

설계와 구현을 통한 임베디드 OS의 이해와 응용 ⑥

하드웨어 인터럽트 처리 및 태스크 추가 루틴



하드웨어의 요청에 의해 임의의 사용자 영역에서 커널 영역으로 진입하는 방법을 하드웨어 인터럽트라고 한다. 이번 호에서는 하드웨어 인터럽트를 처리하는 커널 루틴을 작성해 볼 것이다. 또 multitasking과 관련된 루틴을 추가하기 위해 먼저 두 개의 태스크(task0, task1)를 추가하는 루틴을 작성해 보자.

글: 서민우/과학기술 정보연구소(www.stii.co.kr)
minucy@hanmail.net

먼저 하드웨어 인터럽트를 처리하는 커널 루틴을 작성해 보자. 여기서는 주기적으로 인터럽트를 발생시키는 timer0 디바이스를 사용한다.

다음과 같은 순서로 하드웨어 인터럽트 처리 루틴을 작성한다.

1. timer0 디바이스를 초기화하는 루틴을 작성한다.
2. timer0 디바이스의 하드웨어 인터럽트를 커널에서 받을 수 있도록 interrupt controller를 초기화하는 루틴을 작성한다.
3. 하드웨어 인터럽트를 처리하는 커널 루틴을 작성한다.
4. timer0 interrupt handler를 작성한다.

다음은 timer.c 파일의 내용이다.

```
timer.c
include <timer.h>

void timer_init()
{
    rGPBCON &= ~0x3;
    rGPBCON |= 0x2;
    rGPBUP |= 0x1;
    rTCFG0 &= ~0xff;
    rTCFG1 &= ~0xf;
    rTCFG1 |= 0x1;
    rTCON &= ~0xff;
    rTCNTB0 = (PCLK/1/4)/1000-1;
```

```

rTCMPB0 = rTCNTB0/2;
rTCON |= 0x02;
rINTMOD = 0x0;
rSRCPND |= (1<<10);
rINTPND |= (1<<10);
rINTMSK &= ~(1<<10);
rTCON &= ~0xff;
rTCON |= 0x09;
}

```

timer_init 함수는 timer0 디바이스를 사용하기 위해 PWM timer controller와 interrupt controller의 레지스터를 설정한다. 먼저 B 포트의 0번 핀을 timer0의 TOUT0 핀으로 사용하기 위해 GPBCON, GPCUP 레지스터를 설정한다. GPBCON, GPCUP 레지스터의 주소 값은 include/timer.h 파일에 정의되어 있다. 이 내용은 뒤에서 다시 살펴보기로 하자.

다음은 timer0 controller의 TCFG0, TCFG1, TCON, TCNTB0, TCMPB0 레지스터를 설정하는 부분이다. 그림 1을 보면서 구체적으로 살펴보자.

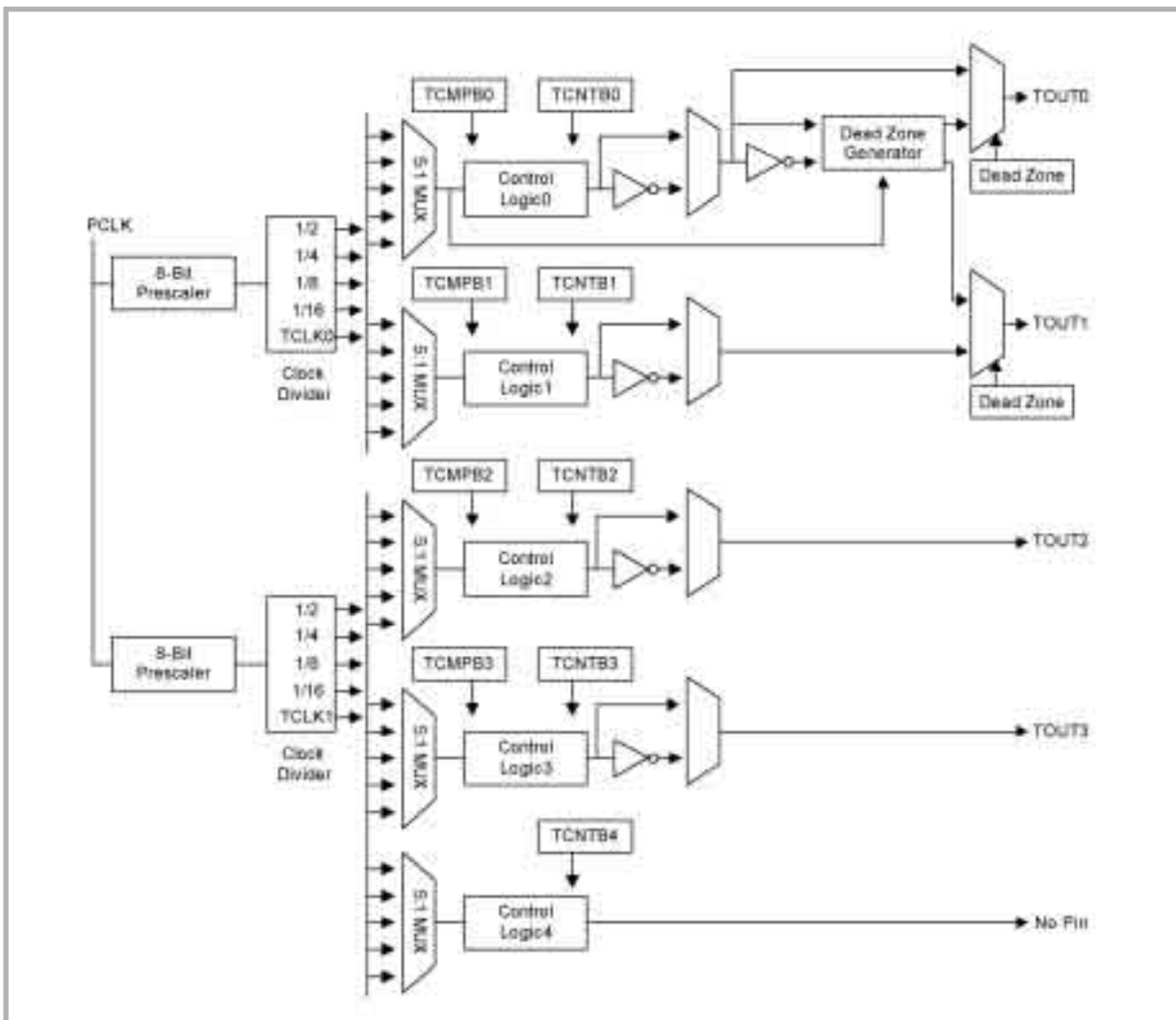


그림 1. PWM Timer 블록 다이어그램

그림 1에서 PCLK 값은 8비트 Prescaler를 거쳐 Clock Divider와 5:1 MUX를 거쳐 Control Logic0의 입력 값으로 들어간다. 8비트 Prescaler는 0~255의 범위를 가지며 PCLK의 값을 1~266의 값으로 나누는 역할을 한다. timer0에 대한 8비트 Prescaler의 값은 TCFG0 레지스터의 하위 8비트를 이용해 설정할 수 있다. timer_init 함수에서는 TCFG0 레지스터의 하위 8비트 값을 0으로 설정한다.

Clock Divider와 5:1 MUX는 8비트 Prescaler에서 Control Logic0으로 들어가는 값을 2, 4, 8 또는 16으로 나누는 역할을 한다. timer0의 Clock Divider와 5:1 MUX에 대한 설정은 TCFG1 레지스터의 하위 4비트를 이용해 설정한다. timer_init 함수에서는 TCFG1 레지스터의 하위 8비트 값을 1로 설정해 8비트 Prescaler에서 Control Logic0으로 들어가는 값을 4로 나누는 역할을 한다.

Control Logic0으로 들어가는 값은 다음 식에 의해서 결정된다.

$$\text{Timer input clock Frequency} = \text{PCLK}/\{\text{prescaler value}+1\}/\{\text{divider value}\}$$

PCLK의 값은 APB 버스로 공급되는 clock 주파수로 include/timer.h 파일에 다음과 같이 정의되어 있다.

```
#define FCLK 399650000
#define HCLK (FCLK/4)
#define PCLK (HCLK/2)
```

즉 FCLK의 값을 8로 나눈 값이다. 따라서 Control Logic0으로 들어가는 clock 주파수는 다음 식에 의해 12489062 값이 된다.

$$(399650000/8)/1/4 = 12489062$$

TCON 레지스터의 하위 8비트는 timer0의 활성화 여부, TCNTB0과 TCMPB0 레지스터의 manual update, TOUT0 값의 inverter 기능 활성화 여부, timer0의 auto reload 활

성화 여부 등을 설정할 때 사용한다. 여기서는 일단 timer0의 모든 기능을 비활성화 시켰다.

PWM 주파수는 TCNTB0 레지스터를 이용해 설정할 수 있다. timer_init 함수는 초당 1000번 인터럽트가 발생하도록 TCNTB0 레지스터를 설정했다. TCON 레지스터의 1번 비트를 1로 설정해 manual update를 수행하면, TCNTB0 레지스터 값은 내부 레지스터 TCNT0으로 전달된다. 그리고 TCON 레지스터의 0번 비트를 1로 설정해 timer0을 활성화시키면, TCNT0 레지스터의 값이 Control Logic0으로 들어오는 clock 주파수에 맞춰 감소하기 시작한다.

TCNT0 레지스터의 값이 0이 되는 순간 interrupt가 발생한다. 이때 TCON 레지스터의 3번 비트가 1 값으로 설정되어 있을 경우에 즉, timer0의 auto reload 기능이 활성화되어 있을 때 TCNT0 레지스터는 TCNTB0 레지스터 값으로 다시 초기화 된다. 그리고 또 다시 Control Logic0으로 들어오는 clock 주파수에 맞춰 감소하기 시작한다.

TOUT0의 값을 언제 low 레벨에서 high 레벨로 내보낼지는 TCMPB0 레지스터를 이용해 결정할 수 있다. TCON 레지스터의 1번 비트를 1로 설정해 manual update를 수행하면 TCMPB0 레지스터 값은 내부 레지스터 TCMP0로 전달된다. TCON 레지스터의 2번 비트가 1 값으로 설정되어 있을 경우에 즉, TOUT0 값의 inverter 기능이 비활성화 되어 있을 때 TCNT0 레지스터의 값이 TCMP0의 값과 같아지는 순간 TOUT0의 값은 low 레벨에서 high 레벨로 바뀐다. 그리고 TCNT0 레지스터가 TCNTB0 레지스터 값으로 다시 초기화 되는 순간에 TOUT0의 값은 high 레벨에서 다시 low 레벨로 바뀐다.

timer_init 함수는 TCNTB0, TCMPB0 레지스터의 값을 설정한 후에 TCON 레지스터의 1번 비트를 1로 설정해 manual update를 수행한다.

다음은 interrupt controller의 INTMOD, SRCPND, INTPND, INTMSK 레지스터를 설정하는 부분이다. 그림 2를 보면서 구체적으로 살펴보기로 하자.

디바이스에서 인터럽트를 요청할 경우 인터럽트 신호는 Request sources(without sub-register)로부터 SRCPND 레지스터를 거쳐 INTMSK(MASK) 레지스터를 거쳐 INTPND 레지스터를 통해 CPU 코어로 전달된다. INTMSK

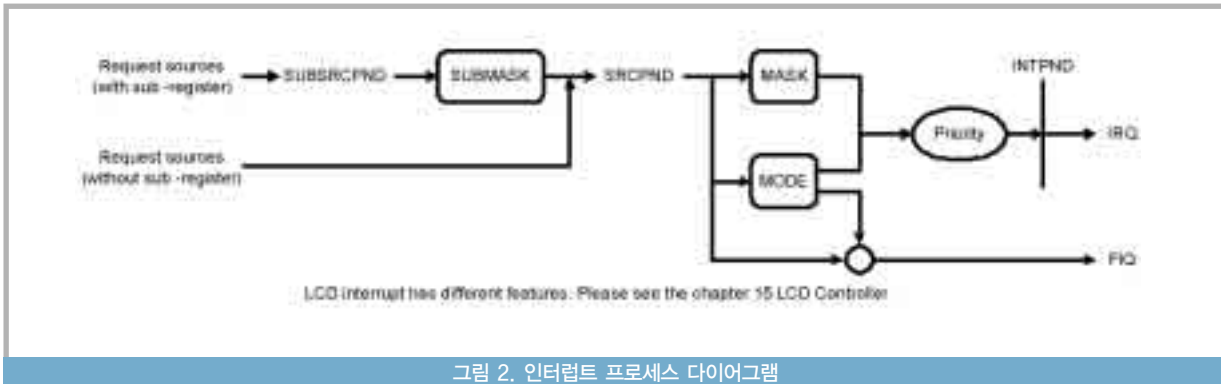


그림 2. 인터럽트 프로세스 다이어그램

레지스터는 SRCPND 레지스터를 통해 들어오는 인터럽트를 걸러내는 역할을 한다. INTMOD(MODE) 레지스터는 디바이스의 인터럽트를 IRQ 모드로 처리할 지, FIQ 모드로 처리할 지의 여부를 결정하는 역할을 한다. Request sources로부터 인터럽트가 들어올 경우 SRCPND와 INTPND 레지스터의 해당되는 비트는 interrupt controller에 의해 1로 설정된다. 1로 설정된 비트는 커널 영역에서 인터럽트를 처리하더라도 그대로 남아있게 되며, 따라서 커널 영역에서 각각의 비트를 0으로 초기화해야 한다. SRCPND와 INTPND 레지스터의 모든 비트는 1로 해졌을 경우 0으로 초기화 할 수 있는 sticky 비트이다.

timer_init 함수는 INTMOD 레지스터를 0으로 설정해 모든 디바이스의 인터럽트 요청을 IRQ 모드로 처리한다. 그리고 INTMSK 레지스터의 10번 비트를 0으로 설정해 timer0 으로부터 들어오는 인터럽트를 허용한다. 주의할 점은 SRCPND, INTPND 레지스터의 10번 비트에 1 값으로 해졌을 경우, 혹시 이전에 timer0 으로부터 인터럽트가 들어올 것을 생각해 SRCPND, INTPND 레지스터의 10번 비트를 0으로 초기화한다.

imer_init 함수의 마지막 부분에서는 TCON 레지스터의 0, 3비트를 1로 설정해 timer0을 활성화시키며, timer0의 auto reload 기능도 활성화시킨다.

지금까지 살펴 본 레지스터의 주소 값은 include/timer.h 파일에 정의되어 있다. 이상에서 timer.c 파일의 내용을 살펴 보았다.

다음은 timer.h 파일의 내용이다. timer.h 파일은 include 디렉토리를 생성하고 그 안에 두어야 한다.

```
include/timer.h

#define FCLK 399650000
#define HCLK (FCLK/4)
#define PCLK (HCLK/2)

#define rGPBCON *(volatile unsigned int *)0xf6000010
#define rGPBUP *(volatile unsigned int *)0xf6000018

#define rTCFG0 *(volatile unsigned int *)0xf1000000
#define rTCFG1 *(volatile unsigned int *)0xf1000004
#define rTCON *(volatile unsigned int *)0xf1000008
#define rTCNTB0 *(volatile unsigned int *)0xf100000c
#define rTCMPB0 *(volatile unsigned int *)0xf1000010
#define rTCNTB1 *(volatile unsigned int *)0xf1000018
#define rTCMPB1 *(volatile unsigned int *)0xf100001c

#define rSRCPND *(volatile unsigned int *)0xea000000
#define rINTMOD *(volatile unsigned int *)0xea000004
#define rINTMSK *(volatile unsigned int *)0xea000008
#define rINTPND *(volatile unsigned int *)0xea000010
#define rINTOFFSET *(volatile unsigned int *)0xea000014
```

다음은 exceptions.S 파일의 내용을 살펴보기로 하자. 하드웨어 인터럽트를 처리하기 위해 coreIRQHandler 함수를 아래와 같이 수정한다.

```
exceptions.S
...
.globl coreIRQHandler
coreIRQHandler:
    sub r14,r14,#4
    stmfd r13!,{r0-r3,r12,r14}
    bl eventsIRQHandler
    ldmfd r13!,{r0-r3,r12,pc}^
```

coreIRQHandler 함수는 먼저 r14(lr) 레지스터의 값을 수정한다. r14(lr) 레지스터는 IRQ 모드의 레지스터이며, 하드웨어 인터럽트가 발생했을 당시 수행하고자 했던 명령어의 다음 명령어 위치 값을 가진다. 따라서 위와 같이 r14 레지스터의 값을 수정해야 한다.

그리고 스택에 r0, r1, r2, r3, r12, r14를 저장한다. r13(sp) 레지스터는 IRQ 모드의 레지스터이고, 하드웨어 인터럽트를 구체적으로 처리하기 위해 eventsIRQHandler 함수를 호출한다. EventsIRQHandler 함수는 다음에 살펴볼 irqhandler.c 파일에 정의되어 있다.

EventsIRQHandler 함수에서 리턴한 후에 스택으로부터 r0, r1, r2, r3, r12 레지스터를 복구하고, 이전에 스택에 저장했던 r14 레지스터의 값을 pc 레지스터로 복구해 인터럽트가 발생했을 당시 수행하고자 했던 명령어로 리턴한다. 이때, spsr 레지스터의 값도 cpsr 레지스터로 옮겨진다.

이상에서 exceptions.S 파일의 내용을 살펴보았다. 다음은 irqhandler.c 파일의 내용을 살펴보기로 하자. irqhandler.c 파일의 내용은 다음과 같다.

```
irqhandler.c
#include <timer.h>

void eventsIRQHandler()
{
    unsigned int rintoffset;
    unsigned int rintpnd;
```

```
    uart_puts("IRQ handler: ");
    rintoffset = rINTOFFSET;
    uart_puts("rINTOFFSET - ");
    uart_puthex(rintoffset);

    rintpnd = rINTPND;
    uart_puts(", rINTPND - ");
    uart_puthexnl(rintpnd);

    rSRCPND |= rintpnd;
    rINTPND |= rintpnd;
}
```

irqhandler.c 파일에는 eventsIRQHandler 함수가 정의되어 있다. EventsIRQHandler 함수는 INTOFFSET, INTPND 레지스터 값을 차례로 읽은 후, 그 값을 출력한다. timer0으로부터 인터럽트가 발생했을 경우 INTPND 레지스터의 10번 비트가 1로 설정되며, INTOFFSET 레지스터는 10 값을 가진다. INTOFFSET 레지스터는 INTPND 레지스터에 1로 설정된 비트 중 최고 우선순위를 갖는 비트의 위치 값을 가지고 있다.

EventsIRQHandler 함수의 마지막에는 SRCPND, INTPND 레지스터의 발생했던 인터럽트에 해당하는 비트를 0으로 초기화한다. EventsIRQHandler 함수는 timer0의 인터럽트를 처리하는 루틴으로 작성되었으나, 일반적인 하드웨어 인터럽트를 처리하는 루틴의 기본 틀로 사용할 수 있다.

이상에서 irqhandler.c 파일의 내용을 살펴보았다. 다음은 main.c 파일의 내용을 살펴보자. main.c 파일의 내용은 아래와 같다.

```
main.c
#include <system.h>
void uart_init();
#define syscall() __asm__ ("swi 1")
```

```
void kmain(void)
{
    uart_init();
    timer_init();
    local_irq_enable();
    while(1) syscall();
}
```

kmain 함수는 timer_init 함수를 호출해 timer0 디바이스를 초기화하고, local_irq_enable 매크로를 이용해 하드웨어 인터럽트를 활성화한다. 그리고 무한히 <swi 1> 명령어를 수행한다. local_irq_enable 매크로는 include/system.h 파일에 정의되어 있다.

다음은 system.h 파일의 내용이다. system.h 파일도 include 디렉토리에 넣어야 한다.

```
include/system.h
#define local_irq_enable() \
({ \
    unsigned long temp; \
    __asm__ __volatile__( \
        "mrs %0,cpsr\n" \
        "bic %0,%0,#128\n" \
        "msr cpsr_c,%0" \
        : "=r"(temp) \
        : \
        : "memory", "cc"); \
    })

#define local_irq_disable() \
({ \
    unsigned long temp; \
    __asm__ __volatile__( \
        "mrs %0,cpsr\n" \
        "orr %0,%0,#128\n" \
```

```
"msr cpsr_c,%0" \
: "=r"(temp) \
: \
: "memory", "cc"); \
})
```

system.h 파일에는 하드웨어 인터럽트를 활성화하는 local_irq_enable 매크로와 비활성화 하는 local_irq_disable 매크로가 inline assembly를 이용해 정의되어 있다. 각각 cpsr의 7번 비트(1 비트)를 0 또는 1로 설정함으로써 하드웨어 인터럽트를 활성화하거나 비활성화 한다.

이상에서 system.h 파일의 내용을 살펴보았다. 다음은 Makefile의 내용을 수정해 보기로 하자.

```
.c.o:
    arm-linux-gcc $< -c
```

아래와 같이 수정한다.

```
.c.o:
    arm-linux-gcc $< -c -include
```

이렇게 수정하면 C 파일을 컴파일 할 경우 include 디렉토리 내의 timer.h 파일과 system.h 파일을 이용할 수 있다. 또 timer.c 파일과 irqhandler.c 파일이 커널에 추가되었으므로 KERNEL_OBJ 변수를 다음과 같이 수정한다.

```
KERNEL_OBJ = head.o main.o mmusetup.o uart.o
exceptions.o swihandler.o timer.o \
    irqhandler.o
```

이상에서 작성한 내용을 컴파일하면 아래와 같은 수행 결과가 나타난다.

```
...
SWI handler: syscall number-0x00000001, cnt-0x00000019
IRQ handler: rINTOFFSET-0x0000000a, rINTPND-
0x000000400
SWI handler: syscall number-0x00000001, cnt-
0x0000001a
IRQ handler: rINTOFFSET-0x0000000a, rINTPND-
0x000000400
SWI handler: syscall number-0x00000001, cnt-
0x0000001b
IRQ handler: rINTOFFSET-0x0000000a, rINTPND-
0x000000400
...
```

SWI handler와 IRQ handler가 무한히 호출되고 있다. INTOFFSET 레지스터의 값이 10, INTPND 레지스터의 10 번 비트가 켜져 있다. 그럼 다음으로 multitasking과 관련된 루틴을 추가하기 위해 두 개의 태스크(task0, task1)를 추가 하는 루틴을 작성해 보자. 먼저 task0.c, task1.c 파일을 다음과 같이 작성한다.

```
task0.c
int main()
{
    while(1) __asm__ ("swi 0x80");
}

task1.c
int main()
{
    while(1) __asm__ ("swi 0x81");
}
```

각각 <swi 0x80>, <swi 0x81> 시스템 콜을 무한히 호출하고 있다. 다음은 cute-boot.lds 파일을 아래와 같이 수정한다.

```
cute-boot.lds
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm",
"elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = 0x00000000;
    . = ALIGN(4);
    .text : {start.o(.text) *(.text)}
    . = ALIGN(4);
    .rodata : {*(.rodata)}
    . = ALIGN(4);
    .data : {*(EXCLUDE_FILE(cuteOS.bin,o task0.bin,o
task1.bin,o).data)}
    . = ALIGN(4);
    _bss_start = .;
    .bss : {*(.bss)}
    _bss_end = .;
    . = ALIGN(4);
    _os_start = .;
    .cuteOS : {cuteOS.bin,o}
    . = ALIGN(4);
    _os_end = .;
    . = ALIGN(4);
    _task0_start = .;
    .task0 : {task0.bin,o}
    . = ALIGN(4);
    _task0_end = .;
    . = ALIGN(4);
    _task1_start = .;
    .task1 : {task1.bin,o}
    . = ALIGN(4);
    _task1_end = .;
    _end = .;
}
```

cuteOS의 binary 이미지 뒤에 task0, task1의 binary 이미지를 차례로 오게 한다는 것을 알 수 있다. cute-boot.lds 파일의 구체적인 내용은 본 기사 2005년 10월호를 참고하기 바란다. start.S 파일의 마지막 부분을 다음과 같이 수정한다.

```
start.S
...
copy_loop:
    ldr r3,[r0],#4
    str r3,[r1],#4
    cmp r0,r2
    blt copy_loop
task0load:
    ldr r0,=_task0_start
    ldr r1,(OS_RAM_BASE+0x100000)
    ldr r2,=_task0_end
1:
    ldr r3,[r0],#4
    str r3,[r1],#4
    cmp r0,r2
    blt 1b
task1load:
    ldr r0,=_task1_start
    ldr r1,(OS_RAM_BASE+0x200000)
    ldr r2,=_task1_end
1:
    ldr r3,[r0],#4
    str r3,[r1],#4
    cmp r0,r2
    blt 1b
    ldr pc,_OS_RAM_BASE
...
```

task0, task1 이미지를 각각 (OS_RAM_BASE+0x100000), (OS_RAM_BASE+0x200000) 번지로 복사한다. start.S 파일

의 구체적인 설명은 본 기사 2005년 10월호를 참고하기 바란다. 다음은 main.c 파일의 내용이다.

```
main.c
#include <system.h>

void uart_init();

#define TASK0L2PT (unsigned int *) (0x100000+0x4400)
#define TASK1L2PT (unsigned int *) (0x100000+0x4800)
#define TASK0BASE (0x30200000|0xff<<4|0x2<<2|0x2)
#define TASK1BASE (0x30300000|0xff<<4|0x2<<2|0x2)
#define MASTERL1PT (unsigned int *) (0x100000)
#define TASK0L2PTBASE (0x30004400|0x03<<5|1<<4|0x1)
#define TASK1L2PTBASE (0x30004800|0x03<<5|1<<4|0x1)

void mmu_fill_l2pt(unsigned int *, unsigned int *,
unsigned int);
void flush_cache_tlb();
void mmu_map_l2pt(unsigned int *, unsigned int);

#define jump2task00 \
({ \
    unsigned long spsr; \
    unsigned long task0addr=0x400000; \
    unsigned long svcspr=0x408000; \
    __asm__ __volatile__( \
        "sub sp,%2,#4\n" \
        "mrs %0,spsr\n" \
        "bic %0,%0,#0xff\n" \
        "orr %0,%0,#0x50\n" \
        "msr spsr_c,%0\n" \
        "movs pc,%1" \
        :: "r" (spsr), "r" (task0addr) \
        , "r" (svcspr) \
    )
}
```



```

: "memory", "cc");
})

void kmain(void)
{
    uart_init();
    timer_init();
    mmu_fill_l2pt(TASK0L2PT, TASK0L2PT+8, TASK0
BASE);
    mmu_fill_l2pt(TASK1L2PT, TASK1L2PT+8, TASK1
BASE);
    flush_cache_tlb();
    mmu_map_l2pt(MASTERL1PT+4, TASK0L2PTBASE);
    uart_puts("jump to task0\n");
    jump2task0();
}

```

kmain 함수를 보기 전에 먼저 앞부분에 정의되어 있는 매크로를 살펴보기로 하자. TASK0L2PT, TASK1L2PT 매크로는 각각 task0, task1을 관리하는 level 2 page table의 가상 주소를 나타낸다. TASK0BASE, TASK1BASE 매크로는 각각 task0, task1의 물리 주소와 AP, C, B 속성 등을 나타낸다.

MASTERL1PT 매크로는 master page table(level 1 page table)의 가상 주소를 나타낸다. TASK0L2PTBASE, TASK1L2PTBASE 매크로는 각각 task0, task1을 관리하는 level 2 page table의 물리 주소와 Domain, C, B 속성 등을 나타낸다. jump2task0 매크로는 task0 루틴으로 뛰기 위해 정의했으며, 뒤에서 살펴보기로 한다. 그럼 kmain 함수를 살펴보기로 하자.

kmain 함수는 mmu_fill_l2pt 함수를 이용해 task0, task1의 level 2 page table을 초기화한다. kmain 함수의 마지막 부분에서 task0 루틴으로 뛰기 위해 flush_cache_tlb 함수를 이용해 cache와 tlb를 비우고, mmu_map_l2pt 함수

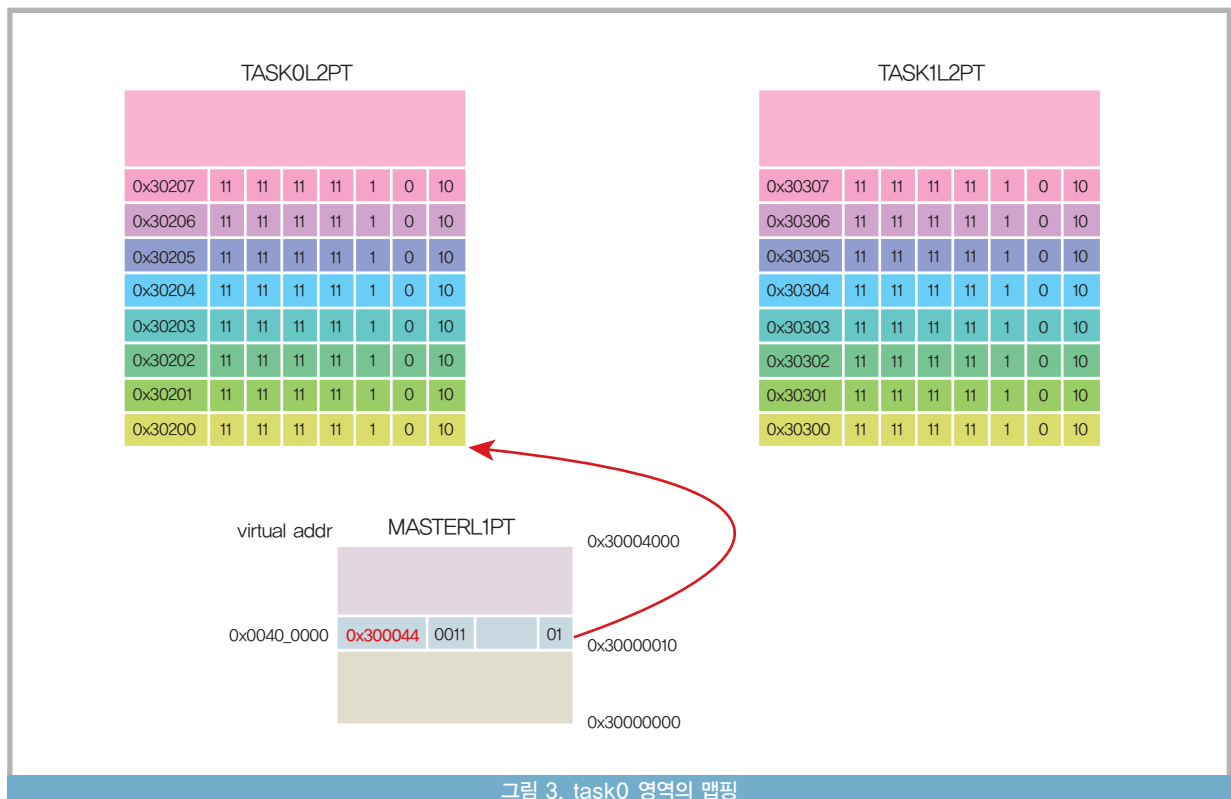


그림 3. task0 영역의 맵핑

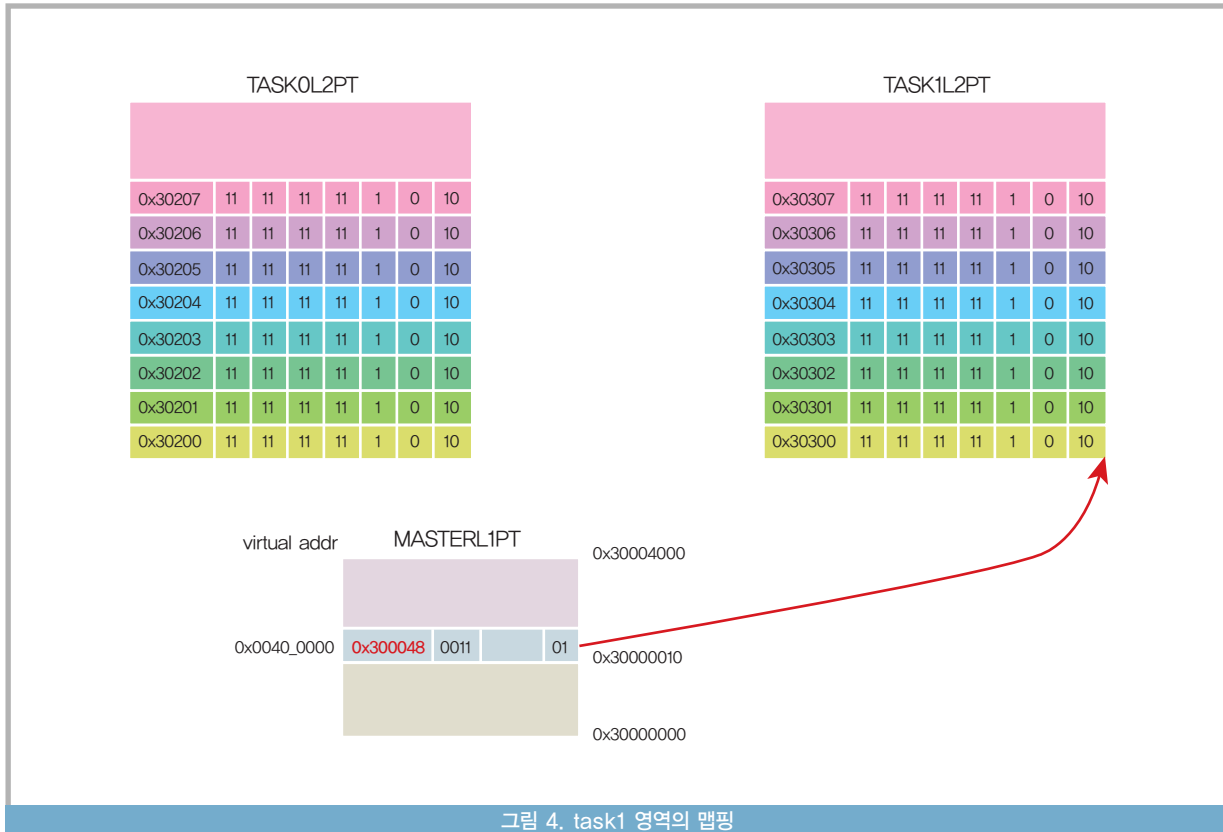


그림 4. task1 영역의 맵핑

를 이용해 master page table에 task0의 level 2 page table을 맵핑시킨다. mmu_fill_l2pt, flush_cache_tlb, mmu_map_l2pt 함수는 뒤에서 살펴볼 mmu.S 파일에 정의되어 있다.

그림 3은 이 과정에 의해서 생성된 task0, task1의 level 2 page table과 master page table을 나타낸 것이다. task0에서 task1로 문맥 전환을 수행할 경우 master page table의 5번째 엔트리에 task1의 level 2 page table을 맵핑시키며, 그림 4와 같이 나타낼 수 있다.

가상 주소에 대한 자세한 내용은 본 기사 2005년 11월 호를 참조하기 바란다. kmain 함수의 마지막 부분에서는 jump2task0 매크로를 이용해 task0 루틴으로 뛰며, 이후에 kmain 루틴으로 다시 돌아오지 않는다.

jump2task0 매크로는 inline assembly를 이용해 정의했으며, SVC 모드의 sp 레지스터를 (0x100000-4) 값으로 초기화한다. 왜냐하면, 후에 task 루틴에서 swi 명령어를 이용

해 SVC 모드로 진입할 경우 SVC 모드에서 사용할 스택의 주소 값이 필요하기 때문이다.

SVC 모드의 spsr 레지스터의 하위 8 비트를 0으로 초기화하고 0x50으로 채운 후에, movs 명령어를 이용해 가상 주소 0x400000 번지로 점프를 수행한다. 이때 CPU 코어의 모드는 USR 모드로 바뀌며, 하드웨어 인터럽트는 활성화된다.

이상에서 main.c 파일의 내용을 살펴보았다. 다음은 mmu.S 파일의 내용을 살펴보기로 하자.

```
mmu.S
.globl flush_cache_tlb
flush_cache_tlb:
    mov r0,#0
    mcr p15,0,r0,c7,c7,0
```

```

mcr p15,0,r0,c8,c7,0
mov pc,lr
.globl mmu_fill_l2pt
    @ r0 - level 2 page table start
    @ r1 - level 2 page table end
    @ r2 - task base addr and attributes
mmu_fill_l2pt:
1:
    str r2,[r0],#4
    cmp r0,r1
    addlt r2,r2,#0x1000
    blt 1b
    mov pc,lr
.globl mmu_map_l2pt
    @ r0 - level 1 page table entry
    @ r1 - level 2 page table base addr and attributes
mmu_map_l2pt:
    str r1,[r0]
    mov pc,lr

```

mmu.S 파일에는 flush_cache_tlb, mmu_fill_l2pt, mmu_map_l2pt 함수가 정의되어 있다. 각각의 함수는 cache와 tlb에 대한 초기화, level 2 page table에 대한 설정, level 2 page table을 level 1 page table에 맵핑시키는 역할을 수행한다.

각 함수들에 대한 구체적인 설명은 본 기사 2005년 11월 호에서 다룬 mmusetup.S 파일의 mmusetup 함수를 참조하기 바란다.

이상에서 mmu.S 파일의 내용을 살펴보았다. 다음은 Makefile의 내용이다.

```

Makefile
.c.o:

```

```

arm-linux-gcc $< -c -include
.S.o:
    arm-linux-gcc $< -c

# BOOTLOADER
OBJ = start.o led.o clksetup.o memsetup.o cuteOS.bin.o
task0.bin.o task1.bin.o
cute-boot: $(OBJ)
    arm-linux-ld $(OBJ) -o cute-boot -Ttext
0x00000000 -N -T cute-boot.lds
    arm-linux-objcopy cute-boot cute-boot.bin -O
binary
start.o: start.S
    arm-linux-gcc start.S -c -DOS_RAM_BASE
=0x30100000

# KERNEL
KERNEL_OBJ = head.o main.o mmusetup.o uart.o
exceptions.o swihandler.o timer.o \
irqhandler.o mmu.o
cuteOS.bin.o: cuteOS
    arm-linux-ld -r -o cuteOS.bin.o -b binary cuteOS.
bin
cuteOS: $(KERNEL_OBJ)
    arm-linux-ld $(KERNEL_OBJ) -o cuteOS -Ttext 0x00
000000 -N
    arm-linux-objcopy cuteOS cuteOS.bin -O binary

# TASK0
task0.bin.o: task0
    arm-linux-ld -r -o task0.bin.o -b binary task0.bin
task0.o: task0.o
    arm-linux-ld task0.o -o task0 -e main -Ttext 0x400

```

```

000 -N
arm-linux-objcopy task0 task0.bin -O binary

# TASK1
task1.bin.o: task1
arm-linux-ld -r -o task1.bin.o -b binary task1.bin
task1: task1.o
arm-linux-ld task1.o -o task1 -e main -Ttext 0x4000
00 -N
arm-linux-objcopy task1 task1.bin -O binary
clean:
rm -f *.o
rm -f cute-boot
rm -f cute-boot.bin
rm -f cuteOS
rm -f cuteOS.bin
rm -f task0
rm -f task0.bin
rm -f task1
rm -f task1.bin

```

먼저 cute-boot 파일을 만들기 위해 OBJ 변수에 task0.bin.o, task1.bin.o 파일과 함께, cuteOS 파일을 만들기 위해 KERNEL_OBJ 변수에 mmu.o 파일을 추가했다. 그리고 task0, task1을 컴파일 하기 위해 각각의 dependency line을 추가했다.

이상에서 Makefile을 살펴보았다. 이상 작성한 내용을 컴파일하면 다음과 같은 결과가 나타난다.

```

...
IRQ handler: rINTOFFSET-0x0000000a, rINTPND-
0x00000400
SWI handler: syscall number-0x00000080, cnt-

```

```

0x00000d3f
IRQ handler: rINTOFFSET-0x0000000a, rINTPND-
0x00000400
SWI handler: syscall number-0x00000080, cnt-
0x00000d40
IRQ handler: rINTOFFSET-0x0000000a, rINTPND-
0x00000400
SWI handler: syscall number-0x00000080, cnt-
0x00000d41
...

```

task0 루틴에 의해 무한히 eventsSWIHandler 함수가 호출된다. 이상에서 두 개의 태스크(task0, task1)를 추가하는 루틴을 작성해 보았다.

다음 호에서는 multitasking을 수행하기 위해 필수적으로 필요한 문맥 전환 루틴과 스케줄링 루틴을 작성해 보고, 이 루틴들을 시스템 콜 처리 루틴과 하드웨어 인터럽트 처리 루틴에 추가해 보자. 또, 하드웨어 인터럽트 처리 루틴과 태스크 루틴간의 간섭, 태스크와 태스크 루틴간의 간섭이 발생하는 원인을 파악해 보기로 하자. **R_{time}**

참고자료

S3C2440A 32-BIT CMOS MICROCONTROLLER USER'S MANUAL Revision 1
 ARM System Developer's Guide : Designing and Optimizing System Software(저자: Andrew Sloss, Dominic Symes, Chris Wright/Morgan Kaufmann)