uC/OS-Ⅱ 뛰어넘기(Ⅴ)

Synchronization과 Intertask Communication

Task는 독립적으로 수행되는 개체이다. 이러한 특성을 갖는 Task들도 서로 정보를 보내주고 어떠한 신호와 흐름에 의해 톱니바퀴가 맞물려 돌듯이 흐름을 타고 동작을 하게 된다. 이번 회에서는 이러한 신호와 흐름을 제어하기 위한 Synchronization과 Intertask Communication에 대해서 공부해보도록 하자.

글: 김대홍/CyberLab 실장, 삼성첨단기술연수소 RTOS 강사
redizi@armkorea.com

Synchronization

우리가 지금까지 공부해온 Task의 성질을 생각해보면 각 Task는 독립적으로 동작을 하며, 여러 개의 Task가 동시에 동작하는 것처럼 보이게 되는 특성을 가지고 있다. 우리는 이것을 멀티태스킹이라고 불렀다.

또한 Task는 문제해결을 위한 단위로 구성하게 된다. 여기까지는 문제가 전혀 없다. 하지만, 우리가 지금까지 배워온 내용들만 가지고 바로 응용프로그램을 짜다보면 여러분들은 중대한 문제에 부딪히게 된다.

키 입력을 받고 키가 눌릴 때마다 소리가 나는 계산기 프로그램을 작성한다고 생각해보자. 여러분들은 이 계산기 프로그램의 구조를 RTOS를 이용하여 어떻게 프로그램을 작성할까?

개개인에 따라서 다르겠지만 필자가 다루고자 하는 내용을 쉽게 설명하기 위해 키입력 기능과 사운드 기능을 위해 별도의 Task를 구성하고, 나머지 기능은 계산알고리즘이란 Task를 구성했다(그림 1).

각자 이러한 구성을 가지고 한번 직접 프로그램을 구성해보면 좀더 쉽게 여기서 다루고자 하는 문제점에 접근할 수 있을 것이다.

여기서 우리가 구현하려는 기능은 키입력을 받았을 때 사운드가 출력이 되는 것이다. 물론 키가 눌려지자마자 소리가 나야 한다.(약 2년 전에 필자의 휴대용전화기는 키를 누르면 1-2초 후에 소리가 나서 눌렸는지 안눌렸는지 확인이 안돼서 같은 문자를 여러 번 누르는 경우가 많아 정신적 치명상을 받은 경우가 있었다.) 즉, 이 2개의 Task는 서로 연동이 돼서 키입력 Task가 동작하자마자 사운드 출력 Task가 동작 되어야하는데 우리는 이것을 동기화(Synchronization)라고 부른다.

우리가 공부한 내용만으로 이것을 구현하려면 방법이 없다. 사실 전역 변수를 사용하여 구현 가능하지만 너무나도 비효율적이고 번거로운 점들이 생기게 된다. 그래서 우리는 효율적으로 동기화 하는 방법에 대해서 살펴보도록 하겠다.

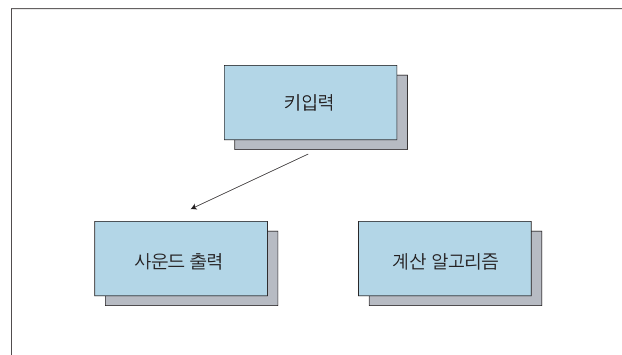

그림 1

Semaphore

여기서 사용되는 semaphore는 공유자원을 위한 기능이 아니다. 동기화를 위한 목적으로 사용되는데 간단하게 구현할 수 있는 동기화 방법 중 하나이다. 리스트 1은 그림 1에 대한 코드를 semaphore로 구성한 것이다.

```
OS_Event * sync
void StartTask(void * para)
{
    . . .
    sync= OSSemCreate(0);  ——————— (1)
    . . .
}

void KeyTask(void * para)
{
    . . .
    while(1)
    {
        KeyInput();
        OSSemPost(sync);  ——————— (2)
        . . .
    }
}

void SoundTask(void * para)
{
    INT8U err;
    . . .
    while(1)
    {
        OSSemPend(sync, 0, &err);  ———— (3)
        Sound();
    . . .
    }
}
```

리스트 1

리스트 1에서 SoundTask가 KeyTask보다 우선순위가 높다고 가정을 해보자.

① semaphore를 만드는 과정이다. 이때 인자가 '0' 인 것을 주의하자.
② SoundTask가 우선순위가 높기 때문에 먼저 수행을 한다. (3)은 수행시에 초기 Sync 표의 개수가 '0' 이기 때문에 표를 가져오지 못하고 받을 때까지 WAIT 상태가 된다.
③ KeyTask()가 수행이 되면, 키입력을 받고 (2)를 수행하여 Sync에 표를 전달하게 된다. 이때, 이것을 기다리고 있던 SoundTask가 Ready가 되고 KeyTask는 선점을 당하게 된다.
④ SoundTask는 (3)에서 깨어나 표를 가져오고 나머지 루틴을 수행을 하게 된다. 이때 표의 개수는 다시 '0' 이 된다. 루프에 의해서 다시 (3)을 수행하게 되면 표의 개수는 '0' 이기 때문에 다시 WAIT 상태가 된다.
⑤ KeyTask가 다시 제어권을 갖게 되고, 나머지 루틴을 수행하게 된다.

이와 같은 수행에 의해 KeyTask에서 키입력을 받는 순간 SoundTask가 바로 응답을 하여 사운드 출력을 수행하게 된다. 이 반응은 실시간적으로 일어나게 되며 이러한 성질이 바로 RTOS의 실시간 응답성이다.

◆ 표의 개수

여기까지 Semaphore를 언급하면서 독자들에게 쉽게 설명하기 위해 내부에 표가 들어 있다고 가정을 했다. 이제 좀 더 정확하게 표현을 하면 Semaphore 내부에는 카운트 변수가 들어 있다.
OSSemPend()가 호출될 때마다 내부 카운트는 1씩 감소되고, 이 카운트 값이 '0' 일 때, OSSemPend()가 호출이 되면 감소시키지 않고 이 함수를 호출한 Task를 WAIT 상태로 만든다. OSSemPost()는 호출될 때마다 카운트가 1씩 증가되고, OSSemPend()에 의해 WAIT 상태에 있는 Task가 있을 때는 그 중에서 가장 우선순위가 높은 Task를 READY 상태로 만든다.

만약, SoundTask의 우선순위가 KeyTask 보다 낮다면 어떠한 현상이 발생할까? 이 부분은 여러분들이 직접 해보시길 바란다.

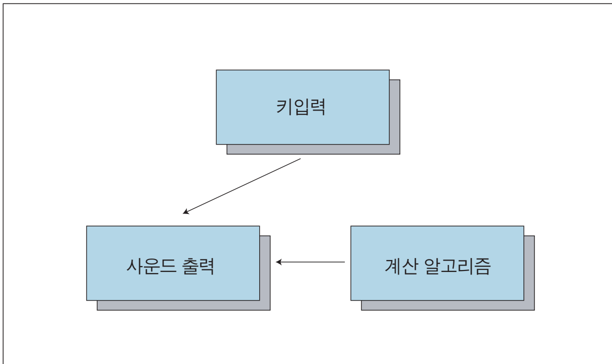

그림 2

Event Flag

이와 같은 상황을 생각해보자(그림 2). 앞의 예에서 계산이 끝났을 때도 사운드 출력이 발생해야 한다면 어떻게 구현할 것인가? 물론 앞에서 배운 semaphore를 이용하여 2개의 semaphore를 생성하여 2개의 Task로부터 신호를 받아 동기화되는 경우에 동작을 시킬 수도 있다. 하지만, 이러한 경우도 효율성이나 구현방법에 있어서 원활하지는 않다. 그래서 이 상황에 맞춰 사용할 수 있도록 uC/OS는 Event flag란 서비스를 제공한다.

```
OS_FLAG_GRP * sync;
void StartTask(void * para)
{
    INT8U err;
     . . .
    sync = OSFlagCreate(0, &err); ———————— (1)
     . . .
}

void KeyTask(void * para)
{
     . . .
    while(1)
    {
        KeyInput();
        OSFlagPost(sync, 0x01, OS_FLAG_SET, &err); —— (2)
         . . .
    }
}

void AlgoTask(void * para)
{
     . . .
    while(1)
    {
         . . .
        OSFlagPost(sync, 0x02, OS_FLAG_SET, &err); —— (3)
         . . .
    }
}

void SoundTask(void * para)
{
    INT8U err;
     . . .
    while(1)
    {
        OSFlagPend(sync, 0x03,
               OS_FLAG_WAIT_SET_ANY+OS_FLAG_CONSUME,
               &err); ———————————————— (4)
        Sound();
     . . .
    }
}
```

리스트 2

리스트 2는 Event flag를 사용하여 KeyTask와 AlgoTask에서 보내는 여러 개의 신호를 SoundTask가 받아서 처리하는 내용이다. 이와 같이 Event Flag는 다양한 비트의 조합에 의해서 동기화가 가능한 서비스이다.

SoundTask는 OSFlagPend() 함수를 호출하여 아래와 같이 색깔로 표시된 비트의 조합에 따라서 신호를 받는다. 리스트 2

에서는 OS_FLAG_WAIT_SET_ANY 옵션을 사용하여 비트 0과 비트 1중 한 개라도 Set이 되면(OR) 동기화되도록 설정을 하였다. 또한 OS_FLAG_CONSUME을 사용하여 조건이 만족되는 경우 해당 비트들을 clear 시켜준다.

…	3	2	1	0

[SoundTask가 사용하는 OS_FLAGS 비트들]

그리고 KeyTask와 AlgoTask는 OSFlagPost() 함수를 호출하여 해당 비트를 Set을 시킨다. 결과적으로 KeyTask와 AlogTask 중 어느 Task에서 든 신호를 보내주기만 하면 SoundTask는 신호를 받아서 동작하게 된다. 만약 OSFlagPend()의 OS_FLAG_WAIT_SET_ANY을 OS_FLAG_WAIT_SET_ALL로 바꾼다면 KeyTask와 AlogTask 둘 다 신호를 보내준 경우에만(AND) 조건을 만족시켜 SoundTask가 동작을 하게 된다.

◆ Set과 Clear

Set은 비트를 1로 Clear는 비트를 0으로 만드는 동작을 의미한다.

◆ OS_FLAGS

여기서는 OS_FLAGS의 데이터 타입이 정해져 있지 않다. 그러나 사용자가 원하는 형태로 정의가 가능하고, 정의는 OS_CFG.H에서 할 수 있다.
x86에서는 디폴트로 INT16U로 정의되어 있다.

[함수원형]
OS_FLAG_GRP * OSFlagCreate(OS_FLAGS flags, INT8U *err)

[설명]
Event flag를 만든다.
[인자]
flagss – 초기 flag 값
err – 에러 값
[결과]
만들어진 Flag event 정보

[함수원형]
OS_FLAGS OSFlagPost(OS_FLAG_GRP * pgrp, OS_FLAGS flags,INT8U opt, INT8U *err)

[설명]
Flag event인 pgrp에 주어진 비트 신호를 준다.
[인자]
grgp – OSFlagCreate()에 의해서 만들어진 Event Flag의 정보
flags – set이나 clear시키고자 하는 비트
opt – flag에서 정의한 비트들을 OS_FLAG_SET으로 set하거나 OS_FLAG_CLEAR로 clear
err – 에러코드
[결과]
수행후의 flag 상태

[함수원형]
OS_FLAGS OSFlagPend(OS_FLAG_GRP * pgrp, OS_FLAGS flags,INT8U wait_type, INT8U timeout, INT8U *err)

[설명]
Flag event인 pgrp에서 주어진 비트신호와 인자로 주어진 비트조합의 조건이 만족될 때까지 WAIT 상태로 대기한다.
[인자]
grgp – OSFlagCreate()에 의해서 만들어진 Event Flag의 정보
flags – 비교하고자 하는 비트 값
wait_type
OS_FLAG_WIAT_CLR_ALL – flag에 해당되는 모든 비트가 clear
OS_FLAG_WIAT_CLR_ANY – flag에 해당되는 어떤 비트가 clear
OS_FLAG_WIAT_SET_ALL – flag에 해당되는 모든 비트가 set

OS_FLAG_WIAT_CLR_ALL – flag에 해당되는 어떤 비트가 set
OS_FLAG_CONSUME – 조건이 만족될 때 flags의 비트들을 clear
timeout – 대기시간
err – 에러코드
[결과]
수행후의 flag 상태

Intertask Communication

앞에서 다뤘던 그림 1의 예에서 좀 더 발전된 형태를 고려해보자. 그림 1의 예에서는 단순히 키입력을 받으면 소리가 난다고 했는데, 누르는 키에 따라서 다른 소리를 내고자 한다면 어떻게 해야 할까. 당연히 눌려진 키 값도 같이 전달이 되어야 한다. 앞에서 배운 Semaphore나 Event Flags들은 단순한 신호전달을 할 뿐, 이러한 서비스를 구현하는데 어려움이 있다.

이때 필요한 정보의 전달을 내부통신(Intertask Communication)이라고 한다.

Mail Box

```
OS_Event * sync;
void StartTask(void * para)
{
    . . .
    sync = OSMboxCreate(0);  ——— (1)
    . . .
}

void KeyTask(void * para)
{
    . . .
    while(1)
    {
        key = KeyInput();
        OSMboxPost(sync, key);  —— (2)
        . . .
    }
}

void SoundTask(void * para)
{
    INT8U err;
    . . .
    while(1)
    {
        . . .
        msg = OSMboxPend(sync, 0, &err); ——(3)
        Sound(msg);
        . . .
    }
}
```

리스트 3

리스트 3은 Mailbox를 이용한 예제이다. Mailbox는 말 그대로 우체통과 같다. (1)은 우체통을 만드는 과정이고 (2)는 우체통에 메일(정보)을 넣어서 (3)의 과정을 통해서 데이터를 받게 된다. 만약 (3)의 과정 중에서 Mailbox에 데이터가 없으면 SoundTask는 WAIT 상태가 된다. 그리고 (2)가 호출되면 메일을 받게 되어 SoundTask는 다시 Ready 상태가 된다.

참고로 주의할 점은 Mailbox는 한 번에 하나의 데이터 밖에 저장이 되지 않는다. 그래서 연속적으로 OSMboxPost를 호출하게 되면 Mailbox의 용량이 넘쳐서 OSMboxPost 호출시 OS_MBOX_FULL이란 에러가 생기게 된다. 그러므로 Mailbox 사용시 이러한 경우를 조심해야 한다.

[함수원형]
OS_EVENT * OSMboxCreate(void * value)

[설명]
Mailbox를 만든다.

[인자]
value – 초기 Mailbox의 값
[결과]
만들어진 Mailbox 정보

[함수원형]
INT8U OSMboxPost(OS_EVENT * pevent, void * msg)

[설명]
Mailbox pevent에 msg를 전달한다.
[인자]
pevent – OSMboxCreate()에 의해서 만들어진 Mailbox 정보
msg – 전달하려는 정보
[결과]
에러코드

[함수원형]
void * OSMboxPend(OS_EVENT * pevent, INT16U timeout, INT8U * err)

[설명]
Mailbox pevent에서 정보를 받는다.
[인자]
pevent – OSMboxCreate()에 의해서 만들어진 Mailbox 정보
timeout – 대기시간
err – 에러코드
[결과]
mailbox에 저장된 정보

Queue

```
OS_Event * sync;
void * QBuff[10];
```

```
void StartTask(void * para)
{
   . . .
   sync = OSQCreate(&QBuff[0], 10); —— (1)
   . . .
}

void KeyTask(void * para)
{
   . . .
   while(1)
   {
      key = KeyInput();
      OSQPost(sync, key); ———— (2)
      . . .
   }
}

void AlgoTask(void * para)
{
   . . .
   while(1)
   {
      . . .
      OSQPost(sync, data); ———— (3)
      . . .
   }
}

void SoundTask(void * para)
{
   INT8U err;
   . . .
   while(1)
```

```
    {
        . . .
        msg = OSQPend(sync, 0, &err); —— (4)
        Sound(msg);
        . . .
    }
}
```

리스트 4

그림 2의 예제에서 데이터를 전송해야 하는 경우 Mailbox를 사용하게 되면 문제가 발생할 수 있다는 것을 쉽게 예상할 수 있다. 2개의 Task가 동시에 Mailbox에 데이터를 전송하게 되면 OS_MBOX_FULL이란 에러가 발생하게 된다. 그래서 대용량의 데이터를 전송하기 위해 Queue란 서비스가 제공된다. Queue의 기본적인 동작방식은 Mailbox와 동일하고 Mailbox와 비교해서 용량이 늘어난 것이라고 생각하면 된다.

리스트 4는 Queue를 이용하여 데이터를 전송하는 예제이다. AlgoTask와 KeyTask가 동시에 SoundTask에 데이터를 전송해도 여러 개의 데이터를 저장할 수 있으므로 데이터 전송이 가능하게 된다.

Queue라고 해서 무한대로 저장할 수 있는 것은 아니다. (1)과 같이 Queue 생성시 Queue의 용량과 저장 장소를 지정하는 것에 의해서 그 크기가 정해진다.

[함수원형]
OS_EVENT * OSQCreate(void * * start, INT8U size)

[설명]
Queue를 만든다.
[인자]
start – 데이터를 저장하기 위한 저장 공간의 시작주소
size – 저장할 수 있는 데이터의 개수
[결과]
만들어진 Queue 정보

[함수원형]
INT8U OSQPost(OS_EVENT * pevent, void * msg)

[설명]
Queue pevent에 msg를 전달한다.
[인자]
pevent – OSQCreate()에 의해서 만들어진 Queue정보
msg – 전달하려는 정보
[결과]
에러코드

[함수원형]
void * OSQPend(OS_EVENT * pevent, INT16U timeout, INT8U *err)

[설명]
Mailbox pevent에서 정보를 받는다.
[인자]
pevent – OSQCreate()에 의해서 만들어진 Queue 정보
timeout – 대기시간
err – 에러코드
[결과]
Queue에 저장된 정보

여기까지 해서 우리가 공부한 동기화나 내부통신, 공유자원의 성질을 간단하게 정리해보면 다음과 같다.

- xxxCreate: 사용하기 위한 서비스를 생성한다.
- xxxPend: 데이터(신호)를 받는다. 받을 수 없는 경우라면 이 함수를 호출한 Task는 WAIT 상태가 된다.
- xxxPost: 데이터(신호)를 보낸다. WAIT 상태에 있는 Task가 존재하면 READY로 만들고 데이터를 깨어난 Task에 전달한다.

다음 시간에는 ARM 포팅에 대해서 다뤄보도록 하겠다. RealTime