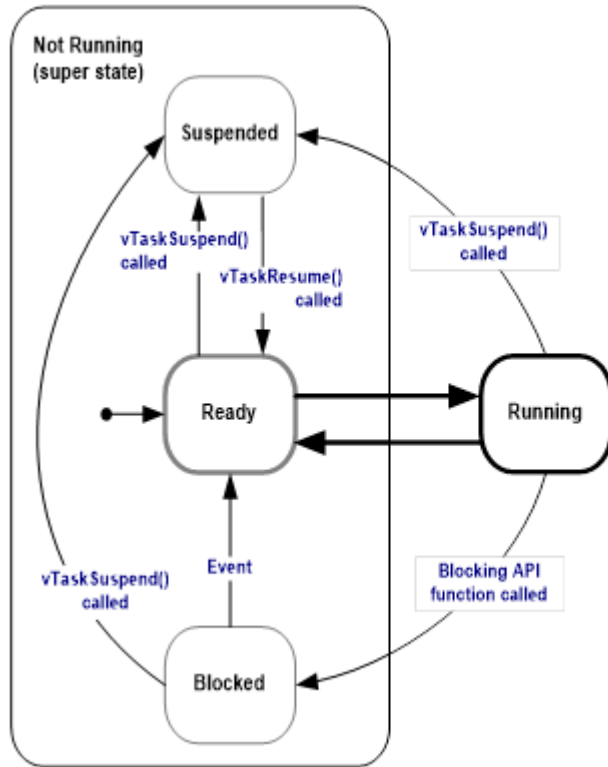
Suspended 역시 Not Running의 하위 상태입니다. Suspended 상태의 작업은 스케줄러에서 사용할 수 없습니다. Suspended 상태로 전환하려면 `vTaskSuspend()` API 함수를 호출하는 방법이 유일합니다. Suspended 상태를 종료할 때도 `vTaskResume()` 또는 `xTaskResumeFromISR()` API 함수를 호출하는 방법 밖에 없습니다. 대부분 애플리케이션은 Suspended 상태를 사용하지 않습니다.

## Ready 상태

Not Running 상태이지만 Blocked 또는 Suspended 상태가 아닌 작업을 Ready 상태라고 합니다. Ready 상태의 작업은 실행 가능하여 언제든지 실행할 준비가 되어 있지만 아직 Running 상태가 아닙니다.

## 상태 전환 다이어그램의 완성

다음 그림은 지나치게 단순화된 상태 다이어그램에 이번 단원에서 설명한 Not Running의 하위 상태를 모두 추가하여 확대한 것입니다. 지금까지 예제에서 생성된 작업은 Blocked 또는 Suspended 상태를 사용하지 않았습니다. 아래 그림에서 굵은 선으로 표시한 것처럼 Ready 상태와 Running 상태 사이에서 전환된 것 뿐입니다.



## Blocked 상태를 사용하여 지연 시간 생성(예제 4)

지금까지 예제에서 생성한 작업은 모두 주기적이었습니다. 한 주기 지연된 후 한 번 더 지연되기 전에 문자열을 출력했습니다. 지연 시간은 NULL 루프를 사용해 매우 조잡하게 생성되었습니다. 작업은 고정 값에 도달할 때까지 증가하는 루프 카운터를 효과적으로 폴링하였습니다. 예제 3에서는 이러한 방법의 단점에 대해 알기 쉽게 설명했습니다. 높은 우선순위의 작업이 NULL 루프를 따라 실행되면서 Running 상태를 유지하여 낮은 우선순위의 작업은 처리 시간이 모두 고갈되었습니다.

폴링 방식은 비효율성이 매우 큰 문제지만 그 밖에도 몇 가지 단점이 있습니다. 폴링 과정에서 작업이 따로 실행하는 작업이 없는데도 불구하고 최대 처리 시간을 사용하여 프로세서 주기만 낭비하게 됩니다. 예제 4에서는 폴링 NULL 루프 대신에 `vTaskDelay()` API 함수를 호출하여 이러한 동작 문제를 해결합니다. `vTaskDelay()` API 함수는 `FreeRTOSConfig.h`에서 `INCLUDE_vTaskDelay`를 1로 설정해야만 사용할 수 있습니다.

`vTaskDelay()`는 고정된 틱 인터럽트 수가 지날 때까지 호출 작업을 Blocked 상태로 전환합니다. 작업이 Blocked 상태일 때는 처리 시간을 사용하지 않고 실제로 처리할 작업이 있을 때만 처리 시간을 사용합니다.

`vTaskDelay()` API 함수 프로토타입은 다음과 같습니다.

```
void vTaskDelay( TickType_t xTicksToDelay );
```

다음 표는 `vTaskDelay()` 파라미터를 나열한 것입니다.

파라미터 이름	설명
<code>xTicksToDelay</code>	호출 작업이 Ready 상태로 다시 전환되기 전에 Blocked 상태를 유지하는 틱 인터럽트 수입니다.

예를 들어 작업이 틱 카운트가 10,000에 도달했을 때 `vTaskDelay( 100 )`을 호출하였다고 가정할 경우 즉시 Blocked 상태로 전환되어 틱 카운트가 10,100에 이를 때까지 Blocked 상태를 유지합니다.

밀리초 단위의 시간은 `pdMS_TO_TICKS()` 매크로를 사용해 틱으로 변환할 수 있습니다. 예를 들어 `vTaskDelay( pdMS_TO_TICKS( 100 ) )`를 호출하면 호출 작업이 100밀리초 동안 Blocked 상태를 유지합니다.

다음 코드에서 작업 예제는 NULL 루프 지연 시간이 지나 `vTaskDelay()` 호출로 바뀌었습니다. 이 코드는 새로운 작업 정의를 나타낸 것입니다.

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

    const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );

    /* The string to print out is passed in via the parameter. Cast this to a character
    pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. This time a call to vTaskDelay() is used which places the
        task into the Blocked state until the delay period has expired. The parameter takes a
        time specified in 'ticks', and the pdMS_TO_TICKS() macro is used (where the xDelay250ms
        constant is declared) to convert 250 milliseconds into an equivalent time in ticks. */
        vTaskDelay( xDelay250ms );
    }
}
```

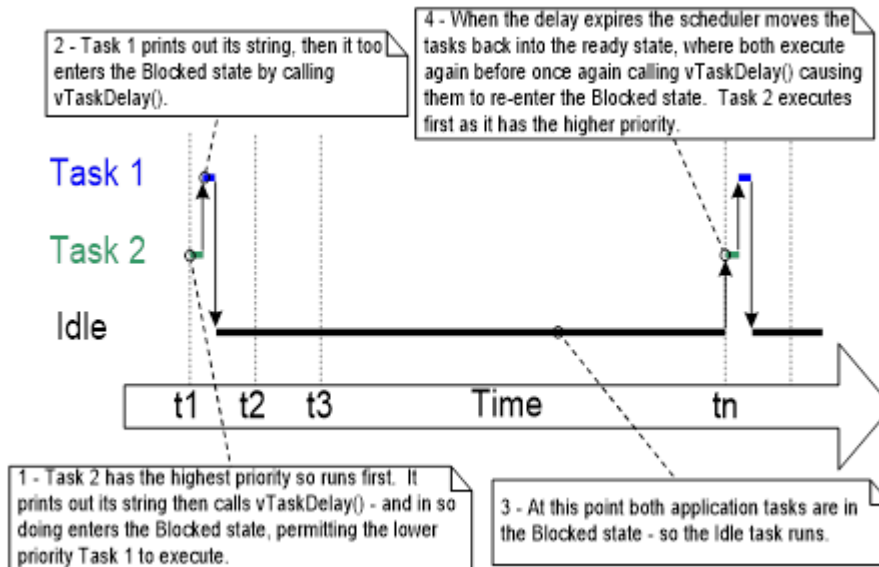
두 작업이 여전히 다른 우선순위로 생성되고 있지만 둘 다 실행됩니다. 아래 출력 화면에서 예상했던 동작을 확인할 수 있습니다.

```
C:\WINDOWS\system32\cmd.exe - rtsdemo
C:\Temp>rtsdemo
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
```

다음 그림의 실행 시퀀스는 작업 2개가 다른 우선순위로 생성되었지만 모두 실행되는 이유를 설명하고 있습니다. 쉽게 이해할 수 있도록 스케줄러 자체의 실행은 제외했습니다.

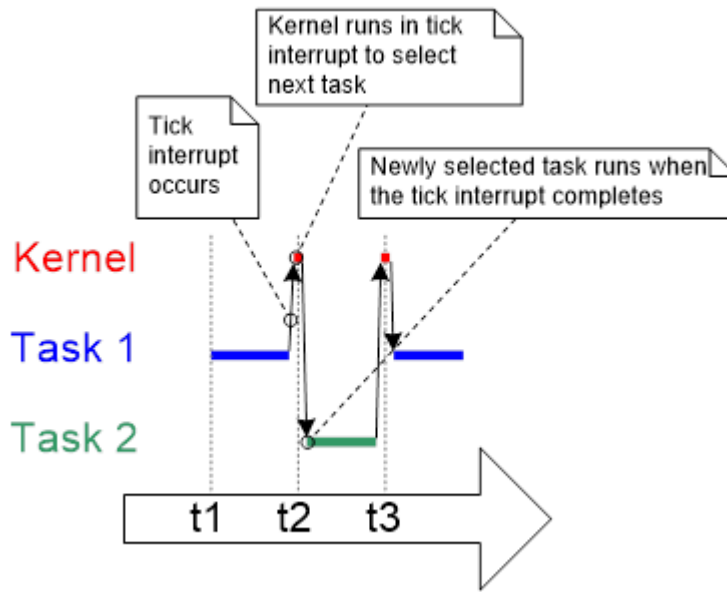
스케줄러 시작과 함께 유휴 작업이 자동으로 생성되어 실행 가능한 작업이 적어도 1개(Ready 상태의 작업이 적어도 1개)는 항상 존재합니다. 유휴 작업에 대한 자세한 내용은 [유휴 작업 및 유휴 작업 후크 \(p. 45\)](#) 단원을 참조하십시오.

다음 그림은 작업이 NULL 루프 대신에 `vTaskDelay()`를 사용할 때 실행 시퀀스를 나타낸 것입니다.



두 작업의 구현체만 바뀌었을 뿐 기능은 그대로입니다. 하지만 위 그림을 [시간 측정 및 틱 인터럽트 \(p. 32\)](#) 단원의 그림과 비교해보면 이 기능이 훨씬 더 효율적으로 구현된 것을 알 수 있습니다.

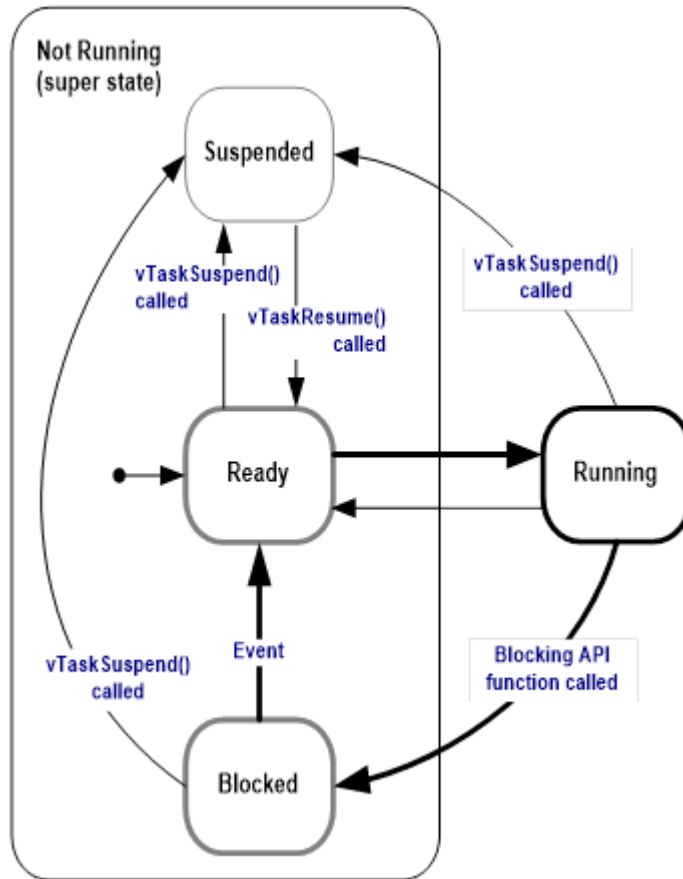
다음 그림은 작업이 전체 지연 시간 동안 Blocked 상태로 전환되는 경우 실행 패턴을 나타낸 것입니다. 여기에서는 실제로 실행해야 할 작업(이 경우에는 메시지 출력)이 있을 때만 작업이 처리 시간을 사용합니다.



작업이 Blocked 상태를 종료할 때마다 다시 Blocked 상태로 전환되기 전에 틱 주기에서 짧은 시간 동안 실행됩니다. 대부분 시간은 실행할 수 있는 애플리케이션 작업(Ready 상태의 애플리케이션 작업)이 없기 때문에 Running 상태로 선택할 수 있는 애플리케이션 작업도 없습니다. 이때는 유향 작업이 실행됩니다. 유향 작업에 할당되는 처리 시간은 시스템의 예비 처리 용량입니다. RTOS를 사용하면 애플리케이션을 완전히 이벤트 중심으로 설계하는 것만으로도 예비 처리 용량을 크게 늘릴 수 있습니다.

다음 그림에서 굵은 선은 예제 4의 작업에서 발생하는 전환을 나타낸 것으로, 이제는 각 작업이 Ready 상태로 돌아가기 전에 Blocked 상태로 전환되고 있습니다.

굵은 선은 작업에서 발생하는 상태 전환을 나타냅니다.



## vTaskDelayUntil() API 함수

vTaskDelayUntil()은 vTaskDelay()와 비슷합니다. vTaskDelay() 파라미터는 vTaskDelay()를 호출하는 작업과 다시 한 번 Blocked 상태를 종료하고 전환되는 동일 작업 사이에서 발생해야 하는 틱 인터럽트 수를 지정합니다. 작업이 Blocked 상태를 유지하는 시간의 길이는 vTaskDelay() 파라미터에서 지정합니다. 하지만 작업이 Blocked 상태를 종료하는 시간은 vTaskDelay()가 호출된 시간에 비례합니다.

vTaskDelayUntil()의 파라미터들은 호출 작업이 Blocked 상태에서 Ready 상태로 전환되어야 하는 틱 카운트 값을 정확하게 지정합니다. vTaskDelayUntil()은 고정된 실행 주기가 필요할 때(작업이 고정된 주기에 따라 주기적으로 실행되어야 하는 경우) 사용해야 하는 API 함수입니다. 호출 작업이 차단 해제되는 시간이 함수(이 경우 vTaskDelay())가 호출된 시간에 비례하기보다는 절대적이기 때문입니다.

vTaskDelayUntil() API 함수 프로토타입은 다음과 같습니다.

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

다음 표는 vTaskDelayUntil() 파라미터를 나열한 것입니다.

파라미터 이름	설명
pxPreviousWakeTime	이 파라미터는 vTaskDelayUntil()이 고정된 주기에 따라 주기적으로 실행되는 작업을 구현하는 데 사용된다는 가정을 전제로 지정됩니다. 이 경우에는 작업이 마지막으로 Blocked 상태를 종료한(대기를 끝낸) 시간이 pxPreviousWakeTime에 저장됩니다. 이



	<p>시간은 작업이 다음에 Blocked 상태를 종료해야 하는 시간을 계산하기 위한 기준으로 사용됩니다.</p> <p>pxPreviousWakeTime에서 가리키는 변수는 vTaskDelayUntil() 함수 내부에서 자동으로 업데이트됩니다. 일반적으로 애플리케이션 코드에서 수정하는 일이 많지 않지만 첫 번째 사용 이전에 현재 틱 카운트로 초기화해야 합니다.</p>
xTimeIncrement	<p>이 파라미터 역시 vTaskDelayUntil()이 고정된 주기에 따라 주기적으로 실행되는 작업을 구현하는데 사용된다는 가정을 전제로 지정됩니다. 주기는 xTimeIncrement 값에서 설정합니다.</p> <p>xTimeIncrement는 틱 단위로 지정됩니다. 밀리초 단위의 시간은 pdMS_TO_TICKS() 매크로를 사용해 틱으로 변환할 수 있습니다.</p>

## vTaskDelayUntil()을 사용하기 위한 작업 예제의 변환 (예제 5)

예제 4에서 생성한 작업 2개는 주기적 작업이지만 vTaskDelay()를 사용한다고 해서 실행 주기가 고정된다는 보장은 없습니다. 작업이 Blocked 상태를 종료하는 시간이 vTaskDelay() 호출 시간에 비례하기 때문입니다. 이때는 vTaskDelay() 대신에 vTaskDelayUntil()을 사용해 작업을 변환하면 잠재적 문제를 해결할 수 있습니다.

다음 코드는 vTaskDelayUntil()을 사용하는 작업 예제의 구현체를 나타낸 것입니다.

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

    TickType_t xLastWakeTime;

    /* The string to print out is passed in by the parameter. Cast this to a character
    pointer. */

    pcTaskName = ( char * ) pvParameters;

    /* The xLastWakeTime variable needs to be initialized with the current tick count.
    This is the only time the variable is written to explicitly. After this, xLastWakeTime is
    automatically updated within vTaskDelayUntil(). */

    xLastWakeTime = xTaskGetTickCount();

    /* As per most tasks, this task is implemented in an infinite loop. */

    for( ;; )
    {
        /* Print out the name of this task. */

        vPrintString( pcTaskName );

        /* This task should execute every 250 milliseconds exactly. As per the
        vTaskDelay() function, time is measured in ticks, and the pdMS_TO_TICKS() macro is
```

```
used to convert milliseconds into ticks. xLastWakeTime is automatically updated within
vTaskDelayUntil(), so is not explicitly updated by the task. */

    vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 250 ) );

}

}
```

## 차단 작업과 비차단 작업의 결합(예제 6)

이전 예제들에서는 폴링 작업과 차단 작업의 동작을 따로 살펴보았습니다. 이번 예제에서는 두 체계를 결합한 실행 시퀀스를 설명하면서 예상되는 시스템 동작을 중심으로 살펴보겠습니다.

1. 작업 2개가 우선순위 1로 생성됩니다. 두 작업은 문자열을 연속해서 출력하는 것 외에는 다른 작업이 없습니다.

두 작업은 자신을 Blocked 상태로 전환할 수 있는 API 함수를 절대로 호출하지 않기 때문에 항상 Ready 또는 Running 상태를 유지합니다. 이러한 작업을 연속 처리 작업라고 합니다. 항상 처리할 작업(여기에서는 간단한 작업이지만)이 있기 때문입니다.

2. 이후 세 번째 작업이 나머지 두 작업의 우선순위보다 높은 우선순위 2로 생성됩니다. 세 번째 작업 역시 문자열을 출력할 뿐이지만 이번에는 주기적으로 실행되기 때문에 vTaskDelayUntil() API 함수를 사용해 반복되는 출력 사이에 Blocked 상태로 전환됩니다.

다음 코드는 연속 처리 작업을 나타낸 것입니다.

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

    /* The string to print out is passed in by the parameter. Cast this to a character
    pointer. */

    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */

    for( ;; )
    {
        /* Print out the name of this task. This task just does this repeatedly without
        ever blocking or delaying. */

        vPrintString( pcTaskName );
    }
}
```

다음 코드는 주기적 작업을 나타낸 것입니다.

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

```
const TickType_t xDelay3ms = pdMS_TO_TICKS( 3 );

/* The xLastWakeTime variable needs to be initialized with the current tick count. Note
that this is the only time the variable is explicitly written to. After this xLastWakeTime
is managed automatically by the vTaskDelayUntil() API function. */

xLastWakeTime = xTaskGetTickCount();

/* As per most tasks, this task is implemented in an infinite loop. */

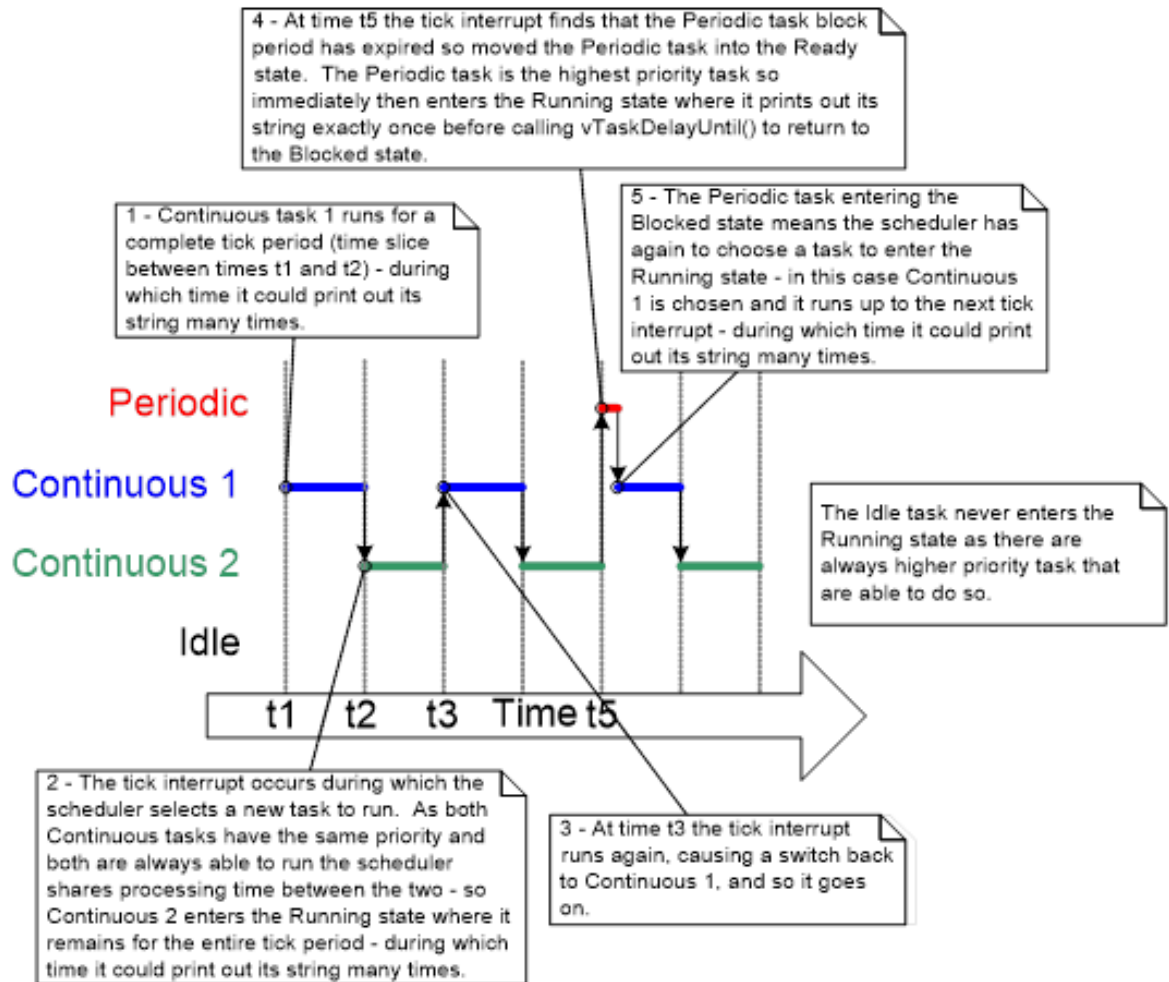
for( ;; )
{
    /* Print out the name of this task. */

    vPrintString( "Periodic task is running\r\n" );

    /* The task should execute every 3 milliseconds exactly. See the declaration of
    xDelay3ms in this function. */

    vTaskDelayUntil( &xLastWakeTime, xDelay3ms );
}
}
```

다음 그림은 실행 시퀀스를 나타낸 것입니다.



## 유휴 작업 및 유휴 작업 후크

예제 4에서 생성한 작업은 대부분 시간을 Blocked 상태로 보냅니다. Blocked 상태일 때는 실행되지 않기 때문에 스케줄러에서 선택할 수도 없습니다.

하지만 Running 상태로 전환할 수 있는 작업이 적어도 1개는 있어야 합니다. FreeRTOS의 저전력 기능을 사용할 때가 바로 그렇습니다. 애플리케이션에서 생성된 작업 중 실행 가능한 작업이 하나도 없을 경우 FreeRTOS의 실행 기반인 마이크로컨트롤러가 저전력 모드로 전환됩니다.

적어도 작업 1개가 Running 상태로 전환될 수 있도록 스케줄러가 vTaskStartScheduler() 호출 시 유휴 작업을 생성합니다. 유휴 작업은 루프를 따르는 것 외에는 거의 아무것도 하지 않기 때문에 첫 번째 예제의 작업처럼 항상 실행 가능한 상태가 됩니다.

유휴 작업의 우선순위는 높은 우선순위의 작업이 Running 상태로 전환되는 것을 막지 못하도록 가장 낮게(우선순위 0) 설정됩니다. 또한 유휴 작업 우선순위로 작업을 생성하여 우선순위를 공유하는 것도 가능합니다. FreeRTOSConfig.h에서 configIDLE\_SHOULD\_YIELD 컴파일 시간 구성 상수를 사용하면 유휴 작업이 처리 시간을 소비하지 못하도록 막아 애플리케이션 작업에 더욱 생산적으로 할당할 수 있습니다. configIDLE\_SHOULD\_YIELD에 대한 자세한 내용은 [스케줄링 알고리즘 \(p. 55\)](#) 단원을 참조하십시오.

가장 낮은 우선순위로 실행하면 높은 우선순위의 작업이 Ready 상태로 전환되자마자 유휴 작업이 Running 상태를 종료할 수 있습니다. 예제 4 실행 시퀀스 그림의 tn에서 이렇게 되는 것을 볼 수 있습니다. 여기에서

Task 2의 Blocked 상태 종료와 함께 바로 실행될 수 있도록 유휴 작업이 즉시 스왑 아웃됩니다. 이때 Task 2가 유휴 작업을 선점하였다고 말합니다. 선점은 선점되는 작업에 대해 아무것도 모른 채 자동으로 발생합니다.

참고: 애플리케이션이 vTaskDelete() API 함수를 사용하는 경우에는 유휴 작업의 처리 시간이 고갈되지 않도록 해야 합니다. 유휴 작업은 작업이 삭제되면 커널 리소스를 해제해야 하기 때문입니다.

## 유휴 작업 후크 함수

유휴 후크(또는 유휴 콜백) 함수를 사용해 애플리케이션 고유의 기능을 유휴 작업에 직접 추가할 수 있습니다. 이 함수는 유휴 작업 루프가 반복될 때마다 한 번씩 유휴 작업에서 자동으로 호출됩니다.

유휴 작업 후크의 일반적인 용도는 다음과 같습니다.

- 낮은 우선순위, 백그라운드 또는 연속 처리 기능 실행
- 예비 처리 용량 측정. (유휴 작업은 높은 우선순위의 애플리케이션 작업이 모두 수행할 작업이 없을 때만 실행되기 때문에 유휴 작업에 할당되는 처리 시간을 측정하면 예비 처리 시간이 얼마나 많은지 쉽게 확인할 수 있습니다)
- 애플리케이션에 처리할 작업이 없을 때마다 프로세서의 저전력 모드 전환을 통해 전력을 자동으로 쉽게 절감할 수 있는 방법 제공. (타이머 미작동 모드(tickless idle mode)를 사용하면 더 많은 전력을 절감할 수 있습니다)

## 유휴 작업 후크 함수의 구현체에 대한 제한 사항

유휴 작업 후크 함수는 다음 규칙을 따라야 합니다.

1. 유휴 작업 후크 함수는 절대로 차단 또는 일시 중지해서는 안 됩니다.

참고: 어떤 식으로든 유휴 작업을 차단하면 모든 작업을 Running 상태로 전환하지 못하는 시나리오를 초래할 수 있습니다.

2. 애플리케이션이 vTaskDelete() API 함수를 사용할 경우에는 유휴 작업 후크가 항상 적정 시간 내에 호출자에게 복귀되어야 합니다. 유휴 작업은 작업이 삭제되면 커널 리소스를 해제해야 하기 때문입니다. 유휴 작업이 유휴 후크 함수에 계속해서 남게 되면 이러한 리소스 해제가 발생하지 않기 때문입니다.

유휴 작업 후크 함수의 이름과 프로토타입은 다음과 같아야 합니다.

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

## 유휴 작업 후크 함수의 정의(예제 7)

예제 4에서 차단하는 vTaskDelay() API 호출을 사용하여 애플리케이션 작업 2개의 Blocked 상태가 지속되면서 유휴 작업 실행에 따른 유휴 시간이 많이 발생했습니다. 예제 7은 다음과 같은 소스의 유휴 후크 함수를 추가하여 이러한 유휴 시간을 사용합니다.

다음 코드는 유휴 후크 함수를 매우 간단하게 나타낸 것입니다.

```
/* Declare a variable that will be incremented by the hook function.*/  
  
volatile uint32_t ulIdleCycleCount = 0UL;  
  
/* Idle hook functions MUST be called vApplicationIdleHook(), take no parameters, and  
return void. */
```

```
void vApplicationIdleHook( void )
{
    /* This hook function does nothing but increment a counter. */
    ulIdleCycleCount++;
}
```

To call the idle hook function, called configUSE\_IDLE\_HOOK must be set to 1 in FreeRTOSConfig.h.

생성된 작업을 구현하는 함수가 다음과 같이 약간 수정되어 ulIdleCycleCount 값을 출력합니다. 다음 코드는 ulIdleCycleCount 값을 출력하는 방법을 나타낸 것입니다.

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

    const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );

    /* The string to print out is passed in by the parameter. Cast this to a character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task AND the number of times ulIdleCycleCount has been incremented. */
        vPrintStringAndNumber( pcTaskName, ulIdleCycleCount );

        /* Delay for a period of 250 milliseconds. */
        vTaskDelay( xDelay250ms );
    }
}
```

예제 7에서 출력되는 화면은 다음과 같습니다. 화면을 보면 애플리케이션 작업이 반복될 때마다 유휴 작업 후크 함수가 약 400만 번씩 호출되고 있습니다. (반복 횟수는 데모가 실행되는 하드웨어의 속도에 따라 달라 집니다)

```
C:\WINDOWS\system32\cmd.exe - rtosdemo
C:\Temp>rtosdemo
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
```

## 작업 우선순위의 변경

### vTaskPrioritySet() API 함수

vTaskPrioritySet() API 함수는 스케줄러 시작 후 작업의 우선순위를 변경하는 데 사용됩니다.

vTaskPrioritySet() API 함수는 FreeRTOSConfig.h에서 INCLUDE\_vTaskPrioritySet를 1로 설정해야만 사용할 수 있습니다.

vTaskPrioritySet() API 함수 프로토타입은 다음과 같습니다.

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

다음 표는 vTaskPrioritySet() 파라미터를 나열한 것입니다.

파라미터 이름	설명
pxTask	우선순위를 수정할 작업(대상 작업)의 핸들입니다. 작업 핸들을 가져오는 방법에 대한 자세한 내용은 xTaskCreate() API 함수에서 pxCreatedTask 파라미터를 참조하십시오.  작업은 유효한 작업 핸들 대신에 NULL을 전달하여 자체 우선순위를 변경할 수 있습니다.
uxNewPriority	대상 작업에서 설정할 우선순위입니다. 우선순위는 최대 허용 우선순위인 (configMAX_PRIORITIES - 1)로 자동으로 제한되며, 여기에서 configMAX_PRIORITIES는 FreeRTOSConfig.h 헤더 파일에서 설정하는 컴파일 시간 상수입니다.

## uxTaskPriorityGet() API 함수

uxTaskPriorityGet() API 함수는 작업의 우선순위를 쿼리하는 데 사용됩니다. uxTaskPriorityGet() API 함수는 FreeRTOSConfig.h에서 INCLUDE\_uxTaskPriorityGet을 1로 설정해야만 사용할 수 있습니다.

uxTaskPriorityGet() API 함수 프로토타입은 다음과 같습니다.

```
UBaseType_t uxTaskPriorityGet( TaskHandle_t pxTask );
```

다음 표는 uxTaskPriorityGet() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
pxTask	우선순위를 쿼리할 작업(대상 작업)의 핸들입니다. 작업 핸들을 가져오는 방법에 대한 자세한 내용은 xTaskCreate() API 함수에서 pxCreatedTask 파라미터를 참조하십시오.  작업은 유효한 작업 핸들 대신에 NULL을 전달하여 자체 우선순위를 쿼리할 수 있습니다.
반환 값	현재 쿼리할 작업에 할당되어 있는 우선순위입니다.

## 작업 우선순위의 변경(예제 8)

스케줄러는 언제나 우선순위가 가장 높은 Ready 상태의 작업을 Running 상태로 전환할 작업으로 선택합니다. 예제 8에서는 vTaskPrioritySet() API 함수를 사용해 서로 관련된 작업 2개의 우선순위를 변경하여 이를 설명합니다.

여기에서 사용되는 샘플 코드는 다른 우선순위의 작업 2개를 생성합니다. 두 작업 모두 Blocked 상태로 전환될 수 있는 API 함수를 호출하지 않기 때문에 항상 Ready 또는 Running 상태를 유지합니다. 따라서 상대적으로 더욱 높은 우선순위의 작업이 항상 스케줄러에서 Running 상태로 선택됩니다.

1. Task 1(바로 아래 코드)이 가장 높은 우선순위로 생성되어 첫 번째로 실행됩니다. Task 1이 문자열을 출력한 후 Task 2의 우선순위를(아래 두 번째 코드) 자신의 우선순위보다 높입니다.
2. Task 2가 상대적으로 더욱 높은 우선순위를 받자마자 실행되기 시작합니다(Running 상태로 전환). 작업은 한 번에 하나만 Running 상태가 될 수 있기 때문에 Task 2가 Running 상태이고, Task 1은 Ready 상태입니다.
3. Task 2가 메시지를 출력하고 자신의 우선순위를 다시 Task 1보다 낮게 설정합니다.
4. Task 2가 우선순위를 다시 낮게 설정하면 Task 1이 다시 가장 높은 우선순위의 작업이 됩니다. Task 1가 Running 상태로 다시 전환되면서 Task 2는 Ready 상태가 됩니다.

```
/*The implementation of Task 1*/
void vTask1( void *pvParameters )
{
    UBaseType_t uxPriority;

    /* This task will always run before Task 2 because it is created with the higher
    priority. Task 1 and Task 2 never block, so both will always be in either the Running or
    the Ready state. Query the priority at which this task is running. Passing in NULL means
    "return the calling task's priority". */
```



```
uxPriority = uxTaskPriorityGet( NULL );

for( ;; )
{
    /* Print out the name of this task. */

    vPrintString( "Task 1 is running\r\n" );

    /* Setting the Task 2 priority above the Task 1 priority will cause Task 2 to
    immediately start running (Task 2 will have the higher priority of the two created tasks).
    Note the use of the handle to Task 2 (xTask2Handle) in the call to vTaskPrioritySet(). The
    code that follows shows how the handle was obtained. */

    vPrintString( "About to raise the Task 2 priority\r\n" );

    vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

    /* Task 1 will only run when it has a priority higher than Task 2. Therefore, for
    this task to reach this point, Task 2 must already have executed and set its priority back
    down to below the priority of this task. */

}
}
```

```
/*The implementation of Task 2 */

void vTask2( void *pvParameters )
{
    UBaseType_t uxPriority;

    /* Task 1 will always run before this task because Task 1 is created with the higher
    priority. Task 1 and Task 2 never block so they will always be in either the Running or
    the Ready state. Query the priority at which this task is running. Passing in NULL means
    "return the calling task's priority". */

    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )
    {
        /* For this task to reach this point Task 1 must have already run and set the
        priority of this task higher than its own. Print out the name of this task. */

        vPrintString( "Task 2 is running\r\n" );

        /* Set the priority of this task back down to its original value. Passing in NULL
        as the task handle means "change the priority of the calling task". Setting the priority
        below that of Task 1 will cause Task 1 to immediately start running again, preempting this
        task. */

        vPrintString( "About to lower the Task 2 priority\r\n" );

        vTaskPrioritySet( NULL, ( uxPriority - 2 ) );

    }
}
```

각 작업 모두 유효한 작업 핸들을 사용하지 않고 단순히 NULL만 사용해도 자신의 우선순위를 쿼리하고 설정할 수 있습니다. 작업 핸들은 Task 1이 Task 2의 우선순위를 변경할 때처럼 자신 외에 다른 작업을 참조할 때만 필요합니다. Task 1이 Task 2의 우선순위를 변경할 수 있도록 다음 코드 주석에도 나와있는 것처럼 Task 2 생성 시 Task 2 핸들을 가져와 저장합니다.

```
/* Declare a variable that is used to hold the handle of Task 2. */
TaskHandle_t xTask2Handle = NULL;

int main( void )
{
    /* Create the first task at priority 2. The task parameter is not used and set to NULL.
    The task handle is also not used so is also set to NULL. */
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 2, NULL );

    /* The task is created at priority 2 _____. */

    /* Create the second task at priority 1, which is lower than the priority given to
    Task 1. Again, the task parameter is not used so it is set to NULL, but this time the task
    handle is required so the address of xTask2Handle is passed in the last parameter. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, &xTask2Handle );

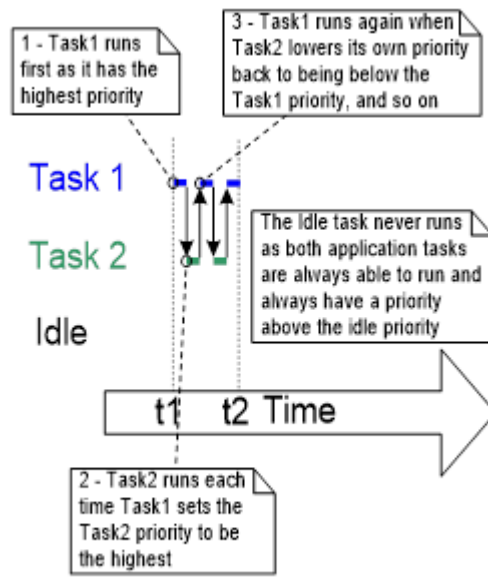
    /* The task handle is the last parameter _____ */

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

    /* If all is well, then main() will never reach here because the scheduler will now be
    running the tasks. If main() does reach here, then it is likely there was insufficient
    heap memory available for the idle task to be created. For information, see Heap Memory
    Management. */

    for( ;; );
}
```

다음 그림은 작업의 실행 시퀀스를 나타낸 것입니다.



출력되는 화면은 다음과 같습니다.

```
C:\Windows\system32\cmd.exe
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
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
```

## 작업 삭제

### vTaskDelete() API 함수

작업은 vTaskDelete() API 함수를 사용하여 자신 또는 다른 작업을 삭제할 수 있습니다. vTaskDelete() API 함수는 FreeRTOSConfig.h에서 INCLUDE\_vTaskDelete를 1로 설정해야만 사용할 수 있습니다.

삭제된 작업은 더 이상 존재하지 않기 때문에 다시 Running 상태로 전환되지 않습니다.

삭제된 작업에 할당되었던 메모리는 유휴 작업이 해제합니다. 따라서 vTaskDelete() API 함수를 사용하는 애플리케이션이 유휴 작업의 처리 시간을 완전히 고갈시키지 않도록 주의해야 합니다.

참고: 작업이 삭제되었을 때는 커널에서 작업에 할당한 메모리만 자동으로 해제됩니다. 작업 구현체에서 할당된 메모리나 기타 리소스는 명시적으로 해제해야 합니다.

vTaskDelete() API 함수 프로토타입은 다음과 같습니다.

```
void vTaskDelete( TaskHandle_t pxTaskToDelete );
```

다음 표는 vTaskDelete() 파라미터를 나열한 것입니다.

파라미터 이름/반환 값	설명
pxTaskToDelete	삭제할 작업(대상 작업)의 핸들입니다. 작업 핸들을 가져오는 방법에 대한 자세한 내용은 xTaskCreate() API 함수에서 pxCreatedTask 파라미터를 참조하십시오. 작업은 유효한 작업 핸들 대신에 NULL을 전달하여 자기 자신을 삭제할 수 있습니다.

## 작업 삭제(예제 9)

다음은 매우 간단한 예제입니다.

1. Task 1이 main()에서 우선순위 1로 생성됩니다. 실행되면서 Task 2를 우선순위 2로 생성합니다. 이제 Task 2의 우선순위가 가장 높기 때문에 바로 실행되기 시작합니다. main() 소스는 첫 번째 코드 목록에 표시됩니다. Task 1 소스는 두 번째 코드 목록에 표시됩니다.
2. Task 2는 자기 자신을 삭제하는 것 외에 다른 작업을 하지 않습니다. NULL을 vTaskDelete()에 전달하는 것으로도 삭제가 가능하지만 여기에서는 설명을 목적으로 작업 핸들을 사용합니다. Task 2 소스는 세 번째 코드 목록에 표시됩니다.
3. Task 2가 삭제되면 Task 1이 다시 가장 높은 우선순위의 작업이 되어 vTaskDelay()를 호출하여 잠시 차단되었던 지점부터 다시 실행됩니다.
4. Task 1이 Blocked 상태일 때는 유휴 작업이 실행되어 이미 삭제된 Task 2에 할당되었던 메모리를 해제합니다.
5. Task 1이 Blocked 상태를 종료하면 다시 가장 높은 우선순위의 Ready 상태 작업이 되어 유휴 작업을 선택합니다. Running 상태로 전환되면서 다시 Task 2를 생성하고, 똑같은 과정이 계속 됩니다.

```
int main( void )
{
    /* Create the first task at priority 1. The task parameter is not used so is set to
    NULL. The task handle is also not used so likewise is set to NULL. */
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 1, NULL );

    /* The task is created at priority 1 _____. */

    /* Start the scheduler so the task starts executing. */
    vTaskStartScheduler();

    /* main() should never reach here as the scheduler has been started.*/

    for( ;; );
}
```

```
TaskHandle_t xTask2Handle = NULL;

void vTask1( void *pvParameters )
{
    const TickType_t xDelay100ms = pdMS_TO_TICKS( 100UL );

    for( ;; )
    {
        /* Print out the name of this task. */

        vPrintString( "Task 1 is running\r\n" );

        /* Create task 2 at a higher priority. Again, the task parameter is not used so it
        is set to NULL, but this time the task handle is required so the address of xTask2Handle
        is passed as the last parameter. */

        xTaskCreate( vTask2, "Task 2", 1000, NULL, 2, &xTask2Handle );

        /* The task handle is the last parameter _____^ */

        /* Task 2 has/had the higher priority, so for Task 1 to reach here, Task 2 must
        have already executed and deleted itself. Delay for 100 milliseconds. */

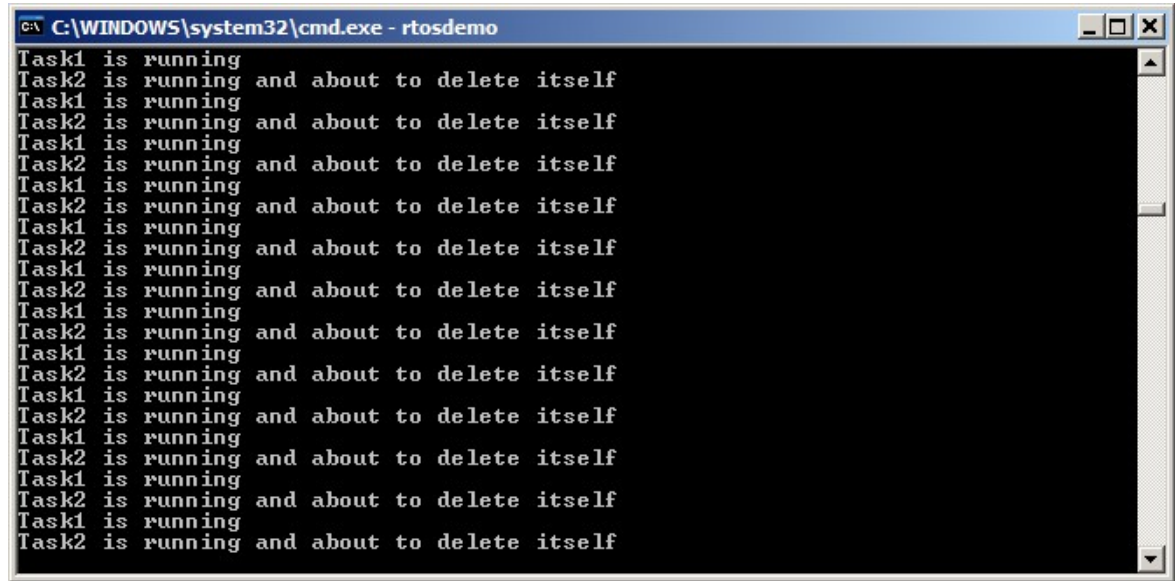
        vTaskDelay( xDelay100ms );
    }
}
```

```
void vTask2( void *pvParameters )
{
    /* Task 2 does nothing but delete itself. To do this, it could call vTaskDelete() using
    NULL as the parameter, but for demonstration purposes, it calls vTaskDelete(), passing its
    own task handle. */

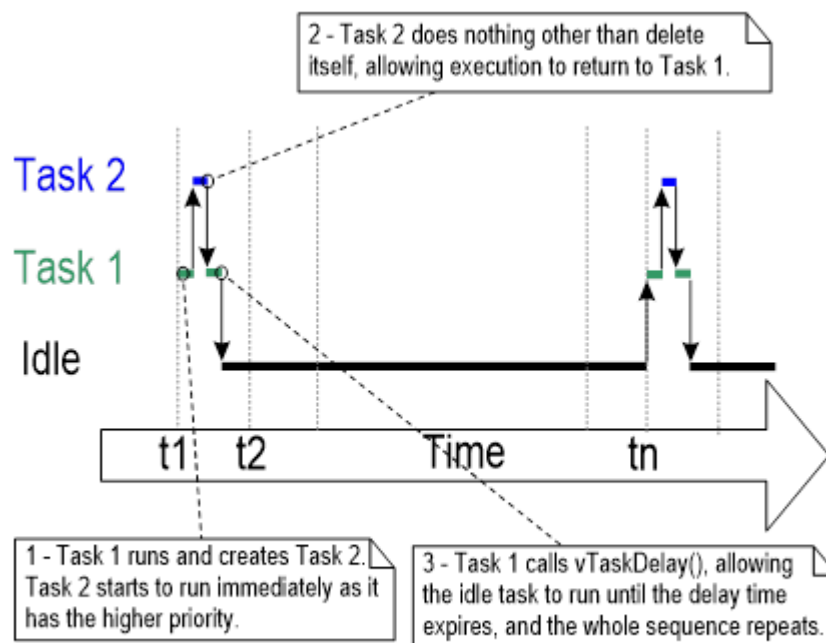
    vPrintString( "Task 2 is running and about to delete itself\r\n" );

    vTaskDelete( xTask2Handle );
}
```

출력되는 화면은 다음과 같습니다.



다음 그림은 실행 시퀀스를 나타낸 것입니다.



## 스케줄링 알고리즘

## 작업 상태 및 이벤트에 대한 요약

실제로 실행 중인(처리 시간을 사용 중인) 작업은 Running 상태입니다. 단일 코어 프로세서에서는 어느 때든  
지 Running 상태의 작업이 1개일 수 밖에 없습니다.

실행되지는 않지만 Blocked 또는 Suspended 상태가 아닌 작업은 Ready 상태입니다. Ready 상태인 작업은 스케줄러에서 Running 상태로 전환할 작업으로 선택할 수 있습니다. 스케줄러는 언제나 우선순위가 가장 높은 Ready 상태의 작업을 선택하여 Running 상태로 전환합니다.

작업은 이벤트가 발생할 때까지 Blocked 상태로 대기할 수 있으며, 이벤트가 발생하면 Ready 상태로 다시 자동 전환됩니다. 한시적 이벤트는 특정 시간(차단 시간이 지날 때 등)에 발생하기 때문에 대체로 주기적 동작이나 제한 시간 동작을 구현할 때 사용됩니다. 동기화 이벤트는 작업 또는 인터럽트 서비스 루틴이 작업 알림, 대기열, 이벤트 그룹 또는 여러 유형의 세마포어 중 하나를 사용해 정보를 전송할 때 발생합니다. 이러한 이벤트는 일반적으로 주변 장치로 데이터를 전송하는 등 비동기 작업 신호를 보낼 때 사용됩니다.

## 스케줄링 알고리즘 구성

스케줄링 알고리즘이란 Running 상태로 전환할 Ready 상태 작업을 결정하는 소프트웨어 루틴을 말합니다.

지금까지 모든 예제들이 동일한 스케줄링 알고리즘을 사용하고 있지만 configUSE\_PREEMPTION 및 configUSE\_TIME\_SLICING 구성 상수에서 변경할 수 있습니다. 두 상수는 모두 FreeRTOSConfig.h에서 정의됩니다.

세 번째 구성 상수인 configUSE\_TICKLESS\_IDLE 역시 스케줄링 알고리즘에 영향을 미칩니다. 이 상수를 사용하면 틱 인터럽트가 장시간 완전히 비활성화될 수 있습니다. configUSE\_TICKLESS\_IDLE은 특히 전력 소비를 최소화해야 하는 애플리케이션에서 사용할 수 있도록 제공되는 고급 옵션입니다. 이번 단원의 설명은 configUSE\_TICKLESS\_IDLE이 0으로, 즉 상수가 정의되지 않아서 기본 값으로 설정되어 있다는 가정을 전제로 합니다.

FreeRTOS 스케줄러는 가능한 모든 구성에서 우선순위를 공유하는 작업이 번갈아 선택되어 Running 상태로 전환될 수 있도록 합니다. 이렇게 순서대로 전환하는 정책을 라운드 로빈 스케줄링이라고 합니다. 라운드 로빈 스케줄링 알고리즘에서는 동일한 우선순위의 작업들이 시간을 동일하게 공유하지 않고 동일한 우선순위의 작업 중에서 Ready 상태 작업만 Running 상태로 번갈아 전환됩니다.

## 시간 분할을 사용한 우선순위 선점형 스케줄링

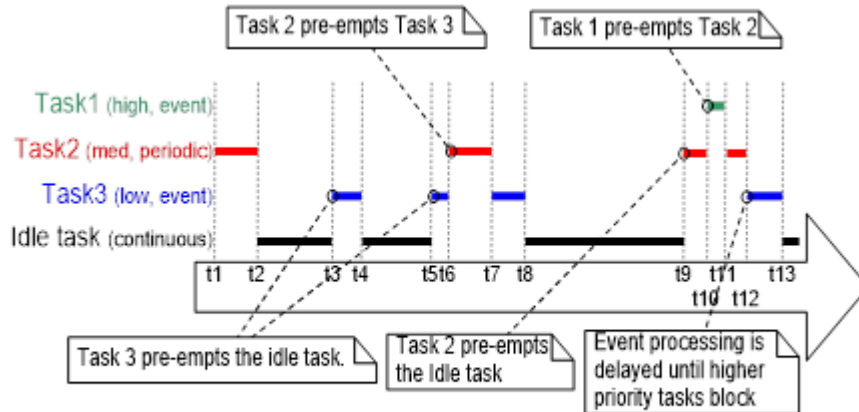
다음 표의 구성은 FreeRTOS 스케줄러가 시간 분할을 사용한 고정 우선순위 선점형 스케줄링이라고 하는 스케줄링 알고리즘을 사용하도록 설정합니다. 이 스케줄링 알고리즘은 대부분 소형 RTOS 애플리케이션에 사용될 뿐만 아니라 지금까지 소개한 모든 예제에서도 사용되었습니다.

상수	값
configUSE_PREEMPTION	1
configUSE_TIME_SLICING	1

다음은 알고리즘에서 사용되는 용어입니다.

- 고정 우선순위 작업: 고정 우선순위라고 하는 스케줄링 알고리즘은 예약된 작업에 할당된 우선순위를 변경하지 않지만 작업 자체가 자신이나 다른 작업의 우선순위를 변경하는 것을 막지는 않습니다.
- 선점 상태: 선점형 스케줄링 알고리즘은 우선순위가 더욱 높은 작업이 Ready 상태로 전환되는 경우 Running 상태의 작업을 즉시 선점합니다. 여기에서 선점이란 말은 강제로(명시적으로 넘겨주거나 차단되지 않고) Running 상태에서 Ready 상태로 전환되어 다른 작업이 Running 상태로 전환될 수 있다는 것을 의미합니다.
- 시간 분할: 시간 분할은 작업이 명시적으로 넘겨주거나 Blocked 상태로 전환되지 않을 때도 동일한 우선순위의 작업 사이에 처리 시간을 공유하는 데 사용됩니다. 시간 분할을 사용하는 스케줄링 알고리즘은 Running 작업과 우선순위가 동일하지만 현재는 Ready 상태인 다른 작업이 있는 경우 각 시간 분할이 끝날 때마다 새로운 작업을 선택하여 Running 상태로 전환합니다. 시간 분할은 RTOS 틱 인터럽트 2개 사이의 시간과 동일합니다.

다음 그림은 애플리케이션 작업마다 고유한 우선순위가 있을 때 어떤 작업을 Running 상태로 전환할지 선택하는 시퀀스를 나타낸 것입니다.



위의 그림은 작업에게 고유한 우선순위가 할당된 가상의 애플리케이션에서 작업 우선순위와 선점을 강조한 실행 패턴입니다.

#### 1. Idle task

Idle task는 가장 낮은 우선순위에서 실행되기 때문에 높은 우선순위의 작업이 Ready 상태로 전환될 때마다(예: t3, t5 및 t9에서) 선점됩니다.

#### 2. Task 3

Task 3은 이벤트 중심 작업이기 때문에 비교적 낮은 우선순위에서 실행되지만 Idle task의 우선순위보다는 높습니다. 대부분 시간을 이벤트가 발생할 때까지 Blocked 상태로 대기하다가 이벤트가 발생하면 Blocked 상태에서 Ready 상태로 전환됩니다. 모든 FreeRTOS 작업 통신 메커니즘(작업 알림, 대기열, 세마포어, 이벤트 그룹 등)을 사용해 이러한 방식으로 이벤트 신호를 전송하고 작업을 차단 해제할 수 있습니다.

이벤트는 t3과 t5에서, 그리고 t9~t12에서도 발생합니다. t3과 t5에서 발생하는 이벤트는 이때 실행 가능한 작업 중에서 가장 높은 우선순위의 작업이 Task 3이기 때문에 바로 처리됩니다. t9~t12에서 발생하는 이벤트는 이때 가장 높은 우선순위의 작업인 Task 1과 Task 2가 여전히 실행 중이기 때문에 t12가 될 때까지 처리되지 않습니다. Task 1과 Task 2가 모두 Blocked 상태가 되어 Task 3이 Ready 상태로 가장 높은 우선순위의 작업이 되는 시간은 t12가 유일합니다.

#### 3. Task 2

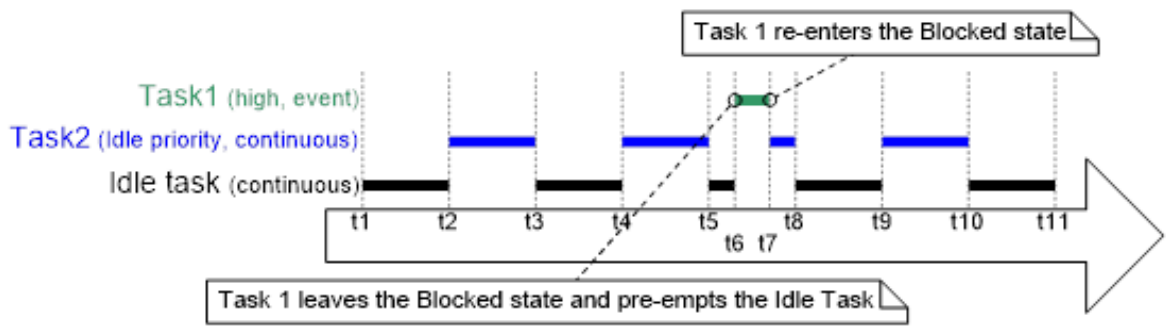
Task 2는 주기적 작업이기 때문에 Task 3의 우선순위보다 높은 우선순위에서 실행되지만 Task 1의 우선순위보다는 낮습니다. 작업의 주기가 있다는 말은 Task 2가 t1, t6 및 t9에서 실행된다는 것을 의미합니다. t6에서는 Task 3이 Running 상태이지만 Task 2의 우선순위가 상대적으로 더 높기 때문에 Task 2가 Task 3을 선점하여 바로 실행됩니다. Task 2가 처리를 마치고 t7에서 다시 Blocked 상태로 전환되고, 이때 Task 3은 다시 Running 상태로 전환되어 처리를 완료합니다. Task 3은 t8에서 차단됩니다.

#### 4. Task 1

Task 1 역시 이벤트 중심 작업입니다. 이는 모든 작업 중에서 가장 높은 우선순위로 실행되기 때문에 시스템의 다른 작업을 모두 선점할 수 있습니다. 유일하게 표시되어 있는 Task 1 이벤트가 t10에서 발생하면서 Task 1이 Task 2를 선점합니다. Task 2는 Task 1이 t11에서 다시 Blocked 상태로 전환된 후에야 이벤트 처리를 완료할 수 있습니다.

다음 그림은 작업 2개가 동일한 우선순위로 실행되는 가상의 애플리케이션에서 작업 우선순위와 시간 분할을 강조한 실행 패턴입니다.





## 1. Idle task와 Task 2

Idle task와 Task 2는 모두 연속 처리 작업이며, 둘 다 우선순위가 0입니다(가장 낮은 우선순위). 스케줄러는 실행 가능한 작업 중에서 높은 우선순위의 작업이 없을 때만 우선순위가 0인 작업들에게 처리 시간을 할당한 후 시간 분할을 통해 우선순위가 0인 작업들에게 할당된 시간을 동일하게 분배합니다. 새로운 시간 분할은 각 틱 인터럽트인 t1, t2, t3, t4, t5, t8, t9, t10 및 t11에서 시작됩니다.

Idle task와 Task 2는 번갈아 Running 상태로 전환되어 동일한 시간 분할 구간인 t5~t8에서 Running 상태가 될 수 있습니다.

## 2. Task 1

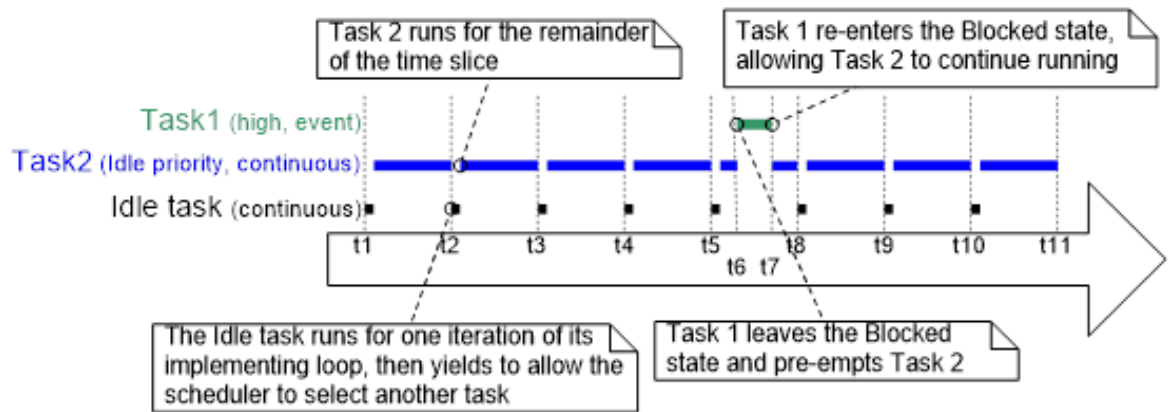
Task 1의 우선순위는 Idle task의 우선순위보다 높습니다. Task 1은 이벤트 중심 작업이기 때문에 대부분 시간을 이벤트가 발생할 때까지 Blocked 상태로 대기하다가 이벤트가 발생하면 Blocked 상태에서 Ready 상태로 전환됩니다. 해당하는 이벤트가 t6에서 발생하면 Task 1이 실행 가능한 작업 중에서 가장 높은 우선순위의 작업이 되어 시간 분할 중간에 Idle task를 선점합니다. 이벤트 처리가 t7에서 완료되고 Task 1은 다시 Blocked 상태로 전환됩니다.

위의 그림은 유휴 작업이 애플리케이션 개발자가 생성한 작업과 처리 시간을 공유하는 것을 나타냅니다. 애플리케이션 개발자가 생성한 유휴 작업이 해야 할 작업이 많지만 전혀 하지 않는다면 누구든지 그렇게 많은 처리 시간을 유휴 작업에게 할당하려고 하지 않습니다. 유휴 작업의 스케줄링 방식은 다음과 같이 configIDLE\_SHOULD\_YIELD 컴파일 시간 구성 상수를 사용해 변경할 수 있습니다.

- configIDLE\_SHOULD\_YIELD가 0으로 설정되면 유휴 작업이 높은 우선순위의 작업에게 선점되지 않는 한 시간 분할 전체가 끝날 때까지 Running 상태를 유지합니다.
- configIDLE\_SHOULD\_YIELD가 1로 설정되면 Ready 상태의 다른 유휴 작업이 있는 경우 유휴 작업이 루프가 반복될 때마다 넘겨줍니다(할당된 시간 분할에 무엇이 남아있든 자발적으로 포기합니다).

위 그림의 실행 패턴은 configIDLE\_SHOULD\_YIELD가 0으로 설정되었을 때 나타나는 패턴입니다.

다음 그림의 실행 패턴은 시나리오는 동일하지만 configIDLE\_SHOULD\_YIELD가 1으로 설정되었을 때 나타나는 패턴입니다. 애플리케이션 작업 2개가 우선순위를 공유할 때 어떤 작업을 Running 상태로 전환할지 선택하는 시퀀스를 나타낸 것입니다.



그 밖에도 위 그림은 configIDLE\_SHOULD\_YIELD가 0으로 설정되었을 때 Idle task 이후 Running 상태로 전환되도록 선택된 작업이 시간 분할 전체에서 실행되지는 않지만 Idle task가 넘겨준 동안에는 시간 분할에 무엇이 남겨지든 상관없이 실행되는 것을 나타내고 있습니다.

## 우선순위 선점형 스케줄링(시간 분할 미사용)

시간 분할을 사용하지 않는 우선순위 선점형 스케줄링은 이전 단원에서 설명한 것과 동일한 작업 선택 및 선점 알고리즘을 유지합니다. 다만 시간 분할을 사용하여 동일한 우선순위의 작업 사이에 처리 시간을 공유하지는 않습니다.

다음 표는 FreeRTOS 스케줄러가 시간 분할 없이 우선순위 선점형 스케줄링을 사용하도록 구성하는 FreeRTOSConfig.h 설정을 나열한 것입니다.

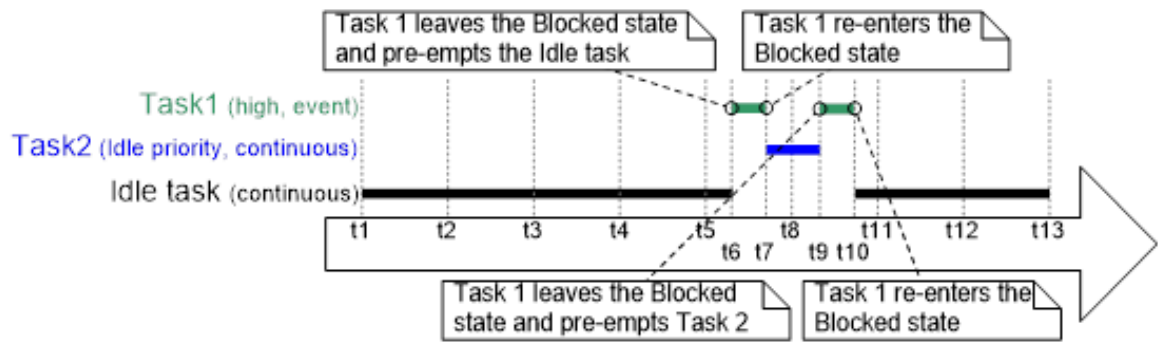
상수	값
configUSE_PREEMPTION	1
configUSE_TIME_SLICING	0

시간 분할을 사용하는 동시에 가장 높은 우선순위의 Ready 작업이 2개 이상인 경우에는 스케줄러가 각 RTOS 틱 인터럽트(시간 분할 끝에 표시되는 틱 인터럽트)마다 Running 상태로 전환할 작업을 새로 선택합니다. 시간 분할을 사용하지 않으면 스케줄러가 다음과 같은 경우에 한해 Running 상태로 전환할 작업을 새로 선택합니다.

- 더욱 높은 우선순위의 작업이 Ready 상태로 전환되는 경우
- Running 상태의 작업이 Blocked 또는 Suspended 상태로 전환되는 경우

시간 분할을 사용하지 않으면 작업의 컨텍스트 전환이 더욱 줄어듭니다. 따라서 시간 분할을 비활성화하면 스케줄러의 처리 오버헤드가 감소합니다. 하지만 처리 시간이 동일한 우선순위의 작업에게 각각 다르게 할당될 수도 있습니다. 시간 분할을 제외한 스케줄러 실행은 고급 기법으로 알려져 있습니다. 따라서 숙련된 사용자만 사용해야 합니다.

다음 그림은 시간 분할을 사용하지 않을 때 동일한 우선순위의 작업에게 처리 시간을 다르게 할당할 수 있는 방식을 나타낸 것입니다.



위 그림에서 configIDLE\_SHOULD\_YIELD는 0으로 설정됩니다.

### 1. 틱 인터럽트

틱 인터럽트는 t1, t2, t3, t4, t5, t8, t11, t12 및 t13에서 발생합니다.

### 2. Task 1

Task 1은 높은 우선순위의 이벤트 중심 작업이기 때문에 대부분 시간을 이벤트가 발생할 때까지 Blocked 상태로 대기합니다. 이후 이벤트가 발생하면 Blocked 상태에서 Ready 상태로 전환됩니다(가장 높은 우선순위의 Ready 상태 작업이기 때문에 이어서 Running 상태로 전환됩니다). 위 그림은 t6~t7에서, 그리고 t9~t10에서 다시 이벤트를 처리하는 Task 1을 나타낸 것입니다.

### 3. Idle task와 Task 2

Idle task와 Task 2는 모두 연속 처리 작업이며, 둘 다 우선순위가 0입니다(유휴 우선순위). 연속 처리 작업은 Blocked 상태로 전환되지 않습니다.

시간 분할을 사용하지 않기 때문에 Running 상태인 유휴 우선순위의 작업은 우선순위가 더 높은 Task 1에게 선점될 때까지 Running 상태를 유지합니다.

위 그림에서 Idle task는 t1에서 Running 상태로 전환된 후 틱 주기가 4회 이상 지난 시간인 t6에서 Task 1에게 선점될 때까지 Running 상태를 유지합니다.

Task 2는 Task 1이 다시 Blocked 상태로 전환되어 다른 이벤트가 발생할 때까지 대기하는 시간인 t7에서 실행되기 시작합니다. Running 상태로 전환된 후 틱 주기 1회에 못 미치는 t9에서 Task 1에게 선점될 때까지 Running 상태를 유지합니다.

이미 Task 2보다 처리 시간이 4배 이상 할당되었지만 Idle task가 t10에서 다시 Running 상태로 전환됩니다.

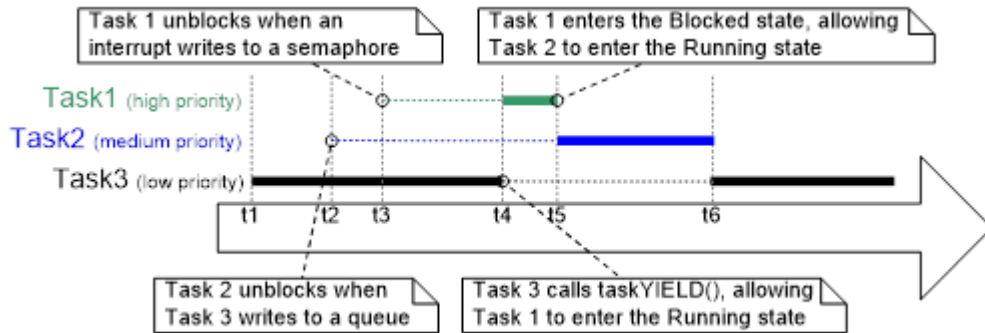
## 협력형 스케줄링

FreeRTOS 역시 협력형 스케줄링을 사용할 수 있습니다. 다음 표는 FreeRTOS 스케줄러가 협력형 스케줄링을 사용하도록 구성하는 FreeRTOSConfig.h 설정을 나열한 것입니다.

상수	값
configUSE_PREEMPTION	0
configUSE_TIME_SLICING	모든 값

협력형 스케줄러를 사용할 때는 Running 상태의 작업이 Blocked 상태로 전환되거나, 혹은 Running 상태의 작업이 taskYIELD()를 호출하여 명시적으로 넘겨줄 때만(수동으로 일정 조정을 요청할 때만) 컨텍스트 전환이 발생합니다. 작업이 절대로 선점되지 않기 때문에 시간 분할은 사용할 수 없습니다.

다음 그림은 협력형 스케줄러의 동작을 나타낸 것입니다. 수평 점선은 작업이 Ready 상태인 시간을 나타냅니다.



#### 1. Task 1

Task 1은 우선순위가 가장 높습니다. Blocked 상태로 시작되어 세마포어를 사용할 수 있을 때까지 대기합니다.

t3에서 인터럽트가 세마포어를 반환하여 Task 1이 Blocked 상태에서 Ready 상태로 전환됩니다. 인터럽트의 세마포어 반환에 대한 자세한 내용은 [인터럽트 관리 \(p. 117\)](#) 단원을 참조하십시오.

t3에서는 Task 1이 가장 높은 우선순위의 Ready 상태 작업입니다. 선점형 스케줄러를 사용했다면 Task 1이 여기에서 Running 상태의 작업이 됩니다. 하지만 협력형 스케줄러를 사용하고 있기 때문에 Task 1은 t4에서 Running 상태의 작업이 taskYIELD()를 호출할 때까지 Ready 상태를 유지합니다.

#### 2. Task 2

Task 2의 우선순위는 Task 1과 Task 3 사이입니다. 이 작업은 Blocked 상태로 시작되어 t2에서 Task 3에서 메시지가 전송될 때까지 대기합니다.

t2에서는 Task 2가 가장 높은 우선순위의 Ready 상태 작업입니다. 선점형 스케줄러를 사용했다면 Task 2가 여기에서 Running 상태의 작업이 됩니다. 하지만 협력형 스케줄러를 사용하고 있기 때문에 Task 2는 Running 상태의 작업이 Blocked 상태로 전환되거나 taskYIELD()를 호출할 때까지 Ready 상태를 유지합니다.

Running 상태의 작업이 t4에서 taskYIELD()를 호출하지만 이때까지 Task 1이 가장 높은 우선순위의 Ready 상태 작업이기 때문에 Task 2는 Task 1이 t5에서 Blocked 상태로 다시 전환될 때까지 Running 상태가 되지 못합니다.

Task 2는 t6에서 다시 Blocked 상태로 전환되어 다음 메시지가 수신될 때까지 대기하고, 이때부터 Task 3이 다시 가장 높은 우선순위의 Ready 상태 작업이 됩니다.

멀티태스킹 애플리케이션의 경우에는 애플리케이션 개발자가 작업 2개 이상이 동시에 리소스에 액세스하지 못하도록 주의해야 합니다. 동시 액세스는 리소스를 손상시킬 수 있기 때문입니다. 다음과 같이 액세스하는 리소스가 UART(직렬 포트)인 시나리오를 가정해보겠습니다. 작업 2개가 문자열을 UART에 작성합니다. Task 1은 "abcdefghijklmnp"를, 그리고 Task 2는 "123456789"를 작성합니다.

1. Task 1이 Running 상태여서 문자열을 작성하기 시작합니다. "abcdefg"를 UART에 작성하지만 추가 문자를 작성하기 전에 Running 상태를 종료합니다.
2. Task 2가 Running 상태로 전환되어 "123456789"를 UART에 작성한 후 Running 상태를 종료합니다.
3. Task 1이 다시 Running 상태로 전환되어 문자열의 나머지 문자를 UART에 작성합니다.

위 시나리오에서 실제로 UART에 작성되는 문자열은 "abcdefg123456789hijklmnop"입니다. Task 1에서 작성한 문자열이 UART에 의도한 대로 이어서 작성되지 않았습니다. 오히려 Task 2에서 UART에 작성한 문자열이 중간에 표시되어 UART가 손상되었습니다.

이때는 협력형 스케줄러를 사용하여 동시 액세스로 인한 문제를 방지할 수 있습니다. 작업 사이에 리소스를 안전하게 공유할 수 있는 방법은 본 안내서의 후반부에서 다루고 있습니다. 대기열이나 세마포어처럼 FreeRTOS에서 제공하는 리소스는 작업 사이에서 언제든지 안전하게 공유할 수 있습니다.

- 선점형 스케줄러를 사용할 때는 다른 작업과 공유하는 리소스의 상태가 일치하지 않을 때를 포함해 Running 상태의 작업을 언제든지 선점할 수 있습니다. UART 예에서 본 것처럼 일치하지 않는 상태의 리소스를 해제하면 데이터가 손상될 수 있습니다.
- 협력형 스케줄러를 사용할 때는 애플리케이션 개발자가 다른 작업으로 전환되는 시점을 제어합니다. 따라서 애플리케이션 개발자는 리소스의 상태가 일치하지 않을 경우 다른 작업으로 전환되지 않도록 주의를 기울여야 합니다.
- UART 예에서 애플리케이션 개발자는 Task 1이 전체 문자열을 UART에 작성할 때까지 Running 상태를 종료하지 않도록 할 수 있으며, 이를 통해 다른 작업의 활성화로 인해 발생할 수 있는 문자열 손상을 방지합니다.

협력형 스케줄러를 사용할 때는 시스템 응답이 비교적 줄어듭니다.

- 선점형 스케줄러를 사용할 때는 가장 높은 우선순위의 Ready 상태 작업이 되자마자 스케줄러가 해당 작업을 바로 실행합니다. 일정한 시간 내에 높은 우선순위의 이벤트에 응답해야 하는 실시간 시스템에서는 종종 선점형 스케줄러가 필수입니다.
- 협력형 스케줄러를 사용할 때는 가장 높은 우선순위의 Ready 상태 작업이 되었더라도 Running 상태의 작업이 Blocked 상태로 전환되거나, 혹은 taskYIELD()를 호출할 때까지 작업 전환은 일어나지 않습니다.

# 대기열 관리

대기열은 작업간, 작업과 인터럽트간, 인터럽트와 작업간 통신 메커니즘을 제공합니다. 이번 단원에서는 작업-작업 통신에 대해서 살펴보겠습니다. 작업-인터럽트 및 인터럽트-작업 통신에 대한 자세한 내용은 [인터럽트 관리 \(p. 117\)](#) 단원을 참조하십시오.

이번 단원에서 다루는 내용은 다음과 같습니다.

- 대기열 생성 방법
- 대기열의 데이터 관리 방법
- 데이터를 대기열로 전송하는 방법
- 데이터를 대기열에서 수신하는 방법
- 대기열에서 차단 의미
- 다수의 대기열에서 차단하는 방법
- 데이터를 대기열에 덮어쓰는 방법
- 대기열을 소거하는 방법
- 대기열에 작성하거나, 혹은 대기열에서 읽어올 때 작업 우선순위가 미치는 영향

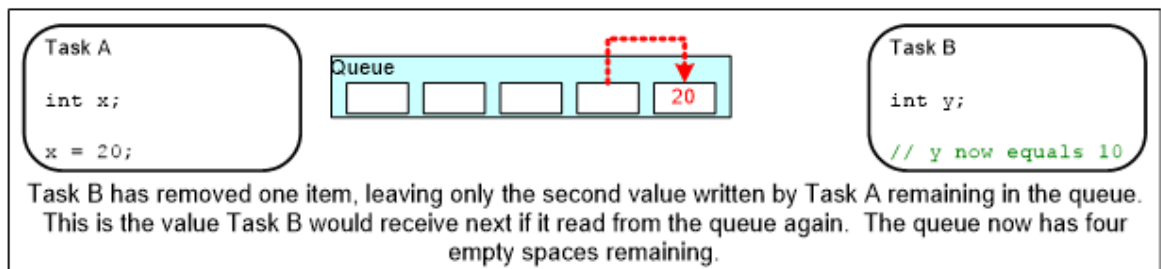
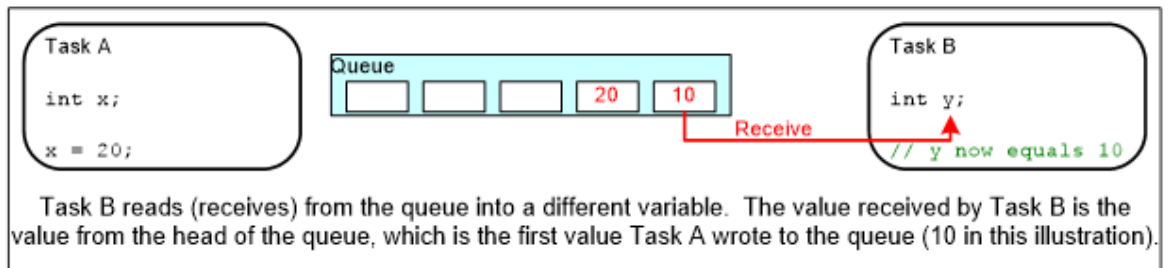
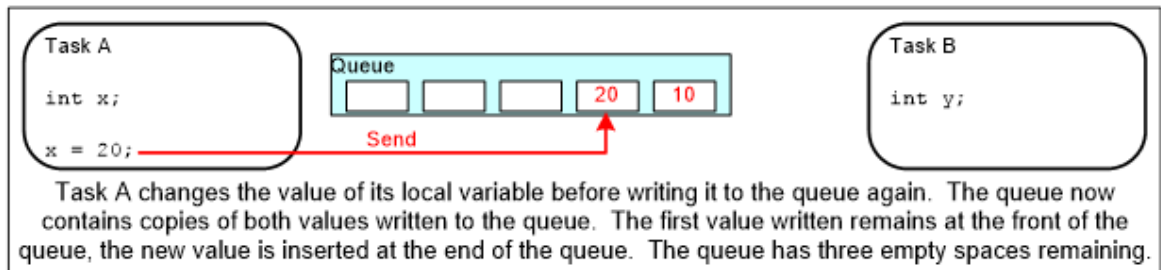
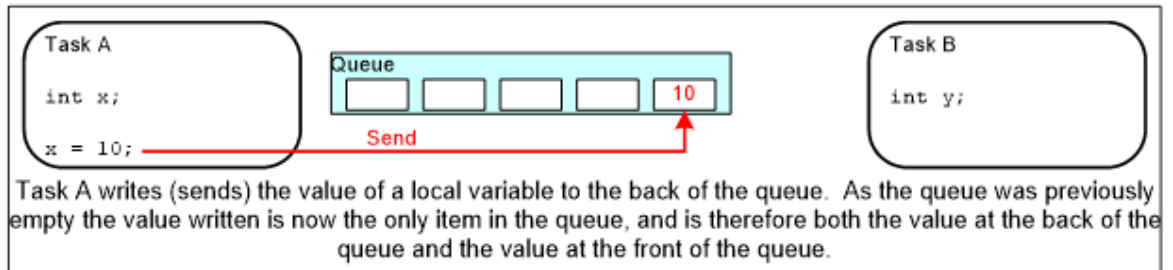
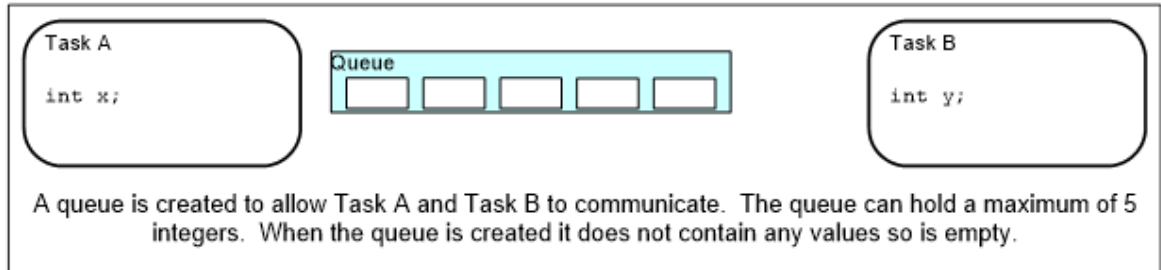
## 대기열의 특성

## 데이터 저장

대기열은 고정된 크기의 데이터 항목을 한정된 수만큼 저장할 수 있습니다. 대기열에 저장할 수 있는 항목의 최대 수를 길이라고 부릅니다. 각 데이터 항목의 길이와 크기는 대기열을 생성할 때 설정합니다.

대기열은 일반적으로 FIFO(First-In, First-Out) 버퍼로 사용되며, FIFO 버퍼에서는 데이터가 대기열 뒤(tail)에 작성되고, 대기열 앞(head)에서 삭제됩니다.

다음 그림은 데이터를 FIFO로 사용되는 대기열에 작성하고, 데이터를 FIFO로 사용되는 대기열에서 읽어오는 것을 나타낸 것입니다. 또한 데이터를 대기열 앞에 작성하거나, 혹은 이미 대기열 앞에 저장된 데이터를 덮어쓰는 것도 가능합니다.



대기열 동작을 구현하는 방법은 다음 두 가지가 있습니다.

#### 1. 복사를 통한 대기열 동작 구현

대기열에 전송할 데이터가 바이트 단위로 대기열에 복사됩니다.

## 2. 참조를 통한 대기열 동작 구현

데이터가 아닌 대기열에 전송할 데이터를 가리키는 포인터가 대기열에 저장됩니다.

FreeRTOS는 복사를 통한 대기열 동작 구현 방법을 사용합니다. 이 방법은 다음과 같은 이유로 참조를 통한 대기열 동작 구현보다 더욱 강력할 뿐만 아니라 사용도 간편한 것으로 알려져 있습니다.

- 변수를 선언한 함수가 종료된 후에는 존재하지 않지만 스택 변수를 대기열로 직접 전송할 수 있습니다.
- 먼저 데이터를 저장할 버퍼를 할당한 후 데이터를 할당된 버퍼로 복사할 필요 없이 데이터를 대기열로 전송할 수 있습니다.
- 전송 작업이 대기열로 전송된 변수 또는 버퍼를 바로 재사용할 수 있습니다.
- 전송 작업과 수신 작업이 완전하게 분리됩니다. 따라서 애플리케이션 설계자는 어떤 작업이 데이터를 소유할지, 혹은 어떤 작업이 데이터를 공개할지 걱정할 필요 없습니다.
- 복사를 통한 대기열 동작은 대기열이 참조를 통한 대기열 동작에 사용되는 것을 제한하지 않습니다. 예를 들어 대기열에 작성할 데이터의 크기가 데이터를 대기열에 복사하는 데 적합하지 않을 때는 데이터를 가리키는 포인터를 대기열에 복사할 수 있습니다.
- RTOS가 데이터를 저장할 때 사용할 메모리를 할당하는 데 전적인 책임을 집니다.
- 메모리 보호 시스템에서는 작업이 액세스할 수 있는 RAM이 제한적입니다. 이때는 전송 작업과 수신 작업이 데이터가 저장된 RAM에 액세스할 수 있는 경우에 한해 참조를 통한 대기열 동작을 사용할 수 있습니다. 하지만 복사를 통한 대기열 동작은 이러한 제한이 없습니다. 커널이 항상 최대 권한으로 실행되기 때문에 대기열을 사용해 메모리 보호 경계를 넘어서 데이터를 전달할 수 있습니다.

## 다수의 작업에서 액세스

대기열은 대기열의 존재에 대해 알고 있는 작업 또는 인터럽트 서비스 레지스터(ISR)가 액세스할 수 있는 객체입니다. 동일한 대기열에 데이터를 작성하거나, 혹은 동일한 대기열에서 데이터를 읽어올 수 있는 작업의 수에는 제한이 없습니다. 대기열에 데이터를 작성하는 작업이 다수인 경우는 매우 흔한 일이지만 데이터를 읽어오는 작업이 다수인 경우는 그렇게 많지 않습니다.

## 대기열 읽기에서 차단

작업이 대기열에서 데이터를 읽어오려고 할 때 옵션으로 차단 시간을 지정할 수 있습니다. 차단 시간이란 대기열이 비어있을 경우 작업이 대기열에서 데이터를 사용할 수 있을 때까지 Blocked 상태로 대기하는 시간을 말합니다. 이후 다른 작업 또는 인터럽트가 데이터를 대기열로 보내면 대기열에서 데이터를 사용할 수 있을 때까지 Blocked 상태로 대기하던 작업이 Ready 상태로 자동 전환됩니다. 또한 데이터를 사용하기 전에 지정된 차단 시간이 지나도 작업이 Blocked 상태에서 Ready 상태로 자동 전환됩니다.

대기열에는 데이터를 읽어오는 작업이 다수일 수 있기 때문에 차단된 상태로 데이터를 기다리는 작업이 대기열 1개마다 1개 이상일 수 있습니다. 이러한 경우에 데이터를 사용할 수 있게 되었을 때 작업은 1개만 차단 해제됩니다. 이때 데이터를 기다리는 작업 중에서 우선순위가 가장 높은 작업이 항상 차단 해제됩니다. 차단 해제되는 작업의 우선순위가 동일한 경우에는 가장 오랜 시간 데이터를 기다린 작업이 차단 해제됩니다.

## 대기열 쓰기에서 차단

작업이 대기열에서 데이터를 읽어올 때처럼 대기열에 데이터를 작성할 때도 옵션으로 차단 시간을 지정할 수 있습니다. 이때 차단 시간이란 대기열이 가득 차있을 경우 작업이 대기열에서 사용할 수 있는 공간이 나올 때까지 Blocked 상태로 대기해야 하는 최대 시간을 말합니다.

대기열에는 데이터를 작성하는 작업이 다수일 수 있기 때문에 차단된 상태로 전송 작업을 마칠 때까지 대기하는 작업이 가득 찬 대기열 1개마다 1개 이상일 수 있습니다. 이러한 경우에 대기열에 사용 가능한 공간이



생겼을 때 작업은 1개만 차단 해제됩니다. 이때 사용 가능한 공간이 생길 때까지 대기하는 작업 중에서 우선 순위가 가장 높은 작업이 항상 차단 해제됩니다. 차단 해제되는 작업의 우선순위가 동일한 경우에는 가장 오랜 시간 공간을 기다린 작업이 차단 해제됩니다.

## 다수의 대기열에서 차단

대기열은 여러 집합으로 그룹화가 가능하기 때문에 작업이 집합에 속한 어떤 대기열에서든 Blocked 상태로 전환되어 데이터를 사용할 수 있을 때까지 대기할 수 있습니다. 대기열 집합에 대한 자세한 내용은 [다수의 대기열에서 수신 \(p. 84\)](#) 단원을 참조하십시오.

## 대기열 사용

### xQueueCreate() API 함수

대기열을 사용하려면 먼저 명시적으로 생성해야 합니다.

대기열은 QueueHandle\_t 형식의 변수인 핸들을 통해 참조됩니다. xQueueCreate() API 함수는 대기열을 생성한 후 생성한 대기열을 참조하는 QueueHandle\_t를 반환합니다.

FreeRTOS V9.0.0에도 컴파일 과정에서 대기열을 정적으로 생성하는 데 필요한 메모리를 할당하는 xQueueCreateStatic() 함수가 포함되어 있습니다. FreeRTOS는 대기열을 생성할 때 FreeRTOS 힙에서 RAM을 할당합니다. 할당된 램은 대기열 데이터 구조와 대기열에 포함된 데이터 항목을 저장하는 데 사용됩니다. 대기열을 생성하는 데 필요한 힙 RAM이 부족한 경우에는 xQueueCreate()가 NULL을 반환합니다.

xQueueCreate() API 함수 프로토타입은 다음과 같습니다.

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize );
```

다음 표는 xQueueCreate() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름	설명
uxQueueLength	생성된 대기열이 어느 때든 한 번에 저장할 수 있는 최대 항목 수입니다.
uxItemSize	대기열에 저장할 수 있는 각 데이터 항목의 크기(바이트)입니다.
반환 값	<p>NULL을 반환하는 경우에는 대기열을 생성할 수 없습니다. FreeRTOS에서 대기열 데이터 구조와 저장 영역을 할당하는 데 필요한 힙 메모리가 부족하기 때문입니다.</p> <p>NULL이 아닌 값을 반환하는 경우에는 대기열이 성공적으로 생성된 것입니다. 반환 값은 생성된 대기열에 대한 핸들로 저장되어야 합니다.</p>

대기열이 생성된 후에는 xQueueReset() API 함수를 사용해 대기열을 최초 비어있는 상태로 복원할 수 있습니다.

## xQueueSendToBack() 및 xQueueSendToFront() API 함수

xQueueSendToBack()은 데이터를 대기열 뒤(tail)로 보내는 데 사용되는 반면, xQueueSendToFront()는 데이터를 대기열 앞(head)으로 보내는 데 사용됩니다.

xQueueSend()는 xQueueSendToBack()과 정확히 동일합니다.

참고: xQueueSendToFront() 또는 xQueueSendToBack() 함수는 인터럽트 서비스 루틴에서 호출하지 마십시오. 대신에 인터럽트-세이프 버전인 xQueueSendToFrontFromISR()과 xQueueSendToBackFromISR()을 사용하십시오.

xQueueSendToFront() API 함수 프로토타입은 다음과 같습니다.

```
BaseType_t xQueueSendToFront( QueueHandle_t xQueue,  
const void * pvItemToQueue,  
TickType_t xTicksToWait );
```

xQueueSendToBack() API 함수 프로토타입은 다음과 같습니다.

```
BaseType_t xQueueSendToBack( QueueHandle_t xQueue,  
const void * pvItemToQueue,  
TickType_t xTicksToWait );
```

다음 표는 xQueueSendToFront() 및 xQueueSendToBack() 함수 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
xQueue	데이터가 전송되는(작성되는) 대기열의 핸들입니다. 대기열 핸들은 대기열 생성을 위해 xQueueCreate()를 호출할 때 반환됩니다.
pvItemToQueue	대기열로 복사할 데이터를 가리키는 포인터입니다.  대기열에 저장할 수 있는 각 항목의 크기는 대기열을 생성할 때 설정됩니다. 따라서 여기에서 설정된 크기의 바이트가 pvItemToQueue에서 대기열 저장 영역으로 복사됩니다.
xTicksToWait	대기열이 가득 차있을 경우 작업이 대기열에 사용할 수 있는 공간이 나올 때까지 Blocked 상태로 대기해야 하는 최대 시간입니다.  xTicksToWait가 0이고 대기열이 이마 가득 찬 상태라면 xQueueSendToFront()와 xQueueSendToBack()이 모두 즉시 값을 반환합니다.  차단 시간은 틱 주기로 지정되기 때문에 틱 주파수에 따라 절대 시간이 달라집니다. 밀리초 단위의 시간은 pdMS_TO_TICKS() 매크로를 사용해 틱 단위의 시간으로 변환할 수 있습니다.

	xTicksToWait를 portMAX_DELAY로 설정하면 작업이 제한 시간 없이 무기한 대기하게 됩니다. 단, INCLUDE_vTaskSuspend가 FreeRTOSConfig.h에서 1로 설정되어 있어야 합니다.
반환 값	<p>가능한 반환 값은 다음 두 가지입니다.</p> <ol style="list-style-type: none"> <li>1. pdPASS <p>데이터가 대기열에 성공적으로 전송된 경우에 한해 반환됩니다.</p> <p>차단 시간이 지정된 경우에는(xTicksToWait가 0이 아닌 경우) 함수 반환 이전에 대기열에서 사용할 수 있는 공간이 나올 때까지 호출 작업이 Blocked 상태로 전환되어 대기하지만 차단 시간이 지나기 전에 데이터가 대기열에 성공적으로 작성되었을 가능성이 있습니다.</p> </li> <li>2. errQUEUE_FULL <p>대기열이 이미 가득 찬 상태여서 데이터를 대기열에 작성할 수 없을 때 반환됩니다.</p> <p>차단 시간이 지정된 경우에는(xTicksToWait가 0이 아닌 경우) 호출 작업이 Blocked 상태로 전환되어 다른 작업 또는 인터럽트가 대기열에 공간을 만들 때까지 대기하지만 이전에 차단 시간이 먼저 지나버린 것입니다.</p> </li> </ol>

## xQueueReceive() API 함수

xQueueReceive() 함수는 대기열에서 데이터 항목을 수신할 때(읽어올 때) 사용됩니다. 수신된 항목은 대기열에서 삭제됩니다.

참고: xQueueReceive()를 인터럽트 서비스 루틴에서 호출하지 마십시오. 대신에 인터럽트 세이프 버전인 xQueueReceiveFromISR() API 함수를 사용하십시오.

xQueueReceive() API 함수 프로토타입은 다음과 같습니다.

```
BaseType_t xQueueReceive( QueueHandle_t xQueue,
void * const pvBuffer,
TickType_t xTicksToWait );
```

다음 표는 xQueueReceive() 함수 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
xQueue	데이터를 수신하는(읽어오는) 대기열의 핸들입니다. 대기열 핸들은 대기열 생성을 위해 xQueueCreate()를 호출할 때 반환됩니다.
pvBuffer	수신된 데이터를 복사할 메모리를 가리키는 포인터입니다.

	대기열에 저장되는 각 데이터 항목의 크기는 대기열을 생성할 때 설정됩니다. pvBuffer가 가리키는 메모리는 이러한 크기의 바이트를 저장할 만큼 충분히 커야 합니다.
xTicksToWait	<p>대기열이 비어있을 경우 작업이 대기열에서 데이터를 사용할 수 있을 때까지 Blocked 상태로 대기해야 하는 최대 시간입니다.</p> <p>xTicksToWait가 0이면 대기열이 비어있더라도 xQueueReceive()가 즉시 값을 반환합니다.</p> <p>차단 시간은 틱 주기로 지정되기 때문에 틱 주파수에 따라 절대 시간이 달라집니다. 밀리초 단위의 시간은 pdMS_TO_TICKS() 매크로를 사용해 틱으로 변환할 수 있습니다.</p> <p>xTicksToWait를 portMAX_DELAY로 설정하면 작업이 제한 시간 없이 무기한 대기하게 됩니다. 단, INCLUDE_vTaskSuspend가 FreeRTOSConfig.h에서 1로 설정되어 있어야 합니다.</p>
반환 값	<p>가능한 반환 값은 다음 두 가지입니다.</p> <ol style="list-style-type: none"> <li>pdPASS <p>데이터를 대기열에서 성공적으로 읽어온 경우에 한해 반환됩니다.</p> <p>차단 시간이 지정된 경우에는(xTicksToWait가 0이 아닌 경우) 대기열에서 데이터를 사용할 수 있을 때까지 호출 작업이 Blocked 상태로 전환되어 대기하지만 차단 시간이 지나기 전에 데이터를 대기열에서 성공적으로 읽어왔을 가능성이 있습니다.</p> </li> <li>errQUEUE_EMPTY <p>대기열이 비어있는 상태여서 데이터를 대기열에서 읽어올 수 없을 때 반환됩니다.</p> <p>차단 시간이 지정된 경우에는(xTicksToWait가 0이 아닌 경우) 호출 작업이 Blocked 상태로 전환되어 다른 작업 또는 인터럽트가 데이터를 대기열에 전송할 때까지 대기하지만 이전에 차단 시간이 먼저 지나버린 것입니다.</p> </li> </ol>

## uxQueueMessagesWaiting() API 함수

uxQueueMessagesWaiting() 함수는 현재 대기열에 저장된 데이터 항목 수를 쿼리하는 데 사용됩니다.

참고: uxQueueMessagesWaiting()를 인터럽트 서비스 루틴에서 호출하지 마십시오. 대신에 인터럽트 세이프 버전인 uxQueueMessagesWaitingFromISR()을 사용하십시오.

uxQueueMessagesWaiting() API 함수 프로토타입은 다음과 같습니다.

```
UBaseType_t uxQueueMessagesWaiting( QueueHandle_t xQueue );
```

다음 표는 uxQueueMessagesWaiting() 함수 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
xQueue	쿼리할 대기열의 핸들입니다. 대기열 핸들은 대기열 생성을 위해 xQueueCreate()를 호출할 때 반환됩니다.
반환 값	쿼리할 대기열에 현재 저장된 항목 수를 반환합니다. 0이 반환되면 대기열이 비어있는 것을 의미합니다.

## 대기열에서 수신할 때 차단(예제 10)

이번 예제에서는 대기열을 생성하는 작업, 데이터를 다수의 작업에서 대기열로 전송하는 작업, 데이터를 대기열에서 수신하는 작업에 대해서 설명합니다. 대기열은 int32\_t 형식의 데이터 항목을 저장할 목적으로 생성됩니다. 대기열로 데이터를 전송하는 작업은 차단 시간을 지정하지 않지만 대기열에서 데이터를 수신하는 작업은 차단 시간을 지정합니다.

대기열로 전송하는 작업의 우선순위는 대기열에서 수신하는 작업의 우선순위보다 낮습니다. 이 말은 대기열에 항목 2개 이상이 포함되어서는 안 된다는 것을 의미합니다. 데이터가 대기열에 전송되자마자 수신 작업이 차단 해제되면서 전송 작업보다 먼저 실행되고 이후 전송된 데이터가 삭제되어 대기열이 다시 비워지기 때문입니다.

다음 코드는 대기열에 작성하는 작업의 구현체를 나타낸 것입니다. 이 작업은 2개의 인스턴스가 생성되었습니다. 하나는 대기열에 값 100을 계속해서 작성하는 인스턴스이고, 다른 하나는 동일한 대기열에 값 200을 계속해서 작성하는 인스턴스입니다. 작업 파라미터는 각 작업 인스턴스에 이 값을 전달하는 데 사용됩니다.

```
static void vSenderTask( void *pvParameters )
{
    int32_t lValueToSend;

    BaseType_t xStatus;

    /* Two instances of this task are created so the value that is sent to the queue
    is passed in through the task parameter. This way, each instance can use a different
    value. The queue was created to hold values of type int32_t, so cast the parameter to the
    required type. */

    lValueToSend = ( int32_t ) pvParameters;

    /* As per most tasks, this task is implemented within an infinite loop. */

    for( ;; )
    {
        /* Send the value to the queue. The first parameter is the queue to which data is
        being sent. The queue was created before the scheduler was started, so before this task
        started to execute. The second parameter is the address of the data to be sent, in this
        case the address of lValueToSend. The third parameter is the Block time, the time the task
        should be kept in the Blocked state to wait for space to become available on the queue
        should the queue already be full. In this case a block time is not specified because the
        queue should never contain more than one item, and therefore never be full. */

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

```
        if( xStatus != pdPASS )
        {
            /* The send operation could not complete because the queue was full. This must
            be an error because the queue should never contain more than one item! */

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

다음 코드는 데이터를 대기열에서 수신하는 작업의 구현체를 나타낸 것입니다. 수신 작업이 차단 시간을 100 밀리초로 지정하기 때문에 Blocked 상태로 전환되어 데이터를 사용할 수 있을 때까지 대기합니다. 이후 데이터를 대기열에서 사용할 수 있을 때 또는 데이터를 사용할 수 없지만 100밀리초가 지났을 때 Blocked 상태를 종료합니다. 이번 예제에서는 제한 시간인 100밀리초가 절대로 지날 수 없습니다. 두 작업이 계속해서 데이터를 대기열에 작성하기 때문입니다.

```
static void vReceiverTask( void *pvParameters )
{
    /* Declare the variable that will hold the values received from the queue. */
    int32_t lReceivedValue;

    BaseType_t xStatus;

    const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );

    /* This task is also defined within an infinite loop. */
    for( ;; )
    {
        /* This call should always find the queue empty because this task will immediately
        remove any data that is written to the queue. */

        if( uxQueueMessagesWaiting( xQueue ) != 0 )
        {
            vPrintString( "Queue should have been empty!\r\n" );
        }

        /* Receive data from the queue. The first parameter is the queue from which data is
        to be received. The queue is created before the scheduler is started, and therefore before
        this task runs for the first time. The second parameter is the buffer into which the
        received data will be placed. In this case the buffer is simply the address of a variable
        that has the required size to hold the received data. The last parameter is the block
        time, the maximum amount of time that the task will remain in the Blocked state to wait
        for data to be available should the queue already be empty. */

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

        if( xStatus == pdPASS )
        {

```

```
        /* Data was successfully received from the queue, print out the received value.
        */

        vPrintStringAndNumber( "Received = ", lReceivedValue );

    }

    else

    {

        /* Data was not received from the queue even after waiting for 100ms.This must
        be an error because the sending tasks are free running and will be continuously writing to
        the queue. */

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

    }

}

}
```

다음 코드에는 main() 함수의 정의가 포함되어 있습니다. 이번에는 스케줄러를 시작하기 전에 대기열과 3개의 작업을 생성합니다. 작업 우선순위는 대기열에 한 번에 항목 2개 이상이 포함되지 않도록 설정되지만 대기열은 최대 5개의 int32\_t 값을 저장할 수 있도록 생성됩니다.

```
/* Declare a variable of type QueueHandle_t. This is used to store the handle to the queue
that is accessed by all three tasks. */

QueueHandle_t xQueue;

int main( void )

{

    /* The queue is created to hold a maximum of 5 values, each of which is large enough to
    hold a variable of type int32_t. */

    xQueue = xQueueCreate( 5, sizeof( int32_t ) );

    if( xQueue != NULL )

    {

        /* Create two instances of the task that will send to the queue. The task parameter
        is used to pass the value that the task will write to the queue, so one task will
        continuously write 100 to the queue while the other task will continuously write 200 to
        the queue. Both tasks are created at priority 1. */

        xTaskCreate( vSenderTask, "Sender1", 1000, ( void * ) 100, 1, NULL );

        xTaskCreate( vSenderTask, "Sender2", 1000, ( void * ) 200, 1, NULL );

        /* Create the task that will read from the queue. The task is created with priority
        2, so above the priority of the sender tasks. */

        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );

        /* Start the scheduler so the created tasks start executing. */

        vTaskStartScheduler();

    }

}
```

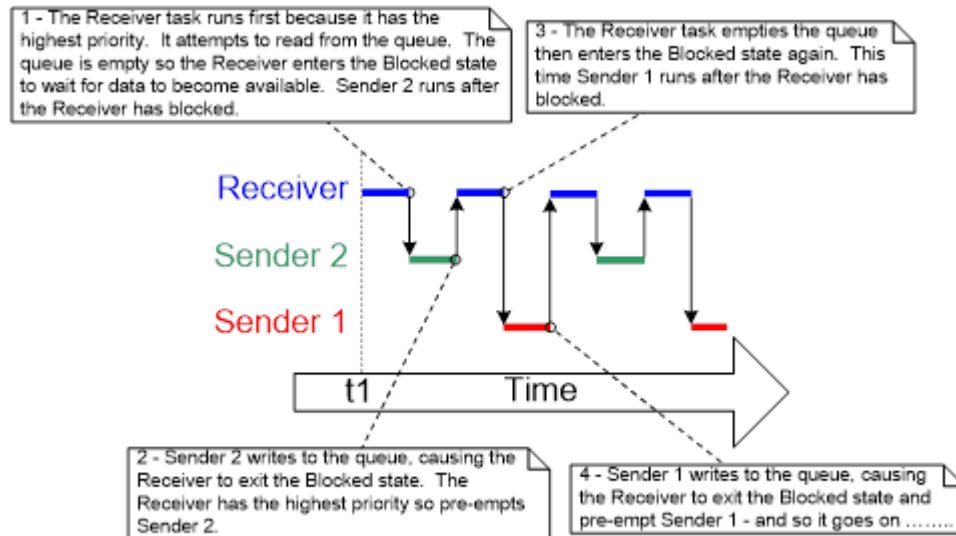
```
else
{
    /* The queue could not be created. */
}

/* If all is well then main() will never reach here as the scheduler will now be
running the tasks. If main() does reach here then it is likely that there was insufficient
FreeRTOS heap memory available for the idle task to be created. For more information, see
Heap Memory Management. */

for( ;; );
}
```

대기열에 전송하는 작업은 모두 동일한 우선순위를 갖습니다. 이렇게 하면 전송 작업 2개가 차례로 데이터를 대기열에 전송합니다.

다음 그림은 실행 시퀀스를 나타낸 것입니다.

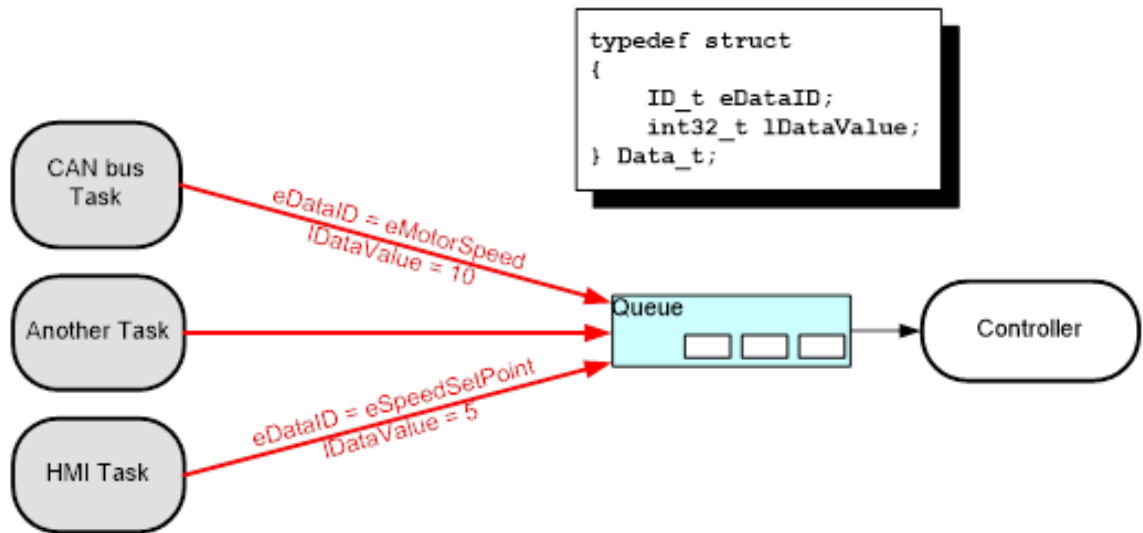


## 다수의 소스에서 데이터를 수신

FreeRTOS 설계에서 작업이 다수의 소스에서 데이터를 수신하는 것은 흔한 일입니다. 하지만 데이터 처리 방식을 결정하려면 수신 작업이 데이터의 수신 위치를 알아야만 합니다. 이때는 단일 대기열을 사용해 데이터 구조를 구조 필드에 포함된 데이터 값 및 소스와 함께 전송하면 쉽게 해결할 수 있습니다.

다음 그림은 데이터 구조가 대기열을 통해 전송되는 시나리오 예를 나타낸 것입니다.





- Data\_t 형식의 구조를 저장할 대기열이 생성됩니다. 구조 멤버에서는 데이터 값과 열거 형식을 메시지 하나로 대기열에 전송할 수 있습니다. 열거 형식은 데이터의 의미를 나타내는 데 사용됩니다.
- 중앙의 Controller 작업은 기본 시스템 함수를 실행하는 데 사용됩니다. 이 작업은 대기열을 통해 주고받는 시스템 상태에 대한 입력 값과 변경 사항에 응답해야 합니다.
- CAN 버스 작업은 CAN 버스 인터페이스 기능을 캡슐화하는 데 사용됩니다. CAN 버스 작업이 메시지를 수신하여 디코딩한 후 디코딩된 메시지를 Data\_t 구조로 Controller 작업에게 전송합니다. 전송된 구조의 eDataID 멤버는 Controller 작업에게 어떤 데이터(이번 경우에는 모터 속도 값)인지 알리는 데 사용됩니다. 전송된 구조의 lDataValue 멤버는 Controller 작업에게 모터 속도 값을 알리는 데 사용됩니다.
- HMI(Human Machine Interface) 작업은 모든 HMI 기능을 캡슐화하는 데 사용됩니다. 머신 오퍼레이터는 HMI 작업 내부에서 감지하여 해석해야 하는 명령과 쿼리 값을 다양한 방법으로 입력할 수 있습니다. 새로운 명령이 입력되면 HMI 작업이 명령을 Controller 작업에게 Data\_t 구조로 전송합니다. 전송된 구조의 eDataID 멤버는 Controller 작업에게 어떤 데이터(이번 경우에는 새로운 설정 값)인지 알리는 데 사용됩니다. 전송된 구조의 lDataValue 멤버는 Controller 작업에게 설정 값을 알리는 데 사용됩니다.

## 대기열에 전송할 때 차단과 대기열을 통해 구조를 전송(예제 11)

이번 예제는 앞의 예제와 비슷하지만 작업 우선순위가 역전됩니다. 따라서 수신 작업의 우선순위가 전송 작업의 우선순위보다 낮습니다. 또한 대기열은 정수가 아닌 구조를 전달하는 데 사용됩니다.

다음 코드는 대기열을 통해 전달해야 할 구조의 정의와 두 변수의 선언을 나타낸 것입니다.

```

/* Define an enumerated type used to identify the source of the data.*/

typedef enum
{
    eSender1,
    eSender2
} DataSource_t;

```

```
/* Define the structure type that will be passed on the queue. */
typedef struct
{
    uint8_t ucValue;

    DataSource_t eDataSource;
} Data_t;

/* Declare two variables of type Data_t that will be passed on the queue. */
static const Data_t xStructsToSend[ 2 ] =
{
    { 100, eSender1 }, /* Used by Sender1. */
    { 200, eSender2 } /* Used by Sender2. */
};
```

이전 예제에서는 수신 작업의 우선순위가 가장 높아서 대기열에 저장되는 항목이 1개를 넘지 않았습니다. 이는 데이터가 대기열에 전송되자마자 수신 작업이 전송 작업보다 앞서 실행되기 때문입니다. 하지만 다음 예제에서는 전송 작업의 우선순위가 가장 높아서 대체로 대기열이 가득 차게 됩니다. 이는 수신 작업이 대기열에서 항목을 삭제하자마자 전송 작업 중 하나가 앞서 실행되면서 대기열을 즉시 채우기 때문입니다. 그러면 전송 작업이 다시 Blocked 상태로 전환되어 대기열에서 사용 가능한 공간이 생길 때까지 대기합니다.

다음 코드는 전송 작업의 구현체를 나타낸 것입니다. 전송 작업이 차단 시간을 100밀리초로 지정하기 때문에 대기열이 가득 찰 때마다 Blocked 상태로 전환되어 사용 가능한 공간이 생길 때까지 대기합니다. 이후 대기열에서 사용 가능한 공간이 생길 때 또는 사용 가능한 공간이 없지만 100밀리초가 지났을 때 Blocked 상태를 종료합니다. 이번 예제에서는 제한 시간인 100밀리초가 절대로 지날 수 없습니다. 수신 작업이 대기열에서 항목을 삭제하여 계속해서 공간을 만들기 때문입니다.

```
static void vSenderTask( void *pvParameters )
{
    BaseType_t xStatus;

    const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );

    /* As per most tasks, this task is implemented within an infinite loop.*/
    for( ;; )
    {
        /* Send to the queue. The second parameter is the address of the structure being sent. The address is passed in as the task parameter so pvParameters is used directly. The third parameter is the Block time, the time the task should be kept in the Blocked state to wait for space to become available on the queue if the queue is already full. A block time is specified because the sending tasks have a higher priority than the receiving task so the queue is expected to become full. The receiving task will remove items from the queue when both sending tasks are in the Blocked state. */

        xStatus = xQueueSendToBack( xQueue, pvParameters, xTicksToWait );

        if( xStatus != pdPASS )
        {

```

```
        /* The send operation could not complete, even after waiting for 100 ms. This
        must be an error because the receiving task should make space in the queue as soon as both
        sending tasks are in the Blocked state. */

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

    }

}

}
```

수신 작업은 우선순위가 가장 낮기 때문에 두 전송 작업이 Blocked 상태일 때만 실행됩니다. 전송 작업은 대기열이 가득 찼을 때만 Blocked 상태로 전환되기 때문에 수신 작업은 대기열이 이미 가득 찼을 때만 실행됩니다. 따라서 차단 시간을 지정하지 않더라도 항상 데이터를 수신할 수 있습니다.

다음 코드는 수신 작업의 구현체를 나타낸 것입니다.

```
static void vReceiverTask( void *pvParameters )
{
    /* Declare the structure that will hold the values received from the queue. */
    Data_t xReceivedStructure;

    BaseType_t xStatus;

    /* This task is also defined within an infinite loop. */
    for( ;; )
    {
        /* Because it has the lowest priority, this task will only run when the sending
        tasks are in the Blocked state. The sending tasks will only enter the Blocked state when
        the queue is full so this task always expects the number of items in the queue to be equal
        to the queue length, which is 3 in this case. */

        if( uxQueueMessagesWaiting( xQueue ) != 3 )
        {
            vPrintString( "Queue should have been full!\r\n" );
        }

        /* Receive from the queue. The second parameter is the buffer into which the
        received data will be placed. In this case, the buffer is simply the address of a variable
        that has the required size to hold the received structure. The last parameter is the block
        time, the maximum amount of time that the task will remain in the Blocked state to wait
        for data to be available if the queue is already empty. In this case, a block time is not
        required because this task will only run when the queue is full. */

        xStatus = xQueueReceive( xQueue, &xReceivedStructure, 0 );

        if( xStatus == pdPASS )
        {
            /* Data was successfully received from the queue, print out the received value
            and the source of the value. */
```

```
        if( xReceivedStructure.eDataSource == eSender1 )
        {
            vPrintStringAndNumber( "From Sender 1 = ", xReceivedStructure.ucValue );
        }
        else
        {
            vPrintStringAndNumber( "From Sender 2 = ", xReceivedStructure.ucValue );
        }
    }
    else
    {
        /* Nothing was received from the queue. This must be an error because this task
        should only run when the queue is full. */
        vPrintString( "Could not receive from the queue.\r\n" );
    }
}
}
```

main() 함수는 이전 예제와 비교하여 약간만 바뀌었습니다. 대기열은 Data\_t 구조 3개를 저장할 목적으로 생성되며, 전송 및 수신 작업의 우선순위가 역전되었습니다. main() 함수의 구현체는 다음과 같습니다.

```
int main( void )
{
    /* The queue is created to hold a maximum of 3 structures of type Data_t. */
    xQueue = xQueueCreate( 3, sizeof( Data_t ) );

    if( xQueue != NULL )
    {
        /* Create two instances of the task that will write to the queue. The parameter
        is used to pass the structure that the task will write to the queue, so one task will
        continuously send xStructsToSend[ 0 ] to the queue while the other task will continuously
        send xStructsToSend[ 1 ]. Both tasks are created at priority 2, which is above the
        priority of the receiver. */
        xTaskCreate( vSenderTask, "Sender1", 1000, &(amp; xStructsToSend[ 0 ] ), 2, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, &(amp; xStructsToSend[ 1 ] ), 2, NULL );

        /* Create the task that will read from the queue. The task is created with priority
        1, so below the priority of the sender tasks. */
        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 1, NULL );

        /* Start the scheduler so the created tasks start executing. */
    }
}
```

```

vTaskStartScheduler();

}

else

{

    /* The queue could not be created. */

}

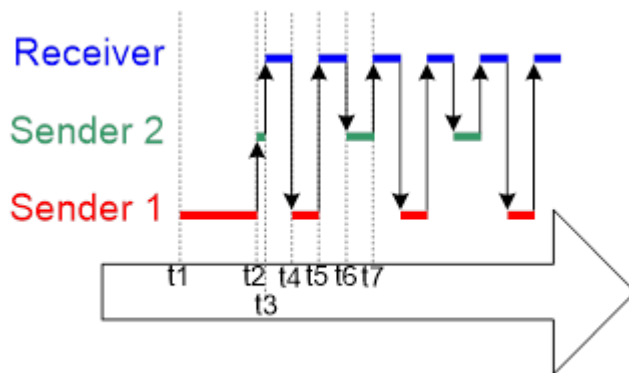
/* If all is well then main() will never reach here as the scheduler will now be
running the tasks. If main() does reach here then it is likely that there was insufficient
heap memory available for the idle task to be created. Chapter 2 provides more information
on heap memory management. */

for( ;; );

}

```

다음 그림은 전송 작업의 우선순위가 수신 작업의 우선순위보다 높은 데서 비롯되는 실행 시퀀스를 나타낸 것입니다.



다음 표는 첫 번째 메시지 4개가 동일한 작업에서 시작되는 이유를 설명한 것입니다.

시간	설명
t1	Sender 1 작업이 실행되어 데이터 항목 3개를 대기열로 전송합니다.
t2	대기열이 가득 차서 Sender 1이 Blocked 상태로 전환되어 다음 전송 작업이 완료될 때까지 대기합니다. 이제 Sender 2 작업이 실행 가능한 작업 중에서 가장 높은 우선순위의 작업이 되어 Running 상태로 전환됩니다.
t3	Sender 2 작업은 대기열이 이미 가득 차있는 것을 알고 Blocked 상태로 전환되어 첫 번째 전송 작업이 완료될 때까지 대기합니다. 이제 Receiver 작업이 실행 가능한 작업 중에서 가장 높은 우선순위의 작업이 되어 Running 상태로 전환됩니다.
t4	우선순위가 수신 작업보다 높은 작업 2개가 대기열에서 사용 가능한 공간이 나올 때까지 대기하며, 이

	후 대기열에서 항목 1개가 삭제되면 Receiver 작업보다 앞서 바로 실행됩니다. Sender 1 및 Sender 2 작업은 우선순위가 동일하기 때문에 스케줄러가 가장 오래 대기한 작업을 Running 상태로 전환할 작업으로 선택합니다(이번 경우에는 Sender 1 작업).
t5	<p>Sender 1 작업이 다른 데이터 항목을 대기열로 전송합니다. 대기열에는 한 공간만 있기 때문에 Sender 1 작업이 Blocked 상태로 전환되어 다음 전송 작업이 완료될 때 까지 대기합니다. Receiver 작업이 실행 가능한 작업 중에서 다시 가장 높은 우선순위의 작업이 되어 Running 상태로 전환됩니다.</p> <p>지금까지 Sender 1 작업이 대기열에 항목 4개를 전송하였지만 Sender 2 작업은 첫 번째 항목을 대기열에 전송할 때까지 여전히 대기하고 있습니다.</p>
t6	우선순위가 수신 작업보다 높은 작업 2개가 대기열에서 사용 가능한 공간이 나올 때까지 대기하며, 이후 대기열에서 항목 1개가 삭제되면 Receiver 작업보다 앞서 바로 실행됩니다. 이번에는 Sender 2가 Sender 1보다 더욱 오래 대기했기 때문에 Sender 2가 Running 상태로 전환됩니다.
t7	Sender 2 작업이 데이터 항목을 대기열로 전송합니다. 대기열에는 한 공간만 있기 때문에 Sender 2가 Blocked 상태로 전환되어 다음 전송 작업이 완료될 때 까지 대기합니다. Sender 1 작업과 Sender 2 작업이 모두 대기열에서 사용 가능한 공간이 나올 때까지 대기하면서 Receiver 작업이 유일하게 Running 상태로 전환될 수 있는 작업이 됩니다.

## 대용량 또는 가변 크기의 데이터 작업

### 포인터를 사용한 대기열 동작

대기열에 저장되는 데이터의 크기가 대용량인 경우에는 데이터를 대기열로/대기열에서 바이트 단위로 복사하는 것보다는 데이터를 가리키는 포인터를 전송하는 것이 좋습니다. 처리 시간이나 대기열을 생성하는 데 필요한 RAM 크기를 고려했을 때 포인터를 전송하는 것이 더욱 효율적입니다. 하지만 포인터를 사용한 대기열 동작에서는 다음과 같은 사항을 주의해야 합니다.

- 가리키는 RAM의 소유자가 명확하게 정의되어야 합니다.

포인터를 사용해 작업 사이에 메모리를 공유할 때는 두 작업이 메모리 내용을 동시에 수정하거나, 혹은 다른 작업으로 메모리 내용이 잘못되거나 달라지지 않도록 해야 합니다. 메모리를 가리키는 포인터가 대기열에 전송될 때까지는 전송 작업에게만 메모리 액세스를 허용하고, 수신 작업에게는 포인터가 대기열에서 수신된 후에 메모리 액세스를 허용하는 것이 바람직합니다.

- 가리키는 RAM이 계속해서 유효해야 합니다.

가리키는 메모리를 동적으로 할당하였거나, 혹은 사전 할당된 버퍼 풀에서 가져온 경우에는 정확하게 작업 1개만 메모리를 해제해야 합니다. 메모리가 해제된 후에 메모리에 액세스하는 작업이 있어서는 안 됩니다.

포인터를 사용해 작업 스택에 할당된 데이터에 액세스해서는 안 됩니다. 스택 프레임이 변경된 후에는 데이터가 유효하지 않기 때문입니다.

아래 코드 예제들은 대기열을 사용해 버퍼를 가리키는 포인터를 작업에서 다른 작업으로 전송하는 방법을 나타낸 것입니다.

다음 코드에서는 포인터를 최대 5개까지 저장할 수 있는 대기열을 생성합니다.

```
/* Declare a variable of type QueueHandle_t to hold the handle of the queue being created.
 */

QueueHandle_t xPointerQueue;

/* Create a queue that can hold a maximum of 5 pointers (in this case, character pointers).
 */

xPointerQueue = xQueueCreate( 5, sizeof( char * ) );
```

다음 코드에서는 버퍼를 할당하고, 문자열을 버퍼에 작성하고, 버퍼를 가리키는 포인터를 대기열에 전송합니다.

```
/* A task that obtains a buffer, writes a string to the buffer, and then sends the address
of the buffer to the queue created in the previous listing. */

void vStringSendingTask( void *pvParameters )
{
    char *pcStringToSend;

    const size_t xMaxStringLength = 50;

    BaseType_t xStringNumber = 0;

    for( ;; )
    {
        /* Obtain a buffer that is at least xMaxStringLength characters big. The
        implementation of prvGetBuffer() is not shown. It might obtain the buffer from a pool of
        preallocated buffers or just allocate the buffer dynamically. */

        pcStringToSend = ( char * ) prvGetBuffer( xMaxStringLength );

        /* Write a string into the buffer. */

        snprintf( pcStringToSend, xMaxStringLength, "String number %d\r\n",
xStringNumber );
        /* Increment the counter so the string is different on each iteration of this task.
        */

        xStringNumber++;

        /* Send the address of the buffer to the queue that was created in the previous
        listing. The address of the buffer is stored in the pcStringToSend variable.*/

        xQueueSend( xPointerQueue, /* The handle of the queue. */ &pcStringToSend, /* The
        address of the pointer that points to the buffer. */ portMAX_DELAY );

    }
}
```

다음 코드에서는 버퍼를 가리키는 포인터를 대기열에서 수신한 다음 버퍼에 저장된 문자열을 출력합니다.

```
/* A task that receives the address of a buffer from the queue created in the first listing
and written to in the second listing. The buffer contains a string, which is printed out.
*/

void vStringReceivingTask( void *pvParameters )
{
    char *pcReceivedString;

    for( ;; )
    {
        /* Receive the address of a buffer. */

        xQueueReceive( xPointerQueue, /* The handle of the queue. */ &pcReceivedString, /*
Store the buffer's address in pcReceivedString. */ portMAX_DELAY );

        /* The buffer holds a string. Print it out. */

        vPrintString( pcReceivedString );

        /* The buffer is not required anymore. Release it so it can be freed or reused. */

        prvReleaseBuffer( pcReceivedString );
    }
}
```

## 대기열을 사용해 여러 형식과 길이의 데이터를 전송

구조를 대기열에 전송하는 기법과 포인터를 대기열에 전송하는 기법은 두 가지 강력한 설계 패턴입니다. 이 두 가지 기법을 결합하면 작업이 단일 대기열을 사용해 모든 데이터 원본에서 형식의 제한 없이 데이터를 수신할 수 있습니다. FreeRTOS+TCP TCP/IP 스택의 구현체는 두 가지 기법을 실제로 결합할 수 있는 예제를 제공합니다.

자체 작업에서 실행되는 TCP/IP 스택은 여러 가지 다른 소스에서 발생하는 이벤트를 처리해야 합니다. 이벤트 유형이 다르면 데이터 형식과 길이도 다릅니다. TCP/IP 스택 외부에서 발생하는 이벤트는 모두 IPStackEvent\_t 형식의 구조로 설명되며, 대기열을 통해 TCP/IP 작업으로 전송됩니다. IPStackEvent\_t 구조에서 pvData 멤버는 값을 직접 저장하거나, 혹은 버퍼를 가리킬 때 사용할 수 있는 포인터입니다.

FreeRTOS+TCP에서 이벤트를 TCP/IP 스택 작업에 전송할 때 사용되는 IPStackEvent\_t 구조는 다음과 같습니다.

```
/* A subset of the enumerated types used in the TCP/IP stack to identify events. */

typedef enum
{
    eNetworkDownEvent = 0, /* The network interface has been lost or needs (re)connecting.
    */

    eNetworkRxEvent, /* A packet has been received from the network. */

    eTCPAcceptEvent, /* FreeRTOS_accept() called to accept or wait for a new client. */
}
```



```
/* Other event types appear here but are not shown in this listing. */
} eIPEvent_t;

/* The structure that describes events and is sent on a queue to the TCP/IP task. */
typedef struct IP_TASK_COMMANDS
{
    /* An enumerated type that identifies the event. See the eIPEvent_t definition. */
    eIPEvent_t eEventType;

    /* A generic pointer that can hold a value or point to a buffer. */
    void *pvData;
} IPStackEvent_t;
```

TCP/IP 이벤트 예제와 관련 데이터에 포함되는 내용은 다음과 같습니다.

- eNetworkRxEvent: 데이터 패킷이 네트워크에서 수신되었습니다.

네트워크에서 수신된 데이터는 IPStackEvent\_t 형식의 구조를 사용해 TCP/IP 작업에 전송됩니다. 이 구조에서 eEventType 멤버는 eNetworkRxEvent로 설정됩니다. 구조에서 pvData 멤버는 수신된 데이터가 저장되는 버퍼를 가리키는 데 사용됩니다.

다음 의사 코드는 IPStackEvent\_t 구조를 사용해 네트워크에서 수신된 데이터를 TCP/IP 작업으로 전송하는 방법을 나타낸 것입니다.

```
void vSendRxDataToTheTCPTask( NetworkBufferDescriptor_t    *pRxedData )
{
    IPStackEvent_t xEventStruct;

    /* Complete the IPStackEvent_t structure. The received data is stored in pRxedData. */
    xEventStruct.eEventType = eNetworkRxEvent;

    xEventStruct.pvData = ( void * ) pRxedData;

    /* Send the IPStackEvent_t structure to the TCP/IP task. */
    xSendEventStructToIPTask( &xEventStruct );
}
```

- eTCPAcceptEvent: 소켓은 클라이언트의 연결을 허용하거나 대기할 때 사용됩니다.

허용 이벤트는 FreeRTOS\_accept()를 호출한 작업에서 IPStackEvent\_t 형식의 구조를 사용해 TCP/IP 작업으로 전송됩니다. 이 구조에서 eEventType 멤버는 eTCPAcceptEvent로 설정됩니다. 구조에서 pvData 멤버는 연결을 허용하는 소켓의 핸들로 설정됩니다.

다음 의사 코드는 IPStackEvent\_t 구조를 사용해 연결을 허용하는 소켓의 핸들을 TCP/IP 작업으로 전송하는 방법을 나타낸 것입니다.

```
void vSendAcceptRequestToTheTCPTask( Socket_t xSocket )
```

```
{  
  
    IPStackEvent_t xEventStruct;  
  
    /* Complete the IPStackEvent_t structure. */  
  
    xEventStruct.eEventType = eTCPAcceptEvent;  
  
    xEventStruct.pvData = ( void * ) xSocket;  
  
    /* Send the IPStackEvent_t structure to the TCP/IP task. */  
  
    xSendEventStructToIPTask( &xEventStruct );  
  
}
```

- eNetworkDownEvent: 네트워크는 연결 또는 재연결이 필요합니다.

네트워크 다운 이벤트가 IPStackEvent\_t 형식의 구조를 사용해 네트워크 인터페이스에서 TCP/IP 작업으로 전송됩니다. 이 구조에서 eEventType 멤버가 eNetworkDownEvent로 설정됩니다. 네트워크 다운 이벤트는 어떤 데이터와도 관련이 없기 때문에 구조의 pvData 멤버는 사용되지 않습니다.

다음 의사 코드는 IPStackEvent\_t 구조를 사용해 네트워크 다운 이벤트를 TCP/IP 작업으로 전송하는 방법을 나타낸 것입니다.

```
void vSendNetworkDownEventToTheTCPTask( Socket_t xSocket )  
{  
  
    IPStackEvent_t xEventStruct;  
  
    /* Complete the IPStackEvent_t structure. */  
  
    xEventStruct.eEventType = eNetworkDownEvent;  
  
    xEventStruct.pvData = NULL; /* Not used, but set to NULL for completeness. */  
  
    /* Send the IPStackEvent_t structure to the TCP/IP task. */  
  
    xSendEventStructToIPTask( &xEventStruct );  
  
}
```

TCP/IP 작업 내부에서 이러한 이벤트를 수신하여 처리하는 코드는 다음과 같습니다. 대기열에서 수신된 IPStackEvent\_t 구조의 eEventType 멤버는 pvData 멤버의 해석 방식을 결정하는 데 사용됩니다. 다음 의사 코드는 IPStackEvent\_t 구조를 사용해 네트워크 다운을 TCP/IP 작업으로 전송하는 방법을 나타낸 것입니다.

```
IPStackEvent_t xReceivedEvent;  
  
/* Block on the network event queue until either an event is received, or xNextIPSleep  
ticks pass without an event being received. eEventType is set to eNoEvent in case the  
call to xQueueReceive() returns because it timed out, rather than because an event was  
received. */  
  
xReceivedEvent.eEventType = eNoEvent;  
  
xQueueReceive( xNetworkEventQueue, &xReceivedEvent, xNextIPSleep );  
  
/* Which event was received, if any? */  
  
switch( xReceivedEvent.eEventType )
```

```
{  
  
    case eNetworkDownEvent :  
  
        /* Attempt to (re)establish a connection. This event is not associated with any  
        data. */  
  
        prvProcessNetworkDownEvent();  
  
        break;  
  
    case eNetworkRxEvent:  
  
        /* The network interface has received a new packet. A pointer to the received data  
        is stored in the pvData member of the received IPStackEvent_t structure. Process the  
        received data. */  
  
        prvHandleEthernetPacket( ( NetworkBufferDescriptor_t * )  
        ( xReceivedEvent.pvData ) );  
  
        break;  
  
    case eTCPAcceptEvent:  
  
        /* The FreeRTOS_accept() API function was called. The handle of the socket that  
        is accepting a connection is stored in the pvData member of the received IPStackEvent_t  
        structure. */  
  
        pxSocket = ( FreeRTOS_Socket_t * ) ( xReceivedEvent.pvData );  
  
        xTCPCheckNewClient( pxSocket );  
  
        break;  
  
        /* Other event types are processed in the same way, but are not shown here. */  
  
}
```

## 다수의 대기열에서 수신

## 대기열 집합

애플리케이션을 설계하다 보면 작업 하나에서 다양한 크기의 데이터와 다양한 의미를 갖는 데이터, 그리고 다른 소스의 데이터를 수신해야 할 때가 많습니다. 이전 단원에서는 단일 대기열에서 데이터 구조를 수신하여 이러한 문제를 효율적으로 해결할 수 있는 방법에 대해서 설명했습니다. 하지만 여러 가지 조건들이 설계를 위한 선택의 폭을 제한하여 일부 데이터 원본에 대해 별도의 대기열을 사용해야 할 때도 간혹 있습니다. 예를 들어 타사 코드를 설계에 통합할 때는 전용 대기열의 존재를 가정할 수 있습니다. 이때 사용할 수 있는 것이 바로 대기열 집합입니다.

대기열 집합을 사용하면 작업이 어떤 대기열에 데이터가 저장되어 있는지 확인하려고 각 대기열을 차례로 폴링하지 않고도 다수의 대기열에서 데이터를 수신할 수 있습니다.

대기열 집합을 사용해 다수의 소스에서 데이터를 수신하는 설계는 단일 대기열에서 데이터 구조를 수신하여 동일한 기능을 실현하는 설계와 비교했을 때 깔끔하거나 효율적이지 않습니다. 이러한 이유로 대기열 집합은 설계상 제약 조건으로 인해 절대적으로 필요한 경우에만 사용하는 것이 좋습니다.

이후 단원에서 살펴볼 내용은 다음과 같습니다.

1. 대기열 집합을 생성합니다.
2. 대기열을 집합에 추가합니다.

세마포어를 대기열 집합에 추가할 수도 있습니다. 세마포어에 대한 내용은 [인터럽트 관리 \(p. 117\)](#) 단원을 참조하십시오.

3. 대기열 집합에서 읽어와서 집합에 속한 대기열 중에서 어떤 대기열에 데이터가 저장되어 있는지 확인합니다.

집합의 멤버인 대기열이 데이터를 수신하면 수신 대기열의 핸들이 대기열 집합으로 전송되고, 작업이 함수를 호출하여 대기열 집합에서 데이터를 읽어올 때 다시 반환됩니다. 따라서 대기열 핸들이 대기열 집합에서 반환되면 핸들을 통한 참조 대기열에 데이터가 저장된 것으로 알려지고, 작업이 해당 대기열에서 직접 데이터를 읽어올 수 있습니다.

참고: 대기열이 대기열 집합의 멤버일 때는 대기열 핸들을 대기열 집합에서 처음 읽어온 경우가 아니라면 대기열에서 데이터를 읽지 마십시오.

대기열 집합 기능을 활성화하려면 FreeRTOSConfig.h에서 configUSE\_QUEUE\_SETS 컴파일 시간 구성 상수를 1로 설정하십시오.

## xQueueCreateSet() API 함수

대기열 집합을 사용하려면 먼저 명시적으로 생성해야 합니다.

대기열 집합은 QueueSetHandle\_t 형식의 변수인 핸들을 통해 참조됩니다. xQueueCreateSet() API 함수는 대기열 집합을 생성한 후 생성한 대기열 집합을 참조하는 QueueSetHandle\_t를 반환합니다.

xQueueCreateSet() API 함수 프로토타입은 다음과 같습니다.

```
QueueSetHandle_t xQueueCreateSet( const UBaseType_t uxEventQueueLength);
```

다음 표는 xQueueCreateSet() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름	설명
uxEventQueueLength	<p>대기열 집합의 멤버인 대기열이 데이터를 수신하면 수신 대기열의 핸들이 대기열 집합으로 전송됩니다. uxEventQueueLength는 생성된 대기열 집합이 언제든지 한 번에 저장할 수 있는 대기열 핸들의 최대 수를 정의합니다.</p> <p>대기열 핸들은 집합에 속한 대기열이 데이터를 수신할 때만 대기열 집합으로 전송됩니다. 대기열이 가득 찼을 때는 데이터를 수신하지 못하기 때문에 집합에 속한 대기열이 모두 가득 찼을 때는 대기열 핸들을 대기열 집합으로 전송하지 못합니다. 따라서 대기열 집합이 한 번에 저장할 수 있는 최대 항목 수는 집합에 속한 모든 대기열 길이의 총합입니다.</p> <p>예를 들어 집합에 비어있는 대기열이 3개이고, 각 대기열의 길이가 5라면 집합에 속한 모든 대기열이 가득 차기 전에 수신할 수 있는 총 항목 수는 15개(대기열 3개 x 각 항목 수 5개)가 됩니다. 이번 예에서는 uxEventQueueLength를 15로 설정해야만 대기열 집합이 전송되는 모든 항목을 수신할 수 있습니다.</p> <p>세마포어를 대기열 집합에 추가할 수도 있습니다. 이진 및 계수 세마포어는 이번 안내서 후반에서 자</p>

	<p>세하게 다릅니다. <code>uxEventQueueLength</code>의 길이를 계산하기 위해 이진 세마포어의 길이를 1로, 그리고 계수 세마포어의 길이를 최대 계수 값으로 가정합니다.</p> <p>또 다른 예를 들어 대기열 집합에 길이가 3인 대기열과 이진 세마포어(길이 1)가 포함되어 있다면 <code>uxEventQueueLength</code>는 <math>4(3+1)</math>로 설정되어야 합니다.</p>
반환 값	<p>NULL을 반환하는 경우에는 대기열 집합을 생성할 수 없습니다. FreeRTOS에서 대기열 집합 데이터 구조와 저장 영역을 할당하는 데 필요한 힙 메모리가 부족하기 때문입니다.</p> <p>NULL이 아닌 값을 반환하는 경우에는 대기열 집합이 성공적으로 생성된 것입니다. 반환 값은 생성된 대기열 집합에 대한 핸들로 저장되어야 합니다.</p>

## xQueueAddToSet() API 함수

`xQueueAddToSet()`는 대기열 또는 세마포어를 대기열 집합에 추가합니다. 세마포어에 대한 자세한 내용은 [인터럽트 관리 \(p. 117\)](#) 단원을 참조하십시오.

`xQueueAddToSet()` API 함수 프로토타입은 다음과 같습니다.

```
BaseType_t xQueueAddToSet( QueueSetMemberHandle_t xQueueOrSemaphore, QueueSetHandle_t xQueueSet );
```

다음 표는 `xQueueAddToSet()` 파라미터와 반환 값을 나열한 것입니다.

## xQueueSelectFromSet() API 함수

`xQueueSelectFromSet()`는 대기열 핸들을 대기열 집합에서 읽어옵니다.

집합의 멤버인 대기열 또는 세마포어가 데이터를 수신하면 수신 대기열 또는 세마포어의 핸들이 대기열 집합으로 전송되고, 작업이 `xQueueSelectFromSet()`를 호출할 때 반환됩니다. `xQueueSelectFromSet()` 호출을 통해 핸들이 반환되면 핸들을 통한 참조 대기열 또는 세마포어에 데이터가 저장된 것으로 알려지고, 호출 작업은 해당 대기열 또는 세마포어에서 데이터를 읽어와야 합니다.

참고: 대기열 또는 세마포어의 핸들이 `xQueueSelectFromSet()` 호출을 통해 처음 반환된 경우가 아니라면 집합의 멤버인 대기열 또는 세마포어에서 데이터를 읽지 마십시오. `xQueueSelectFromSet()` 호출을 통해 대기열 또는 세마포어 핸들이 반환될 때마다 대기열 또는 세마포어에서 읽어올 수 있는 항목은 1개로 제한됩니다.

`xQueueSelectFromSet()` API 함수 프로토타입은 다음과 같습니다.

```
QueueSetMemberHandle_t xQueueSelectFromSet( QueueSetHandle_t xQueueSet, const TickType_t xTicksToWait );
```

다음은 `xQueueSelectFromSet()` 파라미터와 반환 값을 나열한 것입니다.

`xQueueSet`

대기열 또는 세마포어 핸들을 수신 중인(읽어오는) 대기열 집합의 핸들입니다. 대기열 집합 핸들은 대기열 집합 생성을 위해 `xQueueCreateSet()`를 호출할 때 반환됩니다.

`xTicksToWait`

집합에 속한 대기열과 세마포어가 모두 비어있을 경우 호출 작업이 대기열 집합에서 대기열 또는 세마포어 핸들을 수신할 때까지 Blocked 상태로 대기해야 하는 최대 시간입니다. `xTicksToWait`가 0이면 집합에 속한 대기열 및 세마포어가 모두 비어있더라도 `xQueueSelectFromSet()`가 즉시 값을 반환합니다. 차단 시간은 틱 주기로 지정되기 때문에 틱 주파수에 따라 절대 시간이 달라집니다. 밀리초 단위의 시간은 `pdMS_TO_TICKS()` 매크로를 사용해 틱으로 변환할 수 있습니다. `xTicksToWait`를 `portMAX_DELAY`로 설정하면 작업이 제한 시간 없이 무기한 대기하게 됩니다. 단, `INCLUDE_vTaskSuspend`가 `FreeRTOSConfig.h`에서 1로 설정되어 있어야 합니다.

반환 값

NULL이 아닌 반환 값은 데이터가 저장된 것으로 알려진 대기열 또는 세마포어의 핸들입니다. 차단 시간이 지정된 경우에는(`xTicksToWait`가 0이 아닌 경우) 집합에 속한 대기열 또는 세마포어에서 데이터를 사용할 수 있을 때까지 호출 작업이 Blocked 상태로 전환되어 대기하지만 차단 시간이 지나기 전에 핸들을 대기열 집합에서 성공적으로 읽어왔을 가능성이 있습니다. 핸들은 `QueueSetMemberHandle_t` 형식으로 반환되며, 이 형식은 `QueueHandle_t` 형식 또는 `SemaphoreHandle_t` 형식으로 변환하는 것도 가능합니다.

반환 값이 NULL이면 대기열 집합에서 핸들을 읽어올 수 없는 것입니다. 차단 시간이 지정된 경우에는(`xTicksToWait`가 0이 아닌 경우) 호출 작업이 Blocked 상태로 전환되어 다른 작업 또는 인터럽트가 데이터를 집합에 속한 대기열 또는 세마포어에 전송할 때까지 대기하지만 이전에 차단 시간이 먼저 지나버린 것입니다.

## 대기열 집합 사용(예제 12)

이번 예제에서는 전송 작업 2개와 수신 작업 1개를 생성합니다. 전송 작업은 각 작업당 1개씩 2개의 대기열을 통해 데이터를 수신 작업으로 전송합니다. 대기열 2개는 대기열 집합에 추가되고, 수신 작업이 대기열 집합에서 읽어와서 두 대기열 중 어디에 데이터가 저장되어 있는지 확인합니다.

작업과 대기열, 그리고 대기열 집합은 모두 `main()`에서 생성됩니다.

```
/* Declare two variables of type QueueHandle_t. Both queues are added to the same queue set. */

static QueueHandle_t xQueue1 = NULL, xQueue2 = NULL;

/* Declare a variable of type QueueSetHandle_t. This is the queue set to which the two queues are added. */

static QueueSetHandle_t xQueueSet = NULL;

int main( void )
{
    /* Create the two queues, both of which send character pointers. The priority of the receiving task is above the priority of the sending tasks, so the queues will never have more than one item in them at any one time*/

    xQueue1 = xQueueCreate( 1, sizeof( char * ) );

    xQueue2 = xQueueCreate( 1, sizeof( char * ) );

    /* Create the queue set. Two queues will be added to the set, each of which can contain 1 item, so the maximum number of queue handles the queue set will ever have to hold at one time is 2 (2 queues multiplied by 1 item per queue). */
```

```
xQueueSet = xQueueCreateSet( 1 * 2 );

/* Add the two queues to the set. */
xQueueAddToSet( xQueue1, xQueueSet );
xQueueAddToSet( xQueue2, xQueueSet );

/* Create the tasks that send to the queues. */
xTaskCreate( vSenderTask1, "Sender1", 1000, NULL, 1, NULL );
xTaskCreate( vSenderTask2, "Sender2", 1000, NULL, 1, NULL );

/* Create the task that reads from the queue set to determine which of the two queues
contain data. */
xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );

/* Start the scheduler so the created tasks start executing. */
vTaskStartScheduler();

/* As normal, vTaskStartScheduler() should not return, so the following lines will
never execute. */

for( ;; );

return 0;
}
```

첫 번째 전송 작업은 xQueue1을 사용하여 문자 포인터를 100밀리초마다 수신 작업으로 전송합니다. 두 번째 전송 작업은 xQueue2를 사용하여 문자 포인터를 200밀리초마다 수신 작업으로 전송합니다. 문자 포인터는 전송 작업의 식별 문자열을 가리키도록 설정됩니다. 두 전송 작업의 구현체는 다음과 같습니다.

```
void vSenderTask1( void *pvParameters )
{
    const TickType_t xBlockTime = pdMS_TO_TICKS( 100 );

    const char * const pcMessage = "Message from vSenderTask1\r\n";

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Block for 100ms. */
        vTaskDelay( xBlockTime );

        /* Send this task's string to xQueue1. It is not necessary to use a block time,
        even though the queue can only hold one item. This is because the priority of the task
        that reads from the queue is higher than the priority of this task. As soon as this task
        writes to the queue, it will be preempted by the task that reads from the queue, so the
        queue will already be empty again by the time the call to xQueueSend() returns. The block
        time is set to 0. */
        xQueueSend( xQueue1, &pcMessage, 0 );
    }
}
```

```

    }

}

/*-----*/

void vSenderTask2( void *pvParameters )
{
    const TickType_t xBlockTime = pdMS_TO_TICKS( 200 );

    const char * const pcMessage = "Message from vSenderTask2\r\n";

    /* As per most tasks, this task is implemented within an infinite loop. */

    for( ;; )
    {
        /* Block for 200ms. */

        vTaskDelay( xBlockTime );

        /* Send this task's string to xQueue2. It is not necessary to use a block time,
        even though the queue can only hold one item. This is because the priority of the task
        that reads from the queue is higher than the priority of this task. As soon as this task
        writes to the queue, it will be preempted by the task that reads from the queue, so the
        queue will already be empty again by the time the call to xQueueSend() returns. The block
        time is set to 0. */

        xQueueSend( xQueue2, &pcMessage, 0 );

    }
}

```

전송 작업에서 작성되는 대기열은 동일한 대기열 집합의 멤버입니다. 작업이 대기열 중 하나로 전송할 때마다 대기열 핸들이 대기열 집합으로 전송됩니다. 수신 작업은 `xQueueSelectFromSet()`를 호출하여 대기열 집합에서 대기열 핸들을 읽어옵니다. 수신 작업이 집합에서 대기열 핸들을 수신하고, 수신된 핸들의 참조 대기열에 데이터가 저장되어 있는 것을 인지한 후 해당 대기열에서 데이터를 직접 읽어옵니다. 대기열에서 읽어오는 데이터는 수신 작업이 출력할 문자열을 가리키는 포인터입니다.

`xQueueSelectFromSet()` 호출이 제한 시간을 초과하면 NULL을 반환합니다. 이전 코드에서는 `xQueueSelectFromSet()`가 제한 시간을 지정하지 않고 호출되어 제한 시간을 초과하는 일 없이 유효한 대기열 핸들만 반환할 수 있습니다. 따라서 수신 작업이 반환 값을 사용하기 전에 `xQueueSelectFromSet()`에서 NULL이 반환되었는지 확인할 필요도 없습니다.

`xQueueSelectFromSet()`는 핸들의 참조 대기열에 데이터가 저장되어 있는 경우에만 대기열 핸들을 반환하기 때문에 대기열에서 읽어올 때 제한 시간을 사용할 필요는 없습니다.

수신 작업의 구현체는 다음과 같습니다.

```

void vReceiverTask( void *pvParameters )
{
    QueueHandle_t xQueueThatContainsData;

    char *pcReceivedString;

    /* As per most tasks, this task is implemented within an infinite loop.*/

```



```

for( ;; )

{

    /* Block on the queue set to wait for one of the queues in the set to contain
data. Cast the QueueSetMemberHandle_t value returned from xQueueSelectFromSet() to a
QueueHandle_t because it is known all the members of the set are queues (the queue set
does not contain any semaphores). */

    xQueueThatContainsData = ( QueueHandle_t ) xQueueSelectFromSet(xQueueSet,
portMAX_DELAY );

    /* An indefinite block time was used when reading from the queue set, so
xQueueSelectFromSet() will not have returned unless one of the queues in the set contained
data, and xQueueThatContainsData cannot be NULL. Read from the queue. It is not necessary
to specify a block time because it is known the queue contains data. The block time is set
to 0. */

    xQueueReceive( xQueueThatContainsData, &pcReceivedString, 0 );

    /* Print the string received from the queue. */

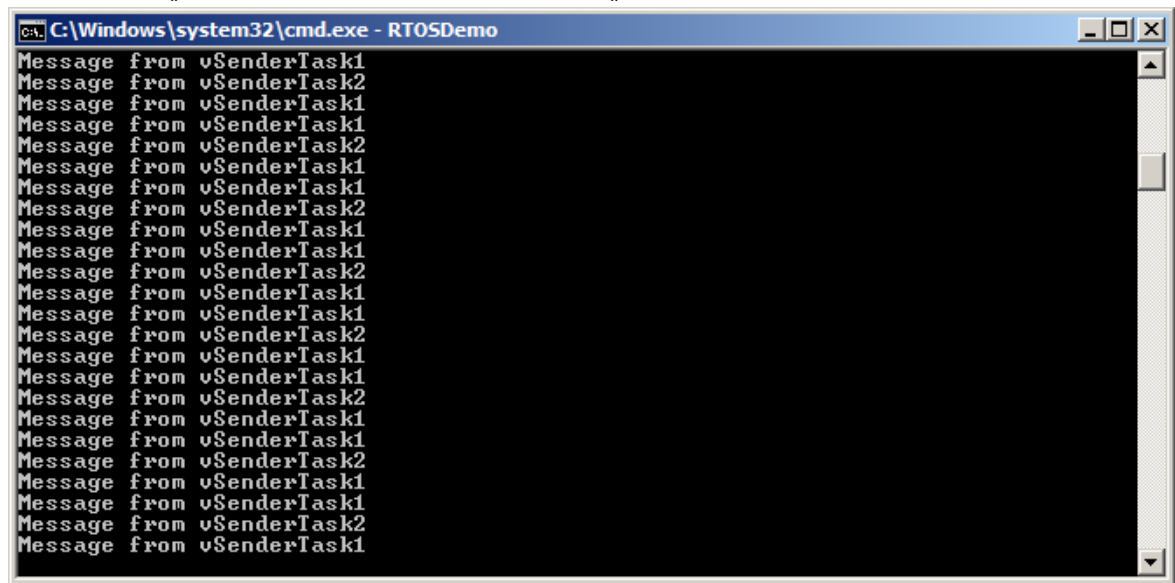
    vPrintString( pcReceivedString );

}

}

```

출력되는 화면은 다음과 같습니다. 수신 작업은 두 전송 작업 모두에서 문자열을 수신합니다. vSenderTask1()에서 사용하는 제한 시간이 vSenderTask2()에서 사용하는 제한 시간의 절반이기 때문에 vSenderTask1()에서 전송되는 문자열이 vSenderTask2()에서 전송하는 문자열의 2배로 출력됩니다.



## 더욱 현실적인 대기열 집합의 사용 사례

이전 예제에서는 대기열 집합에 대기열이 2개만 포함되었고, 대기열이 문자 포인터를 전송하는 데 사용되었습니다. 하지만 실제 애플리케이션에서는 대기열 집합에 대기열과 세마포어가 모두 포함되거나, 혹은 대기열에 저장되는 데이터 형식이 다를 수도 있습니다. 이런 경우에는 반환 값을 사용하기 전에 `xQueueSelectFromSet()`에서 반환되는 값을 테스트해야 합니다.

아래 코드는 집합에 다음과 같은 멤버가 속해 있다고 가정할 때 xQueueSelectFromSet()에서 반환되는 값의 사용 방법을 나타낸 것입니다.

1. 이진 세마포어
2. 문자 포인터를 읽어오는 대기열
3. uint32\_t 값을 읽어오는 대기열

이번 코드는 대기열과 세마포어가 이미 생성되어 대기열 집합에 추가되었다는 가정을 전제로 합니다.

```
/* The handle of the queue from which character pointers are received. */
QueueHandle_t xCharPointerQueue;

/* The handle of the queue from which uint32_t values are received.*/
QueueHandle_t xUInt32tQueue;

/* The handle of the binary semaphore. */
SemaphoreHandle_t xBinarySemaphore;

/* The queue set to which the two queues and the binary semaphore belong. */
QueueSetHandle_t xQueueSet;

void vAMoreRealisticReceiverTask( void *pvParameters )
{
    QueueSetMemberHandle_t xHandle;

    char *pcReceivedString;

    uint32_t ulRecievedValue;

    const TickType_t xDelay100ms = pdMS_TO_TICKS( 100 );

    for( ;; )
    {
        /* Block on the queue set for a maximum of 100ms to wait for one of the members of
        the set to contain data. */

        xHandle = xQueueSelectFromSet( xQueueSet, xDelay100ms );

        /* Test the value returned from xQueueSelectFromSet(). If the returned value
        is NULL, then the call to xQueueSelectFromSet() timed out. If the returned value is
        not NULL, then the returned value will be the handle of one of the set's members. The
        QueueSetMemberHandle_t value can be cast to either a QueueHandle_t or a SemaphoreHandle_t.
        Whether an explicit cast is required depends on the compiler. */

        if( xHandle == NULL )
        {
            /* The call to xQueueSelectFromSet() timed out. */

        }

        else if( xHandle == ( QueueSetMemberHandle_t ) xCharPointerQueue )
        {

```

```
/* The call to xQueueSelectFromSet() returned the handle of the queue that
receives character pointers. Read from the queue. The queue is known to contain data, so a
block time of 0 is used. */

xQueueReceive( xCharPointerQueue, &pcReceivedString, 0 );

/* The received character pointer can be processed here... */

}

else if( xHandle == ( QueueSetMemberHandle_t ) xUint32tQueue )
{

    /* The call to xQueueSelectFromSet() returned the handle of the queue that
    receives uint32_t types. Read from the queue. The queue is known to contain data, so a
    block time of 0 is used. */

    xQueueReceive(xUint32tQueue, &ulRecievedValue, 0 );

    /* The received value can be processed here... */

}

else if( xHandle == ( QueueSetMemberHandle_t ) xBinarySemaphore )
{

    /* The call to xQueueSelectFromSet() returned the handle of the binary
    semaphore. Take the semaphore now. The semaphore is known to be available, so a block time
    of 0 is used. */

    xSemaphoreTake( xBinarySemaphore, 0 );

    /* Whatever processing is necessary when the semaphore is taken can be
    performed here... */

}

}

}
```

## 대기열을 사용하여 사서함 생성

사서함이라는 용어는 아직 임베디드 커뮤니티에서 합의한 의미가 없습니다. 이번 안내서에서는 길이가 1인 대기열을 의미하는 용어로 사용합니다. 사서함을 대기열이라고 얘기할 수 있는 이유는 대기열에 대한 기능적 차이보다는 애플리케이션의 사용 방식에서 찾을 수 있습니다.

- 대기열은 작업에서 다른 작업으로, 혹은 인터럽트 서비스 루틴에서 작업으로 데이터를 전송하는 데 사용됩니다. 발신자는 대기열에 데이터 항목을 추가하고, 수신자는 대기열에서 해당 항목을 삭제합니다. 데이터는 대기열을 통해 발신자에서 수신자로 전달됩니다.
- 사서함은 작업 또는 인터럽트 서비스 루틴이 읽어올 수 있는 데이터를 저장하는 데 사용됩니다. 데이터가 사서함을 통해 전달되지는 않습니다. 다만 덮어쓸 때까지 사서함에 저장될 뿐입니다. 발신자는 사서함의 값을 덮어씁니다. 수신자는 사서함에서 값을 읽어오지만 삭제하지는 않습니다.

xQueueOverwrite() 및 xQueuePeek() API 함수에서는 대기열을 사서함으로 사용할 수 있습니다.

다음 코드는 사서함으로 사용할 목적으로 생성하는 대기열을 나타낸 것입니다.

```

    /* A mailbox can hold a fixed-size data item. The size of the data item is set when the
    mailbox (queue) is created. In this example, the mailbox is created to hold an Example_t
    structure. Example_t includes a timestamp to allow the data held in the mailbox to note
    the time at which the mailbox was last updated. The timestamp used in this example is for
    demonstration purposes only. A mailbox can hold any data the application writer wants, and
    the data does not need to include a timestamp. */

typedef struct xExampleStructure
{
    TickType_t xTimeStamp;

    uint32_t ulValue;
} Example_t;

/* A mailbox is a queue, so its handle is stored in a variable of type
QueueHandle_t. */
QueueHandle_t xMailbox;

void vAFunction( void )
{
    /* Create the queue that is going to be used as a mailbox. The queue has a length of 1
    to allow it to be used with the xQueueOverwrite() API function, which is described below.
    */

    xMailbox = xQueueCreate( 1, sizeof( Example_t ) );
}

```

## xQueueOverwrite() API 함수

xQueueSendToBack() API 함수와 마찬가지로 xQueueOverwrite() API 함수 역시 데이터를 대기열로 전송합니다. 하지만 대기열이 가득 찼을 경우 xQueueOverwrite()는 xQueueSendToBack()과 달리 대기열에 이미 저장된 데이터를 덮어씁니다.

xQueueOverwrite()는 길이가 1인 대기열과 함께 사용해야 합니다. 이러한 제한은 대기열이 가득 찼을 경우 함수의 구현체가 대기열에서 덮어쓸 항목을 임의로 결정하지 못하도록 하기 위해서입니다.

참고: xQueueOverwrite()를 인터럽트 서비스 루틴에서 호출하지 마십시오. 대신에 인터럽트 세이프 버전인 xQueueOverwriteFromISR()을 사용하십시오.

xQueueOverwrite() API 함수 프로토타입은 다음과 같습니다.

```
BaseType_t xQueueOverwrite( QueueHandle_t xQueue, const void * pvItemToQueue );
```

다음 표는 xQueueOverwrite() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
xQueue	데이터가 전송되는(작성되는) 대기열의 핸들입니다. 대기열 핸들은 대기열 생성을 위해 xQueueCreate()를 호출할 때 반환됩니다.

pvltemToQueue	대기열로 복사할 데이터를 가리키는 포인터입니다.  대기열에 저장할 수 있는 각 항목의 크기는 대기열을 생성할 때 설정됩니다. 따라서 여기에서 설정된 크기의 바이트가 pvltemToQueue에서 대기열 저장 영역으로 복사됩니다.
반환 값	xQueueOverwrite()는 대기열이 가득 찼을 때도 대기열에 작성하기 때문에 pdPASS가 유일하게 반환될 수 있는 값입니다.

다음 코드는 앞에서 생성한 사서함(대기열)에 작성할 때 사용되는 xQueueOverwrite()를 나타낸 것입니다.

```
void vUpdateMailbox( uint32_t ulNewValue )
{
    /* Example_t was defined in the earlier code example. */
    Example_t xData;

    /* Write the new data into the Example_t structure.*/
    xData.ulValue = ulNewValue;

    /* Use the RTOS tick count as the timestamp stored in the Example_t structure. */
    xData.xTimeStamp = xTaskGetTickCount();

    /* Send the structure to the mailbox, overwriting any data that is already in the mailbox. */
    xQueueOverwrite( xMailbox, &xData );
}
```

## xQueuePeek() API 함수

xQueuePeek() 함수는 대기열에서 항목을 삭제하지 않고 수신하는(읽어오는) 데 사용됩니다. 이 함수는 대기열에 저장된 데이터 또는 대기열에 저장된 데이터 순서를 수정하지 않고 대기열 앞부터 데이터를 수신합니다.

참고: xQueuePeek()를 인터럽트 서비스 루틴에서 호출하지 마십시오. 대신에 인터럽트 세이프 버전인 xQueuePeekFromISR()을 사용하십시오.

xQueuePeek()는 함수 파라미터와 반환 값이 xQueueReceive()와 동일합니다.

다음 코드는 이전 예제에서 생성한 사서함(대기열)에 게시된 항목을 수신할 때 사용되는 xQueuePeek()를 나타낸 것입니다.

```
BaseType_t vReadMailbox( Example_t *pxData )
{
    TickType_t xPreviousTimeStamp;

    BaseType_t xDataUpdated;
```

```
/* This function updates an Example_t structure with the latest value received from the
mailbox. Record the timestamp already contained in *pxData before it gets overwritten by
the new data. */

xPreviousTimeStamp = pxData->xTimeStamp;

/* Update the Example_t structure pointed to by pxData with the data contained in
the mailbox. If xQueueReceive() was used here, then the mailbox would be left empty
and the data could not then be read by any other tasks. Using xQueuePeek() instead of
xQueueReceive() ensures the data remains in the mailbox. A block time is specified, so
the calling task will be placed in the Blocked state to wait for the mailbox to contain
data should the mailbox be empty. An infinite block time is used, so it is not necessary
to check the value returned from xQueuePeek(). xQueuePeek() will only return when data is
available. */

xQueuePeek( xMailbox, pxData, portMAX_DELAY );

/* Return pdTRUE if the value read from the mailbox has been updated since this
function was last called. Otherwise, return pdFALSE. */

if( pxData->xTimeStamp > xPreviousTimeStamp )
{
    xDataUpdated = pdTRUE;
}
else
{
    xDataUpdated = pdFALSE;
}

return xDataUpdated;
}
```

# 소프트웨어 타이머 관리

이번 단원에서 다루는 내용은 다음과 같습니다.

- 작업의 특성과 소프트웨어 타이머의 특성 비교
- RTOS 데몬 작업
- 타이머 명령 대기열
- 일회성 소프트웨어 타이머와 주기적 소프트웨어 타이머의 차이점
- 소프트웨어 타이머의 주기를 생성하고, 시작하고, 재설정하고, 변경하는 방법

소프트웨어 타이머는 함수가 향후 설정 시간, 혹은 고정된 주기로 실행되도록 예약하는 데 사용됩니다. 소프트웨어 타이머에서 실행되는 함수를 소프트웨어 타이머의 콜백 함수라고 부릅니다.

소프트웨어 타이머는 FreeRTOS 커널에서 직접 제어하여 구현됩니다. 하드웨어 지원은 필요 없습니다. 하드웨어 타이머 또는 하드웨어 카운터와 관련이 없기 때문입니다.

혁신적 설계를 사용해 효율을 극대화한다는 FreeRTOS 철학에 따라 소프트웨어 타이머는 소프트웨어 타이머 콜백 함수가 실행되지 않을 경우 처리 시간을 전혀 사용하지 않습니다.

소프트웨어 타이머 기능은 선택 사항입니다. 소프트웨어 타이머 기능을 추가하는 방법은 다음과 같습니다.

1. FreeRTOS 소스 파일인 FreeRTOS/Source/timers.c를 프로젝트에 포함시켜 빌드합니다.
2. FreeRTOSConfig.h에서 configUSE\_TIMERS를 1로 설정합니다.

## 소프트웨어 타이머 콜백 함수

소프트웨어 타이머 콜백 함수는 C 함수로 구현됩니다. 콜백 함수는 프로토타입이 유일하게 특별하여, 보이드를 반환해야 하는 동시에 파라미터로 소프트웨어 타이머 핸들만 사용합니다.

콜백 함수 프로토타입은 다음과 같습니다.

```
void ATimerCallback( TimerHandle_t xTimer );
```

소프트웨어 타이머 콜백 함수는 처음부터 끝까지 실행 후 정상적으로 종료됩니다. 또한 짧게 작성해야 하고, Blocked 상태가 되어서도 안 됩니다.

참고: 소프트웨어 타이머 콜백 함수는 FreeRTOS 스케줄러가 시작될 때 자동 생성되는 작업의 컨텍스트에서 실행됩니다. 따라서 호출 작업이 Blocked 상태로 전환되는 FreeRTOS API 함수를 호출해서는 안 됩니다. xQueueReceive() 같은 함수는 호출할 수 있지만 함수의 xTicksToWait 파라미터(함수의 차단 시간을 지정)가 0으로 설정된 경우에 한합니다. vTaskDelay() 같은 함수는 항상 호출 작업을 Blocked 상태로 전환하기 때문에 호출할 수 없습니다.

## 소프트웨어 타이머의 속성과 상태

### 소프트웨어 타이머의 주기

소프트웨어 타이머의 주기란 소프트웨어 타이머를 시작한 후부터 콜백 함수를 실행할 때까지 걸리는 시간을 말합니다.

## 일회성 타이머와 오토 리로드 타이머

소프트웨어 타이머는 다음과 같이 두 가지 유형이 있습니다.

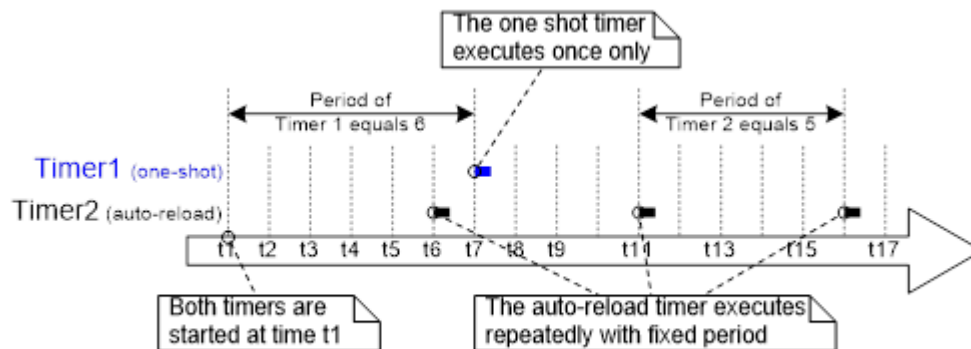
- 일회성 타이머

일회성 타이머는 시작 이후 콜백 함수를 한 번만 실행합니다. 일회성 타이머를 수동으로 다시 시작할 수는 있지만 스스로 다시 시작되지는 않습니다.

- 오토 리로드 타이머

오토 리로드 타이머는 시작 이후 만료될 때마다 자동으로 다시 시작되어 콜백 함수를 주기적으로 실행합니다.

다음 그림은 일회성 타이머와 오토 리로드 타이머의 동작 차이를 나타낸 것입니다. 수직 점선은 틱 인터럽트가 발생하는 시간을 의미합니다.



- Timer 1은 6틱 주기의 일회성 타이머입니다. 이 타이머는 t1에서 시작되기 때문에 콜백 함수가 나중에 6틱이 지난 t7에서 실행됩니다. Timer 1은 일회성 타이머이기 때문에 콜백 함수가 다시 실행되지는 않습니다.
- Timer 2는 5틱 주기의 오토 리로드 타이머입니다. 이 타이머는 t1에서 시작되기 때문에 콜백 함수가 t1 이후 5틱마다 실행됩니다. 그림에서는 t6, t11 및 t16이 이에 해당합니다.

## 소프트웨어 타이머 상태

소프트웨어 타이머는 다음 두 가지 상태 중 하나가 될 수 있습니다.

- Dormant

Dormant 상태의 소프트웨어 타이머가 존재하면 핸들을 통해 참조할 수 있지만 Running 상태가 아니기 때문에 콜백 함수도 실행되지 않습니다.

- Running

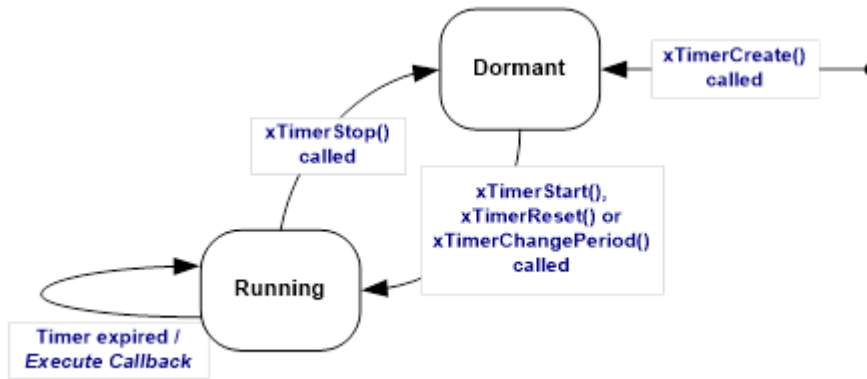
Running 상태의 소프트웨어 타이머는 소프트웨어 타이머가 Running 상태로 전환된 이후, 혹은 소프트웨어 타이머가 마지막으로 재설정된 이후 주기와 동일한 시간이 경과하였을 때 콜백 함수를 실행합니다.

다음 두 그림은 오토 리로드 타이머와 일회성 타이머가 각각 Dormant 상태와 Running 상태 사이에서 전환될 수 있는 경우를 나타낸 것입니다. 두 그림에서 가장 큰 차이는 타이머가 만료된 후 전환되는 상태입니다. 오토 리로드 타이머는 콜백 함수를 실행한 후 다시 Running 상태로 전환됩니다. 일회성 타이머는 콜백 함수를 실행한 후 Dormant 상태로 전환됩니다.

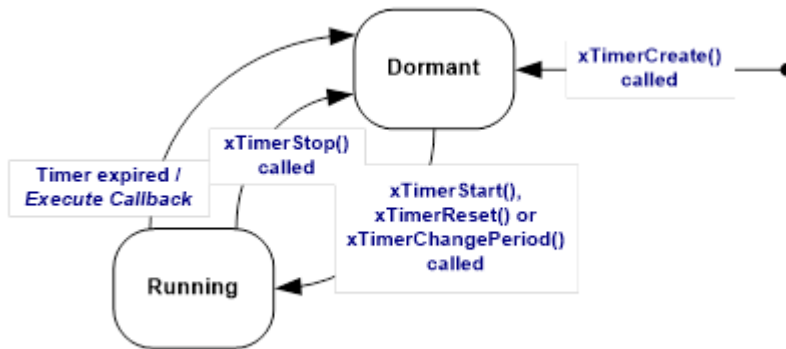
xTimerDelete() API 함수는 타이머를 삭제합니다. 타이머는 언제든지 삭제할 수 있습니다.



다음 그림은 오토 리로드 타이머의 상태와 전환을 나타낸 것입니다.



다음 그림은 일회성 타이머의 상태와 전환을 나타낸 것입니다.



## 소프트웨어 타이머의 컨텍스트

### RTOS 데몬(타이머 서비스) 작업

소프트웨어 타이머 콜백 함수는 모두 동일한 RTOS 데몬(또는 타이머 서비스) 작업의 컨텍스트에서 실행됩니다. (데몬 작업은 처음에 소프트웨어 타이머 콜백 함수를 실행할 목적으로만 사용되어 타이머 서비스 작업라고 불리기도 했습니다. 하지만 지금은 다른 목적으로도 사용되어 더욱 포괄적인 명칭인 RTOS 데몬 작업으로 알려져 있습니다)

데몬 작업은 스케줄러 시작 시 자동 생성되는 표준 FreeRTOS 작업입니다. 이 작업의 우선순위와 스택 크기는 각각 configTIMER\_TASK\_PRIORITY와 configTIMER\_TASK\_STACK\_DEPTH 컴파일 시간 구성 상수에서 설정됩니다. 두 상수는 모두 FreeRTOSConfig.h에서 정의됩니다.

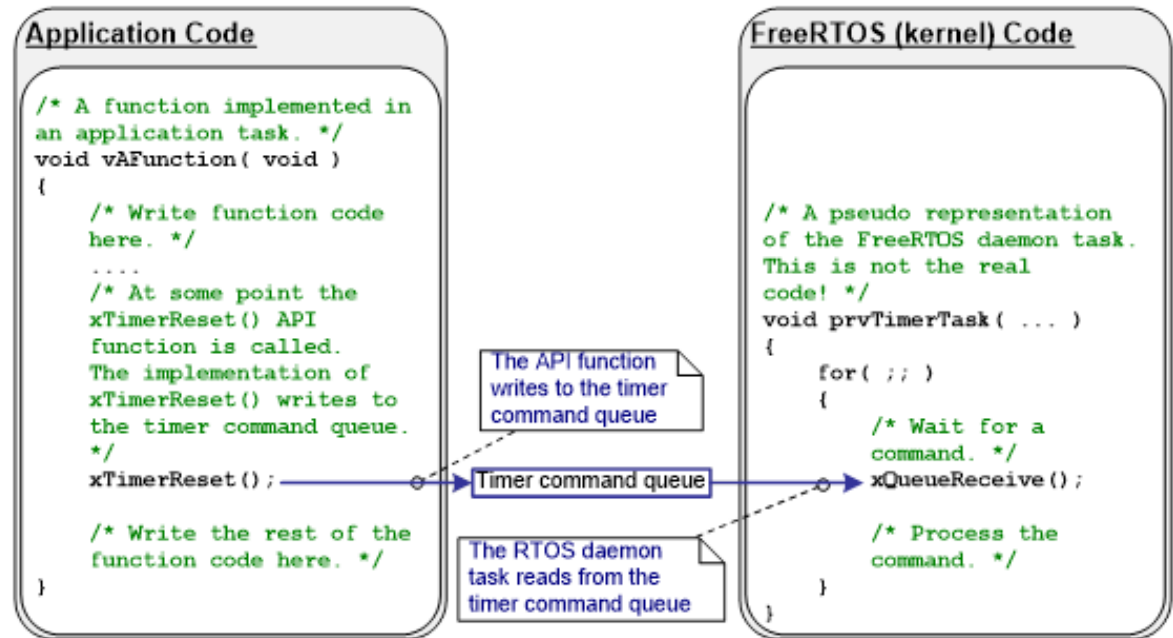
소프트웨어 타이머 콜백 함수는 호출 작업을 Blocked 상태로 전환하는 FreeRTOS API 함수를 호출해서는 안 됩니다. 그러면 데몬 작업이 Blocked 상태로 전환되기 때문입니다.

### 타이머 명령 대기열

소프트웨어 타이머 API 함수는 호출 작업에서 타이머 명령 대기열이라는 대기열의 데몬 작업으로 명령을 전송합니다. 전송되는 명령의 예를 들면 'start a timer', 'stop a timer', 'reset a timer' 등이 있습니다.

타이머 명령 대기열은 스케줄러 시작 시 자동 생성되는 표준 FreeRTOS 대기열입니다. 타이머 명령 대기열의 길이는 FreeRTOSConfig.h의 configTIMER\_QUEUE\_LENGTH 컴파일 시간 구성 상수에서 설정합니다.

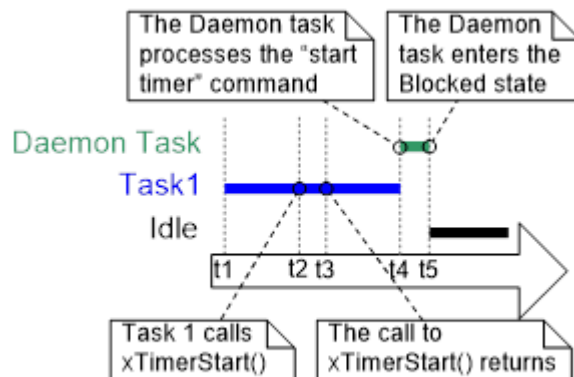
다음 그림은 소프트웨어 타이머 API 함수에서 RTOS 데몬 작업과 통신할 목적으로 사용되는 명령 대기열을 나타낸 것입니다.



## 데몬 작업 예약

데몬 작업은 다른 FreeRTOS 작업과 비슷하게 예약됩니다. 즉 데몬 작업의 우선순위가 실행 가능한 작업 중 가장 높을 때만 명령을 처리하거나 타이머 콜백 함수를 실행합니다. 다음 그림은 `configTIMER_TASK_PRIORITY` 설정이 실행 패턴에 어떠한 영향을 미치는지 나타낸 것입니다.

아래 그림은 데몬 작업의 우선순위가 `xTimerStart()` API 함수를 호출하는 작업의 우선순위보다 낮을 때 실행 패턴을 나타낸 것입니다.



### 1. t1에서

Task 1은 Running 상태이고, 데몬 작업은 Blocked 상태입니다.

명령이 타이머 명령 대기열로 전송되면 데몬 작업이 Blocked 상태를 종료하고 명령을 처리합니다. 소프트웨어 타이머가 만료되면 소프트웨어 타이머의 콜백 함수를 실행합니다.

### 2. t2에서



xTimerStart()가 명령을 타이머 명령 대기열로 전송하여 데몬 작업의 Blocked 상태가 종료됩니다. 데몬 작업의 우선순위가 Task 1의 우선순위보다 높기 때문에 스케줄러가 데몬 작업을 Running 상태로 전환할 작업으로 선택합니다.

Task 1이 xTimerStart() 함수 실행을 마치기 전에 데몬 작업에게 선점되면서 Ready 상태로 전환됩니다.

데몬 작업이 Task 1에서 타이머 명령 대기열로 전송된 명령을 처리하기 시작합니다.

### 3. t3에서

데몬 작업이 Task 1에서 전송된 명령 처리를 완료하고 타이머 명령 대기열에서 추가로 데이터를 수신하려고 합니다. 하지만 타이머 명령 대기열이 비어있기 때문에 데몬 작업은 다시 Blocked 상태로 전환됩니다.

이제 Task 1의 우선순위가 Ready 상태에서 가장 높기 때문에 스케줄러가 Task 1을 Running 상태로 전환할 작업으로 선택합니다.

### 4. t4에서

Task 1이 xTimerStart() 함수 실행을 마치기 전에 데몬 작업에게 선점되었기 때문에 Running 상태로 다시 전환된 후에만 xTimerStart()를 종료합니다(복귀됩니다)

### 5. t5에서

Task 1이 API 함수를 호출하여 Blocked 상태로 전환됩니다. 이제 유휴 작업의 우선순위가 Ready 상태에서 가장 높기 때문에 스케줄러가 유휴 작업을 Running 상태로 전환할 작업으로 선택합니다.

첫 번째 시나리오 그림에서는 명령을 타이머 명령 대기열에게 전송하는 Task 1과 명령을 수신하여 처리하는 데몬 작업 사이에 시간이 흘렀습니다. 데몬 작업은 Task 1이 명령을 전송한 함수에서 복귀되기 전에 Task 1에서 전송된 명령을 수신하여 처리했습니다.

타이머 명령 대기열로 전송된 명령에는 타임스탬프가 포함되어 있습니다. 이 타임스탬프는 명령이 애플리케이션 작업에서 전송되어 데몬 작업에서 처리될 때까지 걸리는 시간을 설명하는 데 사용됩니다. 예를 들어 10틱 주기의 타이머를 시작하려고 'start a timer' 명령을 전송한다면 타임스탬프는 데몬 작업에서 명령이 처리된 후 10틱이 아니라 명령이 전송된 후 10틱이 지나서 타이머가 만료되도록 하는 데 사용됩니다.

## 소프트웨어 타이머의 생성 및 시작

### xTimerCreate() API 함수

FreeRTOS V9.0.0에도 컴파일 과정에서 타이머를 정적으로 생성하는 데 필요한 메모리를 할당하는 xTimerCreateStatic() 함수가 포함되어 있습니다. 소프트웨어 타이머를 사용하려면 먼저 명시적으로 생성해야 합니다.

소프트웨어 타이머는 TimerHandle\_t 형식의 변수를 통해 참조됩니다. xTimerCreate()는 소프트웨어 타이머를 생성하는 데 사용되며, 생성하는 소프트웨어 타이머를 참조할 수 있도록 TimerHandle\_t를 반환합니다. 소프트웨어 타이머는 Dormant 상태로 생성됩니다.

소프트웨어 타이머는 스케줄러 실행 이전에, 혹은 스케줄러가 시작된 이후 작업에서 생성할 수도 있습니다

xTimerCreate() API 함수 프로토타입은 다음과 같습니다.

```
TimerHandle_t xTimerCreate( const char * const pcTimerName, TickType_t xTimerPeriodInTicks,
    UBaseType_t uxAutoReload, void * pvTimerID, TimerCallbackFunction_t pxCallbackFunction );
```

다음 표는 xTimerCreate() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
pcTimerName	타이머를 나타내는 서술형 이름입니다. FreeRTOS에서는 이 이름을 사용하지 않습니다. 이 이름이 포함되는 이유는 디버깅을 위해서입니다. 인간이 판독할 수 있는 이름으로 타이머를 식별하는 것이 핸들로 식별하는 것보다 훨씬 쉽습니다.
xTimerPeriodInTicks	틱 단위로 지정되는 타이머 주기입니다. 밀리초 단위의 시간은 pdMS_TO_TICKS() 매크로를 사용해 틱으로 변환할 수 있습니다.
uxAutoReload	uxAutoReload를 pdTRUE로 설정하면 오토 리로드 타이머가 생성됩니다. uxAutoReload를 pdFALSE로 설정하면 일회성 타이머가 생성됩니다.
pvTimerID	소프트웨어 타이머는 모두 ID 값이 있습니다. 이 ID는 보이드 포인터이기 때문에 애플리케이션 개발자가 어떤 목적으로든 사용할 수 있습니다. ID는 특히 각 타이머에게 저장 공간을 따로 제공할 때 사용할 수 있기 때문에 다수의 소프트웨어 타이머에서 동일한 콜백 함수를 사용할 때 유용합니다.  pvTimerID는 생성되는 작업의 초기 ID 값을 설정합니다.
pxCallbackFunction	소프트웨어 타이머 콜백 함수는 <a href="#">소프트웨어 타이머 콜백 함수 (p. 96)</a> 에 나와있는 프로토타입을 따르는 C 함수일 뿐입니다. pxCallbackFunction 파라미터는 생성되는 소프트웨어 타이머의 콜백 함수로 사용할 함수를 가리키는 포인터(실제 함수 이름)입니다.
반환 값	NULL을 반환하는 경우에는 소프트웨어 타이머를 생성할 수 없습니다. FreeRTOS에서 데이터 구조를 할당하는 데 필요한 힙 메모리가 부족하기 때문입니다.  NULL이 아닌 값을 반환하는 경우에는 소프트웨어 타이머가 성공적으로 생성된 것입니다. 반환 값이 생성된 타이머의 핸들입니다.

## xTimerStart() API 함수

xTimerStart()는 Dormant 상태의 소프트웨어 타이머를 시작하거나, 혹은 Running 상태의 소프트웨어 타이머를 재설정(재시작)하는 데 사용됩니다. 반대로 xTimerStop()은 Running 상태의 소프트웨어 타이머를 중단하는 데 사용됩니다. 소프트웨어 타이머를 중단한다는 말은 Dormant 상태로 전환한다는 것과 동일합니다.

xTimerStart()는 스케줄러를 시작하기 전에 호출할 수 있지만 소프트웨어 타이머는 스케줄러가 시작하는 시간이 될 때까지 시작하지 않습니다.

참고: xTimerStart()를 인터럽트 서비스 루틴에서 호출하지 마십시오. 대신에 인터럽트 세이프 버전인 xTimerStartFromISR()을 사용하십시오.

xTimerStart() API 함수 프로토타입은 다음과 같습니다.

```
TimerHandle_t xTimerStart( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

다음 표는 xTimerStart() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
xTimer	시작 또는 재설정되는 소프트웨어 타이머의 핸들입니다. 이 핸들은 소프트웨어 타이머 생성을 위해 xTimerCreate()를 호출할 때 반환됩니다.
xTicksToWait	<p>xTimerStart()는 타이머 명령 대기열을 사용해 'start a timer' 명령을 데몬 작업에게 전송합니다. xTicksToWait는 대기열이 이미 가득 찬 경우 호출 작업이 타이머 명령 대기열에서 사용할 수 있는 공간이 나올 때까지 Blocked 상태로 대기해야 하는 최대 시간을 지정합니다.</p> <p>xTicksToWait가 0이고 타이머 명령 대기열이 이미 가득 찬 상태라면 xTimerStart()가 즉시 값을 반환합니다.</p> <p>차단 시간은 틱 주기로 지정되기 때문에 틱 주파수에 따라 절대 시간이 달라집니다. 밀리초 단위의 시간은 pdMS_TO_TICKS() 매크로를 사용해 틱으로 변환할 수 있습니다.</p> <p>INCLUDE_vTaskSuspend가 FreeRTOSConfig.h에서 1로 설정되었을 때 xTicksToWait를 portMAX_DELAY로 설정하면 호출 작업이 타이머 명령 대기열에서 사용할 수 있는 공간이 나올 때까지 Blocked 상태로 제한 시간 없이 무기한 대기합니다.</p> <p>스케줄러 시작 이전에 xTimerStart()이 호출되면 xTicksToWait 값이 무시되고 xTicksToWait의 값이 마치 0인 것처럼 xTimerStart()가 동작합니다.</p>
반환 값	<p>가능한 반환 값은 다음 두 가지입니다.</p> <ol style="list-style-type: none"> <li>1. pdPASS <ul style="list-style-type: none"> <li>'start a timer' 명령이 타이머 명령 대기열로 성공적으로 전송된 경우에 한해 pdPASS가 반환됩니다.</li> <li>데몬 작업의 우선순위가 xTimerStart()를 호출한 작업의 우선순위보다 높으면 스케줄러에 따라 시작 명령이 xTimerStart()보다 먼저 처리됩니다. 이는 타이머 명령 대기열에 데이터가 저장되면 데몬 작업이 xTimerStart()를 호출한 작업을 선점하기 때문입니다.</li> <li>차단 시간이 지정된 경우에는(xTicksToWait가 0이 아닌 경우) 함수 반환 이전에 타이머 명령 대기열에서 사용할 수 있는 공간이 나올 때까지 호출 작업이 Blocked 상태로 전환되어 대기하지만 차단 시간이 지나기 전에 데이터가 타이머 명령 대기열에 성공적으로 작성되었을 가능성이 있습니다.</li> </ul> </li> <li>2. pdFALSE</li> </ol>

대기열이 이미 가득 차있어서 'start a timer' 명령이 타이머 명령 대기열에 작성되지 않은 경우 pdFALSE가 반환됩니다.

차단 시간이 지정된 경우에는(xTicksToWait가 0이 아닌 경우) 호출 작업이 Blocked 상태로 전환되어 데몬 작업이 타이머 명령 대기열에 공간을 만들 때까지 대기하지만 이전에 차단 시간이 먼저 지나버린 것입니다.

## 일회성 타이머와 오토 리로드 타이머의 생성(예제 13)

이번 예제에서는 일회성 타이머와 오토 리로드 타이머를 생성하여 시작합니다.

```
/* The periods assigned to the one-shot and auto-reload timers are 3.333 second and half a
second, respectively. */

#define mainONE_SHOT_TIMER_PERIOD pdMS_TO_TICKS( 3333 )

#define mainAUTO_RELOAD_TIMER_PERIOD pdMS_TO_TICKS( 500 )

int main( void )
{
    TimerHandle_t xAutoReloadTimer, xOneShotTimer;

    BaseType_t xTimer1Started, xTimer2Started;

    /* Create the one-shot timer, storing the handle to the created timer in xOneShotTimer.
    */

    xOneShotTimer = xTimerCreate( /* Text name for the software timer - not
used by FreeRTOS. */ "OneShot", /* The software timer's period, in ticks. */
mainONE_SHOT_TIMER_PERIOD, /* Setting uxAutoRealod to pdFALSE creates a one-shot software
timer. */ pdFALSE, /* This example does not use the timer ID. */ 0, /* The callback
function to be used by the software timer being created. */ prvOneShotTimerCallback );

    /* Create the auto-reload timer, storing the handle to the created timer in
xAutoReloadTimer. */

    xAutoReloadTimer = xTimerCreate( /* Text name for the software timer - not
used by FreeRTOS. */ "AutoReload", /* The software timer's period, in ticks. */
mainAUTO_RELOAD_TIMER_PERIOD, /* Setting uxAutoRealod to pdTRUE creates an auto-reload
timer. */ pdTRUE, /* This example does not use the timer ID. */ 0, /* The callback
function to be used by the software timer being created. */ prvAutoReloadTimerCallback );

    /* Check the software timers were created. */

    if( ( xOneShotTimer != NULL ) && ( xAutoReloadTimer != NULL ) )
    {

        /* Start the software timers, using a block time of 0 (no block time). The
scheduler has not been started yet so any block time specified here would be ignored
anyway. */

        xTimer1Started = xTimerStart( xOneShotTimer, 0 );

        xTimer2Started = xTimerStart( xAutoReloadTimer, 0 );
    }
}
```

```
/* The implementation of xTimerStart() uses the timer command queue, and
xTimerStart() will fail if the timer command queue gets full. The timer service task does
not get created until the scheduler is started, so all commands sent to the command queue
will stay in the queue until after the scheduler has been started. Check both calls to
xTimerStart() passed. */

if( ( xTimer1Started == pdPASS ) && ( xTimer2Started == pdPASS ) )
{
    /* Start the scheduler. */
    vTaskStartScheduler();
}

/* As always, this line should not be reached. */
for( ;; );
}
```

타이머의 콜백 함수는 각 타이머가 생성될 때마다 메시지를 출력합니다. 일회성 타이머 콜백 함수의 구현체는 다음과 같습니다.

```
static void prvOneShotTimerCallback( TimerHandle_t xTimer )
{
    TickType_t xTimeNow;

    /* Obtain the current tick count. */
    xTimeNow = xTaskGetTickCount();

    /* Output a string to show the time at which the callback was executed. */
    vPrintStringAndNumber( "One-shot timer callback executing", xTimeNow );

    /* File scope variable. */
    ulCallCount++;
}
```

오토 리로드 타이머 콜백 함수의 구현체는 다음과 같습니다.

```
static void prvAutoReloadTimerCallback( TimerHandle_t xTimer )
{
    TickType_t xTimeNow;

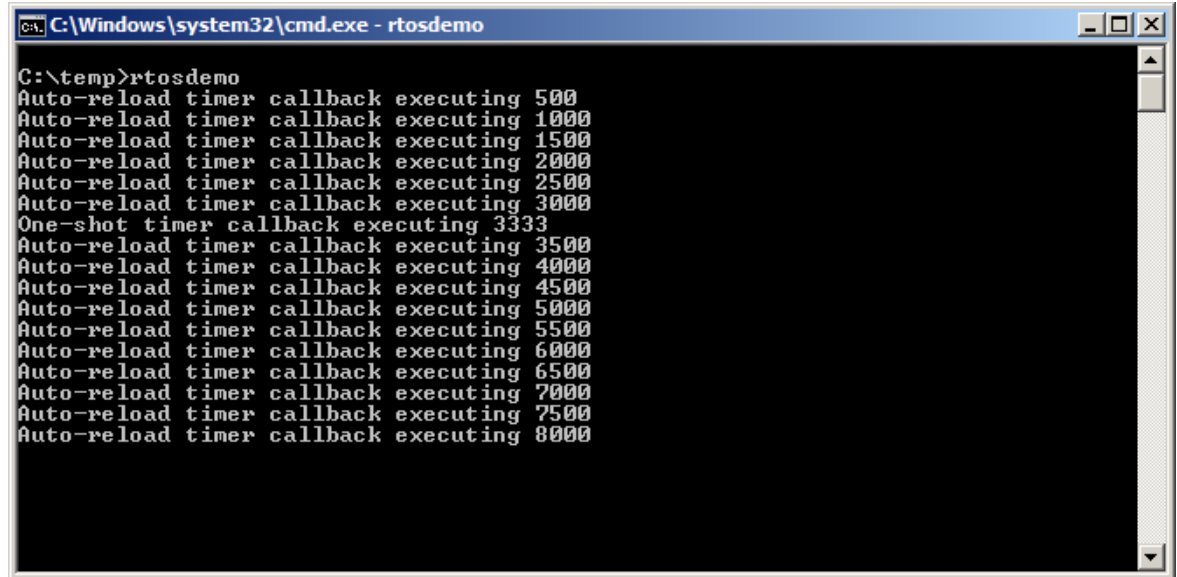
    /* Obtain the current tick count. */
    xTimeNow = xTaskGetTickCount();

    /* Output a string to show the time at which the callback was executed. */
    vPrintStringAndNumber( "Auto-reload timer callback executing", xTimeNow);
}
```



```
    ulCallCount++;  
}
```

위 예제를 실행하면 다음과 같은 화면이 출력됩니다. 출력 화면을 보면 500틱의 고정 주기 (mainAUTO\_RELOAD\_TIMER\_PERIOD가 500으로 설정됨)로 실행되는 오토 리로드 타이머의 콜백 함수와 틱 카운트가 3333에 도달할 때(mainONE\_SHOT\_TIMER\_PERIOD가 3333으로 설정됨) 한 번만 실행되는 일회성 타이머의 콜백 함수가 있습니다.



## 타이머 ID

소프트웨어 타이머는 각각 태그 값이 할당되는 ID가 있으며, 애플리케이션 개발자는 이 태그 값을 어떤 목적으로든 사용할 수 있습니다. ID가 보이드 포인터(void \*)로 저장되기 때문에 정수 값을 직접 저장하거나, 다른 객체를 가리키거나, 혹은 함수 포인터로 사용될 수도 있습니다.

소프트웨어 타이머를 생성할 때 초기 값이 ID에 할당됩니다. 이후 vTimerSetTimerID() API 함수를 사용해 ID를 업데이트하거나, 혹은 pvTimerGetTimerID() API 함수를 사용해 쿼리할 수 있습니다.

vTimerSetTimerID() 및 pvTimerGetTimerID() 함수는 다른 소프트웨어 타이머 API 함수와 달리 소프트웨어 타이머에 직접 액세스합니다. 따라서 명령을 타이머 명령 대기열에 전송하지 않습니다.

## vTimerSetTimerID() API 함수

vTimerSetTimerID() 함수 프로토타입은 다음과 같습니다.

```
void vTimerSetTimerID( const TimerHandle_t xTimer, void *pvNewID );
```

다음 표는 vTimerSetTimerID() 파라미터를 나열한 것입니다.

파라미터 이름/반환 값	설명
xTimer	새로운 ID 값으로 업데이트되는 소프트웨어 타이머의 핸들입니다. 이 핸들은 소프트웨어 타이머 생성을 위해 xTimerCreate()를 호출할 때 반환됩니다.

pvNewID

소프트웨어 타이머의 ID가 설정되는 값입니다.

## pvTimerGetTimerID() API 함수

pvTimerGetTimerID() 함수 프로토타입은 다음과 같습니다.

```
void *pvTimerGetTimerID( TimerHandle_t xTimer );
```

다음 표는 pvTimerGetTimerID() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
xTimer	쿼리하는 소프트웨어 타이머의 핸들입니다. 이 핸들은 소프트웨어 타이머 생성을 위해 xTimerCreate()를 호출할 때 반환됩니다.
반환 값	쿼리하는 소프트웨어 타이머의 ID입니다.

## 콜백 함수 파라미터와 소프트웨어 타이머 ID의 사용(예제 14)

동일한 콜백 함수를 다수의 소프트웨어 타이머에 할당할 수 있습니다. 이러한 경우에는 콜백 함수 파라미터가 만료된 소프트웨어 타이머를 확인하는 데 사용됩니다.

예제 13에서는 서로 다른 두 가지 콜백 함수를 사용했습니다. 하나는 일회성 타이머용이고, 나머지 하나는 오토 리로드 타이머용입니다. 또한 두 소프트웨어 타이머 모두에게 비슷한 기능의 단일 콜백 함수를 할당합니다.

다음 코드는 두 타이머 모두에게 동일한 콜백 함수인 prvTimerCallback()을 사용하는 방법을 나타낸 것입니다.

```
/* Create the one-shot software timer, storing the handle in xOneShotTimer. */
xOneShotTimer = xTimerCreate( "OneShot", mainONE_SHOT_TIMER_PERIOD, pdFALSE, /* The
timer's ID is initialized to 0. */ 0, /* prvTimerCallback() is used by both timers. */
prvTimerCallback );

/* Create the auto-reload software timer, storing the handle in xAutoReloadTimer */
xAutoReloadTimer = xTimerCreate( "AutoReload", mainAUTO_RELOAD_TIMER_PERIOD, pdTRUE, /* The
timer's ID is initialized to 0. */ 0, /* prvTimerCallback() is used by both timers. */
prvTimerCallback );
```

prvTimerCallback()은 두 타이머 중 하나가 만료되면 실행됩니다. prvTimerCallback() 구현체는 함수의 파라미터를 사용해 일회성 타이머 또는 오토 리로드 타이머의 만료에 따른 호출 여부를 확인합니다.

prvTimerCallback()은 소프트웨어 타이머 ID를 각 타이머의 저장 공간으로 사용하는 방법도 설명합니다. 각 소프트웨어 타이머는 자신이 만료된 횟수를 고유 ID에 저장합니다. 오토 리로드 타이머는 저장된 횟수 정보를 사용해 다섯 번째 실행에서 스스로 중단됩니다.

prvTimerCallback() 구현체는 다음과 같습니다.

```
static void prvTimerCallback( TimerHandle_t xTimer )
{
    TickType_t xTimeNow;

    uint32_t ulExecutionCount;

    /* A count of the number of times this software timer has expired is stored in the
    timer's ID. Obtain the ID, increment it, then save it as the new ID value. The ID is a
    void pointer, so is cast to a uint32_t. */

    ulExecutionCount = ( uint32_t ) pvTimerGetTimerID( xTimer );

    ulExecutionCount++;

    vTimerSetTimerID( xTimer, ( void * ) ulExecutionCount );

    /* Obtain the current tick count. */

    xTimeNow = xTaskGetTickCount();

    /* The handle of the one-shot timer was stored in xOneShotTimer when the timer was
    created. Compare the handle passed into this function with xOneShotTimer to determine if
    it was the one-shot or auto-reload timer that expired, then output a string to show the
    time at which the callback was executed. */

    if( xTimer == xOneShotTimer )
    {
        vPrintStringAndNumber( "One-shot timer callback executing", xTimeNow );
    }
    else
    {
        /* xTimer did not equal xOneShotTimer, so it must have been the auto-reload timer
        that expired. */

        vPrintStringAndNumber( "Auto-reload timer callback executing", xTimeNow );

        if( ulExecutionCount == 5 )
        {
            /* Stop the auto-reload timer after it has executed 5 times. This callback
            function executes in the context of the RTOS daemon task, so must not call any functions
            that might place the daemon task into the Blocked state. Therefore, a block time of 0 is
            used. */

            xTimerStop( xTimer, 0 );
        }
    }
}
```

출력되는 화면은 다음과 같습니다. 오토 리로드 타이머는 5회까지만 실행됩니다.

```
C:\Windows\system32\cmd.exe - RTOSDemo
C:\temp>RTOSDemo
Auto-reload timer callback executing 500
Auto-reload timer callback executing 1000
Auto-reload timer callback executing 1500
Auto-reload timer callback executing 2000
Auto-reload timer callback executing 2500
One-shot timer callback executing 3333
```

## 타이머의 주기 변경

모든 공식 FreeRTOS 포트는 프로젝트 예제가 1개 이상 제공됩니다. 대부분 프로젝트 예제는 자체 검사가 가능하여 LED가 프로젝트 상태에 대한 시각적 피드백을 제공합니다. 자체 검사가 항상 통과되었다면 LED가 서서히 깜박입니다. 하지만 차제 검사가 한 번이라도 통과되지 못했다면 LED가 빠르게 깜박입니다.

일부 프로젝트 예제는 작업에서 자체 검사를 실행하고 `vTaskDelay()` 함수를 사용해 LED의 깜박임 속도를 제어합니다. 그 밖에 소프트웨어 타이머 콜백 함수에서 자체 검사를 실행하고 타이머 주기를 사용해 LED의 깜박임 속도를 제어하는 프로젝트 예제도 있습니다.

## xTimerChangePeriod() API 함수

`xTimerChangePeriod()` 함수는 소프트웨어 타이머의 주기를 변경하는 데 사용됩니다.

`xTimerChangePeriod()`가 이미 실행 중인 타이머의 주기를 변경할 경우 타이머가 새로운 주기 값을 사용해 만료되는 시간을 다시 계산합니다. 여기에서 다시 계산되는 만료 시간은 타이머의 최초 시작 시간이 아니라 `xTimerChangePeriod()`가 호출된 시간과 관련이 있습니다.

`xTimerChangePeriod()`가 Dormant 상태의 타이머(실행되지 않고 있는 타이머) 주기를 변경할 경우 타이머가 만료 시간을 계산하여 Running 상태(타이머 실행이 시작되는 상태)로 전환됩니다.

참고: `xTimerChangePeriod()`를 인터럽트 서비스 루틴에서 호출하지 마십시오. 대신에 인터럽트 세이프 버전인 `xTimerChangePeriodFromISR()`을 사용하십시오.

`xTimerChangePeriod()` API 함수 프로토타입은 다음과 같습니다.

```
BaseType_t xTimerChangePeriod( TimerHandle_t xTimer, TickType_t xNewTimerPeriodInTicks,
                               TickType_t xTicksToWait );
```

다음 표는 `xTimerChangePeriod()` 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
--------------	----

xTimer	새로운 주기 값으로 업데이트되는 소프트웨어 타이머의 핸들입니다. 이 핸들은 소프트웨어 타이머 생성을 위해 xTimerCreate()를 호출할 때 반환됩니다.
xTimerPeriodInTicks	틱 단위로 새롭게 지정되는 소프트웨어 타이머 주기입니다. 밀리초 단위의 시간은 pdMS_TO_TICKS() 매크로를 사용해 틱으로 변환할 수 있습니다.
xTicksToWait	<p>xTimerChangePeriod()는 타이머 명령 대기열을 사용해 'change period' 명령을 데몬 작업에게 전송합니다. xTicksToWait는 대기열이 이미 가득 찬 경우 호출 작업이 타이머 명령 대기열에서 사용할 수 있는 공간이 나올 때까지 Blocked 상태로 대기해야 하는 최대 시간을 지정합니다.</p> <p>xTicksToWait가 0이고 타이머 명령 대기열이 이미 가득 찬 상태라면 xTimerChangePeriod()가 즉시 값을 반환합니다.</p> <p>밀리초 단위의 시간은 pdMS_TO_TICKS() 매크로를 사용해 틱으로 변환할 수 있습니다.</p> <p>INCLUDE_vTaskSuspend가 FreeRTOSConfig.h에서 1로 설정되었을 때 xTicksToWait를 portMAX_DELAY로 설정하면 호출 작업이 타이머 명령 대기열에서 사용할 수 있는 공간이 나올 때까지 Blocked 상태로 제한 시간 없이 무기한 대기합니다.</p> <p>스케줄러 시작 이전에 xTimerChangePeriod()가 호출되면 xTicksToWait 값이 무시되고 xTicksToWait의 값이 마치 0인 것처럼 xTimerChangePeriod()가 동작합니다.</p>

반환 값

가능한 반환 값은 다음 두 가지입니다.

1. pdPASS

데이터가 타이머 명령 대기열에 성공적으로 전송된 경우에 한해 pdPASS가 반환됩니다.

차단 시간이 지정된 경우에는(xTicksToWait가 0이 아닌 경우) 함수 반환 이전에 타이머 명령 대기열에서 사용할 수 있는 공간이 나올 때까지 호출 작업이 Blocked 상태로 전환되어 대기하지만 차단 시간이 지나기 전에 데이터가 타이머 명령 대기열에 성공적으로 작성되었을 가능성이 있습니다.

2. pdFALSE

대기열이 이미 가득 차있어서 'change period' 명령이 타이머 명령 대기열에 작성되지 않은 경우 pdFALSE가 반환됩니다.

차단 시간이 지정된 경우에는(xTicksToWait가 0이 아닌 경우) 호출 작업이 Blocked 상태로 전환되어 데몬 작업이 대기열에 공간을 만들 때까지 대기하지만 이전에 차단 시간이 먼저 지나버린 것입니다.

다음 코드는 자체 검사가 통과되지 않은 경우 소프트웨어 타이머 콜백 함수의 자체 검사 기능이 xTimerChangePeriod()를 사용해 LED의 깜박임 속도를 높일 수 있는 방법을 나타낸 것입니다. 자체 검사를 실행하는 소프트웨어 타이머를 검사 타이머라고 부릅니다.

```
/* The check timer is created with a period of 3000 milliseconds, resulting in the LED
   toggling every 3 seconds. If the self-checking functionality detects an unexpected state,
   then the check timer's period is changed to just 200 milliseconds, resulting in a much
   faster toggle rate. */

const TickType_t xHealthyTimerPeriod = pdMS_TO_TICKS( 3000 );

const TickType_t xErrorTimerPeriod = pdMS_TO_TICKS( 200 );

/* The callback function used by the check timer. */

static void prvCheckTimerCallbackFunction( TimerHandle_t xTimer )
{
    static BaseType_t xErrorDetected = pdFALSE;

    if( xErrorDetected == pdFALSE )
    {
        /* No errors have yet been detected. Run the self-checking function again.
           The function asks each task created by the example to report its own status, and also
           checks that all the tasks are actually still running (and so able to report their status
           correctly). */

        if( CheckTasksAreRunningWithoutError() == pdFAIL )
        {
```

```
/* One or more tasks reported an unexpected status. An error might have
occurred. Reduce the check timer's period to increase the rate at which this callback
function executes, and in so doing also increase the rate at which the LED is toggled.
This callback function is executing in the context of the RTOS daemon task, so a block
time of 0 is used to ensure the Daemon task never enters the Blocked state. */

xTimerChangePeriod( xTimer, /* The timer being updated. */ xErrorTimerPeriod, /
* The new period for the timer. */ 0 ); /* Do not block when sending this command. */

}

/* Latch that an error has already been detected. */

xErrorDetected = pdTRUE;

}

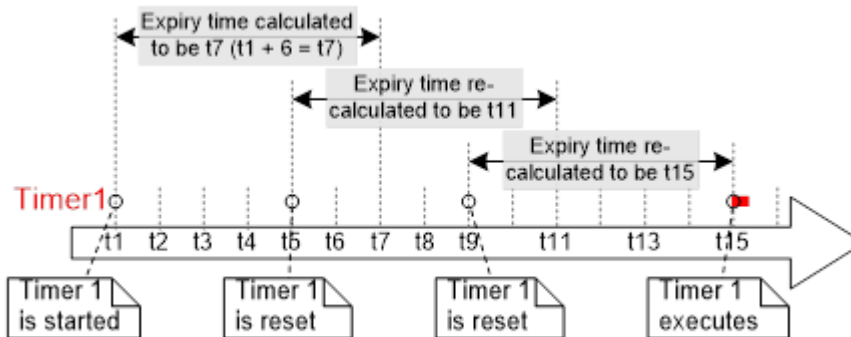
/* Toggle the LED. The rate at which the LED toggles will depend on how
often this function is called, which is determined by the period of the check
timer. The timer's period will have been reduced from 3000ms to just 200ms if
CheckTasksAreRunningWithoutError() has ever returned pdFAIL. */

ToggleLED();

}
```

## 소프트웨어 타이머의 재설정

소프트웨어 타이머의 재설정은 타이머의 재시작을 의미합니다. 이때 타이머의 만료 시간은 타이머의 최초 시작 시간이 아니라 타이머가 재설정된 시간을 기준으로 다시 계산됩니다. 다음 그림은 6틱 주기의 타이머가 시작되었다가 최종 만료되어 콜백 함수를 실행하기 전에 두 번 재설정되는 것을 나타냅니다.



- Timer 1이 t1에 시작됩니다. 6틱 주기이기 때문에 콜백 함수의 실행 시간은 처음에 시작 이후 6틱이 지나는 t7으로 계산됩니다.
- Timer 1이 t7에 이르기 전에, 즉 만료되어 콜백 함수를 실행하기 전에 재설정됩니다. 이에 따라 Timer 1은 t5에서 재설정되어 콜백 함수의 실행 시간이 재설정 이후 6틱이 지나는 t11으로 다시 계산됩니다.
- Timer 1이 t11에 이르기 전에, 즉 만료되어 콜백 함수를 실행하기 전에 다시 재설정됩니다. 이에 따라 Timer 1은 t9에서 재설정되어 콜백 함수의 실행 시간이 재설정 이후 6틱이 지나는 t15으로 다시 계산됩니다.
- Timer 1이 다시 재설정되지 않아 t15에서 만료되고, 그에 따라 콜백 함수가 실행됩니다.

## xTimerReset() API 함수

xTimerReset() API 함수는 타이머를 재설정하는 데 사용됩니다.

또한 xTimerReset()은 Dormant 상태의 타이머를 시작하는 데도 사용됩니다.

참고: xTimerReset()를 인터럽트 서비스 루틴에서 호출하지 마십시오. 대신에 인터럽트 세이프 버전인 xTimerResetFromISR()을 사용하십시오.

xTimerReset() API 함수 프로토타입은 다음과 같습니다.

```
BaseType_t xTimerReset( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

다음 표는 xTimerReset() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
xTimer	재설정 또는 시작되는 소프트웨어 타이머의 핸들입니다. 이 핸들은 소프트웨어 타이머 생성을 위해 xTimerCreate()를 호출할 때 반환됩니다.
xTicksToWait	<p>xTimerChangePeriod()는 타이머 명령 대기열을 사용해 'reset' 명령을 데몬 작업에게 전송합니다. xTicksToWait는 대기열이 이미 가득 찬 경우 호출 작업이 타이머 명령 대기열에서 사용할 수 있는 공간이 나올 때까지 Blocked 상태로 대기해야 하는 최대 시간을 지정합니다.</p> <p>xTicksToWait이 0이고 타이머 명령 대기열이 이미 가득 찬 상태라면 xTimerReset()이 즉시 값을 반환합니다.</p> <p>INCLUDE_vTaskSuspend가 FreeRTOSConfig.h에서 1로 설정되었을 때 xTicksToWait를 portMAX_DELAY로 설정하면 호출 작업이 타이머 명령 대기열에서 사용할 수 있는 공간이 나올 때까지 Blocked 상태로 제한 시간 없이 무기한 대기합니다.</p>
반환 값	<p>가능한 반환 값은 다음 두 가지입니다.</p> <ol style="list-style-type: none"> <li>pdPASS <ul style="list-style-type: none"> <li>데이터가 타이머 명령 대기열에 성공적으로 전송된 경우에 한해 pdPASS가 반환됩니다.</li> <li>차단 시간이 지정된 경우에는(xTicksToWait가 0이 아닌 경우) 함수 반환 이전에 타이머 명령 대기열에서 사용할 수 있는 공간이 나올 때까지 호출 작업이 Blocked 상태로 전환되어 대기하지만 차단 시간이 지나기 전에 데이터가 타이머 명령 대기열에 성공적으로 작성되었을 가능성이 있습니다.</li> </ul> </li> <li>pdFALSE <ul style="list-style-type: none"> <li>대기열이 이미 가득 차있어서 'reset' 명령이 타이머 명령 대기열에 작성되지 않은 경우 pdFALSE가 반환됩니다.</li> </ul> </li> </ol>



차단 시간이 지정된 경우에는(xTicksToWait가 0이 아닌 경우) 호출 작업이 Blocked 상태로 전환되어 데몬 작업이 대기열에 공간을 만들 때까지 대기하지만 이전에 차단 시간이 먼저 지나버린 것입니다.

## 소프트웨어 타이머의 재설정(예제 15)

이번 예제에서는 휴대 전화의 백라이트 동작을 시뮬레이션합니다. 백라이트:

- 키를 누르면 켜집니다.
- 일정 시간 내에 다른 키를 누르면 켜진 상태가 계속 유지됩니다.
- 일정 시간 내에 다른 키를 누르지 않으면 자동으로 꺼집니다.

이러한 동작을 구현하는 데 일회성 소프트웨어 타이머가 사용됩니다.

- 키를 누르면 (시뮬레이션되는) 백라이트가 켜지고, 소프트웨어 타이머의 콜백 함수에서 꺼집니다.
- 키를 누를 때마다 소프트웨어 타이머가 재설정됩니다.
- 따라서 백라이트가 꺼지지 않도록 키를 눌러야 하는 시간은 소프트웨어 타이머의 주기와 동일합니다. 타이머가 만료되기 전에 키를 눌러 소프트웨어 타이머를 재설정하지 않으면 타이머의 콜백 함수가 실행되고 백라이트가 꺼집니다.

xSimulatedBacklightOn 변수에는 백라이트 상태가 저장됩니다. xSimulatedBacklightOn이 pdTRUE로 설정되면 백라이트가 켜집니다. 반대로 xSimulatedBacklightOn이 pdFALSE로 설정되면 백라이트가 꺼집니다.

소프트웨어 타이머 콜백 함수는 다음과 같습니다.

```
static void prvBacklightTimerCallback( TimerHandle_t xTimer )
{
    TickType_t xTimeNow = xTaskGetTickCount();

    /* The backlight timer expired. Turn the backlight off. */
    xSimulatedBacklightOn = pdFALSE;

    /* Print the time at which the backlight was turned off. */
    vPrintStringAndNumber("Timer expired, turning backlight OFF at time\t\t", xTimeNow );
}
```

FreeRTOS를 사용하면 애플리케이션을 이벤트 중심으로 설계할 수 있습니다. 이러한 이벤트 중심 설계는 처리 시간을 효율적으로 사용합니다. 즉 이벤트가 발생한 경우에 한해 처리 시간이 사용됩니다. 발생하지 않은 이벤트를 폴링하느라 처리 시간을 낭비할 필요가 없습니다. FreeRTOS Windows 포트를 사용할 때는 키보드 인터럽트를 처리하는 것이 현실적이지 않기 때문에 이벤트 중심 예제를 사용하지 못합니다. 대신에 비교적 덜 효율적인 폴링 기법을 사용해야 했습니다.

다음 코드는 키보드를 폴링하는 작업을 나타낸 것입니다. 만약 인터럽트 서비스 루틴이라면 xTimerReset() 대신에 xTimerResetFromISR()을 사용해야 합니다.

```
static void vKeyHitTask( void *pvParameters )
```

```
{

    const TickType_t xShortDelay = pdMS_TO_TICKS( 50 );

    TickType_t xTimeNow;

    vPrintString( "Press a key to turn the backlight on.\r\n" );

    /* Ideally an application would be event-driven, and use an interrupt to process key
    presses. It is not practical to use keyboard interrupts when using the FreeRTOS Windows
    port, so this task is used to poll for a key press. */

    for( ;; )
    {
        /* Has a key been pressed? */

        if( _kbhit() != 0 )
        {
            /* A key has been pressed. Record the time. */

            xTimeNow = xTaskGetTickCount();

            if( xSimulatedBacklightOn == pdFALSE )
            {
                /* The backlight was off, so turn it on and print the time at which it was
                turned on. */

                xSimulatedBacklightOn = pdTRUE;

                vPrintStringAndNumber( "Key pressed, turning backlight ON at time\t\t",
                xTimeNow );
            }
            else
            {
                /* The backlight was already on, so print a message to say the timer is
                about to be reset and the time at which it was reset. */

                vPrintStringAndNumber( "Key pressed, resetting software timer at time\t\t",
                xTimeNow );
            }

            /* Reset the software timer. If the backlight was previously off, then this
            call will start the timer. If the backlight was previously on, then this call will restart
            the timer. A real application might read key presses in an interrupt. If this function
            was an interrupt service routine, then xTimerResetFromISR() must be used instead of
            xTimerReset(). */

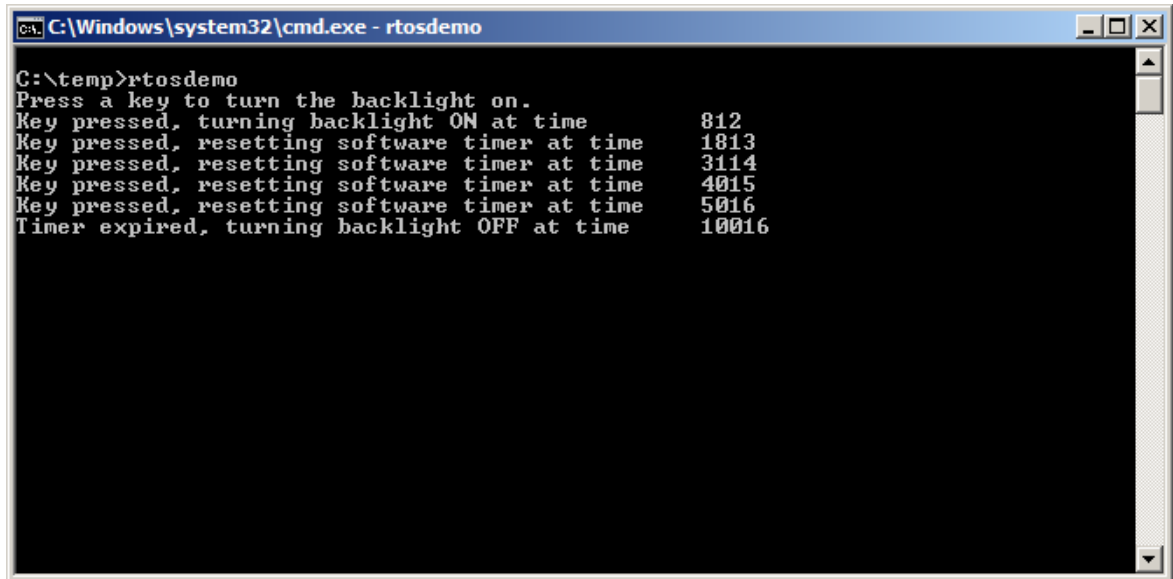
            xTimerReset( xBacklightTimer, xShortDelay );

            /* Read and discard the key that was pressed. It is not required by this simple
            example. */

            ( void ) _getch();
        }
    }
}
```

```
}  
  
}  
  
}
```

출력되는 화면은 다음과 같습니다.



```
C:\temp>rtosdemo  
Press a key to turn the backlight on.  
Key pressed, turning backlight ON at time      812  
Key pressed, resetting software timer at time  1813  
Key pressed, resetting software timer at time  3114  
Key pressed, resetting software timer at time  4015  
Key pressed, resetting software timer at time  5016  
Timer expired, turning backlight OFF at time  10016
```

- 틱 카운트가 812에 이르면 첫 번째 키 누름이 실행됩니다. 이때 백라이트가 켜지고 일회성 타이머가 시작됩니다.
- 틱 카운트가 1813, 3114, 4015, 5016에 이를 때 각각 키 누름이 발생합니다. 이 모든 키 누름이 실행되면 타이머가 만료 이전에 재설정됩니다.
- 틱 카운트가 10016에 이르면 타이머가 만료됩니다. 이때 백라이트가 꺼집니다.

타이머는 5000틱의 주기를 갖고 있습니다. 백라이트는 키를 마지막으로 누른 후 정확히 5000틱에서, 즉 타이머가 마지막으로 재설정된 후 5000틱에서 꺼집니다.

# 인터럽트 관리

이번 단원에서 다루는 내용은 다음과 같습니다.

- 인터럽트 서비스 루틴 내부에서 사용할 수 있는 FreeRTOS API 함수
- 인터럽트 처리를 작업에게 위임하는 방법
- 이진 세마포어와 계수 세마포어를 생성 및 사용하는 방법
- 이진 세마포어와 계수 세마포어의 차이점
- 대기열을 사용해 데이터를 인터럽트 서비스 루틴으로 전달하거나 인터럽트 서비스 루틴에서 전달하는 방법
- FreeRTOS 포트에서 사용할 수 있는 인터럽트 중첩 모델

임베디드 실시간 시스템은 환경에서 이벤트가 발생할 경우 필요한 작업을 수행해야 합니다. 예를 들어 이더넷 주변 장치로 패킷이 수신되면(이벤트) 처리를 위해 TCP/IP 스택으로 전달해야 합니다(작업). 중요한 시스템은 다양한 소스에서 발생하는 이벤트를 살펴봐야 합니다. 소스마다 처리 오버헤드와 응답 시간 요건이 다르기 때문입니다. 따라서 이벤트 처리 구현 전략을 세울 때는 다음과 같은 질문을 생각해봐야 합니다.

1. 이벤트를 어떻게 감지해야 하는가? 일반적으로 인터럽트가 사용되지만 입력 값을 폴링하는 방법도 있습니다.
2. 인터럽트를 사용한다면 인터럽트 서비스 루틴 내부와 외부에서 얼마나 많은 인터럽트를 처리해야 하는가? 인터럽트 서비스 루틴은 각각 최대한 짧게 작성하는 것이 좋습니다.
3. 이벤트가 메인(ISR이 아닌) 코드에게 어떻게 통신하는가, 그리고 메인 코드를 짜임새 있게 작성하여 비동기 방식으로 발생할 수 있는 이벤트를 최대한 처리하려면 어떻게 해야 하는가?

FreeRTOS는 이벤트 처리 전략을 애플리케이션 설계자에게 맡기지 않고 간단하면서 관리가 가능한 방식으로 선택한 전략을 구현할 수 있는 기능을 제공합니다.

다음과 같이 작업의 우선순위와 인터럽트의 우선순위를 구분해야 합니다.

- 작업은 FreeRTOS가 실행되는 하드웨어와 무관한 소프트웨어 기능입니다. 소프트웨어에서 작업의 우선순위는 애플리케이션 개발자가 할당합니다. 소프트웨어 알고리즘(스케줄러)에 따라 Running 상태로 전환할 작업이 결정됩니다.
- 인터럽트 서비스 루틴은 소프트웨어에서 작성되지만 하드웨어가 실행할 인터럽트 서비스 루틴과 실행 시점을 제어한다는 점에서 하드웨어 기능이라고 할 수 있습니다. 작업은 실행 중인 인터럽트 서비스 루틴이 없을 때만 실행되기 때문에 작업의 우선순위가 아무리 높다고 해도 가장 낮은 우선순위의 인터럽트로 인해 중단될 수 있습니다. 작업이 인터럽트 서비스 루틴보다 앞서 실행될 수 있는 방법은 없습니다.

FreeRTOS가 사용되는 아키텍처는 모두 인터럽트를 처리할 수 있지만 인터럽트 입력 및 인터럽트 우선순위 할당에 대한 세부 정보는 아키텍처마다 다릅니다.

## 인터럽트 세이프 API

FreeRTOS API 함수는 종종 인터럽트 서비스 루틴에서 제공하는 기능을 사용해야 할 경우가 있지만 대부분 FreeRTOS API 함수는 인터럽트 서비스 루틴 내부에서 유효하지 않은 작업을 수행합니다. 이러한 작업

들 중에서 가장 주목할 만한 것이 API 함수를 호출한 작업을 Blocked 상태로 전환하는 작업입니다. API 함수가 인터럽트 서비스 루틴에서 호출되면 작업에서는 호출되지 않기 때문에 Blocked 상태로 전환할 수 있는 호출 작업이 없습니다. FreeRTOS는 두 가지 버전의 API 함수, 즉 작업에서 사용하는 버전 1개와 인터럽트 서비스 루틴에서 사용하는 버전 1개를 제공하여 이러한 문제를 해결합니다. ISR에서 사용하는 함수는 이름에 "FromISR"이 첨부됩니다.

참고: ISR 내부에서는 이름에 "FromISR"이 없는 FreeRTOS API 함수를 호출하지 마십시오.

## 인터럽트 세이프 API를 별도로 사용하는 장점

인터럽트에서 API를 별도로 사용하면 작업과 코드가 더욱 효율적일 뿐만 아니라 인터럽트 입력이 더욱 간단합니다. 그 밖에 작업과 ISR 모두에서 호출할 수 있는 API 함수를 단일 버전으로 제공하는 방법을 생각해 보겠습니다. 동일한 버전의 API 함수를 작업과 인터럽트 서비스 루틴 모두에서 호출하는 경우 다음과 같은 결과를 예상할 수 있습니다.

- API 함수가 작업에서 호출되었는지, 혹은 ISR에서 호출되었는지 확인하려면 추가 로직이 필요합니다. 로직이 추가되면 함수에 이르는 경로가 새롭게 늘어나서 함수가 더욱 길어지고, 복잡해지고, 테스트하기 어려워집니다.
- 함수가 작업에서 호출되었을 때, 그리고 ISR에서 호출되었을 때 각각 사용하지 못하는 API 함수 파라미터가 있습니다.
- FreeRTOS 포트마다 실행 컨텍스트(작업 또는 ISR)를 결정하기 위한 메커니즘을 제공해야 합니다.
- 아키텍처에서 실행 컨텍스트(작업 또는 ISR)를 결정하기 쉽지 않을 때는 소프트웨어에서 실행 컨텍스트를 제공할 수 있도록 더욱 복잡한 비표준 인터럽트 입력 코드가 추가로 필요하지만 소모적일 수 있습니다.

## 인터럽트 세이프 API를 별도로 사용하는 단점

두 가지 버전의 API 함수를 사용하면 작업과 ISR이 더욱 효율적이지만 새로운 문제가 발생합니다. FreeRTOS API에 속하지 않지만 FreeRTOS API를 사용하는 함수를 작업과 ISR 모두에서 호출해야 할 경우도 있습니다.

이러한 경우에는 소프트웨어의 설계가 애플리케이션 개발자의 제어에서 유일하게 벗어나기 때문에 타사 코드를 통합할 때만 문제가 됩니다. 이때는 다음 기법 중 한 가지를 사용할 수 있습니다.

1. 인터럽트 처리를 작업에게 위임하십시오. 그러면 API 함수가 작업 컨텍스트에서만 호출됩니다.
2. 인터럽트 중첩을 지원하는 FreeRTOS 포트를 사용하고 있다면 "FromISR"로 끝나는 버전의 API 함수를 사용하십시오. 이 버전의 함수는 작업과 ISR에서 호출할 수 있기 때문입니다. (그 반대는 그렇지 않습니다. 즉 "FromISR"로 끝나지 않는 API 함수를 ISR에서 호출해서는 안 됩니다)
3. 타사 코드에는 일반적으로 함수가 호출되는 컨텍스트(작업 또는 인터럽트)를 테스트한 후 해당 컨텍스트에 적합한 API 함수를 호출할 수 있는 RTOS 추상화 계층이 포함되어 있습니다.

## xHigherPriorityTaskWoken 파라미터

인터럽트에서 컨텍스트 전환이 이루어지는 경우에는 인터럽트가 종료될 때 실행되는 작업과 인터럽트가 시작될 때 실행되었던 작업이 다를 수 있습니다. 인터럽트가 중단하는 작업과 복귀되는 작업이 다르기 때문입니다.

일부 FreeRTOS API 함수는 작업을 Blocked 상태에서 Ready 상태로 전환할 수 있습니다. 이미 xQueueSendToBack() 같은 함수에서도 봤지만 이러한 함수는 대상 대기열에서 데이터를 사용할 수 있을 때까지 Blocked 상태로 대기하는 작업이 있을 경우 작업의 차단을 해제합니다.

FreeRTOS API 함수를 통해 차단 해제된 작업의 우선순위가 Running 상태인 작업의 우선순위보다 높으면 FreeRTOS 스케줄링 정책에 따라 더욱 높은 우선순위의 작업으로 전환되어야 합니다. 실제로 더욱 높은 우선순위의 작업으로 전환되는 시점은 API 함수가 호출되는 컨텍스트에 따라 다릅니다.

- API 함수가 작업에서 호출된 경우:

configUSE\_PREEMPTION이 FreeRTOSConfig.h서 1로 설정되면 API 함수 내에서 자동으로, 즉 API 함수가 종료되기 전에 더욱 높은 우선순위의 작업으로 자동 전환됩니다. 이러한 경우는 타이머 명령 대기열에 대한 쓰기 연산이 있을 경우 명령 대기열에 작성된 함수가 종료되기 전에 RTOS 데몬으로 전환되었던 software\_tier\_management에서도 있었습니다.

- API 함수가 인터럽트에서 호출된 경우:

인터럽트 내부에서는 더욱 높은 우선순위의 작업으로 자동 전환이 일어나지 않습니다. 대신에 애플리케이션 개발자에게 컨텍스트의 전환 필요성을 알릴 수 있는 변수가 설정됩니다. "FromISR"로 끝나는 인터럽트 세이프 API 함수는 pxHigherPriorityTaskWoken이라고 하는 포인터 파라미터를 이러한 목적으로 사용합니다.

컨텍스트 전환이 필요하면 인터럽트 세이프 API 함수가 \*pxHigherPriorityTaskWoken을 pdTRUE로 설정합니다. pxHigherPriorityTaskWoken에서 가리켰던 변수는 컨텍스트 전환 여부를 알 수 있도록 첫 번째 사용 이전에 pdFALSE로 초기화되어야 합니다.

애플리케이션 개발자가 ISR의 컨텍스트 전환을 요청하지 않도록 선택한 경우에는 스케줄러가 다음에 실행될 때까지(최악의 경우 다음 틱 인터럽트가 됨) 더욱 높은 우선순위의 작업이 Ready 상태로 남습니다.

FreeRTOS API 함수는 \*pxHighPriorityTaskWoken을 pdTRUE로만 설정할 수 있습니다. ISR이 FreeRTOS API 함수를 2개 이상 호출하는 경우에는 각 API 함수 호출마다 동일한 변수를 pxHigherPriorityTaskWoken 파라미터 형태로 전달할 수 있습니다. 이렇게 전달된 변수는 첫 번째 사용 이전에 pdFALSE로 초기화되어야 합니다.

인터럽트 세이프 버전의 API 함수 내부에서는 컨텍스트 전환이 자동으로 일어나지 않는 몇 가지 이유가 있습니다.

#### 1. 불필요한 컨텍스트 전환 방지

인터럽트는 작업에서 처리를 수행할 때까지 여러 차례 실행될 수 있습니다. 예를 들어 작업이 인터럽트 방식 UART에서 수신된 문자열을 처리하는 시나리오를 가정하겠습니다. 문자 1개가 수신될 때마다 UART ISR이 작업으로 전환되는 것은 매우 소모적입니다. 작업은 전체 문자열이 수신된 후에만 처리를 수행하기 때문입니다.

#### 2. 실행 시퀀스에 대한 제어

인터럽트는 산발적으로, 예측하지 못한 때 발생할 수 있습니다. FreeRTOS 전문 사용자들도 애플리케이션의 특정 지점에서 다른 작업으로 갑자기 전환되는 것을 일시적으로 피하려고 합니다. FreeRTOS 스케줄러 잠금 메커니즘을 사용하면 가능합니다.

#### 3. 이동성

모든 FreeRTOS 포트에서 사용하려면 메커니즘이 최대한 단순해야 합니다.

#### 4. 효율성

더욱 작은 프로세서 아키텍처를 대상으로 하는 포트에서만 ISR 끝에서 컨텍스트 전환을 요청할 수 있습니다. 이러한 제한을 제거하려면 더욱 복잡한 코드가 추가되어야 합니다. 또한 동일한 ISR 내부에서는 컨텍스트 전환 요청을 2회 이상 생성하지 않고도 FreeRTOS API 함수를 2회 이상 호출할 수 있습니다.

#### 5. RTOS 틱 인터럽트의 실행

애플리케이션 코드를 RTOS 틱 인터럽트에 추가할 수 있습니다. 틱 인터럽트 내부에서 컨텍스트 전환을 시도하면 사용 중인 FreeRTOS 포트에 따라 결과가 달라집니다. 기껏해야 불필요한 스케줄러 호출로 이어질 뿐입니다.

pxHigherPriorityTaskWoken 파라미터의 사용은 선택 사항입니다. 따라서 필요하지 않다면 pxHigherPriorityTaskWoken을 NULL로 설정하십시오.

## portYIELD\_FROM\_ISR() 및 portEND\_SWITCHING\_ISR() 매크로

taskYIELD()는 작업에서 컨텍스트 전환을 요청할 때 호출할 수 있는 매크로입니다. portYIELD\_FROM\_ISR()과 portEND\_SWITCHING\_ISR()은 모두 taskYIELD()의 인터럽트 세이프 버전으로 사용 방식이나 기능이 동일합니다. 지금까지 portEND\_SWITCHING\_ISR()은 FreeRTOS 포트에서 인터럽트 핸들러에게 어셈블리 코드 래퍼 사용을 요구할 때 사용했던 이름이었고, portYIELD\_FROM\_ISR()은 FreeRTOS 포트에서 전체 인터럽트 핸들러를 C로 작성하도록 허용할 때 사용했던 이름입니다. 일부 FreeRTOS 포트는 두 매크로 중 하나만 제공하지만 최신 FreeRTOS 포트일수록 두 매크로를 모두 제공합니다.

```
portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
```

```
portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
```

인터럽트 세이프 API 함수에서 전달된 xHigherPriorityTaskWoken 파라미터는 portYIELD\_FROM\_ISR() 호출 시 직접 파라미터로 사용할 수 있습니다.

portYIELD\_FROM\_ISR() xHigherPriorityTaskWoken 파라미터가 pdFALSE(0)으로 설정되면 컨텍스트 전환이 필요 없어 매크로가 아무런 효과도 없습니다. portYIELD\_FROM\_ISR() xHigherPriorityTaskWoken 파라미터가 pdFALSE이 아니면 컨텍스트 전환이 필요하여 Running 상태의 작업이 바뀔 수도 있습니다. 인터럽트 실행으로 Running 상태의 작업이 바뀌었다더라도 인터럽트는 항상 Running 상태의 작업으로 복귀됩니다.

대부분 FreeRTOS 포트는 ISR 내부 어디에서든지 portYIELD\_FROM\_ISR()의 호출을 허용합니다. 일부 FreeRTOS 포트(주로 작은 크기의 아키텍처에 사용되는 포트)이지만 ISR 끝에서만 portYIELD\_FROM\_ISR()의 호출을 허용하는 경우도 있습니다.

## 인터럽트 처리의 위임

ISR은 다음과 같은 이유로 최대한 짧게 작성하는 것이 가장 좋습니다.

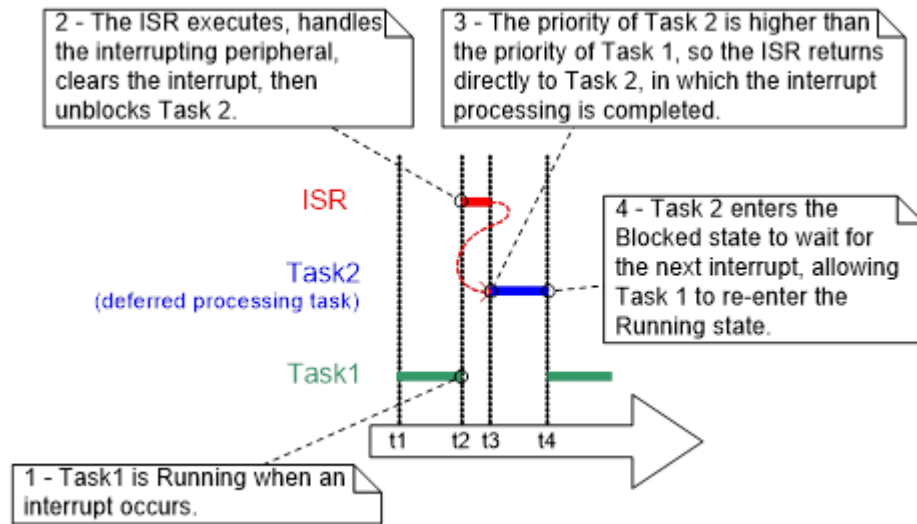
- 작업에게 매우 높은 우선순위가 할당되었다더라도 하드웨어에서 인터럽트가 발생하지 않아야만 실행됩니다.
- ISR은 지터를 추가하여 작업의 시작 시간과 실행 시간을 방해할 수 있습니다.
- FreeRTOS가 실행되는 아키텍처에 따라 ISR이 실행되는 동안 새로운 인터럽트를, 혹은 적어도 새로운 인터럽트의 하위 집합을 허용하지 못할 수도 있습니다.
- 애플리케이션 개발자는 변수와 주변 장치, 그리고 작업과 ISR이 동시에 액세스하는 메모리 버퍼 등 리소스 결과를 살피고 주의해야 합니다.
- 일부 FreeRTOS 포트는 인터럽트 중첩을 허용하지만 인터럽트 중첩은 복잡성을 높여 예측 가능성을 떨어뜨릴 수 있습니다. 인터럽트가 짧을수록 중첩이 일어날 가능성도 낮습니다.

인터럽트 서비스 루틴은 인터럽트 원인을 기록하고 인터럽트를 소거해야 합니다. 인터럽트에서 요구하는 다른 처리까지 작업에서 수행하는 경우가 많아지면 인터럽트 서비스 루틴이 빠르게 종료될 수 있습니다. 인터럽트에서 요구하는 처리가 ISR에서 작업으로 위임되었기 때문에 이것을 인터럽트 처리의 위임이라고 부릅니다.

또한 인터럽트 처리를 작업으로 위임하면 애플리케이션 개발자가 애플리케이션의 다른 작업과 비교하여 처리 우선순위를 결정하고 모든 FreeRTOS API 함수를 사용할 수 있습니다.

인터럽트 처리가 위임되는 작업의 우선순위가 다른 작업의 우선순위보다 높으면 마치 ISR 자체에서 하는 것처럼 인터럽트가 즉시 처리됩니다.

다음 그림은 Task 1이 정규 애플리케이션 작업이고, Task 2가 인터럽트 처리가 위임된 작업인 시나리오를 나타낸 것입니다.



이 그림에서 인터럽트 처리는 t2에서 시작되어 t4에서 실질적으로 종료되지만 ISR에서는 t2부터 t3까지 시간만 소모됩니다. 만약 인터럽트 처리의 위임을 사용하지 않았다면 ISR에서도 t2부터 t4까지 전체 시간이 소모되었을 것입니다.

인터럽트에서 요구하는 모든 처리를 ISR에서 언제 처리하는 것이 가장 좋은지, 혹은 처리의 일부를 언제 작업에 위임하는 것이 가장 좋은지 알려주는 절대적인 규칙은 존재하지 않습니다. 처리를 작업에 위임하는 것은 다음과 같은 경우에 가장 유용합니다.

- 인터럽트에서 요구하는 처리가 중요한 경우입니다. 예를 들어 인터럽트가 아날로그-디지털 변환 결과를 저장하는 작업일 뿐이라면 ISR에서 처리하는 것이 가장 좋습니다. 하지만 변환 결과를 소프트웨어 필터를 통해 전달해야 한다면 작업에서 필터를 실행하는 것이 바람직할 수도 있습니다.
- 인터럽트 처리는 콘솔에 작성하거나 메모리를 할당하는 등 ISR 내부에서 어려운 작업을 수행할 수 있다는 점에서 매우 편리합니다.
- 인터럽트 처리는 결정적이지 않습니다. 즉 처리하는 데 걸리는 시간을 미리 알 수 없습니다.

다음 단원에서는 인터럽트 처리의 위임을 구현할 때 사용할 수 있는 FreeRTOS 기능에 대해서 설명하고 알아보도록 하겠습니다.

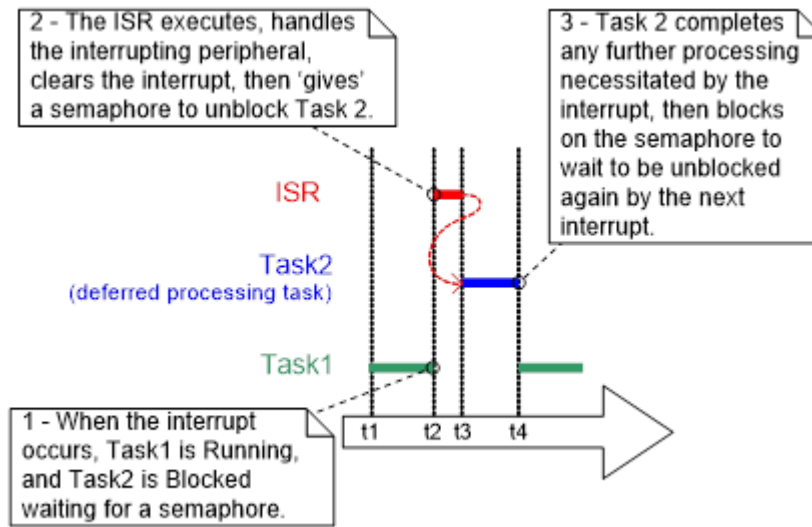
## 동기화에 사용되는 이진 세마포어

특정 인터럽트가 발생할 때마다 이진 세마포어 API의 인터럽트 세이프 버전을 사용해 작업의 차단을 해제하여 작업과 인터럽트를 효과적으로 동기화할 수 있습니다. 이를 통해 인터럽트 이벤트 처리의 대부분을 동기화된 작업 내에서 구현할 수 있을 뿐만 아니라 매우 빠르고 짧은 부분만 ISR에 남게 됩니다. 인터럽트 처리를 작업에 위임할 때는 이진 세마포어를 사용합니다. 하지만 인터럽트에서 작업의 차단을 해제할 때는 이진 세마포어를 사용하는 것보다 작업 직접 알림을 사용하는 것이 더욱 효율적입니다. 작업 직접 알림에 대한 자세한 내용은 [작업 알림 \(p. 191\)](#) 단원을 참조하십시오.

인터럽트 처리가 특정 시간에 중요한 경우에는 위임된 작업이 시스템의 다른 작업보다 앞서 실행될 수 있도록 우선순위를 설정할 수 있습니다. 그런 다음 ISR을 구현하면서 `portYIELD_FROM_ISR()` 호출을 추가하여 ISR이 인터럽트 처리가 위임되는 작업으로 직접 복귀되도록 할 수 있습니다. 이렇게 하면 마치 ISR 내부 자체에서 구현된 것처럼 전체 이벤트 처리가 시간적 끊어지지 않고 연속해서 실행되는 효과가 있습니다.

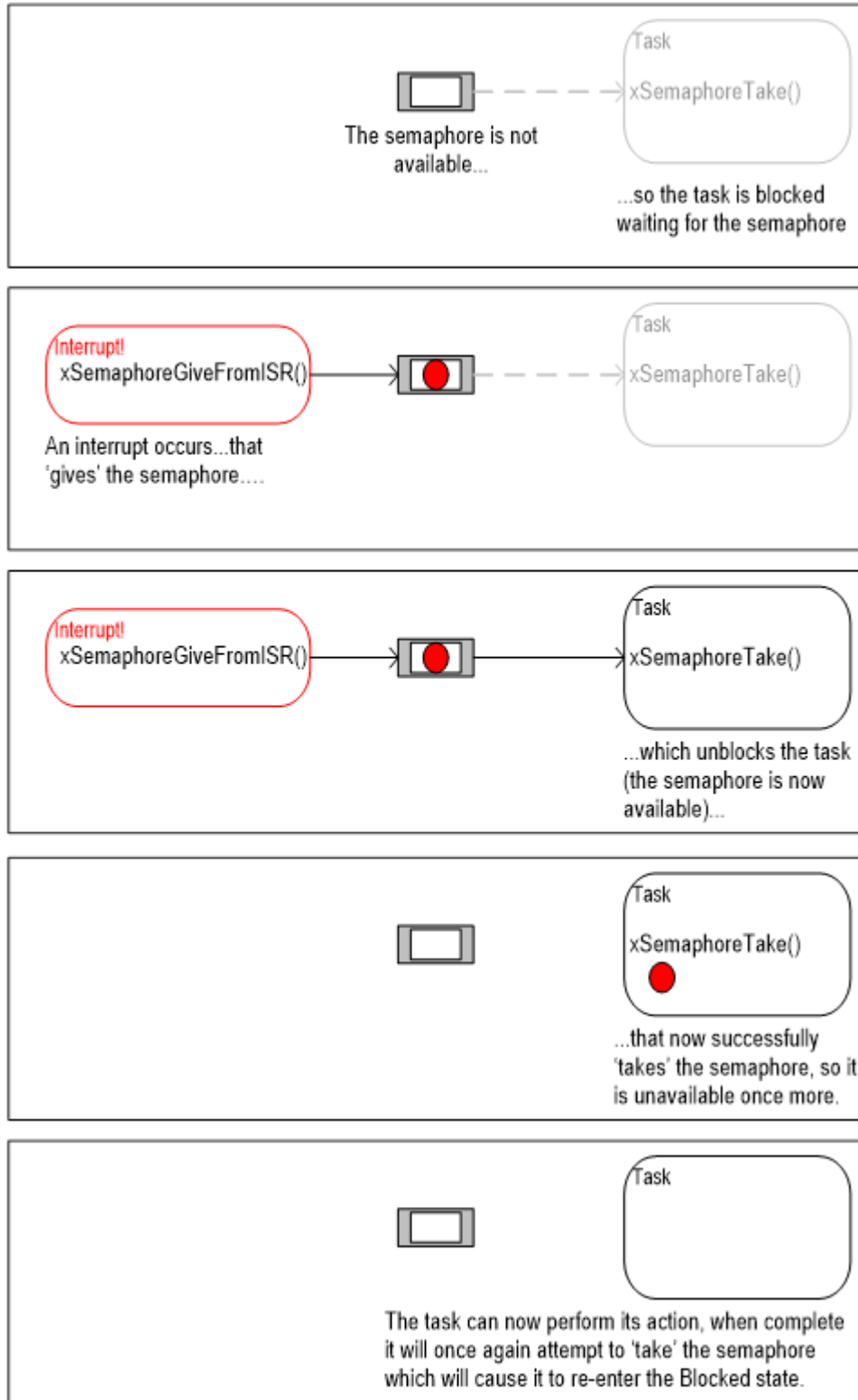
다음 그림은 이전 그림과 동일한 시나리오를 따르지만 세마포어를 사용해 위임된 처리 작업의 실행을 제어할 수 있는 방법을 설명한 텍스트가 추가된 것입니다.





위임된 처리 작업은 Blocked 상태로 전환하여 이벤트가 발생할 때까지 대기하기 위한 수단으로 세마포어 '가져오기(take)' 호출을 사용해 차단됩니다. 이후 이벤트가 발생하면 ISR이 동일한 세마포어에 대해 '반환(give)' 연산을 사용해 필요한 이벤트 처리가 이루어질 수 있도록 작업의 차단을 해제합니다.

세마포어 가져오기와 세마포어 반환은 시나리오에 따라 의미가 달라지는 개념입니다. 이번 인터럽트 동기화 시나리오에서는 개념적으로 봤을 때 이진 세마포어를 길이가 1인 대기열로 간주할 수 있습니다. 대기열에는 언제든지 최대 1개까지 항목이 포함될 수 있으므로 항상 비어있거나 가득 찬 상태가 됩니다(이진). 인터럽트 처리가 효과적으로 위임되는 작업은 `xSemaphoreTake()`를 호출하여 대기열에서 차단 시간과 함께 읽어오려는 시도를 하고, 이때 대기열이 비어있으면 작업이 Blocked 상태로 전환됩니다. 이후 이벤트가 발생하면 ISR이 `xSemaphoreGiveFromISR()` 함수를 사용해 토큰(세마포어)을 대기열에 배치하여 대기열이 가득 차게 됩니다. 이를 통해 작업은 Blocked 상태를 종료하고 토큰을 제거함으로써 대기열이 한 번 더 비워집니다. 이후 작업이 처리를 완료하면 대기열에서 한 번 더 읽어오려는 시도를 하지만 대기열이 비어있는 것을 알고 Blocked 상태로 다시 전환되어 다음 이벤트가 발생할 때까지 대기합니다. 아래 그림은 이러한 시퀀스를 나타낸 것입니다.



이 그림은 처음 가져온 것은 아니지만 세마포어를 반환하는 인터럽트와 절대로 반환하지 않지만 세마포어를 가져오는 작업을 나타낸 것입니다. 바로 이 점이 개념적으로 봤을 때 시나리오가 대기열에 대한 쓰기/읽기 연산과 비슷하다고 언급한 이유입니다. 이러한 시나리오는 [리소스 관리](#) (p. 152)에서 설명한 것처럼 작업이

세마포어를 가져온 후에는 반드시 반환해야 하는 다른 세마포어 사용 시나리오와 규칙이 다르다 보니 종종 혼동을 야기하기도 합니다.

## xSemaphoreCreateBinary() API 함수

FreeRTOS V9.0.0에도 컴파일 과정에서 이진 세마포어를 정적으로 생성하는 데 필요한 메모리를 할당하는 xSemaphoreCreateBinaryStatic() 함수가 포함되어 있습니다. 여러 가지 유형의 FreeRTOS 세마포어 핸들은 SemaphoreHandle\_t 형식의 변수로 저장됩니다.

세마포어를 사용하려면 먼저 생성해야 합니다. 이진 세마포어를 생성할 때는 xSemaphoreCreateBinary() API 함수가 사용됩니다.

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

다음 표는 xSemaphoreCreateBinary() 반환 값을 나열한 것입니다.

파라미터 이름	설명
반환 값	<p>NULL을 반환하는 경우에는 세마포어를 생성할 수 없습니다. FreeRTOS에서 세마포어 데이터 구조를 할당하는 데 필요한 힙 메모리가 부족하기 때문입니다.</p> <p>NULL이 아닌 값을 반환하는 경우에는 세마포어가 성공적으로 생성된 것입니다. 반환 값은 생성된 세마포어에 대한 핸들로 저장되어야 합니다.</p>

## xSemaphoreTake() API 함수

세마포어를 가져온다는 것은 세마포어를 얻거나 받는다는 것을 의미합니다. 세마포어는 사용 가능한 경우에 만 가져올 수 있습니다.

재귀적 뮤텝스를 제외한 모든 유형의 FreeRTOS 세마포어는 xSemaphoreTake() 함수를 사용해 가져올 수 있습니다. 단, xSemaphoreTake() 함수를 인터럽트 서비스 루틴에서 사용해서는 안 됩니다.

```
BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait );
```

다음 표는 xSemaphoreTake() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
xSemaphore	<p>가져올 세마포어입니다.</p> <p>세마포어는 SemaphoreHandle_t 형식의 변수로 참조됩니다. 세마포어를 사용하려면 먼저 명시적으로 생성해야 합니다.</p>
xTicksToWait	<p>아직 사용할 수 없는 경우 세마포어를 사용할 수 있을 때까지 작업이 Blocked 상태로 대기해야 하는 최대 시간입니다.</p> <p>xTicksToWait가 0이면 세마포어를 사용할 수 없더라도 xSemaphoreTake()가 즉시 값을 반환합니다.</p>

	<p>차단 시간은 틱 주기로 지정되기 때문에 틱 주파수에 따라 절대 시간이 달라집니다. 밀리초 단위의 시간은 pdMS_TO_TICKS() 매크로를 사용해 틱으로 변환할 수 있습니다.</p> <p>INCLUDE_vTaskSuspend가 FreeRTOSConfig.h에서 1로 설정된 경우 xTicksToWait를 portMAX_DELAY로 설정하면 작업이 제한 시간 없이 무기한 대기하게 됩니다.</p>
반환 값	<p>가능한 반환 값은 다음 두 가지입니다.</p> <ol style="list-style-type: none"> <li>1. pdPASS <p>xSemaphoreTake()를 호출하여 세마포어를 가져오는 데 성공한 경우에 한해 pdPASS가 반환됩니다.</p> <p>차단 시간이 지정된 경우에는(xTicksToWait가 0이 아닌 경우) 세마포어를 바로 사용할 수 없어서 사용할 수 있을 때까지 호출 작업이 Blocked 상태로 전환되어 대기하지만 차단 시간이 지나기 전에 세마포어를 사용할 수 있게 되었을 가능성이 있습니다.</p> </li> <li>2. pdFALSE <p>세마포어를 사용할 수 없습니다.</p> <p>차단 시간이 지정된 경우에는(xTicksToWait가 0이 아닌 경우) 호출 작업이 Blocked 상태로 전환되어 세마포어를 사용할 수 있을 때까지 대기하지만 이전에 차단 시간이 먼저 지나버린 것입니다.</p> </li> </ol>

## xSemaphoreGiveFromISR() API 함수

이진 및 계수 세마포어는 xSemaphoreGiveFromISR() 함수를 사용해 반환할 수 있습니다.

xSemaphoreGiveFromISR()은 xSemaphoreGive()의 인터럽트 세이프 버전이기 때문에 pxHigherPriorityTaskWoken 파라미터가 있습니다.

```
BaseType_t xSemaphoreGiveFromISR( SemaphoreHandle_t xSemaphore, BaseType_t *pxHigherPriorityTaskWoken );
```

다음은 xSemaphoreGiveFromISR() 파라미터와 반환 값을 나열한 것입니다.

**xSemaphore**

반환할 세마포어입니다. 세마포어는 SemaphoreHandle\_t 형식의 변수로 참조되며, 사용 전에 명시적으로 생성되어야 합니다.

**pxHigherPriorityTaskWoken**

단일 세마포어는 세마포어를 사용할 수 있을 때까지 차단된 상태로 대기하는 작업을 1개 이상 가질 수 있습니다. xSemaphoreGiveFromISR()을 호출하면 세마포어를 사용할 수 있게 되어 세마포어를 사용할 수 있을 때까지 대기하던 작업이 Blocked 상태를 종료하게 됩니다. xSemaphoreGiveFromISR()을 호출하여 작업의 Blocked 상태가 종료되었을 때 차단 해제된 작업의 우선순위가 현재 실행 중인 작업(중단된 작업)보다 높으면 내부적으로 xSemaphoreGiveFromISR()이 \*pxHigherPriorityTaskWoken을 pdTRUE로 설정합니다. xSemaphoreGiveFromISR()이 이 값을 pdTRUE로 설정하면 인터럽트 종료 전에 컨텍스트 전환이 정상적으

로 이루어져야 합니다. 그러면 최고 우선순위의 Ready 상태 작업으로 인터럽트 복귀가 직접 이루어질 수 있습니다.

가능한 반환 값은 다음 두 가지입니다.

pdPASS

xSemaphoreGiveFromISR() 호출에 성공한 경우에 한해 반환됩니다.

pdFAIL

세마포어가 이미 사용 가능한 경우에는 반환을 할 수 없어서 xSemaphoreGiveFromISR()이 pdFAIL을 반환합니다.

## 이진 세마포어를 사용한 작업과 인터럽트의 동기화 (예제 16)

이번 예제에서는 이진 세마포어를 사용해 작업을 인터럽트 서비스 루틴에서 차단 해제하여 작업과 인터럽트를 효과적으로 동기화합니다.

또한 단순한 주기적 작업을 사용해 500밀리초마다 소프트웨어 인터럽트를 생성합니다. 소프트웨어 인터럽트를 사용하는 이유는 일부 대상 환경에서 실제 인터럽트 후킹 복잡성과 비교하여 편리하기 때문입니다.

다음 코드는 주기적 작업의 구현체를 나타낸 것입니다. 작업은 인터럽트 생성 이전과 이후에 문자열을 출력합니다. 실행 시퀀스는 출력에서 확인할 수 있습니다.

```
/* The number of the software interrupt used in this example. The code shown is from the
   Windows project, where numbers 0 to 2 are used by the FreeRTOS Windows port itself, so 3
   is the first number available to the application. */

#define mainINTERRUPT_NUMBER 3

static void vPeriodicTask( void *pvParameters )
{
    const TickType_t xDelay500ms = pdMS_TO_TICKS( 500UL );

    /* As per most tasks, this task is implemented within an infinite loop.*/
    for( ;; )
    {
        /* Block until it is time to generate the software interrupt again. */
        vTaskDelay( xDelay500ms );

        /* Generate the interrupt, printing a message both before and after the interrupt
           has been generated, so the sequence of execution is evident from the output. The syntax
           used to generate a software interrupt is dependent on the FreeRTOS port being used.
           The syntax used below can only be used with the FreeRTOS Windows port, in which such
           interrupts are only simulated.*/

        vPrintString( "Periodic task - About to generate an interrupt.\r\n" );

        vPortGenerateSimulatedInterrupt( mainINTERRUPT_NUMBER );

        vPrintString( "Periodic task - Interrupt generated.\r\n\r\n\r\n" );
    }
};
```

```
}  
  
}
```

다음 코드는 인터럽트 처리가 위임된 작업과, 이진 세마포어를 사용해 소프트웨어 인터럽트와 동기화되는 작업의 구현체를 나타낸 것입니다. 다시 말하지만 문자열은 작업이 반복될 때마다 출력됩니다. 작업과 인터럽트의 실행 시퀀스는 출력에서 확인할 수 있습니다.

인터럽트가 소프트웨어에서 생성되는 경우에는 이 코드가 샘플로 적합하지만 인터럽트가 하드웨어 주변 장치에서 생성되는 시나리오에서는 부적합합니다. 인터럽트가 하드웨어에서 생성되는 경우에는 사용에 적합하도록 코드 구조가 바뀌어야 합니다.

```
static void vHandlerTask( void *pvParameters )  
{  
  
    /* As per most tasks, this task is implemented within an infinite loop.*/  
  
    for( ;; )  
    {  
  
        /* Use the semaphore to wait for the event. The semaphore was created before  
        the scheduler was started, so before this task ran for the first time. The task blocks  
        indefinitely, meaning this function call will only return once the semaphore has been  
        successfully obtained so there is no need to check the value returned by xSemaphoreTake().  
        */  
  
        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );  
  
        /* To get here the event must have occurred. Process the event (in this case, just  
        print out a message). */  
  
        vPrintString( "Handler task - Processing event.\r\n" );  
  
    }  
}
```

다음 코드는 ISR을 나타낸 것입니다. 여기에서는 세마포어를 반환하여 인터럽트 처리가 위임된 작업을 차단 해제하는 것 외에 달리 할 것이 없습니다.

사용되는 변수는 xHigherPriorityTaskWoken입니다. 이 변수는 xSemaphoreGiveFromISR() 호출 이전에 pdFALSE로 설정되며, 이후 portYIELD\_FROM\_ISR()을 호출할 때 파라미터로 사용됩니다. xHigherPriorityTaskWoken이 pdTRUE이면 portYIELD\_FROM\_ISR() 매크로 내부에서 컨텍스트 전환이 요청됩니다.

컨텍스트 전환을 위해 호출되는 ISR과 매크로의 프로토타입은 FreeRTOS Windows 포트의 경우 정확하지만, 그 밖의 FreeRTOS 포트에서는 다를 수도 있습니다. 사용 중인 포트에 필요한 구문을 찾으려면 FreeRTOS.org 웹사이트의 포트별 설명서 페이지와 FreeRTOS 다운로드에서 제공되는 예제를 참조하십시오.

FreeRTOS가 실행되는 대부분 아키텍처와 달리 FreeRTOS Windows 포트는 값을 반환하기 위해 ISR이 필요합니다. Windows 포트와 함께 제공되는 portYIELD\_FROM\_ISR() 매크로 구현체에는 반환 문이 포함되어 있기 때문에 이번 코드에서는 명시적으로 반환되는 값을 표시하지 않았습니다.

```
static uint32_t ulExampleInterruptHandler( void )  
{  
  
    BaseType_t xHigherPriorityTaskWoken;
```

```
/* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE because it
will get set to pdTRUE inside the interrupt-safe API function if a context switch is
required. */

xHigherPriorityTaskWoken = pdFALSE;

/* Give the semaphore to unblock the task, passing in the address of
xHigherPriorityTaskWoken as the interrupt-safe API function's pxHigherPriorityTaskWoken
parameter. */

xSemaphoreGiveFromISR( xBinarySemaphore, &xHigherPriorityTaskWoken );

/* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR(). If
xHigherPriorityTaskWoken was set to pdTRUE inside xSemaphoreGiveFromISR(), then calling
portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still
pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. Unlike most FreeRTOS
ports, the Windows port requires the ISR to return a value. The return statement is inside
the Windows version of portYIELD_FROM_ISR(). */

portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

main() 함수는 이진 세마포어와 작업을 생성하고, 인터럽트 핸들러를 설치한 후 스케줄러를 시작합니다. 다음 코드는 구현체를 나타낸 것입니다.

인터럽트 핸들러를 설치하기 위해 호출되는 함수 구문은 FreeRTOS Windows 포트 전용입니다. 그 밖의 FreeRTOS 포트일 때는 구문이 다를 수 있습니다. 사용 중인 포트에 필요한 구문을 찾으려면 FreeRTOS.org 웹사이트의 포트별 설명서와 FreeRTOS 다운로드에서 제공되는 예제를 참조하십시오.

```
int main( void )
{
    /* Before a semaphore is used, it must be explicitly created. In this example, a binary
    semaphore is created. */

    xBinarySemaphore = xSemaphoreCreateBinary();

    /* Check that the semaphore was created successfully. */

    if( xBinarySemaphore != NULL )
    {
        /* Create the 'handler' task, which is the task to which interrupt processing is
        deferred. This is the task that will be synchronized with the interrupt. The handler task
        is created with a high priority to ensure it runs immediately after the interrupt exits.
        In this case, a priority of 3 is chosen. */

        xTaskCreate( vHandlerTask, "Handler", 1000, NULL, 3, NULL );

        /* Create the task that will periodically generate a software interrupt. This is
        created with a priority below the handler task to ensure it will get preempted each time
        the handler task exits the Blocked state. */

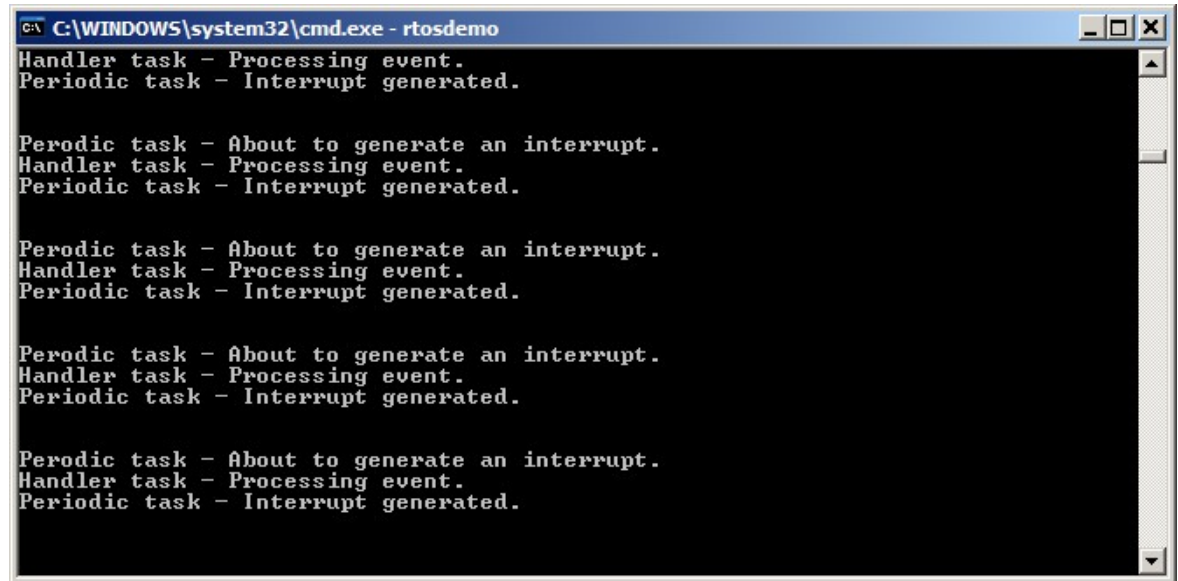
        xTaskCreate( vPeriodicTask, "Periodic", 1000, NULL, 1, NULL );

        /* Install the handler for the software interrupt. The syntax required to do this
        is depends on the FreeRTOS port being used. The syntax shown here can only be used with
        the FreeRTOS Windows port, where such interrupts are only simulated. */

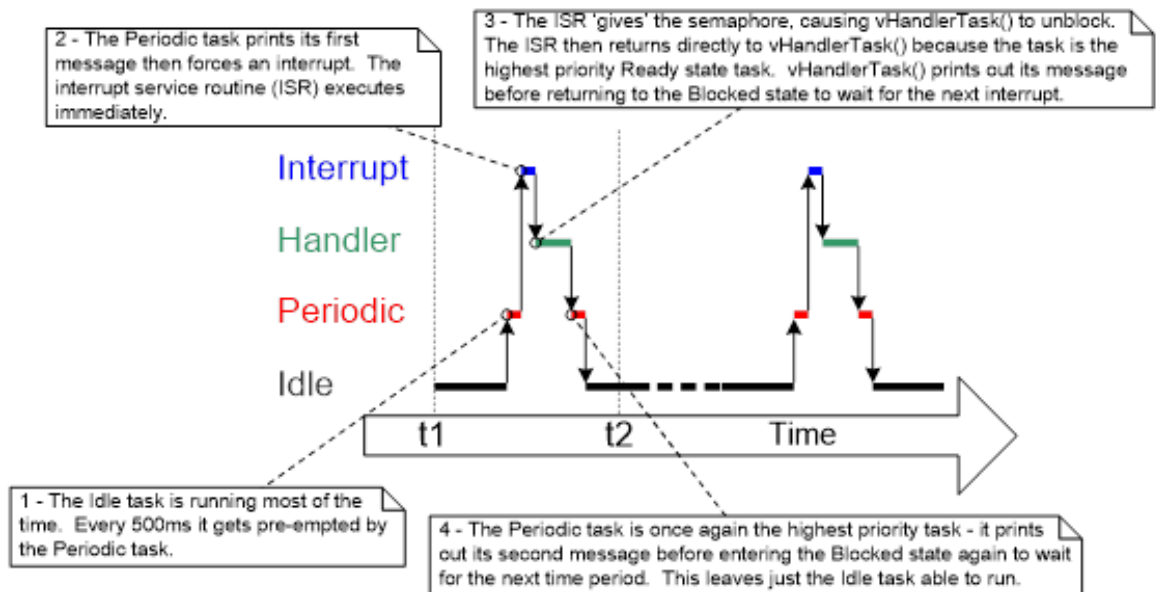
        vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulExampleInterruptHandler );
    }
}
```

```
/* Start the scheduler so the created tasks start executing. */  
vTaskStartScheduler();  
  
}  
  
/* As normal, the following line should never be reached. */  
for( ;; );  
  
}
```

코드 출력은 다음과 같습니다. 예상한 것처럼 인터럽트 생성과 함께 vHandlerTask()가 Running 상태로 전환 되기 때문에 주기적 작업의 출력을 분할하여 작업이 출력됩니다.



실행 시퀀스는 다음과 같습니다.





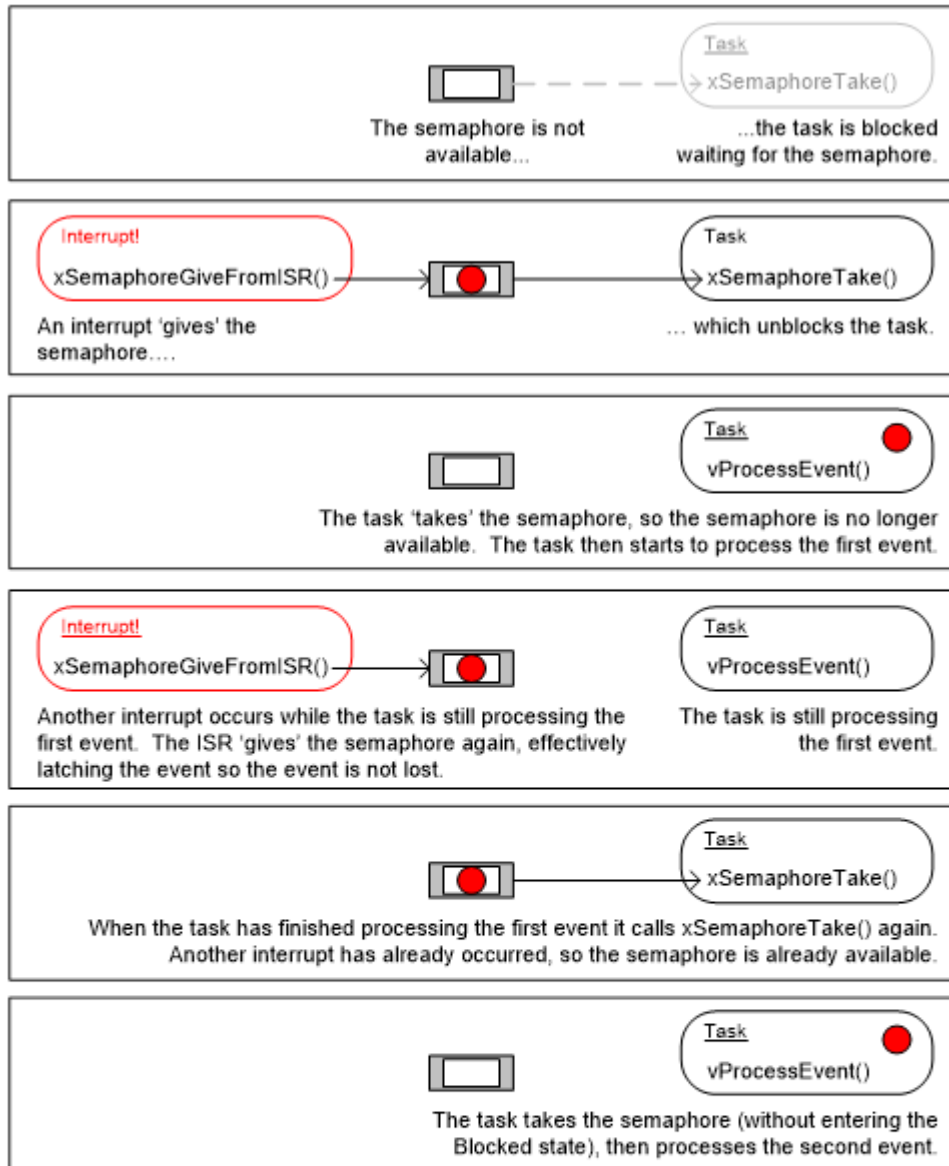
## 예제 16에서 사용되는 작업 구현체 개선

예제 16은 이진 세마포어를 사용해 작업을 인터럽트와 동기화합니다. 실행 시퀀스는 다음과 같습니다.

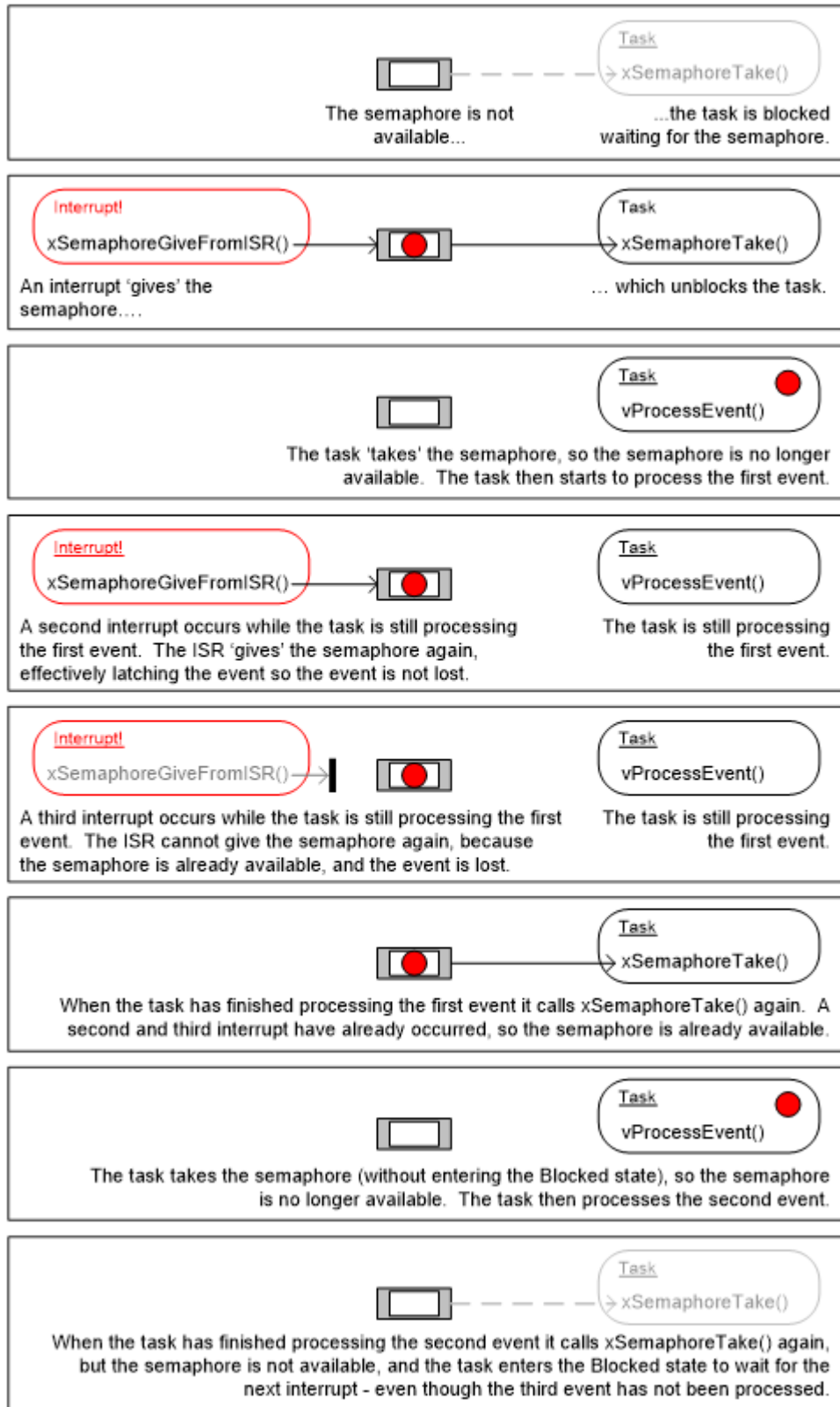
1. 인터럽트가 발생합니다.
2. ISR이 실행되어 세마포어를 반환하고 작업을 차단 해제합니다.
3. 작업이 ISR 직후 실행되면서 세마포어를 가져옵니다.
4. 작업이 이벤트를 처리한 후 세마포어를 다시 가져오려고 하지만 아직 세마포어를 사용할 수 없기 때문에 (다른 인터럽트가 발생하지 않았기 때문에) Blocked 상태로 전환됩니다.

예제 16에서 사용되는 작업 구조는 인터럽트가 비교적 낮은 주파수에서 발생한 경우에만 적합합니다. 작업이 첫 번째 인터럽트의 처리를 완료하기 전에 두 번째와 세 번째 인터럽트가 발생한다면 어떻게 될까요?

두 번째 ISR이 실행되어도 세마포어가 비어있기 때문에 ISR이 세마포어를 반환하며, 작업은 첫 번째 이벤트의 처리를 마치자마자 바로 두 번째 이벤트를 처리합니다. 시나리오는 다음과 같습니다.



세 번째 ISR이 실행될 때는 세마포어가 이미 사용할 수 있는 상태이기 때문에 ISR이 세마포어를 다시 반환하지 않습니다. 따라서 작업은 세 번째 이벤트가 발생한 것을 인지하지 못합니다. 시나리오는 다음과 같습니다.



위임된 인터럽트를 처리하는 작업인 `static void vHandlerTask( void *pvParameters )`는 `xSemaphoreTake()`를 호출할 때마다 이벤트를 하나만 처리하는 구조로 작성됩니다. 예제 16이 이러한 구조에 적합했던 이유는 이벤트를 생성한 인터럽트가 소프트웨어에서 트리거되어 예측 가능한 시간에 발생했기 때문입니다. 하지만 실제 애플리케이션에서는 인터럽트가 하드웨어에서 생성되어 발생하는 시간을 예측할 수 없습니다. 따라서 위임된 인터럽트를 처리하는 작업은 인터럽트가 누락될 가능성을 최소화하기 위해 `xSemaphoreTake()`를 호출할 때마다 이미 사용 가능한 상태의 이벤트를 모두 처리하는 구조로 작성되어야 합니다. 다음 코드는 위임된 UART 인터럽트 핸들러가 어떠한 구조로 작성되는지 나타낸 것입니다. 이 코드는 문자가 수신될 때마다 UART가 수신 인터럽트를 생성하고, 수신된 문자를 하드웨어 FIFO(하드웨어 버퍼)로 보낸다고 가정합니다.

예제 16에서 위임된 인터럽트를 처리하는 작업은 또 한 가지 단점이 있습니다. `xSemaphoreTake()`를 호출했을 때 제한 시간을 사용하지 않고 `portMAX_DELAY`를 `xSemaphoreTake()` `xTicksToWait` 파라미터로 전달했습니다. 그 결과 작업은 세마포어를 사용할 수 있을 때까지 제한 시간 없이 무기한 대기하게 됩니다. 무기한 제한 시간은 예제의 구조를 간소화하여 더욱 쉽게 이해하는 데 도움이 되기 때문에 예제 코드에서 종종 사용됩니다. 하지만 실제 애플리케이션에서는 일반적으로 오류를 복구하는 데 어려움이 있어서 바람직하지 않습니다. 인터럽트에서 세마포어가 반환될 때까지 작업이 대기하고 있지만 하드웨어 오류로 인해 인터럽트가 생성되지 않는 시나리오를 가정하겠습니다.

- 작업이 제한 시간 없이 대기하고 있다면 오류 상태를 인지하지 못해 끊임없이 대기하게 됩니다.
- 작업이 제한 시간과 함께 대기하고 있다면 제한 시간이 지났을 때 `xSemaphoreTake()`가 `pdFAIL`을 반환하기 때문에 작업이 오류를 감지하고 다음 실행에서 오류를 소거할 수 있습니다. 이러한 시나리오를 설명한 예제는 다음과 같습니다.

```
static void vUARTReceiveHandlerTask( void *pvParameters )
{
    /* xMaxExpectedBlockTime holds the maximum time expected between two interrupts. */
    const TickType_t xMaxExpectedBlockTime = pdMS_TO_TICKS( 500 );

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* The semaphore is given by the UART's receive (Rx) interrupt. Wait a maximum of
        xMaxExpectedBlockTime ticks for the next interrupt. */

        if( xSemaphoreTake( xBinarySemaphore, xMaxExpectedBlockTime ) == pdPASS)
        {
            /* The semaphore was obtained. Process ALL pending Rx events before calling
            xSemaphoreTake() again. Each Rx event will have placed a character in the UART's receive
            FIFO, and UART_RxCount() is assumed to return the number of characters in the FIFO. */

            while( UART_RxCount() > 0 )
            {
                /* UART_ProcessNextRxEvent() is assumed to process one Rx character,
                reducing the number of characters in the FIFO by 1. */

                UART_ProcessNextRxEvent();
            }

            /* No more Rx events are pending (there are no more characters in the FIFO),
            so loop back and call xSemaphoreTake() to wait for the next interrupt. Any interrupts
```

```
occurring between this point in the code and the call to xSemaphoreTake() will be latched
in the semaphore, so will not be lost. */

    }

    else

    {

        /* An event was not received within the expected time. Check for and, if
        necessary, clear any error conditions in the UART that might be preventing the UART from
        generating any more interrupts. */

        UART_ClearErrors();

    }

}

}
```

## 계수 세마포어

이진 세마포어를 길이가 1인 대기열로 간주할 수 있듯이 계수 세마포어는 길이가 1이 넘는 대기열로 간주할 수 있습니다. 작업은 대기열의 항목 수만 인식할 뿐 대기열에 저장되는 데이터는 알지 못합니다. 계수 세마포어를 사용하려면 FreeRTOSConfig.h에서 configUSE\_COUNTING\_SEMAPHORES를 1로 설정하십시오.

계수 세마포어가 반환될 때마다 대기열에서는 공간이 1개 더 추가로 사용됩니다. 대기열의 항목 수는 세마포어의 '계수(count)' 값이 됩니다.

일반적으로 계수 세마포어가 사용되는 경우는 다음과 같습니다.

### 1. 이벤트 계수

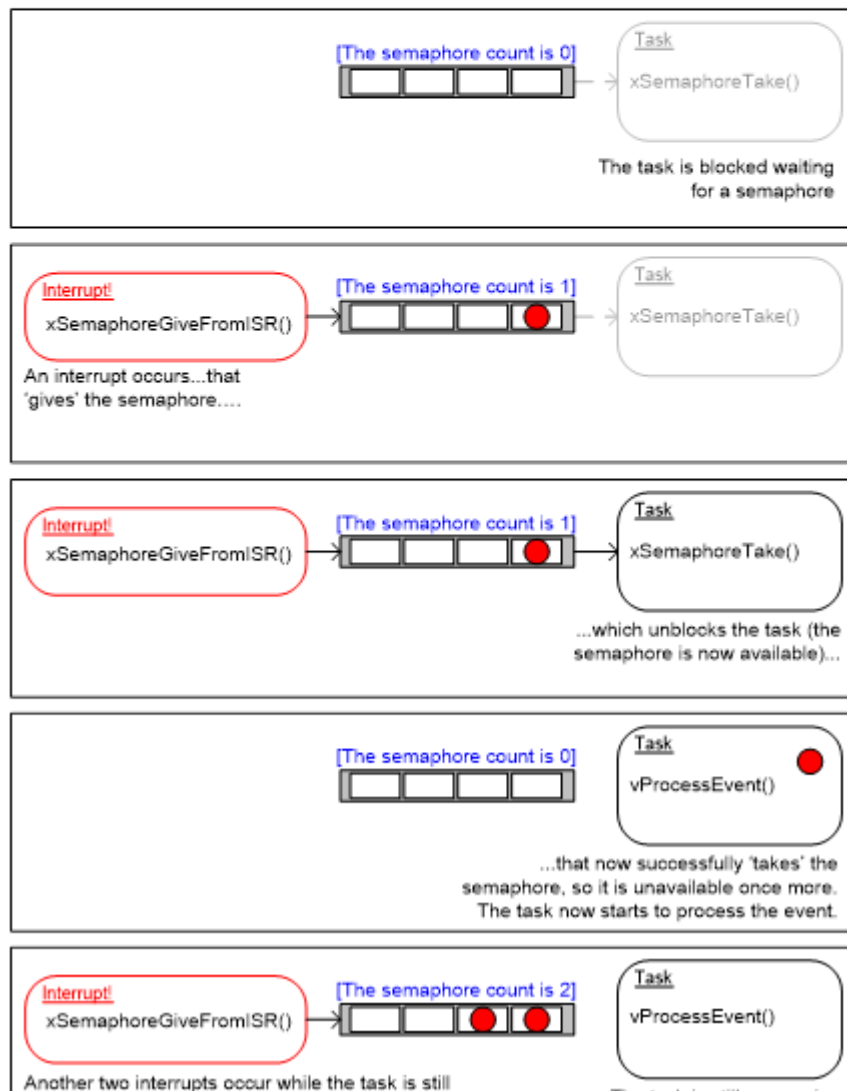
이번 시나리오에서는 이벤트가 발생할 때마다 이벤트 핸들러가 세마포어를 반환하고, 이렇게 반환할 때마다 세마포어의 계수 값은 증가합니다. 작업은 이벤트를 처리할 때마다 세마포어를 가져오고, 이렇게 가져올 때마다 세마포어의 계수 값은 감소합니다. 계수 값은 발생한 이벤트 수와 처리된 이벤트 수의 차이입니다. 이러한 메커니즘은 다음 그림에서 볼 수 있습니다.

이벤트 계수에 사용되는 계수 세마포어는 초기 계수 값이 0으로 생성됩니다.

### 2. 리소스 관리

이번 시나리오에서는 계수 값이 사용 가능한 리소스 수를 가리킵니다. 작업이 리소스를 제어하려면 먼저 세마포어를 가져와야 하며, 이로 인해 세마포어의 계수 값이 감소하게 됩니다. 계수 값이 0에 도달하면 여유 리소스가 없다는 것을 의미합니다. 작업이 리소스 제어를 마치면 세마포어를 반환하고, 이에 따라 세마포어의 계수 값이 증가합니다.

리소스 관리에 사용되는 계수 세마포어는 초기 계수 값이 사용 가능한 리소스 수와 동일하게 생성됩니다.



## xSemaphoreCreateCounting() API 함수

FreeRTOS V9.0.0에도 컴파일 과정에서 계수 세마포어를 정적으로 생성하는 데 필요한 메모리를 할당하는 xSemaphoreCreateCountingStatic() 함수가 포함되어 있습니다. 여러 가지 유형의 모든 FreeRTOS 세마포어 핸들은 SemaphoreHandle\_t 형식의 변수로 저장됩니다.

세마포어를 사용하려면 먼저 생성해야 합니다. 계수 세마포어를 생성할 때는 xSemaphoreCreateCounting() API 함수가 사용됩니다.

다음 표는 xSemaphoreCreateCounting() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
uxMaxCount	<p>세마포어의 최대 계수 값입니다. 대기열을 계속해서 유추하려면 uxMaxCount 값이 실제로 대기열의 길이가 되어야 합니다.</p> <p>세마포어가 이벤트를 계산하거나 잠그기 위해 사용될 때는 uxMaxCount가 잠글 수 있는 이벤트의 최대 수를 가리킵니다.</p> <p>세마포어가 리소스에 대한 액세스 관리에 사용될 때는 uxMaxCount가 사용 가능한 총 리소스 수로 설정되어야 합니다.</p>
uxInitialCount	<p>세마포어가 생성되었을 때 초기 계수 값입니다.</p> <p>세마포어가 이벤트를 계산하거나 잠그기 위해 사용될 때는 uxInitialCount가 0으로 설정되어야 합니다. 세마포어의 생성 시점에는 아직 발생한 이벤트가 없을 가능성이 높기 때문입니다.</p> <p>세마포어가 리소스에 대한 액세스 관리에 사용될 때는 uxInitialCount가 uxMaxCount와 동일하게 설정되어야 합니다. 세마포어의 생성 시점에는 모든 리소스가 사용 가능한 상태일 가능성이 높기 때문입니다.</p>
반환 값	<p>NULL을 반환하는 경우에는 세마포어를 생성할 수 없습니다. FreeRTOS에서 세마포어 데이터 구조를 할당하는 데 필요한 힙 메모리가 부족하기 때문입니다. 자세한 내용은 <a href="#">힙 메모리 관리 (p. 13)</a> 단원을 참조하십시오.</p> <p>NULL이 아닌 값을 반환하는 경우에는 세마포어가 성공적으로 생성된 것입니다. 반환 값은 생성된 세마포어에 대한 핸들로 저장되어야 합니다.</p>

## 계수 세마포어를 사용한 작업과 인터럽트의 동기화 (예제 17)

예제 17은 이진 세마포어 대신에 계수 세마포어를 사용하여 예제 16 구현체를 변경한 것입니다. main()은 xSemaphoreCreateBinary() 호출 대신에 xSemaphoreCreateCounting() 호출이 추가되면서 바뀌었습니다.

다음 코드는 새로운 API 호출을 나타낸 것입니다.

```
/* Before a semaphore is used it must be explicitly created. In this example, a counting semaphore is created. The semaphore is created to have a maximum count value of 10, and an initial count value of 0. */  
  
xCountingSemaphore = xSemaphoreCreateCounting( 10, 0 );
```

인터럽트 서비스 루틴은 높은 주기로 발생하는 다수의 이벤트를 시뮬레이션하기 위해 인터럽트 1회당 세마포어를 한 번 이상 반환하도록 바뀌었습니다. 각 이벤트는 세마포어의 계수 값으로 잠깁니다.

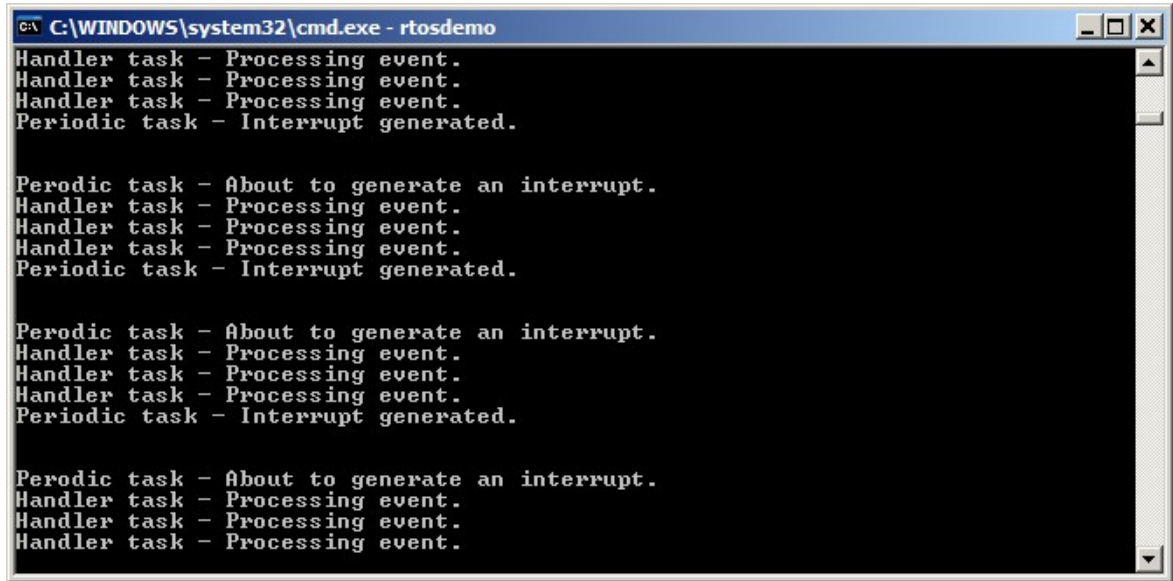
다음 코드는 수정된 인터럽트 서비스 루틴을 나타낸 것입니다.

```
static uint32_t ulExampleInterruptHandler( void )  
{  
  
    BaseType_t xHigherPriorityTaskWoken;  
  
    /* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE because it will get set to pdTRUE inside the interrupt-safe API function if a context switch is required. */  
  
    xHigherPriorityTaskWoken = pdFALSE;  
  
    /* Give the semaphore multiple times. The first will unblock the deferred interrupt handling task. The following gives are to demonstrate that the semaphore latches the events to allow the task to which interrupts are deferred to process them in turn, without events getting lost. This simulates multiple interrupts being received by the processor, even though in this case the events are simulated within a single interrupt occurrence. */  
  
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );  
  
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );  
  
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );  
  
    /* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR(). If xHigherPriorityTaskWoken was set to pdTRUE inside xSemaphoreGiveFromISR(), then calling portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. Unlike most FreeRTOS ports, the Windows port requires the ISR to return a value. The return statement is inside the Windows version of portYIELD_FROM_ISR(). */  
  
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );  
  
}
```

나머지 함수는 모두 예제 16과 동일하게 사용됩니다.

다음은 코드를 실행한 출력 화면입니다. 인터럽트 생성 시 인터럽트 처리가 위임된 작업이 (시뮬레이션) 이벤트 3개를 모두 처리하고 있는 모습을 볼 수 있습니다. 여기에서 작업이 이벤트를 차례대로 처리할 수 있는 이유는 각 이벤트가 세마포어의 계수 값으로 잠기기 때문입니다.





## 작업을 RTOS 데몬 작업으로 위임

지금까지 소개한 인터럽트 처리의 위임 예제에서는 애플리케이션 개발자가 처리 위임 기법을 사용하는 인터럽트마다 작업을 생성해야 했습니다. 하지만 `xTimerPendFunctionCallFromISR()` API 함수를 사용해 인터럽트 처리를 RTOS 데몬 작업에게 위임하는 것도 가능합니다. 그러면 인터럽트마다 작업을 따로 생성할 필요가 없습니다. 인터럽트 처리를 데몬 작업에게 위임하는 것을 인터럽트 처리의 중앙 집중식 위임이라고 부릅니다.

데몬 작업은 소프트웨어 타이머 콜백 함수를 실행할 목적으로만 사용되어 처음에는 타이머 서비스 작업라고 불렸습니다. `xTimerPendFunctionCall()`이 `timers.c`에서 구현되는 이유도 바로 여기에 있으며, 그 밖에도 함수가 구현되는 파일의 이름을 따서 함수의 이름에 접두사를 첨부하는 규칙에 따라 함수의 이름에 첨부되는 접두사도 'Timer'입니다.

`software_timer_management` 단원에서 소프트웨어 타이머와 관련된 FreeRTOS API 함수가 타이머 명령 대기열에서 명령을 데몬 작업에게 어떻게 보내는지 설명했습니다. `xTimerPendFunctionCall()` 및 `xTimerPendFunctionCallFromISR()` API 함수도 동일한 타이머 명령 대기열을 사용해 'execute function' 명령을 데몬 작업에게 보냅니다. 이렇게 데몬 작업에게 전송된 함수는 데몬 작업의 컨텍스트에서 실행됩니다.

인터럽트 처리의 중앙 집중식 위임의 장점

- 낮은 리소스 사용량

위임되는 인터럽트마다 작업을 따로 생성할 필요가 없습니다.

- 단순한 사용자 모델

위임된 인터럽트를 처리하는 함수는 표준 C 함수입니다.

인터럽트 처리의 중앙 집중식 위임의 단점

- 유연성 감소

위임된 인터럽트를 처리하는 작업의 우선순위를 각각 설정할 수 없습니다. 위임된 인터럽트를 처리하는 함수가 데몬 작업의 우선순위에 따라 각각 실행되기 때문입니다. 데몬 작업의 우선순위는 `FreeRTOSConfig.h`의 `configTIMER_TASK_PRIORITY` 컴파일 시간 구성 상수에서 설정합니다.

- 불확실성

xTimerPendFunctionCallFromISR()은 명령을 타이머 명령 대기열의 뒷부분으로 전송합니다. 하지만 xTimerPendFunctionCallFromISR()에서 대기열로 전송된 'execute function' 명령에 앞서 이미 타이머 명령 대기열에 있던 명령들이 데몬 작업에서 먼저 처리됩니다.

인터럽트에 따라 시간 상수도 다르기 때문에 일반적으로 두 방법이 모두 동일한 애플리케이션에서 사용됩니다.

## xTimerPendFunctionCallFromISR() API 함수

xTimerPendFunctionCallFromISR()은 xTimerPendFunctionCall()의 인터럽트 세이프 버전입니다. 두 API 함수 모두 애플리케이션 개발자가 제공하는 함수를 RTOS 데몬 작업에서, RTOS 데몬 작업의 컨텍스트에 따라 실행할 수 있도록 해주는 함수입니다. 실행할 함수와 입력 파라미터 값이 타이머 명령 대기열을 통해 데몬 작업으로 전송됩니다. 함수의 실행 시점은 애플리케이션의 다른 작업에 대한 데몬 작업의 우선순위에 따라 달라집니다.

xTimerPendFunctionCallFromISR() API 함수 프로토타입은 다음과 같습니다.

```
BaseType_t xTimerPendFunctionCallFromISR( PendedFunction_t xFunctionToPend, void
      *pvParameter1, uint32_t ulParameter2, BaseType_t *pxHigherPriorityTaskWoken );
```

xTimerPendFunctionCallFromISR()의 xFunctionToPend 파라미터로 전달되는 함수가 따라야 할 프로토타입은 다음과 같습니다.

```
void vPendableFunction( void *pvParameter1, uint32_t ulParameter2 );
```

다음은 xTimerPendFunctionCallFromISR() 파라미터와 반환 값을 나열한 것입니다.

**xFunctionToPend**

데몬 작업에서 실행할 함수를 가리키는 포인터입니다(실제 함수 이름). 함수의 프로토타입은 코드에 보이는 것과 동일해야 합니다.

**pvParameter1**

데몬 작업에서 실행할 함수로 전달되는 값(함수의 pvParameter1 파라미터)입니다. 이 파라미터는 void \* 형식을 갖기 때문에 어떤 데이터 형식이든 전달할 수 있습니다. 예를 들어 정수 형식을 직접 void \*로 변환할 수 있습니다. 그 밖에 void \*는 구조를 가리키는 데도 사용됩니다.

**ulParameter2**

데몬 작업에서 실행할 함수로 전달되는 값(함수의 ulParameter2 파라미터)입니다.

**pxHigherPriorityTaskWoken**

xTimerPendFunctionCallFromISR()은 타이머 명령 대기열에 명령을 작성합니다. RTOS 데몬 작업이 데이터를 타이머 명령 대기열에서 사용할 수 있을 때까지 Blocked 상태로 기다려야 하는 경우에는 타이머 명령 대기열에 대한 쓰기 연산으로 데몬 작업이 Blocked 상태를 종료하게 됩니다. 데몬 작업의 우선순위가 현재 실행 중인 작업(중단된 작업)보다 높다면 xTimerPendFunctionCallFromISR()이 \*pxHigherPriorityTaskWoken을 pdTRUE로 설정합니다. xTimerPendFunctionCallFromISR()이 이 값을 pdTRUE로 설정하면 인터럽트 종료 전에 컨텍스트 전환이 이루어져야 합니다. 그러면 데몬 작업이 최고 우선순위인 Ready 상태의 작업이 되어 인터럽트 복귀가 데몬 작업으로 직접 이루어질 수 있습니다.

가능한 반환 값은 다음 두 가지입니다.

- pdPASS

'execute function' 명령이 타이머 명령 대기열에 작성되는 경우 반환됩니다.

- pdFAIL

타이머 명령 대기열이 이미 가득 차있어서 'execute function' 명령이 타이머 명령 대기열에 작성되지 않은 경우 반환됩니다. 타이머 명령 길이를 설정하는 방법에 대한 자세한 내용은 software\_tier\_management 단원을 참조하십시오.

## 인터럽트 처리의 중앙 집중식 위임(예제 18)

예제 18은 예제 16과 비슷한 기능을 제공하지만 세마포어를 사용해 작업을 생성하지 않고 인터럽트에 필요한 처리를 실행합니다. 대신에 RTOS 데몬 작업에서 처리를 수행합니다.

예제 18에서 사용되는 인터럽트 서비스 루틴은 다음 코드와 같습니다. xTimerPendFunctionCallFromISR()을 호출하여 vDeferredHandlingFunction() 함수를 가리키는 포인터를 데몬 작업에게 전달합니다. 위임된 인터럽트 처리는 vDeferredHandlingFunction() 함수에서 수행합니다.

인터럽트 서비스 루틴은 실행될 때마다 변수인 ulParameterValue의 값을 높입니다. ulParameterValue는 xTimerPendFunctionCallFromISR() 호출 시 ulParameter2 값으로 사용되기 때문에 데몬 작업에서 vDeferredHandlingFunction()을 실행하면서 vDeferredHandlingFunction()을 호출할 때 ulParameter2 값으로도 사용됩니다. 함수의 다른 파라미터인 pvParameter1은 이번 예제에서 사용되지 않습니다.

```
static uint32_t ulExampleInterruptHandler( void )
{
    static uint32_t ulParameterValue = 0;

    BaseType_t xHigherPriorityTaskWoken;

    /* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE because it
    will get set to pdTRUE inside the interrupt-safe API function if a context switch is
    required. */

    xHigherPriorityTaskWoken = pdFALSE;

    /* Send a pointer to the interrupt's deferred handling function to the daemon task.
    The deferred handling function's pvParameter1 parameter is not used, so just set to NULL.
    The deferred handling function's ulParameter2 parameter is used to pass a number that is
    incremented by one each time this interrupt handler executes. */

    xTimerPendFunctionCallFromISR( vDeferredHandlingFunction, /* Function to
    execute. */ NULL, /* Not used. */ ulParameterValue, /* Incrementing value. */
    &xHigherPriorityTaskWoken );

    ulParameterValue++;

    /* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR(). If
    xHigherPriorityTaskWoken was set to pdTRUE inside xTimerPendFunctionCallFromISR() then
    calling portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is
    still pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. Unlike most FreeRTOS
    ports, the Windows port requires the ISR to return a value. The return statement is inside
    the Windows version of portYIELD_FROM_ISR(). */

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

```
}
```

vDeferredHandlingFunction()의 구현체는 다음과 같습니다. 이 함수를 실행하면 고정 문자열과 ulParameter2 파라미터 값이 출력됩니다.

```
static void vDeferredHandlingFunction( void *pvParameter1, uint32_t ulParameter2 )
{
    /* Process the event - in this case, just print out a message and the value of
    ulParameter2. pvParameter1 is not used in this example. */

    vPrintStringAndNumber( "Handler function - Processing event ", ulParameter2 );
}
```

사용되는 main() 함수는 다음과 같습니다. vPeriodicTask()는 소프트웨어 인터럽트를 주기적으로 생성하는 작업입니다. 이때 생성되는 작업의 우선순위는 데몬 작업의 우선순위보다 낮기 때문에 데몬 작업이 Blocked 상태를 종료하자마자 먼저 실행됩니다.

```
int main( void )
{
    /* The task that generates the software interrupt is created at a priority below
    the priority of the daemon task. The priority of the daemon task is set by the
    configTIMER_TASK_PRIORITY compile time configuration constant in FreeRTOSConfig.h. */

    const UBaseType_t ulPeriodicTaskPriority = configTIMER_TASK_PRIORITY - 1;

    /* Create the task that will periodically generate a software interrupt. */

    xTaskCreate( vPeriodicTask, "Periodic", 1000, NULL, ulPeriodicTaskPriority, NULL );

    /* Install the handler for the software interrupt. The syntax necessary to do this is
    dependent on the FreeRTOS port being used. The syntax shown here can only be used with the
    FreeRTOS windows port, where such interrupts are only simulated. */

    vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulExampleInterruptHandler );

    /* Start the scheduler so the created task starts executing. */

    vTaskStartScheduler();

    /* As normal, the following line should never be reached. */

    for( ;; );
}
```

코드를 통해 출력되는 화면은 다음과 같습니다. 데몬 작업의 우선순위가 소프트웨어 인터럽트를 생성하는 작업의 우선순위보다 높기 때문에 인터럽트 생성과 함께 데몬 작업이 vDeferredHandlingFunction()을 바로 실행합니다. 결과적으로 vDeferredHandlingFunction()에서 출력되는 메시지는 세마포어를 사용해 위임된 인터럽트를 처리하는 작업의 차단을 해제할 때와 마찬가지로 주기적 작업에서 출력되는 두 메시지 사이에 표시됩니다.

```
C:\Windows\system32\cmd.exe - rtdemo

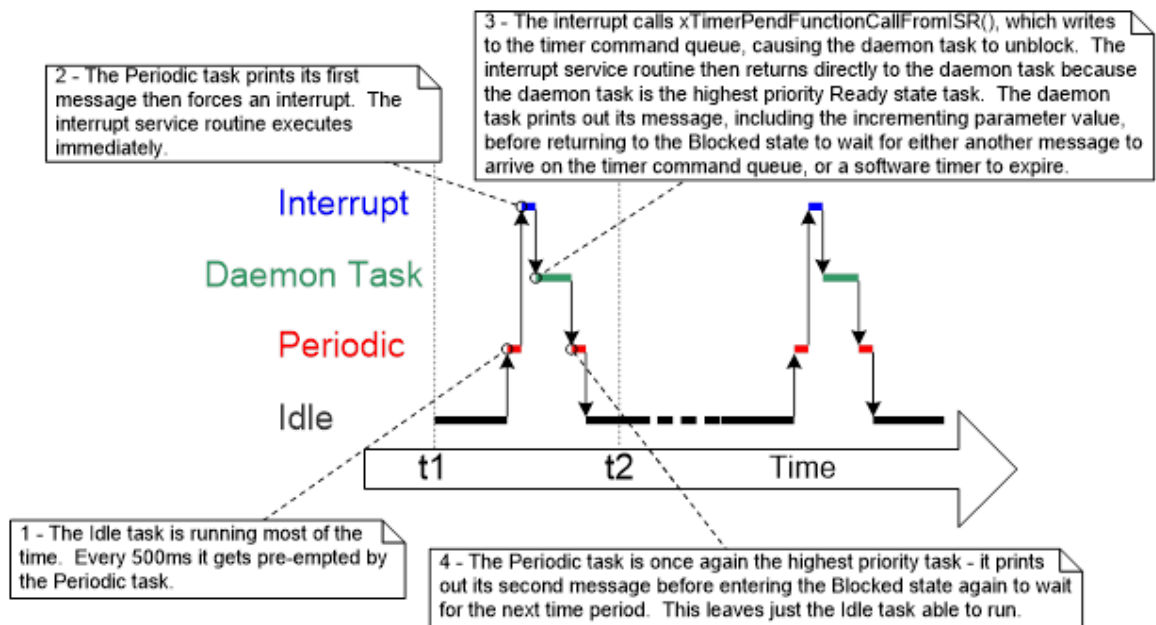
C:\temp>rtdemo
Periodic task - About to generate an interrupt.
Handler function - Processing event 0
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler function - Processing event 1
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler function - Processing event 2
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler function - Processing event 3
Periodic task - Interrupt generated.
```

실행 시퀀스는 다음과 같습니다.



## 인터럽트 서비스 루틴 내부에서 대기열 사용

이진 및 계수 세마포어는 이벤트를 알리는 데 사용됩니다. 대기열은 이벤트를 알리고 데이터를 전송하는 데 사용됩니다.

`xQueueSendToFrontFromISR()`은 `xQueueSendToFront()`를 인터럽트 서비스 루틴에서 안전하게 사용할 수 있는 버전입니다. `xQueueSendToBackFromISR()`은 `xQueueSendToBack()`을 인터럽트 서비스 루틴에서 안전하게 사용할 수 있는 버전입니다. `xQueueReceiveFromISR()`은 `xQueueReceive()`를 인터럽트 서비스 루틴에서 안전하게 사용할 수 있는 버전입니다.

## xQueueSendToFrontFromISR() 및 xQueueSendToBackFromISR() API 함수

xQueueSendToFrontFromISR() API 함수 프로토타입은 다음과 같습니다.

```
BaseType_t xQueueSendToFrontFromISR( QueueHandle_t xQueue, void *pvItemToQueue BaseType_t  
*pxHigherPriorityTaskWoken );
```

xQueueSendToBackFromISR() API 함수 프로토타입은 다음과 같습니다.

```
BaseType_t xQueueSendToBackFromISR( QueueHandle_t xQueue, void *pvItemToQueue BaseType_t  
*pxHigherPriorityTaskWoken );
```

xQueueSendFromISR()과 xQueueSendToBackFromISR()은 기능이 동일한 함수입니다.

다음은 xQueueSendToFrontFromISR() 및 xQueueSendToBackFromISR() 파라미터와 반환 값을 나열한 것입니다.

**xQueue**

데이터가 전송되는(작성되는) 대기열의 핸들입니다. 대기열 핸들은 대기열 생성을 위해 xQueueCreate()를 호출할 때 반환됩니다.

**pvItemToQueue**

대기열로 복사할 데이터를 가리키는 포인터입니다. 대기열에 저장할 수 있는 각 항목의 크기는 대기열을 생성할 때 설정됩니다. 따라서 여기에서 설정된 크기의 바이트가 pvItemToQueue에서 대기열 저장 영역으로 복사됩니다.

**pxHigherPriorityTaskWoken**

단일 대기열은 데이터를 사용할 수 있을 때까지 차단된 상태로 대기하는 작업을 1개 이상 가질 수 있습니다. xQueueSendToFrontFromISR() 또는 xQueueSendToBackFromISR()을 호출하면 데이터를 사용할 수 있게 되어 대기하던 작업이 Blocked 상태를 종료하게 됩니다. API 함수를 호출하여 작업의 Blocked 상태가 종료 되었을 때 차단 해제된 작업의 우선순위가 현재 실행 중인 작업(중단된 작업)보다 높으면 내부적으로 API 함수가 \*pxHigherPriorityTaskWoken을 pdTRUE로 설정합니다. xQueueSendToFrontFromISR() 또는 xQueueSendToBackFromISR()이 이 값을 pdTRUE로 설정하면 인터럽트 종료 전에 컨텍스트 전환이 이루어져야 합니다. 그러면 최고 우선순위의 Ready 상태 작업으로 인터럽트 복귀가 직접 이루어질 수 있습니다.

가능한 반환 값은 다음 두 가지입니다.

- pdPASS

데이터가 대기열에 성공적으로 전송된 경우에 한해 반환됩니다.

- errQUEUE\_FULL

대기열이 이미 가득 찬 상태여서 데이터를 대기열에 전송할 수 없을 때 반환됩니다.

## ISR에서 대기열을 사용할 때 고려해야 할 사항

대기열은 데이터를 인터럽트에서 작업으로 쉽고 편하게 전달할 수 있는 방법입니다. 하지만 데이터가 높은 주기로 수신되는 경우에는 대기열을 사용해도 효율적이지 않습니다.

FreeRTOS 다운로드에 있는 데모 애플리케이션들은 대부분 대기열을 사용해 문자를 UART의 수신 ISR에서 전달하는 UART 드라이버가 포함되어 있습니다. 이러한 데모 애플리케이션들에서 대기열을 사용하는 이유는 ISR의 대기열 사용을 보여주는 동시에 FreeRTOS 포트를 테스트할 목적으로 시스템을 신중하게 로드하기 위해서입니다. 하지만 이러한 방식으로 대기열을 사용하는 ISR은 효율적인 설계라고 할 수 없습니다. 데이터 수신에 느리지 않는 한 프로덕션 코드에서 이러한 기법을 사용하는 것은 바람직하지 않습니다. 다음과 같은 방식이 더욱 효율적인 기법입니다.

- DMA(Direct Memory Access) 하드웨어를 사용해 문자를 수신한 후 버퍼에 저장합니다. 이러한 방법은 실제로 소프트웨어 오버헤드가 없습니다. 그런 다음 작업 직접 알림을 사용해 전송이 끝난 것을 감지한 후에만 버퍼를 처리하도록 작업의 차단을 해제할 수 있습니다. 작업 직접 알림은 ISR에서 작업의 차단을 해제하는 데 가장 효율적인 방법입니다. 자세한 내용은 [작업 알림 \(p. 191\)](#) 단원을 참조하십시오.
- 수신된 문자를 각각 스레드 세이프 RAM 버퍼로 복사합니다. FreeRTOS+TCP에서 제공되는 '스트림 버퍼' 기능을 이러한 목적으로 사용할 수 있습니다. 다시 말하지만 작업 직접 알림을 사용해 전체 메시지가 수신된 후에, 혹은 전송이 끝난 것을 감지한 후에 버퍼를 처리하도록 작업의 차단을 해제할 수 있습니다.
- 수신된 문자를 ISR 내부에서 직접 처리한 후 대기열을 사용해 데이터(원시 데이터 아님) 처리 결과만 작업으로 전송합니다.

## 인터럽트 내부에서 대기열 전송 및 수신(예제 19)

이번 예제에서는 동일한 인터럽트 내부에서 `xQueueSendToBackFromISR()`과 `xQueueReceiveFromISR()`의 사용에 대해 설명합니다. 편의상 인터럽트는 소프트웨어에서 생성됩니다.

주기적 작업이 생성되어 숫자 값 5개를 200밀리초마다 대기열에 전송합니다. 소프트웨어 인터럽트는 값 5개가 모두 전송된 후에만 생성됩니다. 작업 구현체는 다음과 같습니다.

```
static void vIntegerGenerator( void *pvParameters )
{
    TickType_t xLastExecutionTime;

    uint32_t ulValueToSend = 0;

    int i;

    /* Initialize the variable used by the call to vTaskDelayUntil(). */
    xLastExecutionTime = xTaskGetTickCount();

    for( ;; )
    {
        /* This is a periodic task. Block until it is time to run again. The task will
        execute every 200 ms. */
        vTaskDelayUntil( &xLastExecutionTime, pdMS_TO_TICKS( 200 ) );

        /* Send five numbers to the queue, each value one higher than the previous value.
        The numbers are read from the queue by the interrupt service routine. The interrupt
        service routine always empties the queue, so this task is guaranteed to be able to write
        all five values without needing to specify a block time. */
        for( i = 0; i < 5; i++ )
        {
            xQueueSendToBack( xIntegerQueue, &ulValueToSend, 0 );
        }
    }
}
```

```
        ulValueToSend++;

    }

    /* Generate the interrupt so the interrupt service routine can read the values from
    the queue. The syntax used to generate a software interrupt depends on the FreeRTOS port
    being used. The syntax used below can only be used with the FreeRTOS Windows port, in
    which such interrupts are only simulated.*/

    vPrintString( "Generator task - About to generate an interrupt.\r\n" );

    vPortGenerateSimulatedInterrupt( mainINTERRUPT_NUMBER );

    vPrintString( "Generator task - Interrupt generated.\r\n\r\n\r\n" );

}

}
```

주기적 작업에서 대기열에 작성된 모든 값을 읽어와서 대기열이 비워질 때까지 인터럽트 서비스 루틴이 xQueueReceiveFromISR()을 반복해서 호출합니다. 각 수신된 값의 마지막 두 비트는 문자열 배열에 대한 인덱스로 사용됩니다. 그런 다음 xQueueSendFromISR()을 호출하여 해당하는 인덱스 위치에서 문자열을 가리키는 포인터가 다른 대기열로 전송됩니다. 인터럽트 서비스 루틴의 구현체는 다음과 같습니다.

```
static uint32_t ulExampleInterruptHandler( void )
{
    BaseType_t xHigherPriorityTaskWoken;

    uint32_t ulReceivedNumber;

    /* The strings are declared static const to ensure they are not allocated on the
    interrupt service routine's stack, and so exist even when the interrupt service routine is
    not executing. */

    static const char *pcStrings[] =
    {
        "String 0\r\n",
        "String 1\r\n",
        "String 2\r\n",
        "String 3\r\n"
    };

    /* As always, xHigherPriorityTaskWoken is initialized to pdFALSE to be able to detect
    it getting set to pdTRUE inside an interrupt-safe API function. Because an interrupt-safe
    API function can only set xHigherPriorityTaskWoken to pdTRUE, it is safe to use the same
    xHigherPriorityTaskWoken variable in both the call to xQueueReceiveFromISR() and the call
    to xQueueSendToBackFromISR(). */

    xHigherPriorityTaskWoken = pdFALSE;

    /* Read from the queue until the queue is empty. */

    while( xQueueReceiveFromISR( xIntegerQueue, &ulReceivedNumber,
    &xHigherPriorityTaskWoken ) != errQUEUE_EMPTY )
    {
```



```
/* Truncate the received value to the last two bits (values 0 to 3 inclusive), and
then use the truncated value as an index into the pcStrings[] array to select a string
(char *) to send on the other queue. */

ulReceivedNumber &= 0x03;

xQueueSendToBackFromISR( xStringQueue, &pcStrings[ ulReceivedNumber ],
&xHigherPriorityTaskWoken );

}

/* If receiving from xIntegerQueue caused a task to leave the Blocked state, and if the
priority of the task that left the Blocked state is higher than the priority of the task
in the Running state, then xHigherPriorityTaskWoken will have been set to pdTRUE inside
xQueueReceiveFromISR(). If sending to xStringQueue caused a task to leave the Blocked
state, and if the priority of the task that left the Blocked state is higher than the
priority of the task in the Running state, then xHigherPriorityTaskWoken will have been
set to pdTRUE inside xQueueSendToBackFromISR(). xHigherPriorityTaskWoken is used as the
parameter to portYIELD_FROM_ISR(). If xHigherPriorityTaskWoken equals pdTRUE, then calling
portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still
pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. The implementation of
portYIELD_FROM_ISR() used by the Windows port includes a return statement, which is why
this function does not explicitly return a value. */

portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

인터럽트 서비스 루틴에서 문자 포인터를 수신하는 작업은 메시지가 수신될 때까지 대기열에서 차단되며, 메시지가 수신되면 각 문자열을 출력합니다. 구현체는 다음과 같습니다.

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

    for( ;; )
    {
        /* Block on the queue to wait for data to arrive. */
        xQueueReceive( xStringQueue, &pcString, portMAX_DELAY );

        /* Print out the string received. */
        vPrintString( pcString );
    }
}
```

평소와 같이 main()이 스케줄러를 시작하기 전에 필요한 대기열과 작업을 생성합니다. 구현체는 다음과 같습니다.

```
int main( void )
{
    /* Before a queue can be used, it must first be created. Create both queues used by
    this example. One queue can hold variables of type uint32_t. The other queue can hold
```

```
variables of type char*. Both queues can hold a maximum of 10 items. A real application
should check the return values to ensure the queues have been successfully created. */

xIntegerQueue = xQueueCreate( 10, sizeof( uint32_t ) );

xStringQueue = xQueueCreate( 10, sizeof( char * ) );

/* Create the task that uses a queue to pass integers to the interrupt service routine.
The task is created at priority 1. */

xTaskCreate( vIntegerGenerator, "IntGen", 1000, NULL, 1, NULL );

/* Create the task that prints out the strings sent to it from the interrupt service
routine. This task is created at the higher priority of 2. */

xTaskCreate( vStringPrinter, "String", 1000, NULL, 2, NULL );

/* Install the handler for the software interrupt. The syntax required to do this is
depends on the FreeRTOS port being used. The syntax shown here can only be used with the
FreeRTOS Windows port, where such interrupts are only simulated. */

vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulExampleInterruptHandler );

/* Start the scheduler so the created tasks start executing. */

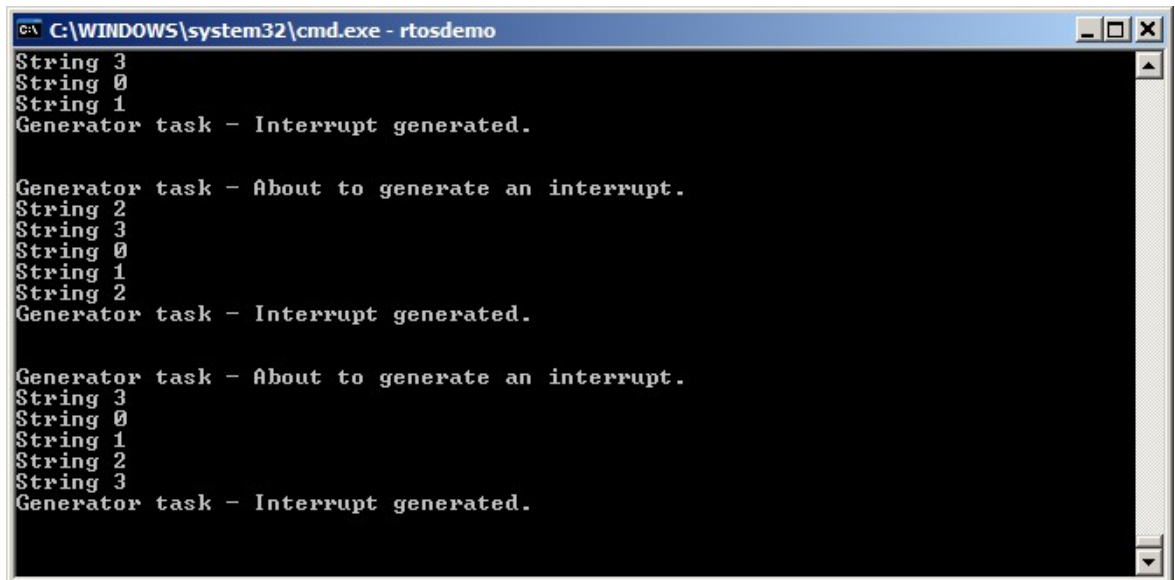
vTaskStartScheduler();

/* If all is well, then main() will never reach here because the scheduler will
now be running the tasks. If main() does reach here, then it is likely that there was
insufficient heap memory available for the idle task to be created. */

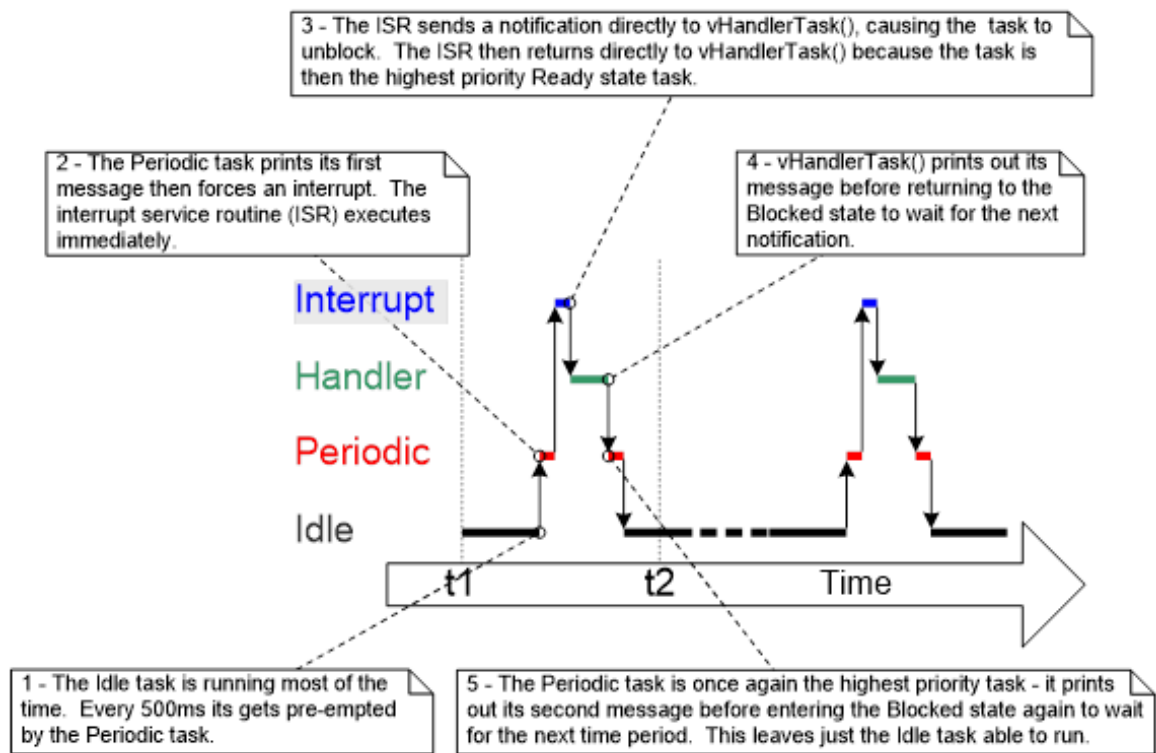
for( ;; );

}
```

출력되는 화면은 다음과 같습니다. 인터럽트가 정수 5개를 모두 수신한 후 응답으로 문자열 5개를 생성하는 것을 볼 수 있습니다.



실행 시퀀스는 다음과 같습니다.



## 인터럽트 중첩

작업 우선순위와 인터럽트 우선순위를 혼동하는 것은 매우 흔한 일입니다. 이번 단원에서는 인터럽트 우선순위, 즉 ISR이 서로서로 실행되는 우선순위에 대해서 살펴보겠습니다. 작업에 할당되는 우선순위는 인터럽트에 할당되는 우선순위와 아무런 관련도 없습니다. ISR의 실행 시점은 하드웨어가 결정하고, 작업의 실행 시점은 소프트웨어가 결정합니다. 하드웨어 인터럽트에 대한 응답으로 실행되는 ISR은 작업을 중단하지만 작업이 ISR보다 먼저 실행되지는 않습니다.

인터럽트 중첩을 지원하는 포트는 다음 표에 나열된 상수 중 1개 또는 2개 모두를 `FreeRTOSConfig.h`에서 정의해야 합니다. `configMAX_SYSCALL_INTERRUPT_PRIORITY`와 `configMAX_API_CALL_INTERRUPT_PRIORITY`는 모두 동일한 속성을 정의합니다. 이전 FreeRTOS 포트 일수록 `configMAX_SYSCALL_INTERRUPT_PRIORITY`를 사용합니다. 그리고 최신 FreeRTOS 포트일수록 `configMAX_API_CALL_INTERRUPT_PRIORITY`를 사용합니다.

다음 표는 인터럽트 중첩을 제어하는 상수를 나열한 것입니다.

상수	설명
<code>configMAX_SYSCALL_INTERRUPT_PRIORITY</code> 또는 <code>configMAX_API_CALL_INTERRUPT_PRIORITY</code>	인터럽트 세이프 FreeRTOS API 함수를 호출할 수 있는 최고의 인터럽트 우선순위를 설정합니다.
<code>configKERNEL_INTERRUPT_PRIORITY</code>	<p>틱 인터럽트에서 사용할 인터럽트 우선순위를 설정하며, 항상 최대한 낮은 인터럽트 우선순위로 설정되어야 합니다.</p> <p>사용 중인 FreeRTOS 포트 역시 <code>configMAX_SYSCALL_INTERRUPT_PRIORITY</code> 상수를 사용하지 않는 경우에는 인터럽트 세이프 FreeRTOS API 함수를 사용하는 모든 인터럽트가</p>

`configKERNEL_INTERRUPT_PRIORITY`에서 정의하는 우선순위로 실행되어야 합니다.

각 인터럽트 소스는 숫자 우선순위와 논리 우선순위를 갖습니다.

- 숫자

인터럽트 우선순위에 할당되는 숫자입니다. 예를 들어 인터럽트에 우선순위 7이 할당되면 숫자 우선순위는 7이 됩니다. 마찬가지로 인터럽트에 우선순위 200이 할당되면 숫자 우선순위는 200이 됩니다.

- 논리

다른 인터럽트에 대한 인터럽트 우선순위를 설명합니다. 우선순위가 다른 인터럽트 2개가 동시에 발생할 경우 프로세서는 논리 우선순위가 더욱 높은 인터럽트에게 ISR을 실행합니다.

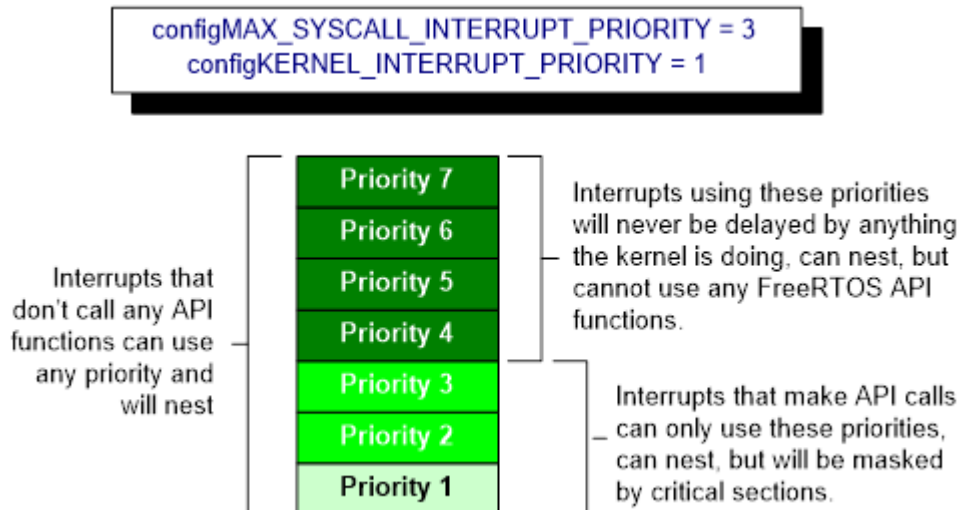
인터럽트는 논리 우선순위가 낮은 인터럽트를 모두 중단(중첩)할 수 있지만 논리 우선순위가 같거나 높은 인터럽트는 중단하지 못합니다.

인터럽트의 숫자 우선순위와 논리 우선순위 간 관계는 프로세서 아키텍처에 따라 달라집니다. 일부 프로세서에서는 인터럽트에 할당되는 숫자 우선순위가 높을수록 인터럽트의 논리 우선순위도 높아집니다. 하지만 인터럽트에 할당되는 숫자 우선순위가 높을수록 인터럽트의 논리 우선순위가 낮아지는 프로세서 아키텍처도 있습니다.

`configMAX_SYSCALL_INTERRUPT_PRIORITY`를 `configKERNEL_INTERRUPT_PRIORITY`보다 더욱 높은 논리 우선순위로 설정하면 완전한 인터럽트 중첩 모델을 생성할 수 있습니다. 다음 그림은 아래와 같은 시나리오를 나타낸 것입니다.

- 프로세서의 고유 인터럽트 우선순위가 7입니다.
- 숫자 우선순위 7이 할당된 인터럽트의 논리 우선순위가 숫자 우선순위 1이 할당된 인터럽트의 논리 우선순위보다 높습니다.
- `configKERNEL_INTERRUPT_PRIORITY`가 1로 설정됩니다.
- `configMAX_SYSCALL_INTERRUPT_PRIORITY`가 3으로 설정됩니다.

아래 그림은 인터럽트 중첩 동작에 영향을 미치는 상수를 나타낸 것입니다.



- 커널 또는 애플리케이션이 중요한 섹션에 있을 때는 우선순위 1~3(3 포함)을 사용하는 인터럽트가 실행되지 않습니다. 이러한 우선순위에서 실행되는 ISR은 인터럽트 세이프 FreeRTOS API 함수를 사용할 수 있습니다. 중요한 섹션에 대한 자세한 내용은 [리소스 관리 \(p. 152\)](#) 단원을 참조하십시오.
- 우선순위 4 이상을 사용하는 인터럽트는 중요한 섹션의 영향을 받지 않기 때문에 스케줄러의 어떠한 작업도 하드웨어 자체의 제한을 벗어나지 않는 범위에서 인터럽트가 바로 실행되는 것을 막지 못합니다. 이러한 우선순위에서 실행되는 ISR은 어떠한 FreeRTOS API 함수도 사용하지 못합니다.
- 일반적으로 매우 엄격한 타이밍 정확도가 필요한 기능(모터 제어 등)은 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY보다 높은 우선순위를 사용해야만 스케줄러가 인터럽트 응답 시간에 지터를 발생시키지 못합니다.

## ARM Cortex-M 및 ARM GIC 사용자

이번 단원은 Cortex-M0 및 Cortex-M0+ 코어에만 일부분 적용됩니다. Cortex-M 프로세서의 인터럽트 구성은 혼동을 일으켜 오류가 발생하기 쉽습니다. FreeRTOS Cortex-M 포트는 configASSERT()가 정의된 경우에 한해 사용자의 개발을 지원할 목적으로 인터럽트 구성을 자동으로 검사합니다. configASSERT()에 대한 자세한 내용은 [개발자 지원 \(p. 217\)](#) 단원을 참조하십시오.

ARM Cortex 코어와 ARM GIC(Generic Interrupt Controller)는 낮은 우선순위의 숫자를 사용해 논리적으로 높은 우선순위의 인터럽트를 표현합니다. 이러한 방법은 직관적으로 보이지 않을 수 있습니다. 예를 들어 인터럽트에 낮은 논리 우선순위를 할당하려면 높은 숫자 우선순위를 할당해야 하기 때문입니다. 반대로 인터럽트에 높은 논리 우선순위를 할당하려면 낮은 숫자 우선순위를 할당해야 합니다.

Cortex-M 인터럽트 컨트롤러는 최대 8비트를 사용해 각 인터럽트 우선순위를 지정할 수 있기 때문에 가장 낮은 우선순위는 255가 됩니다. 가장 높은 우선순위는 0입니다. 하지만 Cortex-M 마이크로컨트롤러는 일반적으로 가능한 8비트의 하위 집합만 구현합니다. 구현되는 비트 수는 마이크로컨트롤러 제품군에 따라 달라집니다.

하위 집합만 구현되는 경우에는 바이트의 최상위 비트만 사용할 수 있습니다. 최하위 비트는 구현되지 않습니다. 구현되지 않은 비트는 어떤 값이든 가질 수 있지만 1로 설정하는 것이 일반적입니다. 다음 그림은 이진수 101의 우선순위가 우선순위 비트 4개를 구현하는 Cortex-M 마이크로컨트롤러에 어떻게 저장되는지 나타낸 것입니다.



최하위 비트 4개는 구현되지 않기 때문에 이진수 101이 최상위 비트 4개로 전환되는 것을 볼 수 있습니다. 구현되지 않은 비트는 모두 1로 설정되었습니다.

일부 라이브러리 함수에서는 구현되지 않은 비트를 구현되는(최상위) 비트로 전환한 후 우선순위 값을 지정할 수 있습니다. 이러한 함수를 사용할 경우 위 그림에 보이는 우선순위를 십진수 95로 지정할 수 있습니다. 십진수 95는 4비트를 통해 전환되어 이진수 101nnnn(n은 구현되지 않은 비트임)이 되는 이진수 101입니다. 구현되지 않은 비트는 모두 1로 설정되어 이진수 1011111이 됩니다.

일부 라이브러리 함수에서는 구현되는(최상위) 비트로 전환되기 전에 우선순위 값을 지정할 수 있습니다. 이러한 함수를 사용할 경우 위 그림에 보이는 우선순위를 십진수 5로 지정해야 합니다. 십진수 5는 전환이 없는 이진수 101입니다.

configMAX\_SYSCALL\_INTERRUPT\_PRIORITY와 configKERNEL\_INTERRUPT\_PRIORITY는 Cortex-M 레지스터에 직접 작성할 수 있도록(즉 우선순위 값이 구현된 비트로 전환된 후) 지정되어야 합니다.

`configKERNEL_INTERRUPT_PRIORITY`는 항상 최대한 낮은 인터럽트 우선순위로 설정되어야 합니다. 구현되지 않은 우선순위 비트는 1로 설정할 수 있습니다. 따라서 상수는 구현되는 우선순위 비트 수에 상관없이 항상 255로 설정할 수 있습니다.

Cortex-M 인터럽트는 가장 높은 우선순위인 0으로 기본 설정됩니다. Cortex-M 하드웨어를 구현할 때는 `configMAX_SYSCALL_INTERRUPT_PRIORITY`를 0으로 설정할 수 없습니다. 따라서 FreeRTOS API를 사용하는 인터럽트의 우선순위를 기본 값으로 놔두어서는 안 됩니다.

# 리소스 관리

이번 단원에서 다루는 내용은 다음과 같습니다.

- 리소스 관리 및 제어가 필요한 시점과 이유
- 임계 영역
- 상호 배제의 의미
- 스케줄러 일시 중지의 의미
- 뮤텍스 사용 방법
- 게이트키퍼 작업의 생성 및 사용 방법
- 우선순위 역전의 의미와 우선순위 상속을 통해 영향을 줄일 수 있는(제거 아님) 방법

멀티태스킹 시스템에서 작업 하나가 리소스 액세스를 시작한 후 Running 상태가 종료되기 전에 액세스를 마치지 못하면 오류가 발생할 수 있습니다. 작업이 바뀐 상태에서 리소스를 해제하면 다른 작업 또는 인터럽트가 동일한 리소스에 액세스할 때 데이터 손상 또는 기타 비슷한 문제를 일으킬 수 있습니다.

다음은 일부 예입니다.

## 1. 주변 장치 액세스

다음과 같이 작업 2개가 액정 디스플레이(LCD)에 작성하는 시나리오를 예로 들어보겠습니다.

- 작업 A가 실행되어 LCD에서 "Hello world"라는 문자열을 작성합니다.
- "Hello w" 문자열까지 출력된 순간 작업 A가 작업 B에게 선점됩니다.
- 작업 B가 Blocked 상태로 전환되기 전에 LCD에게 "Abort, Retry, Fail?"을 작성합니다.
- 작업 A가 선점된 지점부터 이어서 실행되어 나머지 문자("orld")를 출력합니다.

LCD가 손상된 문자열인 "Hello wAbort, Retry, Fail?orld"를 표시합니다.

## 2. 읽기, 수정, 쓰기 작업

다음은 C 코드 라인과 C 코드 라인이 어셈블리 코드로 변환되는 방식을 예제로 나타낸 것입니다. 예제를 보면 먼저 PORTA 값을 메모리에서 레지스터로 읽어와 레지스터에서 수정한 후 다시 메모리로 작성하고 있습니다. 이러한 방식을 읽기, 수정, 쓰기 작업이라고 합니다.

```
/* The C code being compiled. */

PORTA |= 0x01;

/* The assembly code produced when the C code is compiled. */

LOAD R1,[#PORTA] ; Read a value from PORTA into R1

MOVE R2,#0x01 ; Move the absolute constant 1 into R2

OR R1,R2 ; Bitwise OR R1 (PORTA) with R2 (constant 1)

STORE R1,[#PORTA] ; Store the new value back to PORTA
```

이 작업은 비원자적입니다. 작업을 마치는 데 명령어가 2개 이상 필요할 뿐만 아니라 중단될 수도 있기 때문입니다. 다음과 같이 작업 2개가 PORTA라고 하는 메모리 맵 레지스터(MMR)를 업데이트하는 시나리오를 예로 들어보겠습니다.

1. 작업 A가 PORTA 값을 읽기 동작인 레지스터로 로드합니다.
2. 작업 A가 동일한 작업에서 수정 및 쓰기 동작을 마치기 전에 작업 B에게 선점됩니다.
3. 작업 B가 PORTA 값을 업데이트한 후 Blocked 상태로 전환됩니다.
4. 작업 A가 선점된 지점부터 이어서 실행되고, 업데이트된 값을 다시 PORTA에 작성하기 전에 레지스터에 이미 저장된 PORTA 값의 복사본을 수정합니다.

이번 시나리오에서 작업 A는 이전 PORTA 값을 업데이트하여 다시 작성합니다. 작업 B는 작업 A가 PORTA 값의 복사본을 가져온 후에 PORTA를 수정하여, 결국 작업 A는 수정된 값을 다시 PORTA 레지스터에 작성하지 못합니다. 작업 A가 PORTA에 작성할 때는 작업 B가 앞서 실행되면서 수정한 값을 덮어쓰기 때문에 실제로 PORTA 레지스터 값이 손상됩니다.

이번 예에서는 주변 장치 레지스터를 사용했지만 변수에 대해 읽기, 수정, 쓰기 작업을 실행할 때도 동일한 원칙이 적용됩니다.

#### 1. 변수에 대한 비원자적 액세스

비원자적 작업을 예로 들면 데이터 구조의 여러 멤버를 업데이트하거나, 혹은 아키텍처의 자연어 크기보다 큰 변수를 업데이트하는(16비트 시스템에서 32비트 변수를 업데이트하는 등) 경우가 있습니다. 이러한 작업이 중단되면 데이터 손실 또는 손상을 일으킬 수 있습니다.

#### 2. 함수 재진입성

다수의 작업에서, 혹은 작업과 인터럽트 모두에서 함수를 안전하게 호출할 수 있는 경우 함수는 재진입이 가능합니다. 이러한 재진입 가능 함수는 스레드-세이프인 것으로 알려져 있습니다. 데이터 또는 논리 연산이 손상될 염려 없이 다수의 실행 스레드에서 액세스가 가능하기 때문입니다. 작업마다 자체 스택과 프로세서(하드웨어) 레지스터 값 집합을 유지합니다. 함수가 스택이나 레지스터에 저장된 데이터가 아닌 다른 데이터에 액세스하지 않는다면 이 함수는 재진입이 가능한 스레드 세이프입니다. 다음은 재진입 가능 함수의 예제입니다.

```
/* A parameter is passed into the function. This will either be passed on the stack, or
in a processor register. Either way is safe because each task or interrupt that calls
the function maintains its own stack and its own set of register values, so each task or
interrupt that calls the function will have its own copy of lVar1. */
long lAddOneHundred( long lVar1 )

{

    /* This function scope variable will also be allocated to the stack or a register,
depending on the compiler and optimization level. Each task or interrupt that calls this
function will have its own copy of lVar2. */

    long lVar2;

    lVar2 = lVar1 + 100;

    return lVar2;

}
```

다음은 재진입이 가능하지 않은 함수의 예제입니다.

```
/* In this case lVar1 is a global variable, so every task that calls lNonsenseFunction will
access the same single copy of the variable. */

long lVar1;

long lNonsenseFunction( void )

{
```



```
/* lState is static, so is not allocated on the stack. Each task that calls this
function will access the same single copy of the variable. */

static long lState = 0;

long lReturn;

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

        lState = 1;

        break;

    case 1 : lReturn = lVar1 + 20;

        lState = 0;

        break;
}
}
```

## 상호 배제

데이터 일관성을 항상 유지하려면 상호 배제 기법을 사용해 작업 사이, 혹은 작업과 인터럽트 사이의 공유 리소스에 대한 액세스를 관리하십시오. 이러한 기법의 목적은 작업이 재진입이 어려워 스레드 세이프가 아닌 공유 리소스에 액세스하기 시작할 때 리소스가 일관된 상태로 돌아올 때까지 동일한 작업이 리소스에 대한 배타적 액세스를 유지하는 데 있습니다.

FreeRTOS는 상호 배제를 구현할 때 사용할 수 있는 몇 가지 기능을 제공하지만 가능하다면 리소스를 공유하지 않는 동시에 각 리소스에 대한 액세스가 단일 작업에서만 이루어지도록 애플리케이션을 설계할 때 상호 배제 기법이 가장 효과적으로 구현됩니다.

## 임계 영역과 스케줄러의 일시 중지

## 기본 임계 영역

기본 임계 영역이란 `taskENTER_CRITICAL()` 매크로와 `taskEXIT_CRITICAL()` 매크로를 각각 호출하여 둘러싸이는 코드 구간을 말합니다. 그래서 임계 영역을 임계 구간이라고도 합니다.

`taskENTER_CRITICAL()`과 `taskEXIT_CRITICAL()`은 파라미터나 반환 값이 없습니다. (함수 매크로는 실제 함수와 같은 방식으로 값을 반환하지 않지만 매크로를 함수와 동일하게 생각하는 것이 가장 간단합니다)

두 함수의 사용 방법은 다음과 같으며, 여기에서 임계 영역은 레지스터에 대한 액세스를 보호하는 데 사용됩니다.

```
/* Ensure access to the PORTA register cannot be interrupted by placing it within a
critical section. Enter the critical section. */

taskENTER_CRITICAL();
```

```
/* A switch to another task cannot occur between the call to taskENTER_CRITICAL() and the
call to taskEXIT_CRITICAL(). Interrupts might still execute on FreeRTOS ports that allow
interrupt nesting, but only interrupts whose logical priority is above the value assigned
to the configMAX_SYSCALL_INTERRUPT_PRIORITY constant. Those interrupts are not permitted
to call FreeRTOS API functions. */

PORTA |= 0x01;

/* Access to PORTA has finished, so it is safe to exit the critical section. */

taskEXIT_CRITICAL();
```

일부 예제는 vPrintString() 함수를 사용해 문자열을 표준 출력(standard out)에 작성합니다. 여기에서 표준 출력이란 FreeRTOS Windows 포트 사용 시 터미널 창을 말합니다. vPrintString()은 여러 가지 다른 작업에서 호출되기 때문에 이론적으로는 이 함수의 구현체가 다음과 같이 임계 영역을 사용해 표준 출력에 대한 액세스를 보호할 수 있습니다.

```
void vPrintString( const char *pcString )
{
    /* Write the string to stdout, using a critical section as a crude method of mutual
    exclusion. */

    taskENTER_CRITICAL();

    {
        printf( "%s", pcString );

        fflush( stdout );
    }

    taskEXIT_CRITICAL();
}
```

이렇게 구현되는 임계 영역은 상호 배제 기법으로서 매우 조잡합니다. 사용하는 FreeRTOS 포트에 따라 완전히, 혹은 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY에서 설정하는 우선순위까지 인터럽트를 비활성화하여 구현되기 때문입니다. 선제적 컨텍스트 전환은 인터럽트 내부에서만 발생할 수 있기 때문에 인터럽트가 비활성화되어 있는 동안에는 taskENTER\_CRITICAL()을 호출한 작업이 임계 영역이 종료될 때까지 Running 상태를 유지할 수 있습니다.

기본 임계 영역은 매우 짧게 작성되어야 합니다. 그렇지 않으면 인터럽트 응답 시간에 부정적인 영향을 미칩니다. taskENTER\_CRITICAL() 호출은 모두 taskEXIT\_CRITICAL() 호출과 밀접하게 묶여있어야 합니다. 이러한 이유로 위 코드와 같이 임계 영역을 사용해 표준 출력(stdout, 즉 컴퓨터가 출력 데이터를 작성하는 스트림)을 보호해서는 안 됩니다. 터미널에 대한 코드 작성은 비교적 긴 작업이 될 수 있기 때문입니다. 이번 단원의 예제에서는 이를 대체할 수 있는 해결책을 살펴보겠습니다.

임계 영역은 중첩시켜 사용해도 안전합니다. 커널이 중첩 깊이를 계산하기 때문입니다. 임계 영역은 중첩 깊이가 0으로 돌아갈 때만 종료됩니다. 여기에서 중첩 깊이가 0이란 말은 앞서 호출된 모든 taskENTER\_CRITICAL()에 대해 taskEXIT\_CRITICAL() 호출이 1회 실행되었을 때를 말합니다.

taskENTER\_CRITICAL() 및 taskEXIT\_CRITICAL() 호출은 작업이 FreeRTOS가 실행되는 프로세서에서 인터럽트 활성화 상태를 정당하게 변경할 수 있는 유일한 방법입니다. 다른 방법으로 인터럽트 활성화 상태를 변경하면 매크로의 중첩 수가 무효화됩니다.

taskENTER\_CRITICAL()과 taskEXIT\_CRITICAL()은 'FromISR'로 끝나지 않기 때문에 인터럽트 서비스 루틴에서 호출해서는 안 됩니다. taskENTER\_CRITICAL\_FROM\_ISR()이 taskENTER\_CRITICAL()의 인터럽트 세이프 버전이고, taskEXIT\_CRITICAL\_FROM\_ISR()이 taskEXIT\_CRITICAL()의 인터럽트 세이프 버전입니다.

다. 인터럽트 세이프 버전은 인터럽트 중첩이 허용되는 FreeRTOS 포트에게만 제공됩니다. 인터럽트 중첩이 허용되지 않는 포트에서는 사용하지 못합니다.

taskENTER\_CRITICAL\_FROM\_ISR()은 다음과 같이 쌍을 이루는 taskEXIT\_CRITICAL\_FROM\_ISR() 호출로 전달해야 하는 값을 반환합니다.

```
void vAnInterruptServiceRoutine( void )
{
    /* Declare a variable in which the return value from taskENTER_CRITICAL_FROM_ISR() will
    be saved. */
    UBaseType_t uxSavedInterruptStatus;

    /* This part of the ISR can be interrupted by any higher priority interrupt. */

    /* Use taskENTER_CRITICAL_FROM_ISR() to protect a region of this ISR. Save the value
    returned from taskENTER_CRITICAL_FROM_ISR() so it can be passed into the matching call to
    taskEXIT_CRITICAL_FROM_ISR(). */
    uxSavedInterruptStatus = taskENTER_CRITICAL_FROM_ISR();

    /* This part of the ISR is between the call to taskENTER_CRITICAL_FROM_ISR() and
    taskEXIT_CRITICAL_FROM_ISR(), so can only be interrupted by interrupts that have a
    priority above that set by the configMAX_SYSCALL_INTERRUPT_PRIORITY constant. */

    /* Exit the critical section again by calling taskEXIT_CRITICAL_FROM_ISR(), passing in
    the value returned by the matching call to taskENTER_CRITICAL_FROM_ISR(). */
    taskEXIT_CRITICAL_FROM_ISR( uxSavedInterruptStatus );

    /* This part of the ISR can be interrupted by any higher priority interrupt. */
}
```

임계 영역을 시작하고 종료하는 코드를 실행하면서 임계 영역에서 보호하는 코드보다 더욱 많은 처리 시간을 사용하는 것은 소모적입니다. 기본 임계 영역은 시작부터 종료까지 매우 빠를 뿐만 아니라 항상 결정적이기 때문에 보호할 코드 구간이 매우 짧을 때 사용하는 것이 효과적입니다.

## 스케줄러의 일시 중지(잠금)

임계 영역은 스케줄러를 일시 중지하여 생성하는 방법도 있습니다. 스케줄러 일시 중지는 간혹 스케줄러 잠금이라고도 불립니다.

기본 임계 영역은 다른 작업이나 인터럽트가 액세스하지 못하도록 코드 구간을 보호합니다. 하지만 스케줄러를 일시 중지하여 임계 영역을 구현할 경우에는 인터럽트가 활성화 상태를 유지하기 때문에 다른 작업이 액세스하지 못하도록 코드 구간을 보호할 뿐입니다.

인터럽트를 비활성화하면 너무 길어져서 임계 영역을 구현할 수 없을 때는 스케줄러를 일시 중지하여 구현할 수 있습니다. 하지만 스케줄러가 일시 중지된 동안에도 인터럽트가 활성화되기 때문에 스케줄러를 다시 시작하려면(또는 일시 중지 해제하려면) 비교적 긴 작업이 될 수 있습니다. 지금부터는 각 사례에서 가장 효과적으로 사용할 수 있는 방법에 대해 생각해볼 것입니다.

## vTaskSuspendAll() API 함수

vTaskSuspendAll() API 함수 프로토타입은 다음과 같습니다.

```
void vTaskSuspendAll( void );
```

vTaskSuspendAll()을 호출하여 스케줄러가 일시 중지됩니다. 스케줄러가 일시 중지되면 컨텍스트 전환이 일어나지 않지만 인터럽트는 활성화된 상태를 유지합니다. 스케줄러가 일시 중지되었을 때 인터럽트가 컨텍스트 전환을 요청하면 요청이 잠시 보류되었다가 스케줄러가 다시 시작되었을 때(일시 중지가 해제되었을 때) 처리됩니다.

스케줄러가 일시 중지되었을 때 FreeRTOS API 함수를 호출해서는 안 됩니다.

## xTaskResumeAll() API 함수

xTaskResumeAll() API 함수 프로토타입은 다음과 같습니다.

```
BaseType_t xTaskResumeAll( void );
```

xTaskResumeAll()을 호출하여 스케줄러가 다시 시작됩니다(일시 중지가 해제됩니다).

다음 표는 xTaskResumeAll() 반환 값을 나열한 것입니다.

반환 값	설명
반환 값	스케줄러가 일시 중지되었을 때 발생하는 컨텍스트 전환 요청은 잠시 보류되었다가 스케줄러가 다시 시작되었을 때 처리됩니다. 보류 중인 컨텍스트 전환 요청이 xTaskResumeAll()의 반환 이전에 처리되면 pdTRUE가 반환됩니다. 그 밖의 경우에는 pdFALSE가 반환됩니다.

vTaskSuspendAll() 및 xTaskResumeAll() 호출은 중첩시켜 사용해도 안전합니다. 커널이 중첩 깊이를 계산하기 때문입니다. 스케줄러는 중첩 깊이가 0으로 돌아갈 때만 다시 시작됩니다. 여기에서 중첩 깊이가 0이란 말은 앞서 호출된 모든 vTaskSuspendAll()에 대해 xTaskResumeAll() 호출이 1회 실행되었을 때를 말합니다.

다음 코드는 스케줄러를 일시 중지하여 터미널 출력에 대한 액세스를 보호하는 vPrintString()의 구현체를 나타낸 것입니다.

```
void vPrintString( const char *pcString )
{
    /* Write the string to stdout, suspending the scheduler as a method of mutual
    exclusion. */

    vTaskSuspendScheduler();

    {
        printf( "%s", pcString );

        fflush( stdout );
    }

    xTaskResumeScheduler();
}
```

## 뮤텍스(및 이진 세마포어)

뮤텍스(MUTual EXclusion)는 특별한 형식의 이진 세마포어로서 여러 작업의 공유 리소스에 대한 액세스를 제어하는 데 사용됩니다. 뮤텍스를 사용하려면 FreeRTOSConfig.h에서 configUSE\_MUTEXES를 1로 설정해야 합니다.

상호 배제 시나리오에서 뮤텍스를 사용할 경우에는 뮤텍스가 공유 리소스와 연결되는 토큰이라고 생각해도 좋습니다. 즉, 작업이 리소스에 정당하게 액세스하려면 먼저 토큰을 가져와야 합니다(토큰 홀더가 되어야 합니다). 토큰 홀더가 리소스 사용을 마치면 토큰을 다시 반환해야 합니다. 토큰이 반환된 이후에만 다른 작업이 토큰을 가져와 동일한 공유 리소스에 안전하게 액세스할 수 있습니다. 토큰이 없는 작업은 공유 리소스에 액세스하지 못합니다.

뮤텍스와 이진 세마포어가 여러 가지 특성을 공유하기는 하지만 뮤텍스가 상호 배제에 사용되는 시나리오는 이진 세마포어가 동기화에 사용되는 시나리오와 완전히 다릅니다. 기본적인 차이점은 세마포어를 가져온 이후의 상황에 있습니다.

- 상호 배제에 사용된 세마포어는 반드시 복귀되어야 합니다.
- 동기화에 사용된 세마포어는 일반적으로 삭제되기 때문에 복귀되지 않습니다.

이러한 메커니즘은 순전히 애플리케이션 개발자의 통제에 따라 이루어집니다. 작업이 언제든지 리소스에 액세스하지 못할 이유는 없지만 각 작업은 뮤텍스 홀더가 될 수 없다면 리소스에 액세스하지 않는 것으로 약속되어 있습니다.

## xSemaphoreCreateMutex() API 함수

FreeRTOS V9.0.0에도 컴파일 과정에서 뮤텍스를 정적으로 생성하는 데 필요한 메모리를 할당하는 xSemaphoreCreateMutexStatic() 함수가 포함되어 있습니다. 뮤텍스는 세마포어의 한 가지 유형입니다. 여러 가지 유형의 모든 FreeRTOS 세마포어 핸들은 SemaphoreHandle\_t 형식의 변수로 저장됩니다.

뮤텍스를 사용하려면 먼저 생성해야 합니다. 뮤텍스 유형의 세마포어를 생성하려면 xSemaphoreCreateMutex() API 함수를 사용하십시오.

xSemaphoreCreateMutex() API 함수 프로토타입은 다음과 같습니다.

```
SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

다음 표는 xSemaphoreCreateMutex() 반환 값을 나열한 것입니다.

## 세마포어를 사용하기 위한 vPrintString() 재작성(예제 20)

이번 예제에서는 vPrintString()의 새로운 버전인 prvNewPrintString()을 생성한 후 새롭게 생성된 함수를 다수의 작업에서 호출합니다. prvNewPrintString()은 vPrintString()과 기능이 동일하지만 스케줄러를 일시 중지하지 않고 뮤텍스를 사용하여 표준 출력에 대한 액세스를 제어합니다.

prvNewPrintString()의 구현체는 다음과 같습니다.

```
static void prvNewPrintString( const char *pcString )
{
    /* The mutex is created before the scheduler is started, so already exists by the time
    this task executes. Attempt to take the mutex, blocking indefinitely to wait for the mutex
```

```
if it is not available right away. The call to xSemaphoreTake() will only return when the
mutex has been successfully obtained, so there is no need to check the function return
value. If any other delay period was used, then the code must check that xSemaphoreTake()
returns pdTRUE before accessing the shared resource (which in this case is standard out).
Indefinite timeouts are not recommended for production code. */

xSemaphoreTake( xMutex, portMAX_DELAY );

{

    /* The following line will only execute after the mutex has been successfully
    obtained. Standard out can be accessed freely now because only one task can have the mutex
    at any one time. */

    printf( "%s", pcString );

    fflush( stdout );

    /* The mutex MUST be given back! */

}

xSemaphoreGive( xMutex );

}
```

prvNewPrintString()은 prvPrintTask()에서 구현된 작업 인스턴스 2개에서 반복적으로 호출됩니다. 각 호출 사이에는 지연 시간이 무작위로 사용되었습니다. 작업 파라미터는 각 작업 인스턴스에 고유 문자열을 전달하는데 사용되었습니다. prvPrintTask()의 구현체는 다음과 같습니다.

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

    const TickType_t xMaxBlockTimeTicks = 0x20;

    /* Two instances of this task are created. The string printed by the task is passed
    into the task using the task's parameter. The parameter is cast to the required type. */

    pcStringToPrint = ( char * ) pvParameters;

    for( ;; )
    {
        /* Print out the string using the newly defined function. */

        prvNewPrintString( pcStringToPrint );

        /* Wait a pseudo random time. Note that rand() is not necessarily reentrant,
        but in this case it does not really matter because the code does not care what value is
        returned. In a more secure application, a version of rand() that is known to be reentrant
        should be used or calls to rand() should be protected using a critical section. */

        vTaskDelay( ( rand() % xMaxBlockTimeTicks ) );
    }
}
```

평소와 같이 main()이 뮤텍스와 작업을 먼저 생성한 후 스케줄러를 시작합니다.

prvPrintTask() 인스턴스 2개는 다른 우선순위로 생성되기 때문에 낮은 우선순위의 작업이 종종 높은 우선순위의 작업에게 선점되기도 합니다. 뮤텍스는 각 작업이 상호 배제 방식으로 터미널에 액세스할 수 있도록 사용되기 때문에 선점(preemption)이 발생하더라도 문자열이 정확하게 표시될 뿐만 아니라 손상되지 않습니다. xMaxBlockTimeTicks 상수에서 작업이 Blocked 상태로 대기하는 최대 시간을 줄여서 설정하면 선점 주기를 늘릴 수 있습니다.

예제 20을 FreeRTOS Windows 포트에 사용할 때 주의 사항:

- printf()를 호출하면 Windows 시스템 호출이 생성됩니다. Windows 시스템 호출은 FreeRTOS에서 제어하지 못해 불안정을 초래할 수 있습니다.
- Windows 시스템 호출의 실행 방식은 뮤텍스를 사용하지 않을 때조차 손상된 문자열을 거의 보기 어렵다는 것을 의미합니다.

```
int main( void )
{
    /* Before a semaphore is used it must be explicitly created. In this example, a mutex
    type semaphore is created. */

    xMutex = xSemaphoreCreateMutex();

    /* Check that the semaphore was created successfully before creating the tasks. */

    if( xMutex != NULL )
    {
        /* Create two instances of the tasks that write to stdout. The string they write
        is passed in to the task as the task's parameter. The tasks are created at different
        priorities so some preemption will occur. */

        xTaskCreate( prvPrintTask, "Print1", 1000, "Task 1
        *****\r\n", 1, NULL );

        xTaskCreate( prvPrintTask, "Print2", 1000, "Task 2
        -----\r\n", 2, NULL );

        /* Start the scheduler so the created tasks start executing. */

        vTaskStartScheduler();

    }

    /* If all is well, then main() will never reach here because the scheduler will
    now be running the tasks. If main() does reach here, then it is likely that there was
    insufficient heap memory available for the idle task to be created. */

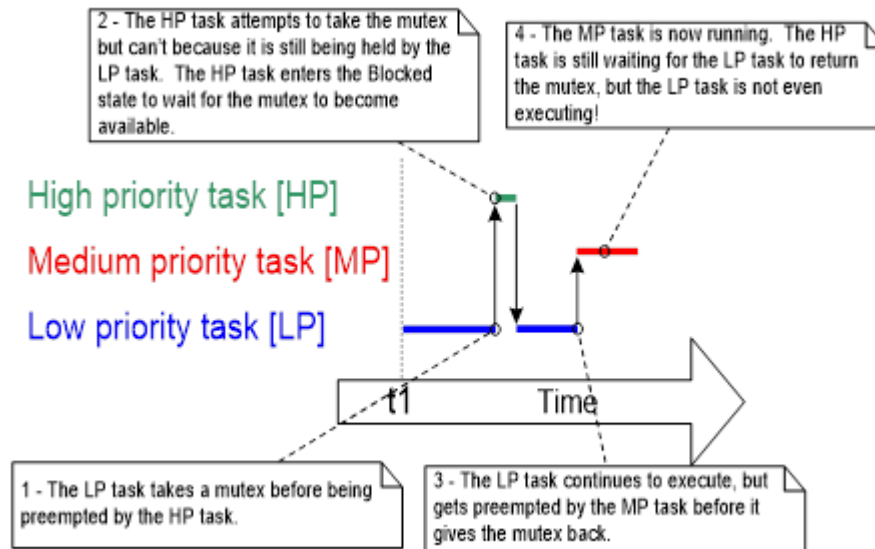
    for( ;; );
}
```

출력되는 화면은 다음과 같습니다.





위 역전이라고 부릅니다. 만약 높은 우선순위의 작업이 세마포어를 사용할 수 있을 때까지 대기하는 도중에 중간 우선순위의 작업까지 실행된다면 이렇게 불합리한 동작은 더욱 악화될 수 있습니다. 결국 낮은 우선순위의 작업은 실행조차 못한 채 높은 우선순위의 작업이 낮은 우선순위의 작업에 뒤처져 대기하는 결과를 초래할 뿐입니다. 아래 그림은 최악의 시나리오를 나타낸 것입니다.

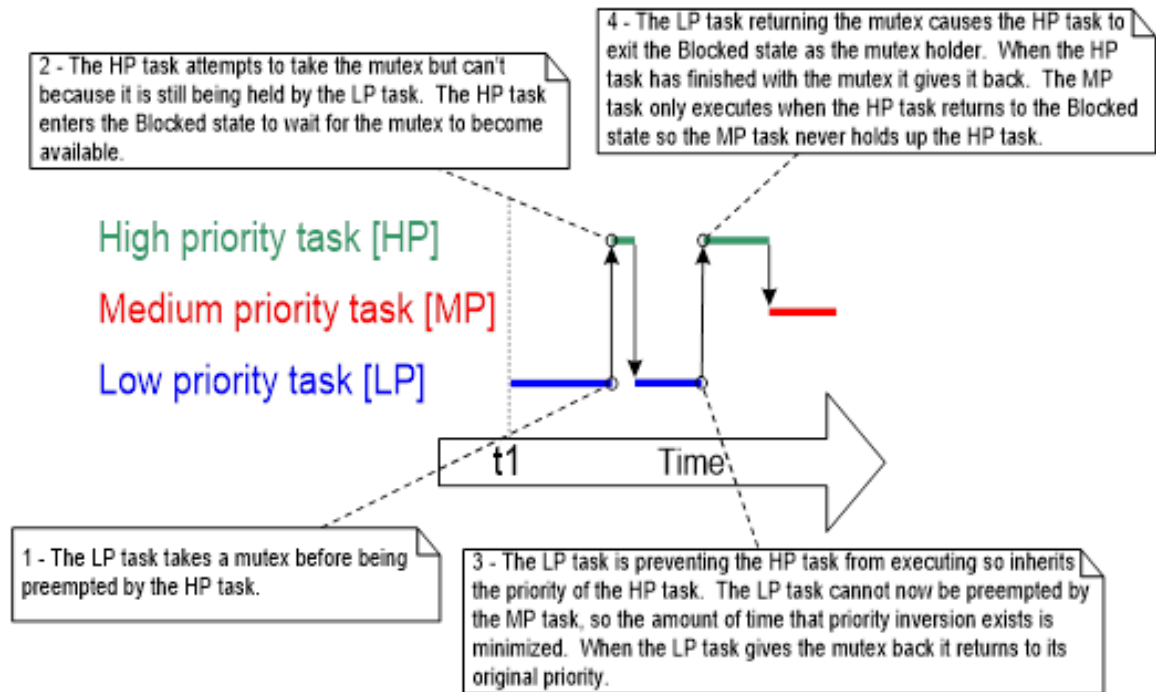


우선순위 역전은 중요한 문제가 될 수 있지만 소형 임베디드 시스템에서는 리소스 액세스 방식에 주의하여 시스템을 설계하면 이러한 문제를 피할 수 있습니다.

## 우선순위 상속

FreeRTOS 뮤텍스와 이진 세마포어는 매우 비슷합니다. 다른 점은 뮤텍스에는 기본적인 우선순위 상속 메커니즘이 적용되는 반면 이진 세마포어에는 그렇지 않다는 것입니다. 우선순위 상속은 우선순위 역전이 미치는 부정적인 영향을 최소화하기 위한 방법입니다. 이 방법은 우선순위 역전을 해결하지는 못하지만 역전되는 시간을 항상 제한함으로써 영향을 줄입니다. 단, 우선순위 상속은 시스템 시간 분석을 복잡하게 하는 단점도 있습니다. 따라서 정확한 시스템 작동이 중요할 때 우선순위 상속에 의지하는 것은 바람직하지 않습니다.

우선순위 상속은 뮤텍스 홀더의 우선순위를 동일한 뮤텍스를 가져오려고 하는 최고 우선순위 작업의 우선순위까지 일시적으로 높여서 이루어집니다. 다시 말해서 뮤텍스를 저장하고 있는 낮은 우선순위의 작업이 뮤텍스를 사용할 수 있을 때까지 대기하고 있는 작업의 우선순위를 상속합니다. 다음 그림은 뮤텍스 홀더가 뮤텍스를 반환할 때 홀더의 우선순위가 최초 값으로 자동 재설정되는 것을 나타냅니다.



우선순위 상속 기능은 뮉텍스를 사용하는 작업의 우선순위에 영향을 미칩니다. 뮉텍스를 인터럽트 서비스 루틴에서 사용해서는 안 되는 이유도 바로 여기에 있습니다.

## 교착 상태(또는 치명적인 포옹)

교착 상태 또는 치명적인 포옹은 뮉텍스를 상호 배제에 사용할 때 발생할 수 있는 또 한 가지 문제입니다.

교착 상태는 작업 2개가 나머지 작업에 저장된 리소스를 사용할 수 있을 때까지 대기하여 더 이상 진행되지 않을 때 발생합니다. 다음과 같이 Task A와 Task B가 작업을 실행하려면 뮉텍스 X와 뮉텍스 Y가 필요한 시나리오를 예로 들겠습니다.

1. Task A가 실행되면서 뮉텍스 X를 성공적으로 가져옵니다.
2. Task A가 Task B에게 선점됩니다.
3. Task B는 뮉텍스 X를 가져오기 전에 뮉텍스 Y를 가져오지만 뮉텍스 X가 Task A에 저장되어 있기 때문에 Task B가 뮉텍스 X를 사용하지는 못합니다. 이에 따라 Task B는 뮉텍스 X가 해제될 때까지 Blocked 상태로 전환됩니다.
4. Task A가 계속해서 실행되면서 뮉텍스 Y를 가져오려고 하지만 뮉텍스 Y는 Task B에 저장되어 있기 때문에 Task A가 뮉텍스 Y를 사용하지 못합니다. 이에 따라 Task A는 뮉텍스 Y가 해제될 때까지 Blocked 상태로 전환됩니다.

Task A는 Task B에 저장된 뮉텍스가 해제될 때까지 대기하고, Task B는 Task A에 저장된 뮉텍스가 해제될 때까지 대기하면서 결국 두 작업 모두 실행하지 못하는 교착 상태에 빠지고 맙니다.

우선순위 역전과 마찬가지로 교착 상태를 피할 수 있는 가장 좋은 방법은 교착 상태가 발생할 수 없도록 시스템을 설계하는 것입니다. 일반적으로 작업이 뮉텍스를 가져올 때까지 제한 시간 없이 무기한 대기하는 것은 좋은 방법이 아닙니다. 대신 뮉텍스를 가져올 때까지 예상되는 최대 대기 시간보다 약간 더 길게 제한 시간을 지정하십시오. 지정된 시간 내에 뮉텍스를 가져오지 못하면 설계 오류로 인한 교착 상태의 징후가 될 수 있습니다.

교착 상태는 소형 임베디드 시스템에게 큰 문제가 아닙니다. 전체 애플리케이션에 대해 잘 알고 있는 시스템 설계자가 있다면 교착 상태가 발생할 수 있는 영역을 찾아내 삭제할 수 있기 때문입니다.

## 재귀적 뮤텍스

작업이 스스로 교착 상태에 빠지게 하는 것도 가능합니다. 작업이 뮤텍스를 먼저 반환하지도 않고 동일한 뮤텍스를 여러 차례 가져오려고 시도하면 교착 상태에 빠집니다. 다음 시나리오를 예로 들겠습니다.

1. 작업이 뮤텍스를 성공적으로 가져옵니다.
2. 뮤텍스가 저장되어 있을 때 작업이 라이브러리 함수를 호출합니다.
3. 라이브러리 함수의 구현체가 동일한 뮤텍스를 가져오려고 시도하지만 Blocked 상태로 전환되어 뮤텍스를 사용할 수 있을 때까지 대기합니다.

작업은 뮤텍스가 반환될 때까지 Blocked 상태로 대기하지만 이미 뮤텍스 홀더가 된 후입니다. 작업이 Blocked 상태에서 자기 자신을 위해 대기하면서 결국 교착 상태에 빠지고 맙니다.

이러한 유형의 교착 상태는 표준 뮤텍스가 아닌 재귀적 뮤텍스를 사용하여 방지할 수 있습니다. 재귀적 뮤텍스는 동일한 작업라고 해도 여러 차례 가져올 수 있습니다. 앞서 재귀적 뮤텍스를 가져오기 위한 모든 호출에 대해 재귀적 뮤텍스를 반환하는 호출이 1회 실행된 후에만 반환됩니다.

표준 뮤텍스와 재귀적 뮤텍스는 다음과 같이 생성 및 사용하는 방법이 비슷합니다.

- 표준 뮤텍스는 `xSemaphoreCreateMutex()`를 사용하여 생성됩니다. 재귀적 뮤텍스는 `xSemaphoreCreateRecursiveMutex()`를 사용하여 생성됩니다. 두 API 함수는 프로토타입이 동일합니다.
- 표준 뮤텍스는 `xSemaphoreTake()`를 사용하여 가져옵니다. 재귀적 뮤텍스는 `xSemaphoreTakeRecursive()`를 사용하여 가져옵니다. 두 API 함수는 프로토타입이 동일합니다.
- 표준 뮤텍스는 `xSemaphoreGive()`를 사용하여 반환됩니다. 재귀적 뮤텍스는 `xSemaphoreGiveRecursive()`를 사용하여 반환됩니다. 두 API 함수는 프로토타입이 동일합니다.

다음 코드는 재귀적 뮤텍스를 생성 및 사용하는 방법을 나타낸 것입니다.

```
/* Recursive mutexes are variables of type SemaphoreHandle_t. */  
  
SemaphoreHandle_t xRecursiveMutex;  
  
/* The implementation of a task that creates and uses a recursive mutex. */  
  
void vTaskFunction( void *pvParameters )  
{  
  
    const TickType_t xMaxBlock20ms = pdMS_TO_TICKS( 20 );  
  
    /* Before a recursive mutex is used it must be explicitly created. */  
  
    xRecursiveMutex = xSemaphoreCreateRecursiveMutex();  
  
    /* Check the semaphore was created successfully. configASSERT() is described in section  
    11.2. */  
  
    configASSERT( xRecursiveMutex );  
  
    /* As per most tasks, this task is implemented as an infinite loop. */  
  
    for( ;; )
```

```

{
    /* ... */

    /* Take the recursive mutex. */
    if( xSemaphoreTakeRecursive( xRecursiveMutex, xMaxBlock20ms ) == pdPASS)
    {
        /* The recursive mutex was successfully obtained. The task can now access the
        resource the mutex is protecting. At this point the recursive call count (which is the
        number of nested calls to xSemaphoreTakeRecursive()) is 1 because the recursive mutex has
        only been taken once. */

        /* While it already holds the recursive mutex, the task takes the mutex
        again. In a real application, this is only likely to occur inside a subfunction called
        by this task because there is no practical reason to knowingly take the same mutex
        more than once. The calling task is already the mutex holder, so the second call to
        xSemaphoreTakeRecursive() does nothing more than increment the recursive call count to 2.
        */

        xSemaphoreTakeRecursive( xRecursiveMutex, xMaxBlock20ms );

        /* ... */

        /* The task returns the mutex after it has finished accessing the resource the
        mutex is protecting. At this point the recursive call count is 2, so the first call to
        xSemaphoreGiveRecursive() does not return the mutex. Instead, it simply decrements the
        recursive call count back to 1. */

        xSemaphoreGiveRecursive( xRecursiveMutex );

        /* The next call to xSemaphoreGiveRecursive() decrements the recursive call
        count to 0, so this time the recursive mutex is returned.*/

        xSemaphoreGiveRecursive( xRecursiveMutex );

        /* Now one call to xSemaphoreGiveRecursive() has been executed for every
        proceeding call to xSemaphoreTakeRecursive(), so the task is no longer the mutex holder.
        */
    }
}

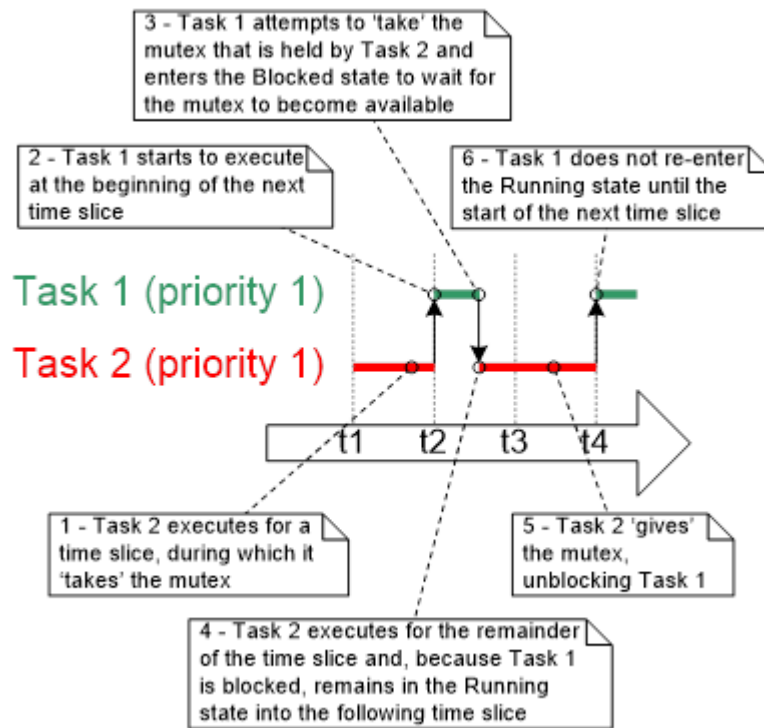
```

## 뮤텍스 및 작업 스케줄링

우선순위가 다른 작업 2개가 동일한 뮤텍스를 사용하는 경우 작업의 실행 순서는 FreeRTOS 스케줄링 정책을 따릅니다. 즉 실행 가능한 작업 중에서 가장 높은 우선순위의 작업이 선택되어 Running 상태로 전환됩니다. 예를 들어 높은 우선순위의 작업이 Blocked 상태에서 낮은 우선순위의 작업에 저장된 뮤텍스를 기다리고 있다면 낮은 우선순위의 작업이 뮤텍스를 반환하자마자 높은 우선순위의 작업이 낮은 우선순위의 작업보다 먼저 실행됩니다. 이후 높은 우선순위의 작업은 뮤텍스 홀더가 됩니다. 이러한 시나리오는 [우선순위 상속 \(p. 162\)](#) 단원에서 이미 다루었습니다.

작업의 우선순위가 동일할 때는 작업 실행 순서를 잘못 생각하는 경우가 많습니다. Task 1과 Task 2의 우선순위가 동일하고, Task 1이 Blocked 상태에서 Task 2에 저장된 뮤텍스가 반환될 때까지 대기하는 경우 Task 1은 Task 2가 뮤텍스를 반환할 때까지 Task 2를 선점하지 않습니다. 대신에 Task 2는 Running 상태를 유지합니다. Task 1은 Blocked 상태에서 Ready 상태로 바뀔 뿐입니다.

아래 그림에서 수직선은 틱 인터럽트가 발생하는 시간을 나타냅니다.



위 그림에서 FreeRTOS 스케줄러는 뮉텍스를 사용할 수 있게 되어도 다음과 같은 이유로 Task 1을 Running 상태의 작업으로 바로 전환하지 않습니다.

1. Task 1과 Task 2의 우선순위가 동일하기 때문에 Task 2가 Blocked 상태로 전환되지만 않으면 다음 틱 인터럽트까지 Task 1로 전환되어서는 안 됩니다(FreeRTOSConfig.h에서 configUSE\_TIME\_SLICING이 1로 설정되어 있다는 가정을 전제로 함).
2. 작업이 짧은 주기의 루프에서 뮉텍스를 사용하는 동시에 뮉텍스를 반환할 때마다 컨텍스트 전환이 일어났다면 작업이 Running 상태를 유지하는 시간이 매우 짧습니다. 작업 2개 이상이 촘촘한 루프에서 동일한 뮉텍스를 사용할 경우에는 작업이 빠르게 전환되어 처리 시간을 낭비하게 됩니다.

다수의 작업이 짧은 주기의 루프에서 뮉텍스를 사용하는 동시에 뮉텍스를 사용하는 작업들의 우선순위가 동일하다면 처리 시간이 각 작업에게 거의 동일하게 할당되어야 합니다. 위 그림은 아래 코드의 작업 인스턴스 2개가 동일한 우선순위로 생성되는 경우 발생할 수 있는 실행 시퀀스를 나타낸 것입니다.

```
/* The implementation of a task that uses a mutex in a tight loop. The task creates a text
string in a local buffer, and then writes the string to a display. Access to the display
is protected by a mutex. */
```

```
void vATask( void *pvParameter )
{
    extern SemaphoreHandle_t xMutex;

    char cTextBuffer[ 128 ];

    for( ;; )
    {
```

```

/* Generate the text string. This is a fast operation. */
vGenerateTextInALocalBuffer( cTextBuffer );

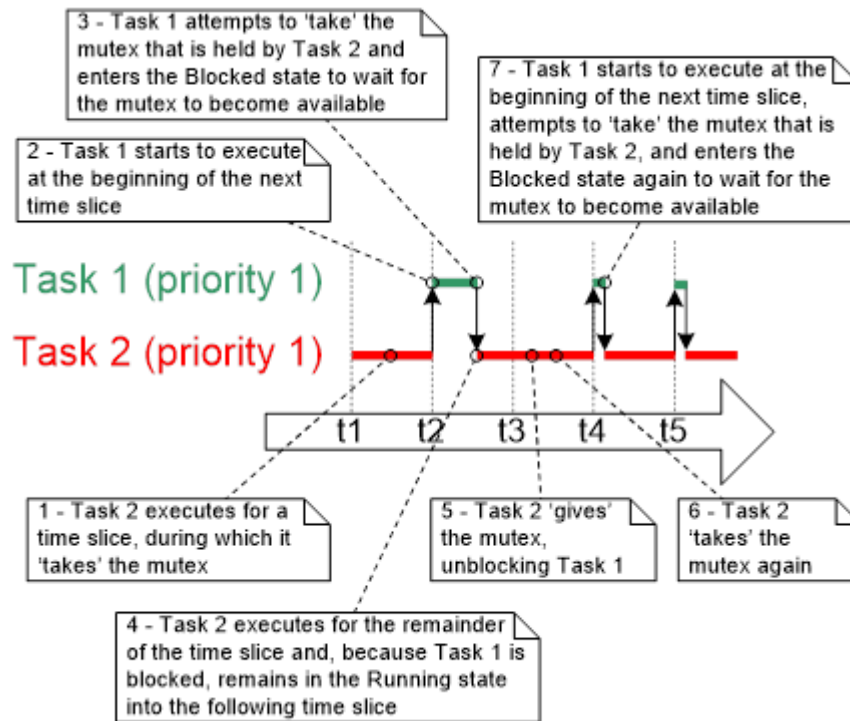
/* Obtain the mutex that is protecting access to the display. */
xSemaphoreTake( xMutex, portMAX_DELAY );

/* Write the generated text to the display. This is a slow operation. */
vCopyTextToFrameBuffer( cTextBuffer );

/* The text has been written to the display, so return the mutex. */
xSemaphoreGive( xMutex );
    }
}
    
```

위 코드의 주석은 문자열 생성은 빠른 작업이고, 디스플레이 업데이트는 느린 작업인 것을 나타냅니다. 따라서 뮤텍스는 디스플레이가 업데이트될 때 저장된 후 런타임 대부분을 작업에 저장됩니다.

아래 그림에서 수직선은 틱 인터럽트가 발생하는 시간을 나타냅니다.



위 그림에서 7단계는 Task 1이 다시 Blocked 상태로 전환되는 것을 나타냅니다. 이는 xSemaphoreTake() API 함수 내에서 일어납니다.

Task 1은 시간 분할의 시작이 Task 2가 뮤텍스 홀더가 아닌 짧은 주기 중 하나와 일치할 때까지 뮤텍스를 가져오지 못합니다.

이러한 시나리오는 xSemaphoreGive() 호출 이후에 taskYIELD() 호출을 추가하여 피할 수 있습니다. 이 방법은 아래 코드에 나와있습니다. 여기에서는 작업에 뮤텍스가 저장되어 있는 동안 틱 카운트가 바뀌면

taskYIELD()가 호출됩니다. 이러한 코드는 작업을 지나치게 빠르게 전환하여 처리 시간을 낭비하지 않는 동시에 루프에서 뮤텝스를 사용하는 작업에게 처리 시간을 더욱 균일하게 할당할 수 있습니다.

```
void vFunction( void *pvParameter )
{
    extern SemaphoreHandle_t xMutex;

    char cTextBuffer[ 128 ];

    TickType_t xTimeAtWhichMutexWasTaken;

    for( ;; )
    {
        /* Generate the text string. This is a fast operation. */
        vGenerateTextInALocalBuffer( cTextBuffer );

        /* Obtain the mutex that is protecting access to the display. */
        xSemaphoreTake( xMutex, portMAX_DELAY );

        /* Record the time at which the mutex was taken. */
        xTimeAtWhichMutexWasTaken = xTaskGetTickCount();

        /* Write the generated text to the display. This is a slow operation. */
        vCopyTextToFrameBuffer( cTextBuffer );

        /* The text has been written to the display, so return the mutex. */
        xSemaphoreGive( xMutex );

        /* If taskYIELD() was called on each iteration, then this task would only ever
        remain in the Running state for a short period of time, and processing time would be
        wasted by rapidly switching between tasks. Therefore, only call taskYIELD() if the tick
        count changed while the mutex was held. */

        if( xTaskGetTickCount() != xTimeAtWhichMutexWasTaken )
        {
            taskYIELD();
        }
    }
}
```

## 게이트키퍼 작업

게이트키퍼 작업은 우선순위 역전이나 교착 상태의 위험 없이 상호 배제를 완벽하게 구현할 수 있는 방법을 제공합니다.

게이트키퍼 작업란 리소스를 단독으로 소유하는 작업을 말합니다. 게이트키퍼 작업만 이 리소스에 직접 액세스할 수 있습니다. 리소스에 액세스해야 하는 다른 작업은 게이트키퍼 서비스를 이용해야만 간접적으로 액세스할 수 있습니다.

## 게이트키퍼 작업을 사용하기 위한 vPrintString() 재작성(예제 21)

이번 예제에서는 vPrintString()을 대체할 수 있는 또 다른 구현체에 대해서 설명합니다. 이번에는 게이트키퍼 작업을 사용해 표준 출력에 대한 액세스를 관리합니다. 작업이 표준 출력에 메시지를 작성할 때 출력 함수를 직접 호출하지 않습니다. 대신에 메시지를 게이트키퍼에게 전송합니다.

게이트키퍼 작업은 FreeRTOS 대기열을 사용해 표준 출력에 대한 액세스를 직렬화합니다. 작업을 내부에서 구현할 때는 표준 출력에 대한 직접 액세스가 작업에게만 허용되기 때문에 상호 배제를 고려할 필요가 없습니다.

게이트키퍼 작업은 대부분 시간을 Blocked 상태로 보내면서 메시지가 대기열에 도착할 때까지 대기합니다. 메시지가 도착하면 게이트키퍼가 메시지를 표준 출력에 작성할 뿐 이후에는 다시 Blocked 상태로 돌아가 다음 메시지를 기다립니다.

인터럽트는 대기열 전송이 가능하기 때문에 인터럽트 서비스 루틴 역시 게이트키퍼 서비스를 안전하게 사용하여 메시지를 터미널에 작성할 수 있습니다. 이번 예제에서 틱 후크 함수는 200틱마다 메시지를 작성하는데 사용됩니다.

틱 후크(또는 틱 콜백)란 틱 인터럽트마다 커널에서 호출되는 함수를 말합니다. 틱 후크 함수를 사용하는 방법은 다음과 같습니다.

1. FreeRTOSConfig.h에서 configUSE\_TICK\_HOOK을 1로 설정합니다.
2. 아래와 같이 정확한 함수 이름과 프로토타입을 사용해 후크 함수의 구현체를 입력합니다.

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

틱 후크 함수는 틱 인터럽트의 컨텍스트에서 실행되기 때문에 매우 짧게 작성되어야 합니다. 또한 스택 공간은 알맞은 크기만 사용해야 하고, 'FromISR()'로 끝나지 않는 FreeRTOS API 함수를 호출해서는 안 됩니다.

게이트키퍼 작업의 구현체는 다음과 같습니다. 스케줄러는 항상 틱 후크 함수에 바로 이어서 실행됩니다. 따라서 틱 후크에서 호출되는 인터럽트 세이프 FreeRTOS API 함수가 pxHigherPriorityTaskWoken 파라미터를 사용할 필요는 없기 때문에, 이 파라미터는 NULL로 설정해도 좋습니다.

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

    /* This is the only task that is allowed to write to standard out. Any other task
    wanting to write a string to the output does not access standard out directly, but
    instead sends the string to this task. Because this is the only task that accesses
    standard out, there are no mutual exclusion or serialization issues to consider within the
    implementation of the task itself. */

    for( ;; )
    {
        /* Wait for a message to arrive. An indefinite block time is specified so there is
        no need to check the return value. The function will return only when a message has been
        successfully received. */

        xQueueReceive( xPrintQueue, &pcMessageToPrint, portMAX_DELAY );

        /* Output the received string. */
    }
}
```



```
printf( "%s", pcMessageToPrint );

fflush( stdout );

/* Loop back to wait for the next message. */

}

}
```

대기열에 작성되는 작업은 다음과 같습니다. 이전과 마찬가지로 작업 인스턴스 2개가 따로 생성되고, 작업이 대기열에 작성하는 문자열이 작업 파라미터를 통해 작업으로 전달됩니다.

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

    const TickType_t xMaxBlockTimeTicks = 0x20;

    /* Two instances of this task are created. The task parameter is used to pass an index
    into an array of strings into the task. Cast this to the required type. */

    iIndexToString = ( int ) pvParameters;

    for( ;; )
    {
        /* Print out the string, not directly, but by passing a pointer to the string to
        the gatekeeper task through a queue. The queue is created before the scheduler is started
        so will already exist by the time this task executes for the first time. A block time is
        not specified because there should always be space in the queue. */

        xQueueSendToBack( xPrintQueue, &(amp; pcStringsToPrint[ iIndexToString ] ), 0 );

        /* Wait a pseudo random time. Note that rand() is not necessarily reentrant, but in
        this case it does not really matter because the code does not care what value is returned.
        In a more secure application, a version of rand() that is known to be reentrant should be
        used or calls to rand() should be protected using a critical section. */

        vTaskDelay( ( rand() % xMaxBlockTimeTicks ) );
    }
}
```

틱 후크 함수는 호출 횟수를 계산하여 200에 도달할 때마다 메시지를 게이트키퍼 작업에게 전송합니다. 여기에서는 설명을 목적으로 틱 후크가 대기열 앞에, 그리고 작업이 대기열 뒤에 작성합니다. 틱 후크 구현체는 다음과 같습니다.

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

    /* Print out a message every 200 ticks. The message is not written out directly, but
    sent to the gatekeeper task. */

    iCount++;
}
```

```
if( iCount >= 200 )
{
    /* Because xQueueSendToFrontFromISR() is being called from the tick hook, it is not
    necessary to use the xHigherPriorityTaskWoken parameter (the third parameter), and the
    parameter is set to NULL. */

    xQueueSendToFrontFromISR( xPrintQueue, &(amp; pcStringsToPrint[ 2 ] ), NULL );

    /* Reset the count ready to print out the string again in 200 ticks time. */

    iCount = 0;
}
}
```

평소와 같이 main()이 예제를 실행하는 데 필요한 대기열과 작업을 생성한 후 스케줄러를 시작합니다. main() 함수의 구현체는 다음과 같습니다.

```
/* Define the strings that the tasks and interrupt will print out via the gatekeeper. */
static char *pcStringsToPrint[] =
{
    "Task 1 *****\r\n", "Task 2
    -----\r\n", "Message printed from the tick
    hook interrupt #####\r\n"
};

/*-----*/

/* Declare a variable of type QueueHandle_t. The queue is used to send messages from the
print tasks and the tick interrupt to the gatekeeper task. */
QueueHandle_t xPrintQueue;

/*-----*/

int main( void )
{
    /* Before a queue is used it must be explicitly created. The queue is created to hold a
    maximum of 5 character pointers. */

    xPrintQueue = xQueueCreate( 5, sizeof( char * ) );

    /* Check the queue was created successfully. */

    if( xPrintQueue != NULL )
    {
        /* Create two instances of the tasks that send messages to the gatekeeper. The
        index to the string the task uses is passed to the task through the task parameter (the
        4th parameter to xTaskCreate()). The tasks are created at different priorities, so the
        higher priority task will occasionally preempt the lower priority task. */

        xTaskCreate( prvPrintTask, "Print1", 1000, ( void * ) 0, 1, NULL );
    }
}
```

```
xTaskCreate( prvPrintTask, "Print2", 1000, ( void * ) 1, 2, NULL );

/* Create the gatekeeper task. This is the only task that is permitted to directly
access standard out. */

xTaskCreate( prvStdioGatekeeperTask, "Gatekeeper", 1000, NULL, 0, NULL
);

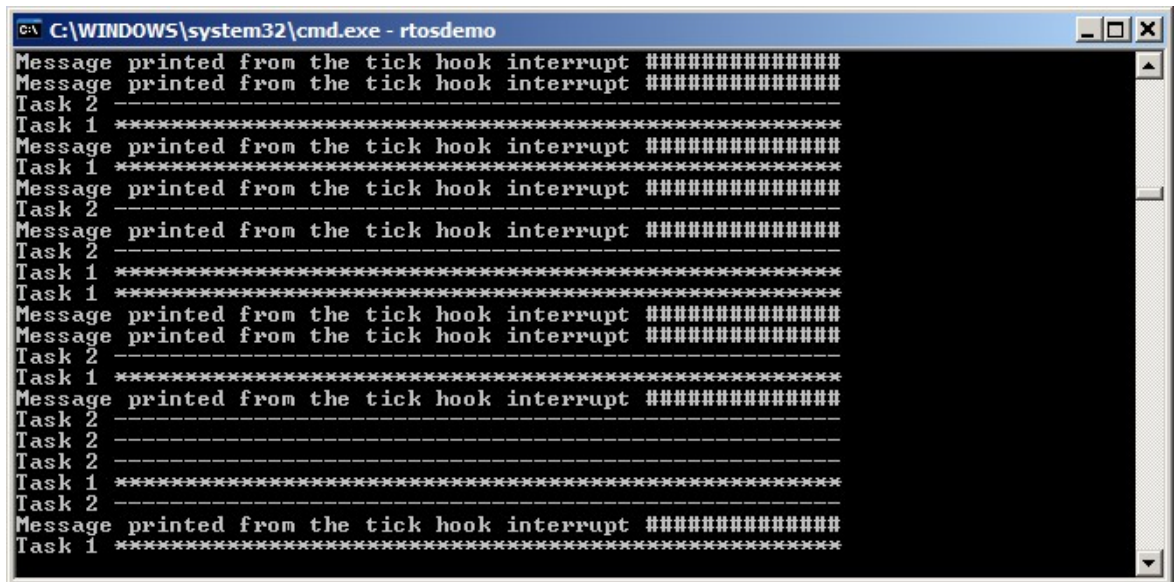
/* Start the scheduler so the created tasks start executing. */

vTaskStartScheduler();

}

/* If all is well, then main() will never reach here because the scheduler will
now be running the tasks. If main() does reach here, then it is likely that there was
insufficient heap memory available for the idle task to be created.*/
for( ;; );
}
```

출력되는 화면은 다음과 같습니다. 출력 화면을 보면 작업에서 전송되는 문자열과 인터럽트에서 전송되는 문자열이 모두 손상 없이 정확하게 출력되고 있습니다.



```
C:\WINDOWS\system32\cmd.exe - rtosdemo
Message printed from the tick hook interrupt #####
Task 2 -----
Task 1 *****
Message printed from the tick hook interrupt #####
Task 1 *****
Message printed from the tick hook interrupt #####
Task 2 -----
Message printed from the tick hook interrupt #####
Task 2 -----
Task 1 *****
Task 1 *****
Message printed from the tick hook interrupt #####
Message printed from the tick hook interrupt #####
Task 2 -----
Task 1 *****
Message printed from the tick hook interrupt #####
Task 2 -----
Task 2 -----
Task 2 -----
Task 1 *****
Task 2 -----
Message printed from the tick hook interrupt #####
Task 1 *****
```

게이트키퍼 작업에 할당되는 우선순위는 출력 작업보다 낮기 때문에 게이트키퍼에게 전송된 메시지는 두 출력 작업이 모두 Blocked 상태가 될 때까지 대기열을 떠나지 않습니다. 상황에 따라 게이트키퍼에게 더욱 높은 우선순위를 할당하여 메시지를 바로 처리해야 할 때도 있습니다. 이렇게 하면 게이트키퍼가 보호 리소스에 대한 액세스를 마칠 때까지 낮은 우선순위의 작업이 지연될 수 밖에 없습니다.

# 이벤트 그룹

이번 단원에서 다루는 내용은 다음과 같습니다.

- 이벤트 그룹의 실제 사용
- 다른 FreeRTOS 기능에 대한 이벤트 그룹의 장점과 단점
- 이벤트 그룹에서 비트를 설정하는 방법
- Blocked 상태에서 비트가 이벤트 그룹에 설정될 때까지 대기하는 방법
- 이벤트 그룹을 사용해 작업 집합을 동기화하는 방법

이벤트 그룹은 작업에 대한 이벤트 정보를 공유할 수 있는 또 하나의 FreeRTOS 기능입니다. 대기열 및 세마포어와 다른 이벤트 그룹의 기능

- 작업이 Blocked 상태에서 2가지 이상의 이벤트가 결합될 때까지 대기할 수 있습니다.
- 이벤트가 발생하면 동일한 이벤트 또는 이벤트 조합을 기다리던 모든 작업의 차단을 해제합니다.

이렇게 독특한 이벤트 그룹 속성은 다수의 작업을 동기화하거나, 이벤트를 1개 이상의 작업에 브로드캐스팅하거나, 작업이 Blocked 상태에서 이벤트가 발생할 때까지 대기하거나, 작업이 Blocked 상태에서 다수의 작업이 완료될 때까지 대기하는 데 매우 유용합니다.

이벤트 그룹 역시 애플리케이션에서 사용하는 RAM을 줄이는 데 유용합니다. 이는 다수의 이진 세마포어를 단일 이벤트 그룹으로 변경할 수 있는 경우가 많기 때문입니다.

이벤트 그룹 기능은 선택 사항입니다. 이벤트 그룹 기능을 추가하려면 FreeRTOS 소스 파일인 `event_groups.c`를 빌드하여 프로젝트에 포함시키면 됩니다.

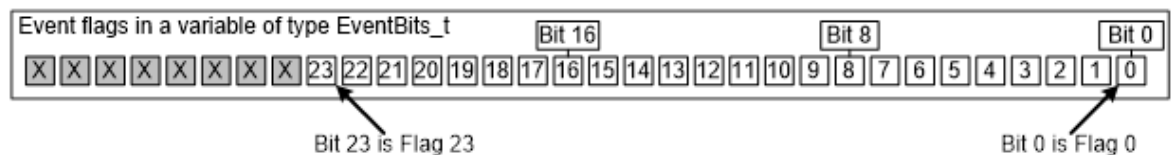
## 이벤트 그룹의 특성

### 이벤트 그룹, 이벤트 플래그 및 이벤트 비트

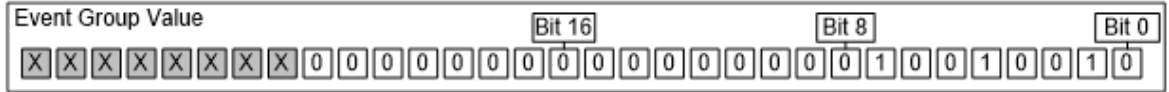
이벤트 플래그는 이벤트 발생 여부를 나타낼 때 사용되는 부울(1 또는 0) 값입니다. 하나의 이벤트 그룹은 이벤트 플래그 집합이라고 할 수 있습니다.

이벤트 플래그는 오직 1 또는 0만 될 수 있으며, 이는 플래그 상태를 단일 비트로 저장하거나, 혹은 이벤트 그룹에 속한 모든 이벤트 플래그 상태를 단일 변수로 저장할 수 있다는 것을 의미합니다. 이벤트 그룹에 속한 각 이벤트 플래그의 상태는 `EventBits_t` 형식의 변수에서 단일 비트로 표현됩니다. 이벤트 플래그가 이벤트 비트로 알려진 이유도 바로 여기에 있습니다. `EventBits_t` 변수에서 비트가 1로 설정되면 해당 비트가 나타내는 이벤트가 발생한 것을 의미합니다. `EventBits_t` 변수에서 비트가 0으로 설정되면 해당 비트가 나타내는 이벤트가 발생하지 않은 것을 의미합니다.

아래 그림은 각 이벤트 플래그가 `EventBits_t` 형식의 변수에서 각 비트로 매핑되는 방법을 나타낸 것입니다.



예를 들어 이벤트 그룹의 값이 0x92(이진수 1001 0010)라고 가정할 경우 이벤트 비트 1, 4 및 7만 설정되기 때문에 비트 1, 4 및 7이 나타내는 이벤트만 발생합니다. 아래 그림은 이벤트 비트 1, 4 및 7만 설정되고, 나머지 이벤트 비트는 모두 소거된 EventBits\_t 형식의 변수를 나타낸 것으로, 이벤트 그룹의 값은 0x92가 됩니다.



이벤트 그룹의 각 비트에 의미를 부여하는 것은 애플리케이션 개발자의 몫입니다. 예를 들어 애플리케이션 개발자는 이벤트 그룹을 생성한 후 다음과 같이 의미를 부여할 수 있습니다.

- 이벤트 그룹에서 비트 0을 정의하여 메시지가 네트워크에서 수신되었다는 의미를 부여합니다.
- 이벤트 그룹에서 비트 1을 정의하여 메시지가 네트워크로 전송 대기 상태라는 의미를 부여합니다.
- 이벤트 그룹에서 비트 2를 정의하여 현재 네트워크 연결을 중단한다는 의미를 부여합니다.

## EventBits\_t 데이터 형식에 대한 기타 정보

이벤트 그룹에 속하는 이벤트 비트의 수는 FreeRTOSConfig.h에서 configUSE\_16\_BIT\_TICKS 컴파일 시간 구성 상수에 따라 달라집니다. 이 상수는 RTOS 틱 카운트를 저장할 때 사용되는 형식을 구성하기 때문에 이벤트 그룹 기능과 무관하게 보일 수도 있습니다. 하지만 이 상수가 EventBits\_t 형식에 미치는 영향에 따라 FreeRTOS의 내부 구현이 결정될 뿐만 아니라, FreeRTOS가 32비트 형식보다는 16비트 형식을 더욱 효율적으로 처리할 수 있는 아키텍처에서 실행될 때는 configUSE\_16\_BIT\_TICKS가 1로 설정되어야 하기 때문에 오히려 타당합니다.

- configUSE\_16\_BIT\_TICKS가 1이면 각 이벤트 그룹에서 사용할 수 있는 이벤트 비트는 8개입니다.
- configUSE\_16\_BIT\_TICKS가 0이면 각 이벤트 그룹에서 사용할 수 있는 이벤트 비트는 24개입니다.

## 다수의 작업에서 액세스

이벤트 그룹은 이벤트 그룹의 존재에 대해 알고 있는 작업 또는 ISR에서 액세스할 수 있는 객체입니다. 동일한 이벤트 그룹에서 비트를 설정할 수 있는 작업의 수와 동일한 이벤트 그룹에서 비트를 읽어올 수 있는 작업의 수는 제한이 없습니다.

## 이벤트 그룹의 실제 사용 예

FreeRTOS+TCP TCP/IP 스택 구현에서는 이벤트 그룹을 사용해 설계를 간소화하는 동시에 리소스 사용량을 최소화할 수 있는 방법을 실제 사례를 들어 설명합니다.

TCP 소켓은 허용 이벤트, 바인딩 이벤트, 읽기 이벤트, 닫기 이벤트 등 여러 가지 다양한 이벤트에 응답해야 합니다. 소켓이 특정 시간에 기대할 수 있는 이벤트는 소켓 상태에 따라 다릅니다. 예를 들어 소켓이 생성 후에도 아직 주소로 바인딩되지 않았다면 읽기 이벤트가 아닌 바인딩 이벤트를 예상할 수 있습니다. (주소가 없으면 데이터를 읽어올 수 없습니다)

FreeRTOS+TCP 소켓의 상태는 FreeRTOS\_Socket\_t라고 하는 구조로 저장됩니다. 여기에는 소켓이 처리해야 하는 각 이벤트마다 정의된 이벤트 비트의 이벤트 그룹이 포함됩니다. 이벤트 또는 이벤트 그룹이 발생할 때까지 대기하도록 차단하는 FreeRTOS+TCP API 호출은 이벤트 그룹에서만 차단합니다.

이벤트 그룹에도 '중단' 비트가 포함되어 있어서 소켓이 해당 시점에 대기하고 있는 이벤트에 상관없이 TCP 연결을 중단할 수 있습니다.

## 이벤트 그룹을 사용한 이벤트 관리

### xEventGroupCreate() API 함수

FreeRTOS V9.0.0에도 컴파일 과정에서 이벤트 그룹을 정적으로 생성하는 데 필요한 메모리를 할당하는 xEventGroupCreateStatic() 함수가 포함되어 있습니다. 이벤트 그룹을 사용하려면 먼저 명시적으로 생성해야 합니다.

이벤트 그룹은 EventGroupHandle\_t 형식의 변수를 사용해 참조됩니다. xEventGroupCreate() API 함수는 이벤트 그룹을 생성하는 데 사용됩니다. 또한 EventGroupHandle\_t를 반환하여 생성할 이벤트 그룹을 참조합니다.

xEventGroupCreate() API 함수 프로토타입은 다음과 같습니다.

```
EventGroupHandle_t xEventGroupCreate( void );
```

다음 표는 xEventGroupCreate() 반환 값을 나열한 것입니다.

파라미터 이름	설명
반환 값	<p>NULL을 반환하는 경우에는 이벤트 그룹을 생성할 수 없습니다. FreeRTOS에서 이벤트 그룹 데이터 구조를 할당하는 데 필요한 힙 메모리가 부족하기 때문입니다. 자세한 내용은 <a href="#">힙 메모리 관리 (p. 13)</a> 단원을 참조하십시오.</p> <p>NULL이 아닌 값을 반환하는 경우에는 이벤트 그룹이 성공적으로 생성된 것입니다. 반환 값은 생성된 이벤트 그룹에 대한 핸들로 저장되어야 합니다.</p>

### xEventGroupSetBits() API 함수

xEventGroupSetBits() API 함수는 이벤트 그룹에서 비트 1개 이상을 설정합니다. 일반적으로 설정된 비트(들)가 나타내는 이벤트가 발생하였다는 것을 작업에게 알리는 데 사용됩니다.

참고: xEventGroupSetBits()를 인터럽트 서비스 루틴에서 호출하지 마십시오. 대신에 인터럽트 세이프 버전인 xEventGroupSetBitsFromISR()을 사용하십시오.

xEventGroupSetBits() API 함수 프로토타입은 다음과 같습니다.

```
EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup, const EventBits_t  
uxBitsToSet );
```

다음 표는 xEventGroupSetBits() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름	설명
xEventGroup	비트가 설정되어 있는 이벤트 그룹의 핸들입니다. 이벤트 그룹 핸들은 이벤트 그룹 생성에 사용되는 xEventGroupCreate() 호출을 통해 반환됩니다.
uxBitsToSet	이벤트 그룹에서 이벤트 비트(들)를 1로 설정하도록 지정하는 비트 마스크입니다. 이벤트 그룹의 값은

	이벤트 그룹의 기존 값과 uxBitsToSet에서 전달되는 값을 비트 논리합으로 연산 처리하여 업데이트됩니다.  예를 들어 uxBitsToSet를 0x04(이진수 0100)로 설정하면 이벤트 그룹에서 이벤트 비트 3이 설정되고(아직 설정되지 않은 경우) 이벤트 그룹에서 나머지 이벤트 비트는 모두 바뀌지 않습니다.
반환 값	xEventGroupSetBits() 호출 시점의 이벤트 그룹 값이 반환됩니다. 다른 작업에서 비트가 다시 소거되었을 가능성도 있기 때문에 uxBitsToSet에서 지정한 비트가 반환 값에 반드시 설정되는 것은 아닙니다.

## xEventGroupSetBitsFromISR() API 함수

xEventGroupSetBitsFromISR()은 xEventGroupSetBits()의 인터럽트 세이프 버전입니다.

세마포어를 반환(give)해도 Blocked 상태를 종료할 수 있는 작업이 겨우 1개라는 사실을 사전에 알고 있기 때문에 세마포어 반환은 확실적인 연산입니다. 하지만 이벤트 그룹에서 비트를 설정하더라도 Blocked 상태를 종료하는 작업 수를 사전에 알 수 없기 때문에 이벤트 그룹의 비트 설정은 확실적인 연산이 아닙니다.

FreeRTOS 설계 및 구현 표준은 인터럽트 서비스 루틴 내에서, 혹은 인터럽트가 비활성화되어 있을 경우 비확정적인 연산을 허용하지 않습니다. 이러한 이유로 xEventGroupSetBitsFromISR()은 인터럽트 서비스 루틴 내에서 이벤트 비트를 직접 설정하지 않습니다. 대신에 이 작업을 RTOS 데몬 작업에게 위임합니다.

xEventGroupSetBitsFromISR() API 함수 프로토타입은 다음과 같습니다.

```
BaseType_t xEventGroupSetBitsFromISR( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet, BaseType_t *pxHigherPriorityTaskWoken );
```

다음은 xEventGroupSetBitsFromISR() 파라미터와 반환 값을 나열한 것입니다.

**xEventGroup**

비트가 설정되어 있는 이벤트 그룹의 핸들입니다. 이벤트 그룹 핸들은 이벤트 그룹 생성에 사용되는 xEventGroupCreate() 호출을 통해 반환됩니다.

**uxBitsToSet**

이벤트 그룹에서 이벤트 비트(들)를 1로 설정하도록 지정하는 비트 마스크입니다. 이벤트 그룹의 값은 이벤트 그룹의 기존 값과 uxBitsToSet로 전달되는 값을 비트 논리합으로 연산 처리하여 업데이트됩니다. 예를 들어 uxBitsToSet를 0x05(이진수 0101)로 설정하면 이벤트 비트 3과 0이 설정되고(아직 설정되지 않은 경우) 이벤트 그룹에서 나머지 이벤트 비트는 모두 바뀌지 않습니다.

**pxHigherPriorityTaskWoken**

xEventGroupSetBitsFromISR()은 인터럽트 서비스 루틴 내에서 이벤트 비트를 직접 설정하지 않습니다. 대신에 타이머 명령 대기열에 대한 명령을 전송하여 RTOS 데몬 작업에게 작업을 위임합니다. 데몬 작업이 타이머 명령 대기열에서 사용할 수 있을 때까지 Blocked 상태로 기다려야 하는 경우에는 타이머 명령 대기열에 대한 쓰기 연산으로 데몬 작업이 Blocked 상태를 종료하게 됩니다. 데몬 작업의 우선순위가 현재 실행 중인 작업(종단된 작업)보다 높다면 xEventGroupSetBitsFromISR()이 pxHigherPriorityTaskWoken을 pdTRUE로 설정합니다.

xEventGroupSetBitsFromISR()이 이 값을 pdTRUE로 설정하면 인터럽트 종료 전에 컨텍스트 전환이 이루어져야 합니다. 그러면 데몬 작업이 최고 우선순위인 Ready 상태의 작업이 되어 인터럽트 복귀가 데몬 작업으로 직접 이루어질 수 있습니다.

가능한 반환 값은 다음 두 가지입니다.

데이터가 타이머 명령 대기열에 성공적으로 전송된 경우에 한해 pdPASS가 반환됩니다.

대기열이 이미 가득 차있어서 'set bits' 명령이 타이머 명령 대기열에 작성되지 않은 경우 pdFALSE가 반환됩니다.

## xEventGroupWaitBits() API 함수

xEventGroupWaitBits() API 함수를 호출하면 작업이 이벤트 그룹의 값을 읽어올 수 있으며, 이벤트 그룹에서 이벤트 비트 1개 이상이 아직 설정되어 있지 않은 경우에는 선택에 따라 설정될 때까지 Blocked 상태로 대기할 수도 있습니다.

xEventGroupWaitBits() API 함수 프로토타입은 다음과 같습니다.

```
EventBits_t xEventGroupWaitBits( const EventGroupHandle_t xEventGroup, const EventBits_t
    uxBitsToWaitFor, const BaseType_t xClearOnExit, const BaseType_t xWaitForAllBits,
    TickType_t xTicksToWait );
```

차단 해제 조건은 스케줄러가 작업의 Blocked 상태 시작 또는 작업 Blocked 상태 종료를 결정할 때 사용되는 조건입니다. 차단 해제 조건은 다음과 같이 uxBitsToWaitFor 및 xWaitForAllBits 파라미터 값을 함께 사용해 지정됩니다.

- uxBitsToWaitFor는 이벤트 그룹에서 테스트할 이벤트 비트를 지정합니다.
- xWaitForAllBits는 비트 논리합(OR) 테스트 또는 비트 논리곱(AND) 테스트 중에서 무엇을 사용할지 지정합니다.

xEventGroupWaitBits() 호출 시점에 차단 해제 조건이 충족되면 작업이 Blocked 상태로 전환되지 않습니다.

다음 표는 작업이 Blocked 상태를 시작하거나 종료하기 위한 조건의 예를 나열한 것입니다. 여기에는 이벤트 그룹의 최하위 이진 비트 4개와 uxBitsToWaitFor 값만 표시되어 있습니다. 두 값의 나머지 비트는 모두 0이라고 가정합니다.

기존 이벤트 그룹 값	uxBitsToWaitFor 값	xWaitForAllBits 값	결과적 동작
0000	0101	pdFALSE	비트 0과 비트 2 모두 이벤트 그룹에서 설정되지 않기 때문에 호출 작업이 Blocked 상태로 전환되며, 비트 0 또는 비트 2가 이벤트 그룹에서 설정되면 Blocked 상태를 종료합니다.
0100	0101	pdTRUE	비트 0과 비트 2 모두 이벤트 그룹에서 설정되지 않기 때문에 호출 작업이 Blocked 상태로 전환되며, 비트 0과 비트 2가 모두 이벤트 그룹에서 설정되면 Blocked 상태를 종료합니다.
0100	0110	pdFALSE	xWaitForAllBits가 pdFALSE이고, uxBitsToWaitFor에서 지정한 두 비트 중 하나가 이미 이벤트 그룹에서 설



			정되어 있기 때문에 호출 작업이 Blocked 상태로 전환되지 않습니다.
0100	0110	pdTRUE	xWaitForAllBits가 pdTRUE이고, uxBitsToWaitFor에서 지정한 두 비트 중 하나만 이미 이벤트 그룹에서 설정되어 있기 때문에 호출 작업이 Blocked 상태로 전환됩니다. 비트 2와 비트 3이 모두 이벤트 그룹에서 설정되면 작업이 Blocked 상태를 종료합니다.

호출 작업은 uxBitsToWaitFor 파라미터를 사용해 테스트할 비트를 지정합니다. 이후 차단 해제 조건이 충족되면 이 비트들을 다시 0으로 소거해야 할 수도 있습니다. 이벤트 비트는 xEventGroupClearBits() API 함수를 사용해 소거할 수 있지만 이 함수를 사용해 이벤트 비트를 수동으로 소거하면 다음과 같은 경우에 애플리케이션 코드의 경합 조건 원인이 됩니다.

- 동일한 이벤트 그룹을 사용하는 작업이 다수인 경우
- 비트가 다른 작업 또는 ISR에서 이벤트 그룹에 설정된 경우

이러한 잠재적 경합 조건을 방지하기 위해서 입력하는 것이 바로 xClearOnExit 파라미터입니다. xClearOnExit가 pdTRUE로 설정되면 이벤트 비트의 테스트 및 소거가 호출 작업에게 원자 단위의 연산(다른 작업 또는 인터럽트로 더 이상 중단될 수 없는 연산)으로 인식됩니다.

다음 표는 xEventGroupWaitBits() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름	설명
xEventGroup	읽고 있는 이벤트 비트가 포함된 이벤트 그룹의 핸들입니다. 이벤트 그룹 핸들은 이벤트 그룹 생성에 사용되는 xEventGroupCreate() 호출을 통해 반환됩니다.
uxBitsToWaitFor	이벤트 그룹에서 테스트할 이벤트 비트(들)를 지정하는 비트 마스크입니다.  예를 들어 호출 작업이 이벤트 비트 0 및/또는 이벤트 비트 2가 이벤트 그룹에서 설정될 때까지 대기하는 경우에는 uxBitsToWaitFor를 0x05(이진수 0101)로 설정합니다. 추가 예는 표 45를 참조하십시오.
xClearOnExit	호출 작업의 차단 해제 조건이 충족된 동시에 xClearOnExit가 pdTRUE로 설정된 경우에는 호출 작업이 xEventGroupWaitBits() API 함수를 종료하기 전에 uxBitsToWaitFor에서 지정한 이벤트 비트가 이벤트 그룹에서 0으로 다시 소거됩니다.  xClearOnExit가 pdFALSE로 설정된 경우, 이벤트 그룹의 이벤트 비트 상태는 xEventGroupWaitBits() API 함수로도 수정되지 않습니다.

<p>xWaitForAllBits</p>	<p>uxBitsToWaitFor 파라미터는 이벤트 그룹에서 이벤트 비트를 지정합니다. xWaitForAllBits는 uxBitsToWaitFor 파라미터에서 지정하는 이벤트 비트 1개 이상이 설정된 경우에 호출 작업을 Blocked 상태에서 제거할지, 혹은 uxBitsToWaitFor 파라미터에서 지정하는 이벤트 비트가 모두 설정된 경우에만 호출 작업을 Blocked 상태에서 제거할지 지정합니다.</p> <p>xWaitForAllBits가 pdFALSE로 설정되면 Blocked 상태로 전환되어 차단 해제 조건이 충족될 때까지 대기하던 작업이 uxBitsToWaitFor에서 지정하는 비트 중 하나라도 설정되었을 때(또는 xTicksToWait 파라미터에서 지정한 제한 시간이 지났을 때) Blocked 상태를 종료합니다.</p> <p>xWaitForAllBits가 pdTRUE로 설정되면 Blocked 상태로 전환되어 차단 해제 조건이 충족될 때까지 대기하던 작업이 uxBitsToWaitFor에서 지정하는 모든 비트가 설정되었을 때(또는 xTicksToWait 파라미터에서 지정한 제한 시간이 지났을 때)에 한해 Blocked 상태를 종료합니다.</p> <p>예는 이전 표를 참조하십시오.</p>
<p>xTicksToWait</p>	<p>작업이 차단 해제 조건이 충족될 때까지 Blocked 상태로 대기해야 하는 최대 시간입니다.</p> <p>xTicksToWait가 0이거나, 혹은 xEventGroupWaitBits() 호출 시점에 차단 해제 조건이 충족되면 xEventGroupWaitBits()가 즉시 값을 반환합니다.</p> <p>차단 시간은 틱 주기로 지정되기 때문에 틱 주파수에 따라 절대 시간이 달라집니다. 밀리초 단위의 시간은 pdMS_TO_TICKS() 매크로를 사용해 틱 단위의 시간으로 변환할 수 있습니다.</p> <p>xTicksToWait를 portMAX_DELAY로 설정하면 작업이 제한 시간 없이 무기한 대기하게 됩니다. 단, INCLUDE_vTaskSuspend가 FreeRTOSConfig.h에서 1로 설정되어 있어야 합니다.</p>
<p>반환 값</p>	<p>호출 작업의 차단 해제 조건이 충족되어 xEventGroupWaitBits()가 반환한 경우에는 호출 작업의 차단 해제 조건이 충족된 시점(xClearOnExit가 pdTRUE인 경우에는 비트가 자동 소거되기 이전 시점)의 이벤트 그룹 값이 반환됩니다. 이러한 경우에는 반환 값 역시 차단 해제 조건을 충족합니다.</p> <p>xTicksToWait 파라미터에서 지정한 차단 시간이 지났기 때문에 xEventGroupWaitBits()가 반환한 경우에는 차단 시간이 지난 시점의 이벤트 그룹 값이 반환됩니다. 이러한 경우에는 반환 값이 차단 해제 조건을 충족하지 못합니다.</p>

## 이벤트 그룹 실험(예제 22)

이번 예제에서는 다음과 같은 이벤트 그룹 사용 방법에 대해서 알아봅니다.

- 이벤트 그룹 생성
- ISR의 이벤트 그룹에서 비트 설정
- 작업의 이벤트 그룹에서 비트 설정
- 이벤트 그룹에서 차단

먼저 `xWaitForAllBits`를 `pdFALSE`로 설정하여 예제를 실행한 다음 `xWaitForAllBits`를 `pdTRUE`로 설정하여 예제를 실행하면서 `xEventGroupWaitBits()` `xWaitForAllBits` 파라미터가 어떤 영향을 미치는지 알아보겠습니다.

이벤트 비트 0과 이벤트 비트 1은 작업에서 설정됩니다. 이벤트 비트 2는 ISR에서 설정됩니다. 이 세 가지 비트는 아래와 같이 `#define` 문을 사용해 서술형 이름이 지정됩니다.

```
/* Definitions for the event bits in the event group. */

#define mainFIRST_TASK_BIT ( 1UL << 0UL ) /* Event bit 0, which is set by a task. */

#define mainSECOND_TASK_BIT ( 1UL << 1UL ) /* Event bit 1, which is set by a task. */

#define mainISR_BIT ( 1UL << 2UL ) /* Event bit 2, which is set by an ISR. */
```

다음 코드는 이벤트 비트 0과 이벤트 비트 1을 설정하는 작업을 구현한 것입니다. 비트 1개를 반복해서 설정하는 루프를 시작하고, 이후 `xEventGroupSetBits()`를 호출할 때마다 각각 200밀리초의 지연 시간이 지나서 나머지 비트를 설정하는 루프를 시작합니다. 각 비트의 설정 이전에 문자열이 출력되어 콘솔에서 실행 순서를 확인할 수 있습니다.

```
static void vEventBitSettingTask( void *pvParameters )
{
    const TickType_t xDelay200ms = pdMS_TO_TICKS( 200UL ), xDontBlock = 0;

    for( ;; )
    {
        /* Delay for a short while before starting the next loop. */
        vTaskDelay( xDelay200ms );

        /* Print out a message to say event bit 0 is about to be set by the task, and then set event bit 0. */
        vPrintString( "Bit setting task -\t about to set bit 0.\r\n" );

        xEventGroupSetBits( xEventGroup, mainFIRST_TASK_BIT );

        /* Delay for a short while before setting the other bit. */
        vTaskDelay( xDelay200ms );

        /* Print out a message to say event bit 1 is about to be set by the task, and then set event bit 1. */
        vPrintString( "Bit setting task -\t about to set bit 1.\r\n" );

        xEventGroupSetBits( xEventGroup, mainSECOND_TASK_BIT );
    }
}
```

```

    }
}

```

다음 코드는 이벤트 그룹에서 비트 2를 설정하는 인터럽트 서비스 루틴을 구현한 것입니다. 여기에서도 비트 설정 이전에 문자열이 출력되어 콘솔에서 실행 순서를 확인할 수 있습니다. 이러한 경우에는 인터럽트 서비스 루틴에서 콘솔 출력이 직접 실행되지 않기 때문에 `xTimerPendFunctionCallFromISR()`을 사용해 RTOS 데몬 작업의 컨텍스트에서 출력을 실행합니다.

앞의 예제들과 마찬가지로 여기에서도 인터럽트 서비스 루틴이 소프트웨어 인터럽트를 강제하는 주기적 작업을 통해 트리거됩니다. 이번 예제에서는 인터럽트가 500밀리초마다 생성됩니다.

```

static uint32_t ulEventBitSettingISR( void )
{
    /* The string is not printed within the interrupt service routine, but is instead
    sent to the RTOS daemon task for printing. It is therefore declared static to ensure the
    compiler does not allocate the string on the stack of the ISR because the ISR's stack
    frame will not exist when the string is printed from the daemon task. */

    static const char *pcString = "Bit setting ISR -\t about to set bit 2.\r\n";

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Print out a message to say bit 2 is about to be set. Messages cannot be printed from
    an ISR, so defer the actual output to the RTOS daemon task by pending a function call to
    run in the context of the RTOS daemon task. */

    xTimerPendFunctionCallFromISR( vPrintStringFromDaemonTask, ( void * ) pcString, 0,
    &xHigherPriorityTaskWoken );

    /* Set bit 2 in the event group. */

    xEventGroupSetBitsFromISR( xEventGroup, mainISR_BIT, &xHigherPriorityTaskWoken );

    /* xTimerPendFunctionCallFromISR() and xEventGroupSetBitsFromISR() both write to the
    timer command queue, and both used the same xHigherPriorityTaskWoken variable. If writing
    to the timer command queue resulted in the RTOS daemon task leaving the Blocked state,
    and if the priority of the RTOS daemon task is higher than the priority of the currently
    executing task (the task this interrupt interrupted), then xHigherPriorityTaskWoken
    will have been set to pdTRUE. xHigherPriorityTaskWoken is used as the parameter
    to portYIELD_FROM_ISR(). If xHigherPriorityTaskWoken equals pdTRUE, then calling
    portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still
    pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. The implementation of
    portYIELD_FROM_ISR() used by the Windows port includes a return statement, which is why
    this function does not explicitly return a value. */

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}

```

다음 코드는 `xEventGroupWaitBits()`를 호출하여 이벤트 그룹에서 차단할 작업을 구현한 것입니다. 작업은 이벤트 그룹에서 설정되는 비트마다 문자열을 출력합니다.

`xEventGroupWaitBits()` `xClearOnExit` 파라미터가 `pdTRUE`로 설정되어 `xEventGroupWaitBits()`를 호출했을 때 값을 반환하게 했던 이벤트 비트(들)가 `xEventGroupWaitBits()`의 값 반환 이전에 자동으로 소거됩니다.

```

static void vEventBitReadingTask( void *pvParameters )
{

```

```
EventBits_t xEventGroupValue;

const EventBits_t xBitsToWaitFor = ( mainFIRST_TASK_BIT | mainSECOND_TASK_BIT |
mainISR_BIT );

for( ;; )
{
    /* Block to wait for event bits to become set within the event group.*/

    xEventGroupValue = xEventGroupWaitBits( /* The event group to read. */
xEventGroup, /* Bits to test. */ xBitsToWaitFor, /* Clear bits on exit if the unblock
condition is met. */ pdTRUE, /* Don't wait for all bits. This parameter is set to pdTRUE
for the second execution. */ pdFALSE, /* Don't time out. */ portMAX_DELAY );

    /* Print a message for each bit that was set. */

    if( ( xEventGroupValue & mainFIRST_TASK_BIT ) != 0 )
    {
        vPrintString( "Bit reading task -\t Event bit 0 was set\r\n" );
    }

    if( ( xEventGroupValue & mainSECOND_TASK_BIT ) != 0 )
    {
        vPrintString( "Bit reading task -\t Event bit 1 was set\r\n" );
    }

    if( ( xEventGroupValue & mainISR_BIT ) != 0 )
    {
        vPrintString( "Bit reading task -\t Event bit 2 was set\r\n" );
    }
}
}
```

main() 함수는 스케줄러를 시작하기 전에 이벤트 그룹과 작업을 생성합니다. 다음 코드는 이 함수의 구현을 나타낸 것입니다. 이벤트 그룹에서 읽어오는 작업의 우선순위는 이벤트 그룹에 작성하는 작업의 우선순위보다 높기 때문에 읽기 작업은 차단 해제 조건이 충족될 때마다 쓰기 작업보다 먼저 실행됩니다.

```
int main( void )
{
    /* Before an event group can be used it must first be created. */

    xEventGroup = xEventGroupCreate();

    /* Create the task that sets event bits in the event group. */

    xTaskCreate( vEventBitSettingTask, "Bit Setter", 1000, NULL, 1, NULL );

    /* Create the task that waits for event bits to get set in the event group. */
}
```

```
xTaskCreate( vEventBitReadingTask, "Bit Reader", 1000, NULL, 2, NULL );

/* Create the task that is used to periodically generate a software interrupt. */

xTaskCreate( vInterruptGenerator, "Int Gen", 1000, NULL, 3, NULL );

/* Install the handler for the software interrupt. The syntax required to do this is
depends on the FreeRTOS port being used. The syntax shown here can only be used with the
FreeRTOS Windows port, where such interrupts are only simulated. */

vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulEventBitSettingISR );

/* Start the scheduler so the created tasks start executing. */

vTaskStartScheduler();

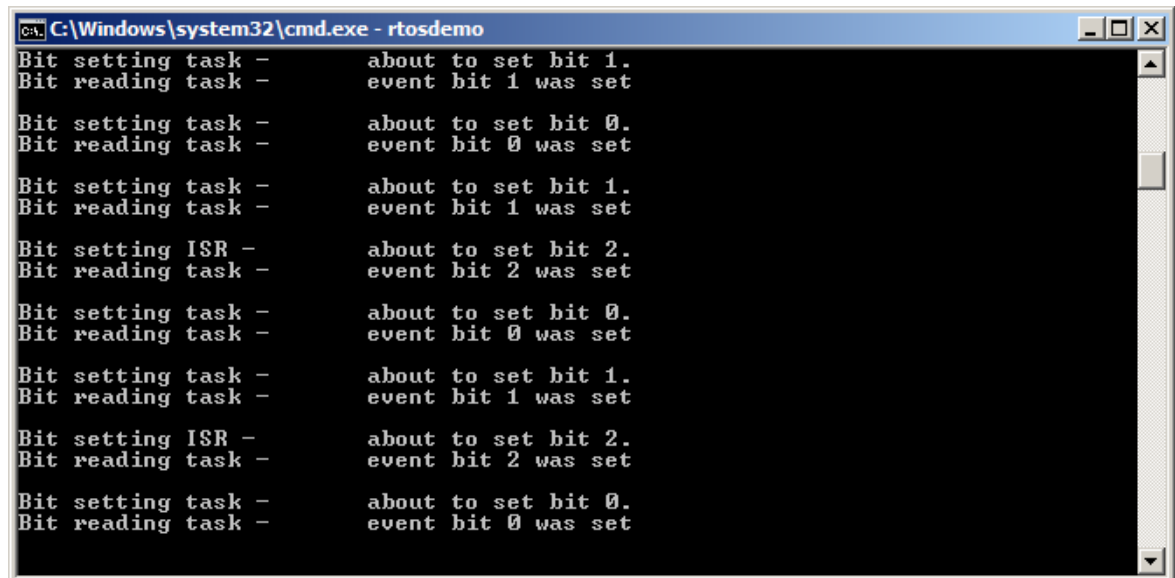
/* The following line should never be reached. */

for( ;; );

return 0;

}
```

xEventGroupWaitBits() xWaitForAllBits 파라미터가 pdFALSE로 설정된 상태에서 예제 22를 실행했을 때 출력 화면은 아래와 같습니다. 이렇게 출력되는 이유는 xEventGroupWaitBits() 호출 시 xWaitForAllBits 파라미터가 pdFALSE로 설정되어 이벤트 비트 중 하나라도 설정될 때마다 이벤트 그룹에서 읽어오는 작업이 Blocked 상태를 종료하고 바로 실행되기 때문입니다.



xEventGroupWaitBits() xWaitForAllBits 파라미터가 pdTRUE로 설정된 상태에서 코드를 실행했을 때 출력 화면은 아래와 같습니다. 이렇게 출력되는 이유는 xWaitForAllBits 파라미터가 pdTRUE로 설정되어 이벤트 비트 3개가 모두 설정된 후에만 이벤트 그룹에서 읽어오는 작업이 Blocked 상태를 종료하기 때문입니다.

```
C:\Windows\system32\cmd.exe - rtdemo
Bit setting task -      about to set bit 1.
Bit setting task -      about to set bit 0.
Bit setting ISR -       about to set bit 2.
Bit reading task -      event bit 0 was set
Bit reading task -      event bit 1 was set
Bit reading task -      event bit 2 was set

Bit setting task -      about to set bit 1.
Bit setting task -      about to set bit 0.
Bit setting ISR -       about to set bit 2.
Bit reading task -      event bit 0 was set
Bit reading task -      event bit 1 was set
Bit reading task -      event bit 2 was set

Bit setting task -      about to set bit 1.
Bit setting task -      about to set bit 0.
Bit setting task -      about to set bit 1.
Bit setting ISR -       about to set bit 2.
Bit reading task -      event bit 0 was set
Bit reading task -      event bit 1 was set
Bit reading task -      event bit 2 was set

Bit setting task -      about to set bit 0.
Bit setting task -      about to set bit 1.
```

## 이벤트 그룹을 사용한 작업 동기화

애플리케이션을 설계하려면 간혹 작업 2개 이상을 서로 동기화해야 할 때가 있습니다. 예를 들어 작업 A가 이벤트를 수신한 후 이벤트에 필요한 일부 처리를 다른 작업 3개(작업 B, 작업 C 및 작업 D)에게 위임한다고 가정하겠습니다. 이때 작업 B, C 및 D가 이전 이벤트의 처리를 끝낼 때까지 작업 A가 다른 이벤트를 수신하지 못한다면 작업 4개를 서로 동기화해야 합니다. 각 작업의 동기화 지점은 해당 작업이 처리를 마치고 나머지 작업이 각각 동일한 처리를 마칠 때까지 더 이상 처리할 수 없을 때입니다. 작업 A는 작업 4개가 모두 동기화 지점에 도달한 후에만 다른 이벤트를 수신할 수 있습니다.

FreeRTOS+TCP 데모 프로젝트 중 하나에서 이러한 유형의 작업 동기화가 필요하지만 비교적 덜 추상적인 예제를 찾아볼 수 있습니다. 이 데모에서는 두 작업 사이에 TCP 소켓을 공유합니다. 한 작업은 데이터를 소켓에 전송하고, 다른 작업이 동일한 소켓에서 데이터를 수신합니다. (현재 작업 사이에서 단일 FreeRTOS+TCP 소켓을 공유할 수 있는 유일한 방법입니다) 한 작업이 소켓에 다시 액세스하지 않는다는 확신이 없는 한 어느 한쪽 작업에서 TCP 소켓을 닫는 것은 위험합니다. 두 작업 중 하나가 소켓을 닫아야 할 경우에는 나머지 작업에게 미리 알리고, 처리 이전에 나머지 작업이 소켓 사용을 중단할 때까지 기다려야 합니다.

소켓에 데이터를 전송하는 작업이 소켓을 닫아야 하는 시나리오는 매우 많습니다. 동기화해야 하는 작업이 2개뿐이기 때문입니다. 하지만 다른 작업이 소켓이 열려야만 처리를 실행하는 경우에는 시나리오가 더욱 복잡해져 동기화해야 할 작업이 얼마나 많아지는지 쉽게 알 수 있습니다.

```
void SocketTxTask( void *pvParameters )
{
    xSocket_t xSocket;

    uint32_t ulTxCount = 0UL;

    for( ;; )
    {
        /* Create a new socket. This task will send to this socket, and another task
        will receive from this socket. */

        xSocket = FreeRTOS_socket( ... );
```

```
/* Connect the socket. */
FreeRTOS_connect( xSocket, ... );

/* Use a queue to send the socket to the task that receives data. */
xQueueSend( xSocketPassingQueue, &xSocket, portMAX_DELAY );

/* Send 1000 messages to the socket before closing the socket. */
for( ulTxCount = 0; ulTxCount < 1000; ulTxCount++ )
{
    if( FreeRTOS_send( xSocket, ... ) < 0 )
    {
        /* Unexpected error - exit the loop, after which the socket
will be closed. */
        break;
    }
}

/* Let the Rx task know the Tx task wants to close the socket. */
TxTaskWantsToCloseSocket();

/* This is the Tx task's synchronization point. The Tx task waits here
for the Rx task to reach its synchronization point. The Rx task will only reach its
synchronization point when it is no longer using the socket, and the socket can be closed
safely. */

xEventGroupSync( ... );

/* Neither task is using the socket. Shut down the connection, and then close
the socket. */
FreeRTOS_shutdown( xSocket, ... );
WaitForSocketToDisconnect();
FreeRTOS_closesocket( xSocket );
}
}

/*-----*/

void SocketRxTask( void *pvParameters )
{
    xSocket_t xSocket;

    for( ;; )
    {
        /* Wait to receive a socket that was created and connected by the Tx task. */

```



```
        xQueueReceive( xSocketPassingQueue, &xSocket, portMAX_DELAY );

        /* Keep receiving from the socket until the Tx task wants to close the socket.
        */

        while( TxTaskWantsToCloseSocket() == pdFALSE )
        {
            /* Receive then process data. */

            FreeRTOS_recv( xSocket, ... );

            ProcessReceivedData();
        }

        /* This is the Rx task's synchronization point. It reaches here only when it is
        no longer using the socket, and it is therefore safe for the Tx task to close the socket.
        */

        xEventGroupSync( ... );
    }
}
```

위의 의사 코드는 TCP 공유 소켓을 닫기 전에 어느 한쪽 작업이 더 이상 소켓을 사용하지 않도록 서로 동기화되는 작업 2개를 나타낸 것입니다.

이벤트 그룹을 사용하여 다음과 같이 동기화 지점을 생성할 수 있습니다.

- 이벤트 그룹에서 동기화에 참여해야 하는 각 작업에게 고유한 이벤트 비트가 할당됩니다.
- 각 작업은 동기화 지점에 도달할 때 고유한 이벤트 비트를 설정합니다.
- 고유한 이벤트가 설정되면 각 작업이 이벤트 그룹에서 차단되어 나머지 모든 동기화 작업을 나타내는 이벤트 비트 역시 설정될 때까지 대기합니다.

이번 시나리오에서는 xEventGroupSetBits() 및 xEventGroupWaitBits() API 함수를 사용할 수 없습니다. 이 두 함수를 사용하면 비트 설정(작업이 동기화 지점에 도달했다는 것을 표시)과 비트 테스트(나머지 동기화 작업의 동기화 지점 도달 여부를 결정)가 서로 다른 2가지 연산으로 실행됩니다. 이것이 문제가 되는 이유를 알아보기 위해 작업 A, 작업 B 및 작업 C가 다음과 같이 이벤트 그룹을 사용해 서로 동기화되는 시나리오를 가정합니다.

1. 작업 A와 작업 B가 이미 동기화 지점에 도달하였기 때문에 이벤트 비트가 이벤트 그룹에서 설정됩니다. 두 작업이 Blocked 상태에서 작업 C의 이벤트 비트가 설정될 때까지 대기합니다.
2. 작업 C가 동기화 지점에 도달한 후 xEventGroupSetBits()를 사용해 이벤트 그룹에서 비트를 설정합니다. 작업 C의 비트가 설정되면 작업 A와 작업 B가 Blocked 상태를 종료하고 이벤트 비트 3개를 모두 소거합니다.
3. 그러면 작업 C가 xEventGroupWaitBits()를 호출하여 이벤트 비트 3개가 모두 설정될 때까지 대기합니다. 하지만 이때는 이벤트 비트 3개가 모두 소거되고, 작업 A와 작업 B가 각각 동기화 지점에서 벗어난 이후이기 때문에 동기화에 실패하게 됩니다.

이벤트 그룹을 성공적으로 사용해 동기화 지점을 생성하려면 이벤트 비트 설정과 이후 이벤트 비트 테스트가 중단되지 않는 단일 연산으로 실행되어야 합니다. xEventGroupSync() API 함수는 이러한 목적으로 사용됩니다.

## xEventGroupSync() API 함수

xEventGroupSync()는 이벤트 그룹을 사용해 작업 2개 이상을 서로 동기화할 목적으로 사용됩니다. 이 함수를 호출하면 작업이 이벤트 그룹에서 이벤트 비트를 1개 이상 설정한 후 이벤트 비트 조합이 동일한 이벤트 그룹에서 중단되지 않는 단일 연산으로 설정될 때까지 대기합니다.

xEventGroupSync() uxBitsToWaitFor 파라미터는 호출 작업의 차단 해제 조건을 지정합니다. 차단 해제 조건이 충족되어 xEventGroupSync()가 반환한 경우에는 uxBitsToWaitFor에서 지정한 이벤트 비트가 xEventGroupSync()의 값 반환 이전에 0으로 다시 소거됩니다.

xEventGroupSync() API 함수 프로토타입은 다음과 같습니다.

```
EventBits_t xEventGroupSync( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet,
    const EventBits_t uxBitsToWaitFor, TickType_t xTicksToWait );
```

다음 표는 xEventGroupSync() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름	설명
xEventGroup	이벤트 비트가 설정 및 테스트되는 이벤트 그룹의 핸들입니다. 이벤트 그룹 핸들은 이벤트 그룹 생성에 사용되는 xEventGroupCreate() 호출을 통해 반환됩니다.
uxBitsToSet	이벤트 그룹에서 이벤트 비트(들)를 1로 설정하도록 지정하는 비트 마스크입니다. 이벤트 그룹의 값은 이벤트 그룹의 기존 값과 uxBitsToSet에서 전달되는 값을 비트 논리합으로 연산 처리하여 업데이트됩니다.  예를 들어 uxBitsToSet를 0x04(이진수 0100)로 설정하면 이벤트 비트 3이 설정되고(아직 설정되지 않은 경우) 이벤트 그룹에서 나머지 이벤트 비트는 모두 바뀌지 않습니다.
uxBitsToWaitFor	이벤트 그룹에서 테스트할 이벤트 비트(들)를 지정하는 비트 마스크입니다.  예를 들어 호출 작업이 이벤트 비트 0, 1 및 2가 이벤트 그룹에서 설정될 때까지 대기하는 경우에는 uxBitsToWaitFor를 0x07(이진수 111)로 설정합니다.
xTicksToWait	작업이 차단 해제 조건이 충족될 때까지 Blocked 상태로 대기해야 하는 최대 시간입니다.  xTicksToWait가 0이거나, 혹은 xEventGroupSync() 호출 시점에 차단 해제 조건이 충족되면 xEventGroupSync()가 즉시 값을 반환합니다.  차단 시간은 틱 주기로 지정되기 때문에 틱 주파수에 따라 절대 시간이 달라집니다. 밀리초 단위의 시간은 pdMS_TO_TICKS() 매크로를 사용해 틱 단위의 시간으로 변환할 수 있습니다.  xTicksToWait를 portMAX_DELAY로 설정하면 작업이 제한 시간 없이 무기한 대기하게 됩니다. 단,

	INCLUDE_vTaskSuspend가 FreeRTOSConfig.h에서 1로 설정되어 있어야 합니다.
반환 값	<p>호출 작업의 차단 해제 조건이 충족되어 xEventGroupSync()가 값을 반환한 경우에는 호출 작업의 차단 해제 조건이 충족된 시점(비트가 0으로 다시 자동 소거되기 이전 시점)의 이벤트 그룹 값이 반환됩니다. 이러한 경우에는 반환 값 역시 호출 작업의 차단 해제 조건을 충족합니다.</p> <p>xTicksToWait 파라미터에서 지정한 차단 시간이 지났기 때문에 xEventGroupSync()가 반환한 경우에는 차단 시간이 지난 시점의 이벤트 그룹 값이 반환됩니다. 이러한 경우에는 반환 값이 호출 작업의 차단 해제 조건을 충족하지 못합니다.</p>

## 작업 동기화(예제 23)

다음 코드는 단일 작업을 구현하는 인스턴스 3개를 동기화하는 방법을 나타낸 것입니다. 작업 파라미터는 작업이 xEventGroupSync() 호출 시 설정하는 이벤트 비트를 각 인스턴스에 전달할 때 사용됩니다.

작업은 xEventGroupSync() 호출 이전에, 그리고 xEventGroupSync() 호출로 값이 반환된 이후에 다시 메시지를 출력합니다. 각 메시지에는 타임스태프가 포함되어 실행 시퀀스를 출력 화면에서 확인할 수 있습니다. 의사 무작위 지연은 모든 작업이 동시에 동기화 지점에 도달하는 것을 방지하기 위해 사용됩니다.

```
static void vSyncingTask( void *pvParameters )
{
    const TickType_t xMaxDelay = pdMS_TO_TICKS( 4000UL );
    const TickType_t xMinDelay = pdMS_TO_TICKS( 200UL );

    TickType_t xDelayTime;

    EventBits_t uxThisTasksSyncBit;

    const EventBits_t uxAllSyncBits = ( mainFIRST_TASK_BIT | mainSECOND_TASK_BIT |
    mainTHIRD_TASK_BIT );

    /* Three instances of this task are created - each task uses a different event bit in
    the synchronization. The event bit to use is passed into each task instance using the task
    parameter. Store it in the uxThisTasksSyncBit variable. */

    uxThisTasksSyncBit = ( EventBits_t ) pvParameters;

    for( ;; )
    {
        /* Simulate this task taking some time to perform an action by delaying for a
        pseudo-random time. This prevents all three instances of this task from reaching the
        synchronization point at the same time, and so allows the example's behavior to be
        observed more easily. */

        xDelayTime = ( rand() % xMaxDelay ) + xMinDelay;

        vTaskDelay( xDelayTime );
    }
}
```

```
/* Print out a message to show this task has reached its synchronization point.
pcTaskGetTaskName() is an API function that returns the name assigned to the task when the
task was created. */

vPrintTwoStrings( pcTaskGetTaskName( NULL ), "reached sync point" );

/* Wait for all the tasks to have reached their respective synchronization points.
*/

xEventGroupSync( /* The event group used to synchronize. */ xEventGroup, /*
The bit set by this task to indicate it has reached the synchronization point. */
uxThisTasksSyncBit, /* The bits to wait for, one bit for each task taking part in the
synchronization. */ uxAllSyncBits, /* Wait indefinitely for all three tasks to reach the
synchronization point. */ portMAX_DELAY );

/* Print out a message to show this task has passed its synchronization point. As
an indefinite delay was used the following line will only be executed after all the tasks
reached their respective synchronization points. */

vPrintTwoStrings( pcTaskGetTaskName( NULL ), "exited sync point" );

}

}
```

main() 함수는 이벤트 그룹과 작업 3개를 모두 차례대로 생성한 후 스케줄러를 시작합니다. 다음 코드는 이 함수의 구현을 나타낸 것입니다.

```
/* Definitions for the event bits in the event group. */

#define mainFIRST_TASK_BIT ( 1UL << 0UL ) /* Event bit 0, set by the first task. */
#define mainSECOND_TASK_BIT( 1UL << 1UL ) /* Event bit 1, set by the second task. */
#define mainTHIRD_TASK_BIT ( 1UL << 2UL ) /* Event bit 2, set by the third task. */

/* Declare the event group used to synchronize the three tasks. */
EventGroupHandle_t xEventGroup;

int main( void )
{
    /* Before an event group can be used it must first be created. */
    xEventGroup = xEventGroupCreate();

    /* Create three instances of the task. Each task is given a different name, which
    is later printed out to give a visual indication of which task is executing. The event bit
    to use when the task reaches its synchronization point is passed into the task using the
    task parameter. */

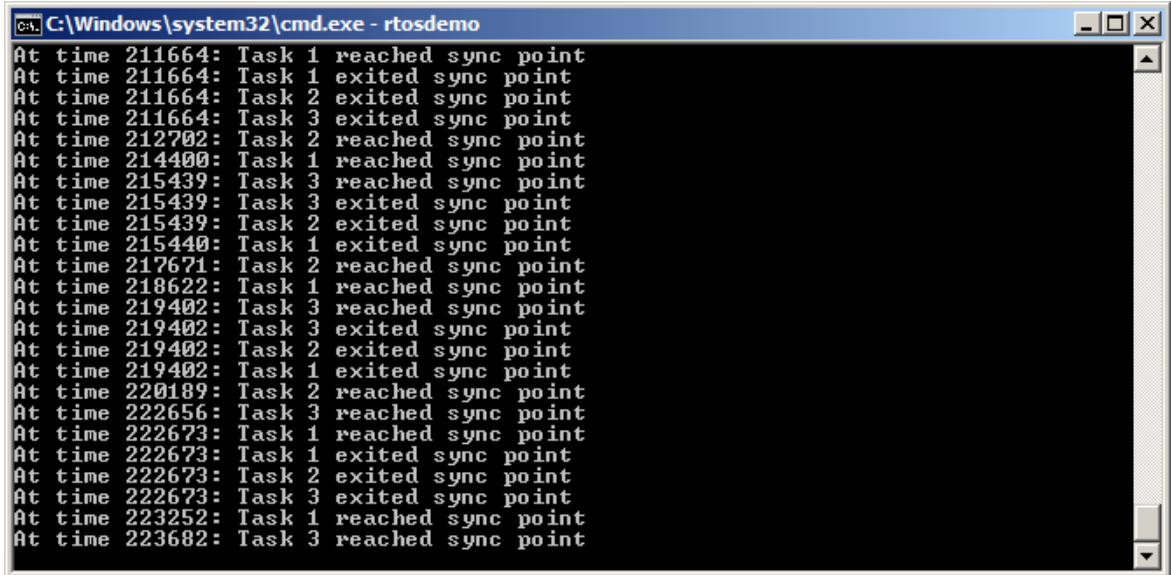
    xTaskCreate( vSyncingTask, "Task 1", 1000, mainFIRST_TASK_BIT, 1, NULL);
    xTaskCreate( vSyncingTask, "Task 2", 1000, mainSECOND_TASK_BIT, 1, NULL );
    xTaskCreate( vSyncingTask, "Task 3", 1000, mainTHIRD_TASK_BIT, 1, NULL );

    /* Start the scheduler so the created tasks start executing. */
    vTaskStartScheduler();

    /* As always, the following line should never be reached. */
}
```

```
for( ;; );  
  
return 0;  
  
}
```

예제 23을 실행하여 출력되는 화면은 다음과 같습니다. 작업마다 동기화 지점에 도달하는 시간이 다르지만 (의사 무작위) 동기화 지점에서 벗어나는 시간(마지막 작업이 동기화 지점에 도달한 시간)은 동일한 것을 볼 수 있습니다. 아래 그림은 FreeRTOS Windows 포트에서 실행할 경우의 예를 나타낸 것이기 때문에 실제로 실시간 동작을 나타내지는 않습니다(특히 Windows 시스템 호출을 사용해 콘솔에 출력하는 경우). 따라서 시간적으로 편차가 존재합니다.



```
C:\Windows\system32\cmd.exe - rtosdemo  
At time 211664: Task 1 reached sync point  
At time 211664: Task 1 exited sync point  
At time 211664: Task 2 exited sync point  
At time 211664: Task 3 exited sync point  
At time 212702: Task 2 reached sync point  
At time 214400: Task 1 reached sync point  
At time 215439: Task 3 reached sync point  
At time 215439: Task 3 exited sync point  
At time 215439: Task 2 exited sync point  
At time 215440: Task 1 exited sync point  
At time 217671: Task 2 reached sync point  
At time 218622: Task 1 reached sync point  
At time 219402: Task 3 reached sync point  
At time 219402: Task 3 exited sync point  
At time 219402: Task 2 exited sync point  
At time 219402: Task 1 exited sync point  
At time 220189: Task 2 reached sync point  
At time 222656: Task 3 reached sync point  
At time 222673: Task 1 reached sync point  
At time 222673: Task 1 exited sync point  
At time 222673: Task 2 exited sync point  
At time 222673: Task 3 exited sync point  
At time 223252: Task 1 reached sync point  
At time 223682: Task 3 reached sync point
```

## 작업 알림

이번 단원에서 다루는 내용은 다음과 같습니다.

- 작업의 알림 상태와 알림 값
- 세마포어 같은 통신 객체 대신에 작업 알림을 사용할 수 있는 방법과 시기
- 통신 객체 대신에 작업 알림을 사용하는 이점

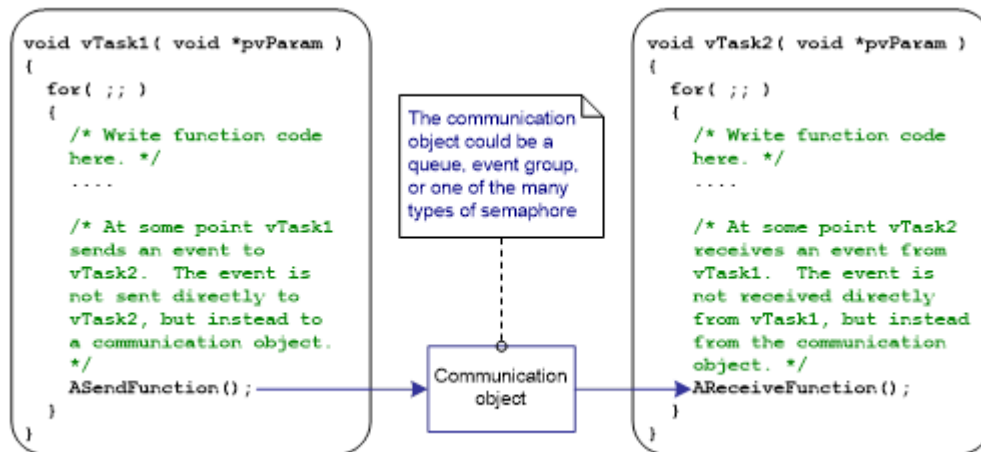
FreeRTOS를 사용하는 애플리케이션은 독립된 작업 집합의 구조로 작성되며, 이러한 독립 작업들은 상호 통신을 통해 전체적으로 유용한 시스템 기능을 제공할 수 있어야 합니다.

## 중재자 객체를 통한 통신

작업이 서로 통신할 수 있는 여러 가지 기존 방법들은 대기열과 이벤트 그룹, 그리고 다양한 유형의 세마포어 같은 통신 객체의 생성이 필요했습니다.

통신 객체를 사용할 때는 이벤트와 데이터가 수신 작업 또는 ISR에게 직접 전송되지 않고 통신 객체에게 전송됩니다. 마찬가지로 작업과 ISR 역시 이벤트를 전송한 작업이나 ISR에게서 직접 이벤트와 데이터를 수신하지 않고 통신 객체를 통해서 이벤트를 수신합니다.

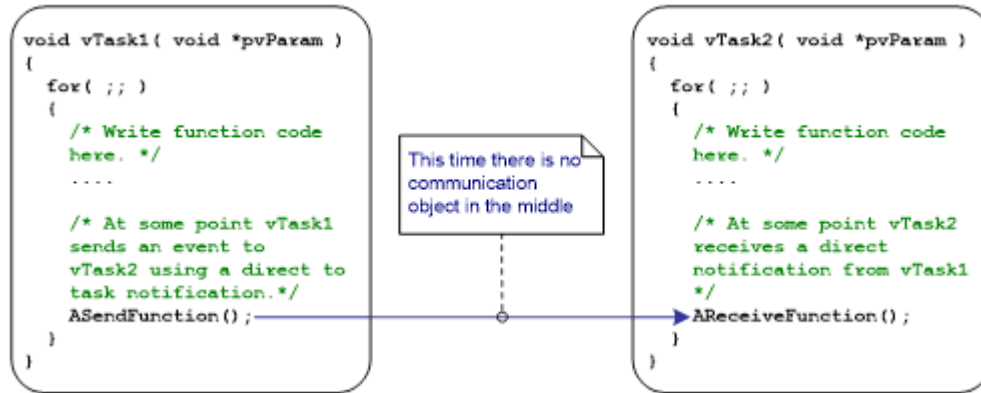
다음 그림은 작업 사이에서 이벤트를 전송할 때 사용되는 통신 객체를 나타낸 것입니다.



## 작업 알림: 작업 직접 통신

작업 알림을 사용하면 통신 객체를 따로 사용하지 않아도 작업이 다른 작업과 상호작용하는 동시에 ISR과 동기화하는 것도 가능합니다. 또한 작업 또는 ISR이 이벤트를 수신 작업에게 직접 전송할 수도 있습니다.

다음 그림은 작업 사이에서 이벤트를 직접 전송할 때 사용되는 작업 알림을 나타낸 것입니다.



작업 알림 기능은 선택 사항입니다. 작업 알림 기능을 추가하려면 FreeRTOSConfig.h에서 configUSE\_TASK\_NOTIFICATIONS를 1로 설정하십시오.

configUSE\_TASK\_NOTIFICATIONS를 1로 설정하면 작업마다 pending 또는 not-pending, 그리고 32비트 무부호 정수인 알림 값으로 구성되는 알림 상태가 표시됩니다. 작업이 알림을 수신하면 알림 상태가 pending으로 설정됩니다. 이후 작업이 알림 값을 읽어오면 알림 상태가 not-pending으로 설정됩니다.

작업은 옵션으로 추가되는 제한 시간을 이용해 알림 상태가 pending이 될 때까지 Blocked 상태로 대기할 수 있습니다.

## 작업 알림의 이점과 제한 사항

작업 알림을 사용해 이벤트 또는 데이터를 작업에 전송하면 대기열, 세마포어 또는 이벤트 그룹을 사용할 때보다 훨씬 빠릅니다.

마찬가지로 작업 알림을 사용해 이벤트 또는 데이터를 작업에 전송하면 대기열, 세마포어 또는 이벤트 그룹을 사용할 때보다 필요한 RAM 용량이 크게 줄어듭니다. 각 통신 객체(대기열, 세마포어 또는 이벤트 그룹)를 사용하려면 먼저 생성해야 하는 반면 작업 알림 기능을 활성화할 경우 작업 1개당 오버헤드가 RAM 8바이트로 고정되기 때문입니다.

## 작업 알림의 제한 사항

다음과 같은 시나리오에서는 작업 알림을 사용할 수 없습니다.

- 이벤트 또는 데이터를 ISR로 전송하는 경우

통신 객체를 사용하면 이벤트와 데이터를 ISR에서 작업으로, 혹은 작업에서 ISR로 전송할 수 있습니다.

하지만 작업 알림을 사용하면 이벤트와 데이터를 ISR에서 작업으로 전송할 수는 있지만 작업에서 ISR로 전송하지는 못합니다.

- 수신 작업을 둘 이상 활성화하는 경우

작업 또는 ISR이 핸들(대기열 핸들, 세마포어 핸들, 이벤트 그룹 핸들 등)을 알고 있다면 통신 객체에 액세스할 수 있습니다. 통신 객체로 전송된 이벤트 또는 데이터를 처리할 수 있는 작업과 ISR의 수에도 제한이 없습니다.

작업 알림은 수신 작업에게 직접 전송되기 때문에 알림이 전송되는 작업에서만 처리가 가능합니다. 하지만 이러한 제한은 대부분 경우 문제가 되지 않습니다. 동일한 통신 객체에게 전송하는 작업과 ISR이 다수일 때는 많지만 동일한 통신 객체에서 수신하는 작업과 ISR이 다수인 경우는 드물기 때문입니다.

- 다수의 데이터 항목을 버퍼에 저장하는 경우

대기열은 데이터 항목을 한 번에 1개 이상 저장할 수 있는 통신 객체입니다. 대기열에 전송되었지만 아직 대기열에서 수신되지 않은 데이터는 대기열 객체 내 버퍼에 저장됩니다.

작업 알림은 수신 작업의 알림 값을 업데이트하여 데이터를 작업으로 전송합니다. 이때 작업의 알림 값은 한 번에 하나만 저장됩니다.

- 다수의 작업에게 브로드캐스팅하는 경우

이벤트 그룹은 한 번에 다수의 작업에게 이벤트를 전송할 때 사용할 수 있는 통신 객체입니다.

작업 알림은 수신 작업에게 직접 전송되기 때문에 수신 작업에서만 처리가 가능합니다.

- 전송이 완료될 때까지 Blocked 상태로 대기하는 경우

통신 객체가 데이터 또는 이벤트를 더 이상 작성할 수 없는 일시적 상태일 경우(대기열이 가득 차서 데이터를 더 이상 대기열에 전송할 수 없는 경우 등) 객체에 작성하려는 작업은 옵션에 따라 Blocked 상태로 전환되어 쓰기 작업이 완료될 때까지 대기할 수 있습니다.

임의의 작업이 이미 대기 중인 알림이 있는 작업에게 작업 알림을 전송하려는 경우 전송 작업은 수신 작업이 알림 상태를 재설정할 때까지 Blocked 상태로 대기하지 못합니다. 이러한 제한 사항은 작업 알림이 사용되는 대부분 경우에서 문제가 되지 않습니다.

## 작업 알림 사용

작업 알림은 종종 이진 세마포어, 계수 세마포어, 이벤트 그룹, 그리고 간혹 대기열을 대신에 사용할 수 있는 매우 강력한 기능입니다.

## 작업 알림 API 옵션

xTaskNotify() API 함수는 작업 알림을 전송할 때, 그리고 xTaskNotifyWait() API 함수는 작업 알림을 수신할 때 사용할 수 있는 함수입니다.

하지만 대부분 경우 xTaskNotify() 및 xTaskNotifyWait() API 함수에서 제공되는 유연성은 필요하지 않습니다. 더욱 간단한 함수로도 충분합니다. xTaskNotifyGive() API 함수는 더욱 간단하지만 유연성에서는 xTaskNotify()에 미치지 못합니다. ulTaskNotifyTake() API 함수 역시 더욱 간단하지만 유연성에서는 xTaskNotifyWait()에 미치지 못합니다.

## xTaskNotifyGive() API 함수

xTaskNotifyGive()는 알림을 작업에게 직접 전송하여 수신 작업의 알림 값을 1씩 올립니다(추가합니다). xTaskNotifyGive()를 호출하면 수신 작업의 알림 상태가 아직 pending이 아닌 경우 pending으로 설정됩니다.

xTaskNotifyGive() API 함수를 입력하면 이진 또는 계수 세마포어를 대신에 작업 알림을 더욱 가볍고 빠르게 사용할 수 있습니다. xTaskNotifyGive() API 함수 프로토타입은 다음과 같습니다.

```
BaseType_t xTaskNotifyGive( TaskHandle_t xTaskToNotify );
```

다음 표는 xTaskNotifyGive() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
xTaskToNotify	알림이 전송되는 작업의 핸들입니다. 작업 핸들을 가져오는 방법에 대한 자세한 내용은 xTaskCreate()



	API 함수에서 pxCreatedTask 파라미터를 참조하십시오.
반환 값	xTaskNotifyGive()는 xTaskNotify()를 호출하는 매크로입니다. 매크로를 통해 xTaskNotify()로 전달되는 파라미터는 pdPASS가 유일한 반환 값이 될 수 있도록 설정됩니다. xTaskNotify()에 대해서는 이번 단원 후반부에서 자세하게 다룹니다.

## vTaskNotifyGiveFromISR() API 함수

vTaskNotifyGiveFromISR()은 xTaskNotifyGive()를 인터럽트 서비스 루틴에서 사용할 수 있는 버전입니다. vTaskNotifyGiveFromISR() API 함수 프로토타입은 다음과 같습니다.

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

다음 표는 vTaskNotifyGiveFromISR() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
xTaskToNotify	알림이 전송되는 작업의 핸들입니다. 작업 핸들을 가져오는 방법에 대한 자세한 내용은 xTaskCreate() API 함수에서 pxCreatedTask 파라미터를 참조하십시오.
pxHigherPriorityTaskWoken	<p>알림이 전송되는 작업이 알림을 수신할 때까지 Blocked 상태로 대기하는 경우 알림이 전송되면 작업이 Blocked 상태를 종료합니다.</p> <p>vTaskNotifyGiveFromISR()을 호출하여 작업의 Blocked 상태가 종료되었을 때 차단 해제된 작업의 우선순위가 현재 실행 중인 작업(중단된 작업)의 우선순위보다 높으면 내부적으로 vTaskNotifyGiveFromISR()이 <b>&lt;problematic&gt;*&lt;/problematic&gt;</b> pxHigherPriorityTaskWoken을 pdTRUE로 설정합니다.</p> <p>vTaskNotifyGiveFromISR()이 이 값을 pdTRUE로 설정하면 인터럽트 종료 전에 컨텍스트 전환이 이루어져야 합니다. 그러면 최고 우선순위의 Ready 상태 작업으로 인터럽트 복귀가 직접 이루어질 수 있습니다.</p> <p>모든 인터럽트-세이프 API 함수가 그렇듯이 pxHigherPriorityTaskWoken 파라미터를 사용하려면 먼저 pdFALSE로 설정해야 합니다.</p>

## ulTaskNotifyTake() API 함수

ulTaskNotifyTake() 함수에서는 작업이 알림 값이 0보다 커질 때까지 Blocked 상태로 대기하다가 값을 반환하기 전에 작업의 알림 값을 1씩 줄이거나(감산하거나) 소거합니다.

ulTaskNotifyTake() API 함수를 입력하면 이진 또는 계수 세마포어를 대신에 작업 알림을 더욱 가볍고 빠르게 사용할 수 있습니다. ulTaskNotifyTake() API 함수 프로토타입은 다음과 같습니다.

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

다음 표는 ulTaskNotifyTake() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
xClearCountOnExit	<p>xClearCountOnExit가 pdTRUE로 설정되면 ulTaskNotifyTake() 호출로 값이 반환되기 전에 호출 작업의 알림 값이 0으로 소거됩니다.</p> <p>xClearCountOnExit가 pdFALSE로 설정되었을 때 호출 작업의 알림 값이 0보다 커지면 ulTaskNotifyTake() 호출로 값이 반환되기 전에 호출 작업의 알림 값이 줄어듭니다.</p>
xTicksToWait	<p>호출 작업이 알림 값이 0보다 커질 때까지 Blocked 상태로 대기해야 하는 최대 시간입니다.</p> <p>차단 시간은 틱 주기로 지정되기 때문에 틱 주파수에 따라 절대 시간이 달라집니다. 밀리초 단위의 시간은 pdMS_TO_TICKS() 매크로를 사용해 틱으로 변환할 수 있습니다.</p> <p>xTicksToWait를 portMAX_DELAY로 설정하면 작업이 제한 시간 없이 무기한 대기하게 됩니다. 단, INCLUDE_vTaskSuspend가 FreeRTOSConfig.h에서 1로 설정되어 있어야 합니다.</p>
반환 값	<p>반환 값은 0으로 소거되거나 줄어들기 전에 xClearCountOnExit 파라미터의 값으로 지정되는 호출 작업의 알림 값입니다.</p> <p>차단 시간이 지정되어 있고(xTicksToWait가 0이 아닌 경우), 반환 값이 0이 아닌 경우에는 호출 작업이 Blocked 상태로 전환되어 알림 값이 0보다 커질 때까지 대기하고, 차단 시간이 지나기 전에 알림 값이 업데이트되었을 가능성이 있습니다.</p> <p>차단 시간이 지정되어 있고(xTicksToWait가 0이 아닌 경우), 반환 값이 0인 경우에는 호출 작업이 Blocked 상태로 전환되어 알림 값이 0보다 커질 때까지 대기하지만 이전에 지정된 차단 시간이 지나버린 것입니다.</p>

## 세마포어 대신에 작업 알림을 사용할 수 있는 방법 1(예제 24)

예제 16에서는 이진 세마포어를 사용해 작업을 인터럽트 서비스 루틴 내부에서 차단 해제하여 작업과 인터럽트를 효과적으로 동기화했습니다. 이번 예제에서는 예제 16의 기능을 그대로 따르지만 이진 세마포어 대신에 작업 직접 알림을 사용합니다.

다음 코드는 인터럽트와 동기화되는 작업의 구현체를 나타낸 것입니다. 예제 16에서 사용한 xSemaphoreTake() 호출이 ulTaskNotifyTake() 호출로 바뀌었습니다.

ulTaskNotifyTake() xClearCountOnExit 파라미터가 pdTRUE로 설정되어 수신 작업의 알림 값이 ulTaskNotifyTake() 반환 이전에 0으로 소거됩니다. 따라서 각 ulTaskNotifyTake() 호출 사이에서 이미 가능한 이벤트는 모두 처리해야 합니다. 예제 16에서는 이진 세마포어를 사용했기 때문에 대기 중인 이벤트 수를 하드웨어에서 확인해야 했지만 이 방법이 항상 가능한 것은 아닙니다. 이번 예제에서는 대기 중인 이벤트의 수가 ulTaskNotifyTake()에서 반환됩니다.

ulTaskNotifyTake 호출 사이에서 발생하는 인터럽트 이벤트는 작업의 알림 값으로 잠기고, 호출 작업에 이미 pending 상태의 알림이 있으면 ulTaskNotifyTake() 호출로 값이 즉시 반환됩니다.

```
/* The rate at which the periodic task generates software interrupts.*/
const TickType_t xInterruptFrequency = pdMS_TO_TICKS( 500UL );

static void vHandlerTask( void *pvParameters )
{
    /* xMaxExpectedBlockTime is set to be a little longer than the maximum expected time
    between events. */

    const TickType_t xMaxExpectedBlockTime = xInterruptFrequency + pdMS_TO_TICKS( 10 );
    uint32_t ulEventsToProcess;

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Wait to receive a notification sent directly to this task from the interrupt
        service routine. */

        ulEventsToProcess = ulTaskNotifyTake( pdTRUE, xMaxExpectedBlockTime );

        if( ulEventsToProcess != 0 )
        {
            /* To get here at least one event must have occurred. Loop here until all
            the pending events have been processed (in this case, just print out a message for each
            event). */

            while( ulEventsToProcess > 0 )
            {
                vPrintString( "Handler task - Processing event.\r\n" );

                ulEventsToProcess--;
            }
        }
        else
        {

```

```
        /* If this part of the function is reached, then an interrupt did not arrive
        within the expected time. In a real application, it might be necessary to perform some
        error recovery operations. */

        }

    }

}
```

소프트웨어 인터럽트를 생성하는 데 사용되는 주기적 작업은 인터럽트 생성 이전과 이후에 메시지를 출력합니다. 이를 통해 실행 시퀀스를 출력 화면에서 확인할 수 있습니다.

다음 코드는 인터럽트 핸들러는 나타낸 것입니다. 인터럽트 핸들러는 인터럽트 핸들링이 위임되는 작업에게 알림을 직접 전송할 뿐입니다.

```
static uint32_t ulExampleInterruptHandler( void )

{

    BaseType_t xHigherPriorityTaskWoken;

    /* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE because it
    will get set to pdTRUE inside the interrupt-safe API function if a context switch is
    required. */

    xHigherPriorityTaskWoken = pdFALSE;

    /* Send a notification directly to the task to which interrupt processing is being
    deferred. */

    vTaskNotifyGiveFromISR( /* The handle of the task to which the notification is
    being sent. The handle was saved when the task was created. */ xHandlerTask, /*
    xHigherPriorityTaskWoken is used in the usual way. */ &xHigherPriorityTaskWoken );

    /* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR(). If
    xHigherPriorityTaskWoken was set to pdTRUE inside vTaskNotifyGiveFromISR(), then calling
    portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still
    pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. The implementation of
    portYIELD_FROM_ISR() used by the Windows port includes a return statement, which is why
    this function does not explicitly return a value. */

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

코드를 실행하여 출력되는 화면은 다음과 같습니다.

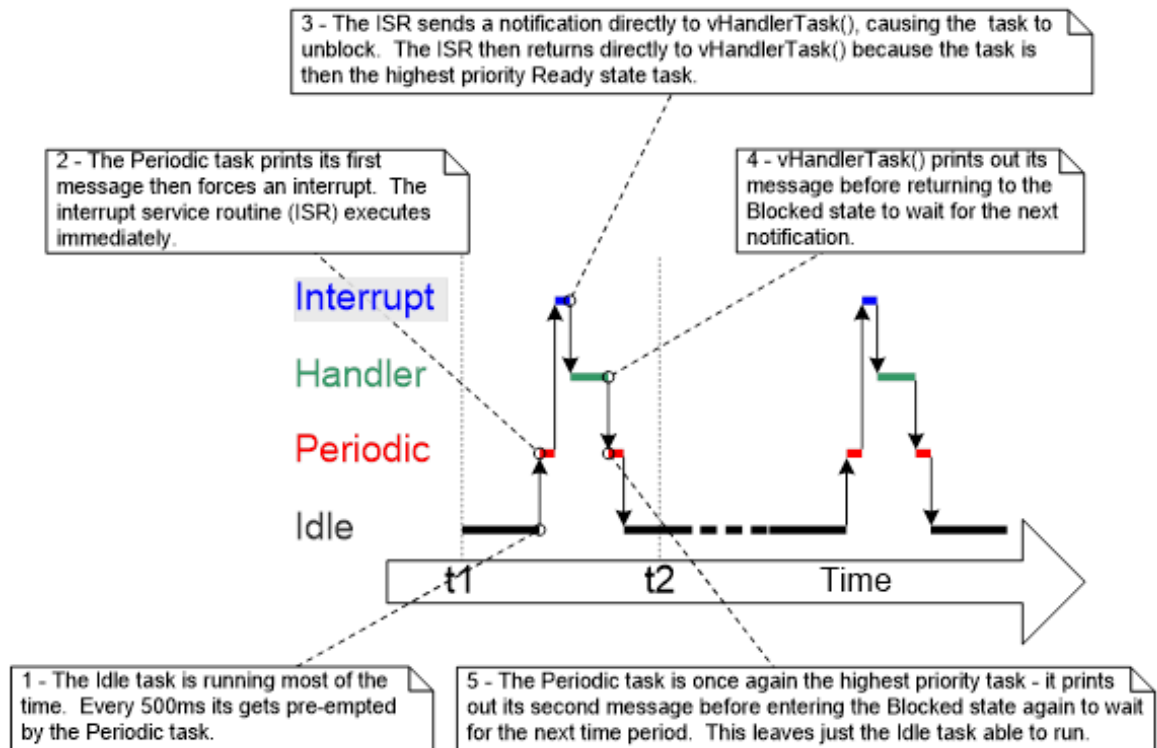
```
C:\WINDOWS\system32\cmd.exe - rtsdemo
Handler task - Processing event.
Periodic task - Interrupt generated.

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

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

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

예상한 것처럼 예제 16을 실행할 때 출력되는 화면과 동일합니다. 인터럽트 생성과 함께 vHandlerTask()가 Running 상태로 전환되기 때문에 주기적 작업의 출력을 분할하여 작업이 출력됩니다. 실행 시퀀스는 다음과 같습니다.



## 세마포어 대신에 작업 알림을 사용할 수 있는 방법 2(예제 25)

예제 24에서는 `ulTaskNotifyTake()` `xClearOnExit` 파라미터가 `pdTRUE`로 설정되었습니다. 예제 25에서는 예제 24를 약간 수정하여 `ulTaskNotifyTake()` `xClearOnExit` 파라미터가 `pdFALSE`로 설정되었을 때 동작에 대해서 설명하겠습니다.

`xClearOnExit`가 `pdFALSE`로 설정되었을 때는 `ulTaskNotifyTake()`를 호출해도 호출 작업의 알림 값이 0으로 소거되지 않고 1씩 감소할 뿐입니다. 따라서 알림 수는 발생한 이벤트 수와 처리된 이벤트 수의 차이가 됩니다. 이에 따라 `vHandlerTask()`의 구조를 다음과 같은 두 가지 방법으로 간소화할 수 있습니다.

1. 처리를 위해 대기하는 이벤트 수가 알림 값에 저장되기 때문에 로컬 변수에 저장할 필요가 없습니다.
2. 각 `ulTaskNotifyTake()` 호출 사이에서 이벤트를 하나씩 처리하기만 하면 됩니다.

`vHandlerTask()`의 구현체는 다음과 같습니다.

```
static void vHandlerTask( void *pvParameters )
{
    /* xMaxExpectedBlockTime is set to be a little longer than the maximum expected time
    between events. */

    const TickType_t xMaxExpectedBlockTime = xInterruptFrequency + pdMS_TO_TICKS( 10 );

    /* As per most tasks, this task is implemented within an infinite loop. */

    for( ;; )
    {
        /* Wait to receive a notification sent directly to this task from the interrupt
        service routine. The xClearCountOnExit parameter is now pdFALSE, so the task's
        notification value will be decremented by ulTaskNotifyTake() and not cleared to zero. */

        if( ulTaskNotifyTake( pdFALSE, xMaxExpectedBlockTime ) != 0 )
        {
            /* To get here, an event must have occurred. Process the event (in this case,
            just print out a message). */

            vPrintString( "Handler task - Processing event.\r\n" );
        }
        else
        {
            /* If this part of the function is reached, then an interrupt did not arrive
            within the expected time. In a real application, it might be necessary to perform some
            error recovery operations. */
        }
    }
}
```

여기에서는 설명을 목적으로 인터럽트 서비스 루틴 역시 인터럽트 1회당 작업 알림 2개 이상을 전송할 수 있도록 수정되었으며, 이때 높은 주기로 발생하는 다수의 인터럽트를 시뮬레이션합니다. 인터럽트 서비스 루틴의 구현체는 다음과 같습니다.

```
static uint32_t ulExampleInterruptHandler(void)
{
    BaseType_t xHigherPriorityTaskWoken;

    xHigherPriorityTaskWoken = pdFALSE;

    /* Send a notification to the handler task multiple times. The first give will unblock
    the task. The following gives are to demonstrate that the receiving task's notification
    value is being used to count (latch) events, allowing the task to process each event in
    turn. */

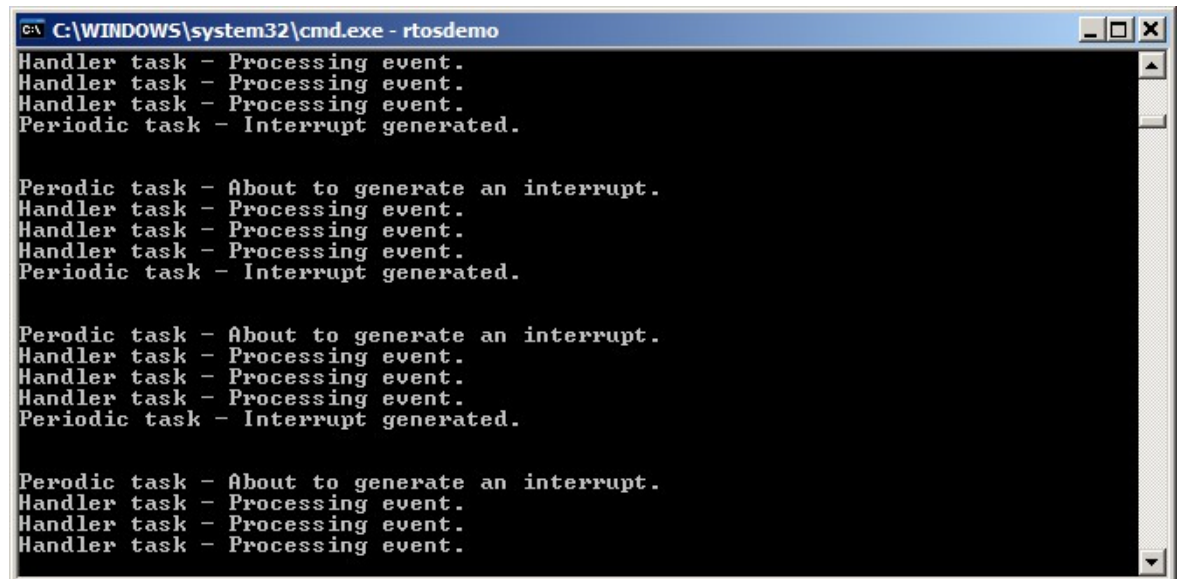
    vTaskNotifyGiveFromISR(xHandlerTask, &xHigherPriorityTaskWoken);

    vTaskNotifyGiveFromISR(xHandlerTask, &xHigherPriorityTaskWoken);

    vTaskNotifyGiveFromISR(xHandlerTask, &xHigherPriorityTaskWoken);

    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

출력되는 화면은 다음과 같습니다. 인터럽트가 생성될 때마다 vHandlerTask()가 이벤트 3개를 모두 처리하고 있는 모습을 볼 수 있습니다.



## xTaskNotify() 및 xTaskNotifyFromISR() API 함수

xTaskNotify()는 수신 작업의 알림 값을 다음 중 한 가지 방법으로 업데이트할 때 사용할 수 있는 버전으로 xTaskNotifyGive()보다 더욱 효과적입니다.

- 수신 작업의 알림 값을 1씩 올립니다. 이때 xTaskNotify()는 xTaskNotifyGive()와 동일합니다.
- 수신 작업의 알림 값에서 비트를 1개 이상 설정합니다. 이벤트 그룹을 대신해 작업의 알림 값을 더욱 가볍고 빠르게 사용할 수 있는 이유도 바로 여기에 있습니다.

- 수신 작업의 알림 값에 완전히 새로운 숫자를 입력합니다. 단, 수신 작업이 마지막 업데이트 이후 알림 값을 읽었다는 가정을 전제로 합니다. 이렇게 하면 작업의 알림 값이 길이가 1인 대기열과 비슷한 기능을 제공할 수 있습니다.
- 수신 작업이 마지막 업데이트 이후 알림 값을 읽지 않았다고 해도 수신 작업의 알림 값에 완전히 새로운 숫자를 입력합니다. 이렇게 하면 작업의 알림 값이 xQueueOverwrite() API 함수와 비슷한 기능을 제공할 수 있습니다. 이러한 결과에 따른 동작을 종종 사서함이라고 부릅니다.

xTaskNotify()는 xTaskNotifyGive()보다 더욱 유연하고 강력합니다. 동시에 사용하기에 조금 더 복잡하기도 합니다.

xTaskNotifyFromISR()은 xTaskNotify()를 인터럽트 서비스 루틴에서 사용할 수 있는 버전입니다. 따라서 pxHigherPriorityTaskWoken 파라미터가 추가됩니다.

xTaskNotify()를 호출하면 수신 작업의 알림 상태가 아직 pending이 아닌 경우 항상 pending으로 설정됩니다.

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

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

다음 표는 xTaskNotify() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
xTaskToNotify	알림이 전송되는 작업의 핸들입니다. 작업 핸들을 가져오는 방법에 대한 자세한 내용은 xTaskCreate() API 함수에서 pxCreatedTask 파라미터를 참조하십시오.
ulValue	ulValue의 사용 방식은 eNotifyAction 값에 따라 달라집니다. 표 52를 참조하십시오.
eNotifyAction	수신 작업의 알림 값 업데이트 방식을 지정하는 열거형 파라미터입니다. 다음 표를 참조하십시오.
반환 값	xTaskNotify()는 아래에서 언급하는 한 가지 경우를 제외하고 pdPASS를 반환합니다.

다음 표는 xTaskNotify() eNotifyAction 파라미터의 유효 값과 그에 따라 수신 작업의 알림 값에 미치는 영향을 나열한 것입니다.

eNotifyAction 값	수신 작업에 미치는 영향
eNoAction	수신 작업의 알림 값이 업데이트 없이 pending으로 설정됩니다. xTaskNotify() ulValue 파라미터는 사용되지 않습니다. 이진 세마포어를 대신해 작업 알림을 더욱 빠르고 가볍게 사용할 수 있는 이유도 eNoAction 작업 때문입니다.
eSetBits	수신 작업의 알림 값이 xTaskNotify() ulValue 파라미터로 전달되는 값과 함께 비트 논리합으로 연산 처리됩니다. 예를 들어 ulValue가 0x01로 설정되면 수신 작업의 알림 값에서 비트 0이 설정됩니다. 또는



	<p>ulValue가 0x06(이진수 0110)로 설정되면 수신 작업의 알림 값에서 비트 1과 비트 2가 설정됩니다.</p> <p>이벤트 그룹을 대신해 작업 알림을 더욱 빠르고 가볍게 사용할 수 있는 이유도 eSetBits 작업 때문입니다.</p>
eIncrement	<p>수신 작업의 알림 값이 증가합니다. xTaskNotify() ulValue 파라미터는 사용되지 않습니다.</p> <p>이진 또는 계수 세마포어를 대신해 작업 알림을 더욱 빠르고 가볍게 사용할 수 있는 이유도 eIncrement 작업 때문입니다. 이 파라미터는 더욱 간단한 xTaskNotifyGive() API 함수와 동일합니다.</p>
eSetValueWithoutOverwrite	<p>xTaskNotify() 호출 이전에 수신 작업에 대기 중인 알림이 있었다면 아무런 작업도 없이 xTaskNotify()가 pdFAIL을 반환합니다.</p> <p>xTaskNotify() 호출 이전에 수신 작업에 대기 중인 알림이 없었다면 수신 작업의 알림 값이 xTaskNotify() ulValue 파라미터에서 전달되는 값으로 설정됩니다.</p>
eSetValueWithOverwrite	<p>xTaskNotify() 호출 이전에 수신 작업에서 대기 중인 알림 유무에 상관없이 수신 작업의 알림 값이 xTaskNotify() ulValue 파라미터에서 전달되는 값으로 설정됩니다.</p>

## xTaskNotifyWait() API 함수

xTaskNotifyWait()는 ulTaskNotifyTake()보다 더욱 효과적인 버전의 함수입니다. 이 함수에서는 호출 작업의 알림 상태가 아직 pending이 아니라면 pending이 될 때까지 선택 사항으로 제한 시간을 지정해 작업이 대기할 수 있습니다. xTaskNotifyWait()는 함수 진입 및 종료 시 호출 작업의 알림 값에서 비트를 소거할 수 있는 옵션을 제공합니다.

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

다음 표는 xTaskNotifyWait() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
ulBitsToClearOnEntry	<p>xTaskNotifyWait() 호출 이전에 호출 작업에 대기 중인 알림이 없었다면 ulBitsToClearOnEntry에서 설정된 모든 비트가 함수 진입 시 작업의 알림 값에서 소거됩니다.</p> <p>예를 들어 ulBitsToClearOnEntry가 0x01로 설정되면 수신 작업의 알림 값에서 비트 0이 소거됩니다. 또는 ulBitsToClearOnEntry가 0xffffffff(ULONG_MAX)로 설정되면 작업의 알림 값에서 모든 비트가 소거되어 실제 값도 0으로 소거됩니다.</p>
ulBitsToClearOnExit	<p>호출 작업이 알림을 수신하였거나, 혹은 xTaskNotifyWait() 호출 시 대기 중인 알림이 이미</p>

	<p>있었던 이유로 xTaskNotifyWait()를 종료하는 경우에는 ulBitsToClearOnExit에서 설정된 모든 비트가 xTaskNotifyWait() 함수 종료 이전에 작업의 알림 값에서 소거됩니다.</p> <p>비트는 작업의 알림 값이 먼저 &lt;problematic&gt;*&lt;/problematic&gt; pulNotificationValue에 저장된 후에 소거됩니다(아래 pulNotificationValue에 대한 설명 참조).</p> <p>예를 들어 ulBitsToClearOnExit가 0x03으로 설정되면 함수 종료 이전에 작업의 알림 값에서 비트 0과 비트 1이 소거됩니다.</p> <p>또는 ulBitsToClearOnExit가 0xffffffff(ULONG_MAX)로 설정되면 작업의 알림 값에서 모든 비트가 소거되어 실제 값도 0으로 소거됩니다.</p>
pulNotificationValue	<p>작업의 알림 값을 전달하는 데 사용됩니다. pulNotificationValue에 복사되는 값은 &lt;problematic&gt;*&lt;/problematic&gt; ulBitsToClearOnExit 설정으로 인해 모든 비트가 소거되기 이전 작업의 알림 값입니다.</p> <p>pulNotificationValue 파라미터는 선택 사항이기 때문에 필요 없다면 NULL로 설정해도 좋습니다.</p>
xTicksToWait	<p>호출 작업이 알림 상태가 pending이 될 때까지 Blocked 상태로 대기해야 하는 최대 시간입니다.</p> <p>차단 시간은 틱 주기로 지정되기 때문에 틱 주파수에 따라 절대 시간이 달라집니다. 밀리초 단위의 시간은 pdMS_TO_TICKS() 매크로를 사용해 틱 단위의 시간으로 변환할 수 있습니다.</p> <p>xTicksToWait를 portMAX_DELAY로 설정하면 작업이 제한 시간 없이 무기한 대기하게 됩니다. 단, INCLUDE_vTaskSuspend가 FreeRTOSConfig.h에서 1로 설정되어 있어야 합니다.</p>

반환 값	<p>가능한 반환 값은 다음 두 가지입니다.</p> <ol style="list-style-type: none"><li>pdTRUE</li></ol> <p>이는 알림이 수신되었거나, 혹은 xTaskNotifyWait() 호출 시 호출 작업에 이미 대기 중인 알림이 있었기 때문에 xTaskNotifyWait()가 반환했다는 것을 나타냅니다.</p> <p>차단 시간이 지정된 경우에는(xTicksToWait가 0이 아닌 경우) 알림 상태가 pending이 될 때까지 호출 작업이 Blocked 상태로 전환되어 대기하지만 차단 시간이 지나기 전에 알림 상태가 pending으로 설정 되었을 가능성이 있습니다.</p> <ol style="list-style-type: none"><li>pdFALSE</li></ol> <p>xTaskNotifyWait()가 작업 알림을 수신한 호출 작업 없이 반환했다는 것을 나타냅니다.</p> <p>xTicksToWait가 0으로 지정되지 않은 경우에는 호출 작업이 Blocked 상태로 전환되어 알림 상태가 pending이 될 때까지 대기하지만 이전에 지정된 차단 시간이 먼저 지나버린 것입니다.</p>
------	---

## 주변 장치 드라이버에서 사용되는 작업 알림: UART 예제

주변 장치 드라이버 라이브러리는 하드웨어 인터페이스를 통해 공통 작업을 수행하는 함수를 제공합니다. 이러한 라이브러리를 위한 주변 장치로는 UART(Universal Asynchronous Receivers and Transmitters), SPI(Serial Peripheral Interface) 포트, ADC(Analog-to-Digital Converter), Ethernet 포트 등이 있습니다. 라이브러리에서 공통으로 제공되는 함수로는 주변 장치를 초기화하는 함수, 데이터를 주변 장치로 전송하는 함수, 그리고 주변 장치에서 데이터를 수신하는 함수가 있습니다.

주변 장치에서 수행하는 몇 가지 작업은 마칠 때까지 비교적 오랜 시간이 걸립니다. 여기에는 고정밀 ADC 변환과 UART를 통한 대용량 데이터 패킷 전송 등이 있습니다. 이 두 가지 경우에는 드라이버 라이브러리 함수를 구현해 주변 장치의 상태 레지스터를 폴링하여(반복적으로 읽어와서) 작업 완료 시간을 알아낼 수 있습니다. 하지만 이러한 방식의 폴링은 생산적인 처리 작업 없이 프로세서의 시간을 100% 모두 사용하기 때문에 거의 항상 소모적일 뿐입니다. 특히 멀티태스킹 시스템에서는 이러한 낭비에 따른 대가가 큼니다. 주변 장치를 폴링하는 작업이 낮은 우선순위의 작업 실행을 막아 생산적인 처리 작업을 수행하지 못할 수도 있기 때문입니다.

이렇게 처리 시간을 낭비할 가능성을 배제하기 위해서는 효율적인 RTOS 인식 장치 드라이버가 인터럽트 중심이 되어 긴 작업을 시작하는 작업에게 작업이 끝날 때까지 Blocked 상태로 전환되어 대기할 수 있는 옵션을 제공해야 합니다. 이렇게 하면 긴 작업을 수행하는 작업이 Blocked 상태일 때 낮은 우선순위의 작업이 실행되어 어떠한 작업도 처리 시간을 생산적 목적 없이 사용하지 않습니다.

RTOS 인식 드라이버 라이브러리가 이진 세마포어를 사용해 작업을 Blocked 상태로 전환하는 것은 흔한 일입니다. 이 기법에 대해서는 다음 의사 코드에서 UART 포트를 통해 데이터를 전송하는 RTOS 인식 라이브러리 함수에 대해 간략히 소개하면서 설명하겠습니다. 다음 코드 목록에서,

- xUART는 UART 주변 장치를 설명하는 구조로서 상태 정보가 여기에 저장됩니다. 이 구조에서 xTxSemaphore 멤버는 SemaphoreHandle\_t 형식의 변수입니다. 또한 세마포어가 이미 생성되었다는 가정을 전제로 합니다.

- xUART\_Send() 함수에는 상호 배제 로직이 포함되어 있지 않습니다. 작업 2개 이상이 xUART\_Send() 함수를 사용하는 경우에는 애플리케이션 개발자가 애플리케이션 자체 내에서 상호 배제를 관리해야 합니다. 예를 들어 작업이 xUART\_Send()를 호출하려면 뮤텍스를 가져와야 할 수도 있습니다.
- xSemaphoreTake() API 함수는 UART 전송이 시작된 후 호출 작업을 Blocked 상태로 전환하는 데 사용됩니다.
- xSemaphoreGiveFromISR() API 함수는 전송이 완료된 후, 즉 UART 주변 장치의 전송 종단 인터럽트 서비스 루틴이 실행될 때 작업의 Blocked 상태를 종료하는 데 사용됩니다.

```
/* Driver library function to send data to a UART. */

BaseType_t xUART_Send( xUART *pxUARTInstance, uint8_t *pucDataSource, size_t uxLength )
{
    BaseType_t xReturn;

    /* Ensure the UART's transmit semaphore is not already available by attempting to take
    the semaphore without a timeout. */

    xSemaphoreTake( pxUARTInstance->TxSemaphore, 0 );

    /* Start the transmission. */

    UART_low_level_send( pxUARTInstance, pucDataSource, uxLength );

    /* Block on the semaphore to wait for the transmission to complete. If the semaphore is
    obtained, then xReturn will get set to pdPASS. If the semaphore take operation times out,
    then xReturn will get set to pdFAIL. If the interrupt occurs between UART_low_level_send()
    being called and xSemaphoreTake() being called, then the event will be latched in the
    binary semaphore, and the call to xSemaphoreTake() will return immediately. */

    xReturn = xSemaphoreTake( pxUARTInstance->TxSemaphore, pxUARTInstance->TxTimeout );

    return xReturn;
}

/*-----*/

/* The service routine for the UART's transmit end interrupt, which executes after the last
byte has been sent to the UART. */

void xUART_TransmitEndISR( xUART *pxUARTInstance )
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Clear the interrupt. */

    UART_low_level_interrupt_clear( pxUARTInstance );

    /* Give the Tx semaphore to signal the end of the transmission. If a task is Blocked
    waiting for the semaphore, then the task will be removed from the Blocked state. */

    xSemaphoreGiveFromISR( pxUARTInstance->TxSemaphore, &xHigherPriorityTaskWoken );

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

이번 코드에서 설명하는 기법은 효과가 있어서 널리 사용되지만 아래와 같이 몇 가지 단점이 있습니다.

- 라이브러리가 다수의 세마포어를 사용하여 필요한 RAM 용량이 증가합니다.
- 세마포어는 생성될 때까지 사용할 수 없기 때문에 세마포어를 사용하는 라이브러리 역시 명시적으로 초기화될 때까지 사용하지 못합니다.
- 세마포어는 광범위한 사용 사례에 적용되는 일반 객체입니다. 여기에는 작업 수에 제한 없이 세마포어를 사용할 수 있을 때까지 Blocked 상태로 대기할 수 있는 로직과 세마포어 사용이 가능해졌을 때 Blocked 상태를 종료할 작업을 (결정적 방식으로) 선택할 수 있는 로직이 포함됩니다. 이러한 로직을 실행하려면 유한한 시간이 필요합니다. 이번 코드의 시나리오에서는 처리 오버헤드가 필요하지 않습니다. 언제든지 세마포어를 사용할 수 있을 때까지 대기하는 작업이 1개로 제한되기 때문입니다.

다음 코드는 이진 세마포어 대신에 작업 알림을 사용해 이러한 단점을 해결하는 방법을 설명한 것입니다.

참고: 라이브러리가 작업 알림을 사용하는 경우에는 라이브러리 함수를 호출하면 호출 작업의 알림 상태 및 값이 바뀔 수 있다고 라이브러리의 설명서에 알기 쉽게 명시해야 합니다.

다음 코드에서,

- xUART 구조에서 xTxSemaphore 멤버는 xTaskToNotify 멤버로 바뀌었습니다. xTaskToNotify는 TaskHandle\_t 형식의 변수이며, UART 작업이 끝날 때까지 대기하는 작업의 핸들을 저장하는 데 사용됩니다.
- xTaskGetCurrentTaskHandle() FreeRTOS API 함수는 Running 상태의 작업 핸들을 가져오는 데 사용됩니다.
- 라이브러리는 FreeRTOS 객체를 생성하지 않기 때문에 RAM 오버헤드가 발생하지 않을 뿐만 아니라 명시적으로 초기화할 필요도 없습니다.
- UART 작업이 끝날 때까지 대기하는 작업에게 작업 알림이 직접 전송되기 때문에 로직을 따로 실행할 필요가 없습니다.

xUART 구조에서 xTaskToNotify 멤버는 작업과 인터럽트 서비스 루틴 모두에서 액세스가 가능하기 때문에 프로세서가 다음과 같이 값을 업데이트하는 방식에 대해서 고려해야 합니다.

- xTaskToNotify가 메모리 쓰기 작업 한 번으로 업데이트되는 경우에는 다음 코드에서 정확히 나타내고 있듯이 임계 영역 외부에서 업데이트할 수 있습니다. 예를 들어 xTaskToNotify가 32비트 변수(TaskHandle\_t가 32비트 형식이었음)이고, FreeRTOS가 실행되는 프로세서가 32비트 프로세서인 경우가 여기에 해당합니다.
- xTaskToNotify를 업데이트하는 데 메모리 쓰기 작업이 두 번 이상 필요한 경우에는 xTaskToNotify를 임계 영역 내부에서만 업데이트해야 합니다. 그렇지 않으면 일치하지 않는 상태일 때 인터럽트 서비스 루틴이 xTaskToNotify에 액세스할 수 있습니다. 예를 들어 xTaskToNotify가 32비트 변수이고, FreeRTOS가 실행되는 프로세서가 16비트 프로세서인 경우가 여기에 해당합니다. 이때는 32비트를 모두 업데이트하려면 16비트 메모리 쓰기 작업이 두 번 필요하기 때문입니다.

FreeRTOS 구현체 내부에서는 TaskHandle\_t가 포인터이므로 sizeof( TaskHandle\_t )는 항상 sizeof( void \* )와 동일합니다.

```
/* Driver library function to send data to a UART. */

BaseType_t xUART_Send( xUART *pxUARTInstance, uint8_t *pucDataSource, size_t uxLength )
{
    BaseType_t xReturn;

    /* Save the handle of the task that called this function. The book text contains notes
    as to whether the following line needs to be protected by a critical section or not. */

    pxUARTInstance->xTaskToNotify = xTaskGetCurrentTaskHandle();
```

```
/* Ensure the calling task does not already have a notification pending by calling
ulTaskNotifyTake() with the xClearCountOnExit parameter set to pdTRUE, and a block time of
0 (don't block). */

ulTaskNotifyTake( pdTRUE, 0 );

/* Start the transmission. */

UART_low_level_send( pxUARTInstance, pucDataSource, uxLength );

/* Block until notified that the transmission is complete. If the notification is
received, then xReturn will be set to 1 because the ISR will have incremented this task's
notification value to 1 (pdTRUE). If the operation times out, then xReturn will be 0
(pdFALSE) because this task's notification value will not have been changed since it was
cleared to 0 above. If the ISR executes between the calls to UART_low_level_send() and
the call to ulTaskNotifyTake(), then the event will be latched in the task's notification
value, and the call to ulTaskNotifyTake() will return immediately.*/

xReturn = ( BaseType_t ) ulTaskNotifyTake( pdTRUE, pxUARTInstance->xTxTimeout );

return xReturn;
}

/*-----*/

/* The ISR that executes after the last byte has been sent to the UART. */

void xUART_TransmitEndISR( xUART *pxUARTInstance )
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* This function should not execute unless there is a task waiting to be notified.
    Test this condition with an assert. This step is not strictly necessary, but will aid
    debugging. configASSERT() is described in Developer Support.*/

    configASSERT( pxUARTInstance->xTaskToNotify != NULL );

    /* Clear the interrupt. */

    UART_low_level_interrupt_clear( pxUARTInstance );

    /* Send a notification directly to the task that called xUART_Send(). If the task
    is Blocked waiting for the notification, then the task will be removed from the Blocked
    state. */

    vTaskNotifyGiveFromISR( pxUARTInstance->xTaskToNotify, &xHigherPriorityTaskWoken );

    /* Now there are no tasks waiting to be notified. Set the xTaskToNotify member of the
    xUART structure back to NULL. This step is not strictly necessary, but will aid debugging.
    */

    pxUARTInstance->xTaskToNotify = NULL;

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

다음 의사 코드에서도 UART 포트를 통해 데이터를 수신하는 RTOS 인식 라이브러리 함수에 대해 간략히 소개하면서 설명하고 있지만 작업 알림은 수신 함수의 세마포어도 대체할 수 있습니다.

- xUART\_Receive() 함수에는 상호 배제 로직이 포함되어 있지 않습니다. 작업 2개 이상이 xUART\_Receive() 함수를 사용하는 경우에는 애플리케이션 개발자가 애플리케이션 자체 내에서 상호 배제를 관리해야 합니다. 예를 들어 작업이 xUART\_Receive()를 호출하려면 뮉텍스를 가져와야 할 수도 있습니다.
- UART의 수신 인터럽트 서비스 루틴은 UART에 수신되는 문자를 RAM 버퍼에 저장합니다. xUART\_Receive() 함수는 RAM 버퍼에서 문자를 반환합니다.
- xUART\_Receive() uxWantedBytes 파라미터는 수신할 문자 수를 지정하는 데 사용됩니다. RAM버퍼에 요청한 문자 수가 아직 저장되어 있지 않으면 호출 함수가 Blocked 상태로 전환되어 버퍼의 문자 수가 늘어났다는 사실을 알 때까지 대기합니다. while() 루프는 수신 버퍼에 요청한 문자 수가 저장되거나, 혹은 제한 시간이 지날 때까지 이러한 시퀀스를 반복하는 데 사용됩니다.
- 호출 작업은 두 번 이상 Blocked 상태로 전환될 수 있습니다. 따라서 제한 시간을 조정할 때는 xUART\_Receive() 호출 이후 경과 시간을 고려해야 합니다. 또한 xUART\_Receive() 내부에서 걸리는 총 시간이 xUART 구조의 xRxTimeout 멤버에서 지정하는 제한 시간을 초과해서도 안 됩니다. 제한 시간은 FreeRTOS vTaskSetTimeOutState() 및 xTaskCheckForTimeOut() 헬퍼 함수를 사용해 조정합니다.

```
/* Driver library function to receive data from a UART. */
size_t xUART_Receive( xUART *pxUARTInstance, uint8_t *pucBuffer, size_t uxWantedBytes )
{
    size_t uxReceived = 0;

    TickType_t xTicksToWait;

    TimeOut_t xTimeOut;

    /* Record the time at which this function was entered. */

    vTaskSetTimeOutState( &xTimeOut );

    /* xTicksToWait is the timeout value. It is initially set to the maximum receive
    timeout for this UART instance. */

    xTicksToWait = pxUARTInstance->xRxTimeout;

    /* Save the handle of the task that called this function. */
    pxUARTInstance->xTaskToNotify = xTaskGetCurrentTaskHandle();

    /* Loop until the buffer contains the wanted number of bytes or a timeout occurs. */
    while( UART_bytes_in_rx_buffer( pxUARTInstance ) < uxWantedBytes )
    {
        /* Look for a timeout, adjusting xTicksToWait to account for the time spent in this
        function so far. */

        if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) != pdFALSE )
        {
            /* Timed out before the wanted number of bytes were available, exit the loop.
            */

            break;
        }
    }
}
```

```
/* The receive buffer does not yet contain the required amount of bytes. Wait for
a maximum of xTicksToWait ticks to be notified that the receive interrupt service routine
has placed more data into the buffer. It does not matter if the calling task already had a
notification pending when it called this function. If it did, it would just iterate around
this while loop one extra time. */

    ulTaskNotifyTake( pdTRUE, xTicksToWait );

}

/* No tasks are waiting for receive notifications, so set xTaskToNotify back to NULL.
*/

pxUARTInstance->xTaskToNotify = NULL;

/* Attempt to read uxWantedBytes from the receive buffer into pucBuffer. The actual
number of bytes read (which might be less than uxWantedBytes) is returned. */

uxReceived = UART_read_from_receive_buffer( pxUARTInstance, pucBuffer, uxWantedBytes );

return uxReceived;

}

/*-----*/

/* The interrupt service routine for the UART's receive interrupt */

void xUART_ReceiveISR( xUART *pxUARTInstance )
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Copy received data into this UART's receive buffer and clear the interrupt. */

    UART_low_level_receive( pxUARTInstance );

    /* If a task is waiting to be notified of the new data, then notify it now. */

    if( pxUARTInstance->xTaskToNotify != NULL )
    {
        vTaskNotifyGiveFromISR( pxUARTInstance->xTaskToNotify,
                                &xHigherPriorityTaskWoken );

        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
    }
}
```

## 주변 장치 드라이버에서 사용되는 작업 알림: ADC 예제

이전 단원에서는 `vTaskNotifyGiveFromISR()`을 사용하여 작업 알림을 인터럽트에서 작업으로 전송하는 방법에 대해서 알아보았습니다. `vTaskNotifyGiveFromISR()`은 간단하게 사용할 수 있는 함수이지만 기능이 제한적이어서 작업 알림을 무의미한 이벤트 형태로 전송할 뿐 데이터를 전송하지는 못합니다. 이번 단원에서는 `xTaskNotifyFromISR()`을 사용해 데이터를 작업 알림 이벤트와 함께 전송하는 방법에 대해서 설명하겠습니다.



다. 이 기법에 대해서는 다음 의사 코드에서 아날로그-디지털 컨버터(ADC)에서 사용되는 RTOS 인식 인터럽트 서비스 루틴에 대해 간략히 소개하면서 설명합니다.

- ADC 변환은 적어도 50밀리초마다 시작된다는 가정을 전제로 합니다.
- ADC\_ConversionEndISR()는 ADC의 변환 중단 인터럽트, 즉 새로운 ADC 값이 생성될 때마다 실행되는 인터럽트를 위한 인터럽트 서비스 루틴입니다.
- vADCTask()에서 구현되는 작업은 ADC에서 생성된 값을 하나씩 처리합니다. 단, 작업 생성 시 작업 핸들이 xADCTaskToNotify에 저장되었다는 가정을 전제로 합니다.
- ADC\_ConversionEndISR()은 xTaskNotifyFromISR()을 사용해 eAction 파라미터를 eSetValueWithoutOverwrite로 설정한 후 작업 알림을 vADCTask() 작업에게 전송하고 ADC 변환 결과를 작업의 알림 값에 작성합니다.
- vADCTask() 작업은 xTaskNotifyWait()를 사용해 새로운 ADC 값이 생성되었다는 사실을 알 때까지 대기하다가 ADC 변환 결과를 알림 값에서 가져옵니다.

```
/* A task that uses an ADC. */

void vADCTask( void *pvParameters )
{
    uint32_t ulADCValue;

    BaseType_t xResult;

    /* The rate at which ADC conversions are triggered. */
    const TickType_t xADCConversionFrequency = pdMS_TO_TICKS( 50 );

    for( ;; )
    {
        /* Wait for the next ADC conversion result. */

        xResult = xTaskNotifyWait( /* The new ADC value will overwrite the old value, so
there is no need to clear any bits before waiting for the new notification value. */ 0, /*
Future ADC values will overwrite the existing value, so there is no need to clear any bits
before exiting xTaskNotifyWait(). */ 0, /* The address of the variable into which the
task's notification value (which holds the latest ADC conversion result) will be copied.
*/ &ulADCValue, /* A new ADC value should be received every xADCConversionFrequency ticks.
*/ xADCConversionFrequency * 2 );

        if( xResult == pdPASS )
        {
            /* A new ADC value was received. Process it now. */

            ProcessADCResult( ulADCValue );

        }
        else
        {
            /* The call to xTaskNotifyWait() did not return within the expected time.
Something must be wrong with the input that triggers the ADC conversion or with the ADC
itself. Handle the error here. */

        }
    }
}
```

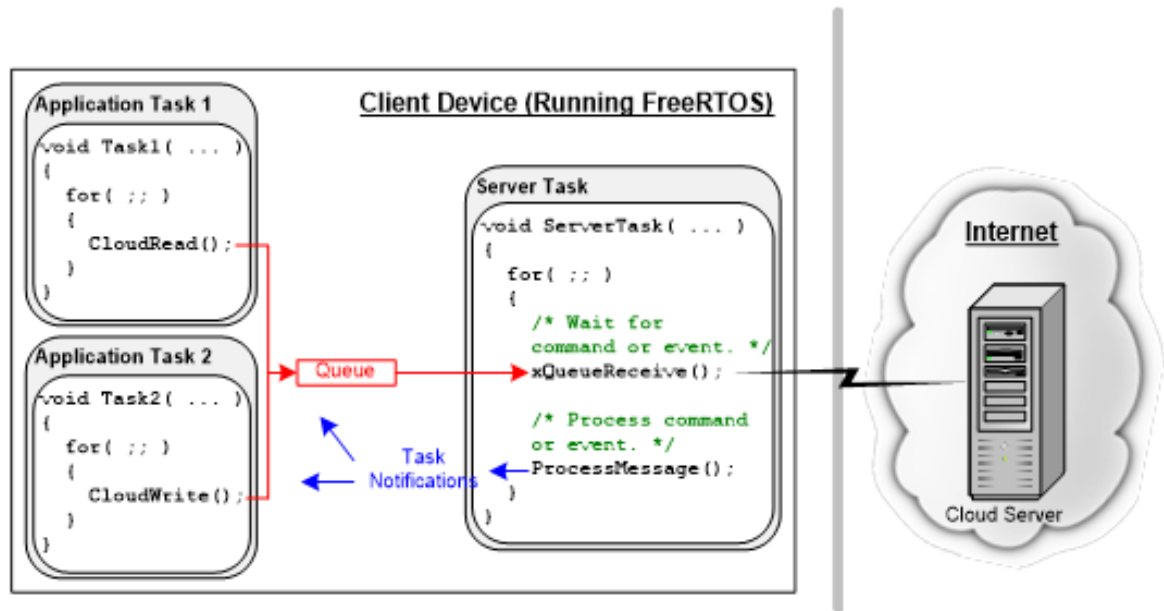
```
    }  
}  
/*-----*/  
/* The interrupt service routine that executes each time an ADC conversion completes. */  
void ADC_ConversionEndISR( xADC *pxADCInstance )  
{  
    uint32_t ulConversionResult;  
    BaseType_t xHigherPriorityTaskWoken = pdFALSE, xResult;  
    /* Read the new ADC value and clear the interrupt. */  
    ulConversionResult = ADC_low_level_read( pxADCInstance );  
    /* Send a notification, and the ADC conversion result, directly to vADCTask(). */  
    xResult = xTaskNotifyFromISR( xADCTaskToNotify, /* xTaskToNotify parameter. */  
    ulConversionResult, /* ulValue parameter. */ eSetValueWithoutOverwrite, /* eAction  
    parameter. */ &xHigherPriorityTaskWoken );  
    /* If the call to xTaskNotifyFromISR() returns pdFAIL then the task is not keeping  
    up with the rate at which ADC values are being generated. configASSERT() is described in  
    section 11.2.*/  
    configASSERT( xResult == pdPASS );  
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );  
}
```

## 애플리케이션 내부에서 직접 사용되는 작업 알림

이번 단원에서는 다음 기능이 포함된 가상의 애플리케이션에서 작업 알림을 사용하는 방법에 대해서 설명하겠습니다.

1. 애플리케이션이 느린 인터넷 연결을 통해 통신하면서 원격 데이터 서버(클라우드 서버)에 데이터를 전송하거나, 혹은 원격 데이터 서버에서 데이터를 요청합니다.
2. 요청 작업이 클라우드 서버에서 데이터를 요청한 후에는 요청한 데이터가 수신될 때까지 Blocked 상태로 대기해야 합니다.
3. 데이터가 클라우드 서버에 전송되면 전송 작업은 클라우드 서버가 데이터를 정확하게 수신했다는 사실이 확인될 때까지 Blocked 상태로 대기해야 합니다.

소프트웨어 설계는 다음과 같습니다.



- 클라우드 서버에 대한 다수의 인터넷 연결을 처리하는 데 따른 복잡성은 단일 FreeRTOS 작업으로 압축됩니다. 작업은 FreeRTOS 애플리케이션 내부에서 프록시 서버 역할을 하며, 이러한 작업을 서버 작업이라고 부릅니다.
- 애플리케이션 작업이 CloudRead()를 호출하여 클라우드 서버에서 데이터를 읽어옵니다. CloudRead()는 클라우드 서버와 직접 통신하지 않습니다. 대신에 읽기 요청을 대기열을 통해 서버 작업에게 전송한 후 요청한 데이터를 서버 작업에서 작업 알림 형태로 수신합니다.
- 애플리케이션 작업이 CloudWrite()를 호출하여 클라우드 서버에 데이터를 작성합니다. CloudWrite()는 클라우드 서버와 직접 통신하지 않습니다. 대신에 쓰기 요청을 대기열을 통해 서버 작업에게 전송한 후 쓰기 작업 결과를 서버 작업에서 작업 알림 형태로 수신합니다.

다음 코드는 CloudRead() 및 CloudWrite() 함수에서 서버 작업으로 전송하는 구조를 나타낸 것입니다.

```
typedef enum CloudOperations
{
    eRead, /* Send data to the cloud server. */
    eWrite /* Receive data from the cloud server. */
} Operation_t;

typedef struct CloudCommand
{
    Operation_t eOperation; /* The operation to perform (read or write). */
    uint32_t ulDataID; /* Identifies the data being read or written. */
    uint32_t ulDataValue; /* Only used when writing data to the cloud server. */
    TaskHandle_t xTaskToNotify; /* The handle of the task performing the operation. */
} CloudCommand_t;
```

CloudRead() 의사 코드는 다음과 같습니다. 함수가 요청을 서버 작업에게 전송한 다음 xTaskNotifyWait()를 호출하여 요청 데이터를 사용할 수 있다는 사실을 알게 될 때까지 Blocked 상태로 대기합니다.

```
/* ulDataID identifies the data to read. pulValue holds the address of the variable into
   which the data received from the cloud server is to be written. */

BaseType_t CloudRead( uint32_t ulDataID, uint32_t *pulValue )
{
    CloudCommand_t xRequest;

    BaseType_t xReturn;

    /* Set the CloudCommand_t structure members to be correct for this read request. */
    xRequest.eOperation = eRead; /* This is a request to read data. */

    xRequest.ulDataID = ulDataID; /* A code that identifies the data to read. */

    xRequest.xTaskToNotify = xTaskGetCurrentTaskHandle(); /* Handle of the calling task. */

    /* Ensure there are no notifications already pending by reading the notification value
       with a block time of 0, and then send the structure to the server task. */

    xTaskNotifyWait( 0, 0, NULL, 0 );

    xQueueSend( xServerTaskQueue, &xRequest, portMAX_DELAY );

    /* Wait for a notification from the server task. The server task writes the value
       received from the cloud server directly into this task's notification value, so there
       is no need to clear any bits in the notification value on entry to or exit from the
       xTaskNotifyWait() function. The received value is written to *pulValue, so pulValue is
       passed as the address to which the notification value is written. */

    xReturn = xTaskNotifyWait( 0, /* No bits cleared on entry. */ 0, /* No bits to clear
       on exit. */ pulValue, /* Notification value into *pulValue. */ pdMS_TO_TICKS( 250 ) );
    /* Wait a maximum of 250ms. */
    /* If xReturn is pdPASS, then the value was obtained. If xReturn is pdFAIL,
       then the request timed out. */

    return xReturn;
}
```

서버 작업의 읽기 요청 관리 방식을 나타내는 의사 코드는 다음과 같습니다. 데이터가 클라우드 서버에서 수신되면 서버 작업이 애플리케이션 작업의 차단을 해제하고 xTaskNotify()를 호출해 eAction 파라미터를 eSetValueWithOverwrite로 설정함으로써 수신된 데이터를 애플리케이션 작업에게 전송합니다.

이러한 방식은 GetCloudData()가 클라우드 서버에서 값을 가져올 때까지 대기할 필요가 없다는 가정을 전제로 하기 때문에 단순한 시나리오입니다.

```
void ServerTask( void *pvParameters )
{
    CloudCommand_t xCommand;

    uint32_t ulReceivedValue;

    for( ;; )
    {
```

```
/* Wait for the next CloudCommand_t structure to be received from a task. */
xQueueReceive( xServerTaskQueue, &xCommand, portMAX_DELAY );

switch( xCommand.eOperation ) /* Was it a read or write request? */
{
    case eRead:

        /* Obtain the requested data item from the remote cloud server. */

        ulReceivedValue = GetCloudData( xCommand.ulDataID );

        /* Call xTaskNotify() to send both a notification and the value received from
        the cloud server to the task that made the request. The handle of the task is obtained
        from the CloudCommand_t structure. */

        xTaskNotify( xCommand.xTaskToNotify, /* The task's handle is in the structure.
        */ ulReceivedValue, /* Cloud data sent as notification value. */ eSetValueWithOverwrite );

        break;

        /* Other switch cases go here. */

}

}
```

CloudWrite() 의사 코드는 다음과 같습니다. 쉽게 설명할 수 있도록 CloudWrite()는 비트 단위의 상태 코드를 반환하며, 상태 코드의 각 비트에는 고유의 의미가 할당됩니다. 상단의 #define 문에는 4가지 상태 비트 예제가 있습니다.

작업이 상태 비트 4개를 소거하고, 요청을 서버 작업에게 전송한 다음 xTaskNotifyWait()를 호출하여 상태 알림을 기다리며 Blocked 상태로 대기합니다.

```
/* Status bits used by the cloud write operation. */

#define SEND_SUCCESSFUL_BIT ( 0x01 << 0 )

#define OPERATION_TIMED_OUT_BIT ( 0x01 << 1 )

#define NO_INTERNET_CONNECTION_BIT ( 0x01 << 2 )

#define CANNOT_LOCATE_CLOUD_SERVER_BIT ( 0x01 << 3 )

/* A mask that has the four status bits set. */

#define CLOUD_WRITE_STATUS_BIT_MASK ( SEND_SUCCESSFUL_BIT \
| OPERATION_TIMED_OUT_BIT \
| NO_INTERNET_CONNECTION_BIT \
| CANNOT_LOCATE_CLOUD_SERVER_BIT )

uint32_t CloudWrite( uint32_t ulDataID, uint32_t ulDataValue )
{
    CloudCommand_t xRequest;
```

```
uint32_t ulNotificationValue;

/* Set the CloudCommand_t structure members to be correct for this write request. */
xRequest.eOperation = eWrite; /* This is a request to write data. */

xRequest.ulDataID = ulDataID; /* A code that identifies the data being written. */

xRequest.ulDataValue = ulDataValue; /* Value of the data written to the cloud server.
*/

xRequest.xTaskToNotify = xTaskGetCurrentTaskHandle(); /* Handle of the calling task. */

/* Clear the three status bits relevant to the write operation by
calling xTaskNotifyWait() with the ulBitsToClearOnExit parameter set to
CLOUD_WRITE_STATUS_BIT_MASK, and a block time of 0. The current notification value is not
required, so the pulNotificationValue parameter is set to NULL. */

xTaskNotifyWait( 0, CLOUD_WRITE_STATUS_BIT_MASK, NULL, 0 );

/* Send the request to the server task. */

xQueueSend( xServerTaskQueue, &xRequest, portMAX_DELAY );

/* Wait for a notification from the server task. The server task writes a bitwise
status code into this task's notification value, which is written to ulNotificationValue.
*/

xTaskNotifyWait( 0, /* No bits cleared on entry. */ CLOUD_WRITE_STATUS_BIT_MASK, /
* Clear relevant bits to 0 on exit. */ &ulNotificationValue, /* Notified value. */
pdMS_TO_TICKS( 250 ) ); /* Wait a maximum of 250ms. */ /* Return the status code to the
calling task. */ return ( ulNotificationValue & CLOUD_WRITE_STATUS_BIT_MASK );
```

서버 작업의 쓰기 요청 관리 방식을 나타내는 의사 코드는 다음과 같습니다. 데이터가 클라우드 서버에 전송되면 서버 작업이 애플리케이션 작업의 차단을 해제하고 xTaskNotify()를 호출해 eAction 파라미터를 eSetBits로 설정함으로써 비트 단위의 상태 코드를 애플리케이션 작업에게 전송합니다. CLOUD\_WRITE\_STATUS\_BIT\_MASK 상수에서 정의하는 비트만 수신 작업의 알림 값에서 변경 가능합니다. 따라서 수신 작업이 알림 값의 다른 비트를 다른 목적으로 사용할 수 있습니다.

이러한 방식은 SetCloudData()가 원격 클라우드 서버에서 확인 값을 가져올 때까지 대기할 필요가 없다는 가정을 전제로 하기 때문에 단순한 시나리오입니다.

```
void ServerTask( void *pvParameters )
{
    CloudCommand_t xCommand;

    uint32_t ulBitwiseStatusCode;

    for( ;; )
    {
        /* Wait for the next message. */

        xQueueReceive( xServerTaskQueue, &xCommand, portMAX_DELAY );

        /* Was it a read or write request? */

        switch( xCommand.eOperation )
        {
```

```
        case eWrite:

            /* Send the data to the remote cloud server. SetCloudData() returns a bitwise
            status code that only uses the bits defined by the CLOUD_WRITE_STATUS_BIT_MASK definition
            (shown in the preceding code). */

            ulBitwiseStatusCode = SetCloudData( xCommand.ulDataID, xCommand.ulDataValue );

            /* Send a notification to the task that made the write request. The eSetBits
            action is used so any status bits set in ulBitwiseStatusCode will be set in the
            notification value of the task being notified. All the other bits remain unchanged. The
            handle of the task is obtained from the CloudCommand_t structure. */

            xTaskNotify( xCommand.xTaskToNotify, /* The task's handle is in the structure.
            */ ulBitwiseStatusCode, /* Cloud data sent as notification value. */ eSetBits );

            break;

            /* Other switch cases go here. */

        }

    }

}
```

## 개발자 지원

이번 단원에서는 다음과 같은 방법으로 생산성을 극대화하는 기능에 대해서 살펴보겠습니다.

- 애플리케이션의 동작에 대한 인사이트 제공
- 최적화할 수 있는 기회 강조
- 발생 시점의 오류 트래핑

## configASSERT()

C에서 `assert()` 매크로를 사용해 프로그램에서 생성된 어설션(가정) 유무를 확인합니다. 어설션은 C 표현식으로 작성됩니다. 이 표현식이 `false(0)`로 평가되면 어설션에 실패한 것으로 간주합니다. 예를 들어 아래는 어설션을 테스트하여 `pxMyPointer` 포인터가 `NULL`이 아닌지 알아보는 코드입니다.

```
/* Test the assertion that pxMyPointer is not NULL */  
  
assert( pxMyPointer != NULL );
```

애플리케이션 개발자는 `assert()` 매크로를 실행하여 어설션에 실패할 경우 실행할 작업을 지정합니다.

FreeRTOS 소스 코드는 `assert()`를 호출하지 않습니다. FreeRTOS를 컴파일할 때 사용되는 컴파일러에서는 `assert()`가 제공되지 않기 때문입니다. 대신에 FreeRTOS 소스 코드에는 `configASSERT()`라고 하는 매크로 호출이 많습니다. 이 매크로는 애플리케이션 개발자가 `FreeRTOSConfig.h`에서 정의할 수 있으며, 동작이 표준 C `assert()`와 정확히 일치합니다.

실패한 어설션은 치명적인 오류로 처리되어야 합니다. 어설션을 실패한 라인 이후로는 실행하지 마십시오.

`configASSERT()`를 사용하면 가장 자주 발생하는 오류 원인을 즉시 트래핑하여 알아낼 수 있기 때문에 생산성이 향상됩니다. 따라서 FreeRTOS 애플리케이션을 개발하거나 디버깅할 때 `configASSERT()`를 정의하는 것이 매우 바람직합니다.

`configASSERT()`를 정의하면 런타임 디버깅에 효과적이지만 애플리케이션 코드 크기가 늘어나 실행 시간이 느려집니다. `configASSERT()` 정의를 입력하지 않으면 기본적으로 비어있는 정의를 사용하기 때문에 `configASSERT()` 호출이 모두 C 프리프로세서에서 완전히 제거됩니다.

## configASSERT() 정의 예제

다음 코드에 보이는 `configASSERT()` 정의는 애플리케이션을 디버거의 제어 하에서 실행할 때 유용합니다. 어설션에 실패하는 라인부터는 실행이 중지되기 때문에 디버그 세션이 일시 중지되면서 어설션을 실패한 라인이 디버거에 표시됩니다.

```
/* Disable interrupts so the tick interrupt stops executing, and then sit in a loop so  
execution does not move past the line that failed the assertion. If the hardware supports  
a debug break instruction, then the debug break instruction can be used in place of the  
for() loop. */  
  
#define configASSERT( x ) if( ( x ) == 0 ) { taskDISABLE_INTERRUPTS();
```



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

configASSERT() 정의는 애플리케이션을 디버거의 제어 하에서 실행하지 않을 때 유용합니다. 이때는 어설션에 실패한 소스 코드 라인을 출력하거나, 혹은 다른 방법으로 기록합니다. 표준 C \_\_FILE\_\_ 매크로를 사용해 소스 파일 이름을 알아내고, C \_\_LINE\_\_ 매크로를 사용해 소스 파일의 라인 수를 알아냄으로써 어설션에 실패한 라인을 구분할 수 있습니다.

이 코드는 어설션에 실패한 소스 코드 라인을 기록하는 configASSERT() 정의를 나타냅니다.

## Tracealyzer

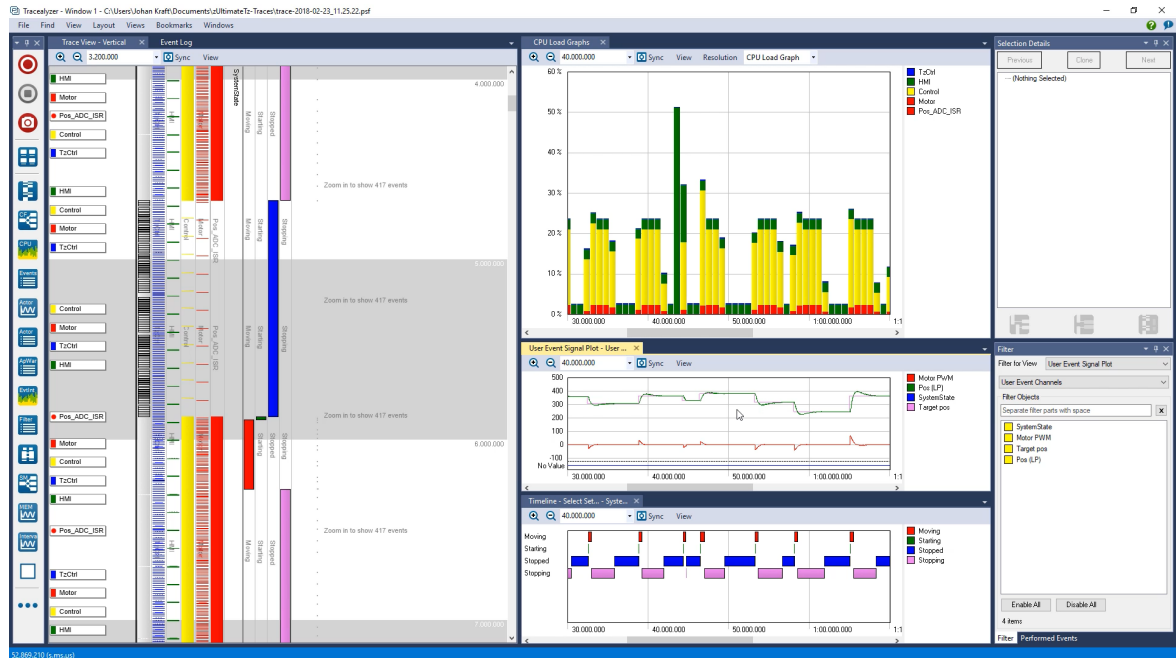
Tracealyzer는 파트너 기업인 Percepio에서 제공하는 런타임 진단 및 최적화 도구입니다.

Tracealyzer는 소중한 동적 동작 정보를 수집하여 서로 연동되는 그래픽 뷰로 표현합니다. 또한 다수의 뷰를 서로 동기화하여 표시하기도 합니다.

이렇게 수집된 정보는 FreeRTOS 애플리케이션을 분석하거나, 문제를 해결하거나, 단순히 최적화하는 데 매우 유용합니다.

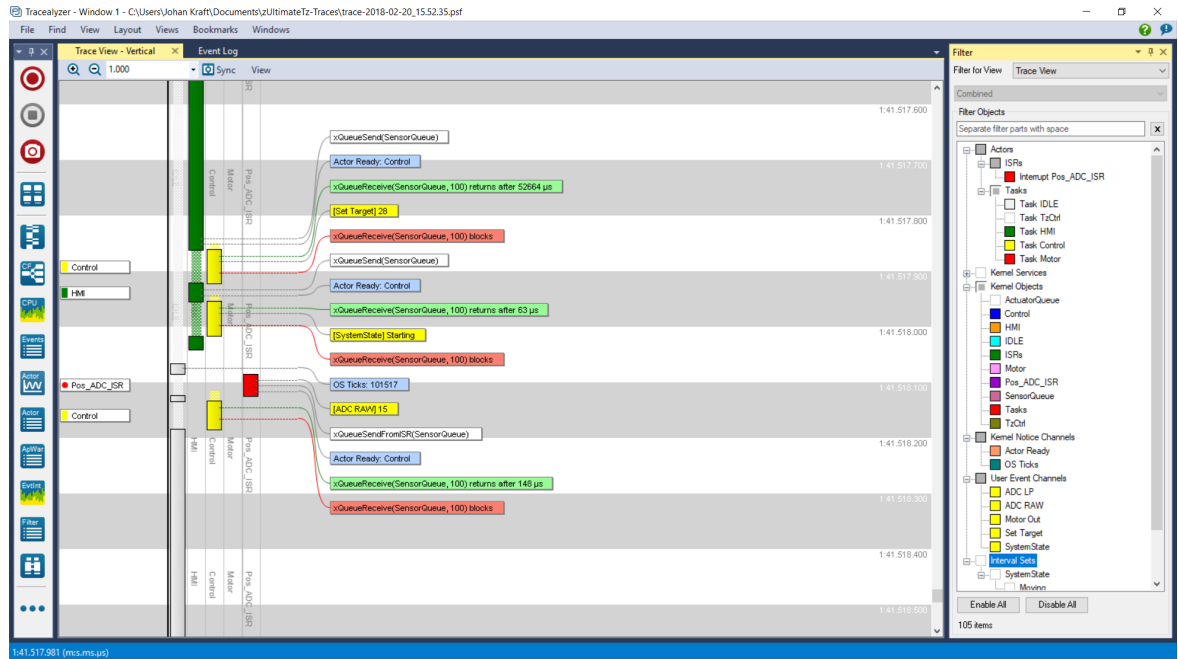
Tracealyzer는 기존 디버거와 함께 사용할 수도 있습니다. 이때는 시간을 기반으로 더욱 높은 수준의 전망을 추가하여 디버거의 뷰를 보완합니다.

Tracealyzer에는 다음과 같이 서로 연동되는 뷰가 20가지 이상 있습니다.

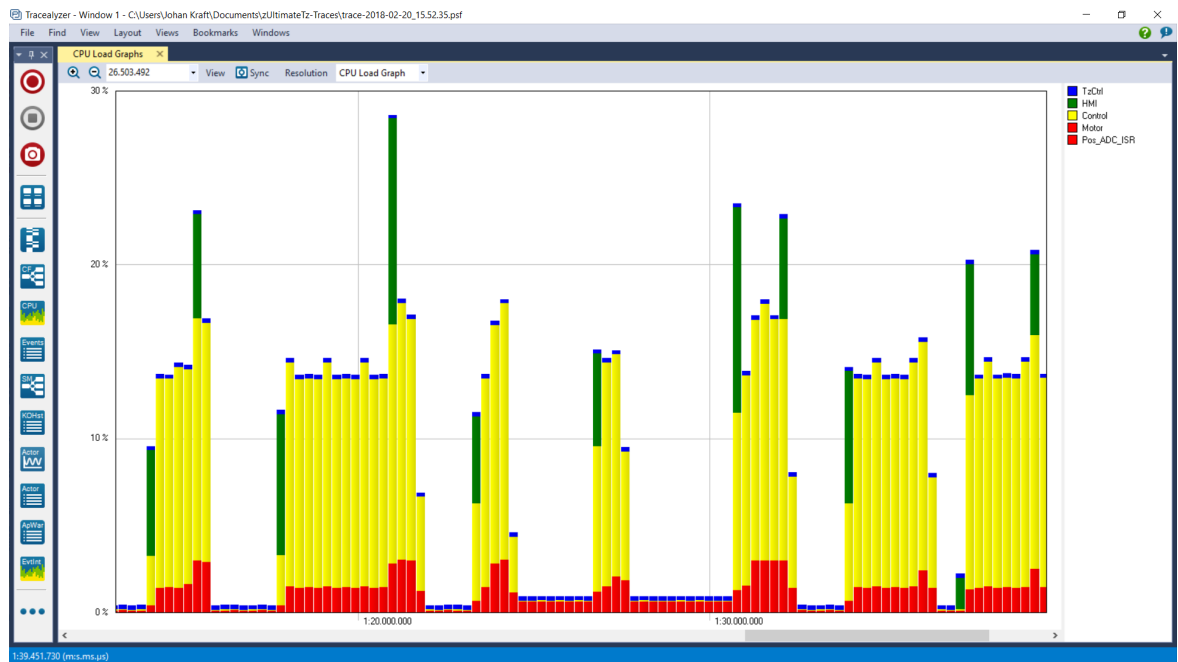


수직 추적 뷰:

## FreeRTOS 커널 개발자 안내서 Tracealyzer

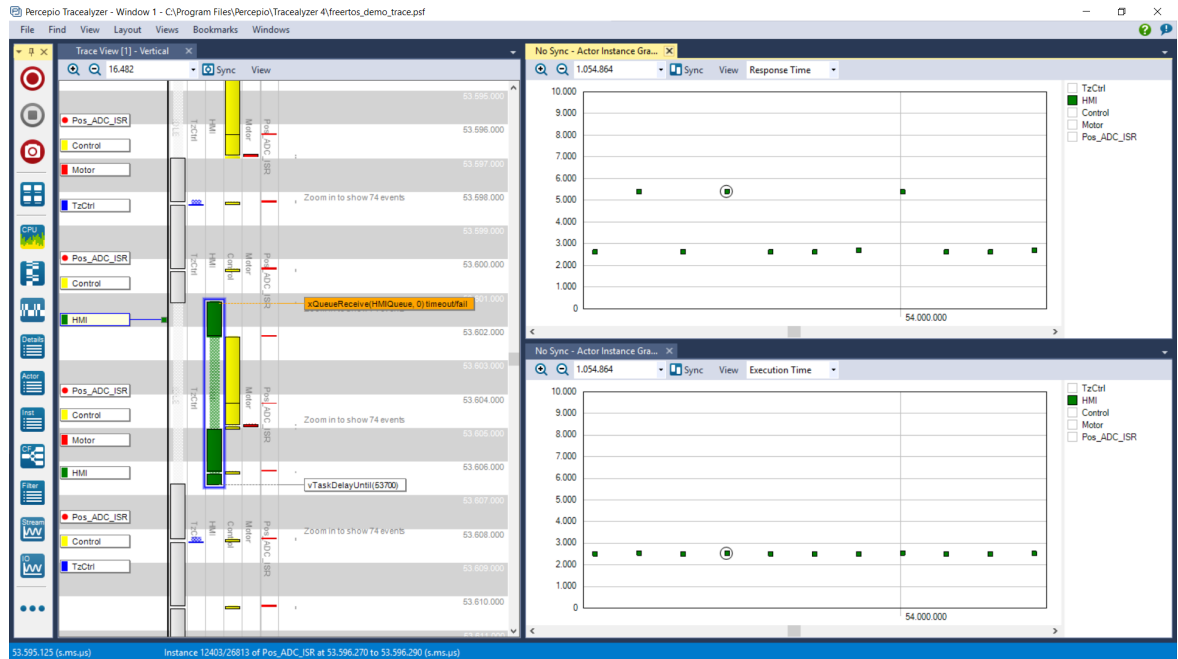


### CPU 부하 뷰:

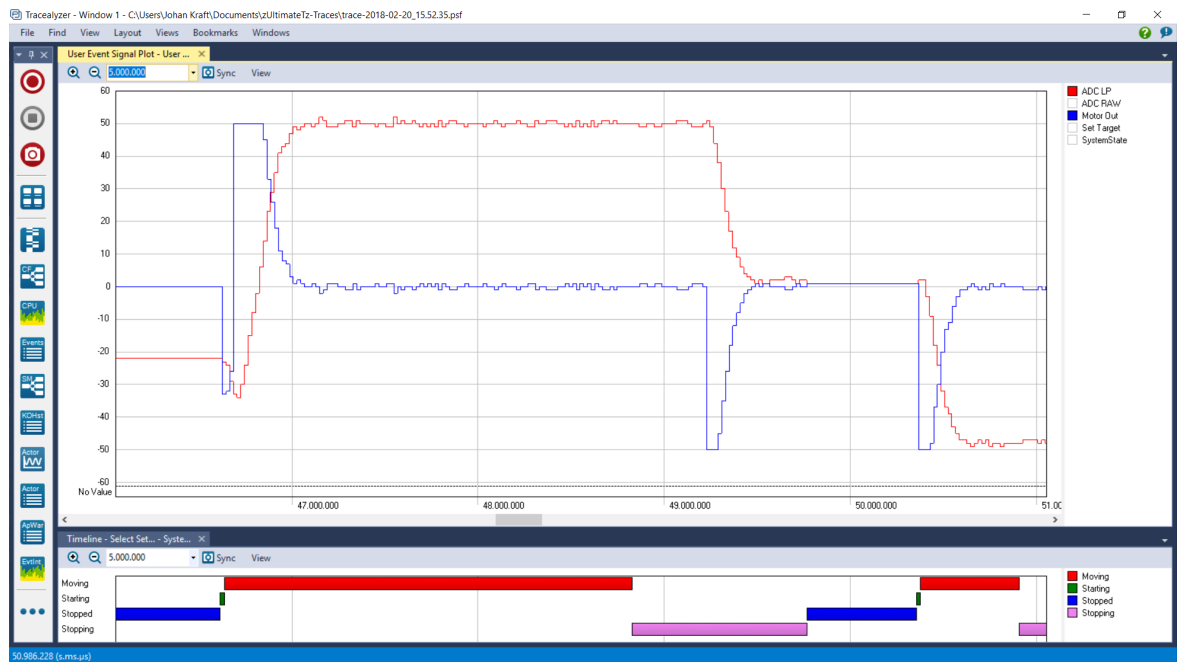


### 응답 시간 뷰:

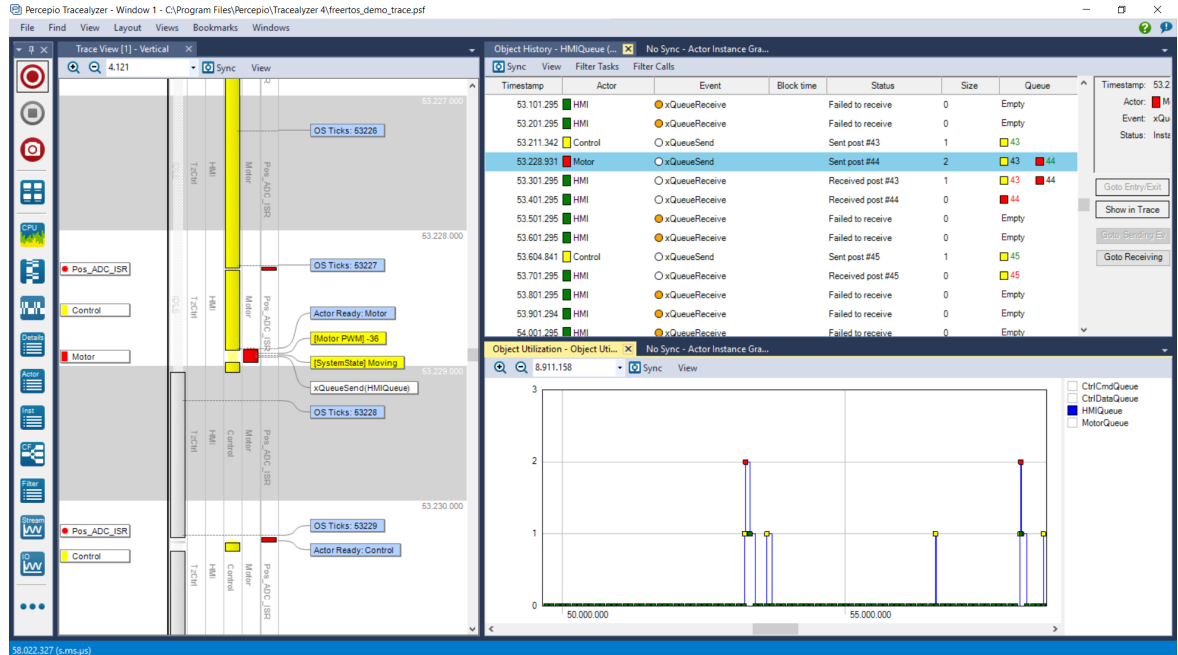
## FreeRTOS 커널 개발자 안내서 Tracealyzer



사용자 이벤트 플롯 뷰:



객체 기록 뷰:



## 디버그 관련 후크(콜백) 함수

malloc 실패 후크를 정의하면 작업, 대기열, 세마포어 또는 이벤트 그룹을 생성하려는 시도가 실패했을 때 애플리케이션 개발자에게 바로 알려줄 수 있습니다. malloc 실패 후크(또는 콜백)에 대한 자세한 내용은 [힙 메모리 관리 \(p. 13\)](#) 단원을 참조하십시오.

스택 오버플로우 후크를 정의하면 작업에 사용되는 스택 크기가 작업에 할당된 스택 공간을 초과했을 때 애플리케이션 개발자에게 알려줄 수 있습니다. 스택 오버플로우 후크에 대한 자세한 내용은 문제 해결에서 [스택 오버플로우 \(p. 234\)](#) 단원을 참조하십시오.

## 런타임 및 작업 상태 정보 보기

### 작업 런타임 통계

작업 런타임 통계는 각 작업마다 수신되는 처리 시간에 대한 정보를 제공합니다. 작업 런타임은 애플리케이션 부팅 이후 작업이 Running 상태를 유지한 총 시간을 나타냅니다.

런타임 통계는 프로젝트 개발 단계에서 프로파일링과 디버깅을 지원할 목적으로 사용됩니다. 여기에서 제공되는 정보는 런타임 통계 클록으로 사용되는 카운터가 오버플로우될 때까지만 유효합니다. 런타임 통계를 수집하면 작업 컨텍스트를 전환하는 시간도 증가합니다.

이진 런타임 통계 정보를 구하려면 `uxTaskGetSystemState()` API 함수를 호출하십시오. 런타임 통계 정보를 사람이 읽을 수 있는 ASCII 테이블로 구하려면 `vTaskGetRunTimeStats()` 헬퍼 함수를 호출하십시오.

### 런타임 통계 클록

런타임 통계는 아주 작은 틱 주기까지 측정해야 합니다. 이러한 이유로 RTOS 틱 카운트는 런타임 통계 클록으로 사용되지 않습니다. 대신에 애플리케이션 코드에서 클록이 제공됩니다. 런타임 통계 클록의 주파수는 틱 인터럽트의 주파수보다 10~100배 빠르게 설정하는 것이 좋습니다. 런타임 통계 클록이 빠를수록 통계의 정확성도 높아지기 때문입니다. 단, 이 경우 시간 값이 더욱 빠르게 오버플로우될 수 있습니다.

따라서 시간 값은 자유롭게 증가하는(free running) 32비트 주변 타이머/카운터에서 생성되며, 이렇게 생성된 값은 다른 처리 오버헤드 없이 읽을 수 있습니다. 사용할 수 있는 주변 장치와 클럭 속도가 이러한 방법을 지원하지 못할 경우에는 비교적 효율성이 떨어지지만 다음과 같은 방법으로 대체할 수 있습니다.

1. 주기적 인터럽트가 원하는 런타임 통계 클럭 주파수에서 생성되도록 주변 장치를 구성한 후 생성되는 인터럽트 수를 카운트하여 런타임 통계 클럭으로 사용합니다.  
  
이 방법은 주기적 인터럽트를 런타임 통계 클럭을 제공할 목적으로만 사용할 경우 매우 비효율적입니다. 하지만 애플리케이션이 적합한 주파수로 주기적인 인터럽트를 이미 사용하고 있다면 생성된 인터럽트 수를 카운트하여 기존 인터럽트 서비스 루틴에 추가하는 것이 간단하고 효율적입니다.
2. 자유롭게 증가하는 16비트 주변 타이머의 현재 값을 32비트 값의 최하위 16비트로, 그리고 타이머가 오버플로우된 횟수를 32비트 값의 최상위 16비트로 사용하여 32비트 값을 생성합니다.

RTOS 틱 카운트와 ARM Cortex-M SysTick 타이머의 현재 값을 결합하여 런타임 통계 클럭을 생성할 수도 있지만 조작이 다소 복잡할 수 있습니다. FreeRTOS 다운로드에 있는 일부 데모 프로젝트는 이러한 방법을 나타냅니다.

## 런타임 통계를 수집하도록 애플리케이션 구성

다음 표는 작업 런타임 통계를 수집하는 데 필요한 매크로를 나열한 것입니다. 매크로에 접두사로 'port'가 사용된 이유는 처음에 RTOS 포트 계층에 추가하려고 했기 때문입니다. 하지만 FreeRTOSConfig.h에서 정의하는 것이 더욱 실용적입니다.

매크로	설명
configGENERATE_RUN_TIME_STATS	이 매크로는 FreeRTOSConfig.h에서 1로 설정되어야 합니다. 이 매크로가 1로 설정되면 스케줄러가 이 표에 있는 나머지 매크로를 필요한 때에 호출합니다.
portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()	이 매크로는 런타임 통계 클럭으로 사용되는 주변 장치를 초기화하도록 입력되어야 합니다.
portGET_RUN_TIME_COUNTER_VALUE() 또는	두 매크로 중 하나는 현재 런타임 통계 클럭 값을 반환하도록 입력되어야 합니다. 이 값은 애플리케이션의 첫 부팅 이후 애플리케이션이 실행된 총 시간(런타임 통계 클럭 단위)을 나타냅니다.
portALT_GET_RUN_TIME_COUNTER_VALUE(Time)	첫 번째 매크로를 사용하는 경우 현재 클럭 값으로 평가되도록 정의되어야 합니다. 두 번째 매크로를 사용하는 경우 'Time' 파라미터를 현재 클럭 값으로 설정하도록 정의되어야 합니다.

## uxTaskGetSystemState() API 함수

uxTaskGetSystemState()는 FreeRTOS 스케줄러에서 제어하는 각 작업의 상태 정보를 스냅샷으로 제공합니다. 정보는 TaskStatus\_t 구조의 배열 형태로 제공되며, 각 작업 배열마다 인덱스가 하나씩 포함됩니다. uxTaskGetSystemState() API 함수 프로토타입은 다음과 같습니다.

다음 표는 uxTaskGetSystemState() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름	설명
pxTaskStatusArray	TaskStatus_t 구조 배열을 가리키는 포인터입니다.

	<p>배열은 각 작업마다 TaskStatus_t 구조를 1개 이상 가지고 있어야 합니다. 작업 수는 uxTaskGetNumberOfTasks() API 함수를 사용해 확인할 수 있습니다.</p> <p>TaskStatus_t 구조는 아래 코드로 표시됩니다. TaskStatus_t 구조 멤버는 다음 표에 설명되어 있습니다.</p>
uxArraySize	<p>pxTaskStatusArray 파라미터에서 가리키는 배열의 크기입니다. 크기는 배열의 바이트 수가 아닌 인덱스 수(배열에 포함되는 TaskStatus_t 구조 수)로 지정됩니다.</p>
pulTotalRunTime	<p>configGENERATE_RUN_TIME_STATS가 FreeRTOSConfig.h에서 1로 설정되면 <b>&lt;problematic&gt;*&lt;/problematic&gt;</b> uxTaskGetSystemState()에서 pulTotalRunTime이 대상 부팅 이후 총 런타임(애플리케이션에서 제공하는 런타임 통계 클록으로 정의)으로 설정됩니다.</p> <p>pulTotalRunTime은 선택 사항이기 때문에 총 런타임이 필요 없으면 NULL로 설정될 수도 있습니다.</p>
반환 값	<p>uxTaskGetSystemState()에서 채워진 TaskStatus_t 구조 수가 반환됩니다.</p> <p>반환 값은 uxTaskGetNumberOfTasks() API 함수에서 반환되는 수와 동일해야 하지만 uxArraySize 파라미터로 전달되는 값이 너무 작으면 0이 됩니다.</p>

아래는 TaskStatus\_t 구조입니다.

<pre>typedef struct xTASK_STATUS {     TaskHandle_t xHandle;      const char *pcTaskName;      UBaseType_t xTaskNumber;      eTaskState eCurrentState;      UBaseType_t uxCurrentPriority;      UBaseType_t uxBasePriority;      uint32_t ulRunTimeCounter;      uint16_t usStackHighWaterMark; } TaskStatus_t;</pre>
---

다음 표는 TaskStatus\_t 구조 멤버를 나열한 것입니다.

파라미터 이름/반환 값	설명
--------------	----

xHandle	구조의 정보와 관련 있는 작업 핸들입니다.
pcTaskName	사람이 읽을 수 있는 작업의 텍스트 이름입니다.
xTaskNumber	<p>각 작업은 고유한 xTaskNumber 값이 하나씩 있습니다.</p> <p>애플리케이션이 런타임에서 작업을 생성 및 삭제하면 작업이 이전에 삭제되었던 작업과 동일한 핸들을 가질 수 있습니다. xTaskNumber를 입력하는 이유는 애플리케이션 코드와 커널 인식 디버거가 여전히 유효한 작업과 삭제되었지만 유효한 작업과 동일한 핸들을 가졌던 작업을 구분하기 위해서입니다.</p>
eCurrentState	<p>작업 상태를 나타내는 열거형 파라미터입니다. eCurrentState의 값은 eRunning, eReady, eBlocked, eSuspended 또는 eDeleted가 될 수 있습니다.</p> <p>작업은 vTaskDelete() 호출로 작업이 삭제된 시간부터 유휴 작업이 삭제된 작업의 내부 데이터 구조 및 스택에 할당되었던 메모리를 해제하는 시간까지 짧은 기간에 eDeleted 상태일 때만 보고됩니다. 이 시간이 지나면 작업이 더 이상 존재하지 않기 때문에 핸들을 사용하려고 해도 아무런 효과가 없습니다.</p>
uxCurrentPriority	uxTaskGetSystemState()를 호출했을 때 작업의 실행 우선순위입니다. 작업이 <a href="#">리소스 관리 (p. 152)</a> 단원에서 설명하는 우선순위 상속 메커니즘에 따라 일시적으로 더욱 높은 우선순위로 할당되었을 경우에만 uxCurrentPriority는 애플리케이션 개발자가 작업에 할당한 우선순위보다 높습니다.
uxBasePriority	애플리케이션 개발자가 작업에 할당하는 우선순위입니다. uxBasePriority는 configUSE_MUTEXES가 FreeRTOSConfig.h에서 1로 설정되었을 때만 유효합니다.
ulRunTimeCounter	작업 생성 이후 작업에서 사용한 총 런타임입니다. 총 런타임은 애플리케이션 개발자가 런타임 통계를 수집할 목적으로 제공하는 클록을 사용해 절대 시간으로 입력됩니다. ulRunTimeCounter는 configGENERATE_RUN_TIME_STATS가 FreeRTOSConfig.h에서 1로 설정된 경우에만 유효합니다.
usStackHighWaterMark	작업의 스택 하이 워터 마크입니다. 하이 워터 마크란 작업 생성 이후 작업에 남아있는 스택 공간의 최소 크기를 말합니다. 이를 통해 작업이 스택 오버플로우에 얼마나 가까워졌는지 알 수 있습니다. 이 값이 0에 가까울수록 작업이 스택 오버플로우에 더욱 가까워집니다. usStackHighWaterMark는 바이트 단위로 지정됩니다.

## vTaskList() 헬퍼 함수

vTaskList()는 uxTaskGetSystemState()와 비슷하게 작업 상태 정보를 제공하지만 이진 값의 배열이 아니고 사람이 읽을 수 있는 ASCII 테이블로 정보를 나타냅니다.

vTaskList()는 프로세서 집약적인 함수이기 때문에 스케줄러가 장시간 일시 중지됩니다. 따라서 헬퍼 함수는 디버깅 목적으로만 사용하고 프로덕션 실시간 시스템에서는 사용하지 않는 것이 좋습니다.

vTaskList()는 configUSE\_TRACE\_FACILITY와 configUSE\_STATS\_FORMATTING\_FUNCTIONS가 둘 다 FreeRTOSConfig.h에서 1로 설정된 경우에 사용할 수 있습니다.

```
void vTaskList( signed char *pcWriteBuffer );
```

다음 표는 vTaskList() 파라미터를 나열한 것입니다.

파라미터 이름	설명
pcWriteBuffer	사람이 읽을 수 있는 형식의 테이블이 작성되는 문자 버퍼를 가리키는 포인터입니다. 버퍼 크기는 전체 테이블을 저장할 만큼 커야 합니다. 경계 검사는 실행하지 않습니다.

vTaskList()에서 생성되는 출력 화면은 아래와 같습니다.

```
tcpip           R           3           393           0
Tmr Svc         R           3           111          48
QConsB1         R           1           143           3
QProdB5         R           0           144           7
QConsB6         R           0           143           8
PolSEM1         R           0           145           11
PolSEM2         R           0           145           12
GenQ            R           0           155           17
MuLow           R           0           147           18
Rec3            R           0           141           30
SUSP_RX         R           0           148           36
Math1           R           0           167           38
Math2           R           0           167           39
```

출력에서:

- 각 행은 단일 작업에 대한 정보를 제공합니다.
- 첫 번째 열은 작업 이름입니다.
- 두 번째 열은 작업 상태로서 'R'은 대기(Ready)를, 'B'는 차단됨(Blocked)을, 'S'는 일시 중지됨(Suspended)을, 그리고 'D'는 작업이 삭제된 것을 의미합니다. 작업은 vTaskDelete() 호출로 작업이 삭제된 시간부터 유휴 작업이 삭제된 작업의 내부 데이터 구조 및 스택에 할당되었던 메모리를 해제하는 시간까지 짧은 기간에 삭제된 상태일 때만 보고됩니다. 이 시간이 지나면 작업이 더 이상 존재하지 않기 때문에 핸들을 사용하려고 해도 아무런 효과가 없습니다.
- 세 번째 열은 작업 우선순위입니다.
- 네 번째 열은 작업의 스택 하이 워터 마크입니다. TaskStatus\_t 구조 멤버가 나열된 표에서 usStackHighWaterMark에 대한 설명을 참조하십시오.



- 다섯 번째 열은 작업에 할당된 고유 번호입니다. TaskStatus\_t 구조 멤버가 나열된 표에서 xTaskNumber에 대한 설명을 참조하십시오.

## vTaskGetRunTimeStats() 헬퍼 함수

vTaskGetRunTimeStats()는 수집된 런타임 통계를 사람이 읽을 수 있는 ASCII 테이블 형식으로 지정합니다.

vTaskGetRunTimeStats()는 프로세서 집약적인 함수이기 때문에 스케줄러가 장시간 일시 중지됩니다. 이러한 이유로 헬퍼 함수는 디버그 목적으로만 사용하고 프로덕션 실시간 시스템에서는 사용하지 않는 것이 좋습니다.

vTaskGetRunTimeStats()는 configGENERATE\_RUN\_TIME\_STATS와 configUSE\_STATS\_FORMATTING\_FUNCTIONS가 둘 다 FreeRTOSConfig.h에서 1로 설정된 경우에 사용할 수 있습니다.

아래 있는 것은 vTaskGetRunTimeStats() API 함수 프로토타입입니다.

```
void vTaskGetRunTimeStats( signed char *pcWriteBuffer );
```

다음 표는 vTaskGetRunTimeStats() 파라미터를 나열한 것입니다.

파라미터 이름	설명
pcWriteBuffer	사람이 읽을 수 있는 형식의 테이블이 작성되는 문자 버퍼를 가리키는 포인터입니다. 버퍼 크기는 전체 테이블을 저장할 만큼 커야 합니다. 경계 검사는 실행하지 않습니다.

vTaskGetRunTimeStats()에서 생성되는 출력 화면은 아래와 같습니다.

```
PolSEM1          994          <1%
PolSEM2        23248          1%
GenQ            194479        16%
MuLow           3690          <1%
Rec3            229450        18%
CNT1            242720        19%
PeekL            94          <1%
CNT_INC          165          <1%
CNT2            243166        20%
SUSP_RX          243192        20%
IDLE              55          <1%
```

출력에서:

- 각 행은 단일 작업에 대한 정보를 제공합니다.
- 첫 번째 열은 작업 이름입니다.
- 두 번째 열은 작업이 Running 상태를 유지한 시간으로 절대 값으로 표현됩니다. TaskStatus\_t 구조 멤버가 나열된 표에서 ulRunTimeCounter에 대한 설명을 참조하십시오.
- 세 번째 열은 작업이 Running 상태를 유지한 시간으로 대상 부팅 이후 총 경과 시간의 비율로 표현됩니다. 표시되는 비율의 총합은 일반적으로 100%보다 작습니다. 이는 통계가 수집 후 가장 가까운 정수 값으로 내려서 계산되기 때문입니다.

## 런타임 통계 생성 및 표시, 유효 예제

이번 예제에서는 가상의 16비트 타이머를 사용하여 32비트 런타임 통계 클록을 생성합니다. 카운터는 16비트 값이 최대 값에 도달할 때마다 인터럽트를 생성하도록 구성되어 오버플로우 인터럽트를 효과적으로 생성합니다. 인터럽트 서비스 루틴은 오버플로우가 발생하는 횟수를 계산합니다.

32비트 값은 오버플로우 발생 수를 32비트 값의 최상위 바이트 2개로, 그리고 현재 16비트 카운터 값을 32비트 값의 최하위 바이트 2개로 사용해 생성됩니다. 다음은 인터럽트 서비스 루틴에 사용되는 의사 코드입니다.

```
void TimerOverflowInterruptHandler( void )
{
    /* Just count the number of interrupts. */

    ulOverflowCount++;

    /* Clear the interrupt. */

    ClearTimerInterrupt();
}
```

다음은 런타임 통계 수집을 활성화하는 코드입니다.

```
/* Set configGENERATE_RUN_TIME_STATS to 1 to enable collection of runtime
statistics. When this is done, both portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() and
portGET_RUN_TIME_COUNTER_VALUE() or portALT_GET_RUN_TIME_COUNTER_VALUE(x) must also be
defined. */

#define configGENERATE_RUN_TIME_STATS 1

/* portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() is defined to call the function that sets up
the hypothetical 16-bit timer. (The function's implementation is not shown.) */

void vSetupTimerForRunTimeStats( void );

#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()

vSetupTimerForRunTimeStats()

/* portALT_GET_RUN_TIME_COUNTER_VALUE() is defined to set its parameter to the current
runtime counter/time value. The returned time value is 32-bits long, and is formed by
shifting the count of 16-bit timer overflows into the top two bytes of a 32-bit number,
and then bitwise ORing the result with the current 16-bit counter value. */

#define portALT_GET_RUN_TIME_COUNTER_VALUE( ulCountValue ) \
{
    extern volatile unsigned long ulOverflowCount;

    /* Disconnect the clock from the counter so it does not change while its value is being
used. */

    PauseTimer();

    /* The number of overflows is shifted into the most significant two bytes of the
returned 32-bit value. */
```

```
    ulCountValue = ( ulOverflowCount << 16UL );

    /* The current counter value is used as the least significant two bytes of the returned
    32-bit value. */

    ulCountValue |= ( unsigned long ) ReadTimerCount();

    /* Reconnect the clock to the counter. */

    ResumeTimer(); \\
}
```

다음은 수집된 런타임 통계를 5초마다 출력하는 작업입니다.

```
/* For clarity, calls to fflush() have been omitted from this codelist. */
static void prvStatsTask( void *pvParameters )
{
    TickType_t xLastExecutionTime;

    /* The buffer used to hold the formatted runtime statistics text needs to be quite
    large. It is therefore declared static to ensure it is not allocated on the task stack.
    This makes this function non-reentrant. */

    static signed char cStringBuffer[ 512 ];

    /* The task will run every 5 seconds. */

    const TickType_t xBlockPeriod = pdMS_TO_TICKS( 5000 );

    /* Initialize xLastExecutionTime to the current time. This is the only time this
    variable needs to be written to explicitly. Afterward, it is updated internally within the
    vTaskDelayUntil() API function. */

    xLastExecutionTime = xTaskGetTickCount();

    /* As per most tasks, this task is implemented in an infinite loop. */

    for( ;; )
    {
        /* Wait until it is time to run this task again. */

        vTaskDelayUntil( &xLastExecutionTime, xBlockPeriod );

        /* Generate a text table from the runtime stats. This must fit into the
        cStringBuffer array. */

        vTaskGetRunTimeStats( cStringBuffer );

        /* Print out column headings for the runtime stats table. */

        printf( "\nTask\t\tAbs\t\t\t%\n" );

        printf("-----\n" );

        /* Print out the runtime stats themselves. The table of data contains multiple
        lines, so the vPrintMultipleLines() function is called instead of calling printf()

```

```
directly. vPrintMultipleLines() simply calls printf() on each line individually, to ensure
the line buffering works as expected. */

    vPrintMultipleLines( cStringBuffer );

}

}
```

## 추적 후크 매크로

추적 매크로는 FreeRTOS 소스 코드에서 중요한 지점마다 위치합니다. 기본적으로 매크로가 비어있기 때문에 코드를 생성하거나 런타임 오버헤드가 발생하지도 않습니다. 애플리케이션 개발자는 기본적으로 비어있는 매크로를 재정의하여 다음과 같이 사용할 수 있습니다.

- FreeRTOS 소스 파일을 수정하지 않고 코드를 FreeRTOS에 삽입할 수 있습니다.
- 대상 하드웨어에서 사용 가능한 방법으로 실행 시퀀스 정보를 자세하게 출력할 수 있습니다. 추적 매크로는 FreeRTOS 소스 코드에서 충분히 많은 곳에 표시되기 때문에 스케줄러 활동 추적 및 프로파일링 로그를 모두 자세하게 생성하는 데 사용할 수 있습니다.

## 사용 가능한 추적 후크 매크로

다음 표는 애플리케이션 개발자에게 가장 유용한 매크로를 나열한 것입니다.

이 표에서는 대부분 설명에 `pxCurrentTCB`라고 하는 변수가 언급되어 있습니다. `pxCurrentTCB`는 Running 상태인 작업 핸들을 저장하는 FreeRTOS 프라이빗 변수입니다. FreeRTOS/Source/tasks.c 소스 파일에서 호출된 매크로에서는 이 변수를 사용할 수 있습니다.

매크로	설명
<code>traceTASK_INCREMENT_TICK(xTickCount)</code>	틱 카운트가 증가한 후 틱 인터럽트에서 호출됩니다. <code>xTickCount</code> 파라미터는 새로운 틱 카운트 값을 매크로에게 전달합니다.
<code>traceTASK_SWITCHED_OUT()</code>	실행할 작업을 새롭게 선택하기 전에 호출됩니다. 이 지점에서는 Running 상태가 종료되는 작업 핸들이 <code>pxCurrentTCB</code> 에 포함됩니다.
<code>traceTASK_SWITCHED_IN()</code>	실행할 작업을 선택한 후에 호출됩니다. 이 지점에서는 Running 상태가 시작되는 작업 핸들이 <code>pxCurrentTCB</code> 에 포함됩니다.
<code>traceBLOCKING_ON_QUEUE_RECEIVE(pxQueue)</code>	현재 실행 중인 작업이 비어있는 대기열에서 읽어오려는 시도, 혹은 비어있는 세마포어 또는 뮤텍스를 '가져오려는(take)' 시도 이후 Blocked 상태로 전환되기 바로 직전에 호출됩니다. <code>pxQueue</code> 파라미터는 대상 대기열 또는 세마포어 핸들을 매크로에 전달합니다.
<code>traceBLOCKING_ON_QUEUE_SEND(pxQueue)</code>	현재 실행 중인 작업이 가득 찬 대기열에 작성하려는 시도 이후 Blocked 상태로 전환되기 직전에 호출됩니다. <code>pxQueue</code> 파라미터는 대상 대기열 핸들을 매크로에 전달합니다.

traceQUEUE_SEND(pxQueue)	대기열 전송 또는 세마포어 '반환(give)'에 성공했을 때 xQueueSend(), xQueueSendToFront(), xQueueSendToBack() 또는 모든 세마포어 '반환' 함수 내에서 호출됩니다. pxQueue 파라미터는 대상 대기열 또는 세마포어 핸들을 매크로에 전달합니다.
traceQUEUE_SEND_FAILED(pxQueue)	대기열 전송 또는 세마포어 '반환(give)' 작업에 실패했을 때 xQueueSend(), xQueueSendToFront(), xQueueSendToBack() 또는 모든 세마포어 '반환' 함수 내에서 호출됩니다. 대기열 전송 또는 세마포어 '반환'은 대기열이 가득 차있거나, 혹은 지정된 차단 시간 동안 가득 찬 상태를 지속하는 경우 실패하게 됩니다. pxQueue 파라미터는 대상 대기열 또는 세마포어 핸들을 매크로에 전달합니다.
traceQUEUE_RECEIVE(pxQueue)	대기열 수신 또는 세마포어 '가져오기(take)'에 성공했을 때 xQueueReceive() 또는 모든 세마포어 '가져오기' 함수 내에서 호출됩니다. pxQueue 파라미터는 대상 대기열 또는 세마포어 핸들을 매크로에 전달합니다.
traceQUEUE_RECEIVE_FAILED(pxQueue)	대기열 수신 또는 세마포어 가져오기 작업에 실패했을 때 xQueueReceive() 또는 모든 세마포어 '가져오기' 함수 내에서 호출됩니다. 대기열 수신 또는 세마포어 '가져오기' 작업은 대기열 또는 세마포어가 비어있거나, 혹은 지정된 차단 시간 동안 비어있는 상태를 지속하는 경우 실패하게 됩니다. pxQueue 파라미터는 대상 대기열 또는 세마포어 핸들을 매크로에 전달합니다.
traceQUEUE_SEND_FROM_ISR(pxQueue)	전송 작업에 성공했을 때 xQueueSendFromISR() 내에서 호출됩니다. pxQueue 파라미터는 대상 대기열 핸들을 매크로에 전달합니다.
traceQUEUE_SEND_FROM_ISR_FAILED(pxQueue)	전송 작업에 실패했을 때 xQueueSendFromISR() 내에서 호출됩니다. 전송 작업은 대기열이 이미 가득 차있을 경우 실패하게 됩니다. pxQueue 파라미터는 대상 대기열 핸들을 매크로에 전달합니다.
traceQUEUE_RECEIVE_FROM_ISR(pxQueue)	수신 작업에 성공했을 때 xQueueReceiveFromISR() 내에서 호출됩니다. pxQueue 파라미터는 대상 대기열 핸들을 매크로에 전달합니다.
traceQUEUE_RECEIVE_FROM_ISR_FAILED(pxQueue)	대기열이 비어있어서 수신 작업에 실패했을 때 xQueueReceiveFromISR() 내에서 호출됩니다. pxQueue 파라미터는 대상 대기열 핸들을 매크로에 전달합니다.
traceTASK_DELAY_UNTIL()	호출 작업이 Blocked 상태로 전환되기 직전에 vTaskDelayUntil() 내에서 호출됩니다.
traceTASK_DELAY()	호출 작업이 Blocked 상태로 전환되기 직전에 vTaskDelay() 내에서 호출됩니다.

## 추적 후크 매크로의 정의

추적 매크로는 각각 비어있는 기본 정의가 있습니다. 이러한 기본 정의는 FreeRTOSConfig.h에 새로운 매크로 정의를 입력하여 재정의할 수 있습니다. 추적 매크로 정의가 길거나 복잡하면 새로운 헤더 파일에 구현한 후 파일 자체를 FreeRTOSConfig.h에 추가하는 방법도 있습니다.

FreeRTOS는 소프트웨어 엔지니어링 모범 사례에 따라 데이터 숨김 정책을 엄격하게 따르고 있습니다. 따라서 추적 매크로에서는 추적 매크로의 데이터 형식과 애플리케이션 코드의 데이터 형식이 서로 다르게 보이도록 사용자 코드를 FreeRTOS 소스 파일에 추가할 수 있습니다.

- FreeRTOS/Source/tasks.c 소스 파일 내부에서는 작업 핸들이 작업에 대한 데이터 구조를 가리키는 포인터입니다. 이것이 작업의 TCB(Task Control Block)입니다. 하지만 FreeRTOS/Source/tasks.c 소스 파일 외부에서는 작업 핸들이 보이드를 가리키는 포인터입니다.
- FreeRTOS/Source/queue.c 소스 파일 내부에서는 대기열 핸들이 대기열에 대한 데이터 구조를 가리키는 포인터입니다. 하지만 FreeRTOS/Source/queue.c 소스 파일 외부에서는 대기열 핸들이 보이드를 가리키는 포인터입니다.

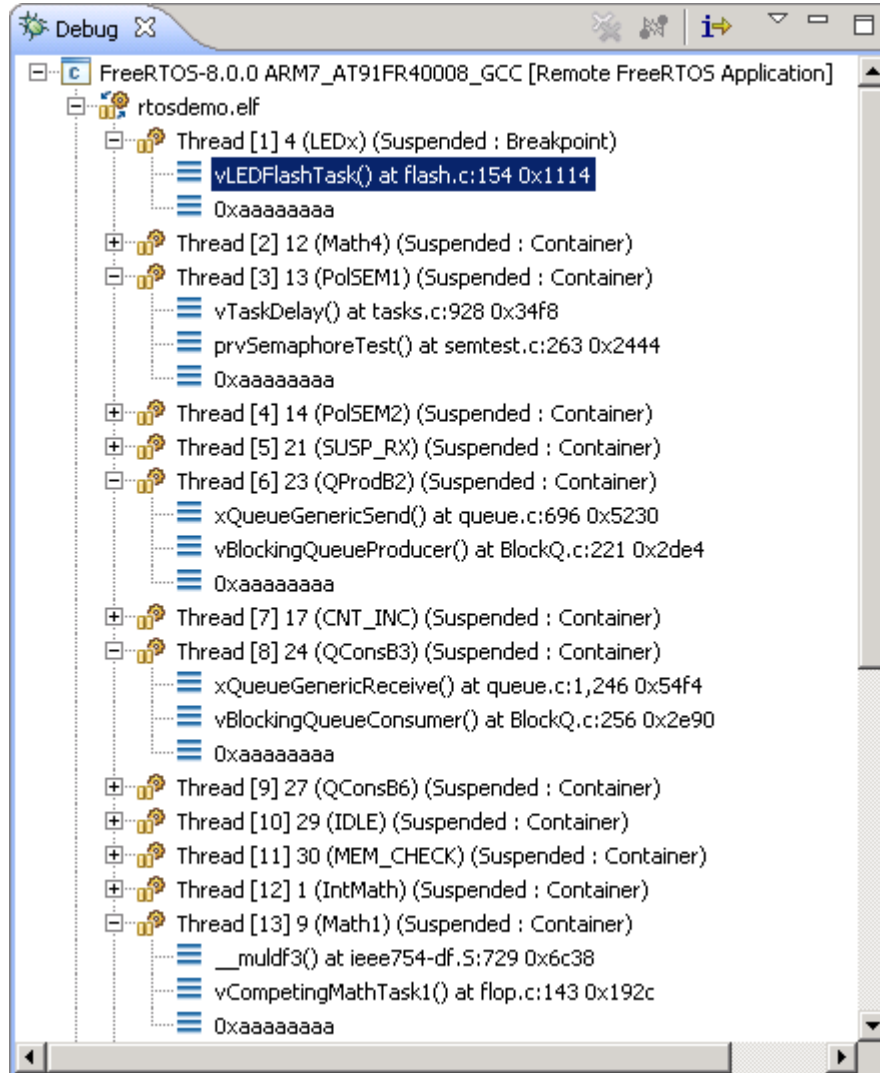
추적 매크로를 사용해 프라이빗 FreeRTOS 데이터 구조에 직접 액세스하는 경우에는 각별한 주의가 필요합니다. 프라이빗 데이터 구조는 FreeRTOS 버전이 다르면 바뀔 수도 있기 때문입니다.

## FreeRTOS 인식형 디버거 플러그인

FreeRTOS 인식 기능을 지원하는 플러그인은 다음 IDE에 사용할 수 있습니다. 단, 이 목록이 전부는 아닙니다.

- Eclipse(StateViewer)
- Eclipse(ThreadSpy)
- IAR
- ARM DS-5
- Atollic TrueStudio
- Microchip MPLAB
- iSYSTEM WinIDEA

다음 그림은 Code Confidence의 FreeRTOS ThreadSpy Eclipse 플러그인입니다.



# 문제 해결

## 단원 소개 및 범위

이번 단원에서는 다음과 같이 새로운 FreeRTOS 사용자들에게 가장 자주 발생하는 문제에 대해서 다루겠습니다.

- 인터럽트 우선순위의 잘못된 할당
- 스택 오버플로우
- printf()의 부적절한 사용

configASSERT()를 사용하면 가장 자주 발생하는 오류 원인을 즉시 트래핑하여 알아낼 수 있기 때문에 생산성이 향상됩니다. 따라서 FreeRTOS 애플리케이션을 개발하거나 디버깅할 때 configASSERT()를 정의하는 것이 매우 바람직합니다. configASSERT()에 대한 자세한 내용은 [개발자 지원 \(p. 217\)](#) 단원을 참조하십시오.

## 인터럽트 우선순위

참고: 인터럽트 우선순위는 지원 요청이 발생하는 가장 큰 원인입니다. 대부분 포트에서 configASSERT()를 정의하면 바로 오류를 포착할 수 있습니다.

사용 중인 FreeRTOS 포트가 인터럽트 중첩을 지원하고, 인터럽트 서비스 루틴이 FreeRTOS API를 사용하는 경우에는 인터럽트의 우선순위를 [인터럽트 관리 \(p. 117\)](#) 단원에서 설명하는 대로 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY 이하로 설정해야 합니다. 우선순위를 올바르게 설정하지 않으면 비효율적인 임계 영역으로 인해 간헐적으로 결함이 발생합니다.

다음과 같은 프로세서에서 FreeRTOS를 실행할 때는 주의가 필요합니다.

- 일부 ARM Cortex 프로세서에서 그렇듯이 인터럽트 우선순위가 가장 높은 우선순위로 기본 설정됩니다. 이러한 프로세서에서는 FreeRTOS API를 사용하는 인터럽트의 우선순위를 초기화하지 않은 채 그대로 두어서는 안 됩니다.
- 높은 우선순위의 숫자가 논리적으로 낮은 인터럽트 우선순위를 표현하여 직관적으로 보이지 않을 수 있는 프로세서에서 주의해야 합니다. 다시 말해서 ARM Cortex 프로세서를 비롯한 기타 가능한 프로세서가 그렇습니다.
- 예를 들어 이러한 종류의 프로세서에서는 우선순위 5로 실행되는 인터럽트가 우선순위 4인 인터럽트에 게 중단될 수 있습니다. 따라서 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY가 5로 설정되어 있으면 FreeRTOS API를 사용하는 인터럽트에 5보다 높거나 같은 우선순위의 숫자만 할당할 수 있습니다. 이 경우 인터럽트 우선순위 5 또는 6은 유효하지만 인터럽트 우선순위 3은 아무런 효과도 없습니다.
- 라이브러리 구현체에 따라 인터럽트의 우선순위를 다르게 지정할 수 있습니다. 이는 특히 ARM Cortex 프로세서를 대상으로 하는 라이브러리와 또 한 번 관련된 것으로서, ARM Cortex 프로세서에서는 인터럽트 우선순위가 하드웨어 레지스터에 작성되기 전에 비트 시프트를 수행하기 때문입니다. 일부 라이브러리는 비트 시프트를 직접 수행하지만 우선순위가 라이브러리 함수에 전달되기 전에 비트 시프트를 수행해야 하는 라이브러리도 있습니다.
- 동일한 아키텍처라고 해도 구현체가 다르면 구현되는 인터럽트 우선순위 비트의 수도 다릅니다. 예를 들어 한 제조사의 Cortex-M 프로세서가 우선순위 비트를 3개 구현한다고 할 때 또 다른 제조사의 Cortex-M 프로세서는 우선순위 비트를 4개 구현할 수도 있습니다.



- 인터럽트의 우선순위를 정의하는 비트가 선점 우선순위를 정의하는 비트와 하위 우선순위를 정의하는 비트 사이에서 분할될 수 있습니다. 모든 비트가 선점 우선순위를 지정할 수 있도록 할당되어 하위 우선순위를 사용하지 않도록 해야 합니다.

일부 FreeRTOS 포트에서는 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY가 configMAX\_API\_CALL\_INTERRUPT\_PRIORITY로 불리기도 합니다.

## 스택 오버플로우

스택 오버플로우는 지원 요청이 두 번째로 많이 발생하는 원인입니다. FreeRTOS는 스택 관련 문제를 찾아 디버깅할 수 있도록 몇 가지 기능을 제공하고 있습니다. (FreeRTOS Windows 포트에서는 이러한 기능들이 지원되지 않습니다)

### uxTaskGetStackHighWaterMark() API 함수

작업은 생성 시 지정되는 총 크기인 스택을 자체적으로 관리합니다. uxTaskGetStackHighWaterMark()는 작업이 할당된 스택 공간의 오버플로우에 얼마나 가까워졌는지 쿼리하는 데 사용됩니다. 이 값을 스택의 하이 워터 마크라고 부릅니다.

uxTaskGetStackHighWaterMark() API 함수 프로토타입은 다음과 같습니다.

```
UBaseType_t uxTaskGetStackHighWaterMark( TaskHandle_t xTask );
```

다음 표는 uxTaskGetStackHighWaterMark() 파라미터와 반환 값을 나열한 것입니다.

파라미터 이름/반환 값	설명
xTask	스택 하이 워터 마크를 쿼리할 작업(대상 작업)의 핸들입니다. 작업 핸들을 가져오는 방법에 대한 자세한 내용은 xTaskCreate() API 함수에서 pxCreatedTask 파라미터를 참조하십시오. 작업은 유효한 작업 핸들 대신에 NULL을 전달하여 자체 스택 하이 워터 마크를 쿼리할 수 있습니다.
반환 값	작업에서 사용하는 스택 크기는 작업이 실행되고, 인터럽트가 처리될 때마다 늘어나거나 줄어듭니다. uxTaskGetStackHighWaterMark()는 작업 실행 이후 사용할 수 있는 나머지 스택 공간의 최소 크기를 반환합니다. 이 값은 스택 사용량이 가장 큰(또는 가장 깊은) 값에 도달했을 때 사용하지 않고 남아있는 스택의 크기를 나타냅니다. 하이 워터 마크가 0에 가까울수록 작업이 스택 오버플로우에 더욱 가까워집니다.

## 런타임 스택 검사에 대한 개요

FreeRTOS는 2가지 런타임 스택 검사 메커니즘이 옵션으로 포함되어 있습니다. 이 두 가지 메커니즘은 FreeRTOSConfig.h의 configCHECK\_FOR\_STACK\_OVERFLOW 컴파일 시간 구성 상수에서 제어합니다. 두 방법 모두 컨텍스트 전환에 걸리는 시간이 늘어납니다.

스택 오버플로우 후크(또는 스택 오버플로우 콜백)란 스택 오버플로우를 감지했을 때 커널에서 호출되는 함수를 말합니다. 스택 오버플로우 후크 함수를 사용하는 방법은 다음과 같습니다.

1. FreeRTOSConfig.h에서 configCHECK\_FOR\_STACK\_OVERFLOW를 1 또는 2로 설정합니다.
2. 아래와 같이 함수 이름과 프로토타입을 사용해 후크 함수의 구현체를 입력합니다.

```
void vApplicationStackOverflowHook( TaskHandle_t *pxTask, signed char*pcTaskName );
```

스택 오버플로우 후크를 입력하는 이유는 스택 오류를 더욱 쉽게 찾아내 디버깅하는 데 있지만 스택 오버플로우를 실제로 해결할 수 있는 방법은 없습니다. 함수의 파라미터가 스택 오버플로우가 발생한 작업의 핸들과 이름을 후크 함수에게 전달합니다.

스택 오버플로우 후크는 인터럽트의 컨텍스트에서 호출됩니다.

일부 마이크로컨트롤러는 잘못된 메모리 액세스를 감지하면 오류 예외를 발생시키기도 합니다. 커널이 스택 오버플로우 후크 함수를 호출하기 전에 오류 예외가 발생하도록 하는 것도 가능합니다.

## 런타임 스택 검사를 위한 방법 1

방법 1은 실행 속도가 빠르지만 컨텍스트 전환 사이에서 발생하는 스택 오버플로우를 놓치기 쉽습니다. 이 방법은 configCHECK\_FOR\_STACK\_OVERFLOW가 1로 설정되어 있을 때 사용합니다.

스왑 아웃될 때마다 작업의 전체 실행 컨텍스트가 스택에 저장됩니다. 이때는 스택 사용량이 최대 값에 도달하는 시점일 가능성이 높습니다. configCHECK\_FOR\_STACK\_OVERFLOW가 1로 설정되면 커널이 컨텍스트 저장 후 스택 포인터가 유효한 스택 공간을 벗어나지 않는지 검사합니다. 이때 스택 포인터가 유효한 범위를 벗어난 것으로 밝혀지면 스택 오버플로우 후크가 호출됩니다.

## 런타임 스택 검사를 위한 방법 2

방법 2에서는 추가 검사를 수행합니다. 이 방법은 configCHECK\_FOR\_STACK\_OVERFLOW가 2로 설정되어 있을 때 사용합니다.

작업이 생성되면 스택은 알려진 패턴으로 채워집니다. 방법 2는 작업 스택 공간에서 마지막 유효 바이트 20개를 테스트하여 이 패턴을 덮어쓰지 않았는지 확인합니다. 바이트 20개 중 하나라도 예상 값에서 바뀌면 스택 오버플로우 후크 함수가 호출됩니다.

방법 2는 방법 1만큼 실행 속도가 빠르지 않지만 바이트를 20개만 테스트하기 때문에 비교적 빠른 편에 속합니다. 또한 모든 스택 오버플로우를 찾아낼 가능성도 가장 높습니다.

## printf() 및 sprintf()의 부적절한 사용

printf()의 부적절한 사용은 오류가 발생하는 공통 원인입니다. 이러한 오류를 알지 못하는 애플리케이션 개발자들은 종종 디버깅을 위해 printf() 호출을 추가하지만 이러한 과정에서 문제가 더욱 커집니다.

다수의 크로스 컴파일러 공급업체들은 소형 임베디드 시스템에서 사용하는 데 적합한 printf() 구현체를 제공합니다. 그렇더라도 이러한 구현체들은 스레드 세이프가 아니어서 인터럽트 서비스 루틴 내부에서 사용하는 데 부적합할 수도 있습니다. 또한 출력 위치에 따라 실행 시간이 비교적 오래 걸릴 수도 있습니다.

소형 임베디드 시스템 전용으로 설계된 printf() 구현체를 사용하지 못해서 일반적인 printf() 구현체를 사용해야 한다면 다음과 같은 이유로 주의해야 합니다.

- printf() 또는 sprintf() 호출을 추가하는 것만으로도 애플리케이션의 실행 파일 크기가 크게 증가할 수 있습니다.
- printf() 및 sprintf()는 malloc()을 호출할 수 있지만 heap\_3이 아닌 다른 메모리 할당 체계를 사용하는 경우에는 잘못될 수 있습니다. 자세한 내용은 [메모리 할당 체계 예제 \(p. 14\)](#) 단원을 참조하십시오.
- printf() 및 sprintf()가 다른 경우에 필요했을 크기보다 몇 배 더 큰 스택을 요구할 수도 있습니다.

## Printf-stdarg.c

FreeRTOS 시연 프로젝트는 대부분 printf-stdarg.c 파일을 사용합니다. 이 파일은 표준 라이브러리 버전 대신에 사용할 수 있도록 sprintf() 구현체를 스택 효율적으로 용량을 최소화하여 제공합니다. 대부분 경우에 sprintf()와 관련 함수를 호출하는 각 작업에 훨씬 작은 크기의 스택을 할당할 수 있기 때문입니다.

printf-stdarg.c 파일은 printf() 출력을 문자 단위로 포트에 보내는 메커니즘을 제공하기도 합니다. 이렇게 하면 속도가 느릴 경우 스택 사용량이 크게 줄어드는 효과가 있습니다.

printf-stdarg.c 복사본이라고 해서 모두가 snprintf()를 구현하는 것은 아닙니다. sprintf()로 직접 매핑되어 버퍼 크기 파라미터를 무시하는 복사본은 snprintf()를 구현하지 못합니다.

Printf-stdarg.c는 오픈 소스이지만 타사가 소유하고 있기 때문에 FreeRTOS에서 별도로 라이선스를 획득해야 합니다. 라이선스 약관은 소스 파일 상단에 포함되어 있습니다.

## 기타 공통 오류

이번 단원에서는 기타 공통 오류와 잠재적 원인, 그리고 해결책에 대해서 설명하겠습니다.

### 증상: 단순 작업을 데모에 추가하면 데모가 충돌을 일으킵니다

작업을 생성하려면 힙에서 메모리를 가져와야 합니다. 데모 애플리케이션 프로젝트는 대부분 데모 작업을 생성할 만큼만 정확한 크기로 힙을 정의하기 때문에 작업이 생성된 후에는 추가 작업, 대기열, 이벤트 그룹 또는 세마포어를 위해 남겨진 힙이 부족합니다.

vTaskStartScheduler()를 호출하면 유휴 작업과 가능하다면 RTOS 데몬 작업까지 자동으로 생성됩니다. vTaskStartScheduler()는 이러한 작업을 생성하는 데 필요한 힙 메모리가 부족할 때만 값을 반환합니다. vTaskStartScheduler() 호출 뒤에 NULL 루프로 [for(;;)]를 추가하면 이러한 오류를 더욱 쉽게 디버깅할 수 있습니다.

작업을 더 추가하려면 힙 크기를 늘리거나 기존 데모 작업 중 일부를 삭제하십시오. 자세한 내용은 [메모리 할당 체계 예제](#) (p. 14) 단원을 참조하십시오.

### 증상: 인터럽트에서 API 함수를 사용하면 애플리케이션이 충돌을 일으킵니다

API 함수는 이름이 '...FromISR()'로 끝나지 않는다면 인터럽트 서비스 루틴에서 사용하지 마십시오. 특히 인터럽트 세이프 매크로를 사용하지 않을 경우 임계 영역을 인터럽트에 생성해서는 안 됩니다. 자세한 내용은 [인터럽트 관리](#) (p. 117) 단원을 참조하십시오.

인터럽트 중첩을 지원하는 FreeRTOS 포트일 경우 인터럽트 우선순위가 configMAX\_SYSCALL\_INTERRUPT\_PRIORITY보다 높게 할당된 인터럽트에서 API 함수를 사용하지 마십시오. 자세한 내용은 [인터럽트 중첩](#) (p. 148) 단원을 참조하십시오.

### 증상: 인터럽트 서비스 루틴에서 애플리케이션이 간혹 충돌을 일으킵니다

이때는 인터럽트로 인해 스택 오버플로우가 발생하지 않는지 가장 먼저 검사해야 합니다. 일부 포트는 인터럽트가 아닌 작업에서만 스택 오버플로우 유무를 검사합니다.

인터럽트의 정의 및 사용 방식은 포트일 때와 컴파일러일 때마다 다릅니다. 따라서 두 번째로 인터럽트 서비스 루틴에서 사용되는 구문, 매크로 및 호출 규칙이 사용 중인 포트의 설명서 페이지에 설명되어 있는 것과 정확히 일치하는지, 그리고 포트와 함께 제공되는 데모 애플리케이션에 설명되어 있는 것과 정확히 일치하는지 검사해야 합니다.

애플리케이션이 낮은 우선순위의 숫자를 사용해 논리적으로 높은 우선순위를 표현하는 프로세서에서 실행되고 있다면 직관적으로 보이지 않을 수도 있기 때문에 각 인터럽트의 우선순위가 이점을 고려해 할당되었는지 확인해야 합니다. 애플리케이션이 각 인터럽트의 우선순위를 가장 높은 우선순위로 기본 설정하는 프로세서에서 실행되고 있다면 각 인터럽트의 우선순위를 바꾸지 않고 기본 값으로 설정해서는 안 됩니다. 자세한 내용은 [인터럽트 중첩 \(p. 148\)](#) 단원을 참조하십시오.

## 증상: 스케줄러가 첫 번째 작업을 시작하려고 하면 충돌을 일으킵니다.

FreeRTOS 인터럽트 핸들러가 설치되어 있는지 확인하십시오. 예제가 필요하다면 포트에 제공되는 데모 애플리케이션을 참조하십시오.

일부 프로세서는 스케줄러를 시작하기 전에 모든 권한이 제공되는 모드(privileged mode)이어야 합니다. 가장 쉬운 방법은 main() 호출 이전에 C 시작 코드에서 프로세서를 모든 권한이 제공되는 모드로 전환하는 것입니다.

## 증상: 인터럽트가 갑자기 비활성화되거나, 혹은 임계 영역이 정확히 중첩되지 않습니다

FreeRTOS API 함수가 스케줄러 시작 이전에 호출되면 인터럽트가 의도적으로 비활성화됩니다. 이때는 첫 번째 작업이 실행을 시작해야만 인터럽트가 다시 활성화됩니다. 이렇게 하면 인터럽트가 시스템 초기화 도중에, 스케줄러 시작 이전에, 스케줄러가 일치하지 않는 상태일 때 FreeRTOS API 함수를 사용하려고 해서 시스템이 충돌을 일으키는 것을 방지할 수 있습니다.

taskENTER\_CRITICAL() 및 taskEXIT\_CRITICAL() 호출을 사용해 마이크로컨트롤러 인터럽트 활성화 비트 또는 우선순위 플래그를 변경하십시오. 그 밖에 다른 방법은 사용하지 마십시오. 두 매크로는 호출 중첩 깊이를 계산하여 호출 중첩이 완전히 0으로 해제되었을 때만 인터럽트가 다시 활성화되도록 할 수 있습니다. 일부 라이브러리 함수는 인터럽트를 활성화하거나 비활성화할 수도 있습니다.

## 증상: 스케줄러가 시작되기 전에도 애플리케이션이 충돌을 일으킵니다.

컨텍스트 전환을 초래할 수 있는 인터럽트 서비스 루틴은 스케줄러가 시작되기 전에 실행하지 마십시오. 대기열이나 세마포어 같은 FreeRTOS 객체로 전송하거나 FreeRTOS 객체에서 수신하는 인터럽트 서비스 루틴 역시 마찬가지입니다. 컨텍스트 전환은 스케줄러가 시작되어야만 가능합니다.

대부분 API 함수는 스케줄러가 시작될 때까지 호출하지 못합니다. API 사용은 vTaskStartScheduler()를 먼저 호출한 후 작업, 대기열 및 세마포어 같은 객체를 사용하는 시점보다는 이러한 객체를 생성하는 시점으로 제한하는 것이 좋습니다.

## 증상: 스케줄러가 일시 중지된 상태에서, 혹은 임계 영역 내부에서 API 함수를 호출하면 애플리케이션이 충돌을 일으킵니다

스케줄러는 vTaskSuspendAll()을 호출하여 일시 중지되고, xTaskResumeAll()을 호출하여 다시 시작됩니다 (일시 중지가 해제됩니다) 임계 영역에 진입할 때는 taskENTER\_CRITICAL()을 호출하고, 임계 영역에서 나올 때는 taskEXIT\_CRITICAL()을 호출합니다.

FreeRTOS 커널 개발자 안내서  
증상: 스케줄러가 일시 중지된 상태에서, 혹은 임계 영역 내부에서 API 함수를 호출하면 애플리케이션이 충돌을 일으킵니다

---

임계 영역 내부에서, 혹은 스케줄러가 일시 중지된 상태일 때는 API 함수를 호출하지 마십시오.