uC/OS 분석 자료

이 문서에서는 uC/OS와 관련된 기본적인 개념들과 커널의 기본적 구조를 소개한다.

1. Real-Time 커널에 관한 기본 개념들

Critical Section

이 부분의 코드는 실행되면 인터럽트가 일어날 수 없다. 따라서 critical code를 실행하기 전에 인터럽트를 금지시키고 실행이 끝나면 인터럽트를 다시 허용하도록 해야 한다.

Resource

태스크가 이용하는 entity. I/O 장치나 변수, 구조체, 배열 등이 될 수 있다.

Shared Resource

하나 이상의 태스크가 동시에 이용할 수 있는 리소스이다. data corruption을 막기 위해서 각 태스크는 미리 해당 자원에 대한 exclusive access를 얻어야 한다.

Multitasking

몇 개의 태스크 사이에서 CPU에 대한 스케줄링과 스위칭을 하는 것이다.

Task

Thread라고도 불리며 일정기간 CPU 자원을 차지하는 간단한 프로그램을 의미한다.
각 태스크에게는 priority가 부여되며 각 태스크만의 CPU 레지스터와 스택 영역을 할당받는다.

* 태스크가 가질 수 있는 상태

State	Description
DORMANT	메모리에는 존재하지만 수행될 수 있는 상태는 아님
READY	수행될 수는 있지만 현 태스크보다 priority가 낮을 때
RUNNING	CPU를 실제로 사용하고 있을 때
DELAYED	얼마의 시간동안 작업이 중지
WAITING FOR AN EVENT	event의 발생을 기다릴 때 (ex: I/O 연산의 종료, shared resource가 사용가능해질 때, timing pulse 등)
INTERRUPTED	인터럽트가 일어나고 CPU가 인터럽트 서비스를 수행할 때

Context Switch or Task Switch

현재 태스크의 내용(CPU register)을 현 태스크의 context storage에 보관하고 다음 태스크의 내용을 꺼내 와서 다음 태스크를 진행하는 것이다.

Kernel

여러 태스크의 CPU 시간 배분과 태스크간의 통신 등을 감독하는 부분이다. 이것의 기초를 이루는 것이 바로 context switching이다.

Scheduler

다음에 수행될 태스크의 순서를 결정한다. 대부분 priority에 기반하는데 이러한 방식의 커널에는 선점형(preemptive)과 비선점형(non-preemptive)이 있다.

Preemptive Kernel

uC/OS가 바로 이 형태의 커널이다. 즉각적인 응답이 중요할 때 바로 이러한 형태의 커널을 사용한다.
실행될 준비가 된 가장 높은 priority를 갖는 태스크는 언제나 CPU 제어권을 받게 된다. 더 높은 priority의 태스크가 실행될 준비가 되면 현 태스크는 preempt(suspend)되고 더 높은 priority의 태스크가 즉시 CPU 제어권을 받게 된다. ISR(interrupt service routine) 도중에 더 높은 priority의 태스크가 ready 상태가 되면 ISR이 끝날 때에는 인터럽트된 태스크가 suspend되고 새로운 태스크가 수행되게 된다.

Reentrancy

Reentrancy 함수는 하나 이상의 태스크가 사용하더라도 data corruption이 일어날 염려가 없는 함수이다. 이것은 어느 때이든지 인터럽트될 수 있으며 지역변수(CPU 레지스터, 스택 내의 변수)를 사용하거나 전역 변수 사용시 데이터를 보호하게 된다.

Dynamic Priorities

응용 프로그램의 실행 중에 태스크의 priority가 지속적으로 변하는 것이다. uC/OS에서의 priority가 이러한 형태이다.

Priority Inversions

높은 priority의 태스크가 어떤 자원을 사용할 준비가 되어 있을 때 낮은 priority의 태스크가 자원을 놓지 않을 때의 문제를 말한다. 이 문제를 해결하기 위해서는 특정 자원에 접근한 낮은 priority 태스크의 priority를 높여 주고 끝났을 때 원래의 값을 복원한다.

Semaphores

Semaphore는 다음과 같은 용도를 갖는다.
- 공유된 자원의 사용을 제어한다. (상호배제 : mutual exclusion)
- event의 발생을 알린다.
- 두 태스크의 동작을 동기화한다.

uC/OS는 binary semaphore와 counting semaphore를 모두 지원한다.
Semaphore와 연관된 연산은 다음과 같은 것이 있다.
INITIALIZE (or CREATE)
Semaphore의 초기값을 준다. 처음 waiting list는 비어 있다.
WAIT (or PEND)
Semaphore를 요구하는 태스크가 WAIT 연산을 행하면 semaphore 값이 감소하다가 0이 되면 해당 태스크는 waiting list에 들어가게 된다.
SIGNAL (or POST)
태스크는 이 연산을 사용하여 semaphore를 놓게 된다.
uC/OS에서는 semaphore를 기다리는 태스크 중 가장 높은 priority를 갖는 것이 semaphore를 갖도록 한다.

Mutual Exclusion

상호 배제를 제공하는 데에는 bianry semaphore가 사용된다. 그러나 semaphore가 캡슐화된 경우에는 태스크가 자원에 접근할 때 semaphore의 획득에 관하여 알지 못할 수 있다.
Counting semaphore는 보다 일반적인 형태로서, 자원이 동시에 하나 이상의 태스크에 의하여 사용될 때 사용될 수 있다.

Real-time kernel의 경우 코드의 critical section에 들어가면 인터럽트를 disable시킨다.

Synchronization

semaphore를 사용하여 어떤 태스크를 ISR 또는 다른 태스크와 동기화시킬 수 있다. 이 경우에 semaphore는 상호 배제를 위하여 사용된 것이 아니라 이벤트의 발생을 알리기 위한 플래그로서 사용된 것이다. 이렇게 sempahore를 사용하는 것을 unilateral randevous라고도 한다.
이 경우 sempahore의 초기값은 0이다. 예를 들어 태스크가 I/O 연산을 시작하고 semaphore를 기다릴 때 I/O 연산이 끝나면 ISR이나 다른 태스크가 semaphore로 신호를 보내고 태스크가 계속 진행되게 된다.
커널이 event를 전달하는 태스크의 선택 :
이벤트 발생을 기다리고 있는 태스크 중 가장 높은 우선 순위의 것

Message Mailboxes

이것은 message exchange라고도 하며, 대개 pointer size 변수이다.
커널에 의하여 제공되는 서비스를 사용하여 태스크나 ISR이 메시지(포인터)를 메일박스에 보내게 되며 하나 이상의 태스크가 이 메시지를 받을 수 있다.
메시지가 메일박스에 놓여지게 되면 uC/OS에서는 메시지를 기다리는 태스크 중 가장 높은 우선 순위를 가지는 태스크에 메시지를 전달하게 된다.

Message Queues

Message queue는 메일박스의 배열로 FIFO 형태로 되어 있다.
비어 있는 큐로부터 메시지를 받으려고 하는 태스크는 서스펜드되어 메시지가 올 때까지 waiting list에 놓여지게 된다.
uC/OS에서는 메시지를 기다리는 태스크 중 가장 높은 우선 순위를 가지는 태스크에게 메시지를 전달한다.

Interrupts

인터럽트 발생이 확인되면 CPU는 context를 저장하고 Interrupt Service Routine(ISR) 서브루틴으로 분기한다.
ISR이 완료되면 프로그램은 실행 준비가 된 태스크 중 가장 우선 순위가 높은 것으로 돌아간다.
Interrupt Nesting : ISR 진행 도중 다시 interrupt 발생

Interrupt Latency, Response and Recovery

Interrupt Latency : Maximum amount of time
The Response Time :
Interrupt latency + Time to save CPU's context + Time to determine if a higher priority task is ready

Non-Maskable Interrupts (NMIs)

인터럽트 중 최대한 빨리 서비스되어야하는 것이나 커널에 의한 latency를 가질 수 없는 것의 경우에 해당된다. 이것은 disabled될 수 없으며 interrupt latency나 response, recovery 타임이 최소이다.
ex) saving important information during a power down
Parameters : 전역변수와 이들 변수들의 크기가 한꺼번에 read/write되어야 한다.

Clock Tick

주기적으로 발생하는 특별한 인터럽트이다. 이것은 태스크가 정수 단위의 클럭 수만큼 지연

되도록 한다. 그리고 태스크가 이벤트 발생을 기다릴 때 타임아웃을 제공한다.

2. 커널 구조

Critical Sections

uC/OS는 코드 중 critical sction에 들어갔을 때 배타적 접근을 보장하기 위해서 이 코드가 수행되기 전 인터럽트를 금지시키고 수행이 완료되었을 때 인터럽트를 다시 허용한다.
이것은 다음과 같은 매크로로 구현된다.

OS_ENTER_CRITICAL()
OS_EXIT_CRITICAL()

이 코드는 language extension이나 inline assembly를 사용하여 구현된다.
*제한 사항
- uC/OS에서 실행되는 태스크는 개별적인 priority 번호를 가지고 있어야 한다.
- NMI에서는 uC/OS system call이 만들어질 수 없다 : uC/OS는 코드의 critical section을 수행할 때 인터럽트를 금지시키므로.

Tasks

태스크는 무한 루프 함수 또는 실행이 끝나면 자신을 제거하는 함수이다. 무한루프는 인터럽트에 의하여 선점(preemted)되어 높은 우선 순위의 태스크가 실행되게 된다.
uC/OS services:
OSTaskDel(), OSTimeDly(), OSSemPend(), OSMboxPend(), OSQPend()

태스크의 선언

```
/*-----------------------------------*/
void far Task(void *data)

        User code;
        OSTaskDel(task's priority);

/*-----------------------------------*/
void far Task(void *data)

        while(1)
                Optional user code;
                Call uCOS service to DELAY or PEND;
                Optional user code;


/*-----------------------------------*/
```

각 태스크는 0에서 62까지의 우선 순위 값을 할당받는다. level이 낮으면 priority는 높아지며 이 번호는 또한 태스크 id로서의 역할을 하기도 한다. 이 번호는 OSTaskChangePrio()와 OSTaskDel() 커널 서비스에 의하여 사용된다. uC/OS는 항상 실행 준비가 된 태스크 중 가장 높은 우선 순위의 것을 실행한다.

Task States

DORMANT : 저장장치에는 있으나 사용가능하지는 않은 상태.
READY : OSTaskCreate() call에 의하여 생성되어 사용 가능해진 상태. 태스크는 multitasking이 시작되기 전이나 태스크를 실행할 때 동적으로 생성 가능하다. 생성된 태스크가 그것을 생성한 태스크보다 우선 순위가 높으면 생성된 태스크는 곧바로 CPU의 제어를 받는다. OSTaskDel()을 호출하면 DORMANT 상태로 돌아간다.
RUNNING : OSStart() 호출로 생성된 태스크 중 가장 우선 순위가 높은 것이 실행된다.
DELAYED : OSTimeDly() 호출에 의하여 지정 시간만큼 자신을 지연시키는 상태로 다음 우선 순위의 태스크가 실행된다. 시간이 경과하면 OSTimeTick() 호출에 의하여 실행될 준비가 된다.
PENDING : 실행중인 태스크가 이벤트 발생을 기다리는 상태. OSSemPend(), OSMboxPend(), OSQPend() 호출에 의하여 이 상태로 들어간다. 이 경우에도 다음 우선순위의 태스크가 실행되며, 이벤트가 발생하면 실행될 준비가 된다. 이벤트 발생 신호는 다른 태스크나 ISR에 의하여 이루어진다.
INTERRUPTED : 실행중인 태스크는 인터럽트를 금지하지 않는 한 항상 이 상태로 들어갈 수 있다. 인터럽트가 발생하면 태스크는 suspend되고 ISR이 실행된다. ISR로부터 돌아오기 전, uC/OS는 인터럽트된 태스크가 가장 높은 우선 순위를 갖고 있는지 판단하여 우선 순위가 바뀌었을 경우에는 가장 높은 우선 순위의 태스크를 실행한다.
* 모든 태스크가 이벤트 발생을 기다리거나 delay되고 있을 때에 uC/OS는 OSTaskIdle()을 실행한다.

Task Control Blocks

태스크가 생성되면 Task Control Block인 OS_TCB를 할당받는다. 이것은 RAM에 상주하면서 태스크의 상태를 관리하는 데 쓰인다.
OSTCBStkPtr : 태스크의 스택 꼭대기를 가리키는 포인터. 이것은 어셈블리 코드의 context switching code부분에서 접근하는 유일한 필드이다. 이것을 맨 위에 배치하는 것이 어셈블리 언어에서 접근하도록 하는 데 좋다.
OSTCBStat : 태스크의 상태. 이 값들은 ucos.c에 있다.
OSTCBPrio : 태스크 priority값. 우선 순위가 높으면 작은 값을 가진다.
OSTCBDly : 태스크가 지연될 필요가 있을 때나 일정한 timeout을 두고 이벤트 발생을 기다릴 때 사용된다. 이것은 지연 혹은 기다릴 시간을 clock tick 수로 나타낸 것이다. 0일 때에는 지연되지 않거나 timeout이 없는 상태이다.
OSTCBX, OSTCBY, OSTCBBitX, OSTCBBitY : 이 값은 태스크가 생성되거나 우선 순위가 바뀔 때 다음과 같이 사용된다.

```
OSTCBX      = priority & 0x07;
OSTDBBitX   = OSMapTbl[priority & 0x07];
OSTCBY      = priority >> 3;
OSTCBBitY   = OSMapTbl[priority >> 3];
```

OSTCBNext, OSTCBPrev : OS_TCB를 양방향으로 링크하는데 사용된다. 이 결합은 OSTimeTick()이 각 태스크의 OSTCBDly 필드값을 갱신하는 데 사용한다. OS_TCB는 태스크 생성시 링크되고 태스크가 제거될 때 링크도 제거된다.
OSTCBEventPtr : event control block의 포인터이다.

태스크 및 OS_TCB의 최대값은 사용자 코드에서 선언된다. 모든 OS_TCB는 OSTCBTbl[]에 놓여지며 이 테이블의 엔트리 수는 uC/OS가 관리할 수 있는 태스크의 수를 나타낸다. 그리고 가외의 OS_TCB 한 개는 idle 태스크를 위한 것이다. 태스크가 생성되면 OSTCBFreeList가 가리키고 있던 OS_TCB가 할당되고 OSTCBFreeList는 체인 내의 다음 OS_TCB를 가리키게 된다. 태스크가 제거되면 OS_TCB는 다시 OS_TCB 체인으로 돌아온

다.

Creating a Task

태스크는 OSTaskCreate() 호출에 의하여 생성된다. 이것은 프로세서에 따라 코드가 다르다. 태스크는 멀티태스킹 시작 전이나 실행 중에 생성될 수 있으나 ISR에 의해서 생성될 수는 없다. 이 함수의 인자는 다음과 같다.
task : 태스크 코드에 대한 포인터
data : 사용자 정의 데이터 영역에 대한 포인터. 이것은 여러 태스크가 같은 코드를 사용할 때 유용하다. 이 경우 태스크는 datark 무엇을 가리키는 것인지 알아야 한다.
pstk : 태스크의 스택 영역에 대한 포인터. 스택의 크기는 태스크 요구 사항에 의하여 정의된다. 스택 크기에는 태스크의 지역 변수, 내포된 함수, 인터럽트를 위한 필요에 대한 byte 수가 연관되어 있다.
p : 태스크의 priority
태스크가 생성되기 전에 OSTaskCreate()는 해당 priority의 태스크가 이미 있는지 확인한다. 해당 우선 순위의 태스크가 사용 가능하면 uC/OS는 인터럽트가 발생한 것처럼 스택을 초기화하고 CPU register를 저장한다.
OsTaskCreate()는 OSTCBInit()을 호출하게 되는데 이것은 free OS_TCB 중에서 하나를 얻어 오는 일을 한다. OS_TCB가 다 쓰이면 에러 코드를 리턴한다. 태스크에 해당하는 OS_TCB의 pointer는 OSTCBPrioTbl[]에 놓여지는데 이것은 태스크의 priority를 index와 같이 사용하는 것이다. 이 OS_TCB는 양방향으로 링크된 OSTCBList에 삽입되는데 이것은 가장 최근에 생성된 태스크의 OS_TCB를 가리킨다. 그리고 나서 태스크는 실행 준비 된 태스크의 리스트에 삽입된다.
태스크가 다른 태스크에 의하여 실행 중 생성되면 스케줄러가 호출되어 생성된 태스크가 그것을 생성한 태스크보다 높은 우선 순위를 갖는지 검사한다.

Deleting a Task

태스크는 OSTaskDel()을 호출하여 DORMANT 상태로 돌아간다. 전달되는 인자는 삭제될 태스크의 priority이다. 이것은 태스크의 우선 순위가 OS_LO_PRIO인지를 검사하여 idle 태스크가 삭제되는 것을 막는다. OSTCBDel()은 삭제될 태스크가 생성되어 있는 것인지 검사한다. 생성된 것이 확인되면 우선 ready 리스트에서 삭제되고 OS_TCB는 체인에서 분리된다. 태스크의 OS_TCB의 OSTCBEventPtr 필드가 0이 아니면 삭제될 태스크는 이벤트를 기다리던 상태였으므로 이벤트 대기 리스트에서 삭제된다. OS_TCB는 free OS_TCB 리스트로 돌아오고 최종적으로 다음 우선 순위의 태스크가 실행 준비된다.
*** OSTaskDel()은 ISR이 호출할 수 없다.

Task Scheduling

가장 높은 우선 순위를 가진 것으로 다음에 실행될 준비가 된 태스크를 찾는 것은 scheduler에 의하여 이루어진다. 태스크 스케줄링은 OSSched()에 의하여 이루어진다. 우선 순위 값이 63이면 항상 idle 태스크로 할당된다.
uC/OS의 스케줄링 타임은 어플리케이션에서 생성된 태스크 수와 상관이 없다. 실행될 준비가 된 각 태스크는 OSRdyGrp와 OSRdyTbl[8]로 된 ready list에 놓여진다. OSRdyGrp의 각 bit는 그룹 내의 어떤 태스크든 항상 실행 될 준비가 되었음을 나타낸다. 태스크가 실행 준비가 되면 ready table인 OSRdyTbl[8]의 해당 비트를 설정한다. 어느 priority의 태스크가 다음에 실행될 것인지를 결정하기 위해 OSRdyTbl[8]에서 bit를 설정한 가장 낮은 priority 번호를 찾는다.
OSRdyGrp와 OSRdyTbl[8]의 관계는 다음과 같다.
Bit 0 in OSRdyGrp is 1 when any bit in OSRdyTbl[0] is 1.
...
Bit 7 in OSRdyGrp is 1 when any bit in OSRdyTbl[0] is 1.

태스크를 ready list에 놓기 위한 코드는 다음과 같다.

```
OSRdyGrp          |= OSMapTbl[p>>3];
OSRdyTbl[p>>3]          |= OSMapTbl[p & 0x07];
(p : priority)
```

태스크 우선 순위의 아래 세 비트는 OSRdyTbl[8]의 bit 위치를 나타내는 것이고 다음 세 비트는 OSRdyTbl[8]에 대한 index이다.
태스크는 프로세스를 뒤집음으로써 ready list에서 제거된다.

```
if((OSRdyTbl[p>>3] &= ~OSMapTbl[p & 0x07]) == 0)
        OSRdyGrp &= ~OSMapTbl[p>>3];
```

이 코드는 OSRdyTbl[8]의 ready bit를 clear하고 그룹 내의 모든 태스크가 실행될 준비가 되지 않았을 경우에 OSR요Grp의 bit를 clear한다.
OSUnMapTbl : priority resolution table
실행준비가 된 태스크 중 가장 높은 우선 순위를 갖는 것의 priority를 찾는 코드는 다음과 같다.

```
y = OSUnMapTbl[OSRdyGrp];
x = OSUnMapTbl[OSRdyTbl[y]];
p = (y << 3) + x;
```

OSSched() : 가장 높은 우선 순위의 태스크가 현 태스크가 아닌지 검사. 이 코드의 모든 부분은 critical section이다.
OS_TASK_SW() : context switching을 수행한다.

LOCK / UNLOCK

OSSchedLock()은 OSSchedUnlock()이 호출될 때까지 태스크 스케줄링을 막는 역할을 하며 이 두 함수는 함께 쓰인다. 여기서 OSLockNesting은 nesting을 위해서 OSShedLock()이 호출된 횟수를 기억하고 있다.
OSSchedUnlock()은 OSLockNesting이 0이 될 때 스케줄러를 호출한다. OSSchedLock()이 호출된 동안에는 어플리케이션이 현재 태스크의 실행을 suspend시키는 시스템 콜을 호출할 수 없다. (OSTimeDly, OSSemPend(), OSMboxPend(), OSQPend())
Highest Priority : Interrupts, Task that locked the scheduler
Lowest Priority : Highest priority task ready to run

Changing Priority

우선 순위의 변환은 OSTaskChangePrio() 호출에 의하여 이루어진다. 이때 요구된 우선 순위 값은 할당되어 있지 않은 상태이어야 한다. 우선 순위의 변환은 다음과 같은 과정을 거친다.
1) 태스크가 실행 준비가 되면 ready list에서 제거되고 태스크가 새 우선 순위를 할당받으면 ready list에 놓여진다. 태스크가 실행 준비가 되지 않으면 태스크는 새 우선 순위에서 실행될 준비가 되지 않는다.
2) 태스크가 이벤트 발생을 기다리면 태스크는 event waiting list에서 제거되고 새 우선 순위의 waiting list에 놓여진다.
3) 태스크가 OS_TCB 체인에서 제거되어 새 우선 순위를 받는 동안 OSTimeTick()에 의하여 태스크가 실행 준비가 되는 것을 방지한다.
인터럽트가 인에이블되면 OS_TCB의 OSTCBX, OSTCBBitX, OSTCBY, OSTCBBitY가 재계산된다.

이 함수는 uC/OS 함수 중 가장 오랫도안 인터럽트를 금지시킨다.

Delaying a task

OSTimeDly()를 사용하여 주어진 clock tick수만큼 태스크를 지연시킨다. 이것은 ready list에서 태스크를 제거하고 OSTCBDly를 tick수만큼 로드한다.

3. 인터럽트 처리

ISR은 어셈블리 루틴으로 작성되어야 한다.
OSIntEnter() 함수는 커널 시스템 호출이 있기 전에 반드시 불려져야 하는 것으로 인터럽트 nesting 레벨을 추적하는 역할을 한다. OSIntEnter()가 리턴되면 사용자 ISR 코드가 실행된다.
OSIntExit() 함수는 인터럽트 nesting 레벨을 1 줄이고 ISR의 종료를 표시한다. 그리고 인터럽트된 태스크보다 높은 우선 순위를 갖는 태스크가 있을 경우 그 태스크로 리턴한다.

Clock Tick

Clcok tick은 주기적으로 발생하는 인터럽트로 일정 수의 tick동안 태스크가 실행을 중단(suspend)하거나 이벤트의 발생을 기다리도록 한다. 이것의 주기가 빨라지면 그만큼 오버헤드도 커진다.
tick ISR은 OSTimeTick()을 호출하는데 이것은 OS_TCB의 OSTCBDly값이 nonzero값이면 감소시킨다. 이것이 0이 되면 태스크는 실행될 준비가 된다.
OSTimeTick()은 태스크 레벨에서 호출될 수 있는데 이렇게 하려면 priority를 5로 할당할 수 있다. (priority 0에서 4까지는 예약됨)

```
TimeTask()

        while(1)
                OSSemPend(...);
                OSTimeTick();
```

tick interrupt가 발생하였다는 것을 알리기 위하여 semaphore(0으로 초기화)를 생성할 수 있다.
OSTimeTick()은 power updIgn로 OSTime이라는 32bit값을 더해 나가는데 이것은 OSTimeGet()함수를 이용하여 얻어낼 수 있다. 이 값의 설정은 OSTimeSet()함수를 이용한다.
1

4. Communication, Synchronization & Coordination

POST : signal the occurrence of the event
PEND : wait for the event to occur

Event Control Blocks (ECB)

이벤트의 상태에 관한 정보를 관리한다.
a) 이벤트 자신의 상태:
- 세마포어 counter
- mailbox message
- queue에 대한 message queue

b) 이벤트 발생을 기다리는 태스크들의 리스트

- ECB의 필드
OSEventGrp : OSRdyGrp과 유사하나 8개 태스크의 그룹 중 하나가 이벤트 발생을 기다린다는 것을 가리킨다는 차이점이 있다.
OSEventTbl[8] : OSRdyTbl[]과 유사하나 이벤트 발생을 기다리는 bit map을 포함한다는 차이점이 있다.
OSEventCnt : 세마포어 count를 저장한다.
OSEventPtr : mailbox 메시지나 queue 데이터 구조에 대한 포인터를 포함한다.

이벤트 발생을 기다리는 각 태스크는 OSEventGrp와 OSEventTbl[8]의 두 변수로 된 waiting list에 놓여진다. 태스크의 priority는 OSEventGrp 내에서 8개 단위로 그룹지어진다. 여기서 각 bit는 그룹내의 태스크 중 하나가 event를 기다린다는 것을 나타낸다.
- relationship between OSEventGrp and OSEventTbl[8]
Bit 0 in OSEventGrp is 1 when any bit in OSEventTbl[0] is 1
...
Bit 7 in OSEventGrp is 1 when any bit in OSEventTbl[0] is 1

태스크가 이벤트 발생을 기다릴 때 다음과 같은 코드를 수행한다.

```
OSEventGrp              |= OSMapTbl[p >> 3];
OSEventTbl[p >> 3]      |= OSMapTbl[p & 0x07];
```

(p : task's priority)

이벤트가 발생하면 기다리던 태스크 중 가장 높은 우선순위의 것이 waiting list에서 제거된다. 깨어날 태스크의 우선 순위를 얻어내는 코드는 다음과 같다.

```
y = OSUnMapTbl[OSEventGrp];
x = OSUnMapTbl[OSEventTbl[y]];
p = (y << 3) + x;
```

할당할 ECB의 수는 응용프로그램에서 필요로 하는 세마포어, mailbox, queue의 수에 따른다. ECB의 총 수는 UCOS.H의 #define OS_MAX_EVENTS로 정의된다.
OSInit() (uC/OS 초기화)가 호출될 때 모든 ECB는 free ECB의 singly linked list로 링크된다. semaphore, mailbox, queue가 생성되면 ECB는 이 리스트에서 제거되어 초기화된다. (semaphore, mailbox, queue는 delete될 수 없으므로 한 번 제거된 ECB는 되돌아올 수 없다.)

SEMAPHORES

uC/OS의 세마포어는 16bit signed 정수이다. 이것을 제어할 수 있는 함수는 다음과 같다.

1) OSSemCreate()
2) OSSemPend()
3) OSSemPost()

OSSemCreate()는 세마포어가 사용할 ECB를 할당하고 세마포어의 초기값을 부여한다. 이 함수의 리턴값은 세마포어에 할당된 ECB의 포인터이다. 이 포인터는 세마포어의 handle로 사용되므로 어플리케이션 내의 변수에 할당되어야 한다. ECB가 모두 사용되면 NULL 포인터가 리턴된다. ECB의 OSEventCnt 필드는 세마포어의 현재 값을 지니고 있고 -63에서 32767까지의 값을 가질 수 있다. 양수값은 자원에 접근할 수 있는 태스크 수나 이벤트 발생 횟수를 나타낸다. 0은 자원이 사용불능이거나 이벤트가 발생하지 않았음을 나타낸다. 음

수값은 사용 가능해지기를 혹은 이벤트 발생을 기다리는 태스크 수를 나타낸다.
OSSemPend()를 세마포어가 양수일 때 호출하면 세마포어 값을 1 줄이고 리턴한다. 세마포어 값이 0혹은 음수일 때에는 세마포어 값을 줄이고 호출한 태스크를 세마포어에 대한 waiting list에 놓는다. 태스크가 세마포어를 기다리는 동안 rescheduling이 일어나서 실행 준비가 된 다음 우선 순위의 태스크가 CPU의 제어권을 갖는다. timeout의 발생은 error로 간주되며 이에 따라 적절한 대처를 하도록 할 수 있다. timeout 값이 0인 것은 세마포어를 마냥 기다리는 것을 의미한다. pending된 태스크가 재개될 때에는 (OSSched() returns to OSSemPend()) 태스크 상태를 검사하여 태스크가 세마포어를 기다리고 있는지 판단한다. 타임아웃의 경우, OSTCBStat가 계속 OS_STAT_SEM으로 설정되어 있으면 감지되는데 이 경우 태스크는 세마포어의 waiting list에서 제거되고 호출자는 타임아웃의 발생을 알게 된다.
*** OSSemPend()는 ISR이 호출할 수 없다.
OSSemPost()를 호출함으로써 semaphore 신호가 이루어진다. 세마포어가 0 혹은 양수값을 가지면 세마포어 값을 증가시키고 리턴한다. 한편 음수이면 태스크는 세마포어 신호를 기다린다. 이 경우 OSSemPost()는 가장 높은 우선순위의 태스크를 waiting list에서 제거하고 OSEventTaskResume()을 호출하여 실행 준비가 되도록 한다. 태스크의 OS_TCB 중 OSTCBDly 필드가 클리어되는데 이것은 OSTimeTick()이 태스크를 지연시키는 것을 막는다.

QUEUES

Queue는 mailbox와 비슷하다. queue를 제어하는 함수는 다음과 같은 것이 있다.
1) OSQCreate()
queue가 사용할 ECB를 할당한다. 이 함수는 ECB의 포인터를 리턴하는데 이것은 응용 프로그램에서 핸들로서 사용하게 된다.
이 함수의 두 인자 중 start는 메시지가 놓일 영역의 시작점을 가리키는데 void의 pointer 배열 형태를 하고 있다. size는 이 배열의 크기를 나타낸다.
2) OSQPend()
이 함수가 호출되었을 때 큐에 메시지가 있으면 OSQOout이 메시지를 가리키게 하여 호출한 태스크에 넘겨 준다. 큐가 비어 있으면 호출한 태스크를 wating list에 포함시킨다. 메시지의 전달은 mailbox에서와 마찬가지로 우선순위에 따라 배당된다. 이 함수 역시 ISR이 사용할 수 없다.
현재 큐의 상태를 관리하기 위하여 queue control block을 사용하는데 이것은 UCOS.H에 있는 OS_Q 데이터구조에 정의되어 있다.
OSQPtr : free QCB에 링크시키는 데 사용되나 한 번 큐가 할당되면 이 필드는 필요가 없다. (한 번 할당된 큐는 계속 할당된 채로 있으므로)
OSQStart : 메모리 내의 메시지 영역의 시작점을 나타낸다.
OSQEnd : 할당된 큐 영역의 끝을 나타낸다.
OSQIn : 다음 메시지가 삽입될 위치이다.
OSQOut : 큐에서 나올 메시지를 가리킨다.
OSQSize : 메시지 저장 영역의 크기를 나타낸다.
OSQEntiries : 메시지 큐의 엔트리 수를 나타낸다.

3) OSQPost()
이 함수를 호출하면 메시지가 태스크로 전달된다. 단 큐가 꽉 차 있을 때에는 에러가 리턴된다.

5. 초기화 및 설정

Initialzation

초기화는 OSInit() 함수를 호출함으로써 이루어진다. 단, UCOS.H 파일에 다음과 같은 것들이 설정되어 있어야 한다.
OS_IDEL_TASK_STK_SIZE : idel 태스크의 크기를 byte로 나타낸 것이다.
OS_MAX_TASKS : uC/OS가 제어할 수 있는 태스크의 최대 수
OS_MAX_EVENTS : 응용 프로그램이 생성할 최대 ECB수를 나타낸다.
OS_MAX_QS : 응용 프로그램이 생성할 메시지 큐의 최대 수를 나타낸다.

OSInit() 호출 후 OSStart() 함수를 호출하게 되면 멀티태스킹이 시작된다. 이것이 호출되기 전에 하나 이상의 태스크가 생성되어 있어야 한다. 이 함수는 highest priority를 결정하고 OSTCBHighRdy를 설정한다. 멀티태스킹이 시작되었다는 것은 OSRunning을 TRUE로 설정함으로써 나타낸다. 가장 높은 우선 순위를 갖는 태스크를 로드하고 실행하기 위해서는 OSStartHighRdy() 함수를 호출하게 되는데 이 함수는 어셈블리로 짜여진 것이다.

Configuration

uC/OS의 설정은 다음과 같은 단계를 거친다.
1) UCOS.H 내의 #define값을 고친다.
    OS_IDLE_TASK_STACK_SIZE
    OS_MAX_TASKS
    OS_MAX_EVENTS
    OS_MAX_QUEUES
    uCOS
2) OSTimeTik()을 호출하는 ISR을 작성한다.
3) context switch 코드인 OS_TASK_SW()에 대하여 인터럽트 벡터를 할당하고 설정한다.
4) tick ISR에 대한 인터럽트 벡터를 할당하고 설정한다.
5) OSStart()를 호출하기 전에 main()에서 OSInit()을 호출한다.
6) semaphore, mailbox, queue를 생성한다.
7) 태스크 스택 영역을 할당한다.
8) 태스크 중 하나를 생성한다.
9) 멀티태스킹을 행할 준비가 되면 OSStart()를 호출한다.