

설계와 구현을 통한 임베디드 OS의 이해와 응용 ③

cuteOS의 시작



지난 호까지 'cuteOS' 구현을 위한 개발환경 구성 방법을 알아보았다. 이번 호에서는 지난 호에 이어 부트로더의 나머지 부분과 cuteOS의 시작 부분을 직접 작성해 보도록 하겠다.

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

먼저 다음과 같은 순서로 부트로더의 나머지 부분을 작성해 나갈 것이다.

1. FCLK, HCLK, PCLK, UCLK 등 CPU와 주변 버스로 공급되는 clock의 속도를 설정한다.
2. SDRAM 등을 접근하기 위해 메모리 컨트롤러를 초기화한다.
3. cuteOS 이미지를 RAM 상으로 복사한 후 cuteOS의 시작 위치로 켜낸다.

다음으로 cuteOS는 최소한의 어셈블리 파일과 C 파일로 구성한다. 다음과 같은 순서로 cuteOS의 시작 부분을 작성해 나가겠다.

1. C 루틴으로 뛰기 위해 SVC32 모드의 스택을 초기화해야 한다.
2. C 루틴으로 켜낸다.

cuteOS의 본격적인 작성은 다음 호부터 진행하기로 하고, cache와 MMU 설정 부분은 그때 다루기로 하겠다. 그럼 지난 호에 이어 start.S 파일의 나머지 부분을 살펴보자.

```
start.S
...
mov    r0,#0x00000010
bl     led_on

bl     clksetup

bl     led_off

mov    r0,#0x00000020
bl     led_on
bl     memsetup
bl     led_off
```

```

mov    r0,#0x00000040
bl     led_on

osload:
ldr     r0,_OS_ROM_BASE
ldr     r1,_OS_RAM_BASE
ldr     r2,_OS_END

copy_loop:
ldr     r3,[r0],#4
str     r3,[r1],#4
cmp     r0,r2
blt     copy_loop

ldr     pc,_OS_RAM_BASE
.balign 4

_OS_ROM_BASE:
.word _os_start

_OS_RAM_BASE:
.word OS_RAM_BASE

_OS_END:
.word _os_end

```

구체적으로 살펴보자.

```

...
mov    r0,#0x00000010
bl     led_on

```

이 부분은 이전 기사에서 작성했던 start.S 파일의 마지막 부분이다. <1: b 1b> 명령어가 빠졌으니 주의하기 바란다.

```
bl    clksetup
```

여기서는 FCLK, HCLK, PCLK, UCLK 등 CPU와 주변 버스로 공급되는 clock의 속도를 설정하기 위해 clksetup 함수를 호출한다. 이 함수는 clksetup.S 파일에 정의되어 있으며 뒤에서 구체적으로 살펴볼 것이다.

```

bl     led_off

mov    r0,#0x00000020
bl     led_on

```

이 부분은 디버깅을 위해 led를 켜다가 두 번째 led를 켜는 부분이다.

```
bl    memsetup
```

이 부분은 메모리 컨트롤러를 초기화하기 위해 memsetup 함수를 호출하는 부분이다. 이 함수는 memsetup.S 파일에 정의되어 있으며, 역시 뒤에서 구체적으로 살펴보기로 한다.

```

bl     led_off

mov    r0,#0x00000040
bl     led_on

```

이 부분은 디버깅을 위해 led를 켜다가 세 번째 led를 켜는 부분이다. 다음은 편의상 파일의 마지막 부분을 먼저 살펴보겠다.

```
.balign 4
```

이 부분은 이후에 올 코드를 4바이트 경계에 놓겠다는 어

셈블러 지시어이다.

```
_OS_ROM_BASE:
.word _os_start
```

여기서 <word>는 32비트 숫자를 정의하는 어셈블러 지시어이다. _os_start는 부트 ROM 상에서의 cuteOS 이미지의 시작 위치를 나타내며, 뒤에서 보게 될 'cute-boot.lds'라는 링커 스크립트 파일에 정의되어 있다.

```
_OS_RAM_BASE:
.word OS_RAM_BASE
```

OS_RAM_BASE는 매크로로서 Makefile에서 0x30100000번지로 정의해 놓았으며, RAM 상에서 cuteOS가 놓일 시작 위치를 나타낸다. 즉, cuteOS는 물리적으로 0x30100000번지에 놓이게 된다. Makefile의 내용은 뒤에서 살펴보기로 하자.

```
_OS_END:
.word _os_end
```

_os_end는 부트 ROM 상에서의 cuteOS 이미지의 마지막 위치를 나타내며, 역시 "cute-boot.lds" 링커 스크립트 파일에 정의되어 있다.

그럼 다시 파일의 앞부분으로 가 보자.

```
osload:

ldr    r0,_OS_ROM_BASE
ldr    r1,_OS_RAM_BASE
ldr    r2,_OS_END
```

이 부분은 r0 레지스터에 부트 ROM 상에 있는 cuteOS 이미지의 시작 주소를, r1 레지스터에 0x30100000 값을, r2 레지스터에 부트 ROM 상에 있는 cuteOS 이미지의 마지막 주소를 넣는 부분이다.

```
copy_loop:
ldr    r3,[r0],#4
str    r3,[r1],#4
cmp    r0,r2
blt    copy_loop
```

이 부분은 ROM 상에 있는 cuteOS 이미지를 4바이트씩 읽어서 램 상으로 복사하는 부분이다. cuteOS 이미지의 마지막 부분까지 복사하면 루프를 빠져 나간다.

```
ldr    pc,_OS_RAM_BASE
```

여기서는 RAM 상에 있는 cuteOS로 된다. 이로써 부트로터의 역할은 끝나게 된다. cuteOS는 head.S 파일과 main.c 파일로 구성되며 뒤에서 살펴보기로 한다.

그러면 cute-boot.lds 파일의 내용을 살펴보기로 하자. 먼저 파일의 내용은 다음과 같다.

```
cute-boot.lds
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm",
"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),data)}

. = ALIGN(4);
__bss_start = .;
.bss : {*(.bss)}
_end = .;

. = ALIGN(4);
__os_start = .;
.cuteOS : {cuteOS,bin,o}
. = ALIGN(4);
__os_end = .;
}
```

cute-boot.lds 파일은 링커 스크립트로써 arm-linux-ld가 최종 결과 파일을 만들어 내는 과정에서 설계도 역할을 한다. 그러면 파일의 내용을 좀 더 자세히 살펴보자.

OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm", "elf32-littlearm")

이 부분은 arm-linux-ld가 만들어 낼 최종 결과 파일의 포맷을 나타낸다. 즉, little endian 포맷의 파일을 생성할 것인지, big endian 포맷의 파일을 생성할 것인지를 결정하는 역할을 한다. 링커 스크립트 내의 <OUTPUT_FORMAT> 키워드는 arm-linux-ld 명령어의 <-EB> 또는 <-EL> 옵션과 같이 사용할 수 있다. 만약 arm-linux-ld 명령어를 <-EB> 옵션과 함께 사용하였다면 두 번째 항목에 해당하는 포맷의 파일을 생성해 내며(elf32-bigarm), arm-linux-ld 명령어를 <-EL> 옵션과 함께 사용하였다면 세 번째 항목에 해당하는 파일 포맷을 생성해낸다(elf32-littlearm). 이 두 옵션을

모두 사용하지 않을 경우 디폴트로 첫 번째 항목에 해당하는 파일 포맷을 생성해 낸다(elf32-littlearm). 여기서는 elf32-littlearm 포맷의 파일을 생성하기로 한다.

OUTPUT_ARCH(arm)

이 부분은 최종 결과 파일이 동작할 CPU의 아키텍처를 나타낸다. 즉, 이 파일은 ARM CPU 상에서 동작한다는 의미이다.

ENTRY(_start)

최종 결과 파일의 시작 지점을 나타낸다. 즉, 여기서 파일의 시작 지점은 _start가 된다. _start는 start.S 파일에 정의되어 있다.

SECTIONS

```
{
...
}
```

이 부분은 링커(arm-linux-ld)가 입력 파일들의 세션들을 결과 파일의 어떤 세션들로 위치시킬지를 결정하는 역할을 한다. 다음의 예를 보자.

.text : {start.o(.text) *(.text)}

이 문장은 start.o 입력 파일의 .text 세션과 그 외의 입력 파일들(*)의 .text 세션을 결과 파일의 .text 세션에 위치시키는 역할을 한다. 참고로 C 파일을 컴파일하여 오브젝트 파일을 생성할 경우, 기본적으로 함수는 .text 세션에, 전역 변수는 .data 세션에, 초기화 되지 않은 전역 변수는 .bss 세션에 놓인다. 그 외에도 여러 가지 세션들이 있으며, 또한 새로운

세션을 정의해서 쓸 수도 있다.

```
. = 0x00000000;
```

이 부분은 현재의 위치는 0x00000000 번지라는 의미이다.

```
. = ALIGN(4);
```

이 부분은 현재의 위치를 4바이트 경계에 놓겠다는 의미이다.

```
.rodata : {*(.rodata)}
```

이 부분은 모든 입력 파일의 .rodata 세션을 결과 파일의 .rodata 세션에 놓겠다는 의미이다.

```
.data : {*(EXCLUDE_FILE(cuteOS.bin.o).data)}
```

이 부분은 cuteOS.bin.o 파일을 제외한 모든 입력 파일의 .data 세션을 결과 파일의 .data 세션에 놓겠다는 의미이다. cuteOS.bin.o 파일은 cuteOS의 순수 바이너리 이미지이다. cuteOS.bin.o 파일을 만들어 내는 과정은 뒤에서 살펴보기로 하자.

```
__bss_start = .;
```

이 부분은 __bss_start 심벌의 주소 값은 현재 위치와 같다는 의미이다. arm-linux-nm 명령어를 이용하여 이 부분을 확인해 보기 바란다.

```
.bss : {*(.bss)}
```

이 부분은 모든 입력 파일의 .bss 세션을 결과 파일의 .bss 세션에 놓겠다는 의미이다.

```
_os_start = .;
cuteOS : {cuteOS.bin.o}
    = ALIGN(4);
_os_end = .;
```

이 부분은 결과 파일의 .cuteOS 세션에 cuteOS.bin.o 파일을 놓는 역할을 한다. _os_start와 _os_end 심벌이 .cuteOS 세션의 앞뒤에 오는 걸 확인해 볼 수 있다. 즉, _os_start 심벌은 cuteOS 이미지의 시작위치를 나타내며, _os_end 심벌은 cuteOS 이미지의 끝 위치를 나타낸다.

이상에서 링커 스크립트의 내용을 알아보았다. 다음은 Makefile의 내용을 들여다보기로 하자. 먼저 Makefile의 내용은 다음과 같다.

```
Makefile
# BOOTLOADER
OBJ = start.o led.o clksetup.o memsetup.o cuteOS.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

led.o: led.S
    arm-linux-gcc led.S -c

clksetup.o: clksetup.S
    arm-linux-gcc clksetup.S -c
```

```

memsetup.o: memsetup.S
    arm-linux-gcc memsetup.S -c

# KERNEL
cuteOS.bin.o: cuteOS
    arm-linux-ld -r -o cuteOS.bin.o -b binary cuteOS.bin

cuteOS: head.o main.o
    arm-linux-ld head.o main.o -o cuteOS -Ttext
    0x30100000 -N
    arm-linux-objcopy cuteOS cuteOS.bin -O binary

head.o: head.S
    arm-linux-gcc head.S -c

main.o: main.c
    arm-linux-gcc main.c -c

clean:
    rm -f *.o
    rm -f cute-boot
    rm -f cute-boot.bin
    rm -f cuteOS
    rm -f cuteOS.bin
  
```

Makefile에 대한 기본적인 내용은 이전 기사에서 언급하였다. 여기서는 새로 추가되거나 변경된 부분 위주로 설명하기로 하겠다. Makefile은 크게 두 부분으로 나뉘어져 있다. 하나는 부트로더를 만들어내는 부분이고, 또 하나는 커널을 만들어내는 부분이다. 위의 Makefile을 이용하여 make 명령어를 수행할 경우 다음과 같이 나타난다.

```

$ make
arm-linux-gcc start.S -c -DOS_RAM_BASE=0x30100000
arm-linux-gcc led.S -c
arm-linux-gcc clksetup.S -c
  
```

```

arm-linux-gcc memsetup.S -c
arm-linux-gcc head.S -c
arm-linux-gcc main.c -c
arm-linux-ld head.o main.o -o cuteOS -Ttext
    0x30100000 -N
arm-linux-objcopy cuteOS cuteOS.bin -O binary
arm-linux-ld -r -o cuteOS.bin.o -b binary cuteOS.bin
arm-linux-ld start.o led.o clksetup.o memsetup.o
    cuteOS.bin.o -o cute-boot -Ttext 0x00000000 -N -T
    cute-boot.lds
arm-linux-objcopy cute-boot cute-boot.bin -O binary
  
```

여기서는 arm-linux-gcc를 이용하여 start.S, led.S, clksetup.S, memsetup.S, head.S, main.c 파일을 각각 start.o, led.o, clksetup.o, memsetup.o, head.o, main.o 파일로 만든다.

그리고 arm-linux-ld 명령어를 이용하여 head.o, main.o 파일로 cuteOS 결과 파일을 만들어낸다(head.S 파일과 main.c 파일은 뒤에서 살펴보기로 하자). cuteOS가 동작할 메모리 위치는 0x30100000 번지가 된다. -N 옵션은 .data 세션을 page 경계에 놓지 말고 .text 세션의 바로 뒤에 붙이는 역할을 한다. 파일의 크기를 줄이기 위해 이 옵션을 사용했다.

그리고 arm-linux-objcopy 명령어를 이용하여 cuteOS 파일로 순수 바이너리 이미지인 cuteOS.bin 파일을 만들어낸다. 이 파일이 실제 RAM 상에서 동작할 커널의 이미지이다.

다음은 cuteOS.bin 입력 파일을 cuteOS.bin.o 결과 파일로 만드는 과정이다. 여기서 <-b binary> 옵션은 입력 파일 cuteOS.bin 파일이 바이너리 파일이란 의미이며, -o는 output을 의미한다. 또한 -r 옵션은 arm-linux-ld의 입력 파일로 다시 사용할 수 있도록 결과 파일을 만들라는 의미이며, relocateable의 약자이다. 이 파일은 다음에 오는 arm-linux-ld 명령어를 이용하여 cute-boot 결과 파일을 만드는 과정에서 입력 파일로 사용하게 된다.

즉, arm-linux-ld 명령어를 이용하여 start.o led.o clksetup.o memsetup.o cuteOS.bin.o 입력 파일로 cute-

boot 결과 파일을 만든다. <-T cute-boot.lds> 옵션은 arm-linux-ld가 사용할 링커 스크립트 파일로 cute-boot.lds 파일을 사용하라는 의미이다.

마지막으로 arm-linux-objcopy 명령어를 이용하여 cute-boot.bin 파일을 만든다. 이 파일을 JTAG를 이용하여 ROM 상에 다운 받아 실행하면 된다.

다음은 clksetup.S 파일의 내용이다.

```
clksetup.S
#define LOCKTIME 0x4c000000
#define UPLLCON 0x4c000008
#define MPLLCON 0x4c000004
#define CLKDIVN 0x4c000014
#define CAMDIVN 0x4c000018

#define UPLLVAL ((60<<12)|(4<<4)|1)
#define MPLLVAL ((0x6e<<12)|(3<<4)|1)

.globl clksetup
clksetup:
    ldr r0,=LOCKTIME
    mov r1,#0xffffffff
    str r1,[r0]

    ldr r0,=CAMDIVN
    mov r1,#0
    str r1,[r0]

    ldr r0,=CLKDIVN
    mov r1,#0xd
    str r1,[r0]

    mrc p15,0,r1,c1,c0,0
    orr r1,r1,#0xc0000000
    mcr p15,0,r1,c1,c0,0

    ldr r0,=UPLLCON
```

```
ldr r1,=UPLLVAL
str r1,[r0]
```

```
ldr r0,=MPLLCON
ldr r1,=MPLLVAL
str r1,[r0]
```

```
mov pc,lr
```

파일의 내용을 살펴보기 전에 먼저 S3C2440A의 clock과 관련된 기능을 이해하고 넘어가자.

S3C2440A에는 CPU, AHB 버스, APB 버스, USB 버스로 clock을 공급하는 clock 생성기가 있다. 이 clock 생성기는 MPLL과 UPLL을 포함하고 있다. MPLL에서는 CPU, AHB 버스, APB 버스로 공급하는 clock을 생성하며, UPLL에서는 USB 버스로 공급하는 clock을 생성해 낸다.

MPLL로부터 CPU로 공급하는 clock을 FCLK, FCLK을 1,2,3,4, 또는 6으로 나누어 AHB 버스로 공급하는 clock을 HCLK, HCLK를 1 또는 2로 나누어 APB 버스로 공급하는 clock을 PCLK이라 한다. 또 UPLL의 값을 1 또는 2로 나누어 USB 버스로 공급하는 clock을 UCLK이라 한다.

S3C2440A의 clock 생성기는 외부 크리스털(XTiPl)로부터 16.9344MHz의 값을 입력 받아 MPLL과 UPLL로 공급을 한다. 그러면 이 두 PLL은 PLL에 대한 설정에 따라 CPU와 버스에서 사용할 적절한 속도의 높은 주파수의 clock을 생성해 낸다. 참고로 필자는 MPLL로부터 나오는 clock의 속도가 399.65MHz가 되도록 MPLL을 설정했으며, UPLL로부터 나오는 clock의 속도는 95.96MHz가 되도록 UPLL을 설정했다.

Power-On Reset시, 즉, S3C2440A에 전원을 넣을 때, PLL이 일시적으로 불안정하기 때문에 외부 크리스털(XTiPl)로부터 오는 clock(16.9344MHz)을 직접 FCLK으로 공급하며, 부팅 초기 루틴에서 PLLCON 레지스터를 이용하여 PLL의 clock 속도를 설정한 후에야 MPLL로부터 나오는 clock을 FCLK으로 공급할 수 있다.

그런데 PLL을 설정할 경우 PLL에서 나오는 clock이 일시

적으로 불안정하게 되며, 따라서 일정기간 clock의 공급을 차단하게 되는데 그 차단되는 시간을 LOCKTIME 레지스터를 이용하여 조절한다.

즉, PLL을 설정한 후 설정된 LOCKTIME 레지스터 값에 해당하는 시간동안 MPLL로부터 나오는 clock이 FCLK으로 전달되지 않는다. 설정된 LOCK TIME 레지스터 값에 해당하는 시간이 흐른 이후에야 비로소 MPLL로부터 FCLK으로 clock이 공급된다. 이 경우 LOCKTIME 레지스터의 시간 기준은 외부 크리스탈(XTiPl)로부터 오는 clock에 의해 결정된다.

또한 PLL로부터 CPU나 각각의 버스로 clock이 공급되는 도중이라도 그 설정 내용을 변경할 경우 LOCKTIME 레지스터에 설정된 값에 해당하는 시간이 흐른 이후에야 CPU나 각각의 버스로 clock이 공급된다.

그러면 clksetup.S의 내용을 자세히 들여다보기로 하자.

```
ldr r0,=LOCKTIME
mov r1,#0xffffffff
str r1,[r0]
```

이 부분은 LOCK TIME 레지스터의 내용을 설정하는 부분이다. LOCKTIME 레지스터는 0x4c000000 번지에 있으며, 표 1과 같은 구조로 되어 있다(254P 참조).

| LOCK TIME COUNT REGISTER (LOCKTIME) | | | | |
|-------------------------------------|------------|-----|------------------------------|-------------|
| Register | Address | R/W | Description | Reset Value |
| LOCKTIME | 0x4C000000 | R/W | PLL lock time count register | 0xFFFFFFFF |

| LOCKTIME | Bit | Description | Initial State |
|----------|---------|--|---------------|
| U_LTIME | [31:16] | UPLL lock time count value for UCLK. (U_LTIME > 300uS) | 0xFFFF |
| M_LTIME | [15:0] | MPLL lock time count value for FCLK, HCLK, and PCLK (M_LTIME > 300uS) | 0xFFFF |

표 1. LOCK TIME 카운트 레지스터

```
ldr r0,=CAMDIVN
mov r1,#0
str r1,[r0]
```

```
ldr r0,=CLKDIVN
mov r1,#0xd
str r1,[r0]
```

이 부분은 FCLK, HCLK, PCLK의 속도 비율을 1:4:8로 맞추는 부분이다. CAMDIVN 레지스터와 CLKDIVN 레지스터는 각각 0x4c000018, 0x4c000014 번지에 있다.

CAMERA CLOCK DIVIDER (CAMDIVN) REGISTER

| Register | Address | R/W | Description | Reset Value |
|----------|------------|-----|-------------------------------|-------------|
| CAMDIVN | 0x4C000018 | R/W | Camera clock divider register | 0x00000000 |

표 2. CAMDIVN 레지스터

먼저 CAMDIVN 레지스터는 표 2와 같은 구조로 되어 있다(259P 참조). 또한 CAMDIVN 레지스터의 속도 비율과 관련된 9번 비트는 표 3과 같은 의미를 갖는다.

| HCLK4_HALF | [9] | HDIVN division rate change bit, when CLKDIVN[2:1]=10b. 0: HCLK = FCLK/4 1: HCLK = FCLK/8 Refer the CLKDIV register. | 0 |
|------------|-----|--|---|
|------------|-----|--|---|

표 3. CAMDIVN 레지스터의 속도 비율

다음으로 CLKDIVN 레지스터는 표 4와 같은 구조로 되어 있다. (258P 참조)

CLOCK DIVIDER CONTROL (CLKDIVN) REGISTER

| Register | Address | R/W | Description | Reset Value |
|----------|------------|-----|--------------------------------|-------------|
| CLKDIVN | 0x4C000014 | R/W | Clock divider control register | 0x00000000 |

| CLKDIVN | Bit | Description | Initial State |
|-----------|-------|--|---------------|
| DIVN_UPLL | [3] | UCLK select register(UCLK must be 48MHz for USB) 0: UCLK = UPLL clock 1: UCLK = UPLL clock / 2 Set to 0, when UPLL clock is set as 48MHz Set to 1, when UPLL clock is set as 96MHz. | 0 |
| HDIVN | [2:1] | 00: HCLK = FCLK/1 01: HCLK = FCLK/2 10: HCLK = FCLK/4 when CAMDIVN[9] = 0. HCLK = FCLK/8 when CAMDIVN[9] = 1. 11: HCLK = FCLK/3 when CAMDIVN[8] = 0. HCLK = FCLK/6 when CAMDIVN[8] = 1. | 00 |
| PDIVN | [0] | 0: PCLK has the clock same as the HCLK/1. 1: PCLK has the clock same as the HCLK/2. | 0 |

표 4. CLOCK DIVIDER 컨트롤 (CAMDIVN) 레지스터


```
mrc p15,0,r1,c1,c0,0
orr r1,r1,#0xc0000000
mcr p15,0,r1,c1,c0,0
```

이 부분은 CPU의 버스 모드를 fast 버스 모드로부터 asynchronous 버스 모드로 변경하는 부분이다. CLKDIVN 레지스터의 HDIVN 부분이 '0'이 아니고, CPU 버스 모드가 fast 버스 모드일 경우, CPU는 HCLK에 동기해서 동작을 하기 때문에 위의 코드가 필요하다. 코드의 내용을 좀 더 살펴보자.

```
mrc p15,0,r1,c1,c0,0
```

이 명령어는 15번 코프로세서의 c1 레지스터의 값을 r1 레지스터로 가져오는 역할을 한다. 15번 코프로세서는 cache나 MMU에 대한 설정 등을 할 때도 사용한다. c1 레지스터는 컨트롤 레지스터이다.

```
orr r1,r1,#0xc0000000
mcr p15,0,r1,c1,c0,0
```

c1 레지스터의 최상위 2비트를 이진수 11로 설정함으로써 CPU의 버스 모드를 asynchronous 버스 모드로 변경한다.

```
ldr r0,=UPLLCON
ldr r1,=MPLLVAL
str r1,[r0]
```

이 부분은 USB 버스로 공급하는 clock의 속도를 (95.96/2) MHz로 설정하는 부분이다.

```
ldr r0,=MPLLCON
```

```
ldr r1,=MPLLVAL
str r1,[r0]
```

이 부분은 CPU로 공급하는 clock의 속도를 399.65MHz로 설정하는 부분이다.

MPLLCON 레지스터와 UPLLCON 레지스터는 표 5와 같은 구조로 되어 있다(255P 참조).

| PLL CONTROL REGISTER (MPLLCON & UPLLCON) | | | | |
|--|------------|-----|-----------------------------|-------------|
| Register | Address | R/W | Description | Reset Value |
| MPLLCON | 0x4C000004 | R/W | MPLL configuration register | 0x00096030 |
| UPLLCON | 0x4C000008 | R/W | UPLL configuration register | 0x0004d030 |

| PLLCON | Bit | Description | Initial State |
|--------|---------|----------------------|---------------|
| MDIV | [19:12] | Main divider control | 0x95 / 0x4d |
| PDIV | [9:4] | Pre-divider control | 0x03 / 0x03 |
| SDIV | [1:0] | Post divider control | 0x0 / 0x0 |

표 5. PLL 컨트롤 레지스터

MPLL과 UPLL로부터 생성되는 clock의 속도는 다음의 공식을 따른다. 참고로 S3C2440A에서 Fin은 16.9344MHz이다.

$$M_{pll} = (2 * m * Fin) / (p * 2^s)$$

$$m = (MDIV + 8)$$

$$p = (PDIV + 2)$$

$$s = SDIV$$

$$U_{pll} = (m * Fin) / (p * 2^s)$$

$$m = (MDIV + 8)$$

$$p = (PDIV + 2)$$

$$s = SDIV$$

이상에서 clksetup.S 파일의 내용을 살펴보았다. 다음은 memsetup.S 파일의 내용이다.

```
memsetup,S
#define BWSCON 0x48000000
```

```
.globl memsetup
memsetup:
    ldr r0,=SMRDATA
    ldr r1,=_start
    sub r0,r0,r1
    ldr r1,=BWSCON
    add r2,r0,#13*4
0:
    ldr r3,[r0],#4
    str r3,[r1],#4
    cmp r2,r0
    bne 0b

    mov pc,lr

.ltorg
SMRDATA:
    .word 0x221d4d20
    .word 0x00000700
    .word 0x00000700
    .word 0x00003f4c
    .word 0x00001f4c
    .word 0x00003f4c
    .word 0x00000700
    .word 0x00018009
    .word 0x00018009
    .word 0x008e0459
    .word 0x00000031
    .word 0x00000030
    .word 0x00000030
```

S3C2440A는 총 8개의 메모리 뱅크를 지원하며, 각각의 뱅크는 최대 128MB의 크기를 가질 수 있다. 이 중 앞의 6개의 뱅크에는 ROM이나 SRAM을 붙일 수 있으며, 마지막 2개의 뱅크에는 ROM, SRAM, 또는 SDRAM을 붙일 수 있다. 본인이 사용하는 보드에는 첫 번째 뱅크에 1MB 크기의 NOR flash ROM, 7번째 뱅크와 8번째 뱅크에 각각 32MB 크기의 SDRAM이 있다.

S3C2440A의 메모리 뱅크는 다음의 13개의 레지스터를 이용하여 설정할 수 있다. 각각의 레지스터의 역할을 간략하게 살펴보기로 하자.

| | |
|---------------------|--------------|
| BWSCON (0x48000000) | |
| BANKCON0 | (0x48000004) |
| BANKCON1 | (0x48000008) |
| BANKCON2 | (0x4800000c) |
| BANKCON3 | (0x48000010) |
| BANKCON4 | (0x48000014) |
| BANKCON5 | (0x48000018) |
| BANKCON6 | (0x4800001c) |
| BANKCON7 | (0x48000020) |
| REFRESH | (0x48000024) |
| BANKSIZE | (0x48000028) |
| MRSRB6 | (0x4800002c) |
| MRSRB7 | (0x48000030) |

- BWSCON: BWSCON 레지스터는 각각의 뱅크에 대한 데이터 버스 폭에 대한 설정(예를 들어 8비트로 접근할지, 16비트로 접근할지, 32비트로 접근할지), 데이터를 각각의 뱅크로부터 읽는 경우 그 시간이 길 경우에 WAIT 신호를 고려할 지 등을 결정한다.
- BANKCONn: BANKCONn($n = 0, 1, 2, 3, 4, 5$) 레지스터를 이용하여 n 에 해당하는 뱅크에 대한 메모리 접근 타이밍을 설정할 수 있다. 예를 들어 ROM이나 SRAM등을 접근할 때는 Address, Chip Selection, Data 등의 순서로 신호를 내 보낸다. 이 경우 각각의 신호간 적당한 시간 간격을 주어야 하며, 이러한 시간에 대한 설정을 이 레지스터를 이용하여 설정한다.

또한 BANKCONn($n = 6, 7$) 레지스터는 n 에 해당하는 뱅크에 ROM이나 SRAM이 있을 경우는 앞에 설명한 내용과 같은 역할을 하며, SDRAM이 있을 경우는 tRCD(RAS to CAS delay)나 컬럼 어드레스의 크기가 얼마지를 설정한다. SDRAM의 경우 데이터를 유지하기 위해 주기적으로 데이터를 관리해야 하는데, 이와 관련된 기능은 REFRESH 레

지스터를 이용하여 설정한다.

- BANKSIZE: BANKSIZE 레지스터는 SDRAM을 접근할 때 하나 이상의 데이터를 읽어오는 burst operation이냐, 뱅크 6과 7에 대한 메모리 맵 등을 설정하기 위해 사용한다.
- MRSRBn: MRSRBn(n = 6, 7) 레지스터는 burst length, CAS latency, burst type 등 SDRAM과 관련된 기능을 설정한다.

이상에서 메모리 컨트롤러와 관련한 13개 레지스터의 내용을 살펴보았다. 이 중 SDRAM과 관련한 구체적인 설명은 여기서 다루지 않기로 한다. 다음의 문서를 참고하기 바란다.

SDRAM Device Operations
SDRAM Timing Diagram
256Mb E-die SDRAM Specification
S3C2440A 32-BIT CMOS MICROCONTROLLER USER'S MANUAL Revision 1

memsetup.S 파일에서는 BWSCON 레지스터부터 시작해서 13개의 레지스터에 SMRDATA 번지에 있는 13개의 32비트 데이터를 써넣는 역할을 한다.

다음은 head.S 파일의 내용을 살펴보기로 하자.

```
head.S
.globl _start
_start:
stack_setup:
    ldr r0,=_start
    sub sp,r0,#4
    ldr pc,=kmain
```

여기서는 sp의 값을 (0x30100000-4)으로 설정한다. 그리고 kmain 루틴으로 쏜다. kmain 루틴은 main.c 파일에 정의되어 있다.

다음은 main.c 파일의 내용을 살펴보자.

```
main.c
#define rGPFDAT (*(volatile unsigned long *)0x56000054)
void kmain(void)
{
    int i;
    while(1) {
        rGPFDAT = rGPFDAT|0x000000f0;
        for(i=0;i<0x10000;i++);
        rGPFDAT = rGPFDAT&~0x000000f0;
        for(i=0;i<0x10000;i++);
    }
}
```

여기서는 led를 켜다가 키는 동작을 무한히 반복한다. 지금까지 cute-boot 부트로더에서 cuteOS로 뛰는 과정을 보이기 위해 cuteOS 부분을 최대한 간단히 작성했다.

이상에서 부트로더의 나머지 부분과 cuteOS의 시작 부분을 작성해 보았다. 다음 호에서는 cuteOS 부분을 본격적으로 작성해 보도록 한다. head.S 파일에 MMU와 cache 설정 부분을 작성해 보고, exception 벡터 테이블, software interrupt 처리 루틴, 하드웨어 인터럽트 처리 루틴 등을 살펴보기로 하자. ^{Real}Time

참고 자료

<http://sourceforge.net/projects/u-boot>
u-boot-1.1.2.tar.bz2
<http://emlinux.co.kr/index.php>
u-boot-1.0.0-emlinux.tgz

S3C2440A 32-BIT CMOS MICROCONTROLLER USER'S MANUAL Revision 1