개요

이 문서는 Richard Barry가 만든 오픈소스 실시간 내장형 운영체제인 FreeRTOS의 구조에 대해 설명합니다. OS 를 구성하는 요소 중 태스크(Task)와 코루틴(Co-routine), 큐(Queue) 그리고 세마포어(Semaphore) 가 FreeRTOS에서 어떤 방식으로 구현되어 있는지에 관해 설명합니다.

Contents

Tasks

태스크의 상태

태스크는 아래의 상태 중 하나로 존재 한다.

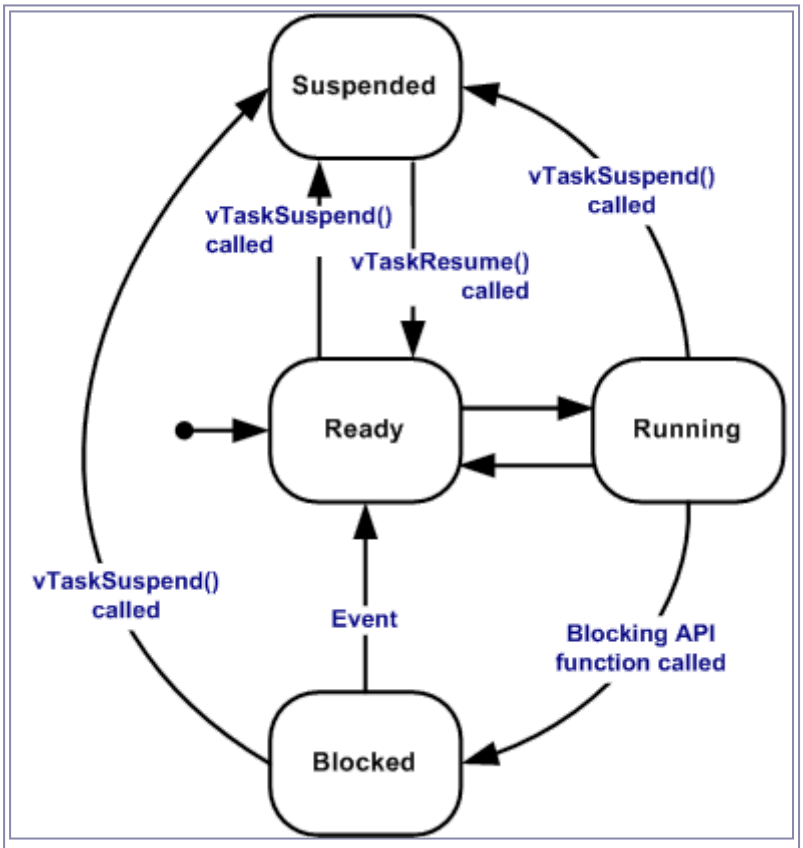
- 실행(Running)

태스크 가 실제로 실행될 때 이것을 실행중이라 한다 . 태스크가 현재 프로세서를 사용하고 있다.

- 준비(Ready)

준비 상태의 태스크는 실행 가능하지만 우선순위가 같거나 더 높은 다른 태스크 가 현재 이미 실행중인 상태에 있기 때문에 실행될 수 없는 상태를 말한다.

- 대기(Blocked)



태스크 상태 천이도

만약 태스크가 시간적 또는 외부적 이벤트를 위해 기다리고 있다면 이것은 대기 상태에 있다고 한다. 예를 들어 만약 태스크가 vTaskDelay() 함수를 호출한다면 이것은 지연 기간이 종료될 때까지 대기 중인 상태가 된다. 태스크는 또한 큐와 세마포어 이벤트를 기다릴 때 대기 상태가 될 수도 있다. 대기 중인 상태에 있는 태스크는 항상 '제한 시간'을 가지고 있다. 이 제한 시간이 지나면 태스크 는 대기 상태에서 벗어난다 (unblocked) 대기 중인 태스크는 스케줄링 대상에서 제외된다.

- 중지(Suspended)

중지 상태에 있는 태스크는 스케줄링 할 수 없다. 태스크는 오직 명시적으로 vTaskSuspend() 나 xTaskResume() 의 API 를 호출할 경우에만 중지 상태가 되거나 벗어날 수 있다. '제한 시간'은 명시

될 수 없다.

태스크의 우선순위

각각의 태스크에는 0부터 (configMAX_PRIORITIES - 1) 까지의 우선순위가 할당된다.

configMAX_PRIORITIES 는 FreeRTOSConfig.h 에 정의되어 있고 응용프로그램에 따라 다른 값으로 지정될 수 있다. 하지만 configMAX_PRIORITIES에 더 큰 값을 지정할수록 FreeRTOS 커널은 더 많은 RAM 을 소비한다.

숫자가 작을수록 낮은 우선순위의 태스크를 의미한다. idle 태스크의 우선순위는 tskIDLE_PRIORITY 로 정의되어 있고 기본적으로 0이다. 스케줄러는 항상 대기 또는 실행중인 태스크 중에서 가장 높은 우선순위의 태스크를 실행한다.

태스크의 구현

태스크는 아래와 같은 방식으로 구현할 수 있다.

```
1. void vATaskFunction( void *pvParameters )
2. {
3.    for( ;; )
4.    {

5.    }
6. }
```

1. pdTASK_CODE 형은 void 를 반환하고 매개변수로 void 포인터를 받는 함수로써 정의된다. 태스크를 구현하는 모든 함수는 이런 형식으로 작성되어야 한다. 그리고 void 포인터 매개변수는 어떤 형의 정보라도 태스크에 전달할 수 있도록 해준다.

3. 태스크 함수는 반환하지 않는다. 그래서 보통 무한루프로 작성된다.

4~5. 태스크 응용프로그램 코드가 여기 들어간다.

이렇게 구현된 태스크는 xTaskCreate() 함수를 호출함으로써 생성되고 vTaskDelete() 함수를 호출함으로써 삭제된다.

태스크 생성 매크로

태스크 함수는 portTASK_FUNCTION 이나 portTASK_FUNCTION_PROTO 매크로를 사용하여 정의될 수도 있다. 이 매크로들을 사용하면 컴파일러의 특정 문법을 써서 함수 정의 와 원형(prototype) 각각을 추가할 수 있게 해준다. (이식 환경에서 특별히 요구하지 않는 한 사용되지 않는다) 예를 들어 함수의 원형은 아래의 두 가지 방법으로 정의될 수 있다.:

```
void vATaskFunction( void *pvParameters );
```

또는

```
portTASK_FUNCTION_PROTO( vATaskFunction, pvParameters );
```

마찬가지로 함수의 구현도 아래와 같이 가능하다.

```
1. portTASK_FUNCTION( vATaskFunction, pvParameters )
2. {
3.    for( ;; )
```

```
4.    {
5.    }
6. }
```

4~5. 응용프로그램 코드를 이 사이에 넣는다.

Idle 태스크

idle 태스크는 스케줄러가 시작될 때 자동적으로 생성된다. idle 태스크는 삭제된 태스크에게 할당된 메모리를 해제시켜주는 기능을 한다. 그래서 vTaskDelete() 함수를 사용할 때는 idle 태스크가 CPU 처리 시간을 받지 못하는 상황(starvation)이 되지 않도록 주의할 필요가 있다.

idle 태스크는 다른 기능이 없다. 그래서 어떠한 상황에서든지 CPU 처리 시간을 다른 태스크에게 빼앗겨도 상관없다. 일반 응용프로그램 태스크가 Idle 태스크와 같은 우선순위(tskIDLE_PRIORITY)를 가지는 것도 가능하다.

Idle 태스크 훅

idle 태스크 훅 이란 idle 태스크 각각의 주기 동안 호출되는 함수이다. 만약 응용프로그램의 기능이 idle 태스크의 우선순위와 같게 동작하기를 원한다면 두 가지 방법이 있다.

1. **idle 태스크 훅 안에서 기능을 구현**한다 : 항상 적어도 하나의 태스크는 준비상태에 있어야만 한다. 그래서 훅 함수가 태스크를 대기 상태로 만드는 vTaskDelay() 같은 API를 호출하는 것은 바람직하지 못하다. 다만 코루틴(coroutine)의 경우에는 훅 함수 내부에서 그러한 API를 호출하는 것이 허용된다.
2. 기능을 구현하기 위해 **idle 태스크와 같은 우선순위를 가지는 태스크를 생성**한다 : 이것은 더욱 유연한 해결책이 되겠지만 더 많은 RAM 을 사용하므로 오버헤드가 있다.

Idle 태스크 훅을 생성하기 위해서는 FreeRTOSConfig.h 에 있는 configUSE_IDLE_HOOK 를 1로 설정한 후 아래의 원형을 가지는 함수를 정의한다.

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

idle 훅을 사용하면 간단하게 프로세서를 절전 상태로 만들 수 있다.

TCB

태스크가 가지고 있어야만 하는 정보들

- 스택에 관련된 정보

태스크를 수행하기 위해서는 자신만의 스택영역이 필요하다. 이 영역에는 지역변수, 프로그램 카운터, 레지스터 정보 등을 저장하여 문맥 전환이 일어날 때 정보를 잃어버리지 않도록 한다. 이를 위해 태스크 제어 블록은 스택의 시작 지점, 끝 지점, 그리고 스택의 크기에 관한 정보를 알아야 한다.

- 큐에 관련된 정보

태스크의 상태에는 준비, 대기, 실행, 중지 상태가 있다. 이 상태들을 구분하기 위하여 태스크는 각각의 상태에 해당하는 큐에 들어간다. 그래서

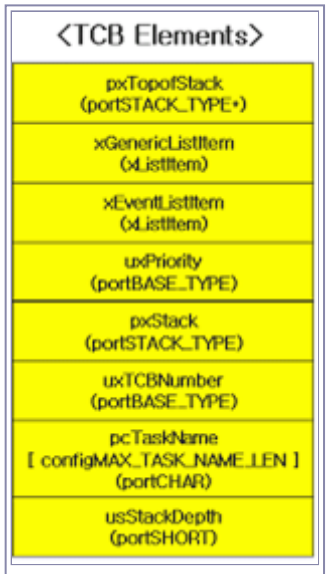
이런 큐를 위한 구조를 가지고 있어야만 한다.

- 태스크 우선순위

FreeRTOS 는 우선순위 기반의 스케줄링을 하기 때문에 각 태스크는 반드시 우선순위를 가져야 한다. 이를 위해 태스크 제어 블록은 우선순위에 관한 정보를 가져야 한다.

- 기타 정보

디버깅을 편리하게 하기 위해서는 태스크마다 고유의 이름과 번호를 가지는 것이 좋다. 이를 위해 위에서 언급한 것들과는 달리 필수요소는 아니지만 태스크 제어 블록이 태스크의 이름과 번호에 관한 정보를 가질 수 있다.



TCB 블록 구조

태스크 제어 블록(TCB)

TCB 란 태스크 제어 블록(Task Control Block) 의 약자이다. 태스크가 필요로 하는 여러 가지 정보들을 가지고 있는 블록이다. 위에서 언급한 태스크가 가져야할 여러 가지 정보를 실제로 구현해 놓은 것이라고 할 수 있다. 아래 그림은 실제로 FreeRTOS에서 TCB를 구현하는 소스코드 이다.

tskTaskControlBlock - TASKS.C

```
 1. typedef struct tskTaskControlBlock
 2. {
 3.     volatile portSTACK_TYPE *pxTopOfStack;
 4.     xListItem xGenericListItem;
 5.     xListItem xEventListItem;
 6.     unsigned portBASE_TYPE uxPriority;
 7.     portSTACK_TYPE *pxStack;
 8.     unsigned portBASE_TYPE uxTCBNumber;
 9.     signed portCHAR pcTaskName[ configMAX_TASK_NAME_LEN ];
10.     unsigned portSHORT usStackDepth;
11. } tskTCB;
```

위와 같이 TCB 는 구조체로 정의되어 있으며 태스크가 필요한 정보를 모두 가지고 있다.

태스크의 생성과 삭제

태스크를 생성하는 함수인 xTaskCreate()와 vTaskDelete() API의 원형만 알아도 응용프로그램을 구현할 수는 있지만 함수의 내부 구조를 알아야 더욱 안정적인 프로그램을 만들 수 있다. 그래서 여기서 간단히 xTaskCreate() 와 vTaskDelete() API 의 내부 구조를 살펴보고 대략적으로 어떤 방식으로 구현이 되어있는지를 소개한다. 그래서 FreeRTOS 내부적으로 태스크가 어떻게 생성되고 삭제되는 지 알아 본다.

태스크의 생성

xTaskCreate() - TASKS.C

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

```
                                    xTaskHandle *pxCreatedTask
                                    )
 2. {
 3. signed portBASE_TYPE xReturn;
 4. tskTCB * pxNewTCB;
 5. static unsigned portBASE_TYPE uxTaskNumber = 0;
 6.    pxNewTCB = prvAllocateTCBAndStack( usStackDepth );

 7.    if( pxNewTCB != NULL )
 8.    {
 9.       portSTACK_TYPE *pxTopOfStack;

10.       prvInitialiseTCBVariables( pxNewTCB,
                                     usStackDepth,
                                     pcName,
                                     uxPriority
                                     );

11.       #if portSTACK_GROWTH < 0
12.       {
13.          pxTopOfStack = pxNewTCB->pxStack + ( pxNewTCB->usStackDepth - 1 );
14.       }
15.       #else
16.       {
17.          pxTopOfStack = pxNewTCB->pxStack;
18.       }
19.       #endif

20.       pxNewTCB->pxTopOfStack = pxPortInitialiseStack( pxTopOfStack,
                                                          pvTaskCode,
                                                          pvParameters
                                                          );
```

10. 태스크에 대한 초기 정보 설정

20. 태스크를 위한 스택 생성

xTaskCreate()는 먼저 태스크 제어 블록을 생성하기 위해 tskTCB* pxNewTCB 라는 TCB 에 대한 포인터 변수를 선언한다. 그리고 매개변수로 전달 받은 태스크에 관련된 정보(스택의 크기, 태스크 이름, 우선순위)를 이 TCB 에 저장한다. 그 후 명시된 스택 크기만큼 생성하고 태스크로 넘어온 매개변수와 태스크가 수행될 지점의 주소를 스택에 저장한다.

```
21.       portENTER_CRITICAL();
22.       {
23.          uxCurrentNumberOfTasks++;
24.          if( uxCurrentNumberOfTasks == ( unsigned portBASE_TYPE ) 1 )
25.          {
26.             pxCurrentTCB =  pxNewTCB;

27.             prvInitialiseTaskLists();
28.          }
29.          else
30.          {
31.             if( xSchedulerRunning == pdFALSE )
32.             {
33.                if( pxCurrentTCB->uxPriority <= uxPriority )
34.                {
35.                   pxCurrentTCB = pxNewTCB;
36.                }
37.             }
38.          }

39.          if( pxNewTCB->uxPriority > uxTopUsedPriority )
40.          {
41.             uxTopUsedPriority = pxNewTCB->uxPriority;
42.          }
```

```
43.          pxNewTCB->uxTCBNumber = uxTaskNumber;
44.          uxTaskNumber++;

45.          prvAddTaskToReadyQueue( pxNewTCB );

46.          xReturn = pdPASS;
47.       }
48.       portEXIT_CRITICAL();
49.    }
```

21. 크리티컬 섹션 시작.

45. 준비 큐에 태스크를 추가함

48. 크리티컬 섹션 끝

이 후 현재 생성되는 태스크를 우선순위에 따라 레디 큐에 넣는다.

```
50.    else
51.    {
52.       xReturn = errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY;
53.    }

54.    if( xReturn == pdPASS )
55.    {
56.       if( ( void * ) pxCreatedTask != NULL )
57.       {
58.          *pxCreatedTask = ( xTaskHandle ) pxNewTCB;
59.       }

60.       if( xSchedulerRunning != pdFALSE )
61.       {
62.          if( pxCurrentTCB->uxPriority < uxPriority )
63.          {
64.             taskYIELD();
65.          }
66.       }
67.    }

68.    return xReturn;
69. }
```

58. 태스크 생성에 성공하면 핸들을 저장한다.

64. 생성된 태스크가 현재 실행중인 태스크보다 우선순위가 높을 경우에는 문맥 전환을 함.

마지막으로 태스크가 성공적으로 생성되면 태스크에 대한 핸들을 저장한다. 그리고 taskYIELD() 함수를 호출하여 문맥 전환이 일어난다.

태스크의 삭제

vTaskDelete()함수는 태스크를 모든 대기 큐에서 삭제하고 스케줄링의 대상에서 제외시키는 함수이다. 여기서 주의 할 것은 vTaskDelete()함수가 호출되었다고 해서 태스크에 **할당된 모든 메모리가 그 즉시 해제되는 것이 아니라는 것**이다.

vTaskDelete() - TASKS.C

```
1. #if ( INCLUDE_vTaskDelete == 1 )

2.    void vTaskDelete( xTaskHandle pxTaskToDelete )
3.    {
4.    tskTCB *pxTCB;
```

```
5.        taskENTER_CRITICAL();
6.        {
7.           if( pxTaskToDelete == pxCurrentTCB )
8.           {
9.              pxTaskToDelete = NULL;
10.          }

11.          pxTCB = prvGetTCBFromHandle( pxTaskToDelete );

12.          vListRemove( &( pxTCB->xGenericListItem ) );

13.          if( pxTCB->xEventListItem.pvContainer )
14.          {
15.             vListRemove( &( pxTCB->xEventListItem ) );
16.          }

17.          vListInsertEnd( ( xList * ) &xTasksWaitingTermination,
                             &( pxTCB->xGenericListItem )
                              );

18.          ++uxTasksDeleted;
19.       }
20.       taskEXIT_CRITICAL();
```

2. 인자로 삭제되어야 할 태스크의 핸들이 넘어온다.

7. 삭제될 태스크가 현재 실행중인 태스크인가를 판별한다.

11. 자신을 삭제하기 위한 핸들을 받아온다.

12. 삭제할 태스크를 준비 리스트에서 제거한다.

13. 삭제 대상이 이벤트를 기다리고 있는 중이라면 이벤트 리스트에서도 제거한다.

17. 할당된 메모리를 앞으로 해제하기 위한 해제 대기 리스트에 TCB를 올려놓는다.

```
21.       if( xSchedulerRunning != pdFALSE )
22.       {
23.          if( ( void * ) pxTaskToDelete == NULL )
24.          {
25.             taskYIELD();
26.          }
27.       }
28.    }

29. #endif
```

vTaskDelete()함수는 태스크의 핸들을 매개변수로 받는다. 그리고 이 핸들을 이용하여 TCB 에 대한 포인터를 얻어온다. 이제 TCB를 이용하여 모든 대기 큐에서 태스크를 삭제하고 xTasksWaitingTermination 리스트에 태스크를 추가 한다. 그래서 나중에 실제로 메모리를 해제 시켜주는 작업을 따로 해주도록 한다. 이 모든 작업이 끝나면 taskYIELD() 함수를 호출해서 새로운 태스크가 실행 중인 상태에 들어가도록 해 준다.

코루틴(Co-routines)

코루틴이란 무엇인가

메인 루틴과 서브루틴처럼 주종 관계에 있는 것이 아니라 동등한 관계를 가지고 서로를 호출할 수 있는 루틴들을 가리킨다. 코루틴도 서브루틴과 유사한 형식으로 호출되지만 서브루틴이 호출될 때

마다 처음부터 다시 실행되는데 비해 코루틴은 가장 최근에 실행을 마친 곳에서부터 실행을 재개한다. 게임 프로그램에서 각 경기자의 루틴은 서로 코루틴이 된다.

코루틴의 상태

코루틴은 다음 중 하나의 상태에 있다.

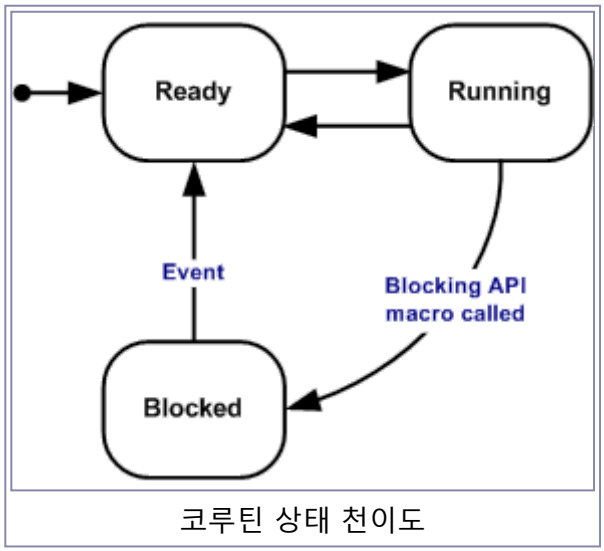
- 실행(Running)

코루틴이 실제로 실행되고 있을 때 실행상태에 있다고 한다. 현재 프로세서를 사용하고 있는 상태이다.

- 준비(Ready)

준비상태의 코루틴은 실행 가능하지만 현재 실행중이 아닌 상태이다 (대기중인 상황과는 다르다). 코루틴은 다음과 같은 이유로 준비상태가 된다.

1. 우선순위가 같거나 더 높은 코루틴이 현재 실행상태에 있을 경우
2. 태스크가 실행상태에 있을 경우.



코루틴 상태 천이도

(응용프로그램이 태스크와 코루틴을 함께 사용할 때)

- 대기(Blocked)

코루틴이 현재 시간적인 또는 외부적인 이벤트를 기다리고 있을 경우 대기상태에 있다고 한다. 예를 들어 코루틴이 crDELAY()를 호출할 경우 지연 기간이 종료될 때 까지 대기상태에 있게 된다. 대기 중인 코루틴은 스케줄링이 불가능하다.

코루틴의 우선순위

각각의 코루틴은 0에서 (configMAX_CO_ROUTINE_PRIORITIES - 1) 까지의 우선순위를 할당받는다. **숫자가 작을수록 코루틴의 수선순위는 낮아진다**. configMAX_CO_ROUTINE_PRIORITIES 는 FreeRTOSConfig.h 에 정의되어 있고 응용프로그램 자체적으로 따라 다른 값을 지정할 수 있다. configMAX_CO_ROUTINE_PRIORITIES 값을 높게 지정할수록 더 많은 RAM이 소비된다.

코루틴의 우선순위는 오직 같은 코루틴 사이에서만 의미가 있다. 만약 태스크와 코루틴을 같은 응용프로그램에서 함께 사용할 경우 **태스크가 코루틴보다 항상 우선**적으로 실행된다.

코루틴의 구현

코루틴은 다음과 같은 구조로 이루어진다. 그리고 몇 가지 지켜야할 사항이 있다.

```
1. void vACoRoutineFunction( xCoRoutineHandle xHandle,
                             unsigned portBASE_TYPE uxIndex )
2. {
3.     crSTART( xHandle );
4.     for( ;; )
5.     {
```

```
6.     }
7.     crEND();
8. }
```

1. 선언될 때의 형태이다.

3. 코루틴은 반드시 crSTART()를 호출함으로써 시작한다.

4. 코루틴은 반환할 수 없다. 그래서 보통 무한루프로 구현된다.

5~6. 여기에 코루틴 응용프로그램이 들어간다.

7. 코루틴은 반드시 crEND()를 호출함으로써 종료된다.

crCOROUTINE_CODE, xCoRoutineHandle 타입과 corCoRoutineControlBlock 구조체

crCOROUTINE_CODE 타입은 void를 반환하는 함수로 정의되어 있고 xCoRoutineHandle 과 이것의 매개변수의 인덱스를 받는다. 코루틴을 구현하는 모든 함수는 이 타입이어야만 한다. 많은 코루틴이 하나의 코루틴 함수에서 만들어 질수 있다. uxIndex 매개 변수는 코루틴 사이를 구별하는 수단으로 사용된다.

corCoRoutineControlBlock - CROUTINE.H

```
 1. typedef void * xCoRoutineHandle;

 2. typedef void (*crCOROUTINE_CODE)( xCoRoutineHandle,
                                      unsigned portBASE_TYPE );

 3. typedef struct corCoRoutineControlBlock
 4. {
 5.    crCOROUTINE_CODE pxCoRoutineFunction;
 6.    xListItem xGenericListItem;
 7.    xListItem xEventListItem;
 8.    unsigned portBASE_TYPE uxPriority;
 9.    unsigned portBASE_TYPE uxIndex;
10.    unsigned portSHORT uxState;
11. } corCRCB;
```

1. 구현된 코루틴 제어 블록(co-routine contrrol block - CRCB)을 감추기 위해 사용되었다. 그러나 제어 블록 구조체는 코루틴이 동작하기 위해서 헤더에 포함되어있어야 한다.

2. 코루틴 함수가 따라야만 하는 원형이다.

6. CRCB 를 'ready' 와 'blocked' 큐에 놓기 위해 사용되는 리스트 항목.

7. CRCB 를 이벤트 리스트에 넣기 위해 사용되는 리스트 항목.

8. 다른 코루틴에 대한 이 코루틴의 우선순위.

9. 복수의 코루틴이 하나의 코루틴함수를 사용할 때 서로를 구분하는데 쓰이는 식별자.

10. 코루틴 구현을 위해 내부적으로 사용되는 상태변수.

crSTART() 와 crEND()

모든 코루틴 함수는 반드시 crSTART()를 호출함으로써 시작하고 crEND()를 호출함으로써 끝내야 한다. (매크로 내부에서 switch {, } 구문을 구현하기 때문)

crSTART(), crEND() - CROUTINE.H

```
1. #define crSTART( pxCRCB ) switch( ( ( corCRCB * )pxCRCB )->uxState )
2.                                { case 0:
3. #define crEND()
4.                                }
```

1~4. 매크로를 이용해서 CRCB의 uxState 값이 0일때만 코루틴이 실행되도록 한다.

준비 또는 대기상태인 코루틴을 위한 리스트

lists - CROUTINE.C

```
1. static xList pxReadyCoRoutineLists [ configMAX_CO_ROUTINE_PRIORITIES ];
2. static xList xDelayedCoRoutineList1;
3. static xList xDelayedCoRoutineList2;
4. static xList * pxDelayedCoRoutineList;
5. static xList * pxOverflowDelayedCoRoutineList;
6. static xList xPendingReadyList;
```

1 . 우선순위에 따라 정리되어 'ready' 상태에 있는 코루틴들의 리스트.

2, 3. 지연된 코루틴들. (두개의 리스트가 사용된다. 현재 tick count에서 오버플로된 코루틴을 위해서 하나가 더 필요하다)

4. 현재 사용되고 있는 지연 리스트를 가리키는 포인터.

5. 오버플로딘 코루틴을 위한 지연 리스트를 가리키는 포인터.

6. 외부 이벤트에 의해 'ready' 상태가 되었지만 리스트에 즉시 추가되지 못하는 코루틴을 위한 임시 리스트.

코루틴의 생성

코루틴은 xCoRoutineCreate() API를 불러서 생성할 수 있다.

xCoRoutineCreate() - CROUTINE.C

```
 1. signed portBASE_TYPE xCoRoutineCreate( crCOROUTINE_CODE pxCoRoutineCode,
                                           unsigned portBASE_TYPE uxPriority,
                                           unsigned portBASE_TYPE uxIndex
                                           )
 2. {
 3. signed portBASE_TYPE xReturn;
 4. corCRCB *pxCoRoutine;
 5.    pxCoRoutine = ( corCRCB * ) pvPortMalloc( sizeof( corCRCB ) );
 6.    if( pxCoRoutine )
 7.    {
 8.       if( pxCurrentCoRoutine == NULL )
 9.       {
10.          pxCurrentCoRoutine = pxCoRoutine;
11.          prvInitialiseCoRoutineLists();
12.       }
13.       if( uxPriority >= configMAX_CO_ROUTINE_PRIORITIES )
14.       {
15.          uxPriority = configMAX_CO_ROUTINE_PRIORITIES - 1;
16.       }
17.       pxCoRoutine->uxState = corINITIAL_STATE;
18.       pxCoRoutine->uxPriority = uxPriority;
19.       pxCoRoutine->uxIndex = uxIndex;
20.       pxCoRoutine->pxCoRoutineFunction = pxCoRoutineCode;
```

```
21.       vListInitialiseItem( &( pxCoRoutine->xGenericListItem ) );
22.       vListInitialiseItem( &( pxCoRoutine->xEventListItem ) );
```

5. CRCB를 위한 메모리를 할당한다.

8. 현재 실행중인 코루틴이 없으면

10. 이것은 최초로 실행되는 코루틴이고,

11. 코루틴을 위한 자료구조를 초기화한다.

13. 만약 우선순위가 범위를 벗어난 경우

15. 범위에 맞도록 조정한다.

17~20. CRCB를 현재의 함수 인자에 따라 설정한다.

21, 22. 다른 모든 CRCB 수치를 초기화한다.

```
23.       listSET_LIST_ITEM_OWNER( &( pxCoRoutine->xGenericListItem ), pxCoRoutine );
24.       listSET_LIST_ITEM_OWNER( &( pxCoRoutine->xEventListItem ), pxCoRoutine );
25.       listSET_LIST_ITEM_VALUE( &( pxCoRoutine->xEventListItem ),
                                   configMAX_PRIORITIES - ( portTickType ) uxPriority );
26.       prvAddCoRoutineToReadyQueue( pxCoRoutine );
27.       xReturn = pdPASS;
28.    }
29.    else
30.    {
           xReturn = errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY;
31.    }
32.    return xReturn;
33. }
```

23, 24. CRCB를 리스트에 연결한다.

25. 이벤트 리스트는 항상 우선순위 순서대로 놓여있어야 한다.

26. CRCB를 'ready' 리스트에 추가한다.

코루틴의 스케줄링

코루틴 은 vCoRoutineSchedule()을 반복적으로 호출해서 스케줄링한다. vCoRoutineSchedule()을 호출하기 가장 좋은 장소는 idle task hook 내부이다. 사용자가 작성한 응용프로그램이 오직 코루틴 만 사용한다 하더라도 스케줄러가 시작될 때 idle task 가 자동적으로 생성되기 때문에 코루틴을 스케줄링 할 수 있다. vCoRoutineSchedule() 은 실행 가능한 가장 높은 우선순위의 코루틴을 실행시킨다. 코루틴은 대기 상태가 되거나 태스크에 의해 선점될 때까지 실행된다. 코루틴은 협동적으로 실행되기 때문에 하나의 코루틴이 다른 것에 의해 선점되지 않는다. 하지만 태스크에 의해서는 항상 선점된다. 만약 응용프로그램이 태스크와 코루틴 모두를 포함한다면 vCoRoutineSchedule() 은 idle task hook 내부에서 호출 되어야 한다.

vCoRoutineSchedule() - CROUTINE.C

```
 1. void vCoRoutineSchedule( void )
 2. {
 3.    prvCheckPendingReadyList();
 4.    prvCheckDelayedList();
 5.    while( listLIST_IS_EMPTY( &( pxReadyCoRoutineLists[ uxTopCoRoutineReadyPriority ] ) ) )
 6.    {
```

```
 7.       if( uxTopCoRoutineReadyPriority == 0 )
 8.       {
 9.          return;
10.       }
11.       --uxTopCoRoutineReadyPriority;
12.    }
13.    listGET_OWNER_OF_NEXT_ENTRY( pxCurrentCoRoutine,
                                    &( pxReadyCoRoutineLists[ uxTopCoRoutineReadyPriority ] )
                                     );
14. (pxCurrentCoRoutine->pxCoRoutineFunction )( pxCurrentCoRoutine, pxCurrentCoRoutine->uxIndex );
15. return;
16. }
```

3. 이벤트에 의해 'ready' 상태가 되었으나 리스트에 추가되지 못한 코루틴이 있는가 확인.

4. 지연된 코루틴이 타임아웃 되었는가 확인함

5~12. 'ready' 상태의 코루틴 중 가장 높은 우선순위의 코루틴이 들어있는 ready queue 를 찾음

13. listGET_OWNER_OF_NEXT_ENTRY() 에 의해 같은 우선순위의 코루틴은 CPU time을 공유한다

14. 코루틴을 실행한다.

태스크와 코루틴을 함께 사용할 때

idle task에서 코루틴을 스케줄링 하는 것은 태스크와 코루틴을 같은 응용프로그램 안에서 쉽게 쓸 수 있도록 해준다. 이 경우에서는 idle task 보다 높은 우선순위의 실행 가능한 태스크가 없을 경우에만 코루틴 이 실행된다.

사용상의 제약

코루틴이 태스크에 비해서 상대적으로 더 작은 RAM을 사용한다는 장점이 있지만 다음과 같은 이유로 인해 사용상의 제약이 있다. 그래서 코루틴은 태스크보다 사용하기 더 어렵다.

- 코루틴의 스택은 코루틴이 대기 상태가 될 경우 유지되지 않는다. 이것은 스택에 할당된 변수들이 값을 잃어버릴 가능성이 있다는 것을 의미한다. 이를 극복하기 위해서 코루틴을 대기 상태로 만드는 함수의 호출을 거치면서도 유지되어야만 하는 값들은 반드시 static 으로 선언되어야 한다. 다음의 경우를 보자.

```
 1. void vACoRoutineFunction( xCoRoutineHandle xHandle,
                              unsigned portBASE_TYPE uxIndex )
 2. {
 3. static char c = 'a';
 4.    crSTART( xHandle );
 5.    for( ;; )
 6.    {
 7.       c = 'b';
 8.       crDELAY( xHandle, 10 );
 9.    }
10.    crEND();
11. }
```

7.에서 c='b'로 할당한 값은 8.의 지연 시간 동안에 바뀔 수 있다. 여기서는 3.에서 변수 c를 static으로 선언했기 때문에 그 값을 보장할 수 있다.

스택을 공유하기 때문에 생기는 또다른 문제점으로는 코루틴을 대기 상태로 만드는 API 함수는 코루틴 자체에서만 호출되어야 한다는 것이다. 즉, 코루틴에서 다른 함수를 호출하고 그 안에서 코루틴을 대기 상태로 만들면 안 된다.

```
1. void vACoRoutineFunction( xCoRoutineHandle xHandle,
                             unsigned portBASE_TYPE uxIndex )
2. {
3.    crSTART( xHandle );
4.    for( ;; )
5.    {
6.       crDELAY( xHandle, 10 );
7.       vACalledFunction();
8.    }
9.    crEND();
10. }
11. void vACalledFunction( void )
12. {

13. }
```

6.에서 crDELAY()를 호출하는것은 허용되지만 7.에서 호출된 vACalledFunction() 내부에서 crDELAY()를 호출하면 안 된다.

- 기본적으로 FreeRTOS에서 구현된 코루틴은 switch구문 안에서 코루틴을 대기 상태로 만드는 호출을 허용하지 않는다.

```
1. void vACoRoutineFunction( xCoRoutineHandle xHandle,
                             unsigned portBASE_TYPE uxIndex )
2. {
3.    crSTART( xHandle );
4.    for( ;; )
5.    {
6.       crDELAY( xHandle, 10 );
7.       switch( aVariable )
8.       {
9.          case 1 :
10.            break;
11.         default:
12.      }
13.   }
14.   crEND();
15. }
```

앞의 경우와 비슷하게 여기서도 6.에서는 crDELAY()를 호출할 수 있으나 7~12.의 switch 구문 내에서는 호출하면 안 된다.

간단한 예제

코루틴을 사용한 간단한 예제를 살펴보자.

LED를 깜빡이는 간단한 코루틴 작성

아래의 코드는 LED를 주기적으로 깜박이는 간단한 코루틴이다. (간단한 코드이므로 설명은 생략한다)

```
1. void vFlashCoRoutine( xCoRoutineHandle xHandle,
                         unsigned portBASE_TYPE uxIndex )
2. {
3.    crSTART( xHandle );
4.    for( ;; )
5.    {
```

```
6.        crDELAY( xHandle, 10 );
7.        vParTestToggleLED( 0 );
8.     }
9.     crEND();
10. }
```

코루틴의 스케줄링

앞에서 설명했듯이 코루틴 은 vCoRoutineSchedule()을 반복적으로 호출함으로써 스케줄링된다. 이것을 위한 가장 적당한 장소는 idle task hook을 작성함으로서 idle task 내부에서 작동시키는 것이다. 먼저 FreeRTOSConfig.h 안에 있는 configUSE_IDLE_HOOK 이 1로 설정되었는지 확인하고 다음과 같은 idle task hook을 작성한다.

```
1. void vApplicationIdleHook( void )
2. {
3.    vCoRoutineSchedule( void );
4. }
```

또 다른 방법으로는 idle task 가 아무 일도 하지 않는다면 vCoRoutine Schedule()에서 무한루프를 사용해 반복적으로 호출하는 것도 괜찮다.

```
1. void vApplicationIdleHook( void )
2. {
3.    for( ;; )
4.    {
5.       vCoRoutineSchedule( void );
6.    }
7. }
```

코루틴을 생성하고 스케줄러를 시작시키기

코루틴 은 main() 내부에서 생성한다.

```
1. #include "task.h"
2. #include "croutine.h"
3. #define PRIORITY_0 0

4. void main( void )
5. {
6.    xCoRoutineCreate( vFlashCoRoutine, PRIORITY_0, 0 );
7.    vTaskStartScheduler();
8. }
```

6. 코루틴을 생성한다. 우선순위는 0이며 인덱스는 생략되었다.

(6, 7 사이에서 태스크 역시 생성할 수 있다)

7. 스케줄러를 시작한다.

확장된 예제 - 인덱스를 사용하기

같은 함수에서 8개의 코루틴을 생성하고 싶다고 가정하자. 각각의 코루틴 은 서로 다른 LED를 다른 주기로 깜빡이게 할 것이다. 인덱스는 코루틴 들을 코루틴 함수 안에서 구분하기위해 사용한다.

먼저 8개의 코루틴을 만들고 각각에 서로 다른 index를 할당해 넘겨준다.

```
1. #include "task.h"
2. #include "croutine.h"
3. #define PRIORITY_0        0
4. #define NUM_COROUTINES    8
```

```
 5. void main( void )
 6. {
 7.    int i;
 8.    for( i = 0; i < NUM_COROUTINES; i++ )
 9.    {
10.       xCoRoutineCreate( vFlashCoRoutine, PRIORITY_0, i );
11.    }
12.    vTaskStartScheduler();
13. }
```

8~11.에서 i를 0에서부터 1씩 증가시키면서 이것을 인덱스로 넘겨주고 코루틴을 반복적으로 생성한다.

12. 그리고 스케줄러를 시작한다.

이제 vFlashCoRoutine()을 조금 더 확장해서 서로 다른 LED 가 다른 주기로 깜박이게 해 보자.

```
 1. const int iFlashRates[ NUM_COROUTINES ] = { 10, 20, 30, 40, 50, 60, 70, 80 };
 2. const int iLEDToFlash[ NUM_COROUTINES ] = { 0, 1, 2, 3, 4, 5, 6, 7 };

 3.void vFlashCoRoutine( xCoRoutineHandle xHandle,
                         unsigned portBASE_TYPE uxIndex )
 4. {
 5.    crSTART( xHandle );
 6.    for( ;; )
 7.    {
 8.       crDELAY( xHandle, iFlashRate[ uxIndex ] );
 9.       vParTestToggleLED( iLEDToFlash[ uxIndex ] );
10.    }
11.    crEND();
12. }
```

8. 고정된 지연값 대신 iFlashRate에 해당하는 시간만큼 지연시킨다.

9. 해당하는 LED를 찾아서 깜빡인다.

큐(Queue)

큐의 정의

- 메시지 큐

메시지 큐는 태스크 또는 ISR이 다른 태스크에게 포인터 크기의 변수를 보낼 수 있도록 해 준다. 즉 태스크 간 통신(IPC)을 위한 도구이다. 각각의 포인터는 보통 초기화 될 때 응용프로그램에 특화된 메시지를 담은 자료구조를 가리키게 된다.

구현

항목들은 '참조'가 아닌 '복사'되어 큐에 넣어진다. 그러므로 큰 항목들을 큐에 넣을 때는 그 항목들을 가리키는 포인터들을 큐에 넣는 것이 바람직하다. FreeRTOS 데모 응용프로그램 파일인 'blockQ.c' 와 'pollQ.c' 에 큐의 사용에 관한 예를 보여준다. FreeRTOS에서 사용된 큐의 구현은 응용프로그램 코드에서도 사용할 수 있다. 태스크와 코루틴은 자료가 큐에서 사용가능해지거나 기록하기에 충분한 공간이 확보되기를 기다리는 동안 대기 상태가 될 수 있다.

큐의 자료구조

QueueDefinition - QUEUE.C

```
 1. typedef struct QueueDefinition
 2. {
 3.     signed portCHAR *pcHead;
 4.     signed portCHAR *pcTail;
```

3. 큐가 저장되는 영역의 시작점을 가리키는 포인터.

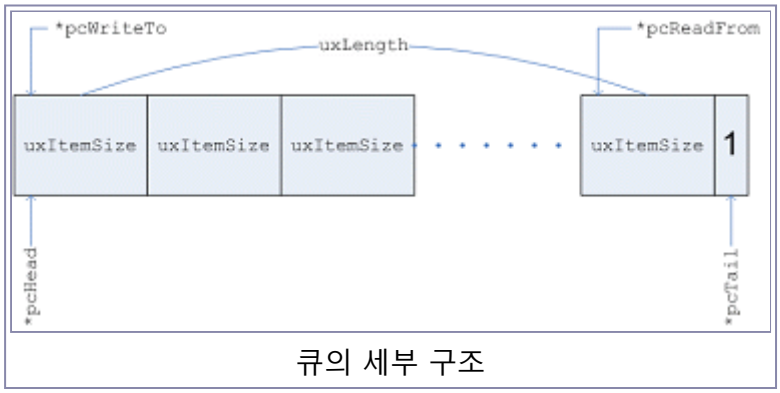
4. 큐가 정의될 때 모든 항목들을 저장하는데 필요한 것 보다 한 바이트 많게 메모리가 할당된다. 그 마지막 바이트를 가리키는 포인터.

```
 5.     signed portCHAR *pcWriteTo;
 6.     signed portCHAR *pcReadFrom;

 7.     xList xTasksWaitingToSend;
 8.     xList xTasksWaitingToReceive;

 9.     unsigned portBASE_TYPE uxMessagesWaiting;
10.     unsigned portBASE_TYPE uxLength;
11.     unsigned portBASE_TYPE uxItemSize;

12.     signed portBASE_TYPE xRxLock;
13.     signed portBASE_TYPE xTxLock;
14. } xQUEUE;
```

5. 이번에 큐에 들어갈 항목이 저장될 지점을 가리키는 포인터.

6. 이번에 꺼내서 읽을 항목의 시작지점을 가리키는 포인터. 초기화될 때 이것은 마지막 항목이 시작하는 지점을 가리킨다.

7. 큐에 메시지를 넣기 위해 대기하고 있는 태스크의 리스트. (우선순위 순서대로 저장되어 있다.)



큐의 세부 구조

8. 큐에서 메시지를 가져오기 위해 대기하고 있는 태스크의 리스트. (우선순위 순서대로 저장되어 있다.)

9. 현재 큐에 들어있는 메시지의 개수

10. 현재 큐의 길이. 큐에 들어갈 수 있는 메시지의 개수를 의미한다. (바이트 크기가 아님)

11. 큐에 있는 메시지 각각의 크기. (바이트 단위)

12. 큐가 잠겨있는 동안 큐에서 빠져나간 메시지의 개수. (큐가 잠겨있지 않은 경우에 이 값은 queueUNLOCKED 값을 가진다.)

13. 큐가 잠겨있는 동안 큐에 들어간 메시지의 개수. (큐가 잠겨있지 않은 경우에 이 값은 queueUNLOCKED 값을 가진다.)

큐의 생성

xQueueCreate() - QUEUE.C

```
 1. xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength,
                               unsigned portBASE_TYPE uxItemSize )
 2. {
 3. xQUEUE *pxNewQueue;
 4. size_t xQueueSizeInBytes;
 5.    if( uxQueueLength > ( unsigned portBASE_TYPE ) 0 )
 6.    {
 7.       pxNewQueue = ( xQUEUE * ) pvPortMalloc( sizeof( xQUEUE ) );
 8.       if( pxNewQueue != NULL )
 9.       {
10.          xQueueSizeInBytes = ( size_t ) ( uxQueueLength * uxItemSize ) + ( size_t ) 1;
11.          pxNewQueue->pcHead = ( signed portCHAR * ) pvPortMalloc( xQueueSizeInBytes );
12.          if( pxNewQueue->pcHead != NULL )
13.          {
14.             pxNewQueue->pcTail = pxNewQueue->pcHead + ( uxQueueLength * uxItemSize );
15.             pxNewQueue->uxMessagesWaiting = 0;
16.             pxNewQueue->pcWriteTo = pxNewQueue->pcHead;
17.             pxNewQueue->pcReadFrom = pxNewQueue->pcHead + ( ( uxQueueLength - 1 ) * uxItemSize );
18.             pxNewQueue->uxLength = uxQueueLength;
19.             pxNewQueue->uxItemSize = uxItemSize;
20.             pxNewQueue->xRxLock = queueUNLOCKED;
21.             pxNewQueue->xTxLock = queueUNLOCKED;
22.             vListInitialise( &( pxNewQueue->xTasksWaitingToSend ) );
23.             vListInitialise( &( pxNewQueue->xTasksWaitingToReceive ) );
24.             return  pxNewQueue;
25.          }
26.          else
27.          {
28.             vPortFree( pxNewQueue );
29.          }
30.       }
31.    }
32.    return NULL;
33. }
```

5. 새로운 큐를 위한 구조체를 생성한다.

10. 큐의 정확한 바이트 단위의 크기를 계산한다. 이것은 실제로 필요한 길이보다 한 바이트 크다. 이렇게 하면 반복 비교(wrap checking)를 보다 쉽 고 빠르게 처리할 수 있다.

14. 앞서 정의된 대로 각각의 구조체 멤버를 초기화한다.

22. 이벤트 큐 역시 정상 상태로 초기화한다.

32. (에러) 만약 큐를 생성하는데 있어 메모리가 부족하다거나 하는 문제가 생기면 이곳에 도달하게 된다.

큐의 삭제

vQueueDelete() - QUEUE.C

```
1. void vQueueDelete( xQueueHandle pxQueue )
2. {
3.    vPortFree( pxQueue->pcHead );
4.    vPortFree( pxQueue );
5. }
```

3과 4에서 큐에 할당된 메모리를 해제한다.

큐에 메시지 보내기

큐에 메시지를 보낼 때 중요한 점은, xTaskResumeAll()이 불러지기 전까지는 다른 쓰레드나 ISR에서 준비 (ready) 지연 (delayed) 리스트를 수정하면 안 된다는 것이다. 다음과 같은 곳에서 준비/지연 리스트가 수정될 수 있다.

- vTaskDelay() - 이것은 스케줄러가 중지(suspended)되었을 때 호출되어서는 안 된다. (ISR에서 호출 불가)
- vTaskPrioritySet() - 접근 과정에서 크리티컬 섹션이 있다.
- vTaskSwitchContext() - 스케줄러가 중지되었을 때에는 실행되지 않는다.
- prvCheckDelayedTasks() - 스케줄러가 중지되었을 때에는 실행되지 않는다.
- xTaskCreate() - 접근 과정에서 크리티컬 섹션이 있다.
- vTaskResume() - 접근 과정에서 크리티컬 섹션이 있다.
- xTaskResumeAll() - 접근 과정에서 크리티컬 섹션이 있다.
- xTaskRemoveFromEventList - 이것은 스케줄러가 중지되었는지 확인해보고 만약 그렇다면 이벤트 리스트에서 제거된 TCB를 xPendingReadyList에 추가한다.

또 한 가지 중요한 점은 인터럽트가 여기서 수정되는 큐의 이벤트 리스트에 접근하지 못하게 하는 것이다. 다음과 같은 곳에서 이벤트 리스트의 수정이 일어난다.

- xQueueSendFromISR() - 이것은 큐의 잠금을 확인해서 큐가 현재 사용되고 있는지를 판단하고, 만약 큐가 잠겨있다면 송신 잠금 계수 (Tx. lock count)를 증가시켜서, 지금은 자료를 받아가기 위해 대기하고 있지만 큐의 잠금이 해제되면 준비 상태가 되는 태스크가 있음을 명시한다. 반면에 큐가 잠겨있지 않다면 태스크는 이벤트 리스트에서 옮겨질 수 있지만 스케줄러의 잠금이 풀릴 때 까지는 지연 리스트에서 제거되거나 준비 리스트에 추가되지 않는다.
- xQueueReceiveFromISR() - xQueueSendFromISR()에 따라서 행동한다.

xQueueSend() - QUEUE.C

```
1. signed portBASE_TYPE xQueueSend( xQueueHandle pxQueue,
                                    const void *pvItemToQueue,
                                    portTickType xTicksToWait )
2. {
3. signed portBASE_TYPE xReturn;
4. xTimeOutType xTimeOut;

5.    vTaskSuspendAll();

6.    vTaskSetTimeOutState( &xTimeOut );

7.    prvLockQueue( pxQueue );
```

5. 스케줄러를 중지시킴으로써 다른 태스크들이 큐에 접근하지 못하도록 한다.

6. 나중에 참조하기 위해 현재의 시간 상태를 보존한다. (타임아웃 계산을 위해)

7. xRxLock, xTxLock 변수를 각각 1 증가시킴으로써 큐를 잠근다. 이 두 변수의 초기값은 queueUNLOCKED = -1 이다. 이것이 0보다 크거나 같게 되어 인터럽트가 이벤트 리스트에 접근하지 못하도록 한다.

```
8.    do
9.    {
10.      if( prvIsQueueFull( pxQueue ) )
11.      {
12.         if( xTicksToWait > ( portTickType ) 0 )
13.         {
14.            vTaskPlaceOnEventList( &( pxQueue->xTasksWaitingToSend ), xTicksToWait );
15.            taskENTER_CRITICAL();
16.            {
17.               prvUnlockQueue( pxQueue );
```

```
18.                 if( !xTaskResumeAll() )
19.                 {
20.                    taskYIELD();
21.                 }

22.                 vTaskSuspendAll();
23.                 prvLockQueue( pxQueue );
24.              }
25.              taskEXIT_CRITICAL();
26.           }
27.        }
```

10. 큐가 이미 가득 찼을 경우에는 이 태스크를 대기 상태로 만들어야 한다.

12. 큐가 가득 찬 상태이다. 이 태스크를 대기 상태로 만들고 큐에 빈 공간이 생길 때 까지 기다릴 것인가 아니면 넣지 않고 그냥 떠날 것인가를 판단한다.

14. 이제 이 태스크를 xTasksWaitingToSend 이벤트 리스트에 두려고 한다. 그리고 지연이 경과하거나 큐의 저장 공간이 충분해질 때 깨어날 것이다.앞에서 상세하게 설명했듯이 어떤 것에 의해서도 변경되지 않는 이벤트 리스트나 스케줄러가 중지되고 큐가 잠긴 상태에서의 준비 리스트에는 상호배제(mutual exclusion)가 필요하지 않다.큐가 가득 찼는지의 여부를 확인한 시점 이후에 ISR이 큐에서 자료를 가져가는 수가 있다. 그런 경우에 자료는 큐에서 복사되고 큐 변수들도 갱신(update)되지만 큐가 아직 잠겨있기 때문에 이벤트 리스트는 어떤 것이 현재 대기 중인지를 알지 못한다.

15. 현재 태스크가 대기상태이므로 강제로 문맥 전환을 하게 된다. 이것은 우리가 전환하고자 하는 태스크가 자신의 문맥을 가지고 있기 때문에 크리티컬 섹션 안에서 이루어진다. 후에 대기 상태에서 벗어나 여기로 돌아올 때 정상적으로 크리티컬 섹션을 종료할 것이다.ISR이 잠겨있지 않은 무관한 큐에 이벤트를 발생시킬 가능성이 있다. 이런 경우에는 그 큐의 이벤트 리스트가 갱신되지만 준비 리스트는 변경되지 않고 그대로이다. 대신에 준비 태스크는 미결(pending) 준비 리스트에 추가된다.

17. 여기서 인터럽트가 비활성화 된 상태이기 때문에 안전하게 큐와 스케줄러의 잠금을 해제할 수 있다. 어떤 것도 잠겨진 상태로 양보가 일어나서는 안된다. 그러나 크리티컬 섹션에서는 가능하다. 미결 준비 리스트에 있는 태스크는 이 큐에서 이벤트를 기다리는 태스크가 될 수 없다. xTaskRemoveFromEventList() 에 포함된 설명을 참고하라.

18. 스케줄러를 재개할 경우 양보가 발생할 수 있다. 그럴 경우 여기서는 다시 양보할 수 있는 지점이 없다.

22. 크리티컬 섹션에서 나가기 전에 배제 접근 (exclusive access)를 보장해야만 한다.

```
28.        taskENTER_CRITICAL();
29.        {
30.           if( pxQueue->uxMessagesWaiting < pxQueue->uxLength )
31.           {
32.              prvCopyQueueData( pxQueue, pvItemToQueue );
33.              xReturn = pdPASS;

34.              ++( pxQueue->xTxLock );
35.           }
36.           else
37.           {
38.              xReturn = errQUEUE_FULL;
39.           }
40.        }
41.        taskEXIT_CRITICAL();
```

28. 큐에서의 공간 문제가 해결되어 대기 상태에서 벗어날 수 있다. 반대로 크리티컬 섹션을 빠져나왔으므로 ISR이 큐에 post 할 수도 있기 때문에 다시 (저장)공간이 없어질 수도 있다. 태스크와 ISR이 동일한 큐에 post 할 때만 이런 경우가 발생한다.

32. 여기서는 큐에 공간이 있다. 큐에 자료를 복사해 넣는다.

34. TxLock count 를 갱신해서 prvUnlockQueue로 하여금 큐에서 사용가능한 자료를 기다리고 있는 태스크가 있는지를 확인하여 알 수 있게 한다.

```
42.       if( xReturn == errQUEUE_FULL )
43.       {
44.          if( xTicksToWait > 0 )
45.          {
46.             if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) == pdFALSE )
47.             {
48.                xReturn = queueERRONEOUS_UNBLOCK;
49.             }
50.          }
51.       }
52.    } while( xReturn == queueERRONEOUS_UNBLOCK );
53.    prvUnlockQueue( pxQueue );
54.    xTaskResumeAll();

55.    return xReturn;
56. }
```

큐로부터 메시지 받기

기본적으로 xQueueSend()와 비슷하다. xQueueSend()의 설명을 참고하라.

xQueueReceive() - QUEUE.C

```
 1. signed portBASE_TYPE xQueueReceive( xQueueHandle pxQueue,
                                        void *pvBuffer,
                                        portTickType xTicksToWait )
 2. {
 3. signed portBASE_TYPE xReturn;
 4. xTimeOutType xTimeOut;

 5.    vTaskSuspendAll();

 6.    vTaskSetTimeOutState( &xTimeOut );

 7.    prvLockQueue( pxQueue );

 8.    do
 9.    {
10.       if( prvIsQueueEmpty( pxQueue ) )
11.       {
12.          if( xTicksToWait > ( portTickType ) 0 )
13.          {
14.             vTaskPlaceOnEventList( &( pxQueue->xTasksWaitingToReceive ),
                                          xTicksToWait );
15.             taskENTER_CRITICAL();
16.             {
17.                prvUnlockQueue( pxQueue );
18.                if( !xTaskResumeAll() )
19.                {
20.                   taskYIELD();
21.                }
22.                vTaskSuspendAll();
23.                prvLockQueue( pxQueue );
24.             }
25.             taskEXIT_CRITICAL();
```

```
26.          }
27.       }
28.       taskENTER_CRITICAL();
29.       {
30.          if( pxQueue->uxMessagesWaiting > ( unsigned portBASE_TYPE ) 0 )
31.          {
32.             pxQueue->pcReadFrom += pxQueue->uxItemSize;
33.             if( pxQueue->pcReadFrom >= pxQueue->pcTail )
34.             {
35.                pxQueue->pcReadFrom = pxQueue->pcHead;
36.             }
37.             --( pxQueue->uxMessagesWaiting );
38.             memcpy( ( void * ) pvBuffer,
                        ( void * ) pxQueue->pcReadFrom,
                        ( unsigned ) pxQueue->uxItemSize );

39.             ++( pxQueue->xRxLock );
40.             xReturn = pdPASS;
41.          }
42.          else
43.          {
44.             xReturn = errQUEUE_EMPTY;
45.          }
46.       }
47.       taskEXIT_CRITICAL();

48.       if( xReturn == errQUEUE_EMPTY )
49.       {
50.          if( xTicksToWait > 0 )
51.          {
52.             if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) == pdFALSE )
53.             {
54.                xReturn = queueERRONEOUS_UNBLOCK;
55.             }
56.          }
57.       }
58.    } while( xReturn == queueERRONEOUS_UNBLOCK );

59.    prvUnlockQueue( pxQueue );
60.    xTaskResumeAll();

61.    return xReturn;
62. }
```

5. 스케줄러를 중지시킴으로써 다른 태스크들이 큐에 접근하지 못하도록 한다.

6. 나중에 참조하기 위해 현재의 시간 상태를 보존한다. (타임아웃 계산을 위해)

7. xRxLock, xTxLock 변수를 각각 1 증가시킴으로써 큐를 잠근다. 이 두 변수의 초기값은 queueUNLOCKED = -1 이다. 이것이 0보다 크거나 같게 되 어 인터럽트가 이벤트 리스트에 접근하지 못하도록 한다.

10. 만약 큐에서 가져올 수 있는 메시지가 없다면 이 태스크는 대기 상태가 될 것이다.

12. 큐에 메시지가 없다. 대기 상태가 되어 기다릴 것인가 아니면 아무것도 하지 않고 빠져나갈 것인가를 결정한다.

39. lock 카운트를 증가시켜서 prvUnlockQueue로 하여금 큐에 가능한 공간이 생기기를 기다리고 있는 태스크가 있는지 확인하여 알도록 한다.

59. 이제 여기서는 더 이상 배제 접근 (exclusive access) 이 필요하지 않다.

ISR안에서 큐로 메시지 보내기

xQueueSend()와 유사하지만 여기서는 큐에 공간이 없더라도 태스크를 대기상태로 만들지 않는다. 또한 큐를 읽을 때 대기 상태가 된 태스크를 바로 깨우지 않는다. 대신 문맥 전환이 필요한지의 여부를 가리키는 플래그(flag)를 돌려준다. (즉 이 작업을 통해 더 높은 우선순위의 태스크가 깨어났다는 사실을 알려주는 것이다)

xQueueSendFromISR() - QUEUE.C

```
 1. signed portBASE_TYPE xQueueSendFromISR( xQueueHandle pxQueue,
                                            const void *pvItemToQueue,
                                            signed portBASE_TYPE xTaskPreviouslyWoken )
 2. {
 3.     if( pxQueue->uxMessagesWaiting < pxQueue->uxLength )
 4.     {
 5.         prvCopyQueueData( pxQueue, pvItemToQueue );

 6.         if( pxQueue->xTxLock == queueUNLOCKED )
 7.         {
 8.             if( !xTaskPreviouslyWoken )
 9.             {
10.                 if( !listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToReceive ) ) )
11.                 {
12.                     if( xTaskRemoveFromEventList(
                                    &( pxQueue->xTasksWaitingToReceive ) ) != pdFALSE )
13.                     {
14.                         return pdTRUE;
15.                     }
16.                 }
17.             }
18.         }
19.         else
20.         {
21.             ++( pxQueue->xTxLock );
22.         }
23.     }

24.     return xTaskPreviouslyWoken;
25. }
```

6. 만약 큐가 잠겨있다면 이벤트 리스트를 변경하지 않는다. 나중에 잠금이 해제되면 변경된다.

8. 하나의 ISR에서는 하나의 태스크만을 깨우기를 원한다. 그래서 태스크가 이미 깨어나 있는지 확인한다.

14. 대기 중인 태스크가 보다 높은 우선순위를 가지고 있을 경우 문맥 전환이 필요함을 알린다.

21. lock 카운트를 증가시킨다. 큐의 잠금을 해제하는 태스크가 그것이 잠겨있는 동안 자료가 들어왔음을 알 수 있다.

ISR안에서 큐로부터 메시지 받기

xQueueReceiveFromISR() - QUEUE.C

```
 1. signed portBASE_TYPE xQueueReceiveFromISR( xQueueHandle pxQueue,
                                               void *pvBuffer,
                                               signed portBASE_TYPE *pxTaskWoken )
 2. {
 3. signed portBASE_TYPE xReturn;

 4.     if( pxQueue->uxMessagesWaiting > ( unsigned portBASE_TYPE ) 0 )
 5.     {
 6.         pxQueue->pcReadFrom += pxQueue->uxItemSize;
 7.         if( pxQueue->pcReadFrom >= pxQueue->pcTail )
```

```
 8.       {
 9.          pxQueue->pcReadFrom = pxQueue->pcHead;
10.       }
11.       --( pxQueue->uxMessagesWaiting );
12.       memcpy( ( void * ) pvBuffer,
                  ( void * ) pxQueue->pcReadFrom,
                  ( unsigned ) pxQueue->uxItemSize );
13.       if( pxQueue->xRxLock == queueUNLOCKED )
14.       {
15.          if( !( *pxTaskWoken ) )
16.          {
17.             if( !listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToSend ) ) )
18.             {
19.                if( xTaskRemoveFromEventList(
                                  &( pxQueue->xTasksWaitingToSend ) ) != pdFALSE )
20.                {
21.                   *pxTaskWoken = pdTRUE;
22.                }
23.             }
24.          }
25.       }
26.       else
27.       {
28.          ++( pxQueue->xRxLock );
29.       }
30.       xReturn = pdPASS;
31.    }
32.    else
33.    {
34.       xReturn = pdFAIL;
35.    }

36.    return xReturn;
37. }
```

4. ISR내부에서는 대기 상태로 될 수 없다. 그러므로 가져올 수 있는 자료가 있는지 만을 살펴본다.

6. 큐에서 자료를 복사해온다.

13. 만약 큐가 잠겨있으면 이벤트 리스트를 수정하지 않는다. 대신 lock 카운트를 증가시켜서 큐의 잠금을 해제하는 태스크로 하여금 큐가 잠겨있는 동 안 ISR이 자료를 제거했다는 것을 알게 한다.

15. 하나의 ISR은 하나의 태스크를 깨워야 한다. 그래서 이미 깨워졌는지 체크한다.

21. 대기 중인 태스크의 우선순위가 높을 경우 문맥 전환을 하게 만든다.

28. lock 카운트를 증가시켜서 큐의 잠금을 해제하는 태스크로 하여금 큐가 잠겨있는 동안 자료가 제거되었음을 알게 한다.

큐에 저장된 메시지의 개수 알아보기

uxQueueMessagesWaiting() - QUEUE.C

```
1. unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle pxQueue )
2. {
3. unsigned portBASE_TYPE uxReturn;

4.    taskENTER_CRITICAL();
5.       uxReturn = pxQueue->uxMessagesWaiting;
6.    taskEXIT_CRITICAL();

7.    return uxReturn;
8. }
```

5. 단순히 큐의 멤버를 반환하는 것으로 기능을 구현하였다.

코루틴을 위한 큐 API

코루틴에서는 다음과 같이 별도로 준비된 API를 통해 큐에 접근한다.

- xQueueCRSend()
- xQueueCRReceive()
- xQueueCRSendFromISR()
- xQueueCRReceiveFromISR()

내부적으로 사용되는 기타 함수들

다음 함수들은 내부에서 큐의 연산을 위해 사용되는 함수들이다.

- prvUnlockQueue()
- prvIsQueueEmpty()
- prvIsQueueFull()
- prvCopyQueueData (macro)
- prvLockQueue (macro)

세마포어(Semaphore)

세마포어

세마포어는 보호된 변수이며 멀티프로그래밍 환경에서 제한된 공유자원에 접근하는 고전적 방법이다. 이것은 Edsger Dijkstra에 의해 고안되고, 그가 만든 THE("Technische Hogeschool Eindhoven", 아인트호벤 기술대학) 운영체제에서 처음으로 사용되었다.

세마포어의 값은 공유 자원의 개수로 초기화된다. 공유 자원이 한 개인 특수한 경우 이를 바이너리 세마포어(binary semaphore) 라고 한다. 일반적인 경우의 세마포어는 카운팅 세마포어(counting semaphore) 이다.

세마포어는 '식사 중인 철학자들 문제'(dining philosophers problem)를 해결하는 고전적인 해결법이지만 모든 교착상태(deadlock)를 완벽히 해결하지는 못한다.

세마포어는 다음 연산에 의해서만 접근 가능하다.

P V operations

```
 1. P(Semaphore s)
 2. {
 3.    wait until s > 0, then s := s-1;
 4. }

 5. V(Semaphore s)
 6. {
 7.    s := s+1;
 8. }

 9. Init(Semaphore s, Integer v)
10. {
```

```
11.     s := v;
12. }
```

3과 7에서 s에 대한 연산이 이루어지고 있는 동안에 어떤 인터럽트도 발생해서는 안 된다(atomic). 하드웨어적으로 이를 지원하든지 그렇지 않다면 인터럽트를 비활성화 시켜서 이를 구현한다.

FreeRTOS 에서의 세마포어 구현

FreeRTOS에서는 매크로를 이용해서 세마포어를 구현한다. 구현된 큐를 이용한 매크로만을 이용해서 별도의 코드 추가 없이 필요한 모든 세마포어 관련 요소를 사용할 수 있다. 기본적으로 바이너리 세마포어가 구현되어 있고, 필요하다면 카운팅 세마포어로 쉽게 확장 가능하다.

세마포어 기본 매크로 상수

이름	타입	값
semBINARY_SEMAPHORE_QUEUE_LENGTH	unsigned portCHAR	1
semSEMAPHORE_QUEUE_ITEM_LENGTH	unsigned portCHAR	0
semGIVE_BLOCK_TIME	portTickType	0

큐의 길이가 1이란 것은 바이너리 세마포어를 뜻하고, 이 큐에 데이터가 저장되는지 알 필요가 없으므로 메시지 길이는 0이 된다. 단지 큐가 차있는가 비어있는가만을 판단하면 된다.

바이너리 세마포어의 생성

vSemaphoreCreateBinary() - SEMPHR.H

```
1. #define vSemaphoreCreateBinary( xSemaphore )
2. {
3.     xSemaphore = xQueueCreate( ( unsigned portCHAR ) 1,
                                  semSEMAPHORE_QUEUE_ITEM_LENGTH );
4.     if( xSemaphore != NULL )
5.     {
6.         xSemaphoreGive( xSemaphore );
7.     }
8. }
```

세마포어의 획득

vSemaphoreTake() - SEMPHR.H

```
1. #define xSemaphoreTake( xSemaphore, xBlockTime )

2. xQueueReceive( ( xQueueHandle ) xSemaphore, NULL, xBlockTime )
```

사용되기 전에 반드시 이전에 vSemaphoreCreateBinary()를 사용하여 세마포어가 생성되어 있어야만 한다.

세마포어의 반환

vSemaphoreGive() - SEMPHR.H

```
1. #define xSemaphoreGive( xSemaphore )

2. xQueueSend( ( xQueueHandle ) xSemaphore, NULL, semGIVE_BLOCK_TIME )
```

앞에서와 마찬가지로 반드시 vSemaphoreCreateBinary()로 세마포어가 생성된 후 vSemaphoreTake()에 의해 획득된 상태이어야만 한다. 이 매크로 함수는 ISR에서는 사용될 수 없는데 ISR에서는 그 대신 xSemaphoreGiveFromISR()을 사용해야만 한다.

ISR 내부에서의 세마포어 반환

xSemaphoreGiveFromISR() - SEMPHR.H

```
1. #define xSemaphoreGiveFromISR( xSemaphore, xTaskPreviouslyWoken )

2. xQueueSendFromISR( ( xQueueHandle ) xSemaphore,
                      NULL,
                      xTaskPreviouslyWoken )
```

세마포어를 반환하면서 태스크가 깨어나 문맥 전환이 필요한 경우에는 반환값이 pdTRUE가 된다.

세마포어 예제

Example - http://www.freertos.org/a00123.html

```
 1. xSemaphoreHandle xSemaphore = NULL;

 2. void vATask( void * pvParameters )
 3. {
 4.    vSemaphoreCreateBinary( xSemaphore );

 5.    if( xSemaphore != NULL )
 6.    {
 7.       if( xSemaphoreGive( xSemaphore ) != pdTRUE )
 8.       {
 9.       }
10.       if( xSemaphoreTake( xSemaphore, ( portTickType ) 0 ) )
11.       {
12.          if( xSemaphoreGive( xSemaphore ) != pdTRUE )
13.          {
14.          }
15.       }
16.    }
17. }
```

4. 공유 자원을 보호하기 위한 바이너리 세마포어 생성

7. 여기서 세마포어를 획득하지 않은 시점에 반환을 시도하면 실패한다.

10. 세마포를 획득하려고 시도한다. 만약 획득이 불가능하면 'block'하지 않는다.

11. 이제 세마포어를 획득하였으므로 공유 자원에 대한 접근이 가능해졌다.

12. 공유자원 접근을 마치고 세마포어를 반환한다.

13. 앞에서 획득한 자원을 반환하기 때문에 실제로 반환이 실패하여 이 부분이 실행될 리는 없다.

참고 문헌

---

1. http://www.freertos.org