uC/OS-II 뛰어넘기(VI)

커널 포팅하기

우리는 uC/OS-II의 함수들과 그 동작들에 대해서 살펴보면서, PC 환경에서 테스트한다는 가정을 했었다. 이번 회에는 현재 시장에서 많이 사용되고 있는 ARM에 uC/OS를 포팅하면서 Context Switching의 의미와 구현과정을 살펴보고, 임베디드 시스템에서 uC/OS를 동작시켜 보도록 하자.

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

이번 회에는 커널 포팅에 대해서 다뤄 보도록 하겠다. 먼저 원활한 진행을 위해 독자들께 한 가지 조언을 하고자 한다. 필자가 오프라인에서 uC/OS 포팅과 관련된 강의를 하면서 느끼는 몇 가지 어려운 점들이 있는데, 그 중에 하나가 커널 구조에 대한 이해이다.

강의를 진행하다보면 커널에 대한 이해가 부족해 포팅하는데 있어 어려움을 겪는 분들을 많이 보아왔다. 물론 커널에 대한 이해 없이도 포팅은 가능하지만, 경험상 uC/OS를 포팅하는데 있어서 커널의 이해는 상당히 많은 도움이 된다는 것을 느낀다. 이는 uC/OS의 구조상 더욱 요청되는 사항이다.

강좌를 진행하면서 다루지 않은 부분이 있다면 커널 분석이다. 이 부분은 이미 충분히 설명된 원서와 번역본이 있고 연재를 통한 분석은 부족한 점들이 많을 것으로 생각돼서 다루지 않았지만, 가능하다면 꼭 그 내용을 공부해보시라고 권해 드리고 싶다.

본론으로 들어가서 우리는 uC/OS를 ARM으로 포팅을 하려고 한다. 그러기 위해서는 우리가 무엇을 포팅해야 할 지와 여기서 발생되는 문제들을 생각해 보고 포팅을 구현해 보도록 해보자.

포팅 파일

먼저 포팅을 하기 위해서는 무엇을 포팅해야 할 지 알아야 할 것이다. 그림 1은 우리가 포팅해야 할 파일들을 잘 말해주고 있다.

OS_CPU.H

① 형선언

CPU에 의존적인 정의 부분들을 선언하고 있다. 예를 들어,

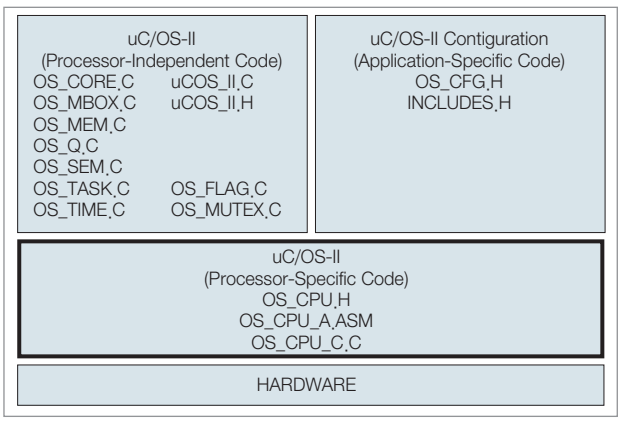


그림 1. uC/OS 커널 파일 구조

x86의 도스 모드에서 int형은 16 bit이지만 윈도우에서의 int형은 32 bit이다. 물론 우리가 사용할 ARM에서도 int형은 32 bit이다.

② OS_XXX_CRITICAL()

OS_ENTER_CRITICAL(), OS_EXIT_CRITICAL()를 정의한다. 이 함수들의 속성은 인터럽트를 enable/disable 시키는 동작을 하는데, 이것은 프로세서마다 구현하는 방법이 달라진다.

③ OS_TASK_SW()

소프트웨어 인터럽트(Software Interrupt)를 이용하여 OSCtxSw() 함수를 호출하는 함수로 #define으로 정의되어 있다.

④OS_STK_GROTH

스택이 자라는 방향을 설정한다.

OS_CPU_A.ASM

이 파일은 4개의 파일로 구성된다. ①, ②, ③은 Context Switching 함수로 스케줄러에 의해서 호출된다.(OS_CORE.C 참조) ④는 타이머 인터럽트(Timer Interrupt) 서비스 루틴이다.

①OSStartHighRdy()
②OSCtxSw()
③OSIntCtxSw()
④OSTickISR()

OS_CPU_C.C

OSxxxHook() 함수들은 커널을 수정하지 않고 OS의 동작을 제어하거나 기타 다른 동작들을 하기 위한 함수들이다. OSTaskStkInit()은 Context switching시 초기의 Task의 Stack 모양을 잡아주는 초기화 함수이다.

① OSTaskStkInit()
② OSxxxxHook()

Poring

앞에서는 우리가 포팅해야 할 파일들과 그 내용을 살펴봤고 이번에는 포팅 할 함수들을 다시 정리해 보면,

① OSStartHighRdy()
② OSCtxSw()
③ OSIntCtxSw()
④ OSTaskStkInit()
⑤ OSTickISR()

이와 같다.

이 중에서 ①, ②, ③은 Context Switching 함수이고, ④는 Context Switching을 하기 위한 스택을 초기화 하는 함수이다. 그러므로 우리가 포팅의 내용을 진행하는데 있어서 Context Switching 함수는 그 중심에 있다고 할 수 있다.

Context Switching 함수를 포팅하기 위해서는 Context Switching 함수의 동작 과정과 동작 원리에 대해서 이해해야 할 뿐만 아니라, 그 밖에도 다음과 같은 사실들을 염두에 두어야 한다.

X86에서의 Context Switching

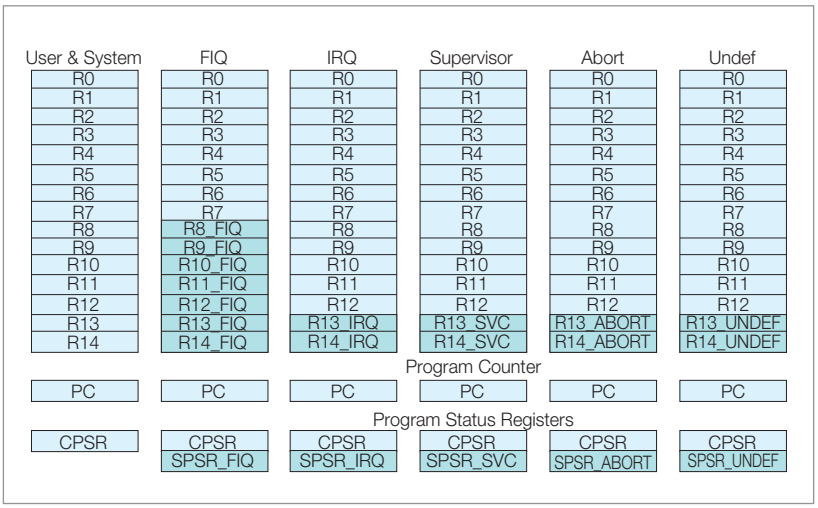
uC/OS-II 홈페이지에는 x86에서 수행되는 Context Switching 과정이 플래시로 설명이 잘 되어있다. 이것을 참고하면 많은 도움이 되니 꼭 방문해서 과정을 살펴보시기 바란다.

www.ucos-ii.com

uC/OS는 x86을 기본 모델로 하고 있다. 우리가 사용하는 ARM은 x86과 구조가 다르기 때문에 포팅시 고려해야 할 문제들이 많이 발생하게 된다.

• 인터럽트의 발생과정

x86은 인터럽트 발생시 인터럽트 종류에 따라서 해당 인터럽



ARM의 모드

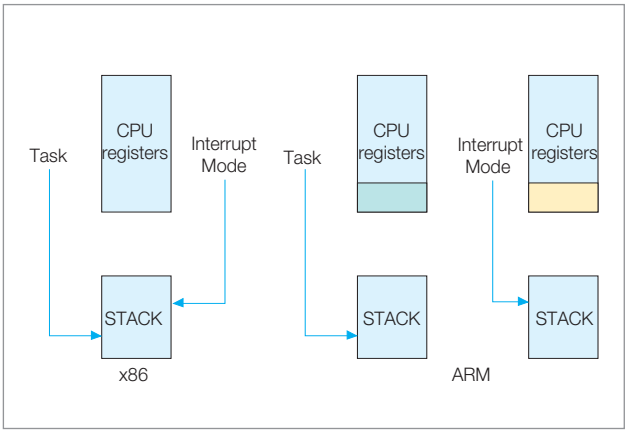


그림 2. 인터럽트 전환과정

트 서비스 루틴이 수행되는 Vector 방식을 가지고 있지만, 우리가 사용하는 ARM은 인터럽트 발생시(IRQ, FIQ exception) 인터럽트의 종류와 상관없이 같은 번지의 명령을 수행하는 Non vector 방식을 사용한다. 또한 ARM에서는 IRQ Mode로의 전환이 이루어진다는 것을 고려해야 한다.

• 모드의 차이

ARM은 여러 개의 모드가 존재하고 exception 발생에 따라서 모드 전환이 이루어진다. 하지만 x86에는 모드란 것이 없어서 ARM으로 포팅하는데 있어서 이 부분이 가장 까다로운 부분이 된다.

uC/OS는 Context Switching을 하기 위해서 소프트웨어 인터럽트와 하드웨어 인터럽트를 사용한다. ARM에서도 이러한 인터럽트(ARM에서는 exception이란 용어를 사용함)를 사용하게 되는데, 이때 모드 전환이 이루어진다.

이 부분이 ARM으로의 포팅에서 까다로운 점이 된다. x86에서는 인터럽트 발생시 Flag register와 리턴하기 위한 주소를 스택에 저장하게 된다. 그리고 인터럽트 루틴에서는 이 스택에 부가적인 CPU의 register들을 저장하여 context switching을 구현하게 된다.

하지만, ARM에서는 exception 발생시 각각의 모드에 스택이 존재하게 된다. 그리고 exception 발생시 x86의 Flag 레지스터(Register)에 해당되는 CPSR는 SPSR로 저장되고 리턴 주소가 스택이 아닌 LR 레지스터로 저장이 된다. 또한 모드 변경이 이루어지면 일부 레지스터들은 해당 모드의 전용 레지스터 셋으로 바뀌게 된다.

즉, x86에서는 모든 레지스터들이 스택으로 저장이 되고, 스택과 인터럽트 루틴에서 같은 스택을 공유하게 된다. 하지만, ARM에서는 스택을 공유하지 못하고, 레지스터가 스택으로 저장이 되지 않는다는 결정적인 차이점을 갖게 된다.

이 시점에서 우리는 어떻게 포팅을 할 것인가를 판단해야 한다.

• x86과 같은 모델

x86에서 일어나는 과정을 ARM에서 그대로 구현하는 방법이다. 하지만 이 경우 ARM의 특성을 활용해서 사용하기에 어려움이 따르게 된다.

• ARM 독자적 모델

ARM의 특성을 살려서 포팅을 한다. 물론 ARM의 특성을 활용하기 때문에 안정적인 부분이나 효율성에서는 우수하지만 uC/OS 자체가 x86에서 출발한 것이기 때문에 제약이 따를 수 있다는 점을 참고해야 한다.

필자는 여기서 'ARM 독자적 모델'을 이용한 방법을 구현하도록 하겠다.

* ARM 독자적 모델

여기서 사용하는 ARM 독자적 모델로의 포팅을 하기 위해서는 커널의 수정이 불가피하다.

```
[ucos_ii.h]
typedef struct os_tcb
{
  INT32U    armReg[17];/ * by redizi, for ARM  * /
  OS_STK    * OSTCBStkPtr;
```

ARM에서 모드마다 독자적인 스택을 갖게 되므로 x86과 같이 Context의 내용을 Task 각각의 스택에 저장하기가 여의치 않다.(가능하지만 좀 더 스마트한 방법을 찾기 위해서... :)) 그래서 여기서는 context를 스택이 아닌 TCB에 저장하여 이 부분을 해결하였다.

참고로 이 포팅 방법 이외에도 여러 가지 가능한 방법들이 있으므로 필자의 방법이 최선이라고 말할 수 없음을 미리 밝혀둔다. 필자 또한 공부 중에 여러 가지 포팅 방법 중 한 가지를 여러분께 제시한 것이니 독자여러분들의 아이디어에 따라서 여러 가지 활용방법이 있으니 열심히 도전해 보시길 바란다.

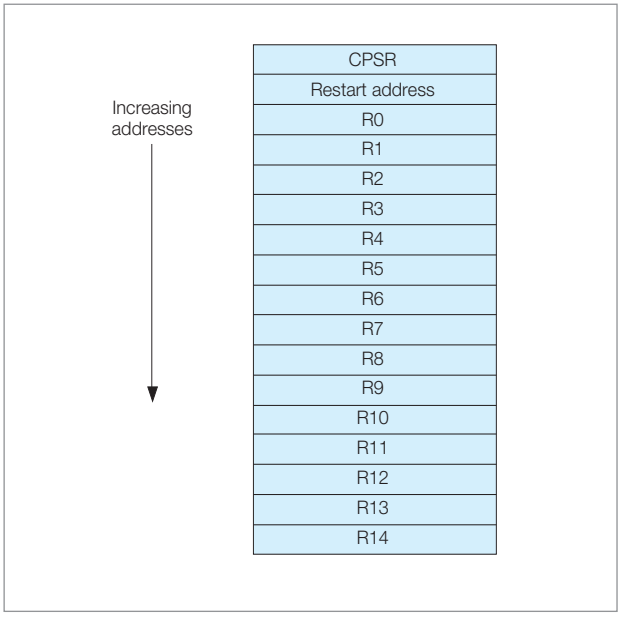


그림 3. Stack Frame

Stack Frame

그림 3은 우리가 Context Switching을 하기 위해서 사용할 스택의 모양이다. 우리가 context의 내용을 저장하고 복구할 때 이와 같은 모양으로 구현하게 된다. 하지만, 앞에서 밝힌 것과 같이 우리는 Context를 스택이 아닌 TCB에 저장할 것이므로 원래 포팅 대상인 OSTaskStkInit()이 아닌 OSTCBInitHook()에서 Context 초기화를 구현했다.

```
void  OSTCBInitHook (OS_TCB  * ptcb)
{
    INT32U  * stk;

    stk = (INT32U * )ptcb->OSTCBStkPtr;

    ptcb->armReg[ 0] = (INT32U)0x00000010;

    ptcb->armReg[ 1] = (INT32U) * (stk-1);

    ptcb->armReg[ 2] = (INT32U) * (stk-2);

    ptcb->armReg[ 3] = (INT32U)0;

    ptcb->armReg[ 4] = (INT32U)0;

    ptcb->armReg[ 5] = (INT32U)0;

    ptcb->armReg[ 6] = (INT32U)0;

    ptcb->armReg[ 7] = (INT32U)0;

    ptcb->armReg[ 8] = (INT32U)0;

    ptcb->armReg[ 9] = (INT32U)0;

    ptcb->armReg[10] = (INT32U)0;

    ptcb->armReg[11] = (INT32U)0;

    ptcb->armReg[12] = (INT32U)0;

    ptcb->armReg[13] = (INT32U)0;

    ptcb->armReg[14] = (INT32U)0;

    ptcb->armReg[15] = (INT32U)stk;

    ptcb->armReg[16] = (INT32U) * (stk-1);
}
```

소스 1. OSTCBInitHook

```
OS_STK *OSTaskStkInit (void (*task)(void *pd), void *pdata,
OS_STK *ptos, INT16U opt)
{
    INT32U *stk;
    opt    = opt;
    stk    = (INT32U *)ptos;
    *(stk-1) = (INT32U)task;
    *(stk-2) = (INT32U)pdata;
    return ((OS_STK *)stk);
}
```

소스 2. OSTaskStkInit

Context Switching

context switching을 위한 방법으로 우리는 다음과 같은 알고리즘을 사용할 것이다.

```
;R0 <- context를 저장할 주소
MRS     r12,  SPSR
STR     r12,  [r0], #8
LDMFD  sp!,  {r2, r3 }
STMIA  r0!,  {r2, r3 }
LDMFD  sp!, {r2, r3, r12, r14 }
STR     r14,  [r0, #-12]
STMIA  r0,   {r2-r14 }^
    .....
    .....
;R2 <- context를 읽어들일 주소
LDMIA  r2!,  {r12, r14 }
MSR     spsr_fsxc, r12
LDMIA  r2,  {r0-r14 }^
NOP
MOVS   pc,  r14
```

소스 3. Context Switching

여기서 Task는 USER 모드에서 동작된다는 것을 가정하고 구현이 된 것이다. ARM에는 User Bank Transfer라는 특수한 명령이 있어서 OS를 사용하는 경우 이것을 이용하여 Context Switching을 구현할 수 있다. 소스 3은 이 명령과 USER 모드를 이용하여 context switching을 구현한 것이다.

소스 4는 이 알고리즘을 이용하여 3개의 context switching 함수를 구현하였다. 참고로 OSIntCtxSw()와 OSCtxSw()는 코드가 거의 같아서 공통 부분을 같이 사용하였다.

```
OSStartHighRdy
    BL          OSTaskSwHook
    LDR         r0, =OSRunning
    MOV         r1, #1
    STRB        r1, [r0]
    LDR         r0, =OSTCBHighRdy
    LDR         r0, [r0]
;; context restored
    LDMIA       r0!, {r12, r14}
    MSR         spsr_fsxc, r12
    LDMIA       r0, {r0-r14}^
    NOP
    MOVS        pc, r14

OSIntCtxSw    ---①
;  add  sp, sp, #12          ; SDT Release
;  add  sp, sp, #8           ; ADS Debug
;  add  sp, sp, #            ; ADS Release

OSCtxSw
;; context saved
    LDR         r0, =OSTCBCur
    LDR         r0, [r0]
    MRS         r12,        SPSR
    STR         r12,        [r0], #8
    LDMFD       sp!,        {r2, r3}
    STMIA       r0!,        {r2, r3}
```

```
    LDMFD       sp!,        {r2, r3, r12, r14}
    STR         r14,        [r0, #-12]
    STMIA       r0,         {r2-r14}^

;; set variables
    BL          OSTaskSwHook
    LDR         r1, =OSPrioHighRdy
    LDR         r0, =OSPrioCur
    LDRB        r3, [r1]
    STRB        r3, [r0]
    LDR         r1, =OSTCBHighRdy
    LDR         r0, =OSTCBCur
    LDR         r2, [r1]
    STR         r2, [r0]

;; context restored
    LDMIA       r2!, {r12, r14}
    MSR         spsr_fsxc, r12
    LDMIA       r2, {r0-r14}^
    NOP
    MOVS        pc, r14
```

소스 4. Context Switching 함수들

Exception

uC/OS 동작 중의 context switching이 실행되는 과정은 전부 exception에 의해서 일어난다.

> * OSStartHighRdy()
> 이 context switching 함수는 OSStart() 함수에 의해서 호출되면 OS 실행 중 처음 단 한번만 수행이 된다. 그러므로 OS 동작 중에 수행되는 Context Switching함수는 OSIntCtxSW() 함수와 OSCtxSw() 함수이다.

즉, 앞의 context switching이 제대로 동작하기 위해서는 exception handler와 같이 연동이 되어야 한다.

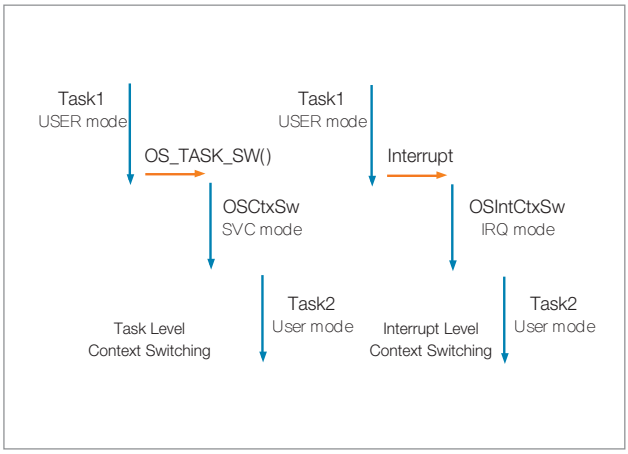


그림 4. 동작 중의 Context Switching 과정

그림 4를 참고하면 OSCtxSw() 함수는 Supervision 모드, OSIntCtxSw() 함수는 IRQ 모드에서 동작을 하게 된다. ARM의 구조상 USER 모드에서, 이 두 개의 모드로 진입을 하기 위해서는 exception이 발생이 되어야 하는데, 이때 각 exception을 핸들링 해주기 위한 exception handler라는 루틴이 수행되어야 한다. 그리고 이 handler를 통해서 우리가 사용하는 context switching함수에 도달하게 된다.

소스 5와 소스 6은 IRQ exception에 의해 동작하는 IRQ handler와 SWI exception에 의해 동작하는 SWI handler의 소스코드이다. 각 리스트의 ①은 context switching 함수와 연동되는 초기화 설정 부분이다. 그리고 이 부분은 ARM에서 C를 사용하기 위한 ATPCS라는 ARM C 컴파일러 스펙과도 연동이 되어있다.

이 부분에 대해서는 ARM의 해당 매뉴얼을 참고하기 바란다.

```
UCOS_SWI
    STMFD       sp!, {r0-r3,r12,lr}  ---①
    LDR         r0, [lr, #-4]
    BIC         r0, r0, #0xff000000

;; Context Switching
    CMP         r0, #0x80
    BLEQ        OSCtxSw
```

```
;; prepare the argument 2
    MOV         r1, sp
    BL          SWI_Handler
    LDMFD       sp!, {r0-r3,r12,pc}^
```

소스 5. SWI exception handler

```
;; for S3C2410
;; Interrupt Control
INTOFFSET     EQU  0x4a000014

UCOS_IRQ
    SUB         lr, lr, #4
    STMFD       sp!, {r0-r3,r12,lr} ---- ①

;; OSIntEnter()
    LDR         r0, =OSIntNesting
    LDRB        r1, [r0]
    ADD         r1, r1, #1
    STRB        r1, [r0]

;; process ISR
    LDR         r0, =INTOFFSET
    LDR         r0, [r0]
    MOV         r0, r0, lsl #2
    LDR         r2, =IRQ_TABLE

    ADD         lr, pc, #0
    LDR         pc, [r2, r0]

    BL          OSIntExit

    LDMFD       sp!, {r0-r3,r12,pc}^
```

소스 6. IRQ exception handler

Compiler

지금까지 언급은 안했지만, 진행해 온 개발환경은 ADS에 있는 특수한 예약어와 그 특성들을 이용했다. 즉, 포팅은 컴파일러와 같은 개발환경에도 영향을 미치게 된다. 그래서 소스 4와 같이 같은 회사의 컴파일러라도 사용하는 환경과 조건에 따라서 차이가 나게 된다.

또한 완전히 다른 회사의 컴파일러라면 포팅시 한번 더 고려를 해봐야 할 것이다. 여기서 다루지는 않았지만 동일한 방식의 포팅을 GCC 컴파일러 환경에서 구현한 포팅도 자료실에 같이 있으니 비교하면서 공부를 해본다면 많은 도움이 되리라 생각한다.

이제 uC/OS에 대한 마지막 부분의 내용까지 다 다루었다. 가능하면 많은 부분을 다루고 도움이 되는 글을 쓰기 위해 노력을 했는데 어느 덧 반년이란 시간이 넘어서 드디어 마무리를 짓게 되어 감개무량하다. 하지만, 필자의 개인적 사정에 의해 중간중간 빠지는 불성실함과 부족함에 대해 독자들께 사과를 드리고 대신 앞으로 더욱 좋은 내용으로 만나 뵐 것을 약속드린다. RealTime

※ 전체소스파일

여기서 사용된 소스는 CyberLab 홈페이지 자료실에 있다. 그리고 궁금한 점이 있다면 홈페이지에 있는 게시판의 OS란을 통해 여러분들께 답변을 드리도록 하겠다.

참고문헌

MicroC/OS-II, Second Edition, Jean J. Labrpsse. CMPBooks
ARM Developer Suite Developer Guide
ARM Architecture Reference Guide