RTOS
(Real Time Operating System)
&
FreeRTOS Porting

RTOS 개요

- **Real Time 이란?**
  - 설계자가 정한 시간 내에 태스크(작업) 처리 또는 실행이 완료되어야 하는 시스템
  - 태스크 처리 또는 실행 속도와는 무관

- **Operating System 이란?**
  - 하드웨어와 소프트웨어를 관리하는 관리 프로그램
  - 일상세계에서의 정부(government)와 비슷한 역할
  - 프로그램 또는 태스크 실행관리, 메모리, 주변장치와 같은 하드웨어 리소스(Resource)관리

- **RTOS (Real Time Operating System) 란?**
  - Real Time 성능을 보장하는 Operating System

RTOS의 특성

- OS specific features
  - Multi-Tasking 지원
  - 선점형 스케줄링 지원(Preemptive scheduling)
- Embedded system features
  - ROMable : Low memory footprint
  - Fast execution : Exclude genreral purpose & features
  - Scalable : ROM/RAM size & OS services
  - Portable : C language
- Objects
  - Task, Semaphore, Mutex, Message Queue, Event, ISR
- Services
  - Timer process, Interrupt Process, Resource management

RTOS 주요 요건

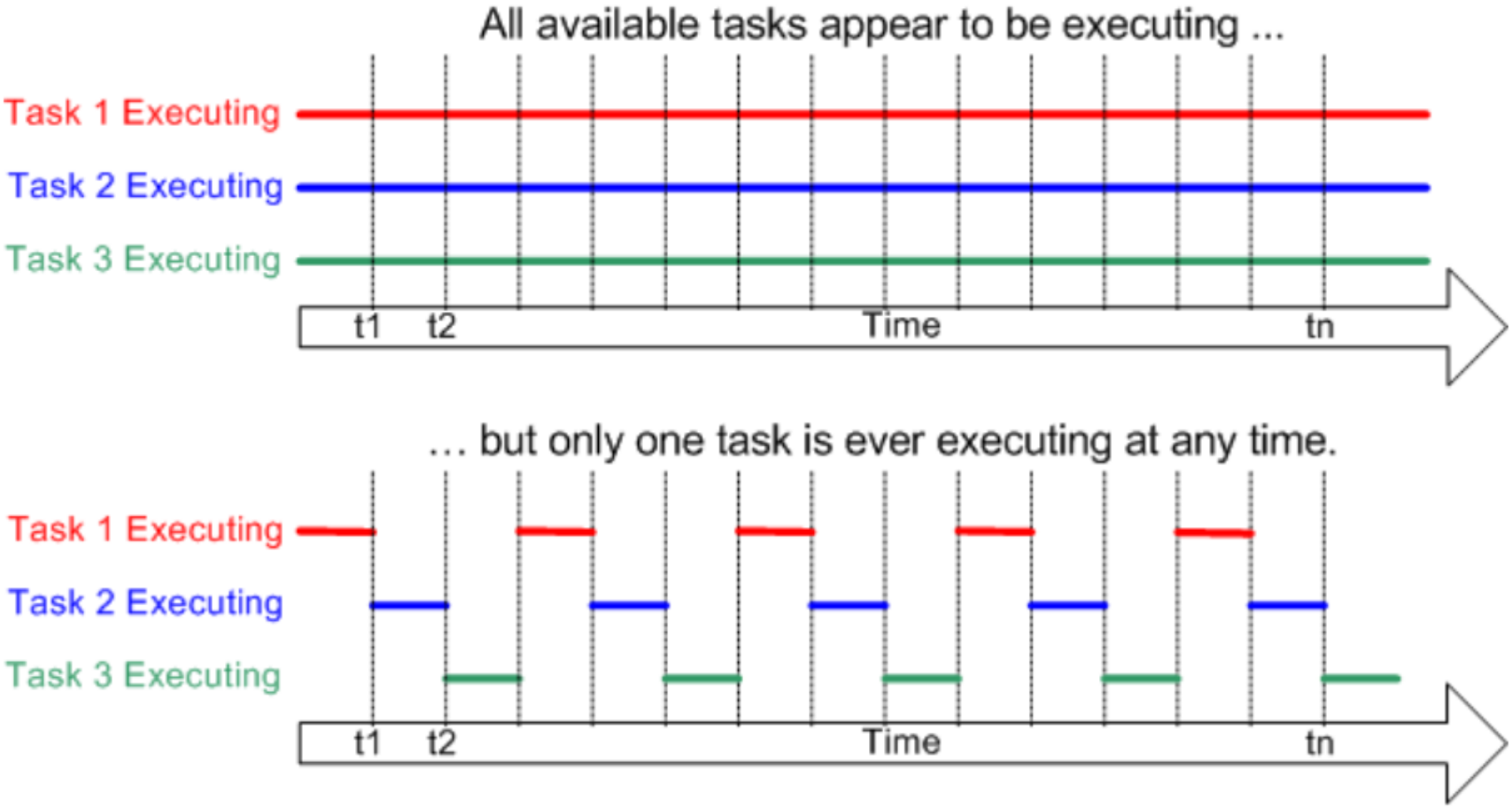
- 신뢰성 (Reliability)
  - 임베디드 시스템에 필수 요소
- 예측성 (Predictability)
  - 실시간 성능을 만족시키기 위함, 확정적(deterministic)과 혼용
- 고성능 (Performance)
  - 실행해야 할 태스크가 많아질 수록, 사용하는 오브젝트가 많을수록 필요
- 간결성 (Compactness)
  - 하드웨어 설계에 따라 cost가 민감한 설계 요구사항을 만족하기 위함
- 확장성 (Scalability)
  - 파일시스템, 네트워크 스택과 같은 미들웨어를 쉽게 추가 삭제
  - 동일한 RTOS로 여러 개의 프로젝트 진행시 시간 비용 절감 이득 효과

Task scheduling 및 동작

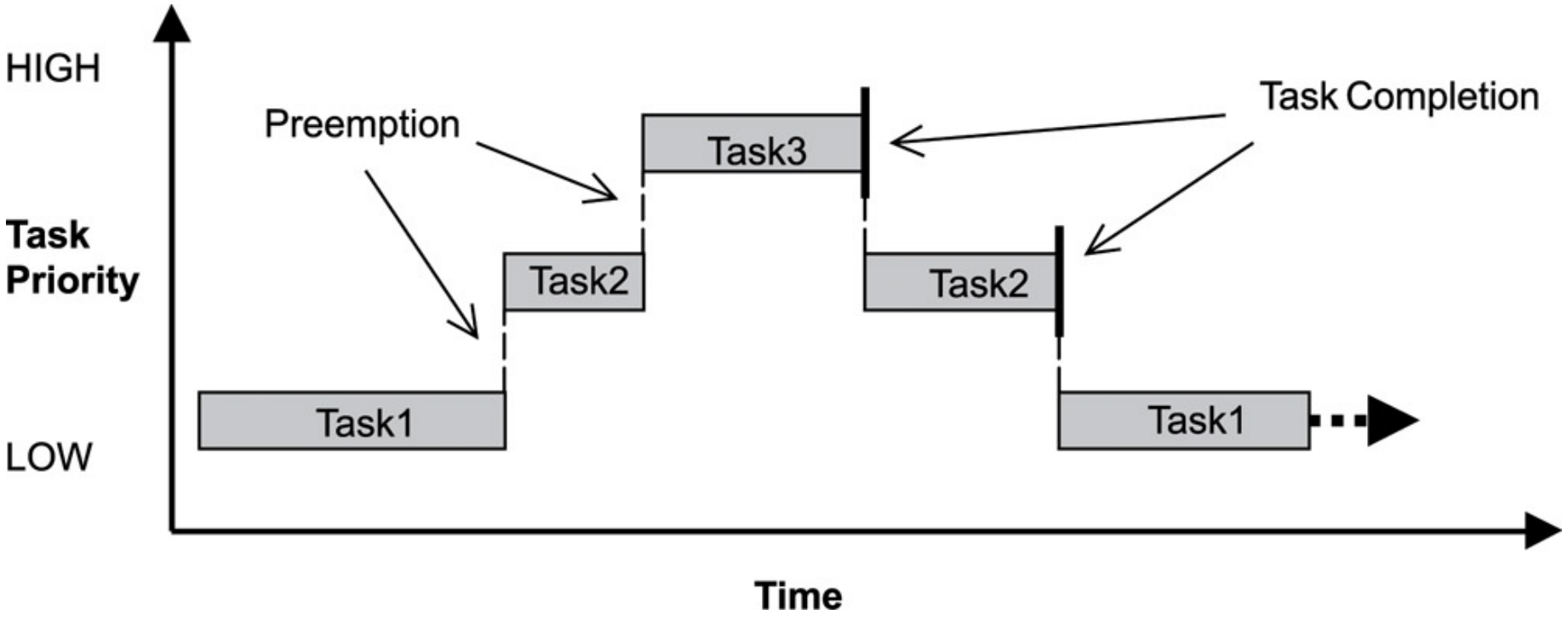
- Task
  - 스케줄링 가능한 일련의 명령어들로 구성된 독립적인 실행 단위
  - 각 실행 단위들(Task)이 동시에 실행하는 것 처럼 보이는 것이 멀티 태스킹
  - 스케줄러는 정확한 시간에 적절한 태스크가 실행되는 것을 보장
  - Task 개수가 증가할 수록 CPU의 overhead 증가

- Scheduler
  - Task 실행 전환(switching)을 관리
  - 세부적으로는 문맥전환(Context Switching)의 역할
  - 라운드 로빈(round-robin) 스케줄링
  - 우선순위에 기반한 선점형(preemptive) 스케줄링

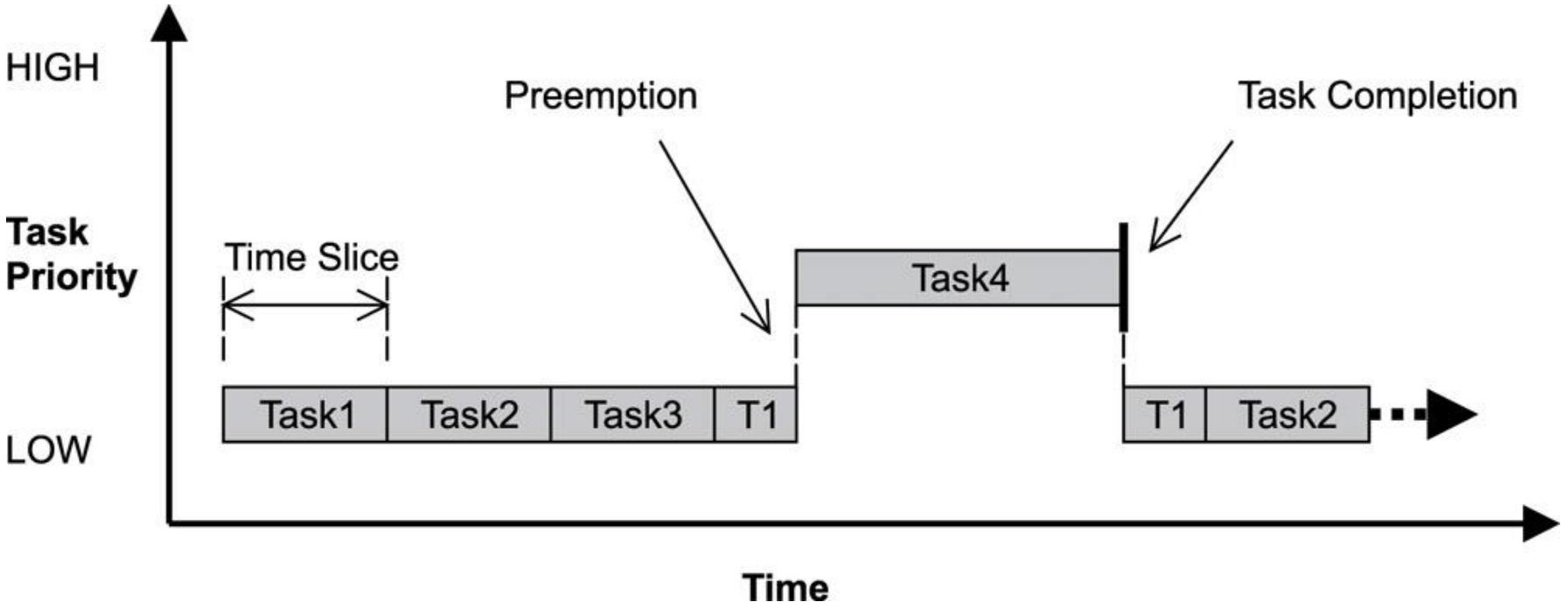
Multi-tasking



Priority based Preemption scheduling

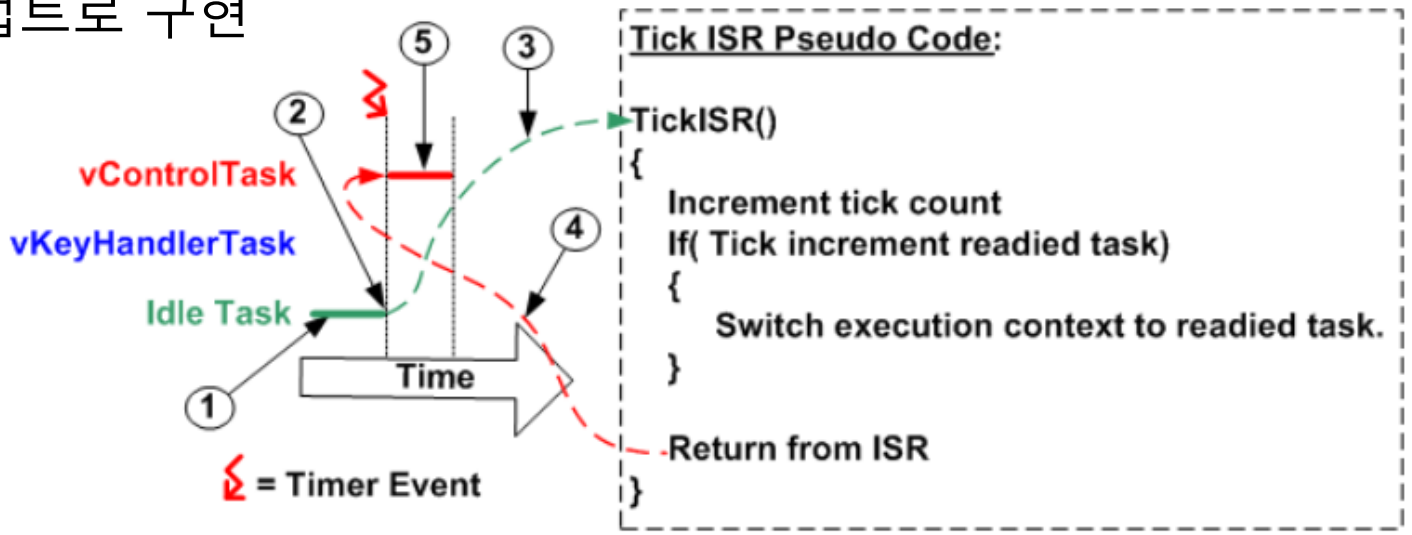


Round Robin scheduling with priority based preemption scheduling

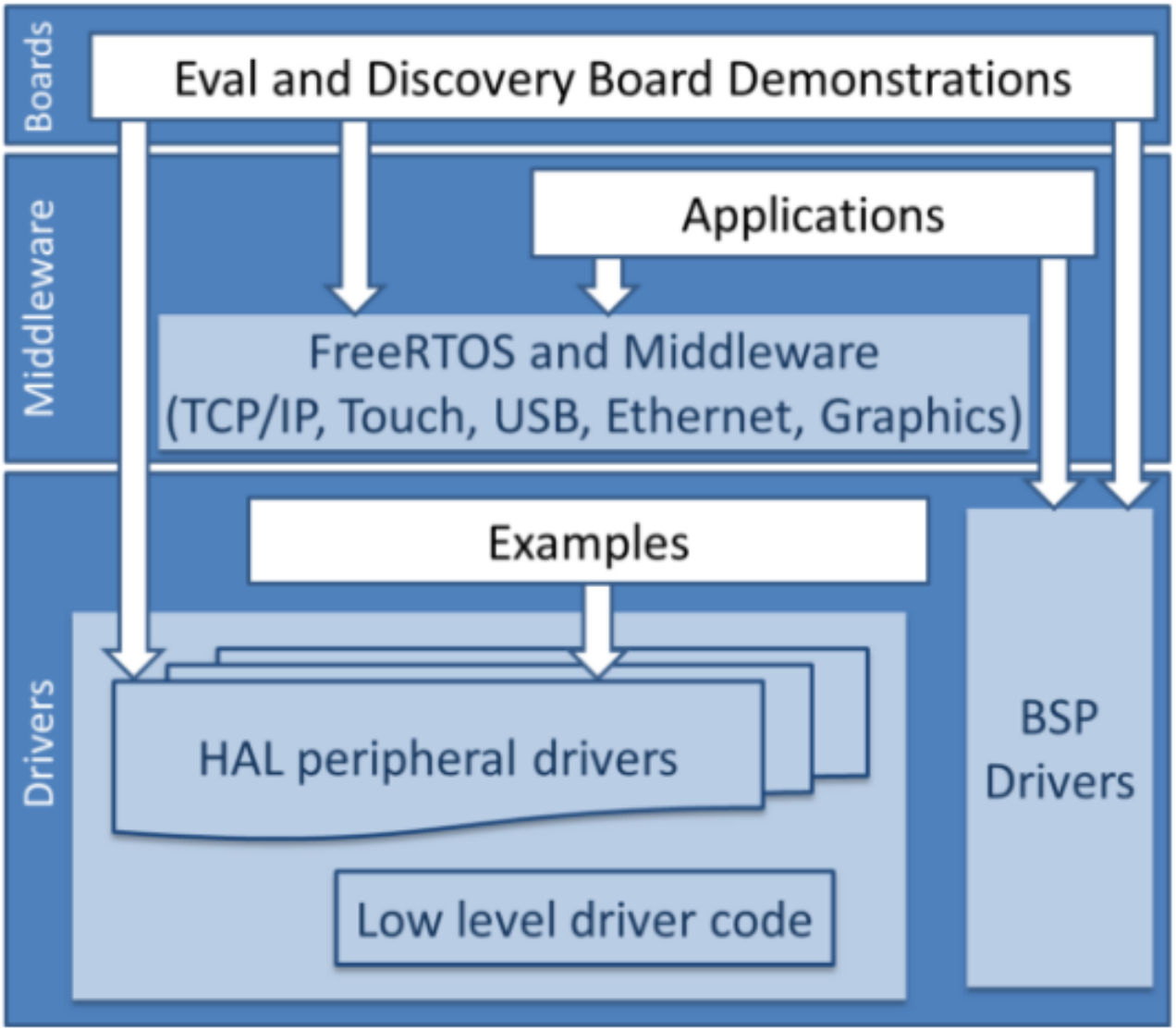


Clock Tick

- 일명 Heart beat 라고도 함
- RTOS 커널 내부의 기준 시간
  - task switching 여부를 주기적으로 판단하기 위해 kernel scheduler 호출
  - 시간관리 서비스 함수, 각 종류별 오브젝트의 실행 블로킹(blocking) 시간 관리 등
- System performance에 따라 보통 1ms~10ms 주기의 타이머 인터럽트로 구현

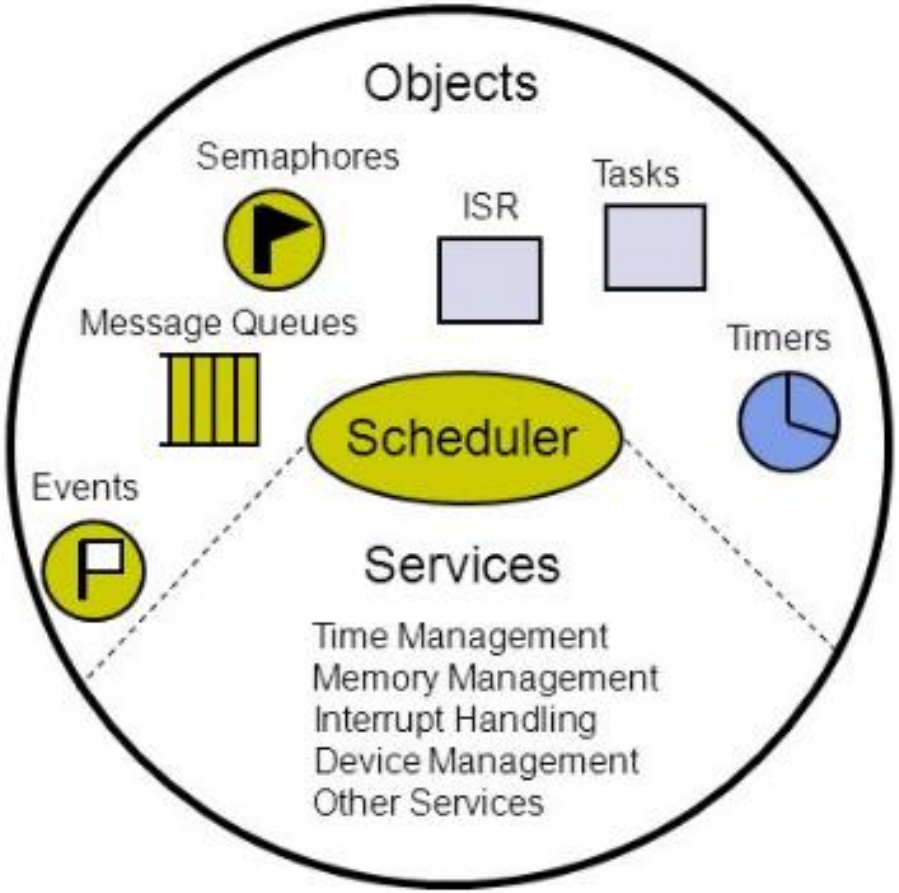


FreeRTOS Software Stack Example



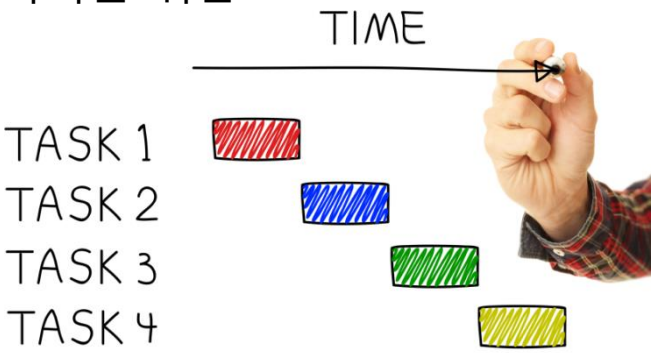
RTOS Objects

- Task
- Semaphore
- Mutex
- Message Queue
- Message Mailbox
- Events
- Timers
- ISR



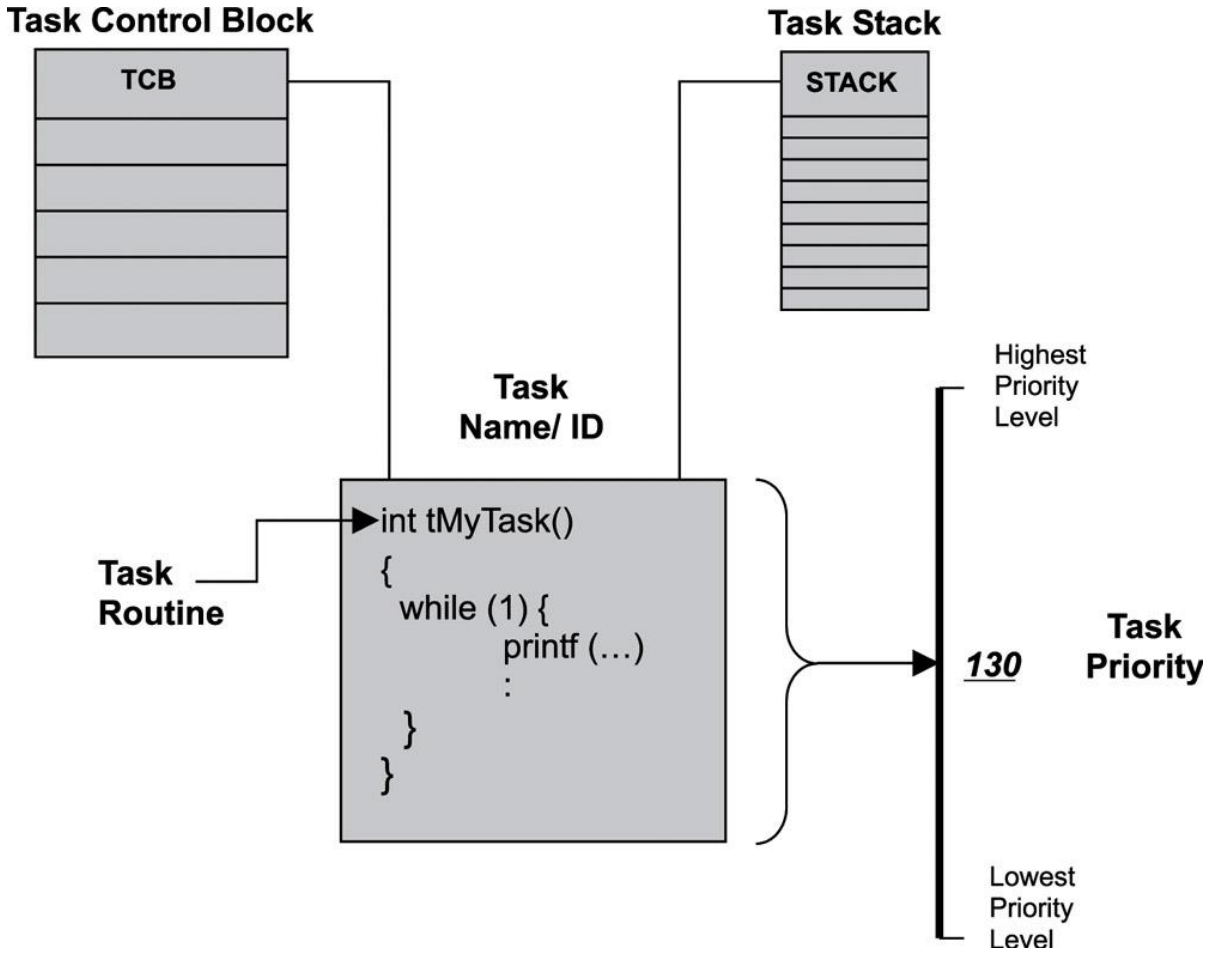
Task 정의

- 병행 설계(Concurrent design) 필요
  - 한번에 하나의 동작만을 순차적으로 처리하는 응용 프로그램은 실시간 성능을 만족 시키기 어려움
  - 여러 가지 입출력에 대한 실시간 성능을 만족시키기 위함
- 스케줄링이 가능한 작고 순차적인 프로그램 단위로 쪼갬
- 태스크는 다른 태스크들과 CPU를 사용하기 위해 경쟁하는 독립적인 프로그램 실행 단위
- 응용 프로그램을 여러 개의 태스크로 분류
  - 정해진 시간제약 안에서 최적화된 입출력 작업 처리를 위함
- 태스크는 여러 설정 값과 관련된 자료구조로 정의
- 태스크는 스케줄링이 가능



Task 구성

- 태스크 오브젝트(Task Object)
  - 이름, ID, 우선순위, TCB(Task Control Block), 스택, 태스크 루틴



System Task

- 시스템 태스크란?
  - 시스템 시작 시 커널이 생성하는 태스크
  - 시스템 레벨 우선순위 할당
  - 응용 프로그램은 시스템 레벨 우선순위를 변경하거나 사용하면 안됨

- 시스템 태스크의 예
  - 초기화 태스크
  - 유휴(IDLE) 태스크
  - 기록(Logging) 태스크
  - 예외 처리(Exception-handling) 태스크
  - 디버그 에이전트(Debug agent) 태스크

Task Body & Idle Task

- Task Body
  - 무한 실행 루프

```
portTASK_FUNCTION( vATaskFunction, pvParameters )
{
    for( ;; )
    {
        — Task application code here. —
    }
}
```

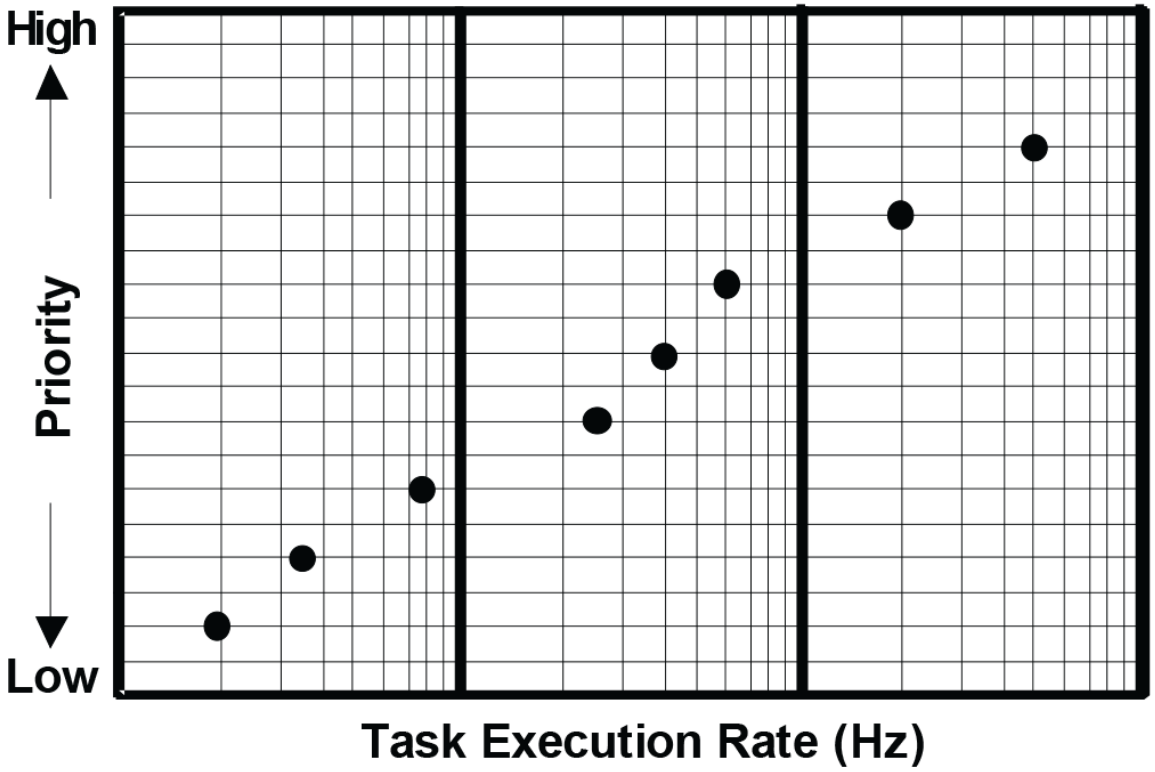
- Idle Task
  - 커널이 실행될 때 반드시 실행되는 태스크
  - 응용 프로그램의 태스크들 중 프로세서를 점유하고 있는 태스크가 없을 때 실행됨
  - Hooking 함수를 추가하여, Idle 태스크가 실행될 때 추가적인 기능을 처리할 수 있음 (예, sleep enter)

Task Priority

- 실시간(Real-Time) 성능을 만족시키기 위해 필요
  - 우선순위가 높은 태스크가 항상 프로세서를 점유
    -> Priority based Preemptive scheduling
  - 보통 하나의 태스크가 하나의 우선순위를 가진다
  - 동일한 우선순위를 가지는 태스크들은 Round-Robin 스케줄링으로 운용된다
  - 시스템 태스크에 할당된 우선순위는 응용 프로그램에서 사용 금지

Task Priority (계속)

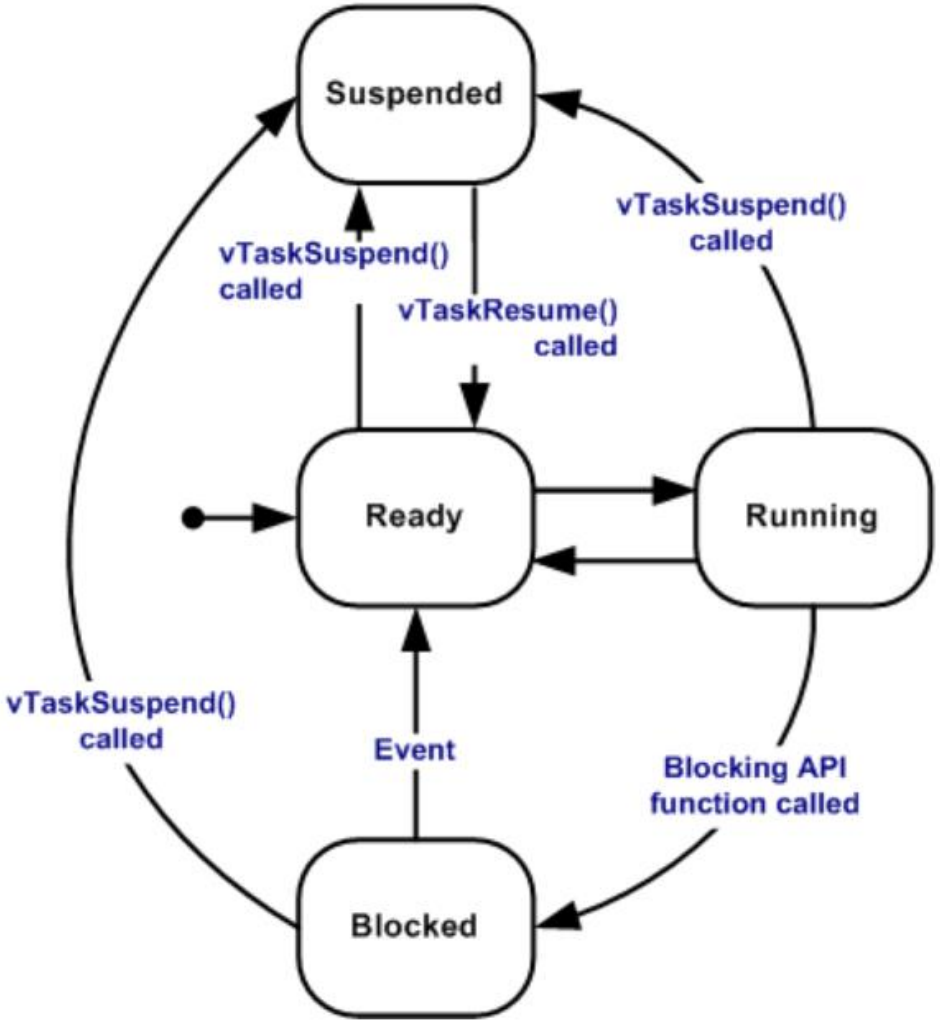
- 태스크 우선순위 부여방법 예
  - RMS(Rate Monotonic Scheduling)방식
  - 실행주기가 빠른 태스크의 우선순위를 높게 책정



Task States

- Running
  - 태스크가 실제로 실행되고 있을 때
  - 프로세서를 현재 사용중인 태스크의 상태
- Ready
  - 실행 가능하지만 현재 실행 중이지 않은 상태
  - 우선순위가 동등 하거나 더 높은 다른 태스크가 이미 Running state에 있기 때문에 현재 실행 중이지 않은 상태
- Blocked
  - 외부 이벤트를 기다리고 있는 상태
  - Timeout을 가지고 블럭킹 상태로 진입 가능
  - 바로 Running state로 갈수는 없음
- Suspended
  - 블럭킹 상태와 비슷, 바로 Running state로 갈수는 없음
  - 블럭킹 상태와 달리 Timeout을 가지고 self-release할 수 없음
  - 외부 API에 의해 suspend 상태로 진입하거나 나올 수 있음

Task State Machine



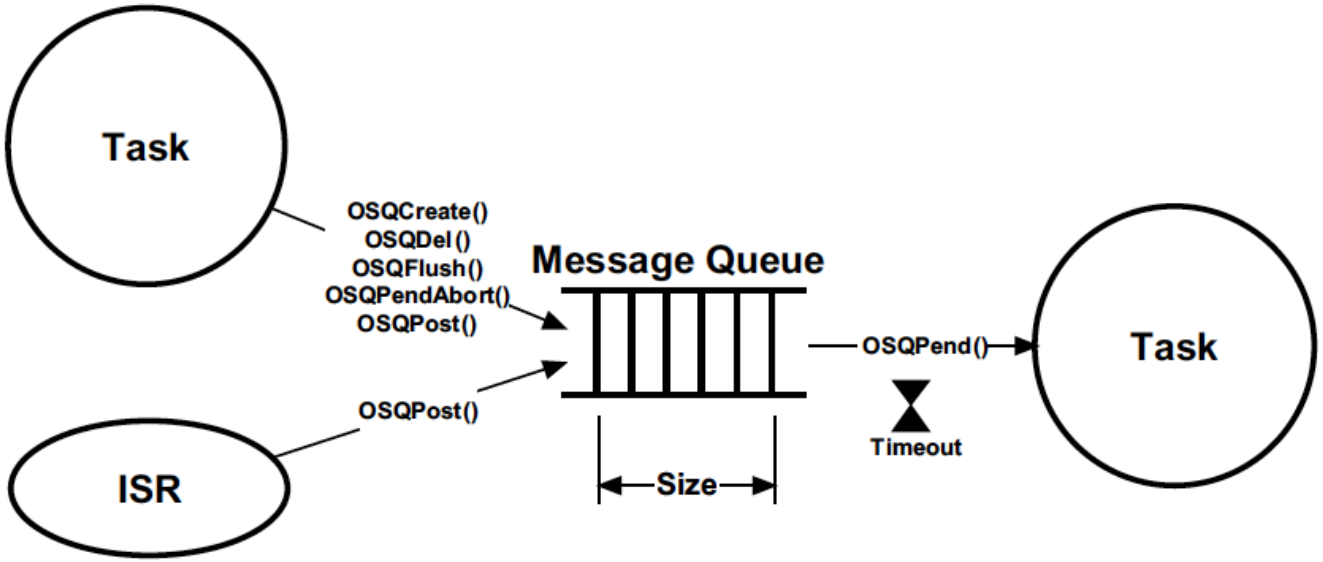
Valid task state transitions

Task APIs

- 태스크 생성 및 삭제
  - xTaskCreate, vTaskDelete
- 태스크 제어
  - vTaskDelay, vTaskPriorityGet/Set, vTaskSuspend/Resume
- 태스크 유틸리티
  - vTaskGetTickCount
- RTOS 커널 제어
  - taskENTER_CRITICAL, taskEXIT_CRITICAL, vTaskStartScheduler, vTaskSuspendAll, vTaskResumeAll

Message Queue의 정의

- 버퍼 형태의 오브젝트
  - 태스크간 또는 ISR과의 메시지(데이터)를 주고 받음
  - 데이터 통신과 동기화 수행
  - 파이프라인과 비슷
  - 수신자가 메시지를 읽어갈 때 까지 임시로 보간
  - 송신 태스크와 수신 태스크를 분리하여 관리
  - Element 개수가 하나 일 때는 Mailbox라고도 부름



Message Queue의 구성

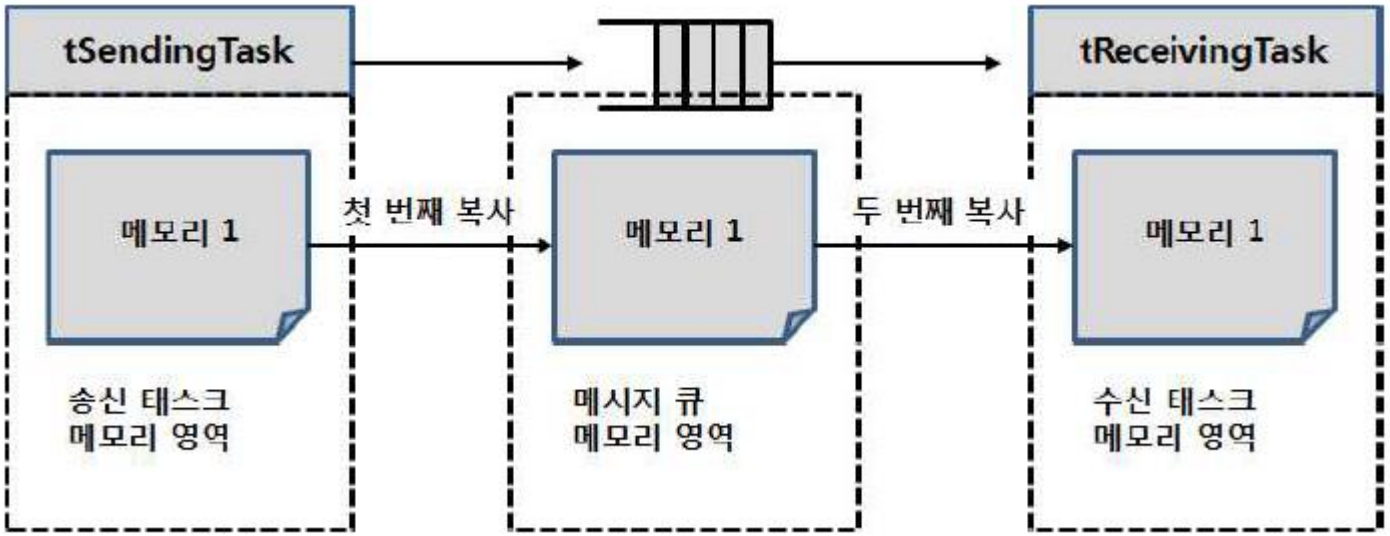
- 메시지를 저장할 수 있는 여러 개의 요소(element/item) 로 구성
  - 선두(front/head)와 후미(end/tail) 요소를 가짐
  - 큐는 비어있는 상태(empty)로 시작
  - 먼저 들어온(수신된) 메시지부터 내보내는(Get) FIFO(First In First Out)방식
  - 우선순위(송신 태스크) 기반으로 메시지를 내보내는(Get) 우선순위 기반

- Total Length
  - 큐에 있는 요소의 총 개수를 의미
  - 개발자가 큐를 생성할 때 지정함
  - Total length에 따라 큐가 생성 될 때 사용할 시스템 메모리의 크기가 결정됨
  - Memory 사용량에 민감한 deeply embedded system에서는 저당한 크기를 가지도록 관리할 필요가 있음

Message Queue 동작-1

- 복사(Copy)에 의한 동작
  - 바이트 단위로 복사된 데이터가 큐에 보내진다는 것을 의미

- 참조(Reference)에 의한 동작
  - 큐에 보내지는 데이터의 포인터만이 보관된다.
  - 데이터 자체는 아님



Message Queue 동작-2

- FreeRTOS는 다음과 같은 이유로 복사에 의한 큐 방식을 사용한다.
  - Stack에 있는 값(지역변수 값)이 직접 큐에 보내짐. 따라서 함수가 return해도 값이 보존됨
  - 큐에 보내진 변수나 버퍼를 즉시 re-use할 수 있음
  - 데이터를 전송한 태스크와 수신한 태스크간에 완전한 디-커플링이 가능. 설계자가 데이터를 소유한 태스크 또는 가져간 태스크들을 고려할 필요가 없음
  - 참조에 의한 큐 동작을 못하는 것도 아님. 예를 들어 일련의 크기를 가지는 데이터를 복사에 의해 큐로 보내는 것은 실용적이지 못하다. 그 대신 데이터 포인터가 큐에 복사될 수 있다.
  - 데이터 전송을 위해 먼저 버퍼를 할당하지 않고, 할당되어 있는 버퍼에 데이터를 복사하여 큐에 보낼 수 있음
  - RTOS 커널이 데이터 저장에 사용되는 메모리 할당에 전적으로 책임 진다.
  - 메모리 보호 시스템(Memory Protected System)인 경우, Task는 메모리 access에 제한이 있게 된다. 참조에 의한 동작은 데이터가 저장된 메모리로의 접근이 보내는 task와 받는 task 둘 다 access가능할 때만 사용할 수 있다. 반면에 복사에 의한 동작은 그런 제약이 내포되어 있지 않고, 커널이 항상 관리하기 때문에, 보호된 메모리 사이의 데이터 교환에 큐를 사용할 수 있다.

Message Queue 동작-3

- 여러 개의 Task에 의한 access
  - 큐는 여러 개의 Task 또는 ISR에 의해 access될 수 있음
  - 여러 개의 태스크가 동일한 큐에 write할 수 있음
  - 여러 개의 태스크가 동일한 큐에서 read할 수 있음
  - 실제로는 여러 개의 Task가 동일한 큐에 write하는 경우는 많지만, 여러 개의 Task가 동일한 큐에서 read하는 경우는 일반적이지 않다.

Message Queue 동작-4

- 큐 read시에 블러킹(blocking)
  - 큐에서 데이터를 read하려고 할 때, timeout(blocking time)을 옵션으로 부가함
  - 이 timeout은 큐가 비어있을 때, 큐로부터 데이터를 read할 수 있을 때까지 Task를 Blocked State로 유지시키는 시간임
  - 큐로부터 데이터를 사용할 수 있기를 기다리는 Blocked State인 Task는 다른 Task 또는 ISR이 큐에 데이터를 write(send)하는 순간 자동으로 Ready State로 이동함
  - 큐에 사용할 수 있는 데이터가 write되지 않더라도 지정한 timeout이 경과하면 자동으로 Blocked State에서 Ready State로 이동함
  - 동일한 큐에 대해 Multiple Reader Task가 존재하는 경우, 데이터 read를 위해 Blocked State로 되어 있는 Task가 하나 이상일 수 있음. 이때 데이터가 큐에 저장되면 하나의 Task만이 Blocked State에서 나와 Ready State로 이동함. 여러 개의 Blocked State인 Task에서 Ready State로 이동하는 Task는 우선순위 기준 또는 동일한 우선순위에서는 가장 오래 기다린 Task가 해당 됨

Message Queue 동작-5

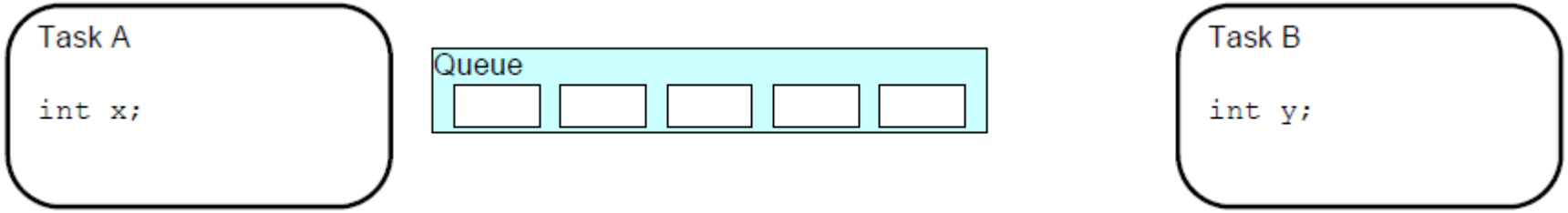
- 큐 write시에 블러킹(blocking)
  - 큐에서 데이터를 write하려고 할 때, timeout(blocking time)을 옵션으로 부가함
  - 이 timeout은 큐가 꽉 차 있을 때, 큐에 데이터를 write할 수 있을 때까지 Task를 Blocked State로 유지시키는 시간임
  - 큐에 데이터를 넣을 수 있기를 기다리는 Blocked State인 Task는 다른 Task 또는 ISR이 큐에서 데이터를 read(receive)하는 순간 자동으로 Ready State로 이동함
  - 큐에 넣을 수 있는 데이터 공간이 생기지 않더라도 지정한 timeout이 경과하면 자동으로 Blocked State에서 Ready State로 이동함
  - 동일한 큐에 대해 Multiple Writer Task가 존재하는 경우, 데이터 write를 위해 Blocked State로 되어 있는 Task가 하나 이상일 수 있음. 이때 데이터가 큐로부터 빠져나가면 하나의 Task만이 Blocked State에서 나와 Ready State로 이동함. 여러 개의 Blocked State인 Task에서 Ready State로 이동하는 Task는 우선순위 기준 또는 동일한 우선순위에서는 가장 오래 기다린 Task가 해당 됨

Message Queue APIs

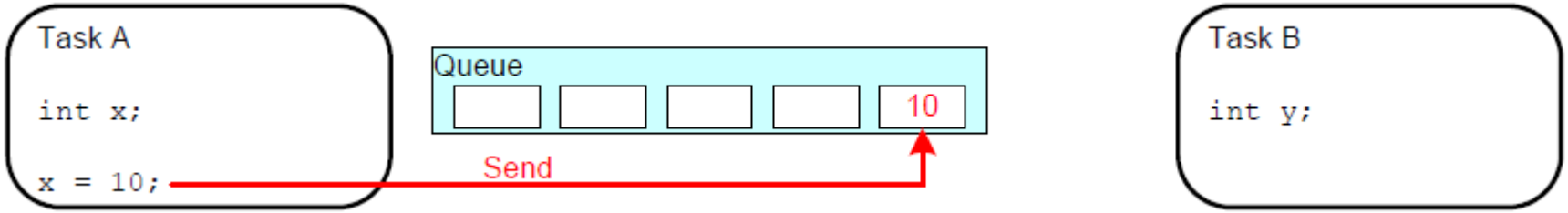
- 메시지 큐 생성 및 삭제
  - xQueueCreate, vQueueDelete
- 메시지 큐 전송 및 수신
  - xQueueSend, xQueueReceive
  - xQueueSendFromISR, xQueueReceiveFromISR
- 메시지 큐 제어
  - xQueueReset, xQueuePeek, xQueueOverwrite, uxQueueMessageWaiting, uxQueueSpaceAvailable
- 메시지 큐 ISR APIs
  - Message Queue generic API와 달리 호출한 Task를 Blocked State로 이동시키지 않기 위해 ISR에서 호출할 수 있는 함수가 별도로 존재함

Message Queue 동작 예

- Task A와 Task B가 통신할 수 있도록 큐가 생성한다.
  - 생성된 큐는 최대 5개의 정수를 저장할 수 있다.
  - 큐가 생성될 때는 어떠한 값도 없는 빈(empty) 상태이다.

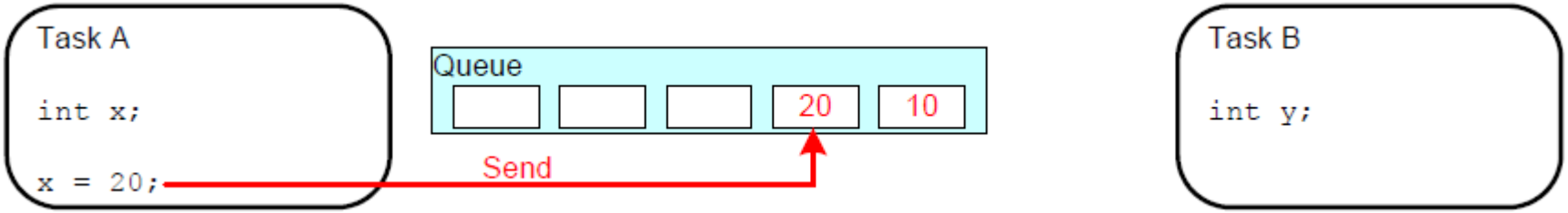


- Task A가 지역변수의 값(x=10)을 큐의 맨 뒤에 write(send) 한다.
  - 큐는 빈(empty) 상태에서 시작했기 때문에 쓰여진 값은 처음이자 마지막 값이 된다.

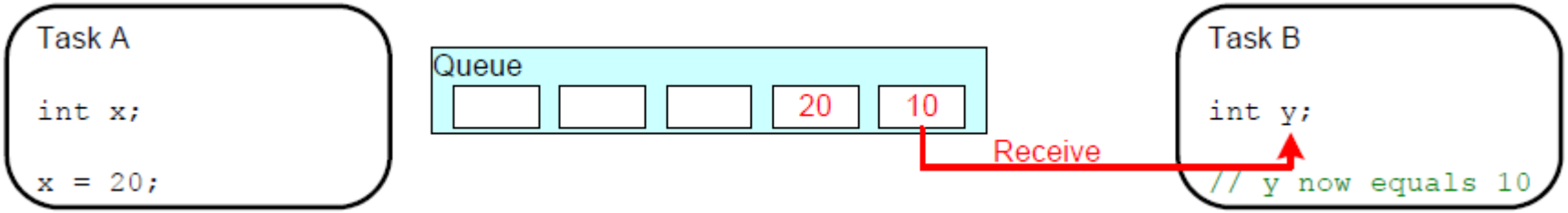


Message Queue 동작 예(계속)

- Task A는 지역변수의 값을 바꾸어(x=20), 큐에 다시 write한다.
  - 큐에는 복사된 두 개의 값(10과 20)이 포함된다.
  - 처음 쓰여진 값이 front가 되고, 나중에 쓰여진 값이 end가 된다.
  - 큐에는 세 개의 빈 공간이 남아 있다.

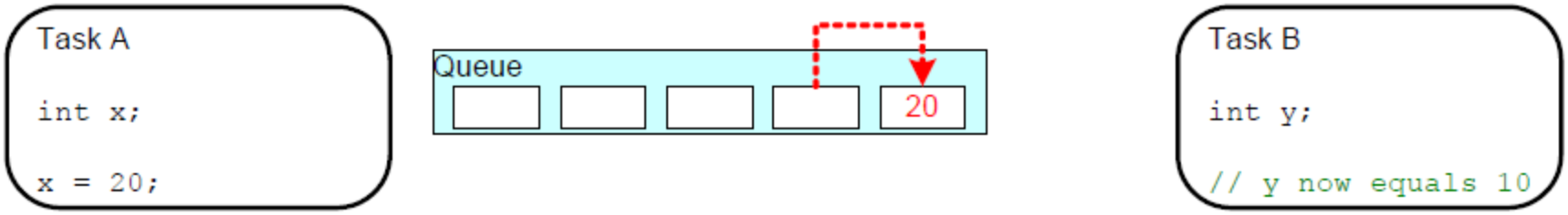


- Task B가 다른 변수로 큐에 있는 값을 read(receive)한다.
  - 이때 읽혀진 값은 큐의 head 값이다.
  - Task A가 처음 큐에 write했던 값 (그림에서는 10)



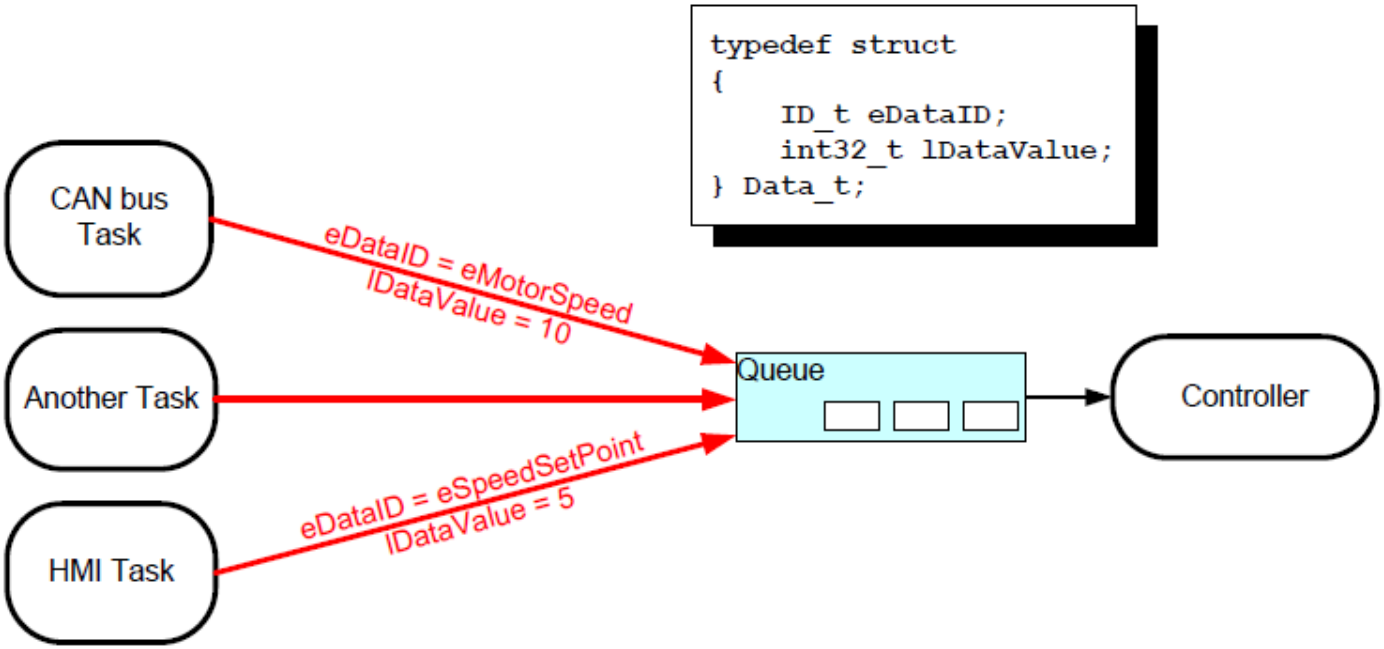
Message Queue 동작 예(계속)

- Task B가 item 하나를 지웠다(read/receive)
  - 큐에는 Task A가 두 번째 write했던 값이 남아 있다.
  - 이 값은 Task B가 다시 큐에서 read할 때 읽혀지는 값이다.
  - 큐에는 이제 네 개의 빈 공간이 남아 있다.



Message Queue Tip-1

- 여러 개의 Task로부터 데이터를 받을 때
  - 메시지 큐 설계 시 하나 이상의 Task로부터 데이터를 수신하는 경우가 일반적
  - 메시지 큐로부터 데이터를 읽어가는 Task는 데이터 송신자가 누구인지 확인 필요
  - 데이터와 Task ID를 가지는 구조체(structure)를 사용하는 것이 일반적



Message Queue Tip-2

- 사이즈가 크거나 또는 가변적 크기의 데이터 처리
  - 메시지 큐에 포인터를 전송(copy)하는 방식 사용
  - 사이즈가 큰 데이터를 큐로 전송할 때, 바이트 단위로 데이터를 복사하는 대신 포인터를 전송하는 것이 전송시간이나 메모리 사용면에서 보다 효율적임

- 메시지 큐에 포인터 전달 시 주의점
  - 지정된 메모리의 소유자가 누구인지 확실히 해야 함
    - 포인터를 통해 Task간 메모리를 공유할 때, 두 Task가 동시에 메모리를 수정할 수 없도록 해야 함
    - 이외에도 다른 Task나 동작에 의해 메모리 내용이 변경되거나 하지 않도록 해야 함
    - 큐에 포인터를 송신하는 Task는 큐에 메모리 포인터가 전달된 이후에 Access해야만 함
    - 큐로부터 포인터를 수신하는 Task는 큐에서 포인터를 전달 받은 이후에만 Access해야 함

Message Queue Tip-2 (계속)

- 포인터 전송에 사용되는 메모리는 유효해야 함
  - 동적 메모리 할당(dynamic memory allocation) 또는 미리 할당해 놓은 버퍼 메모리를 사용하는 경우, 한 개의 Task가 메모리를 free하도록 해야 하며, free된 메모리가 Task에 의해 Access되어서는 안됨
  - 포인터가 Task Stack에 할당된 데이터를 Access하지 않도록 해야 함. Stack 메모리는 frame이 변경될 수 있고, 이 때 데이터가 유효하지 않을 수 있기 때문 임

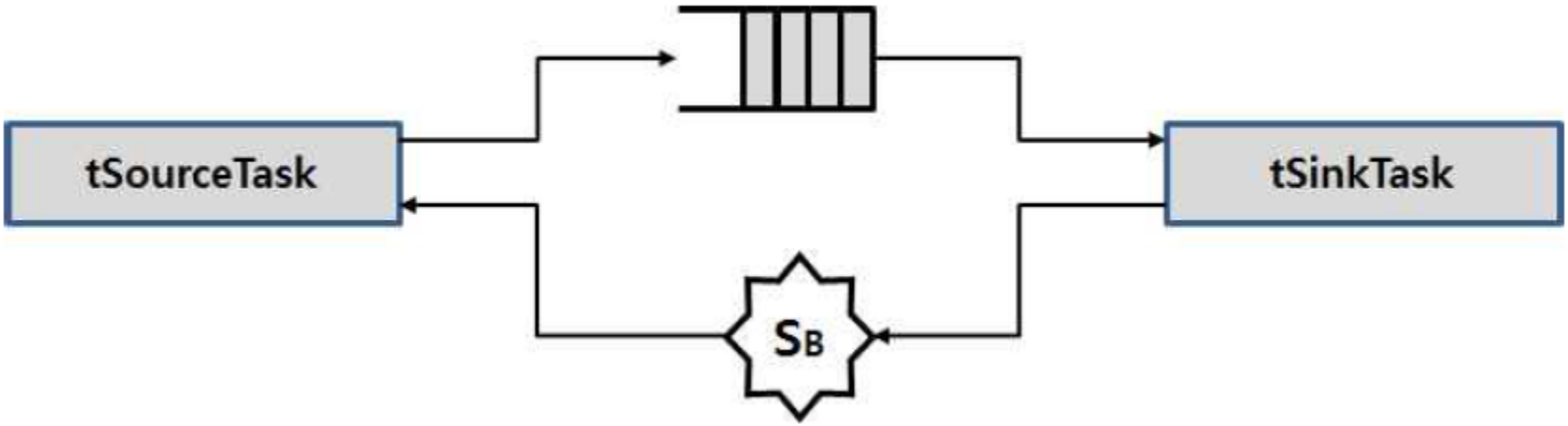
Message Queue Tip-3

- Single Task에서 여러 개의 큐로부터 데이터를 받을 필요가 있는 경우 Queue Set 사용 가능
  - Queue Set은 FreeRTOS에서만 제공하는 기능
  - Single Queue 사용시 보다 깔끔하거나 효율적이지 않기 때문에 제약사항 때문에 불필요하게 사용해야 하는 경우에만 사용하길 추천함

- Queue Set
  - 수신되는 데이터 크기도 다르고, 그 내용도 다르며, 전송되는 source도 다른 경우 사용
  - Integration하려는 Third party code가 전용 큐를 가지고 있는 경우 사용할 수 있음
  - Task가 하나 이상, 각각의 Queue에 데이터가 있는지 여부를 polling하지 않고, 데이터를 수신 가능

Message Queue Tip-4

- Interlock 방식의 단방향 데이터 통신
  - 송신 Task가 수신 Task로부터 데이터를 성공적으로 받았다는 확인응답(Acknowledgement)를 받아야 하는 경우 사용
  - 안정성이 요구되는 통신이나 Task 동기화 시 필요함



Semaphore의 정의

- 열쇠(Key) 또는 토큰(Token)의 역할을 하는 커널 오브젝트
  - 공유자원(Shared Resource)에 대한 배타적 접근(Mutual Exclusion)
  - 태스크 동기화(Task Synchronization)
  - 세마포어 획득(Acquire/Get)을 통해 동작이나 접근에 대한 ownership 획득
  - 세마포어 반환(Release/Put)을 통해 동작이나 접근에 대한 ownership 반납
  - 세마포어 획득 또는 반환할 수 있는 횟수가 1회인 경우 Binary semaphore
  - 세마포어 획득 또는 반환할 수 있는 횟수가 1회 이상인 경우 Counting semaphore

Semaphore의 구성

- Binary Semaphore
  - 세마포어 초기 값으로 '1' 또는 '0'을 가짐
    - 초기 값 '1'인 경우는 공유 자원 Access시 사용
    - 초기 값 '0'인 경우는 태스크 동기화시 사용
  - 세마포어를 획득(Acquire)하면 이 값은 '0'이 됨(empty/available)
  - 세마포어를 반납(Release)하면 이 값은 '1'이 됨(full/unavailable)

- Counting Semaphore
  - 여러 차례 획득 및 반환이 가능
  - 세마포어 생성시 세마포어의 초기 토큰 개수 지정
  - 태스크는 토큰이 '0'이 될 때까지 계속 토큰 획득이 가능
  - 세마포어 토큰 카운팅이 '0'이 되면 사용 불능 상태로 변경
  - 다시 사용 가능 상태가 되려면 어떤 태스크가 세마포어에 토큰을 반환해야만 함
  - 토큰 반환시 1씩 증가

Semaphore 동작

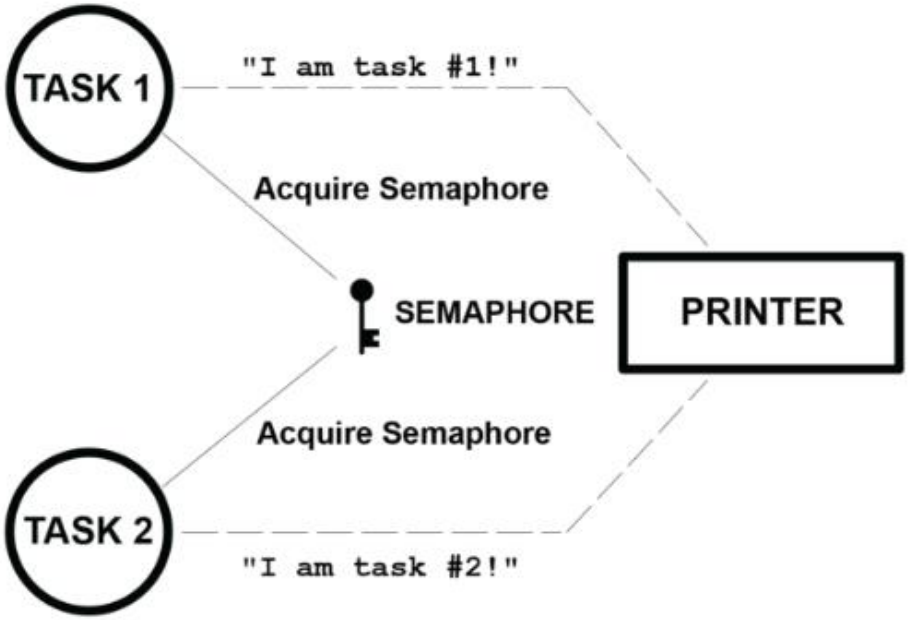
- Semaphore Taking시 블러킹(blocking)
  - 세마포어를 획득하려고 할 때, timeout(blocking time)을 옵션으로 부가함
  - 이 timeout은 세마포어가 없을 때(카운트 값 '0'), 세마포어를 획득할 수 있을 때까지 Task를 Blocked State로 유지시키는 시간임
  - 세마포어를 획득할 수 있기를 기다리는 Blocked State인 Task는 다른 Task 또는 ISR이 세마포어를 반납(give)하는 순간 자동으로 Ready State로 이동함
  - 세마포어를 획득할 수 없더라도 지정한 timeout이 경과하면 자동으로 Blocked State에서 Ready State로 이동함
  - 동일한 세마포어에 대해 Multiple Task가 해당 세마포어를 획득하려고 존재하는 경우, 세마포어를 획득하기 위해 Blocked State로 되어 있는 Task가 하나 이상일 수 있음. 이때 다른 태스크가 세마포어를 반납하면 하나의 Task만이 Blocked State에서 나와 Ready State로 이동함. 여러 개의 Blocked State인 Task에서 Ready State로 이동하는 Task는 우선순위 기준 또는 동일한 우선순위에서는 가장 오래 기다린 Task가 해당 됨

Semaphore APIs

- 세마포어 생성 및 삭제
  - xSemaphoreCreateBinary, xSemaphoreCreateCounting,
  - vSemaphoreDelete
- 세마포어 전송 및 수신
  - xSemaphoreTake, xSemaphoreGive
  - xSemaphoreTakeFromISR, xSemaphoreGiveFromISR
- 세마포어 제어
  - xSemaphoreGetCount
- 세마포어 ISR APIs
  - 세마포어 generic API와 달리 호출한 Task를 Blocked State로 이동시키지 않기 위해 ISR에서 호출할 수 있는 함수가 별도로 존재함

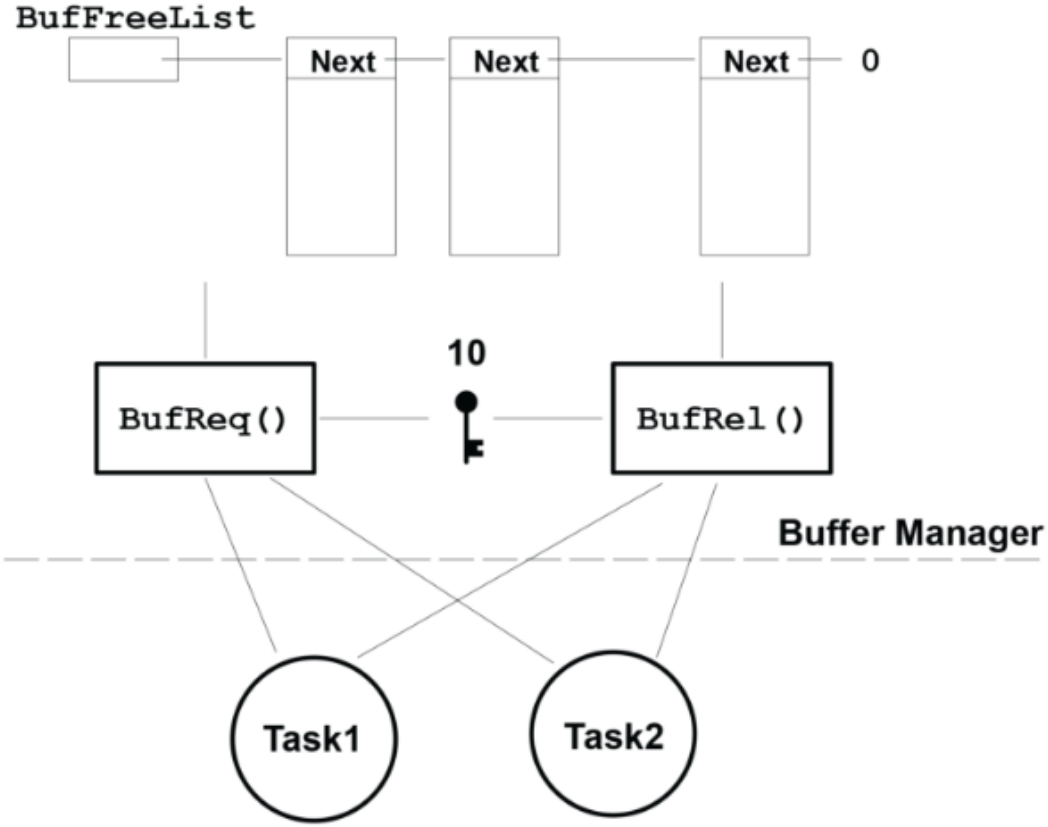
Semaphore 동작 예-1

- 공유 자원(Shared Resource) Access
  - 태스크들이 I/O 자원을 서로 공유할 때 유용함
  - 자원을 Access하려는 태스크가 먼저 세마포어를 획득
  - 이때는 배타적 접근을 알리기 위해 열쇠 표시를 사용
  - 아래는 하나의 프린터 접근을 위해 Binary Semaphore 사용



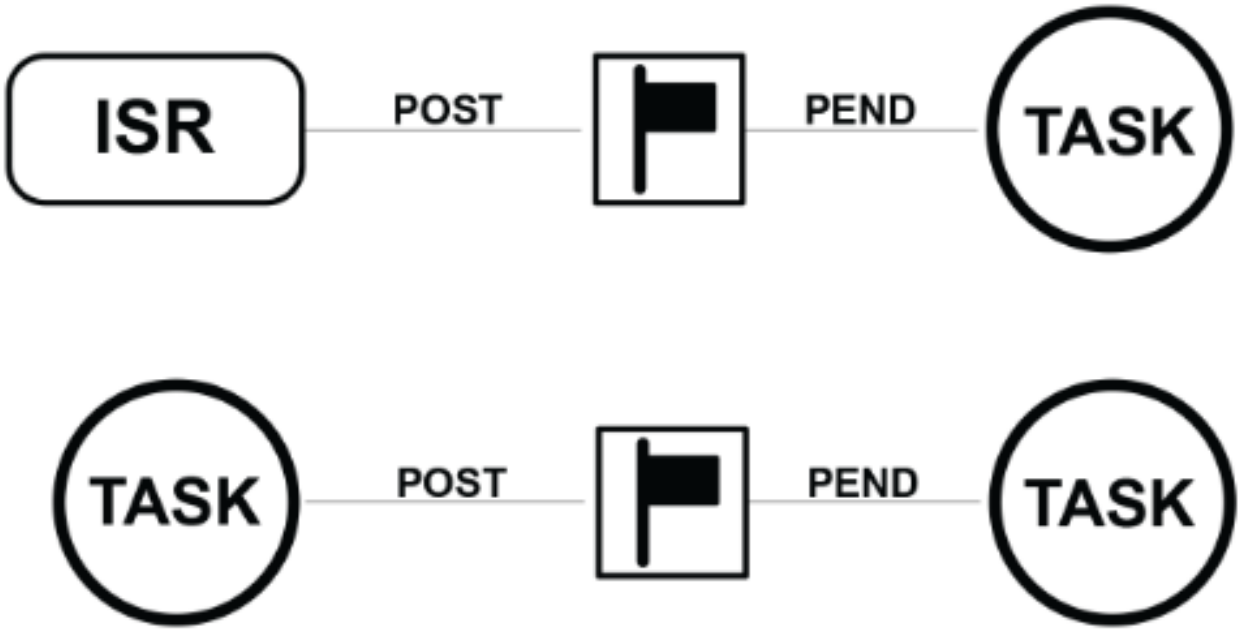
Semaphore 동작 예-2

- 공유 자원(Shared Resource) Access
  - 아래는 버퍼 메모리 풀(Buffer Memory Pool)접근을 위해 Counting(10) Semaphore 사용



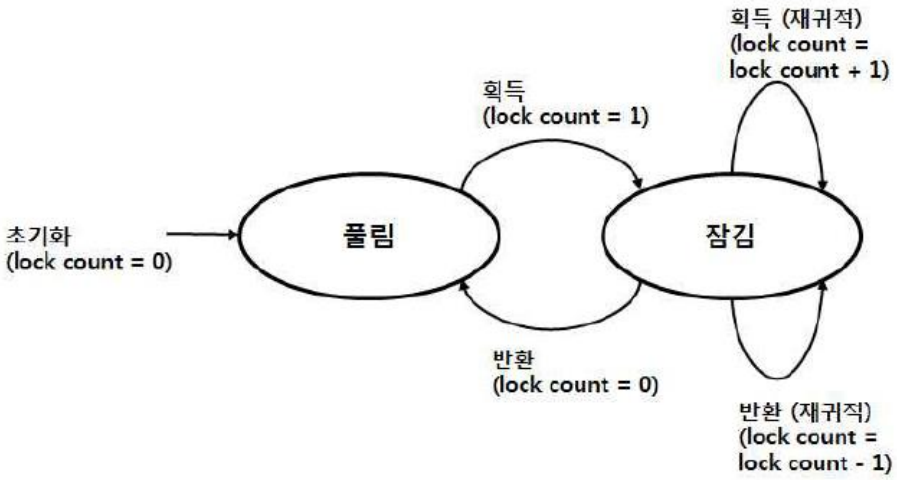
Semaphore 동작 예-3

- 태스크 동기화(Task Synchronization)
  - 태스크는 ISR 또는 다른 태스크와 같이 세마포어를 사용해 동기화될 수 있음
  - 이때는 이벤트(event) 발생을 알리기 위해 깃발 표시를 사용



MUTEX의 정의

- 두 개 이상의 태스크간에 공유되는 자원을 Access하는 용도의 특별한 바이너리 세마포어
  - 상호 배제 세마포어(**MUT**ual **Ex**clusion)
  - 소유권(Ownership)
  - 재귀적 접근(Recursive Access)
  - 안전한 태스크 삭제(Task Deletion)
  - 우선순위 역전 회피 프로토콜 지원
  - 잠김(Lock), 풀림(Unlock) 지원



MUTEX의 구성

- 뮤텍스 소유권(Ownership)
  - 태스크가 뮤텍스를 획득할 경우 뮤텍스의 소유권을 얻는다
  - 반면, 태스크가 뮤텍스를 반환할 경우 뮤텍스의 소유권을 잃는다
  - 특정 태스크가 뮤텍스를 획득하는 경우, 다른 태스크들은 그 뮤텍스를 풀거나 잠글 수 없다
  - 단순한 바이너리 세마포어의 경우 세마포어를 획득한 태스크가 아니라도 세마포어를 반환할 수 있다

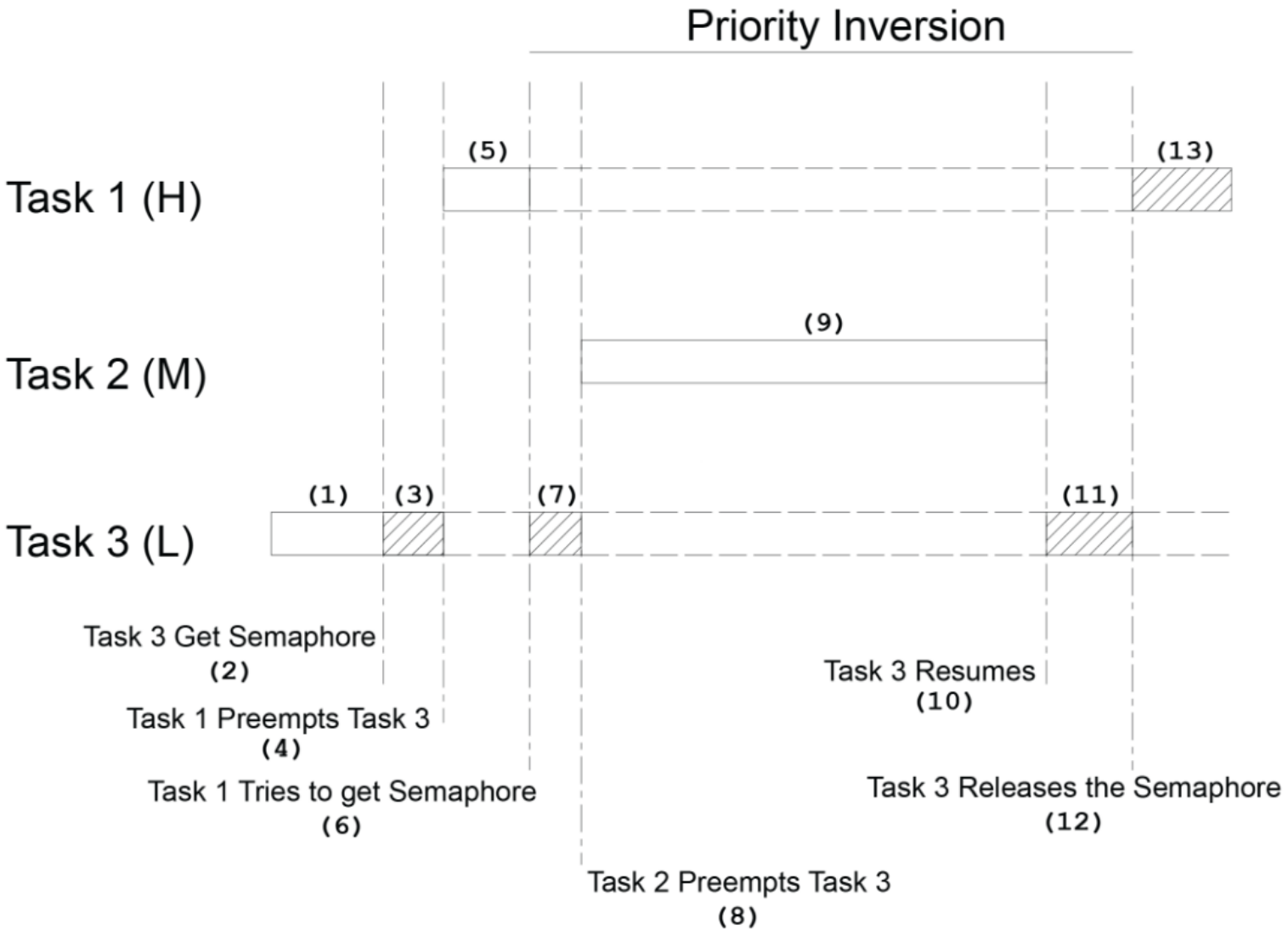
- 재귀적 잠금(Recursive Locking)
  - 뮤텍스를 획득한 태스크가 잠긴 상태의 뮤텍스를 몇 번이든 획득할 수 있음
  - 공유자원에 독점적으로 접근하고자 하는 태스크가 그 공유자원에 접근하는 루틴을 여러 차례 호출하는 경우 유용

MUTEX의 구성(계속)

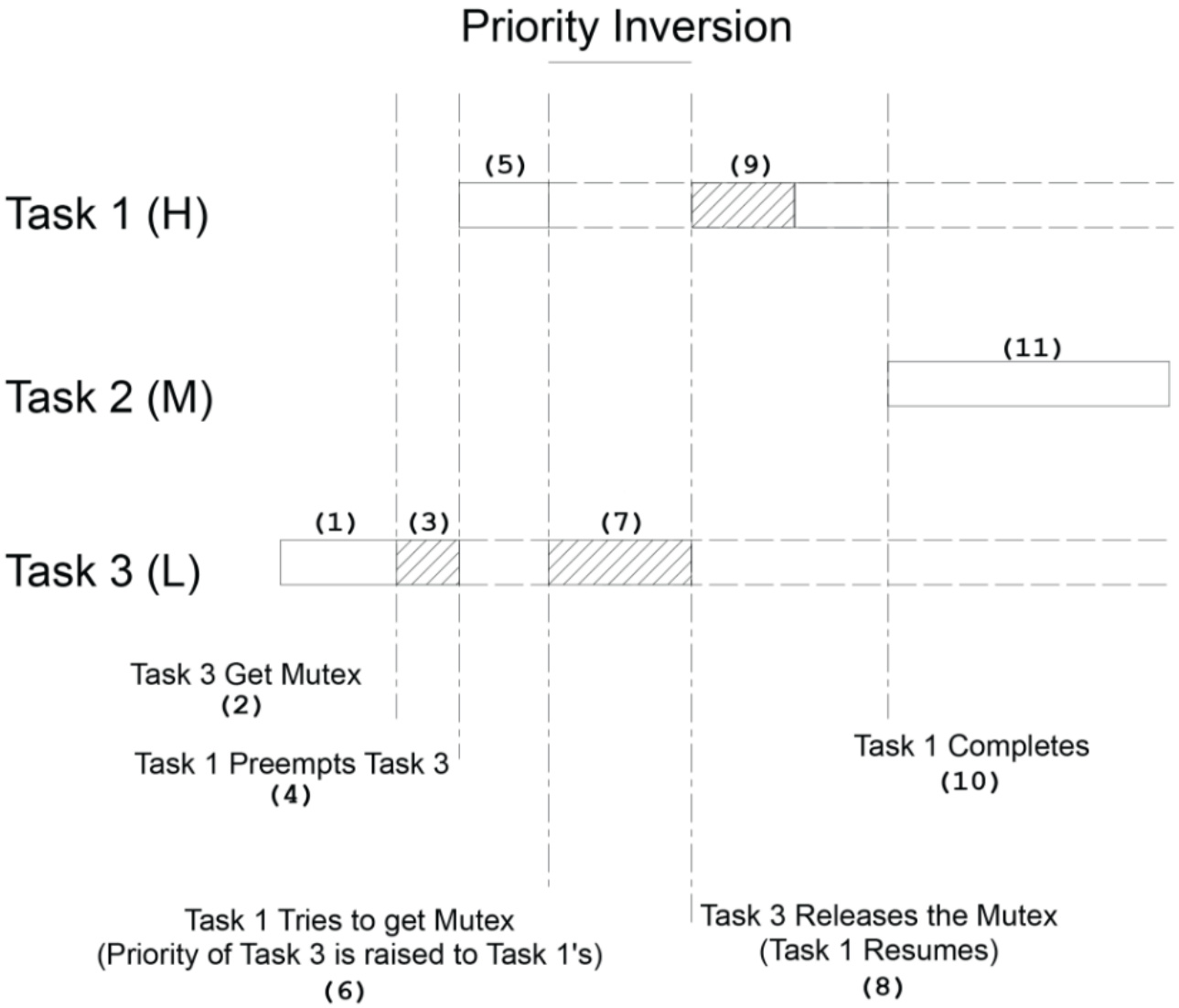
- 안전한 태스크 삭제(Task Deletion Safety)
  - 태스크 삭제 방지 기능이 활성화 되어 뮤텍스를 소유한 태스크는 다른 태스크에 의해 삭제되지 않음(실수 방지)

- **우선순위 역전 회피(Priority Inversion Avoidance)**
  - RTOS application 설계오류에 의해 발생
  - 우선순위가 낮은 태스크가 공유자원을 점유하고 있을 때, 우선순위가 높은 태스크가 동일한 공유자원에 접근하려 할 때 Blocked State가 되고, 이 때 중간 우선순위의 태스크가 낮은 우선순위의 태스크를 선점하여 프로세서를 점유하는 현상
    - 우선순위 상속(Priority Inheritance)을 통해 문제 해결

Priority Inversion



Priority Inheritance



MUTEX 동작

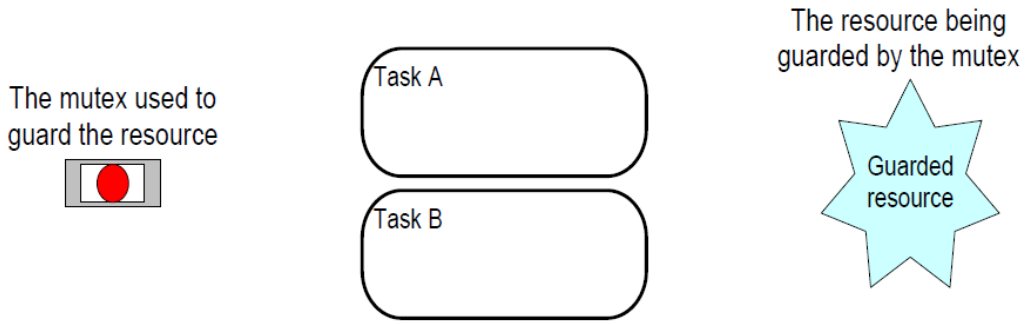
- Mutex Taking시 블러킹(blocking)
  - 세마포어와 마찬가지로 timeout(blocking time)을 옵션으로 부가함
  - 이 timeout은 뮤텍스가 비어있을 때(empty), 뮤텍스를 획득할 수 있을 때까지 Task를 Blocked State로 유지시키는 시간임
  - 세마포어와 달리 Priority Inheritance 지원
    - Priority Inversion 문제를 보완하기 위한 보조적 기능
  - 뮤텍스를 획득할 수 있기를 기다리는 Blocked State인 Task는 다른 Task가 뮤텍스를 반납(give)하는 순간 자동으로 Ready State로 이동함
  - 뮤텍스를 획득할 수 없더라도 지정한 timeout이 경과하면 자동으로 Blocked State에서 Ready State로 이동함
  - 동일한 뮤텍스에 대해 Multiple Task가 해당 뮤텍스를 획득하려고 존재하는 경우, 뮤텍스를 획득하기 위해 Blocked State로 되어 있는 Task가 하나 이상일 수 있음. 이때 다른 태스크가 뮤텍스를 반납하면 하나의 Task만이 Blocked State에서 나와 Ready State로 이동함. 여러 개의 Blocked State인 Task에서 Ready State로 이동하는 Task는 우선순위 기준 또는 동일한 우선순위에서는 가장 오래 기다린 Task가 해당 됨

MUTEX APIs

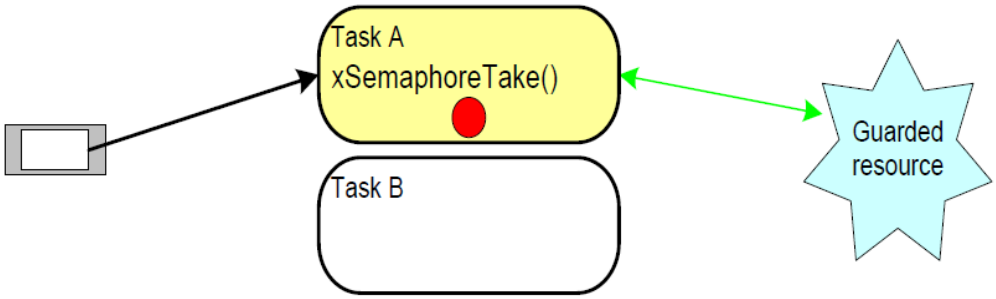
- MUTEX 생성 및 삭제
  - xSemaphoreCreateMutex, xSemaphoreCreateRecursiveMutex
  - vSemaphoreDelete
- MUTEX 전송 및 수신
  - xSemaphoreTake, xSemaphoreGive
  - xSemaphoreTakeRecursive, xSemaphoreGiveRecursive
- MUTEX 제어
  - xSemaphoreGetMutexHolder
- MUTEX ISR APIs
  - 인터럽트는 태스크 기반하의 priority inheritance에 해당하지 않기 때문에 ISR에서 사용되지 않음

MUTEX 동작 예

- Task A와 Task B가 각각 자원(Resource)를 access하려 한다.
  - 자원은 이미 뮤텍스에 의해 보호되고 있다.
  - 뮤텍스를 가진 태스크가 아닌 태스크는 자원에 access할 수 없다.

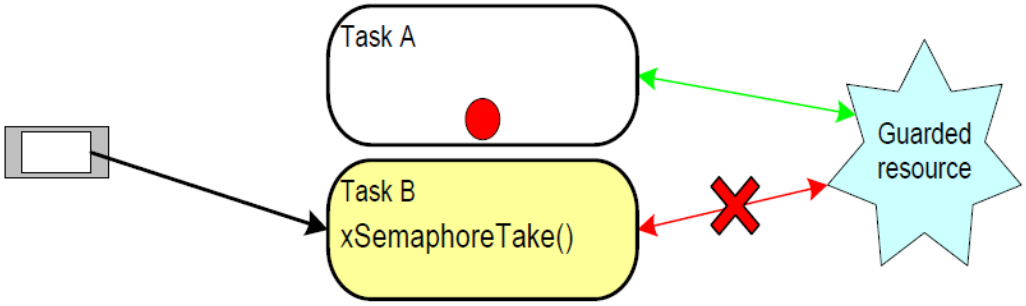


- Task A가 뮤텍스를 가져오려고 시도한다.
  - 뮤텍스는 사용가능한 상태이기 때문에, Task A가 성공적으로 뮤텍스를 가져오게 되며, 자원을 access할 수 있게 된다.

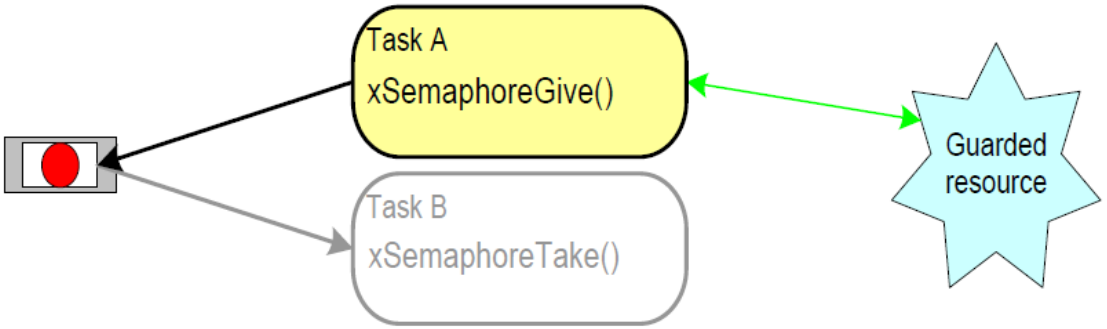


MUTEX 동작 예(계속)

- Task B가 동일한 뮤텍스를 가져오려고 시도한다.
  - Task A가 이미 동일한 뮤텍스를 가지고 있기 때문에 시도는 실패한다.
  - Task B는 보호된 자원을 access할 수 없다.

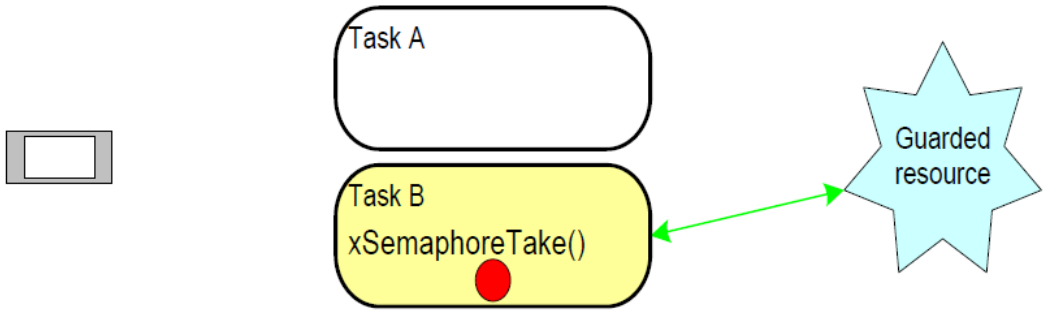


- Task B가 뮤텍스를 기다리는 Blocked state로 들어간다.
  - Task A가 다시 실행하면서, 자원에 대한 처리를 끝내고 뮤텍스를 다시 반납한다.

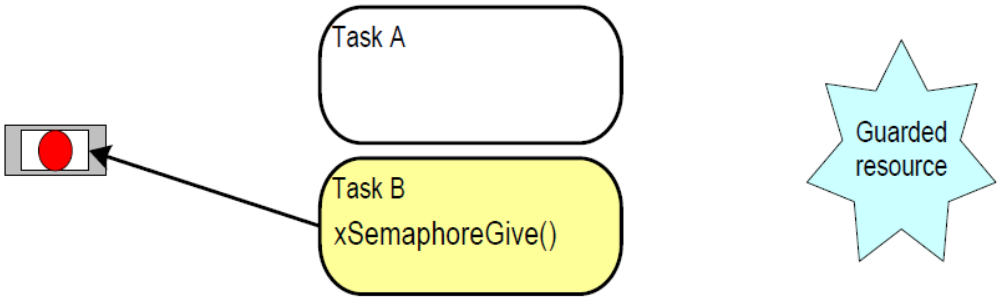


MUTEX 동작 예(계속)

- Task A가 뮤텍스를 반납하면서 Task B가 Blocked state에서 나온다.
  - Task A의 반납으로 뮤텍스는 사용 가능한 상태가 된다.
  - Task B가 뮤텍스를 성공적으로 얻어 자원을 access할 수 있게 된다.



- Task B가 자원처리를 끝내고 뮤텍스를 반납한다.
  - 뮤텍스는 두 개의 태스크에 의해 다시 사용 가능한 상태가 된다.



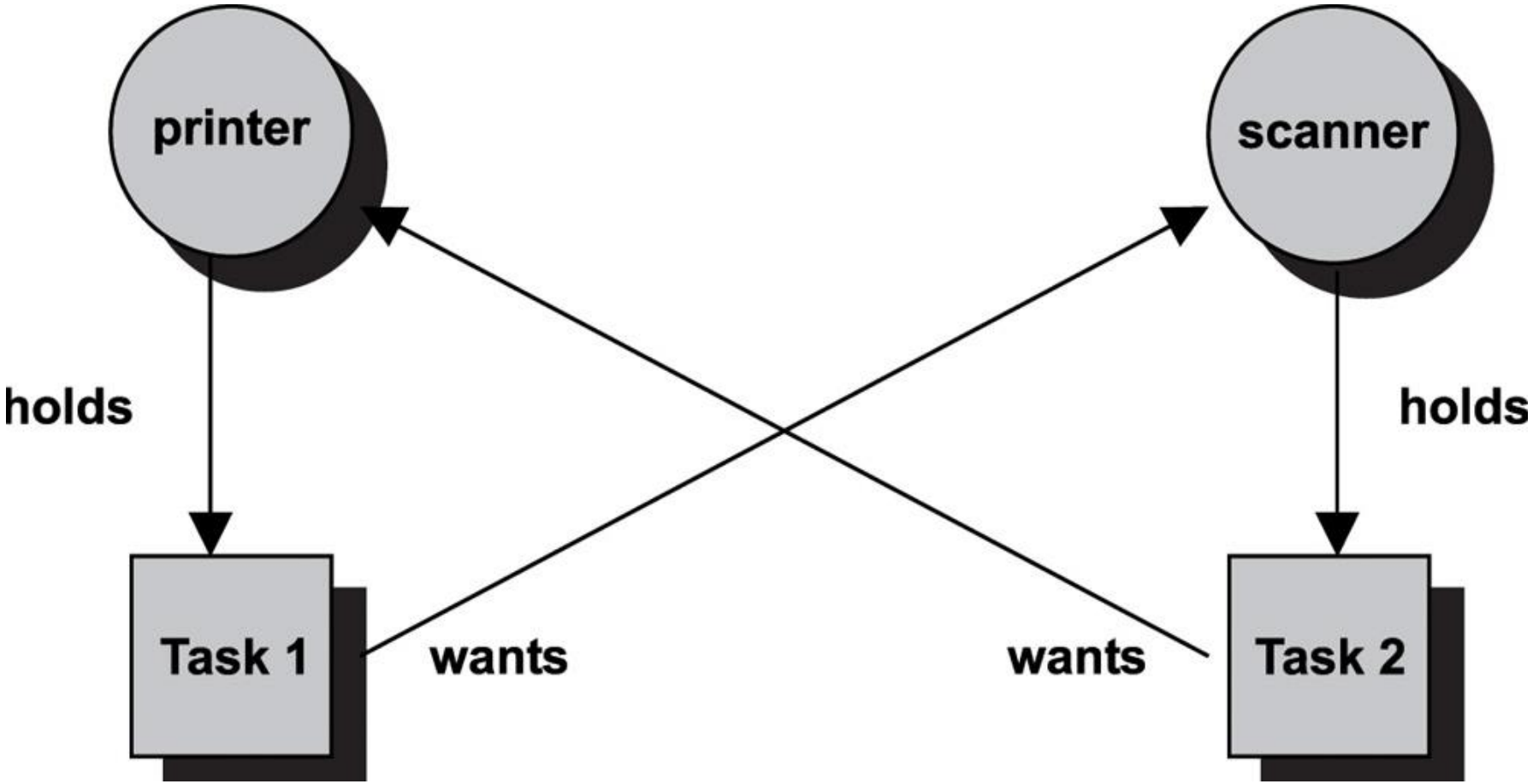
MUTEX Issue

- Deadlock(or Deadly Embrace)
  - 두 개의 태스크가 다른 태스크에 의해 점유되어 있는 자원을 사용할 수 있을 때까지 기다리면서 발생하는 태스크 실행 중단

- 다음과 같은 시나리오가 가능하다.
  1. Task A가 실행하고 Mutex X를 성공적으로 가져온다.
  2. Task A는 Task B에 의해 선점(pre-emped) 당한다.
  3. Task B는 Mutex Y를 성공적으로 가져온다.
     - 이후 Task B는 Mutex X를 가져오려 하지만, Task A가 가지고 있기 때문에 Task B는 Blocked state로 이동한다.
  4. Task A가 실행을 계속하고 Mutex Y를 가져오려 시도한다.
     - Mutex Y는 Task B가 가지고 있기 때문에, Task A는 Blocked state로 이동한다.

  - Task A는 Task B가 가진 Mutex를 기다리고, Task B는 Task A가 가진 Mutex를 기다린다. 이로 인해 두 개의 태스크 모두 실행할 수 없는 deadlock이 발생된다.

MUTEX Issue (계속)



MUTEX Issue (계속)

- Deadlock(or Deadly Embrace) 해결
  - 설계 시에 잠재적인 데드락 문제를 고려하여 데드락이 발생하지 않도록 설계할 필요가 있음
  - 뮤텍스를 무한히 기다리게 하거나 하는 등의 조치(timeout이 무한대)는 신중해야 함
  - 실제적으로 작은 임베디드 시스템에서는 설계자가 모든 동작을 예견하고 확정하고 있기 때문에 발생율이 적음
  - 태스크 실행이 상당히 많은 어플리케이션에서 발생할 가능성이 높음

Event Processing

- Event Bits (Event Flags)
  - 이벤트(event)가 발생했는지 여부를 나타낼 때 사용
  - Event flag라고도 함

- Bit definition-1
  - 비트(bit) 또는 플래그(flag)가 '1'이면 메시지가 수신되어 처리 준비가 되었음을 나타냄, '0'이면 메시지가 수신되지 않았고 처리 준비가 되어 있지 않음을 나타냄
- Bit definition-2
  - 비트(bit) 또는 플래그(flag)가 '1'이면 버퍼 메시지를 네트워크상에 전송할 준비가 되었음을 나타냄, '0'이면 네트워크 상에 송신할 버퍼 메시지가 없음을 나타냄
- Bit definition-3
  - 비트(bit) 또는 플래그(flag)가 '1'이면 heartbeat 메시지를 네트워크상에 전송할 시간이 되었음을 나타냄, '0'이면 heartbeat 메시지를 네트워크상에 전송할 시간이 아직 안되었음을 나타냄

Event Processing (계속)

- Event Groups
  - Event bit들의 집합체(set)
  - 이벤트 그룹 내에 있는 각각의 이벤트 비트들은 비트 번호로 참조

- Event bit-1
  - 이벤트 그룹 내에 있는 비트 번호 0번이 "메시지가 수신되어 처리 준비가 되었음"을 나타낼 수 있다.

- Event bit-2
  - 동일한 이벤트 그룹 내에 있는 비트 번호 1번이 "버퍼 메시지가 네트워크상에 전송될 준비가 되었음"을 나타낼 수 있다.

- Event bit-3
  - 동일한 이벤트 그룹 내에 있는 비트 번호 2번이 "heartbeat 메시지를 네트워크상에 전송할 시간이 되었음"을 나타낼 수 있다.

Event 동작 예

- 24 비트 이벤트 그룹-예
  - 앞서 설명한 세 개의 이벤트 예를 아래와 같이 표현함
  - 아래 그림에서는 이벤트 비트 2번만이 설정(set) 되었음

- Event Group APIs
  - 이벤트 그룹 내에 있는 하나 이상의 비트를 set하고 clear하는 API 존재
  - 이벤트 그룹 내에서 하나 이상의 이벤트 비트의 집합이 set되기를 기다리는 pend API 존재

Event Group APIs

- Event Group 생성 및 삭제
  - xEventGroupCreate, vSemaphoreDelete
- Event Group pendind
  - xEventGroupWaitBits
- Event Group 제어
  - xEventGroupSetBits, xEventGroupClearBits, xEventGroupGetBits
- Event Group ISR APIs
  - Event Group generic API와 달리 호출한 Task를 Blocked State로 이동시키지 않기 위해 ISR에서 호출할 수 있는 함수가 별도로 존재함

Interrupt Management

- Task
  - 하드웨어와 관계없는 소프트웨어 기능
  - 우선순위는 소프트웨어에 의해 부여됨
  - 어떤 태스크가 Running state로 있게 될지 여부는 소프트웨어 알고리즘(scheduler)이 결정
  - 태스크는 아래 ISR이 실행되지 않을 때만 실행 가능

- ISR(Interrupt Service Routine)
  - ISR 역시 소프트웨어로 작성
  - 언제 어떤 ISR이 실행될 지 여부가 하드웨어로 제어되는 하드웨어 기능
  - 최하의 우선순위를 가지는 ISR이 최고의 우선순위를 가지는 태스크의 실행을 선점할 수 있음
  - 태스크가 ISR을 선점할 방법은 없음

Interrupt safe API

- ISR은 태스크와 같은 소프트웨어 알고리즘에 의한 스케줄링 대상이 아님

- 태스크와 같이 Blocked state가 되어서는 안됨
  - FreeRTOS의 경우, ISR에서 호출할 수 있는 API들이 별도로 존재
  - 동일한 이름에 접미어로 "FromISR"이 붙음
  - Interrupt Safe API라고 호칭
  - Interrupt Safe API에 xHigherPriorityTaskWoken parameter 사용

- ISR 실행 후 interrupt return시 별도의 ISR return함수 호출
  - FreeRTOS의 경우, 다음의 macro API 호출
  - portYIELD_FROM_ISR()
    - xHigherPriorityTaskWoken의 값에 따라 context switching이 발생

Interrupt 처리

- ISR은 다음과 같은 이유로 가능한 짧게 작성되어 빠르게 처리되어야 한다.
  - 가장 높은 우선순위의 태스크라 해도 인터럽트가 실행하지 않을 때만 실행할 수 있음
  - 태스크의 실행시간, 시작시간 모두에 방해가 될 수 있음(jitter)
  - RTOS가 실행되는 프로세서에 따라 다르지만, ISR 실행 중 다른 새로운 인터럽트를 처리할 수 없을 수 있음
  - 변수, 주변장치, 메모리 버퍼와 같은 자원을 태스크와 ISR이 동시에 access되어지는 상황에 대한 보증 및 결과에 대한 고려 필요
  - RTOS가 실행되는 프로세서에 따라 다르지만, 인터럽트 중첩(nesting)이 지원되는 경우, 복잡성이 증가하고 예측성이 떨어짐
  - ISR이 짧으면 조금 더 나음

Deferred Interrupt 처리

- ISR 처리 속도를 높이기 위한 방법
  - 인터럽트 발생시 ISR에서는 인터럽트 발생 원인 저장
  - 인터럽트 clear
  - 인터럽트를 처리하는 Task 기동(triggering)

- 특징
  - 프로그래머가 처리 우선순위를 부여할 수 있음
  - ISR 처리 태스크의 우선순위가 가장 높다면, ISR exit후 바로 실행 가능
  - ISR전용 커널 오브젝트 이외에 모든 커널 오브젝트 사용 가능

- 유용한 예
  - ISR이 ADC(Analog to Digital Converter)의 결과 저장시
    - ADC Data에 대한 S/W filter 적용시 유용
  - ISR 처리 시간이 얼마나 걸릴지 모르는 경우 (Non-deterministic)
    - H/W 버퍼가 없거나 부족한 경우 console data string 출력(TXD)의 경우
    - Memory allocation이 필요한 경우(non-deterministic)

Memory Management

- 동적 메모리 할당(Dynamic Memory Allocation)
- Standard C API에 malloc(), free() 존재
  - 메모리 조각(fragmentation)이 발생하며 이로 인해 인접된 단일 메모리 영역을 얻을 수 없음
  - Malloc() 함수에 의해 요청한 크기의 메모리 크기를 충족시키기 위해 인접한 free 메모리 블럭들을 정렬 시키기 위해 사용되는 알고리즘의 실행 시간이 불규칙적임

  → non-deterministic

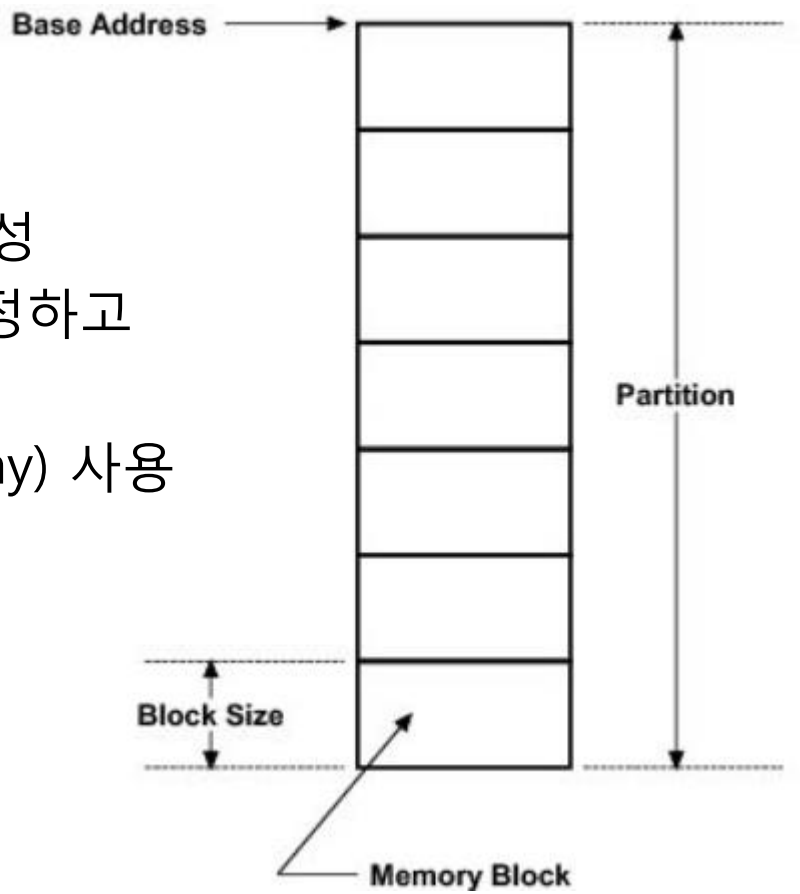
  → Embedded RTOS application에 적합하지 않음

- Embedded RTOS application에 필요한 동적 메모리 관리 요구사항
  - 메모리 조각의 최소화(Minimize fragmentation)
  - 최소한의 관리 오버헤드(Minimize management overhead)
  - 일정한 메모리 할당 시간(Consistent time of memory allocation)

→ Standard C API에서 제공하는 malloc(), free() 함수 대신 RTOS 개별적으로 동적 메모리 할당을 위한 API들을 제공함

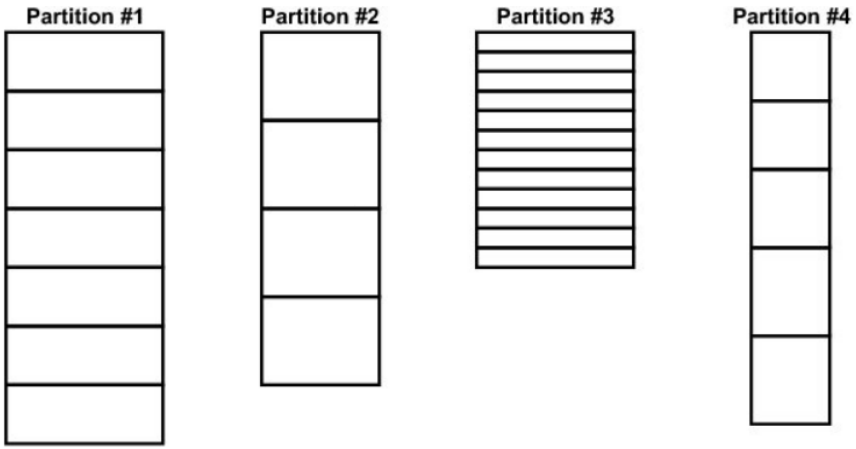
Memory Partition 방식

- Block Size 동일
- Partition은 여러 개의 Block들로 구성
- Allocation, Deallocation 시간이 일정하고 deterministic함
- Partition 자체는 static memory(array) 사용
- 일반적으로 많이 사용하는 방식



Multiple Memory Partition

- Application에 보통 하나 이상의 memory partition이 존재
- 각각의 partition은 다른 memory block 크기와 개수를 가짐
- Application은 요청에 따라 각기 다른 크기의 memory block을 얻을 수 있음
- Memory Free시에는 원래의 partition 위치로 return됨
- Memory fragmentation이 발생하지 않음
- Memory block의 크기와 개수, Memory partition을 얼마나 만들 것인지 여부는 application에서 결정함



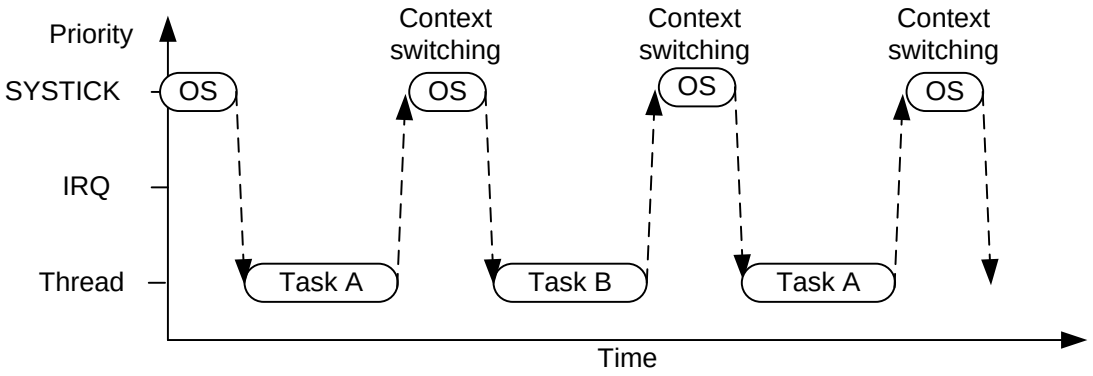
FreeRTOS Memory Management

- Embedded System은 System별로 다양한 메모리 크기와 프로세서 성능을 가짐
  - 이식(porting)이 용이하도록 FreeRTOS Standard API 존재
    → **pvPortMalloc(), vPortFree()**
  - 다섯 가지의 Dynamic Memory Allocation 알고리즘 제공
    → 다섯 개의 개별 파일들이 존재
    → 시스템 성능에 따라 선택 사용 가능

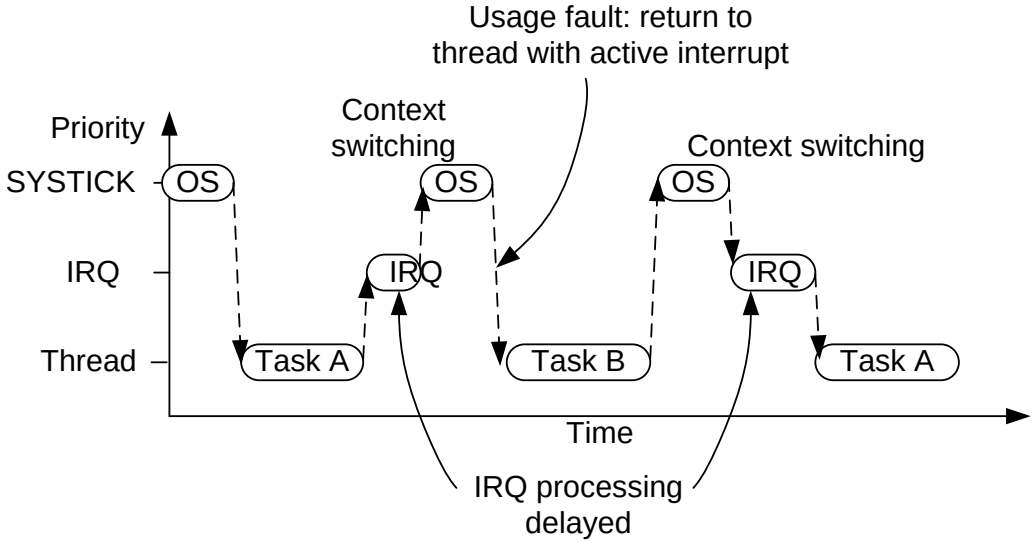
- FreeRTOS 다섯 가지 Heap management method
  1. heap_1 : 가장 간단하며, memory를 free를 허용하지 않음
  2. heap_2 : best fit 알고리즘을 사용하여 memory를 free를 허용하지만, 인접한 메모리 조각을 합체하지는 않음
  3. heap_3 : C 라이브러리 함수인 malloc(), free()함수를 wrapping시킴
  4. heap_4 : heap_2와 같이 best fit 알고리즘을 사용하여 memory를 free를 허용하며, 인접한 메모리 조각을 합체하는 알고리즘이 존재
  5. heap_5 : heap_4와 동일하며, 인접하지 않은 여러 개의 메모리 영역으로 heap영역이 걸쳐지도록 함

FreeRTOS porting

- 일반적인 OS 스케줄링



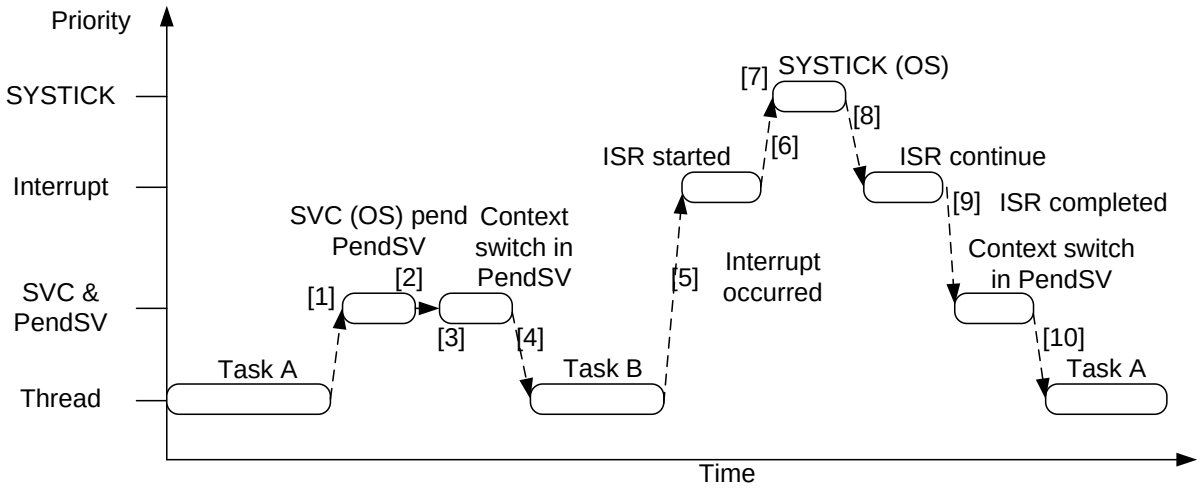
- 일반적인 OS 스케줄링을 Cortex-M Family에 적용할 경우



FreeRTOS porting

- 일반적인 OS 스케줄링을 Cortex-M Family에 적용할 경우
  - 인터럽트 실행 도중 보다 높은 우선순위의 인터럽트(여기서의 예는 systick timer intr.)가 발생해 실행
  - 이때 이곳에서 태스크 스위칭이 일어나는 경우
  - 처음에 선점된 인터럽트는 다른 태스크 실행이 끝나고 실행될 수 있음
    - 본의 아니게 지연되고, 그에 딸린 태스크(Task A)도 함께 실행이 지연된다.
    - 실시간성이 보장되지 않는다.
  - 이를 위해 Cortex-M Family는 PendSV 익셉션을 지원

- PendSV 익셉션 적용시



FreeRTOS porting

- PendSV 익셉션 적용하여 OS 스케줄링 실행시
  - 태스크 스위칭을 직접 호출하지 않음
  - 익셉션(인터럽트) 우선순위가 가장 낮게 설정된 PendSV에서 실행
  - 현재 실행중인 인터럽트가 종료된 후 선점된 원래의 인터럽트가 실행을 계속 재개할 수 있다.
  - 모든 인터럽트 실행이 종료된 다음 우선순위가 가장 낮은 PendSV가 실행
  - 이 때 태스크 스위칭이 일어나 인터럽트 지연과 같은 문제를 예방할 수 있다.

  - FreeRTOS는 SysTick과 PendSV가 모두 최하위 익셉션 우선순위를 가진다.
  - 참고로 두 익셉션의 우선순위는 동일하다.
    - Port.c, portasm.s 파일 참조

FreeRTOS porting

- 대략 20개의 다른 컴파일러로 빌드 가능
- 30개 이상의 프로세서에서 실행 가능
- FreeRTOS는 라이브러리 개념으로 소스파일로 제공됨
  - 포팅 대상용 소스파일 존재
  - 포팅 대상이 아닌 공통의 소스 파일 존재
  - 어플리케이션에서 FreeRTOS의 소스파일들을 추가하여 빌드해야 함
  - 배포판에 존재하는 데모 어플리케이션용 프로젝트 활용

- FreeRTOSConfig.h
  - FreeRTOS 설정 파일
    - configUSE_PREEMPTION의 경우, 선점형 스케줄링 사용 여부를 나타냄
  - 어플리케이션에 특화된 정의들도 포함
    - FreeRTOS 소스코드 디렉토리가 아닌 빌드 되는 어플리케이션 디렉토리에 위치 되어야 함
  - 배포판에 존재하는 데모 어플리케이션용 프로젝트에 있는 FreeRTOSConfig.h 파일을 사용

FreeRTOS porting

- FreeRTOS 배포판의 디렉토리 구조

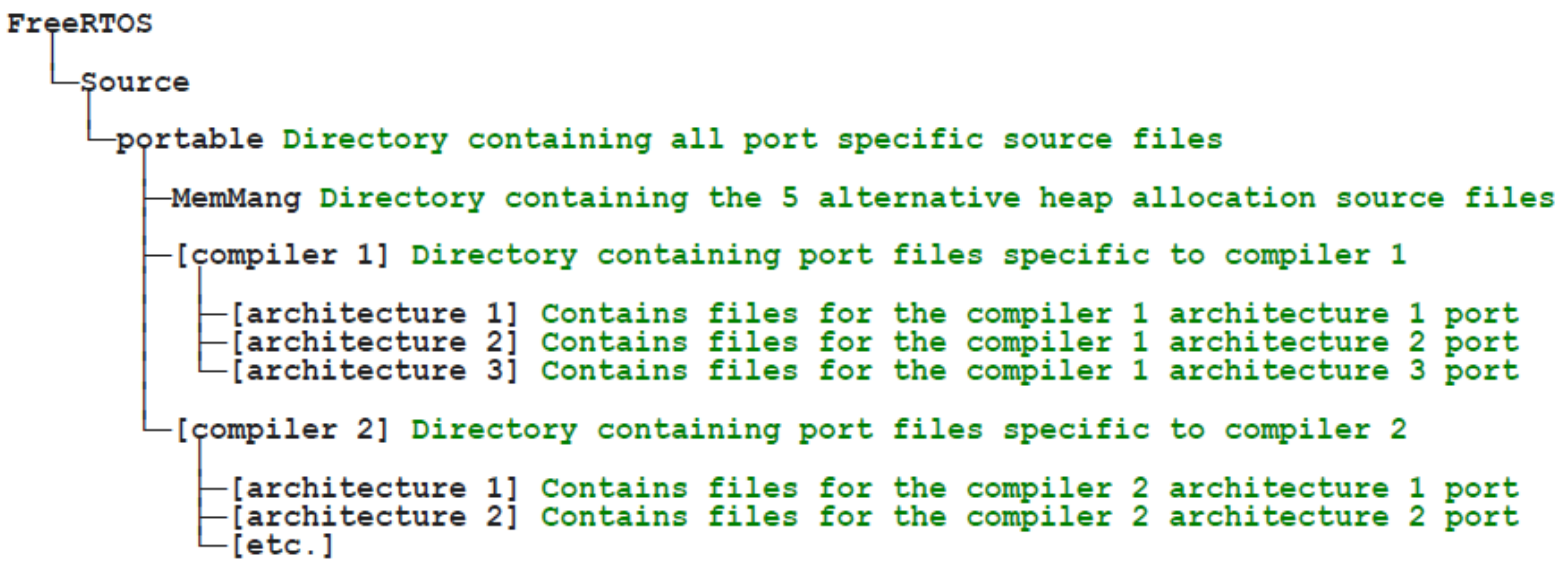
```
FreeRTOS
    ├─Source  Directory containing the FreeRTOS source files
    └─Demo    Directory containing pre-configured and port specific FreeRTOS demo projects
FreeRTOS-Plus
    ├─Source  Directory containing source code for some FreeRTOS+ ecosystem components
    └─Demo    Directory containing demo projects for FreeRTOS+ ecosystem components
```

- FreeRTOS 공통 파일 (FreeRTOS/Source)
  - task.c, list.c : 커널 핵심 파일
  - queue.c, timers.c, event_groups.c, croutine.c : 커널 서비스 파일

```
FreeRTOS
    └─Source
        ├─tasks.c          FreeRTOS source file - always required
        ├─list.c           FreeRTOS source file - always required
        ├─queue.c          FreeRTOS source file - nearly always required
        ├─timers.c         FreeRTOS source file - optional
        ├─event_groups.c   FreeRTOS source file - optional
        └─croutine.c       FreeRTOS source file - optional
```

FreeRTOS porting

- FreeRTOS 포팅을 위한 특정 소스 파일
  - FreeRTOS/Source/portable 디렉토리에 포함
  - Portable 디렉토리는 컴파일러별, 그 다음에 프로세서 아키텍처별 계층 구조로 구성됨
  - '*컴파일러*'라는 이름의 컴파일러를 사용하는 '*아키텍처*'라는 이름의 아키텍처를 가지는 프로세서의 경우
    - FreeRTOS/Source/portable/[*compiler*]/[*architecture*] 디렉토리에 위치



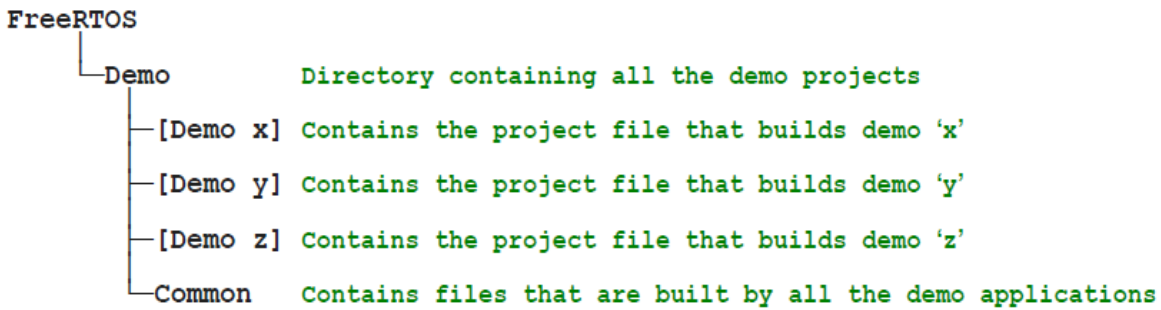
빌드 환경 설정

- 빌드 경로(Build Path) 포함하기
  - 다음의 세 가지 경로가 빌드 경로에 포함되어야 함
    - FreeRTOS/Source/include : 코어 FreeRTOS 헤더 파일들에 대한 경로
    - FreeRTOS/Source/portable/[*compiler*]/[*architecture*] : 사용하는 FreeRTOS에 특정되는 소스 파일들에 대한 경로
    - FreeRTOSConfig.h 헤더 파일에 대한 경로

- 어플리케이션에서 포함시켜야 할 헤더 파일
  - FreeRTOS.h 헤더 파일
    - 기본적으로 사용하는 FreeRTOS API 헤더 파일
  - 'task.h', 'queue.h', 'semphr.h', 'timers.h' 또는 'event_groups.h'
    - 해당 서비스를 사용할 때 포함시켜야 할 헤더파일
    - 하나 이상일 수 있음

데모 어플리케이션

- 데모 어플리케이션 및 그 목적
  - 적절한 파일이 포함되고 적절한 컴파일러 옵션이 설정되어 있는 미리 구성되어 있는 프로젝트와 실행 예를 제공
  - 최소한의 셋 업 또는 지식을 얻기 전에 우선 실험 가능
  - FreeRTOS API가 어떻게 사용될 수 있는지 보기
  - 실제 어플리케이션이 생성될 수 있는 베이스(base)로의 역할
- 데모 프로젝트
  - FreeRTOS/Demo 디렉토리 밑에 있는 단일한 서브-디렉토리에 위치
    - 서브-디렉토리명은 포팅이 어떤 데모 프로젝트에 연관이 있는지를 나타냄
  - FreeRTOS.org 웹 사이트에 데모 어플리케이션에 대한 설명이 존재
  - 데모 프로젝트에 포함되어 있는 main.c 파일내 주석 참조



FreeRTOS 프로젝트 생성

- 제공되는 데모 프로젝트들 중 하나를 선택하여 적용
  1. 제공되는 데모 어플리케이션을 열어 빌드 되고 예상한 데로 실행하는지 확인한다.
  2. 데모 태스크들을 정의하는 소스 파일들을 삭제한다. Demo/Common 디렉토리 내에 위치한 모든 파일들은 프로젝트에서 삭제될 수 있다.
  3. 리스트 1에서 보여지는 것과 같이 prvSetupHardware()와 vTaskStartScheduler()를 제외하고 main() 내에서 호출하는 모든 함수를 지운다.
  4. 프로젝트가 여전히 빌드 되는지 체크한다.

```
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;
}
```

FreeRTOS 프로젝트 생성

- 처음부터 새로운 프로젝트 생성하기
  1. 여러분이 선택한 도구를 사용한다면, 어떠한 FreeRTOS 소스 파일도 아직 포함하지 않는 새로운 프로젝트를 생성한다.
  2. 새로운 프로젝트가 빌드 되고, 여러분 타겟 하드웨어에 다운로드 되어 실행되는지 확인한다.
  3. 여러분이 이미 동작하는 프로젝트를 확실히 가지고 있을 때에만, 표 1에 자세히 나와 있는 FreeRTOS 소스 파일들을 프로젝트에 추가한다.
  4. 사용하려는 포팅용으로 제공되는 데모 프로젝트에서 사용되는 FreeRTOSConfig.h 헤더 파일을 프로젝트 디렉토리로 복사한다.
  5. 프로젝트가 헤더 파일 위치를 찾을 수 있도록 다음과 같은 디렉토리를 경로에 추가한다.
  6. FreeRTOS/Source/include
  7. FreeRTOS/Source/portable/[*compiler*]/[*architecture*] ([*compiler*]와 [*architecture*]는 여러분이 선택한 포팅용에 적합한 위치)
  8. FreeRTOSConfig.h 헤더 파일이 포함되어 있는 디렉토리
  9. 관계된 데모 프로젝트에서 컴파일러 설정을 복사한다.
  10. 필요할 것으로 보이는 모든 FreeRTOS 인터럽트 핸들러를 설치한다. 참고자료로서, 사용하려는 포팅 내용을 설명하고, 포팅용으로 제공되는 데모 프로젝트를 설명하는 웹 페이지를 사용하라.

FreeRTOS 프로젝트 생성

- 어플리케이션 프로젝트에 포함되어야 할 FreeRTOS 소스 파일들

File	Location
tasks.c	FreeRTOS/Source
queue.c	FreeRTOS/Source
list.c	FreeRTOS/Source
timers.c	FreeRTOS/Source
event_groups.c	FreeRTOS/Source
All C and assembler files	FreeRTOS/Source/portable/[compiler]/[architecture]
heap_n.c	From FreeRTOS V9.0.0 FreeRTOS applications can be completely statically allocated, removing the need to include a heap memory manager: FreeRTOS/Source/portable/MemMang, where n is either 1, 2, 3, 4 or 5. Refer to Chapter 2, Heap Memory Management, for more information.

FreeRTOS 데이터 형(Type)

- 공통적으로 가지는 데이터 형 두 가지 (portmacro.h에 정의됨)
  - TickType_t
    - Tick count값을 가지면서 시간을 명시하는 용도의 변수형을 지정할 때 사용
    - FreeRTOSConfig.h에 있는 configUSE_16_BIT_TICKS에 따라 16-비트 또는 32-비트 둘 중 하나의 형으로 설정 가능
  - BaseType_t
    - 아주 제한된 범위의 값을 가질 수 있는 리턴 형에 사용됨
    - 보통 pdTRUE/pdFALSE 형의 불리언(Booleans) 형으로 사용됨
    - 32-비트 아키텍처에서는 32-비트 형, 16-비트 아키텍처에서는 16-비트 형, 8-비트 아키텍처에서는 8-비트 데이터 형이 됨

- 변수 이름
  - 데이터 형을 접두어로 지정하여 변수 이름 작성
    - Ex: char -> 'c', int16_t(short) -> 's', int32_t(long) -> 'l', 그 외 구조체 등 'x'
  - 변수가 unsigned인 경우 'u', 포인터인 경우 'p'
    - Ex: uint8_t -> 'uc', character pointer -> 'pc'

FreeRTOS Naming convention

- 함수 이름
  - 리턴하는 데이터 형과 함수가 정의된 파일 둘 다가 접두사로 쓰인다
    - vTaskPrioritySet() 함수는 **task**.c 파일에 정의되어 있고 리턴 데이터 형은 void이다.
    - xQueueReceive() 함수는 **queue**.c 파일에 정의되어 있고 리턴 데이터 형은 BaseType_t이다.
    - pvTimerGetTimerID() 함수는 **timers**.c 파일에 정의되어 있고 리턴 데이터 형은 void를 가리키는 pointer이다.
  - 파일 내에서만 호출하고 사용되는 파일 전용 static 함수는 'prv' 접두사가 붙는다.

- 매크로 이름
  - 대문자로 쓰여지며, 매크로가 저장되어 있는 곳을 나타내기 위해 소문자가 접두사로 붙는다
  - 세마포어 API는 거의 전부 매크로 집합으로 쓰여져 있지만, 매크로 이름 규칙이 아닌 함수 이름 규칙을 따르고 있다는 것에 주의해라.

FreeRTOS Naming convention

Table 3. Macro prefixes

Prefix	Location of macro definition
port (for example, portMAX_DELAY)	portable.h or portmacro.h
task (for example, taskENTER_CRITICAL())	task.h
pd (for example, pdTRUE)	projdefs.h
config (for example, configUSE_PREEMPTION)	FreeRTOSConfig.h
err (for example, errQUEUE_FULL)	projdefs.h

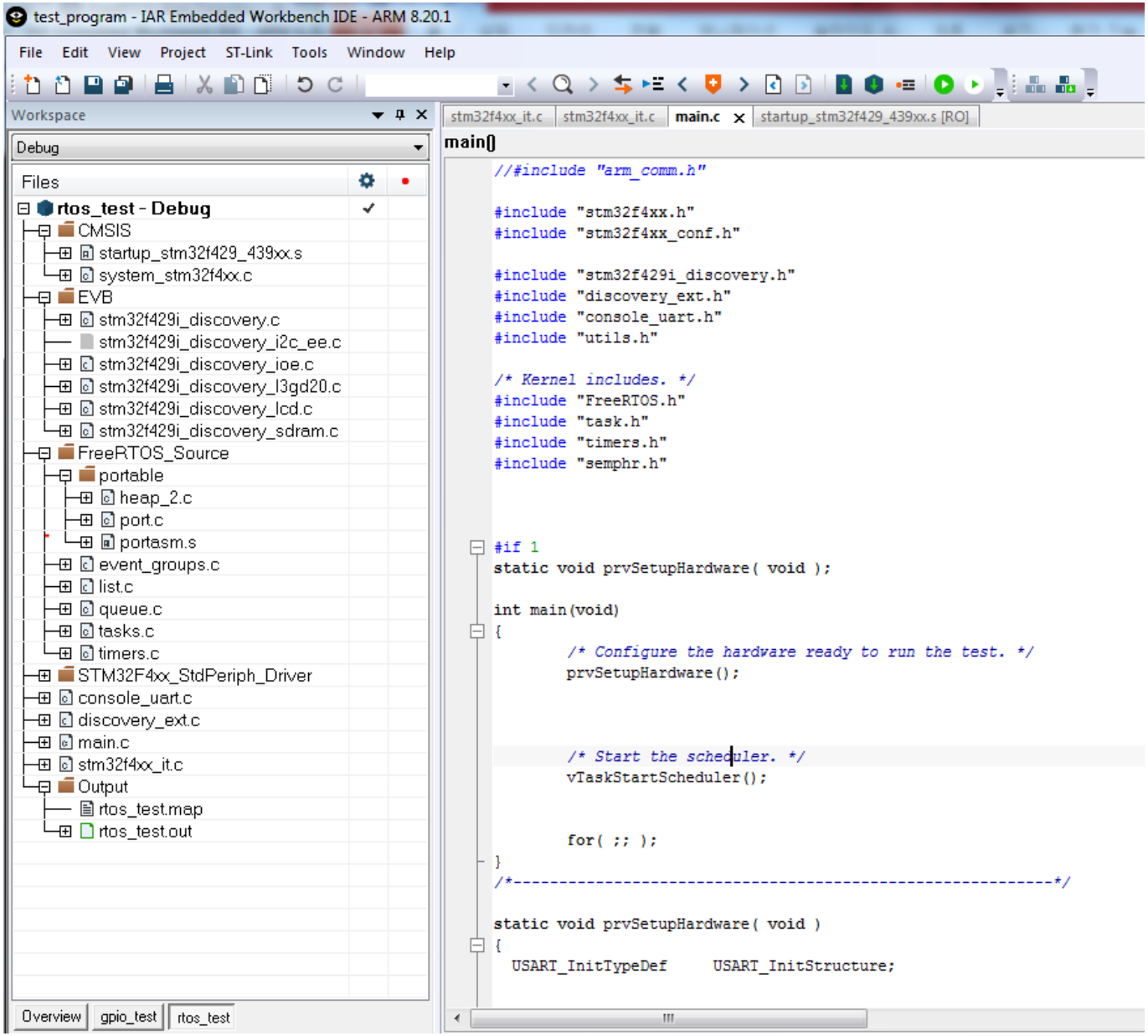
Table 4. Common macro definitions

Macro	Value
pdTRUE	1
pdFALSE	0
pdPASS	1
pdFAIL	0

FreeRTOS 테스트 프로젝트 생성

- GPIO_TEST 실습과 FreeRTOS 라이브러리를 모두 포함하는 프로젝트 생성
  - FreeRTOS V9.0.0에는 실습용 Discovery board와 동일한 데모 어플리케이션이 포함되어 있지 않음
  - GPIO_TEST에서 만든 Workspace에 rtos_test 프로젝트를 생성하여 추가한다
  - rtos_test 프로젝트에 FreeRTOS 소스 파일들을 추가한다.
    - 빌드용 헤더 파일 경로를 추가한다.
  - rtos_test 프로젝트에 gpio_test에 적용한 파일들을 모두 추가한다.
    - 빌드용 헤더 파일 경로를 추가한다.
  - assembly file 빌드시 발생되는 warning[25]는 disable시킴
  - 위와 같은 작업 후 프로젝트 빌드한 후 에러나 경고 없이 빌드 성공 확인

FreeRTOS 테스트 프로젝트 생성



실습-1 : Multi-tasking

- LED 점멸(Blinky) 테스트
  - vLedTask()를 생성
  - 500ms 주기로 LED3, LED4를 토글링 한다.

```
static portTASK_FUNCTION( vLEDBlnkyTask, pvParameters )
{
  /* The parameters are not used. */
  ( void ) pvParameters;

  for(;;)
  {
    vTaskDelay(500/portTICK_PERIOD_MS);
    STM_EVAL_LEDToggle(LED3);
    STM_EVAL_LEDToggle(LED4);
  }
}


int main(void)
{
  /* Configure the hardware ready to run the test. */
  prvSetupHardware();

  xTaskCreate( vLEDBlnkyTask, "LEDBLINKY", ledSTACK_SIZE, NULL, mainLED_BLINKY_TASK_PRIORITY, ( TaskHandle_t * ) NULL );

  /* Start the scheduler. */
  vTaskStartScheduler();

  for( ;; );
}
```

실습-2 : 인터럽트 처리

- PUSH BUTTON 입력(Input) 테스트
  - GPIO external interrupt를 사용하여 ISR에서 LED를 토글링한다.
  - USER Push Button : GPIO Port A0
  - 버튼이 눌렸을 때 '1'이 입력되는 Active "High" 입력
  - EXTI0를 사용

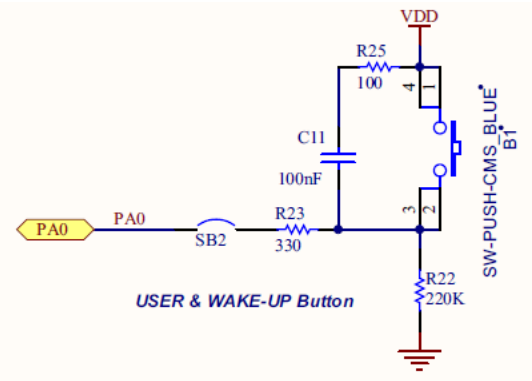
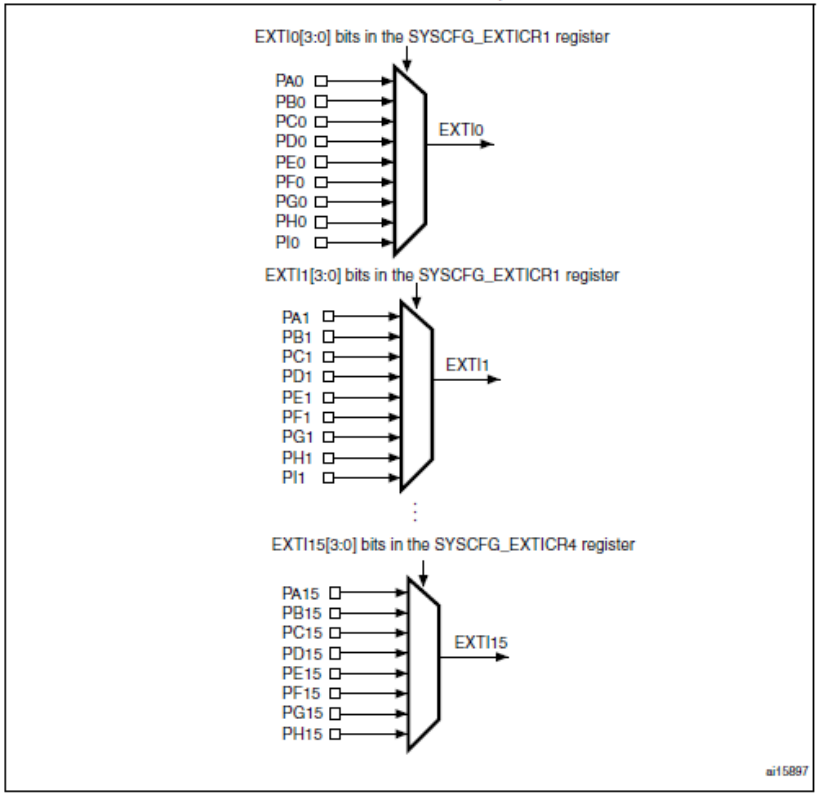


Figure 42. External interrupt/event GPIO mapping (STM32F405xx/07xx and STM32F415xx/17xx)



실습-2 : 인터럽트 처리

- ISR(Interrupt Service Routine) 처리 내용
  - ISR에서 빠져나올 때 Pending register를 통해 해당 인터럽트의 pending bit를 clear 시켜야 함

12.3.6 Pending register (EXTI_PR)

Address offset: 0x14
Reset value: undefined

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									PR22	PR21	PR20	PR19	PR18	PR17	PR16
									rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PR15	PR14	PR13	PR12	PR11	PR10	PR9	PR8	PR7	PR6	PR5	PR4	PR3	PR2	PR1	PR0
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

Bits 31:23 Reserved, must be kept at reset value.

Bits 22:0 **PRx:** Pending bit
0: No trigger request occurred
1: selected trigger request occurred
This bit is set when the selected edge event arrives on the external interrupt line.
This bit is cleared by programming it to '1'.

실습-2 : 인터럽트 처리

- ISR(Interrupt Service Routine) 초기화

```
static void prvSetupHardware( void )
{
  USART_InitTypeDef     USART_InitStructure;



  /* Setup STM32 system (clock, PLL and Flash configuration) */
  SystemInit();
  SystemCoreClockUpdate();


  STM_EVAL_LEDInit(LED3);
  STM_EVAL_LEDInit(LED4);

  STM_EVAL_PBInit(BUTTON_USER, BUTTON_MODE_EXTI);
```

```
void EXTI0_IRQHandler(void)
{
  STM_EVAL_LEDToggle(LED3);
  STM_EVAL_LEDToggle(LED4);

  EXTI->PR = USER_BUTTON_EXTI_LINE;
}
```

실습-3 : 태스크간 메시지 전달

- Button 처리 태스크를 생성한다.
- 실습 2에서 작성한 Push button ISR에서 큐를 통해 메시지를 Button 처리 태스크로 전송한다.
  - 메시지 전달용 큐 생성
  - Button 처리 태스크에서 실습2 ISR의 내용을 수행

```
int main(void)
{

  /* Configure the hardware ready to run the test. */
  prvSetupHardware();


  /* Create a queue capable of containing 10 unsigned long values. */
  g_xQueue1 = xQueueCreate( 5, sizeof( uint32_t ) );
```

```
void EXTI0_IRQHandler(void)
{
  BaseType_t xHigherPriorityTaskWoken;
  static uint32_t      u32SendData = 0;

  /* We have not woken a task at the start of the ISR. */
  xHigherPriorityTaskWoken = pdFALSE;

  u32SendData           ^=      0xFFFFFFFF;
  /* Post the byte. */
  xQueueSendFromISR( g_xQueue1, &u32SendData, &xHigherPriorityTaskWoken );

  //STM_EVAL_LEDToggle(LED3);
  //STM_EVAL_LEDToggle(LED4);

  EXTI->PR = USER_BUTTON_EXTI_LINE;
}
```

```
static portTASK_FUNCTION( vPushButtonTask, pvParameters )
{
  uint32_t      u32QRecv;


  /* The parameters are not used. */
  ( void ) pvParameters;


  for(;;)
  {

    if( xQueueReceive( g_xQueue1, &( u32QRecv ), ( TickType_t ) 10 ) )
    {
      if(u32QRecv != 0)
        STM_EVAL_LEDOn(LED4);
      else
        STM_EVAL_LEDOff(LED4);
    }
  }
}
```

실습-4 : 멀티 태스킹

- 실습 1과 실습3을 병행 실행시킨다.
  - 실습 1 LED test
    - LED3을 blinking한다 (0.5초 간격)
  - 실습 3 Pushbutton test
    - 버튼을 누를 때마다 LED4를 토글링 한다.

Thank you