ARM 리눅스 커널 포팅 (MBA2410)

본 문서는 ARM 리눅스 커널 부팅 과정과 아이지 시스템 MBA2410에 최신 리눅스 커널(2.6.15)을 포팅하는 과정을 기술하고 있다. 부트로더인 U-Boot에서 커널로 진입한 이후 아키텍처에 종속된 setup_arch()의 실행이 완료되기까지의 흐름을 짚어 보았다. S3C2410(ARM920T) 기반의 MBA2410은 SMDK2410와 하드웨어 구성이 대부분 동일하므로 리눅스 커널에서 지원하는 SMDK2410 소스 코드를 활용하면 어렵지 않게 포팅 작업을 수행할 수 있다.

호스트 PC에 설치되는 각종 프로그램의 설치 및 사용법은 다루지 않는다. 일반적으로 널리 사용되는 프로그램이므로 관련 매뉴얼이나 자료를 쉽게 찾을 수 있을 것이다. 부트로더는 커널 진입 과정을 설명하면서 일부 상세 내용을 언급하기는 하지만 MBA2410에 포팅하기 위한 과정을 기술하지는 않는다. 부트로더 설정은 별도의 문서를 참고하기 바란다.

단국대학교 연구방 14기 김영식
http://09room.net

장비 제공 : 단국대학교 전자계산공학 연구실
http://microlab.dankook.ac.kr

■ 변경사항

- 2006년 3월 17일 - 초안 작성

■ MBA2410 하드웨어 구성

Processor	Samsung S3C2410 (ARM920T)
Flash Memory	8M bit x 1
SDRAM	64MB (32MB x 2)
LCD	Color TFT LCD (320x240)
UART	3 channel (including IrDA)
USB	2 channel (Host & Slave)
Ethernet	1 channel (CS8900A)
	Touch panel Interface

	RTC IIC with KS24C080 ADC Interface SPI Interface IIS Interface (Sound CODEC audio I/O) PCMCIA Interface SD host (MMC) Interface Smart Media Card Key Matrix

■ 리눅스 커널 버전

포팅에 사용할 리눅스 커널은 포팅 작업 당시의 최신 안정화 버전 2.6.15.x에 프리패치가 적용된 2.6.16-rc3 버전을 사용했다. 2.6.15 버전을 사용해도 문제가 없으나 S3C24xx 관련 소스 코드에 패치된 내용이 있어 최근 내용을 반영하고자 가장 최신 버전을 선택했다.

- Linux Kernel 2.6 (ftp://kernel.org/pub/linux/kernel/v2.6/)
- Linux Kernel 2.6 prepatch (ftp://kernel.org/pub/linux/kernel/v2.6/testing/)
- Samsung S3C2410, SMDK2410 (http://www.samsung.com/Products/Semiconductor/MobileSolutions/MobileASSP/MobileComputing/S3C2410/S3C2410.htm)
- AIJI System MBA2410 (http://www.aijisystem.co.kr/korea/product/evboard/MBA-2410.htm)
- AIJI System SMDK2410 (http://www.aijisystem.co.kr/korea/product/evboard/SMDK2410.htm)
- S3C2410 LINUX KERNEL INSTALL GUIDE (http://www-core.kaist.ac.kr/~jplee/data/Kernel%20Install%20Guide.pdf)

■ 개발 환경

- Host PC #1 (Linux - tftp, nfs, samba, minicom)
- Host PC #2 (Windows - spider)
- ELDK(Embedded Linux Development Kit) 4.0
- Das U-Boot 1.1.4

포팅 작업은 주로 리눅스 호스트 PC에서 이뤄지고, 윈도 호스트 PC에서는 JTAG 디버깅(spider)과 플래시 라이팅(FlashUP) 작업을 수행한다. 리눅스 호스트에는 tftp, nfs, samba 데몬이 실행되고 있다. tftpd는 부트로더에서 커널 이미지를 다운로드할 때 필요하

고, nfsd는 NFS 루트 파일 시스템을 마운트하기 위해 사용된다. Samba는 리눅스 호스트에 존재하는 소스 파일이나 커널 이미지를 윈도 호스트에서 공유폴더로 접근할 때 유용하다. Minicom은 시리얼 통신용 프로그램으로 부트로더와 커널의 출력 결과를 LCD 화면 대신 호스트 PC에서 확인할 수 있게 도와준다. ELDK는 크로스 컴파일 툴체인과 루트 파일 시스템을 제공한다.

- Linux NFS (http://nfs.sourceforge.net)
- Samba (http://www.samba.org)
- Minicom (http://alioth.debian.org/projects/minicom)
- Embedded Linux Development Kit (http://www.denx.de/wiki/publish/DULG/DULG-tqm8xxl.html#Section_3)
- Das U-Boot - Universal Bootloader (http://sourceforge.net/projects/u-boot)

■ 부트로더, 커널 진입

리눅스 커널이 부팅되려면 부트로더가 반드시 필요하다. PC에서는 하드웨어 초기화를 BIOS가 처리하지만 임베디드 환경에는 BIOS가 없기 때문에 부트로더가 커널 진입은 물론 하드웨어 초기화까지 담당한다.

ARM 아키텍처에서 사용할 수 있는 다양한 부트로더(U-Boot, RedBoot, Blob, Angel 등) 가운데 Das U-Boot를 선택했다. U-Boot는 PPC용 부트로더에서 시작해서 현재는 ARM, MIPS, x86 등 다양한 아키텍처에서 사용할 수 있다. 특히 SMDK2410 보드를 지원하기 때문에 약간의 수정만으로도 MBA2410으로 포팅할 수 있다.

U-Boot가 실행되면 시리얼을 통해 명령을 주고 받을 수 있는 커맨드라인 환경을 만날 수 있다. 이제부터 부트로더에서 커널로 어떻게 넘어가느냐가 관건이다. 커널이 어디에 저장되어 있는지, 어느 위치로 재배치되는지, 어떤 정보를 어떻게 넘겨줘야 하는지 생각해봐야 한다.

```
U-Boot 1.1.4 (Feb 10 2006 - 22:33:03)

U-Boot code: 33F80000 -> 33F95ECC  BSS: -> 33F9A1E4
RAM Configuration:
Bank #0: 30000000 64 MB
Flash:  1 MB
In:     serial
Out:    serial
Err:    serial
MBA2410 #
```

포팅 단계에서 커널은 호스트 PC에서 계속적으로 재컴파일해야 하기 때문에 플래시 메모리에 저장하는 것은 매우 비효율적이다. 영구적으로 저장할 수는 없어도 네트워크를 통해 필요할 때마다 다운로드 받는 방법을 권장한다. 빠른 전송을 위해서 MBA2410과 호스트 PC를 이더넷 케이블로 연결하고 tftp로 커널 이미지를 다운로드하도록 한다.

잠깐 MBA2410의 메모리 구성을 짚고 넘어가자. MBA2410은 32MB SDRAM 2개가 하나의 뱅크를 이뤄 총 64MB 메모리를 사용할 수 있다. 각 메모리 IC는 nGCS6 핀과 연결되어 있다. nGCS6 핀은 6번 뱅크에 해당하므로 물리 주소 0x30000000 번지부터 시작되고 마지막 주소는 0x33FFFFFF이다. (S3C2410 User's Manual 5-2)

U-Boot의 tftp 전송 명령은 다음과 같다. 파일이 저장될 대상 주소를 지정해야 한다. MBA2410의 RAM 영역은 0x30000000~0x33FFFFFF이므로 이 영역에 해당하는 주소를 선택해서 넣어 준다. 여기에 몇가지 고려해야 할 사항이 있다.

```
MBA2410 # tftp [addr] [image]
```

u-boot/cpu/arm920t/start.S

U-Boot는 플래시 메모리에 저장되어 보드에 전원이 공급될 때마다 자동으로 실행된다. 이 때 (특별히 CONFIG_SKIP_RELOCATE_UBOOT 매크로가 정의되지 않은 이상) 읽고 쓸 수 있는 RAM 영역으로 재배치(relocation)가 이뤄진다. 플래시 메모리에 있던 U-Boot 실행 파일의 각 섹션이 쓰기가 자유로운 RAM에 자리를 잡는다는 의미이다. RAM으로 재배치되어야 stack이나 bss 섹션을 사용할 수 있기 때문이다. U-Boot의 재배치 시작 주소는 board/mba2410/config.mk에 정의되어 있다.

u-boot/board/mba2410/config.mk

```
TEXT_BASE = 0x33F80000
```

TEXT_BASE부터 text, data, rodata, bss 섹션이 나란하게 배치된다. 결과적으로 0x33F80000부터 나머지 공간(512K)은 커널 진입 전까지 사용해서는 안된다. 자세한 재배치 구조는 링커 스크립트(board/mba2410/u-boot.lds)나 MAP 파일(u-boot.map)을 참고하기 바란다.

u-boot/u-boot.map

```
0x00000000                . = 0x0
```

```
                0x00000000                . = ALIGN (0x4)

.text           0x33f80000     0x111b8  cpu/arm920t/start.o(.text)
.text           0x33f80000      0x400   cpu/arm920t/start.o
                0x33f80048              _bss_start
                0x33f8004c              _bss_end
                0x33f80044              _armboot_start
                0x33f80000              _start
*(.text)
.text           0x33f80400      0x64 board/mba2410/libmba2410.a(lowlevel_init.o)
                0x33f80404              lowlevel_init
                                        :
                                        :
```

arch/arm/boot/compressed/head.S

리눅스 커널 이미지는 크기를 줄이기 위해 압축된 형태로 저장한다. (vmlinux→zImage, bzImage) 압축 이미지에는 압축 해제 코드가 담겨 있어 압축 해제 코드가 실행되면 미리 지정된 커널 이미지 물리 주소(ZRELADDR=zreladdr-y=0x30008000)에 커널 이미지를 풀어 놓는다.

<table>
<tr><td rowspan="3">compressed kernel
(zImage)</td><td>압축 해제 코드 (head.o, misc.o)</td><td>text</td></tr>
<tr><td>압축된 커널 이미지 (piggy.gz)</td><td>piggydata</td></tr>
<tr><td>초기 데이터 (head.o, misc.o)</td><td>got, data</td></tr>
</table>

결국 tftp로 다운로드한 이미지는 ZRELADDR+[kernel size] 이후 영역이나 ZRELADDR 이전 영역에 위치해야 한다. 후자의 경우 메모리가 두 조각으로 나뉘므로 효율적인 메모리 사용을 위해 전자의 조건을 만족시켜야 한다. 커널 영역과 U-Boot 사이 임의의 주소 0x31000000에 커널 이미지를 다운로드하도록 했다. 사실 압축 이미지와 커널 이미지 영역이 겹쳐도 상관없다. 겹칠 경우 압축 이미지 뒤쪽에 커널 이미지를 풀고, 풀린 이미지를 다시 ZRELADDR으로 재배치하는 과정이 추가된다.

arch/arm/mach-s3c2410/Makefile.boot

```
zreladdr-y      := 0x30008000
```

SDRAM 32MB x 2 Bank 6	30008000-30407FFF	decompressed kernel
	: : :	compressed kernel
	33F80000-33FFFFFF	U-Boot

이제 커널로 진입해야 할 차례다. 부트로더에서 압축 이미지로 점프하면 ZRELADDR에 압축이 풀리고, 커널 이미지로 제어권을 넘겨주지만 이것으로 다 끝난 것이 아니다. 압축 이미지로 가기 전에 부트로더가 해야 할 준비 작업이 남아있다. 자세히 파고들면 FIQ, IRQ 인터럽트를 막아야 하고, D-Cache와 MMU는 꺼놓고, SVC모드에서 들어가야 하는 등 여러 조건이 요구되지만 이는 U-Boot가 알아서 하는 일이고, 크게 두 가지만 신경쓰면 된다.

첫째, 부트로더에서 커널에 필요한 정보를 tagged list로 만들어서 전달한다. 예를 들어 메모리 뱅크 정보, 커널 파라메터, 램디스크 위치, 프래임 퍼버 주소 등을 따로 커널 파라메터로 지정하거나 커널 소스를 수정하지 않고도 부트로더에서 설정된 값으로 직접 넘겨줄 수 있다. 메모리 뱅크 정보와 커널 파라메터를 tagged list에 추가하기 위해 mba2410.h에 다음 매크로를 정의한다.

u-boot/include/configs/mba2410.h

```
>   #define CONFIG_SETUP_MEMORY_TAGS            1
>   #define CONFIG_CMDLINE_TAG                  1
```

앞에서 정의된 태그 관련 매크로를 바탕으로 tagged list를 생성하는 코드는 armlinux.c에서 확인할 수 있다. tagged list에는 시작(ATAG_CORE)과 끝(ATAG_NONE)을 나타내는 태그가 존재하고 그 사이에 부트로더에서 지정한 태그가 위치한다.

u-boot/lib-arm/armlinux.c

```
void do_bootm_linux(...)
{
                :
        setup_start_tag();        /* ATAG_CORE */
                :
        setup_memory_tag();     /* ATAG_MEM, CONFIG_SETUP_MEMORY_TAGS */
        setup_commandline_tag(); /* ATAG_CMDLINE, CONFIG_CMDLINE_TAG */
                :
        setup_end_tag();          /* ATAG_NONE */
                :
```

```
}
```

hdr.tag	**ATAG_CORE**
hdr.size	ATAG_CORE 태그 크기 / 4
hdr.tag	**ATAG_MEM**
hdr.size	ATAG_MEM 태그 크기 / 4
u.mem.start	메모리 시작 주소
u.mem.size	메모리 크기
: 뱅크 개수 만큼 ATAG_MEM 태그 반복 :	
hdr.tag	**ATAG_CMDLINE**
hdr.size	ATAG_CMDLINE 태그 크기 / 4
u.cmdline...	bootargs 문자열
hdr.tag	**ATAG_NONE**
hdr.size	0

tagged list는 커널이 읽어올 수 있는 주소에 위치해야 한다. 보통 제로 페이지에 구성하고 (RAM의 처음 16KB 영역을 권장), tagged list의 위치는 bd->bi_boot_params에 주소값을 넣어줌으로서 설정할 수 있다.

u-boot/board/mba2410/mba2410.c

```
        gd->bd->bi_boot_params = 0x30000100;  /* board_init();
```

참고로 MBA2410의 메모리 뱅크 정보는 dram_init()에서 초기화된다. 커널 파라메터 (bootargs)는 u-boot/include/configs/mba2410.h에 CONFIG_BOOTARGS 매크로로 초기값이 정의되어 있고, bootargs 환경변수가 커널 파라메터 문자열로 복사된다.

```
int dram_init (void)
{
        DECLARE_GLOBAL_DATA_PTR;

        gd->bd->bi_dram[0].start = PHYS_SDRAM_1;
        gd->bd->bi_dram[0].size = PHYS_SDRAM_1_SIZE;

        return 0;
}
```

SDRAM 32MB x 2 Bank 6	30000000-300000FF	
	30000100-	**tagged list**
	30008000-30407FFF	decompressed kernel
	: :	
	:	compressed kernel
	:	
	33F80000-33FFFFFF	U-Boot

둘째, R0-R2 레지스터에 적절한 값을 넣어준다.

- R0 - 0
- R1 - 머신 타입 Type ID
- R2 - tagged list 시작 주소

R2 레지스터는 앞에서 설명한 tagged list의 시작 주소 gd->bd->bi_boot_params 를 넣는다. R1 레지스터에는 MBA2410의 고유한 ID 값을 지정하는데, 커널에서 R1 레지스터 값과 MBA2410 머신 ID를 비교하기 때문에 반드시 U-Boot에서 지정한 머신 ID와 커널에서 지정한 머신 ID가 서로 일치해야 한다. 49858은 임의로 지정한 값이다.

u-boot/include/asm-arm/mach-types.h

```
#define MACH_TYPE_MBA2410               49858
```

u-boot/board/mba2410/mba2410.c

```
gd->bd->bi_arch_number = MACH_TYPE_MBA2410; /* board_init() */
```

arch/arm/tools/mach-types

```
mba2410         MACH_MBA2410            MBA2410         49858
```

u-boot/lib-arm/armlinux.c

마지막으로 R0-R2 레지스터를 설정하고 커널로 진입하는 부분이다.

```
void do_bootm_linux(...)
{
        void (*theKernel)(int zero, int arch, uint params);
                                :
        theKernel = (void (*)(int, int, uint))ntohl(hdr->ih_ep);
                                :
        theKernel (0, bd->bi_arch_number, bd->bi_boot_params);
}
```

U-Boot에서는 zImage 파일을 그대로 올리는 것이 아니라 mkimage로 zImage에 헤더 정보를 붙인 이미지를 사용한다. 헤더 정보 중 hdr->ih_ep는 커널의 시작 주소(entry point)를 나타낸다. theKernel 함수 포인터가 ih_ep 번지를 가리키게 하고, theKernel()를 호출함으로서 제어권은 커널로 넘어간다.

theKernel()의 파라메터로 R0~R2까지 넣어야 할 값들이 지정되었다. AAPCS에 의해 함수 호출시 처음 파라메터 4개는 스택이 아닌 R0~R3 레지스터를 통해 전달된다. theKernel 호출 결과 R0에는 0, R1에는 bd->bi_arch_number, R2에는 bd->bi_boot_params이 차례로 들어간다.

- Das U-Boot (http://www.denx.de/wiki/view/DULG/UBoot)
- About TEXTADDR, ZTEXTADDR, PAGE_OFFSET etc... (http://lists.arm.linux.org.uk/pipermail/linux-arm-kernel/2001-July/004064.html)
- ARM Linux - Machine Type table (http://www.arm.linux.org.uk/developer/machines)
- Booting ARM Linux (Documentation/arm/Booting)
- Procedure Call Standard for the ARM Arcitecture (http://www.arm.com/miscPDFs/8031.pdf)
- S3C2410 User's Manual Rev 1.2 (http://www.samsung.com/Products/Semiconductor/SystemLSI/MobileSolutions/MobileASSP/MobileComputing/S3C2410X/um_s3c2410s_rev12_030428.pdf)

■ U-Boot 커널 부팅(bootm)의 실체

커널 진입에 대해 설명했지만 사실 U-Boot 커널 이미지 부팅 명령인 bootm에는 미묘한 문제가 숨어 있다.

bootm 명령으로 부팅 가능한 이미지를 만들려면 커널 이미지에 mkimage로 헤더 정보를 덧붙여야 한다. mkimage 사용법을 보면 헤더에 포함되는 정보를 한눈에 볼 수 있다. 눈여겨봐야 할 옵션으로 -a (load address)와 -e (entry point)가 있다.

```
Usage: mkimage -l image
          -l ==> list image header information
       mkimage [-x] -A arch -O os -T type -C comp -a addr -e ep -n name -d
data_file[:data_file...] image
          -A ==> set architecture to 'arch'
          -O ==> set operating system to 'os'
          -T ==> set image type to 'type'
          -C ==> set compression type 'comp'
          -a ==> set load address to 'addr' (hex)
          -e ==> set entry point to 'ep' (hex)
          -n ==> set image name to 'name'
          -d ==> use image data from 'datafile'
          -x ==> set XIP (execute in place)
```

보통 mkimage로 직접 헤더 정보를 붙이기보다 편의상 make에 uImage 타깃을 지정해서 커널 컴파일과 동시에 생성된 uImage를 사용한다. make uImage로 만들어진 uImage의 헤더 정보는 다음과 같다.

```
$ mkimage -l uImage
Image Name:   Linux-2.6.16-rc3
Created:      Mon Feb 27 02:54:09 2006
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    891552 Bytes = 870.66 kB = 0.85 MB
Load Address: 0x30008000
Entry Point:  0x30008000
```

Load Address와 Entry Point가 모두 ZRELADDR (0x30008000)을 나타내고 있다. 분명 앞에서 커널 진입시 Entry Point(hdr->ih_ep)로 점프하는 것을 확인했다. 하지만 이 순간 0x30008000 주소에는 아무것도 없지 않은가?

<table>
<tr><td rowspan="6">SDRAM
32MB x 2
Bank 6</td><td>30000000-300000FF</td><td></td></tr>
<tr><td>30000100-</td><td>tagged list</td></tr>
<tr><td>30008000-</td><td>???</td></tr>
<tr><td rowspan="3">:
:
:</td><td></td></tr>
<tr><td>compressed kernel</td></tr>
<tr><td></td></tr>
<tr><td>33F80000-33FFFFFF</td><td>U-Boot</td></tr>
</table>

u-boot/common/cmd_bootm.c

무슨 일인지 bootm 명령이 수행되는 do_bootm()부터 분석해보자. do_bootm()이 호출되면 Magic Number와 CRC 체크를 통해 이미지의 유효성을 따져 묻는다. 이어서 Load Address (0x30008000)와 bootm의 파라메터 - uImage가 위치한 주소, 0x31000000 - 를 비교해서 주소가 다를 경우 압축된 커널 이미지 전체를 Load Address로 복사하고 do_bootm_linux() 호출을 통해 Entry Point (0x30008000)로 점프한다.

<table>
<tr><td rowspan="9">SDRAM
32MB x 2
Bank 6</td><td>30000000-300000FF</td><td colspan="2"></td></tr>
<tr><td>30000100-</td><td colspan="2">tagged list</td></tr>
<tr><td>30008000-</td><td colspan="2">compressed kernel</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>31000000-3100003F</td><td rowspan="2">uImage</td><td>header (64bytes)</td></tr>
<tr><td>31000040-</td><td>compressed kernel</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>33F80000-33FFFFFF</td><td colspan="2">U-Boot</td></tr>
</table>

이후 Entry Point (ZRELADDR)에 있던 압축 해제 코드가 실행되면서 커널 이미지가 존재해야할 ZRELADDR에 압축을 풀지 못하고 압축 이미지 뒷부분에 풀어놓는다.

<table>
<tr><td rowspan="9">SDRAM
32MB x 2
Bank 6</td><td>30000000-300000FF</td><td colspan="2"></td></tr>
<tr><td>30000100-</td><td colspan="2">tagged list</td></tr>
<tr><td>30008000-</td><td colspan="2">compressed kernel</td></tr>
<tr><td></td><td colspan="2">decompressed kernel</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>31000000-3100003F</td><td rowspan="2">uImage</td><td>header (64bytes)</td></tr>
<tr><td>31000040-</td><td>compressed kernel</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>33F80000-33FFFFFF</td><td colspan="2">U-Boot</td></tr>
</table>

커널 이미지가 생성되었기 때문에 압축 이미지는 더 이상 필요 없다. 커널 이미지를 ZRELADDR로 재배치하고 정상적으로 ZRELADDR에서 커널 이미지를 실행한다. 커널로 정상적으로 부팅되긴 하지만 0x31000000 주소 선택 노력이 무색해지는 복잡한 과정을 거쳤다. 원하지 않은 이미지 데이터 복사, 재배치 과정이 추가된 것이다. 어떻게 하면 불필요한 과정을 제거하고 uImage의 압축 커널 직접 실행할 수 있을까?

<table>
<tr><td rowspan="9">SDRAM
32MB x 2
Bank 6</td><td>30000000-300000FF</td><td colspan="2"></td></tr>
<tr><td>30000100-</td><td colspan="2">tagged list</td></tr>
<tr><td>30008000-30407FFF</td><td colspan="2">decompressed kernel</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>31000000-3100003F</td><td rowspan="2">uImage</td><td>header (64bytes)</td></tr>
<tr><td>31000040-</td><td>compressed kernel</td></tr>
<tr><td></td><td colspan="2"></td></tr>
<tr><td>33F80000-33FFFFFF</td><td colspan="2">U-Boot</td></tr>
</table>

해답은 uImage의 Load Address와 Entry Point에 있다. Load Address와 uImage 위치를 비교해서 다를 경우 이미지 데이터를 복사한다면 Load Address를 uImage의 주소로 설정하면 된다. 또한 do_bootm_linux()에서 Entry Point로 분기한다면 Entry Point를 uImage 내의 압축 이미지 시작 주소를 가리키게 하면 된다.

```
$ mkimage -A arm -O linux -T kernel -C none -a 0x31000000 -e 0x31000040 -n
"Linux Kernel 2.6.x" -d zImage uImage
```

Load Address는 0x31000000, Entry Point는 uImage 헤더 크기가 64바이트이므로 Load Address에서 헤더만큼 건너뛴 0x31000040으로 설정한다. 이제부터 bootm 명령을 내리면 0x31000040의 압축 해제 코드가 실행되면서 ZRELADDR에 커널 이미지가 배치된다.

30000000-300000FF		
30000100-	tagged list	
30008000-	**???**	
31000000-3100003F	uImage	header (64bytes)
31000040-		compressed kernel
33F80000-33FFFFFF	U-Boot	

← Load Address
← Entry Point

SDRAM 32MB x 2 Bank 6	30000000-300000FF		
	30000100-	**tagged list**	
	30008000-	decompressed kernel	
	31000000-3100003F	uImage	header (64bytes)
	31000040-		compressed kernel
	33F80000-33FFFFFF	U-Boot	

make uImage를 그대로 사용하려면 arch/arm/boot/Makefile에서 다음 줄을 찾아 -a, -e 옵션을 수정하면 된다.

arch/arm/boot/Makefile

```
cmd_uimage = $(CONFIG_SHELL) $(MKIMAGE) -A arm -O linux -T kernel \
                -C none -a $(ZRELADDR) -e $(ZRELADDR) \
```

```
-n 'Linux-$(KERNELRELEASE)' -d $< $@
```

- Das U-Boot: The Universal Boot Loader (http://linuxdevices.com/articles/AT5085702347.html)

■ 커널 진입 이후 (head.S)

zImage의 압축 해제 코드가 실행되고 난 직후 상황이다. 커널 이미지가 ZRELADDR에 위치하고, R0에는 0, R1에는 머신 ID, R2에는 tagged list 물리 주소가 각각 저장되어 있다. 지금부터 커널의 실행 경로를 밟아보도록 하자.

arch/arm/boot/head.S

arch/arm/boot/head.S의 stext가 커널의 실제 출발점이다. SVC 모드로 전환한 후 __lookup_processor_type으로 분기한다.

[__lookup_processor_type]

CP15의 레지스터 0에서 아키텍처 버전을 읽어 proc_info_list의 cpu_val과 동일한지 비교한다. (S3C2410 User's Manual A2-7) proc_info_list의 구조는 다음과 같으며 초기값은 proc-arm920.S에서 살펴볼 수 있다. ARM 아키텍처 종류별로 서로 다른 proc-arm*.S이 존재한다. S3C2410은 ASM920T이므로 proc-arm920.S이 선택적으로 컴파일되어 초기값이 결정된다.

include/asm-arm/procinfo.h

```
29 struct proc_info_list {
30          unsigned int            cpu_val;
31          unsigned int            cpu_mask;
32          unsigned long           __cpu_mmu_flags;        /* used by head.S */
33          unsigned long           __cpu_flush;            /* used by head.S */
34          const char              *arch_name;
35          const char              *elf_name;
36          unsigned int            elf_hwcap;
37          const char              *cpu_name;
38          struct processor        *proc;
39          struct cpu_tlb_fns      *tlb;
```

```
40          struct cpu_user_fns     *user;
41          struct cpu_cache_fns    *cache;
42  };
```

arch/arm/boot/head.S

```
455         .section ".proc.info.init", #alloc, #execinstr
456
457         .type    __arm920_proc_info,#object
458  __arm920_proc_info:
459         .long    0x41009200                 @ cpu_val
460         .long    0xff00fff0                 @ cpu_mask
461         .long    PMD_TYPE_SECT | \
462                  PMD_SECT_BUFFERABLE | \
463                  PMD_SECT_CACHEABLE | \
464                  PMD_BIT4 | \
465                  PMD_SECT_AP_WRITE | \
466                  PMD_SECT_AP_READ           @ __cpu_mmu_flags
467         b        __arm920_setup             @ __cpu_flush
468         .long    cpu_arch_name              @ *arch_name, "armv4t"
469         .long    cpu_elf_name               @ *elf_name, "v4"
470         .long    HWCAP_SWP | HWCAP_HALF | HWCAP_THUMB  @ elf_hwcap
471         .long    cpu_arm920_name            @ *cpu_name, "ARM920T"
                        :
```

proc_info_list 데이터가 여러 개일 경우 같은 버전이 나올 때까지 반복 비교한다.

[__lookup_machine_type]

R1 레지스터에 저장된 머신 ID와 __arch_info_begin에 위치한 machine_desc 구조체의 nr 값과 비교한다. nr의 초기값은 MACH_TYPE_MBA2410으로 커널 컴파일 이후 arch/arm/tools/mach-types 파일을 바탕으로 자동 생성되는 include/asm-asm/mach-types.h에 매크로가 정의되어 있다.

include/asm-arm/mach/arch.h

```
53  #define MACHINE_START(_type,_name)                      \
54  static const struct machine_desc __mach_desc_##_type    \
55   __attribute_used__                                      \
56   __attribute__((__section__(".arch.info.init"))) = {    \
```

```
57          .nr             = MACH_TYPE_##_type,            \
58          .name           = _name,
59
60  #define MACHINE_END                                     \
61  };
```

```
19  struct machine_desc {
20          /*
21           * Note! The first five elements are used
22           * by assembler code in head-armv.S
23           */
24          unsigned int            nr;             /* architecture number  */
25          unsigned int __deprecated phys_ram;     /* start of physical ram */
26          unsigned int            phys_io;        /* start of physical io */
27          unsigned int            io_pg_offst;    /* byte offset for io
28                                                   * page tabe entry      */
                                  :
```

arm/mach-s3c2410/mach-mba2410.c

```
115 MACHINE_START(MBA2410, "MBA2410")
```

[__create_page_tables] arch/arm/boot/head.S

현재는 MMU가 꺼진 상태이지만 이후 MMU를 켜고 나서도 커널 이미지의 메모리 영역을 정상적으로 참조할 수 있도록 페이지 테이블을 생성한다. 먼저 1차 페이지 테이블(16K)을 0으로 초기화한다. 현재 실행 위치(pc를 포함하는 1MB 영역)의 물리 주소에 대한 페이지 테이블 디스크립터를 page proc_info_list의 __cpu_mmu_flags로 설정한다. 초기값은 2차 페이지 테이블을 거치지 않고 매핑되는 섹션 타입이다. (S3C2410 User's Manual A3-8) 커널의 가상 주소(TEXTADDR / 1MB, 0xC0000000부터 4MB 영역, 커널 최대 크기)에 대한 섹션 디스크립터를 설정한다. 마지막으로 PAGE_OFFSET의 1MB를 PHYS_OFFSET으로 매핑하지만 이미 TEXTADDR과 동일한 위치의 매핑 값이므로 현재의 메모리 맵에서는 의미가 없다.

	offset	**L1 descriptor**									
swapper_pg_dir 0xC0004000	0x000 * 4										
	:	:									
	0x300 * 4	**0x300**		11		0000	1	C	B	1	0

	:	:								
	0xC00 * 4	**0x300**		11		0000	1	C	B	1	0
	0xC01 * 4	**0x301**		11		0000	1	C	B	1	0
	0xC02 * 4	**0x302**		11		0000	1	C	B	1	0
	0xC03 * 4	**0x303**		11		0000	1	C	B	1	0
	:	:								
	0xFFF * 4									

[__arm920_setup] arch/arm/mm/proc-arm920.S

proc_info_list의 __cpu_flush에 저장되어 있던 분기 명령이 실행되면서 __arm920_setup으로 이동한다. I-Cache와 D-Cache를 무효화(invalidate)하고 쓰기 버퍼의 내용도 비운다. 더불어 I/D-TLB도 무효화한다.

CP15 제어 레지스터를 읽어서 MMU, Cache 관련 비트를 설정한다. (R0 레지스터)

비트		의 미
CR_V	1	벡터 테이블 주소 (high memory - 0xFFFF0000)
CR_I	1	I-Cache enable
CR_Z	0	예약됨
CR_F	0	예약됨
CR_R	0	ROM 보호 비트 (S3C2410 User's Manual A3-19 참고)
CR_S	1	시스템 보호 비트 (S3C2410 User's Manual A3-19 참고)
CR_D	1	예약됨
CR_P	1	예약됨
CR_W	0	예약됨
CR_C	1	D-Cache enable
CR_A	0	데이터 주소 정렬 disable
CR_M	1	MMU enable

[__enable_mmu] arch/arm/boot/head.S

커널 환경설정에 따른 제어 레지스터 값을 재설정한다.

커널 컴파일 옵션	비트	의 미
CONFIG_ALIGNMENT_TRAP	CR_A	데이터 주소 정렬 여부 검사
CONFIG_CPU_DCACHE_DISABLE	CR_C	I-Cache disable
CONFIG_CPU_BPREDICT_DISABLE	CR_Z	예약됨
CONFIG_CPU_ICACHE_DISABLE	CR_I	I-Cache disable

도메인에 대한 접근 정보를 설정한다. 리눅스 커널에서는 3개의 도메인을 사용한다. Manager와 Client의 차이점은 섹션, 페이지 디스크립터의 액세스 권한 비트(AP) 검사 여부에 있다.

No	Domain	Type
0	DOMAIN_KERNEL	DOMAIN_MANAGER
0	DOMAIN_TABLE	DOMAIN_MANAGER
1	DOMAIN_USER	DOMAIN_MANAGER
2	DOMAIN_IO	DOMAIN_CLIENT

CP15 TTB(Translation Table Base) 레지스터에 __create_page_tables에서 설정한 1차 페이지 테이블 주소를 저장한다.

[__turn_mmu_on]

__arm920_setup과 __enable_mmu에서 설정한 제어 레지스터의 값(R0 레지스터)을 실제로 CP15 제어 레지스터에 저장한다. 이제부터 MMU가 작동하고 가상 메모리 공간을 사용하게 된다. 모든 메모리 주소 참조는 주소 변환 과정을 거친다.

[__mmap_switched]

__data_loc과 __data_start의 위치가 다를 경우 __data_loc에 위치한 데이터 세그먼트를 __data_start로 복사한다. ROM에 있는 data 섹션(__data_loc)의 내용을 RAM(__data_start)으로 복사해야 하는 XIP 커널에서 발생한다. bss 세그먼트를 0으로 초기화한다.

CP15 제어 레지스터에서 읽어온 아키텍처 버전(R9, [__lookup_processor_type])과 부트로더에서 넘긴 머신 ID(R1, [__lookup_machine_type])를 저장한다. 이 값들은 arch/arm/kernel/setup.c의 processor_id와 __machine_arch_type 변수를 통해 참조할 수 있다.

__enable_mmu에서 처리했던 CP15 제어 레지스터의 값(R0 레지스터)을 arch/arm/kernel/entry-armv.S의 cr_alignment에 저장하고, A bit(Alignment Fault Checking)을 클리어 한 값을 cr_no_alignment에 저장한다.

```
:__lookup_processor_type
        @ 아키텍처 ID 검사
:__lookup_machine_type
        @ 머신 ID 검사
:__create_page_tables
        @ 1차 페이지 테이블 초기화
:__arm920_setup
        @ 캐시, TLB 무효화
:__enable_mmu
        @ 도메인 액세스 제어 레지스터 설정
        @ 페이지 테이블 위치 지정 (TTB 레지스터)
:__turn_mmu_on
        @ MMU 동작
:__mmap_switched
        @ bss 초기화
                :
        b start_kernel
```

start_kernel()로 분기한다. 이제부터 다소 딱딱한 ARM 명령 셋에서 벗어나 반가운 C언어 코드를 만나볼 수 있다.

- S3C2410 User's Manual Rev 1.2 (http://www.samsung.com/Products/Semiconductor/SystemLSI/MobileSolutions/MobileASSP/MobileComputing/S3C2410X/um_s3c2410s_rev12_030428.pdf)
- ARM System Developer's Guide - Sloss, Symes, Wright (사이텍미디어)

■ 커널 진입 이후 (start_kernel)

kernel/main.c

```
441 asmlinkage void __init start_kernel(void)
442 {
                        :
449         lock_kernel();
```

```
450         page_address_init();
451         printk(KERN_NOTICE);
452         printk(linux_banner);
453         setup_arch(&command_line);
454         setup_per_cpu_areas();
                        :
```

리눅스 커널 부팅의 모든 과정은 start_kernel()에서 이뤄진다고 해도 과언이 아니다. 커널 버전 정보를 출력하면서부터 init 프로세스(프로세스 1)가 실행되기까지 많은 함수들이 start_kernel()로부터 호출된다.

가장 먼저 커널 초기화 코드에 대한 락(lock)을 획득한다. (#449) start_kernel()에는 스핀락을 이용한 Big Kernel Lock(BLK), 세마포어를 사용한 Big Kernel Semaphore(BKS) 구현이 존재한다. BKS은 락을 소유한 코드가 선점당해도 세마포어를 실제로 반납하지 않아 불필요하게 락을 해제하고, 얻는 과정을 제거한 것이 특징이다.

커널 환경설정에 따라 page_address_init()에서는 high memory 관리에 필요한 free 연결 리스트와 해시 테이블을 초기화한다. (#450) 현재의 메모리 구성에서는 모든 RAM 영역이 가상 메모리에 매핑되므로 high memory는 신경 쓰지 않아도 된다.

커널 버전 정보를 printk로 출력하고 setup_arch() 함수를 호출한다. (#454)

[setup_arch()] arch/arm/kernel/setup.c

```
729 void __init setup_arch(char **cmdline_p)
730 {
                        :
735         setup_processor();
736         mdesc = setup_machine(machine_arch_type);
737         machine_name = mdesc->name;
                        :
```

setup_processor()에서는 lookup_processor_type()를 호출한다. head.S에서 호출되었던 __lookup_processor_type과 동일한 루틴이다. 아키텍처 ID를 다시 한 번 비교를 하고 일치하는 proc_arch_list의 주소를 리턴한다. (#735)

[setup_processor()]

```
284 static void __init setup_processor(void)
285 {
286         struct proc_info_list *list;
                        :
293         list = lookup_processor_type();
                        :
323         cpu_proc_init();
324 }
```

cpu_proc_init()는 매크로로 정의되어 있다. (#323) arm/arm/mm/proc_arm920.S의 cpu_arm920_proc_init로 분기하지만 아무런 일도 수행하지 않고 바로 리턴된다.

[setup_arch()]

setup_arch()로 돌아와서 setup_machine()를 호출한다. setup_machine()에서도 setup_processor()와 마찬가지로 head.S의 __lookup_machine_type 호출해서 머신 ID를 체크한 다음 일치하는 mach_info의 주소를 리턴한다.

```
729 void __init setup_arch(char **cmdline_p)
730 {
                                        :
742         if (mdesc->boot_params)
743                 tags = phys_to_virt(mdesc->boot_params);
                                        :
749         if (tags->hdr.tag != ATAG_CORE)
750                 convert_to_tag_list(tags);
751         if (tags->hdr.tag != ATAG_CORE)
752                 tags = (struct tag *)&init_tags;
753
754         if (mdesc->fixup)
755                 mdesc->fixup(mdesc, tags, &from, &meminfo);
756
757         if (tags->hdr.tag == ATAG_CORE) {
758                 if (meminfo.nr_banks != 0)
759                         squash_mem_tags(tags);
760                 parse_tags(tags);
761         }
                                        :
```

tagged list를 처리하는 부분이다.. 물리 주소 0x30000100 값이 들어 있는 mdesc->boot_params를 가상 주소로 변환한다. (#743) (mdesc의 초기화 코드는 arch/arm/mach-s3c2410/mach-mba2410.c에서 찾을 수 있다.) 만약 tags에 ATAG_CORE 태그가 존재

하지 않는다면 예전 파라메터 형식으로 간주하고 tagged list로 변환하는 과정을 거친다. (#750) 현재 메모리 뱅크가 설정되어 있다면 (meminfo.nr_banks != 0) tagged list의 ATAG_MEM를 ATAG_NONE으로 바꿔서 미리 설정된 메모리 구성을 사용한다. (#759) parse_tags()에서 tag를 하나하나 읽어서 처리한다. (#760)

[parse_tag()]

```
674 static int __init parse_tag(const struct tag *tag)
675 {
676         extern struct tagtable __tagtable_begin, __tagtable_end;
677         struct tagtable *t;
678
679         for (t = &__tagtable_begin; t < &__tagtable_end; t++)
680                 if (tag->hdr.tag == t->tag) {
681                         t->parse(tag);
682                         break;
683                 }
684
685         return t < &__tagtable_end;
686 }
```

parse_tags()에서는 개별 태그에 대해 parse_tag()를 호출한다. 위의 코드를 보면 커널에서 처리할 수 있는 태그 정보는 __tagtable_begin에 위치하는 것을 알 수 있다.

arch/arm/kernel/vmlinux.lds.S

```
 38                 __tagtable_begin = .;
 39                         *(.taglist.init)
 40                 __tagtable_end = .;
```

__tagtable_begin과 __tagtable_end 사이에는.taglist.init 섹션이 자리 잡고 있다. taglist.init 섹션을 초기화하기 위해 다음 매크로가 setup.c에서 사용되고 있다.

include/asm-arm/setup.h

```
174 #define __tag __attribute_used__ __attribute__((__section__(".taglist.init")))
175 #define __tagtable(tag, fn) \
176 static struct tagtable __tagtable_##fn __tag = { tag, fn }
```

gcc 확장 문법이 다소 복잡하게 느껴질 수 있지만 의미는 간단하다. tagtable 구조체 변

수를 초기화하면서 data 섹션이 아닌 taglist.init 섹션에 넣으라는 의미이다.

[parse_tag_core()] arch/arm/kernel/setup.c

태그 파싱 함수와 그에 대응되는 __tagtable 매크로다. 아래 방식처럼 태그 파싱 함수를 만들고 __tagtable 매크로를 사용하면 나만의 고유한 태그를 만들어 부트로더와 정보를 교환할 수 있다.

```
561 static int __init parse_tag_core(const struct tag *tag)
562 {
563         if (tag->hdr.size > 2) {
564                 if ((tag->u.core.flags & 1) == 0)
565                         root_mountflags &= ~MS_RDONLY;
566                 ROOT_DEV = old_decode_dev(tag->u.core.rootdev);
567         }
568         return 0;
569 }
570
571 __tagtable(ATAG_CORE, parse_tag_core);
```

[setup_arch()]

```
729 void __init setup_arch(char **cmdline_p)
730 {
                                        :
763         init_mm.start_code = (unsigned long) &_text;
764         init_mm.end_code   = (unsigned long) &_etext;
765         init_mm.end_data   = (unsigned long) &_edata;
766         init_mm.brk        = (unsigned long) &_end;
                                        :
```

init_mm는 프로세스 0 - idle 프로세스의 메모리 정보를 담고 있는 mm_struct 구조체 전역 변수다. 프로세스 0에 대한 커널 코드 영역 시작 주소(_text), 마지막 주소(_etext, _etext는 text와 rodata를 포함), 데이터 영역의 마지막 주소(_edata), 힙 영역의 마지막 주소(_end, bss 섹션 포함)를 저장한다.

아래는 init_mm 변수 선언 및 초기화 코드이다. pgd(page global directory)를 앞에서 설정했던 swapper_pg_dir 값(0xC0004000)으로 초기화한 것을 확인할 수 있다.

arch/arm/kernel/init_task.c

```
19 struct mm_struct init_mm = INIT_MM(init_mm);
```

include/linux/init_task.h

```
44 #define INIT_MM(name) \
45 {                                                       \
46         .mm_rb          = RB_ROOT,                          \
47         .pgd            = swapper_pg_dir,                   \
48         .mm_users       = ATOMIC_INIT(2),                   \
49         .mm_count       = ATOMIC_INIT(1),                   \
50         .mmap_sem       = __RWSEM_INITIALIZER(name.mmap_sem),   \
51         .page_table_lock =  SPIN_LOCK_UNLOCKED,                 \
52         .mmlist         = LIST_HEAD_INIT(name.mmlist),          \
53         .cpu_vm_mask    = CPU_MASK_ALL,                     \
54 }
```

[setup_arch()] arch/arm/kernel/setup.c

```
729 void __init setup_arch(char **cmdline_p)
730 {
                                        :
768         memcpy(saved_command_line, from, COMMAND_LINE_SIZE);
769         saved_command_line[COMMAND_LINE_SIZE-1] = '\0';
770         parse_cmdline(cmdline_p, from);
771         paging_init(&meminfo, mdesc);
                                        :
```

parse_cmdline()는 커널 파라메터 문자열을 읽어서 개별 파라메터에 대한 해당 파싱함수가 등록되어 있으면 바로 처리하고, 그렇지 않으면 cmdline_p에 내용을 복사한다. (#770) 파싱 함수를 처리 과정이 tagged list의 경우와 유사하므로 자세한 내용은 생략한다. setup.c에는 mem, initrd 파라메터를 처리할 수 있는 파싱함수가 위치하고 있다.

paging_init()는 부트 메모리 할당자(bootmem allocator)를 초기화 하고 페이지 테이블을 설정하는 역할을 담당한다. (#771) 부트 메모리 할당자는 페이지 할당자(page allocator)를 사용하기 전까지 커널 부팅 과정에서만 임시적으로 사용되는 메모리 할당자이다.

[paging_init()] arch/arm/mm/init.c

```
508 void __init paging_init(struct meminfo *mi, struct machine_desc *mdesc)
```

```
509 {
510         void *zero_page;
511
512         build_mem_type_table();
513         bootmem_init(mi);
                                        :
```

boot_mem_type_table()은 캐시 정책, 아키텍처 버전에 따른 메모리 타입(mem_types[])별 페이지 테이블 디스크립터의 관련 플래그를 설정한다. (#512)

이어 호출되는 bootmem_init()는 커널 초기화 과정에서 사용되는 부트 메모리 할당자를 초기화한다. (#513)

[bootmem_init()] arch/arm/mm/init.c

초기화 과정에서 head.S에서 생성했던 페이지 테이블에서 0~PAGE_OFFSET(0xC0000000) 구간에 대한 디스크립터를 클리어하고, 첫 번째 메모리 뱅크를 제외한 VMALLOC_END까지의 커널 공간에 대한 매핑을 삭제한다. S3C2410에서는 VMALLOC_END가 0xE0000000으로 정의되어 있다. (include/asm-arm/arch-s3c2410/vmalloc.h)

	offset	L1 descriptor									
swapper_pg_dir 0xC0004000	0x000 * 4										
	:	:									
	0xC00 * 4	**0x300**		11		0000	1	C	B	1	0
	0xC01 * 4	**0x301**		11		0000	1	C	B	1	0
	0xC02 * 4	**0x302**		11		0000	1	C	B	1	0
	0xC03 * 4	**0x303**		11		0000	1	C	B	1	0
	:	:									
	0xFFF * 4										

```
352 static void __init bootmem_init(struct meminfo *mi)
                                        :
394         for_each_node(node) {
395                 unsigned long end_pfn;
396
397                 end_pfn = bootmem_init_node(node, initrd_node, mi);
                                        :
402                 if (end_pfn > memend_pfn)
```

```
403                         memend_pfn = end_pfn;
404         }
```

메모리 노드마다 반복하며 bootmem_init_node()를 호출한다. (#397) 노드는 물리적으로 연속된 메모리를 의미한다. 하나의 노드에는 여러 개의 연속된 뱅크가 속할 수 있다. MBA2410은 하나의 노드와 하나의 뱅크만 가지고 있으므로 노드와 뱅크에 대한 반복문은 무시해도 상관없다.

[bootmem_init_node()] arch/arm/mm/init.c

```
238 static unsigned long __init
239 bootmem_init_node(int node, int initrd_node, struct meminfo *mi)
240 {
                                        :
253         for_each_nodebank(i, mi, node) {
                                        :
265                 map.pfn = __phys_to_pfn(mi->bank[i].start);
266                 map.virtual = __phys_to_virt(mi->bank[i].start);
267                 map.length = mi->bank[i].size;
268                 map.type = MT_MEMORY;
269
270                 create_mapping(&map);
271         }
                                        :
```

각 뱅크별로 메모리에 대한 매핑이 create_mapping()에서 이뤄진다. (#270) create_mapping()을 살펴보기 전에 ARM 리눅스의 페이지 테이블 처리 방식을 이해해야 한다. 아마 bootmem_init()에서 페이지 테이블을 클리어 하는 과정을 유심히 따라가다 보면 조금 이상한 점을 발견하게 될 것이다. PGDIR_SHIFT가 20이 아닌 21이고, pmd_clear()에서 두 개의 pmd 디스크립터를 한 번에 클리어하고 있다.

리눅스 커널은 3차 페이지 테이블(pgd, pmd, pte)을 사용한다. 반면 ARM920T MMU는 하드웨어적으로 2차 페이지 테이블을 이용하기 때문에 ARM 리눅스 커널은 pmd(page middle directory)를 사용하지 않는다. 실제 pgd(page global directory)와 pte(page table entry)만 존재하고, pmd는 단순히 pgd를 가리킨다.

리눅스 VM 시스템에서는 페이지별로 dirty, young과 같은 ARM MMU의 페이지 테이블 디스크립터에는 존재하지 않는 플래그를 필요로 한다. 이러한 차이점을 보완하기 위해 하나의 pgd당 2개의 페이지 테이블을 두고 있다. 하나는 하드웨어와 호환되는 H/W 페이지 테이블, 다른 하나는 리눅스 관련 플래그가 담긴 리눅스 페이지 테이블이다.

Linux view	**hardware view**		**2nd level** (256 × 4 × 4 = 4k)
Linux entry	h/w entry 0	→	h/w pt 0 (256 enties)
	h/w entry 1	→	h/w pt 1 (256 enties)
:	:		Linux pt 0 (256 enties)
2048 entries	4096 entries		Linux pt 1 (256 enties)

1차 페이지 테이블(pgd)의 엔트리 개수는 4096개에서 2048개로 줄어드는 대신 하나의 엔트리가 4바이트에서 8바이트로 늘어났다. 실제로 메모리에 저장된 1차 페이지 테이블 구조가 바뀐 건 아니다. 커널에서 페이지 테이블을 논리적으로 바라보는 관점만 바뀌었을 뿐이다. 반면 2차 페이지 테이블에는 2개의 H/W 페이지 테이블이 나란하게 있고, 뒤이어 리눅스 페이지 테이블 2개가 위치하고 있다. 이렇게 하면 2차 페이지 테이블을 하나의 페이지 4K 공간에 꽉 차게 사용할 수 있다.

만약 1차 페이지 테이블에 2차 페이지 테이블을 연결해야 한다면 pt 0와 pt 1을 모두 설정해야 한다. 커널 입장에서는 2차 페이지 테이블을 512개의 엔트리로 보고 있지만 MMU는 종전대로 1차 페이지 테이블은 4바이트씩 4096개, 2차 페이지 테이블 엔트리는 256개로 알고 접근하기 때문이다. pmd_clear()에서 연속된 두 개의 pmd 디스크립터를 클리어하는 것도 바로 이런 이유에서였다.

[create_mapping()] arch/arm/mm/mm-armv.c

```
484 void __init create_mapping(struct map_desc *md)
485 {
                                        :
505         domain     = mem_types[md->type].domain;
506         prot_pte   = __pgprot(mem_types[md->type].prot_pte);
507         prot_l1    = mem_types[md->type].prot_l1 | PMD_DOMAIN(domain);
508         prot_sect = mem_types[md->type].prot_sect | PMD_DOMAIN(domain);
                                        :
535         virt    = md->virtual;
536         off    -= virt;
537         length = md->length;
                                        :
547        while ((virt & 0xfffff || (virt + off) & 0xfffff) && length >= PAGE_SIZE) {
548                 alloc_init_page(virt, virt + off, prot_l1, prot_pte);
549
550                 virt   += PAGE_SIZE;
551                 length -= PAGE_SIZE;
552         }
                                        :
590         while (length >= (PGDIR_SIZE / 2)) {
```

```
591                 alloc_init_section(virt, virt + off, prot_sect);
592
593                 virt   += (PGDIR_SIZE / 2);
594                 length -= (PGDIR_SIZE / 2);
595         }
596
597         while (length >= PAGE_SIZE) {
598                 alloc_init_page(virt, virt + off, prot_l1, prot_pte);
599
600                 virt   += PAGE_SIZE;
601                 length -= PAGE_SIZE;
602         }
603 }
```

create_mapping()의 파라메터 md에는 MBA2410 첫 번째 뱅크에 대한 RAM 정보가 담겨 있다. (#484)

매핑하려는 가상 주소와 물리 주소가 1MB 단위로 정렬되어 있지 않은 경우 alloc_init_page()로 2차 페이지 테이블까지 설정하고(#548), 정렬되어 있으면 alloc_init_section()에서 1MB(PGDIR_SIZE/2)씩 RAM 전 영역에 대한 섹션 디스크립터를 설정한다. (#5491)

	offset	L1 descriptor									
swapper_pg_dir 0xC0004000	0x000 * 4										
	:	:									
	0xC00 * 4	**0x300**		01		0000	1	?	?	1	0
	0xC01 * 4	**0x301**		01		0000	1	?	?	1	0
	:	:									
	0xC38 * 4	**0x338**		01		0000	1	?	?	1	0
	0xC39 * 4	**0x339**		01		0000	1	?	?	1	0
	:	:									
	0xFFF * 4										

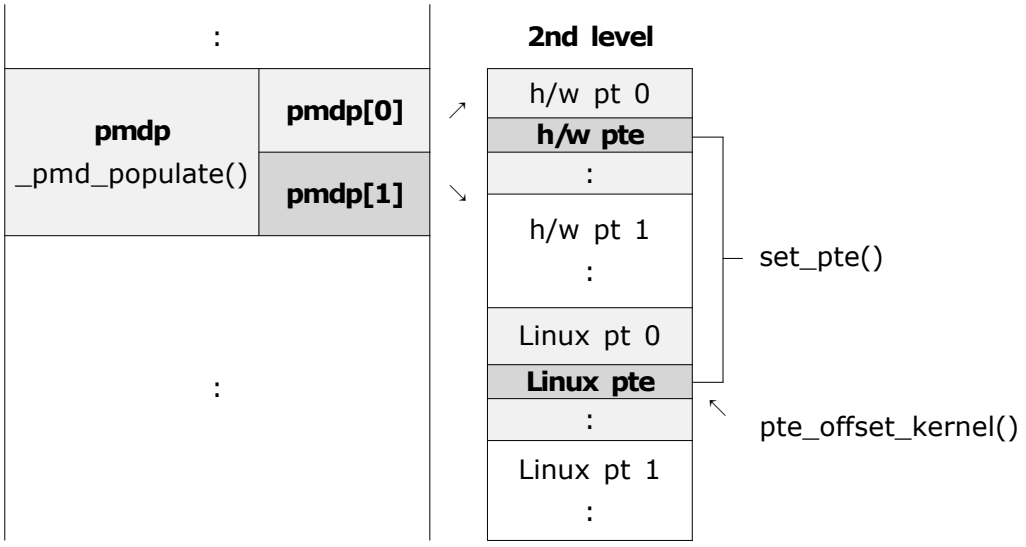
1MB 단위로 매핑하고 남은 메모리는 4KB(PAGE_SIZE) 페이지씩 나눠서 2차 페이지 테이블을 설정한다. (#598)

MBA2410의 64MB RAM는 주소와 크기가 모두 1MB 단위로 나눠떨어지기 때문에 2차 페이지 테이블 설정은 하지 않지만 ARM 리눅스의 페이지 테이블 설정 예를 알아보기 위해 잠시 그 내용을 살펴보고 넘어가자.

[alloc_init_page()] arch/arm/mm/mm-armv.c

```
277 static inline void
278 alloc_init_page(unsigned long virt, unsigned long phys, unsigned int prot_l1,
pgprot_t prot)
279 {
280         pmd_t *pmdp = pmd_off_k(virt);
281         pte_t *ptep;
282
283         if (pmd_none(*pmdp)) {
284                 ptep = alloc_bootmem_low_pages(2 * PTRS_PER_PTE *
285                                                sizeof(pte_t));
286
287                 __pmd_populate(pmdp, __pa(ptep) | prot_l1);
288         }
289         ptep = pte_offset_kernel(pmdp, virt);
290
291         set_pte(ptep, pfn_pte(phys >> PAGE_SHIFT, prot));
292 }
```

주어진 가상 주소에 대한 pmd가 설정되어 있지 않으면 부트 메모리 할당자(bootmem allocator)로부터 2차 페이지 테이블의 내용이 담긴 4KB만큼의 메모리를 할당 받는다. (#284) 1차 페이지 테이블이 디스크립터가 새로 할당 받은 ptep를 가리키도록 한다. (#287) 커널은 2개의 pgd를 하나로 보기 때문에 __pmd_populate()에서는 pmdp[0]과 pmdp[1], 두 개의 pmd 디스크립터를 설정한다. pte_offset_kernel()은 리눅스 2차 페이지 테이블 엔트리 주소를 리턴한다. (#289) 마지막으로 리눅스 페이지 테이블과 h/w 페이지 테이블 엔트리를 설정한다. (#291) set_pte()는 cpu_set_pte()로 정의되어 있고, cpu_set_pte()는 다시 cpu_arm920_set_pte()로 정의한다. (arch/arm/mm/proc-arm920.S)



사실 이 시점에서는 아직 부트 메모리 할당자가 초기화되지 않았다. alloc_bootmem_low_page()가 호출되어서는 안 된다.

[bootmem_init_node()] arch/arm/mm/init.c

```
238 static unsigned long __init
239 bootmem_init_node(int node, int initrd_node, struct meminfo *mi)
                                        :
282         boot_pages = bootmem_bootmap_pages(end_pfn - start_pfn);
283         boot_pfn = find_bootmap_pfn(node, mi, boot_pages);
                                        :
289         node_set_online(node);
290         pgdat = NODE_DATA(node);
291         init_bootmem_node(pgdat, boot_pfn, start_pfn, end_pfn);
292
293         for_each_nodebank(i, mi, node)
294                 free_bootmem_node(pgdat, mi->bank[i].start, mi->bank[i].size);
                                        :
299         reserve_bootmem_node(pgdat, boot_pfn << PAGE_SHIFT,
300                                 boot_pages << PAGE_SHIFT);
                                        :
317         if (node == 0)
318                 reserve_node_zero(pgdat);
                                        :
```

부트 메모리 할당자는 노드 내의 각 페이지별로 1비트씩 차지하는 비트맵 데이터를 가지고 있다. (pg_dat->bdata->node_bootmem_map[]) 비트맵 데이터가 1이면 해당 페이지는 사용 중인 상태를 나타낸다.

bootmem_bootmap_pages()는 현재의 노드를 나타낼 수 있는 비트맵 데이터 크기를 페이지 단위로 리턴하고(#282), find_bootmap_pfn()에서는 노드 내의 비트맵 데이터가 위치할 페이지 번호를 결정한다. (#283) 커널 이미지가 존재하는 노드에서는 커널 이미지 이후 페이지(_end <)에 비트맵 데이터를 위치시킨다. 커널이 없다면 노드의 첫 페이지로 결정된다. 현 노드를 온라인 상태로 표시한다. (#289) NODE_DATA()는 현 시스템은 하나의 노드만 사용(물리 메모리가 연속)하는 UMA 시스템이므로 contig_page_data의 주소를 나타낸다. (#290)

[struct pglist_data] include/linux/mmzone.h

```
287 typedef struct pglist_data {
```

```
288         struct zone node_zones[MAX_NR_ZONES];
289         struct zonelist node_zonelists[GFP_ZONETYPES];
290         int nr_zones;
291 #ifdef CONFIG_FLAT_NODE_MEM_MAP
292         struct page *node_mem_map;
293 #endif
294         struct bootmem_data *bdata;
                :
305         unsigned long node_start_pfn;
306           unsigned long node_present_pages; /* total number of physical pages
*/
307         unsigned long node_spanned_pages; /* total size of physical page
308                                              range, including holes */
309         int node_id;
310         struct pglist_data *pgdat_next;
311         wait_queue_head_t kswapd_wait;
312         struct task_struct *kswapd;
313         int kswapd_max_order;
314 } pg_data_t;
```

[struct pglist_data] include/linux/mmzone.h

```
2132 #ifndef CONFIG_NEED_MULTIPLE_NODES
2133 static bootmem_data_t contig_bootmem_data;
2134 struct pglist_data contig_page_data = { .bdata = &contig_bootmem_data };
2135
2136 EXPORT_SYMBOL(contig_page_data);
2137 #endif
```

<table>
<tr><td>0x30000000</td><td></td></tr>
<tr><td>0x30000100</td><td>tagged list</td></tr>
<tr><td>0x30004000</td><td>1차 페이지 테이블 (16K)
(pgd_t) swapper_pg_dir[PTRS_PER_PGD]</td></tr>
<tr><td>0x30008000
~ _end</td><td>커널 이미지</td></tr>
<tr><td>n개의 비트
(n = 페이지 수)</td><td>bootmem 할당자 비트맵
(void *) contig_bootmem_data->bdata->node_bootmem_map
1 1 1 1 1 1 1 1 … 0 0 0 0 0 0 0 0 0 0</td></tr>
<tr><td>n = 총 페이지 수</td><td>페이지 디스크립터 배열</td></tr>
</table>

	(struct page *) contig_bootmem_data->node_mem_map				
	page #1	page #2	page #3	…	page #n
:	:				

*** node_bootmem_map과 node_mem_map의 실 주소와 크기는 페이지 단위 정렬**

contig_page_data와 pgdat의 타입인 pglist_data 구조체는 노드에 대한 정보를 담는다. NUMA 시스템이 아니라면 단 하나의 노드만 존재하므로 소스 코드에서 보이는 pgdat는 항상 contig_page_data를 참조한다고 생각하자.

init_bootmem_node()에서 부트 메모리 할당자를 초기화 한다. (#291) 초기화 과정에서 비트맵 데이터를 모두 1로 지정한다. 모든 페이지가 사용 중이라는 의미이다. 뱅크마다 반복하며 free_bootmem_node()을 호출한다. (#294) 뱅크 내 페이지에 해당되는 비트맵 데이터가 0으로 클리어 된다. init_bootmem_node()에서 예약한 페이지 중 실제 뱅크에 속하는 페이지에 대해서만 다시 사용할 수 있게 해제한 것이다.

이 상태에서 현재 사용 중인 페이지에 대한 비트맵을 1로 설정한다. 커널 이미지, 비트맵 데이터, 페이지 테이블이 위치한 페이지가 그 대상이다. (#299) 이것으로 비트맵 데이터에 대한 설정이 완료되었다.

부트 메모리 할당자(bootmem allocator)는 커널 부팅 과정에서만 사용되고, 커널 부팅 이후에는 페이지 할당자(page allocator, zone allocator)로 그 역할을 넘겨준다. 지금부터 나머지 코드는 페이지 할당자와 관계된 과정이다. 그렇다고 곧 페이지 할당자가 사용되는 것은 아니고 단지 준비 단계일 뿐이다.

페이지 할당자는 메모리 영역을 3개 zone으로 나눠 관리한다. DMA 전송이 가능한 DMA zone, 커널 메모리 공간에 직접 매핑할 수 없는 HIGHMEM zone, 커널 공간에 매핑되어 있으면서 DMA 전송이 불가능한 나머지 메모리 영역은 normal zone에 속한다.

[bootmem_init_node()] arch/arm/mm/init.c

```
238 static unsigned long __init
239 bootmem_init_node(int node, int initrd_node, struct meminfo *mi)
                                        :
331         zone_size[0] = end_pfn - start_pfn;
                                        :
337         zhole_size[0] = zone_size[0];
338         for_each_nodebank(i, mi, node)
339                 zhole_size[0] -= mi->bank[i].size >> PAGE_SHIFT;
                                        :
```

```
345         arch_adjust_zones(node, zone_size, zhole_size);
346
347         free_area_init_node(node, pgdat, zone_size, start_pfn, zhole_size);
348
349         return end_pfn;
350 }
```

zone과 zhole 크기를 초기화한다. (#331~339) zhole은 zone에 속하지 못한 메모리 영역이다. 이를 테면 페이지 단위로 할당하고 남은 뱅크와 뱅크 사이의 경계 영역이 될 수 있을 것이다. arch_adjust_zones()는 아키텍처 특성에 다른 zone 크기 정보를 재설정하는 일을 담당한다. (#345) S3C2410에서는 아무 일도 하지 않는다. 모든 메모리 크기를 DMA zone(zone_size[0])에 설정하고 free_area_init_node()을 호출한다. (#347)

[free_area_init_node()] arch/arm/mm/page_alloc.c

```
2119 void __init free_area_init_node(int nid, struct pglist_data *pgdat,
2120                 unsigned long *zones_size, unsigned long node_start_pfn,
2121                 unsigned long *zholes_size)
2122 {
2123         pgdat->node_id = nid;
2124         pgdat->node_start_pfn = node_start_pfn;
2125         calculate_zone_totalpages(pgdat, zones_size, zholes_size);
2126
2127         alloc_node_mem_map(pgdat);
2128
2129         free_area_init_core(pgdat, zones_size, zholes_size);
2130 }
```

calculate_zone_totalpages()에서는 노드의 총 메모리 페이지를 계산하고, alloc_node_mem_map()에서 노드 내 각각의 페이지에 대한 정보를 담고 있을 page 구조체 배열을 부트 메모리 할당자로부터 할당받는다. (#2125) 첫 번째 노드일 경우 전역 변수 mem_map은 노드 0의 node_mem_map을 가리킨다. 부트 메모리 할당자에서 개별 페이지의 정보로서 단순히 비트맵에 저장되어 있는 0과 1이 전부였지만 페이지 할당자는 페이지별로 다양한 정보를 페이지 디스크립터(struct page)에 담아서 관리한다.

[free_area_init_core()]

```
2039 static void __init free_area_init_core(struct pglist_data *pgdat,
2040                 unsigned long *zones_size, unsigned long *zholes_size)
2041 {
```

```
                                        :
2051        for (j = 0; j < MAX_NR_ZONES; j++) {
                                        :
2074                zone_pcp_init(zone);
                                        :
2085                zonetable_add(zone, nid, j, zone_start_pfn, size);
2086                init_currently_empty_zone(zone, zone_start_pfn, size);
                                        :
```

free_area_init_core()에서 pgdat 내의 zone 대한 정보를 하나씩 채워 넣는다. CPU별 pageset을 설정한다. (#2074) zone_table[]에 zone을 추가한다. (#2085) init_currently_empty_zone()에서는 페이지 디스크립터의 내용을 초기화하면서 페이지를 예약 상태(PG_reserved)로 설정하고 zone 디스크립터의 free_area를 초기화 한다. (#2086) 모든 페이지가 예약된 상태이므로 현재의 free_area는 빈 상태다.

[paging_init()] arch/arm/mm/init.c

페이지 디스크립터 초기화 이후 리턴의 리턴을 거쳐 콜스택을 따라 올라가 보면 paging_init()에 이르게 된다. bootmem_init()에 이어서 devicemaps_init() 차례다. (#514)

```
508 void __init paging_init(struct meminfo *mi, struct machine_desc *mdesc)
509 {
                                        :
514         devicemaps_init(mdesc);
515
516         top_pmd = pmd_off_k(0xffff0000);
                                        :
```

[devicemaps_init()]

부트로더에서 커널로 진입할 당시를 떠올려보면 IRQ, FIQ 인터럽트 발생을 막아 놓은 상태에서 0번 주소에는 부트로더에서 설정한 인터럽트 벡터 테이블이 존재한다. 커널 코드가 실행되면서 head.S의 __turn_mmu_on에서 벡터 테이블 주소를 0xFFFF0000로 옮긴 이후 벡터 테이블에 대해서 전혀 신경쓰지 않았다.

```
426 static void __init devicemaps_init(struct machine_desc *mdesc)
427 {
                                        :
435         vectors = alloc_bootmem_low_pages(PAGE_SIZE);
436         BUG_ON(!vectors);
```

```
437
438         for (addr = VMALLOC_END; addr; addr += PGDIR_SIZE)
439                 pmd_clear(pmd_off_k(addr));
                                        :
476         map.pfn = __phys_to_pfn(virt_to_phys(vectors));
477         map.virtual = 0xffff0000;
478         map.length = PAGE_SIZE;
479         map.type = MT_HIGH_VECTORS;
480         create_mapping(&map);
481
482         if (!vectors_high()) {
483                 map.virtual = 0;
484                 map.type = MT_LOW_VECTORS;
485                 create_mapping(&map);
486         }
                                        :
491         if (mdesc->map_io)
492                 mdesc->map_io();
                                        :
500         local_flush_tlb_all();
501         flush_cache_all();
502 }
```

벡터 테이블 설정을 위한 1개의 페이지를 할당받는다. (#435) 가상 주소 VMALLOC_END(0xE0000000)부터 마지막 주소까지 pmd를 클리어 한다. (#439) 할당 받은 페이지를 MT_HIGH_VECTORS 타입으로 가상 주소 0xFFFF0000에 매핑한다. (#480) 2차 페이지 테이블이 생성된다.

	offset	L1 descriptor									
swapper_pg_dir 0xC0004000	0x000 * 4										
	:	:									
	0xC00 * 4	**0x300**		01		0000	1	?	?	1	0
	0xC01 * 4	**0x301**		01		0000	1	?	?	1	0
	:	:									
	0xC38 * 4	**0x338**		01		0000	1	?	?	1	0
	0xC39 * 4	**0x339**		01		0000	1	?	?	1	0
	:	:									
	0xFFE * 4	**0x?????**				0001	1	?	?	0	1
	0xFFF * 4	**0x????? + 1**				0001	1	?	?	0	1

장치별 메모리를 매핑하는 mdesc->map_io()가 호출된다. (#492) 시스템마다 장치 구성이나 메모리 매핑이 다르기 때문에 커널 포팅을 하면서 눈 여겨봐야 할 부분이다. 포팅 과정에서도 다루겠지만 mdesc->map_io 함수 포인터는 mba2410_map_io()를 가리키고 mba2410_map_io()에서는 (RAM을 제외하고) MBA2410의 각 장치에 대한 메모리 매핑이 이뤄진다. (arch/arm/mach-s3c2410/mach-mba2410.c)

[paging_init()]

```
508 void __init paging_init(struct meminfo *mi, struct machine_desc *mdesc)
509 {
                                        :
521         zero_page = alloc_bootmem_low_pages(PAGE_SIZE);
522         memzero(zero_page, PAGE_SIZE);
523         empty_zero_page = virt_to_page(zero_page);
524         flush_dcache_page(empty_zero_page);
525 }
```

제로 페이지를 생성한다. 하나의 페이지를 할당받아(#521) 0으로 초기화하고(#522) 캐시를 플러시한다. (#524)

[setup_arch()] arch/arm/kernel/setup.c

```
729 void __init setup_arch(char **cmdline_p)
730 {
                                        :
772         request_standard_resources(&meminfo, mdesc);
773
774         cpu_init();
                                        :
779         init_arch_irq = mdesc->init_irq;
780         system_timer = mdesc->timer;
781         init_machine = mdesc->init_machine;
                                        :
790 }
```

setup_arch()의 마지막 부분이다. request_standard_resource()에서는 시스템 메모리(iomem_resource)와 IO 포트(ioport_resource)에 대한 리소스 트리를 생성한다. (#772) cpu_init()에서는 CPU와 캐시 정보를 화면에 출력하고 각 CPU 모드별 스택을 초기화한다. (#774)

아키텍처 관련 포인터를 지정한다. (#779~781) 포팅 단계의 'LCD 설정'에서 LCD 초기

화 루틴을 init_machine으로 지정하는 것을 볼 수 있다. init_machine()는 커널 부팅이 거의 완료되고 init 프로세스를 생성하기 전 단계에서 호출된다. init_machine() 뿐 아니라 __initcall_start와 __initcall_end 사이에 위치한 함수가 모두 호출된다. (start_kernel() → rest_init() → do_basic_setup() → do_initcalls())

arch/arm/kernel/vmlinux.lds.S

```
27        .init : {                    /* Init code and data          */
                                        :
48                __initcall_start = .;
49                        *(.initcall1.init)
50                        *(.initcall2.init)
51                        *(.initcall3.init)
52                        *(.initcall4.init)
53                        *(.initcall5.init)
54                        *(.initcall6.init)
55                        *(.initcall7.init)
56                __initcall_end = .;
```

함수를 .initcall*.init에 넣기 위해 *_initcall 매크로가 사용된다.

[customize_machine()] arch/arm/kernel/setup.c

```
720 static int __init customize_machine(void)
721 {
722         /* customizes platform devices, or adds new ones */
723         if (init_machine)
724                 init_machine();
725         return 0;
726 }
727 arch_initcall(customize_machine);
```

이것으로 목표했던 setup_arch()까지 분석을 끝마쳤다. 사실 따져보면 start_kernel()에서 호출되는 많은 함수 가운데 setup_arch()는 고작 세 번째 함수이다. setup_arch()에서 하는 일이 많기는 하지만 부팅이 완료되기까지 앞으로 분석해야 할 내용이 산재해 있다. 나머지 부분은 여러분들의 몫으로 돌리고, 이제 본격적인 포팅 작업에 들어가 보고자 한다.

- The Big Kernel Lock lives on (http://lwn.net/Articles/86859/)
- The Big Kernel Semaphore? (http://lwn.net/Articles/102253/)
- Linux: Making The BKL Preemptable (http://kerneltrap.org/node/3843)

- Feature: High Memory In The Linux Kernel (http://kerneltrap.org/node/2450)
- Variable Attribute (http://gcc.gnu.org/onlinedocs/gcc-4.0.2/gcc/Variable-Attributes.html)
- RFC: Major re-jig of the ARM page table handling (http://lists.arm.linux.org.uk/pipermail/linux-arm-kernel/2002-March/008089.html)
- Outline of the Linux Memory Management System (http://home.earthlink.net/~jknapka/linux-mm/vmoutline.html)
- Memory Management in Linux (http://puggy.symonds.net/~abhi/files/mm/)
- Physical Memory Management in Linux (http://hzliu.myweb.hinet.net/linux-kernel/Physical%20Memory%20Management%20in%20Linux.pdf)
- The Linux Kernel Primer - Claudia Salzberg Rodriguez 외 2인 (Prentice Hall PTR)

■ 커널 컴파일 준비

커널 소스를 컴파일 하기 위한 준비 과정이다. 커널 트리에서 MBA2410과 하드웨어 구성의 유사한 SMDK2410 관련 매크로와 소스 파일에 대한 복사본은 만드는 방식으로 이뤄진다.

Makefile

Makefile을 열고 ARCH와 CROSS_COMPILE 환경변수를 찾아서 다음과 같이 수정한다.

```
175 ARCH = arm
176 CROSS_COMPILE = arm-linux-
```

컴파일시 make에 옵션으로 지정해도 무방하다.

```
$ make ARCH="arm" CROSS_COMPILE="arm-linux-"
```

arch/arm/mach-s3c2410/mba2410.c

```
linux $ cd arch/arm/mach-s3c2410
mach-s3c2410 $ cp smdk2410.c mba2410.c
```

smdk2410.c 파일의 복사본을 생성한 후 편집기로 mba2410.c 파일을 열어서 'smdk' 문자열을 'mba'로 치환한다. 새로운 소스 파일이 컴파일 될 수 있도록 Makefile 파일에 관련 내용을 수정한다.

arch/arm/mach-s3c2410/Makefile

```
44 obj-$(CONFIG_ARCH_SMDK2410)     += mach-smdk2410.o
>  obj-$(CONFIG_MACH_MBA2410)      += mach-mba2410.o
```

커널에는 아키텍처와 아키텍처 구현, 즉 머신을 나타내는 매크로로 각각 CONFIG_ARCH_xxx, CONFIG_MACH_xxx를 사용한다. SMDK2410라면 CONFIG_MACH_SMDK2410을 사용하는 것이 올바르나 93년 이전에는 아키텍처와 머신 구분 없이 CONFIG_ARCH_xxx만 사용했기 때문에 다소 혼란스럽지만 CONFIG_ARCH_S3C2410, CONFIG_ARCH_SMDK2410으로 나타내고 있다. 현재는 아키텍처와 머신을 구분할 것을 권장하고 있다. MBA2410의 아키텍처는 CONFIG_ARCH_S3C2410, 머신은 CONFIG_MACH_MBA2410으로 지정한다.

arch/arm/mach-s3c2410/Kconfig

커널 환경 설정(make menuconfig)에서 CONFIG_MACH_MBA2410을 선택할 수 있게 Kconfig 파일을 수정한다. Kconfig 내에서는 CONFIG_ 접두어를 생략한다.

```
46 config ARCH_SMDK2410
47         bool "SMDK2410/A9M2410"
48         select CPU_S3C2410
49         help
50           Say Y here if you are using the SMDK2410 or the derived ...
51           <http://www.fsforth.de>
52
>  config MACH_MBA2410
>          bool "MBA2410/A9M2410"
>          select CPU_S3C2410
>          help
>            Say Y here if you are using the MBA2410 or the derived ...
```

arch/arm/tools/mach-types

부트로더에서 R2 레지스터에 머신 ID를 넣어서 커널로 넘어가면 커널에서는 R2의 머신 ID와 mach-types에서 정의한 머신 ID가 올바른지 체크한다. mach-types에서 추가해야 할 머신 ID는 임의로 정하되 U-Boot에서 설정한 값과 같아야 한다.

```
>mba2410            MACH_MBA2410            MBA2410            49858
```

위와 같이 추가하면 커널 컴파일 과정에서 /include/asm-arm/mach-types.h 파일이 생성된다. mach-types.h에는 머신 ID와 머신 ID를 검사하는 조건 구문에 대한 매크로가 담겨 있다.

.config

커널 컴파일에 앞서 커널 환경설정(Linux Kernel Configuration)을 통해 .config 파일에 컴파일 되기 원하는 항목을 넣어야 한다. 다양한 커널 환경설정 방법 가운데 콘솔 상에서 메뉴 선택이 가능한 make menuconfig를 주로 사용한다.

```
linux $ make menuconfig
```

ARCH_S3C2410과 MACH_MBA2410을 찾아서 선택한다. 매크로 이름은 알고 있는데, 정확한 위치를 모른다면 / 를 눌러서 매크로 이름으로 검색할 수 있다.

```
System Type > ARM system type > Samsung S3C2410 (ARCH_S3C2410)
System Type > S3C24XX Implementations > MBA2410 (MACH_MBA2410)
```

arch/arm/boot/uImage

```
linux $ make uImage
```

커널 환경설정에서 설정 내용을 저장하면 .config 파일이 생성된다. 마지막으로 커널 컴파일만 남아있다. 현 상태에서 make를 실행하면 되지만 U-Boot에서 커널로 부팅하기 위해서는 커널 이미지에 헤더 정보를 덧붙인 이미지를 사용해야 한다. 헤더 정보는 mkimage로 넣을 수 있다. make에 uImage 타겟을 지정하면 커널 컴파일과 U-Boot 이미지 생성(uImage)이 한 번에 처리된다.

- Kernel Compilation (http://www.arm.linux.org.uk/docs/kerncomp.php)
- kbuild (Documentation/kbuild)

■ 테스트 (Starting Kernel...)

커널 이미지가 제대로 동작하는지 확인해보자.

```
$ cp ~/mba2410/linux/arch/arm/boot/uImage /tftproot/
$ minicom
                                        :
MBA2410 # tftp 31000000 uImage
TFTP from server 192.168.1.105; our IP address is 192.168.1.200
Filename 'uImage'.
Load address: 0x31000000
Loading: ##################################################
         ##################################################
         ##################################################
done
Bytes transferred = 935228 (e453c hex)
MBA2410 # bootm
## Booting image at 31000000 ...
   Image Name:   Linux-2.6.16-rc3
   Created:      2006-02-24   9:00:54 UTC
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    = 913.2 kB
   Lss: 30008000
   Entry Point:  30008000
   Verifying Checksum ... OK
OK

Starting kernel ...
Uncompressing Linux.....................................................................
```

커널 이미지 압축이 풀리고 나서 화면에 반응이 없다. 커널 내부에서는 열심히 printk()로 메시지를 출력하지만 이 상태에서는 시리얼 콘솔로 결과를 받아 볼 수 없다. 현재 진행 상황을 알아볼 수 없을까? arch/arm/kernel/debug.S에 문자열을 시리얼로 출력해주는 printascii 프로시저가 디버깅용으로 존재한다. printk() 소스를 열어서 printascii 프로시저로 내용을 출력하도록 바꿔주면 된다.

[printk()] linux/printk.c

```
514 asmlinkage int printk(const char *fmt, ...)
515 {
516         va_list args;
517         int r;
518
519         va_start(args, fmt);
520         r = vprintk(fmt, args);
521         va_end(args);
```

```
522
523         return r;
524 }
```

printk()는 가변인자만 처리할 뿐 실제 출력은 vprintk() 함수가 담당한다. 다시 vprintk() 를 따라가 보자. 최종 출력 문자열은 vscnprint() 함수에서 완성된다.

[vprintk()]

```
547         /* Emit the output into the temporary buffer */
548         printed_len = vscnprintf(printk_buf, sizeof(printk_buf), fmt, args);
>           printascii(printk_buf);
```

vsnprint() 함수 호출 다음 줄에 printascii 프로시저 호출 문장을 넣었다. printascii 프로시저는 내부에서 R0~R3 레지스터를 사용한다. C에서는 문제가 되지 않지만 어셈블리어 소스에서는 유의해야 할 사항이다.

```
                                        :
<5>Linux version 2.6.16-rc3 (root@pc05) (gcc version 4.0.0 (DENX ELDK 4.0
4.0.06CPU: ARM920Tid(wb) [41129200] revision 0 (ARMv4T)
Machine: MBA2410
Memory policy: ECC disabled, Data cache writeback
<7>On node 0 totalpages: 16384
<7>  DMA zone: 16384 pages, LIFO batch:3
<7>  DMA32 zone: 0 pages, LIFO batch:0
<7>  Normal zone: 0 pages, LIFO batch:0
<3>CPU S3C2400 support not enabled
<0>Kernel panic - not syncing: Unsupported S3C24XX CPU
```

다시 테스트한 결과, 드디어 부팅 메시지가 나타났다. 로그 레벨에 관계없이 모든 메시지를 볼 수 있다.

메시지를 찬찬히 뜯어보면 머신(MBA2410)이 올바르게 나오고, 메모리는 DMA zone에 16386 페이지가 할당되었다. ARM 리눅스에서는 1페이지가 4K이므로 64MB를 올바르게 인식했다. 메모리 구성에 대한 설정은 않고 부트로더에서 tagged list로 넘겨받은 정보를 사용하고 있다.

문제는 마지막 줄인데, 아무래도 CPU를 인식하지 못하는 것으로 추정된다.

커널 부팅 과정을 따라가다 보면 물리 메모리를 가상 메모리로 맵핑하는 과정이 존재한

다. 즉, 가상 메모리에 대한 페이지 테이블 설정하는 과정이다. 메모리 매핑뿐만 아니라 특수 레지스터 영역의 매핑이 이뤄진다. mach-mba2410.c의 mba2410_map_io()에서 MBA2410의 IO 매핑 정보가 담긴 mba2410_io_desc (현 상태에선 아무런 정보도 없음)를 파라메터로 넘겨주면서 s3c24xx_init_io() 함수를 호출한다.

[s3c24xx_init_io()] arch/arm/mach-s3c2410/cpu.c

```
160 void __init s3c24xx_init_io(struct map_desc *mach_desc, int size)
161 {
162         unsigned long idcode = 0x0;
163
164         /* initialise the io descriptors we need for initialisation */
165         iotable_init(s3c_iodesc, ARRAY_SIZE(s3c_iodesc));
166
167 #ifndef CONFIG_CPU_S3C2400
168         idcode = __raw_readl(S3C2410_GSTATUS1);
169 #endif
170
171         cpu = s3c_lookup_cpu(idcode);
172
173         if (cpu == NULL) {              /* 커널 패닉 발생! */
174                 printk(KERN_ERR "Unknown CPU type 0x%08lx\n", idcode);
175                 panic("Unknown S3C24XX CPU");
176         }
                                        :
185         (cpu->map_io)(mach_desc, size);
186 }
```

s3c24xx_init_io() 함수 구현을 들여다보면 또 다른 매핑 정보인 s3c_iodesc를 가지고 iotable_init()를 호출한다. 실제 페이지 테이블 설정이 이뤄지는 곳이다.

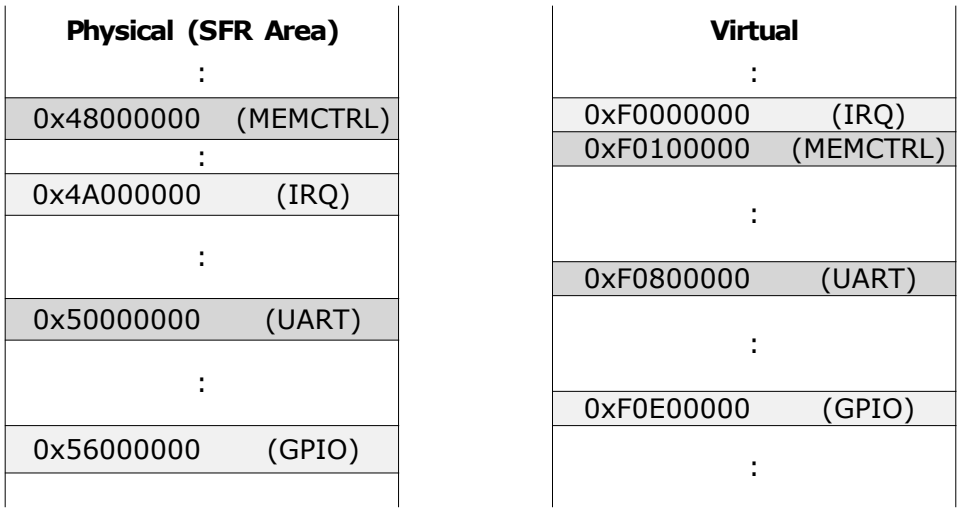
```
115 static struct map_desc s3c_iodesc[] __initdata = {
116         IODESC_ENT(GPIO),
117         IODESC_ENT(IRQ),
118         IODESC_ENT(MEMCTRL),
119         IODESC_ENT(UART)
120 };
```

s3c_iodesc 초기화 데이터 부분의 매크로를 제거하면 다음과 같이 볼 수 있다.

```
115 static struct map_desc s3c_iodesc[] __initdata = {
```

```
116         0xF0E00000, __phys_to_pfn(0x56000000), 0x100000, 0, /* GPIO */
117         0xF0000000, __phys_to_pfn(0x4A000000), 0x100000, 0, /* IRQ */
118         0xF0100000, __phys_to_pfn(0x48000000), 0x100000, 0, /* MEMCTRL */
119         0xF0800000, __phys_to_pfn(0x50000000), 0x100000, 0, /* UART */
120 };
```

각 항목은 가상 주소, 물리 주소의 페이지 번호, 영역 크기, 메모리 유형의 의미를 지니고 있다.



위의 그림과 같이 메모리 매핑을 실시(#165)한 후 곧바로 GSTATUS1 레지스터의 데이터를 읽어서 idcode에 저장한다. (#168) s3c_lookup_cpu()에서는 idcode로 S3C24XX의 모델명을 구분한다. 문제는 여기서 발생하는데 GSTATUS1(S3C2410 User's Manual 9-28)에서 읽는 값이 항상 0이다.

디버거로 GSTATUS1의 물리 주소(0x560000B0)의 값을 읽어보면 0x32410000이 들어있는 것을 확인할 수 있다. 반면 가상 주소(0xF0E00000)에서 읽은 값은 0이라는 것은 가상 메모리 처리에 문제가 있다고 의심해볼 수 있다. 위의 그림을 다시 살펴보면 GSTATUS1에 해당하는 메모리 영역은 iotable_init()에서 페이지 테이블 설정이 있었고 곧바로 메모리를 참조한다. 바로 전에 설정한 페이지 테이블이 적용되지 않고 TLB에 있던 주소 변환 값을 사용하기 때문에 올바른 물리 주소를 참조할 수 없었던 것이다. 커널 버그라고 의심되는 부분이다. iotable_init()를 호출한 다음에는 반드시 TLB와 Cache를 무효화시켜야 한다.

```
>#include <asm/tlb.h>
>#include <asm/cache.h>
                                    :
164         /* initialise the io descriptors we need for initialisation */
```

```
165         iotable_init(s3c_iodesc, ARRAY_SIZE(s3c_iodesc));
>           local_flush_tlb_all();
>           flush_cache_all();
```

- Cache and TLB Flushing Under Linux (Documentation/cachetlb.txt)
- S3C2410 User's Manual Rev 1.2 (http://www.samsung.com/Products/Semiconductor/SystemLSI/MobileSolutions/MobileASSP/MobileComputing/S3C2410X/um_s3c2410s_rev12_030428.pdf)

■ 이더넷 드라이버

NFS 파일 시스템을 사용하려면 이더넷 드라이버를 설정해야 한다. MBA2410에서는 이더넷 컨트롤러로 Cirrus Logic CS8900A를 사용한다. drivers/net 디렉토리에서 파일을 찾아보면 cs89x0.c 파일이 있긴 하지만 커널 환경설정 어디에도 CS89x0 선택 항목이 나타나지 않는다. drivers/net/Kconfig를 살펴보면 CS89x0이 CONFIG_NET_PCI와 CONFIG_ISA에 의존하는 것을 알 수 있다. CONFIG_MACH_MBA2410 항목이 선택되면 자동으로 CONFIG_NET_PCI와 CONFIG_ISA가 선택되도록 수정한다.

arch/arm/mach-s3c2410/Kconfig

```
config MACH_MBA2410
        bool "MBA2410/A9M2410"
        select CPU_S3C2410
>       select NET_PCI
>       select ISA
        help
          Say Y here if you are using the MBA2410 or the derived module ...
```

```
Device Drivers > Network device support > Ethernet (10 or 100 Mbit) > CS89x0 support (CS89x0)
```

커널 환경설정에서 CS89x0를 선택한 후 테스트 해보자.

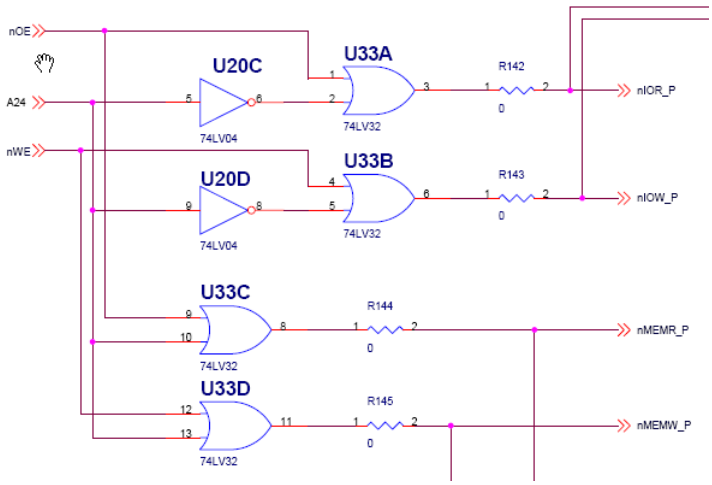
```
                                        :
cs89x0:cs89x0_probe(0x0)
<1>Unable to handle kernel paging request at virtual address f200030a
```

```
<1>pgd = c0004000
<1>[f200030a] *pgd=00000000
Internal error: Oops: 5 [#1]
CPU: 0
PC is at readword+0x1c/0x2c
LR is at cs89x0_probe1+0xe8/0x878
010f958>]    lr :5088>]    Not tainted
sp : c02abef4  ip : c02abf04  fp : c02abf00
r10: 00000300  r9 : 00000000  r8 : 00000000
r7 : c034b800  r6 : c034ba60  r5 : c034b800  r4 : 00000300
r3 : f200030a  r2 : 00000000  r1 : 0000030a  r0 : 00000300
Flags: Nzcv  IRQ  Mode SVC_32  Sernel
Control: 717F  Table: 30004000  DAC: 00000017
Process swapper (pid: 1, stack limit = 0xc02aa194)
Stack: (0xc02abef4 to 0)
bee0:                                     c02abf88 c02abf04 c0015088
bf00: c010f94c c002f4d4 c002e760 ffffffff ffffffff c02abf68 c02abf24 c00e2444
bf20: c00e1c48 0000000a ffffffff 000000 3fcb4800 c034b800 c019d52d
bf40: 00000000 00000000 c034b800 00000000 00000000 00000000 c02abf78
c02abf68
bf60: c002f550 c0018be0 c034b800 00000000 000 00000000 00000abfa4
bf80: c02abf8c c00158ac c0014fb0 c0018ba8 00000001 00000000 c02abfc0
c02abfa8
bfa0: c0014c8c c0015828 00000001 00000000 0000000c02abfc4 c0014d58:
c0014c64 c0
bfe0: 00000000 00000000 00000000 c02abff8 c0030990 c0019068 00000000
00000000
Backtrace:
[<c010f9dword+0x0/0x2c) 0015088>] (cs89x0_probe1+0xe8/0x878)
[<c0014fa0>]        (cs89x0_probe1+0x0/0x878)        from        [<c00158ac>]
(cs89x0_probe+0x915818) r6 = 00000000  r5 = 00000001  r4 = C0018BA8
[<c0014c54>]          (probe_list2+0x0/0x6c)          from          [<c0014d58>]
(net_olddevs_init+0x98/0x) r6 = 00000000  r5 = 00000000  r4 = 00000001
[<c0014cc0>] (net_olddevs_init+0x0/0xdc) frc>] (init+0x84/0x5 = C02AA000  r4 =
[<c0019058>] (init+0x0/0x20c) from [<c0030990>] (do_exit+0x0/0x700)
 r6 = 00000000  r5 = 00000000  r4 = 00000000 334f2 (e1d300b0)
 <0>Kernel panic - not syncing: Attempted to kill init!
```

옵스! CS89x0 드라이버를 올리는 과정에서 Oops 메시지가 나타났다. 0xF200030A에 대한 pgd를 참조한 값이 0인 것으로 보아 가상 메모리 매핑 정보가 존재하지 않는다. 게다가 0xF2000030A 자체가 CS8900A에 접근할 수 있는 주소가 아닐 가능성이 있다.

CS8900A가 어떻게 연결되어 있는지부터 확인해보자. 회로도를 보면 CS8900A는 nGCS 3 핀에 연결되어 있다. nGCS3은 4번 뱅크이고, 물리 주소로는 0x18000000~0x200000

00 영역이다. (S3C2410 User's Manual 5-2) 어드레스 라인 A0~A19는 SA0~SA19 (20비트, 1MB)에 각각 연결되어 있다. 그럼 0x18000000~0x1810000까지가 CS8900A에 매핑된 영역일까? 좀 더 유심히 관찰할 필요가 있다. A24핀이 nOE, nWE핀과 각각 OR 게이트의 입력으로 들어가서 빠져나온 출력이 nIO (nIOR, nIOW)와 nMEM (nMEMR, nMEMW)핀들과 연결되어 있다.



CS8900A는 내부 레지스터나 버퍼에 접근하기 위한 두 가지가 모드를 지원한다. 메모리 모드와 IO 모드가 바로 그것인데, 메모리 모드는 MEMR/MEMW 핀, IO 모드는 IOR/IOW 핀이 연관되어 있다. MEMx와 IOx는 Active Low 입력 핀이다.

MBA2410에서는 기본 모드인 IO 모드로 동작한다. IOR이나 IOW 핀에 Low가 인가되어야 읽기/쓰기 작업을 수행할 수 있다. IOR, IOW가 Low가 되려면 A24 핀이 High 상태가 되어야 한다. 물론 읽기, 쓰기 상태에 따라 nOE, nWE도 Low가 되어야 하지만 현재 관심 대상은 A24핀이다. A24핀이 High 상태가 되려면 0x18000000가 아닌 0x19000000 주소(25번째 비트 On)로 접근해야 한다. 이것으로 CS8900A의 메모리 매핑 시작 주소가 0x19000000로 확인되었다.

현재 0x19000000 주소가 가상 메모리에 매핑되어 있는지 확인해야 한다. 처음부터 눈으로 소스 코드를 일일이 찾아가거나 페이지 테이블에 직접 접근해서 알아 볼 수도 있지만 iotable_io() 함수의 파라메터 값들을 printk로 출력해보면 손쉽게 매핑 정보를 확인할 수 있다. 아직 0x19000000의 매핑 정보는 찾아 볼 수 없다. 직접 추가해야 한다.

커널 부팅시 mba2410_map_io()에서 mba2410_iodesc에 메모리 매핑 정보를 담은 구조체 배열을 넘겨준다는 사실을 기억한다면 바로 이곳에 CS8900A 매핑 데이터를 삽입하면 해결된다.

먼저 CS8900A 물리 메모리와 가상 메모리에 대한 매크로를 map.h에 추가한다.

include/asm-arm/arch-s3c2410/map.h

```
152 /* SDI */
153 #define S3C24XX_VA_SDI      S3C2410_ADDR(0x01200000)
154 #define S3C2410_PA_SDI      (0x5A000000)
155 #define S3C24XX_SZ_SDI      SZ_1M
156
>   /* CS8900 */
>   #define S3C24XX_VA_CS8900    S3C2410_ADDR(0x01300000)
>   #define S3C2410_PA_CS8900    (0x19000000)
>   #define S3C24XX_SZ_CS8900    SZ_1M
>   #define S3C24XX_PA_CS8900    S3C2410_PA_CS8900
```

mach-mba2410.c의 mba2410_iodesc 배열에는 초기화 값을 넣는다.

arch/arm/mach-s3c2410/mach-mba2410.c

```
57 static struct map_desc smdk2410_iodesc[] __initdata = {
>  IODESC_ENT(CS8900)
59 };
```

drivers/net/cs89x0.c

CS89x0 드라이버 구현 파일인 cs89x0.c를 열어보면 smdk2410 보드와 관련된 부분은 없다. 포트 주소와 IRQ 정보를 알맞게 추가해야 한다.

```
155 #endif
156
>  #ifdef CONFIG_ARCH_S3C2410
>  #include <asm/arch/map.h>
>  #include <asm/arch/irqs.h>
>  #include <asm/arch/regs-mem.h>
>  #include <asm/arch/regs-gpio.h>
>  #endif
                                        :
                                        :
191 #elif defined(CONFIG_ARCH_PNX010X)
192 #include <asm/irq.h>
```

```
193 #include <asm/arch/gpio.h>
194 #define CIRRUS_DEFAULT_BASE     IO_ADDRESS(EXT_STATIC2_s0_BASE + 0x2
    00000)      /* = Physical address 0x48200000 */
195 #define CIRRUS_DEFAULT_IRQ      VH_INTC_INT_NUM_CASCADED_INTERRUPT_
    1 /* Event inputs bank 1 - ID 35/bit 3 */
196 static unsigned int netcard_portlist[] __initdata = {CIRRUS_DEFAULT_BASE, 0};
197 static unsigned int cs8900_irq_map[] = {CIRRUS_DEFAULT_IRQ, 0, 0, 0};
>   #elif defined(CONFIG_ARCH_S3C2410)
>   static unsigned int netcard_portlist[] __initdata = { S3C24XX_VA_CS8900 + DEF
    AU\LTIOBASE, 0};
>   static unsigned int cs8900_irq_map[] = { IRQ_EINT9, 0, 0, 0 };
>   #ifdef request_region
>   #undef request_region
>   #endif
>   #ifdef release_region
>   #undef release_region
>   #endif
>   #define request_region(a, s, n) request_mem_region(a, s, n)
>   #define release_region(a, s) release_mem_region(a, s)
198 #else
199 static unsigned int netcard_portlist[] __initdata =
200    { 0x300, 0x320, 0x340, 0x360, 0x200, 0x220, 0x240, 0x260, 0x280, 0x2a0,
    0x2c0, 0x2e0, 0};
201 static unsigned int cs8900_irq_map[] = {10,11,12,5};
202 #endif
                                        :
                                        :
325         io = dev->base_addr;
326         irq = dev->irq;
327
>   #ifdef CONFIG_ARCH_S3C2410     // initialize CS8900, jdh added
>       __raw_writel((__raw_readl(S3C2410_GPGCON)&~(0x3<<2))|(0x2<<2),S3C2
    410_GPGCON);
>       __raw_writel((__raw_readl(S3C2410_EXTINT1)&~(0x7<<4))|(0x4<<4),S3C2
    410_EXTINT1);
>   #endif                                  // jdh ended
>
328         if (net_debug)
329                 printk("cs89x0:cs89x0_probe(0x%x)\n", io);
                                        :
                                        :
350         return ERR_PTR(err);
351 }
352 #endif
```

```
353
>   #if defined(CONFIG_ARCH_S3C2410)
>   static u16
>   readword(unsigned long base_addr, int portno)
>   {
>           return __raw_readw(base_addr + portno);
>   }
>
>   static void
>   writeword(unsigned long base_addr, int portno, u16 value)
>   {
>           __raw_writew(value, base_addr + portno);
>   }
>   #elif defined(CONFIG_MACH_IXDP2351)
<   #if defined(CONFIG_MACH_IXDP2351)
                                        :
                                        :
633         reset_chip(dev);
634
>   #ifdef CONFIG_ARCH_S3C2410
>           lp->force = FORCE_RJ45;
>           lp->auto_neg_cnf = IMM_BIT;
>
>           dev->dev_addr[0] = 0x09;   /* set MAC address */
>           dev->dev_addr[1] = 0x90;
>           dev->dev_addr[2] = 0x99;
>           dev->dev_addr[3] = 0x09;
>           dev->dev_addr[4] = 0x90;
>           dev->dev_addr[5] = 0x99;
>   #endif
                                        :
                                        :
<    #if !defined(CONFIG_MACH_IXDP2351) && !defined(CONFIG_ARCH_IXDP2X01) &
      & !defined(CONFIG_ARCH_PNX010X)
>    #if !defined(CONFIG_MACH_MBA2410) && !defined(CONFIG_MACH_IXDP2351) &
      & !defined(CONFIG_ARCH_IXDP2X01) && !defined(CONFIG_ARCH_PNX010X)
1314                if (((1 << dev->irq) & lp->irq_map) == 0) {
1315                        printk(KERN_ERR "%s: IRQ %d is not in our map of all
     owable IRQs, which is %x\n",
1316                               dev->name, dev->irq, lp->irq_map);
```

CS8900A는 EEPROM에 대한 인터페이스를 제공하지만 MBA2410에는 별도로 부착된 EEPROM이 없다. MAC 어드레스가 저장돼 있어야할 EEPROM이 없기 때문에 모듈이 초기화될 때마다 임의의 값으로 MAC 어드레스를 설정하도록 했다. (#634)

ARM 리눅스의 NR_IRQS 값은 128이다. CS8900A에서 사용하는 IRQ_EINT9의 커널 내부 IRQ 번호는 53번이다. lp->irq_map의 32비트로는 IRQ 맵 정보를 담을 수 없다. (#1314)

```
                                        :
cs89x0:cs89x0_probe(0x0)
cs89x0.c: v2.4.3-pre1 Russell Nelson <nelson@crynwr.com>, Andrew Morton
<andrewm@uow.edu.au>
<6>eth0: cs8900 rev K found at 0xf1300300
<6>cs89x0 media RJ-45, IRQ 53, programmed I/O, MAC 09:90:99:09:90:99
cs89x0_probe1() successful
<7>CLASS: registering class device: ID = 'eth0'
<7>cs89x0: request_region(0xf1300300, 0x10) failed
<4>cs89x0: no cs8900 or cs8920 detected.  Be sure to disable PnP with SETUP
                                        :
```

- CS8900A Product Data Sheet (http://www.cirrus.com/en/pubs/proDatasheet/CS8900A_F3.pdf)
- aesop-2440a 2.6.13 CS89x0.c (http://kelp.or.kr/korweblog/stories.php?story=05/12/30/4249634&topic=45)
- Oops Tracing (Documentation/oops-tracing.txt)

■ NFS 파일 시스템

다시 테스트 해보면 이더넷 드라이버까지는 문제없이 설정했지만 루트 디바이스를 찾을 수 없어서 커널 패닉이 발생한다.

```
                                        :
VFS: Cannot open root device "<NULL>" or unknown-block(2,0)
Please append a correct "root=" boot option
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(2,0)
```

커널이 온전히 부팅되려면 루트 파일 시스템이 마운트 되어야 한다. 루트 파일 시스템에는 리눅스에서 필요한 기본 명령 프로그램, 프로그램을 실행하기 위한 라이브러리, 설정 파일, 디바이스 파일 등이 일정한 디렉터리 구조로 구성되어 있다.

루트 파일 시스템으로는 원격의 파일 시스템을 루트 파일 시스템으로 마운트하는 NFS를 이용하려고 한다. 루트 파일 시스템은 ELDK에 포함된 것을 사용한다.

커널 환경설정에서 NFS 부팅을 위한 다음 옵션을 선택한다.

```
Networking > Networking support > Networking options > TCP/IP networking > IP: kernel level autoconfiguration (IP_PNP)
File systems > Network File Systems > NFS file system support (NFS_FS)
File systems > Network File Systems > Root file system on NFS (ROOT_FS)
```

커널을 다시 컴파일하고 테스트하기에 전에 U-Boot의 bootargs 환경변수를 아래와 같이 설정한다.

```
MBA2410 # set bootargs root=/dev/nfs rw nfsroot=192.168.1.105:/opt/eldk/arm ip=192.168.1.200:192.168.1.105::255.255.255.0:::off
MBA2410 # saveenv
```

NFS 서버 주소는 192.168.1.105, NFS 경로는 /opt/eldk/arm, MBA2410에 부여된 IP는 192.168.1.200이다.

NFS 서버나 MBA2410에서 이상 없이 설정했다면 루트 파일 시스템 마운트 문제는 해결됐다. 하지만 다른 문제가 남아 있다. 콘솔 디바이스를 지정해야 쉘 프롬프트를 볼 수 있다. 기본 콘솔인 tty 터미널은 현재 LCD를 사용할 수 있는 상태가 아니기 때문에 시리얼(ttyS)을 콘솔 디바이스로 설정해야 한다.

일반적으로 시리얼 장치명은 ttyS를 사용하지만 S3C2410 시리얼처럼 다른 이름을 사용하는 경우도 있다. S3C2410의 시리얼은 ttySAC를 사용하고 메이저 넘버는 204, 마이너 넘버는 64번을 사용한다. 이 장치를 이용하려면 /dev에 mknod로 ttySAC 장치 파일를 만들거나 아니면 ttySAC를 ttyS로 변경하는 방법이 있다.

다음은 ttySAC 장치 파일를 생성하는 방법이다.

```
# cd dev
dev # mknod ttySAC0 c 204 64
dev # mknod ttySAC1 c 204 65
```

ttySAC를 ttyS로 바꾸려면 소스 코드를 수정해야 한다.

drivers/serial/s3c2410.c

```
149 /* UART name and device definitions */
150
<    #define S3C24XX_SERIAL_NAME     "ttySAC"
<    #define S3C24XX_SERIAL_DEVFS    "tts/"
<    #define S3C24XX_SERIAL_MAJOR    204
<    #define S3C24XX_SERIAL_MINOR    64
>    #define S3C24XX_SERIAL_NAME     "ttyS"
>    #define S3C24XX_SERIAL_DEVFS    "tts/"
>    #define S3C24XX_SERIAL_MAJOR    204
>    #define S3C24XX_SERIAL_MINOR    4
```

ttyS도 mknod를 통해서 장치 파일를 만들 수 있지만 ELDK에서는 ELDK_MAKEDEV 스크립트로 주요 장치 노드들을 한번에 생성할 수 있다.

```
# cd dev
# /mnt/cdrom/ELDK_MAKEDEV
```

커널 파라메터 console에서 장치명과 전송 속도를 명시한다.

```
MBA2410 # set bootargs root=/dev/nfs rw nfsroot=192.168.1.105:/opt/eldk/arm
ip=192.168.1.200:192.168.1.105::255.255.255.0:::off console=ttyS0,115200
MBA2410 # saveenv
```

콘솔 장치까지 설정하고 나면 vprintk()에 추가한 printascii는 더 이상 필요없다. printk()로 출력하는 메시지를 시리얼로 통해 받아 볼 수 있기 때문이다. login 화면을 보았는가? root 사용자로 로그인해보자.

- Embedded Linux Development Kit (http://www.denx.de/wiki/DULG/ELDK)
- Mounting the root filesystem via NFS (Documentation/nfsroot.txt)
- Linux Serial Console (Documentation/serial-console.txt)
- Kernel Parameters (Documentation/kernel-parameters.txt)

■ LCD 설정

시리얼 터미널 대신 LCD 화면으로 콘솔 출력 화면을 볼 수 없을까? 부팅 메시지에서 LCD 장치에 대한 내용을 찾아보자.

```
                                        :
<3>s3c2410-lcd s3c2410-lcd: no platform data for lcd, cannot attach
                                        :
```

LCD에 대한 플랫폼 데이터가 없다고 한다. 에러 메시지의 출처는 s3c2410fb_probe() 함수이다. 커널에 의해 s3c2410fb_probe()가 호출되기 전에 프레임 버퍼 LCD 관련 플랫폼 데이터를 미리 초기화시켜야 한다. 막연하게 느껴질 수도 있지만 걱정하지 않아도 된다. 다행스럽게도 mach-smdk2440.c에 선언된 플랫폼 데이터를 그대로 가져다 사용하면 문제없이 LCD를 동작시킬 수 있다.

mach-smdk2440.c의 smdk2440_lcd_cfg 변수 선언을 비롯한 관련 코드를 복사해서 mach-mba2410.c에 붙여 넣는다.

arch/arm/mach-s3c2410/mach-mba2410.c

```
>   /* LCD driver info */
>
>   static struct s3c2410fb_mach_info mba2410_lcd_cfg __initdata = {
>       .regs    = {
>               .lcdcon1 = S3C2410_LCDCON1_TFT16BPP |
>                                   S3C2410_LCDCON1_TFT |
>                                   S3C2410_LCDCON1_CLKVAL(0x04),
>
>               .lcdcon2 = S3C2410_LCDCON2_VBPD(7) |
>                                   S3C2410_LCDCON2_LINEVAL(319) |
>                                   S3C2410_LCDCON2_VFPD(6) |
>                                   S3C2410_LCDCON2_VSPW(3),
>
>               .lcdcon3 = S3C2410_LCDCON3_HBPD(19) |
>                                   S3C2410_LCDCON3_HOZVAL(239) |
>                                   S3C2410_LCDCON3_HFPD(7),
>
>               .lcdcon4 = S3C2410_LCDCON4_MVAL(0) |
>                                   S3C2410_LCDCON4_HSPW(3),
>
>               .lcdcon5 = S3C2410_LCDCON5_FRM565 |
>                                   S3C2410_LCDCON5_INVVLINE |
>                                   S3C2410_LCDCON5_INVVFRAME |
>                                   S3C2410_LCDCON5_PWREN |
>                                   S3C2410_LCDCON5_HWSWP,
>       },
```

```
>
>  #if 0
>       /* currently setup by downloader */
>       .gpccon         = 0xaa940659,
>       .gpccon_mask    = 0xffffffff,
>       .gpcup          = 0x0000ffff,
>       .gpcup_mask     = 0xffffffff,
>       .gpdcon         = 0xaa84aaa0,
>       .gpdcon_mask    = 0xffffffff,
>       .gpdup          = 0x0000faff,
>       .gpdup_mask     = 0xffffffff,
>  #endif
>
>       .lpcsel         = ((0xCE6) & ~7) | 1<<4,
>
>       .width          = 240,
>       .height         = 320,
>
>       .xres           = {
>               .min    = 240,
>               .max    = 240,
>               .defval = 240,
>       },
>
>       .yres           = {
>               .min    = 320,
>               .max    = 320,
>               .defval = 320,
>       },
>
>       .bpp            = {
>               .min    = 16,
>               .max    = 16,
>               .defval = 16,
>       },
>  };
>
>   static void __init mba2410_machine_init(void)
>   {
>       s3c24xx_fb_set_platdata(&mba2410_lcd_cfg);
>
>       s3c2410_pm_init();
>   }
114
```

```
115 MACHINE_START(MBA2410, "MBA2410") /* @TODO: request a new identifier
    and switch
116                                     * to MBA2410 */
117         /* Maintainer: Jonas Dietsche */
118         .phys_io        = S3C2410_PA_UART,
119         .io_pg_offst    = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
120         .boot_params    = S3C2410_SDRAM_PA + 0x100,
121         .map_io         = mba2410_map_io,
122         .init_irq       = mba2410_init_irq,
123         .timer          = &s3c24xx_timer,
>           .init_machine   = mba2410_machine_init,
124 MACHINE_END
```

커널 환경설정에서 프레임버퍼 관련 항목을 선택한다.

```
Device Drivers > Graphics Support > Support form frame buffer devices (FB)
Device Drivers > Graphics Support > S3C2410 LCD framebuffer support (FB_S3C2410)
Device Drivers > Graphics Support > Console display driver support > Framebuffer Console support (FRAMEBUFFER_CONSOLE)
```

추가로 다음 항목을 선택하면 콘솔 화면을 회전시키거나 폰트 종류를 선택할 수 있다.

```
Device Drivers > Graphics Support > Console display driver support > Framebuffer Console support > Framebuffer Console Rotation (FRAMEBUFFER_CONSOLE_ROTATION)
Device Drivers > Graphics Support > Console display driver support > Select compiled-in fonts (FONTS)
```

커널 파라메터에서 콘솔 디바이스를 tty0으로 설정한다.

```
MBA2410 # set bootargs root=/dev/nfs rw nfsroot=192.168.1.105:/opt/eldk/arm
ip=192.168.1.200:192.168.1.105::255.255.255.0:::off console=tty0
MBA2410 # saveenv
```

이제 LCD에 리눅스 부팅 화면이 나타날 것이다. ttyS를 콘솔 디바이스로 사용할 때는 키 입력을 시리얼 통신을 통해 주고 받았지만 tty로 바꾸고 나면 별도의 입력 디바이스가 없기 때문에 키 입력을 받을 수 없는 문제점이 있다.

- MBA2410 리눅스 환경에서 LCD Display 문제점 해결 (http://www.aijisystem.com/root/customer/down_view.php?uid=2565&sub_name=cu_down1&kind=cu_down1&re_href=down1.php&page=1)
- The Framebuffer Console (Documentation/fb/fbcon.txt)
- The Framebuffer Device (Documentation/fb/framebuffer.txt)

■ 키 매트릭스 드라이버

MBA2410에는 12개의 버튼이 3x4 매트릭스 형태로 부착되어 있다. (드라이버 소스 코드에는 키 매트릭스 대신 키패드라는 호칭을 사용했다.) 키 매트릭스를 사용하기 위해 간단한 디바이스 드라이버를 제작했다.

MBA2410의 회로도는 공개되어 있지 않기 때문에 본 문서상에 나타낼 수 없다. MBA2410 매뉴얼을 참조하기 바란다. (문서를 작성하고 있는 현재 대여했던 보드와 회로도가 들어 있는 매뉴얼을 반납한 상태에서 쓰는 내용이라 회로도의 설명 내용이 사실과 다를 수 있고, 자세한 설명을 할 수 없는 점 양해 바란다.)

12개의 버튼이 각각 S3C2410의 12개의 핀에 연결된 것이 아니라 3개의 열과 4개의 행으로 총 7개의 핀이 사용된다. 열에 대한 핀(EINT2, EINT11, EINT19)은 출력 포트로 설정해서 해당 열을 선택(Low 신호 출력)하는 역할을 담당하고, 행에 대한 핀(EINT4~EINT7)은 인터럽트 포트로 설정해서 버튼이 눌리면(Active Low) 입력 이벤트가 발생할 수 있도록 해야 한다.

drivers/input/keyboard/mba2410_kpad.c

```
 1 #include <linux/input.h>
 2 #include <linux/module.h>
 3 #include <linux/init.h>
 4 #include <linux/interrupt.h>
 5
 6 #include <asm/irq.h>
 7 #include <asm/io.h>
 8 #include <asm/system.h>
 9 #include <asm/atomic.h>
10 #include <asm/arch/regs-gpio.h>
11
12 MODULE_LICENSE("GPL");
13
```

```
14 #define INIT_IRQ4                   1
15 #define INIT_IRQ5                   2
16 #define INIT_IRQ6                   4
17 #define INIT_IRQ7                   8
18 #define INIT_TIMER          16
19
20 static int init_flags;
21 static int last_error;
22
23 #define GPFCON_EINT2_SHIFT        4
24 #define GPFCON_EINT4_SHIFT        8
25 #define GPFCON_EINT5_SHIFT        10
26 #define GPFCON_EINT6_SHIFT        12
27 #define GPFCON_EINT7_SHIFT        14
28 #define GPGCON_EINT11_SHIFT       6
29 #define GPGCON_EINT19_SHIFT       22
30
31 #define GPFDAT_EINT2_SHIFT        2
32 #define GPGDAT_EINT11_SHIFT       3
33 #define GPGDAT_EINT19_SHIFT       11
34
35 #define COLUMN_DELAY              (HZ / 20)
36
37 #define KEY_DELAY           3
38
39 enum KPAD_COLUMN { KPAD_COL0, KPAD_COL1, KPAD_COL2 };
40
41 static struct input_dev* kpad_dev;
42 static int column = -1;
43 static int key_delay;
44
45 static void reset_column(void);
46
47 static int kpad_matrix[4][3] = {
48     { KEY_1, KEY_2, KEY_3 },
49     { KEY_4, KEY_5, KEY_6 },
50     { KEY_7, KEY_8, KEY_9 },
51     { KEY_SPACE, KEY_0, KEY_ENTER }
52 };
53
54 static void init_regs(void)
55 {
56     unsigned long reg_value, combined;
57
```

```
 58     /* set EINT2 as OUTPUT port, EINT4~7 as INTERRUPT. */
 59     reg_value = __raw_readl(S3C2410_GPFCON);
 60     combined = (3 << GPFCON_EINT2_SHIFT) | (3 << GPFCON_EINT4_SHIFT)
| (3 << GPFCON_EINT5_SHIFT);
 61     combined |= (3 << GPFCON_EINT6_SHIFT) | (3 <<
GPFCON_EINT7_SHIFT);
 62     __raw_writel((reg_value & ~combined) | S3C2410_GPF7_EINT7 |
S3C2410_GPF6_EINT6 | S3C2410_GPF5_EINT5 | S3C2410_GPF4_EINT4 |
S3C2410_GPF2_OUTP, S3C2410_GPFCON);
 63
 64     /* set EINT11, EINT19 as OUTPUT */
 65     reg_value = __raw_readl(S3C2410_GPGCON);
 66     combined = (3 << GPGCON_EINT11_SHIFT) | (3 <<
GPGCON_EINT19_SHIFT);
 67     __raw_writel((reg_value & ~combined) | S3C2410_GPG3_OUTP |
S3C2410_GPG11_OUTP, S3C2410_GPGCON);
 68
 69     reset_column();
 70 }
 71
 72 static void reset_column(void)
 73 {
 74     unsigned long reg_value, combined;
 75
 76     /* disable all buttons */
 77     reg_value = __raw_readb(S3C2410_GPFDAT);
 78     __raw_writeb(reg_value | (1 << GPFDAT_EINT2_SHIFT),
S3C2410_GPFDAT);
 79
 80     reg_value = __raw_readw(S3C2410_GPGDAT);
 81     combined = (1 << GPGDAT_EINT11_SHIFT) | (1 <<
GPGDAT_EINT19_SHIFT);
 82     __raw_writew(reg_value | combined, S3C2410_GPGDAT);
 83
 84     column = -1;
 85 }
 86
 87 static void select_column(int col)
 88 {
 89     unsigned long reg_value;
 87
 90     reset_column();
 91
 92     /* enable buttons on the "col"th column */
```

```
 93
 94      switch (col) {
 95              case KPAD_COL0:
 96                      reg_value = __raw_readb(S3C2410_GPFDAT);
 97                      __raw_writeb((reg_value          &          ~(1          <<
GPFDAT_EINT2_SHIFT)), S3C2410_GPFDAT);
 98                      column = KPAD_COL0;
 99                      break;
100              case KPAD_COL1:
101                      /* we have to toggle both pins at once */
102                      reg_value = __raw_readw(S3C2410_GPGDAT);
103                      __raw_writew((reg_value          &          ~(1          <<
GPGDAT_EINT11_SHIFT)) | (1 << GPGDAT_EINT19_SHIFT), S3C2410_GPGDAT);
104                      column = KPAD_COL1;
105                      break;
106             case KPAD_COL2:
107                      /* we have to toggle both pins at once */
108                      reg_value = __raw_readw(S3C2410_GPGDAT);
109                      __raw_writew((reg_value          &          ~(1          <<
GPGDAT_EINT19_SHIFT)) | (1 << GPGDAT_EINT11_SHIFT), S3C2410_GPGDAT);
110                      column = KPAD_COL2;
111                      break;
112             default:
113                      printk("Invalid column\n");
114                      break;
115     }
116 }
117
118 static void rotate_column(unsigned long unused)
119 {
120     unsigned long flags;
121
122     local_irq_save(flags);
123
124     if (key_delay) key_delay--;
125
126     if (column < KPAD_COL0 || column > KPAD_COL2) {
127             select_column(KPAD_COL0);
128     } else {
129             select_column ((column + 1) % (KPAD_COL2 + 1));
130     }
131
132     mod_timer(&kpad_dev->timer, jiffies + COLUMN_DELAY);
133
```

```
134     local_irq_restore(flags);
135 }
136
137 irqreturn_t kpad_interrupt(int irq, void *dummy, struct pt_regs *fp)
138 {
139     unsigned long flags;
140     int key;
141
142     if (key_delay) return IRQ_HANDLED;
143
144     local_irq_save(flags);
145
146     key = kpad_matrix[irq - IRQ_EINT4][column];
147
148 //  printk("%d Key Pressed : IRQ %d Column %d\n", key, irq, column);
149
150     input_report_key(kpad_dev, key, 1);
151     input_sync(kpad_dev);
152     input_report_key(kpad_dev, key, 0);
153     input_sync(kpad_dev);
154
155     key_delay = KEY_DELAY;
156
157     local_irq_restore(flags);
158
159     return IRQ_HANDLED;
160 }
161
162 static int kpad_cleanup(void)
163 {
164     input_unregister_device(kpad_dev);
165
166     if (init_flags & INIT_IRQ4) free_irq(IRQ_EINT4, kpad_interrupt);
167     if (init_flags & INIT_IRQ5) free_irq(IRQ_EINT5, kpad_interrupt);
168     if (init_flags & INIT_IRQ6) free_irq(IRQ_EINT6, kpad_interrupt);
169     if (init_flags & INIT_IRQ7) free_irq(IRQ_EINT7, kpad_interrupt);
170
171     if (init_flags & INIT_TIMER) del_timer(&kpad_dev->timer);
172 }
173
174 static int __init kpad_init(void)
175 {
176     init_regs();
177
```

```
178     kpad_dev = input_allocate_device();
179
180     if (!kpad_dev) {
181             printk(KERN_ERR "Can't allocate input_device\n");
182             return -ENOMEM;
183     }
184
185     if (request_irq(IRQ_EINT4, kpad_interrupt, 0, "kpad_row0", NULL)) {
186             printk(KERN_ERR "Can't allocate irq %d\n", IRQ_EINT4);
187             last_error = -EBUSY;
188             goto fail;
189     } else init_flags |= INIT_IRQ4;
190
191     if (request_irq(IRQ_EINT5, kpad_interrupt, 0, "kpad_row1", NULL)) {
192             printk(KERN_ERR "Can't allocate irq %d\n", IRQ_EINT5);
193             last_error = -EBUSY;
194             goto fail;
195     } else init_flags |= INIT_IRQ5;
196
197     if (request_irq(IRQ_EINT6, kpad_interrupt, 0, "kpad_row2", NULL)) {
198             printk(KERN_ERR "Can't allocate irq %d\n", IRQ_EINT6);
199             last_error = -EBUSY;
200             goto fail;
201     } else init_flags |= INIT_IRQ6;
202
203     if (request_irq(IRQ_EINT7, kpad_interrupt, 0, "kpad_row3", NULL)) {
204             printk(KERN_ERR "Can't allocate irq %d\n", IRQ_EINT7);
205             last_error = -EBUSY;
206             goto fail;
207     } else init_flags |= INIT_IRQ7;
208
209     kpad_dev->name = "MBA2410 Keypad Driver";
210     kpad_dev->phys = "foobar/input0";
211     kpad_dev->id.bustype = BUS_HOST;
212     kpad_dev->id.vendor = 0x0909;
213     kpad_dev->id.product = 0x0009;
214     kpad_dev->id.version = 0x001;
215
216     set_bit(EV_KEY, kpad_dev->evbit);
217     set_bit(KEY_1, kpad_dev->keybit);
218     set_bit(KEY_2, kpad_dev->keybit);
219     set_bit(KEY_3, kpad_dev->keybit);
220     set_bit(KEY_4, kpad_dev->keybit);
221     set_bit(KEY_5, kpad_dev->keybit);
```

```
222     set_bit(KEY_6, kpad_dev->keybit);
223     set_bit(KEY_7, kpad_dev->keybit);
224     set_bit(KEY_8, kpad_dev->keybit);
225     set_bit(KEY_9, kpad_dev->keybit);
226     set_bit(KEY_SPACE, kpad_dev->keybit);
227     set_bit(KEY_0, kpad_dev->keybit);
228     set_bit(KEY_ENTER, kpad_dev->keybit);
229
230     input_register_device(kpad_dev);
231
232     key_delay = 0;
233
234     init_timer(&kpad_dev->timer);
235
236     kpad_dev->timer.function = rotate_column;
237     kpad_dev->timer.expires = jiffies + HZ;
238
239     add_timer(&kpad_dev->timer);
240     init_flags |= INIT_TIMER;
241
242     return 0;
243
244 fail:
245     kpad_cleanup();
246
247     return last_error;
248 }
249
250 static void __exit kpad_exit(void)
251 {
252     kpad_cleanup();
253 }
254
255 module_init(kpad_init);
256 module_exit(kpad_exit);
```

행과 열 서로 교차하며 있기 때문에 2개 이상의 열을 선택하면 어느 열의 버튼이 눌렸는지 구분할 수 없다. 한 번에 하나씩 차례대로 반복해서 열을 선택해야 한다. 1 / (HZ / 20)초마다 한 번씩 타이머 인터럽트를 발생시켜 열을 번갈아가며 바꾸도록 했다. ARM 리눅스의 HZ 값은 200이다.

EINT4~EINT7까지 4개의 행에 대한 IRQ를 요청한다. 버튼에 대한 인터럽트 발생시 kpad_interrupt() 핸들러가 실행된다. 어느 버튼이 눌렸는지 판단한 다음 input_report_key()로 해당 키 값을 input core에 넘겨준다. 버튼을 한번만 눌러도 미세한 잡음으로 여러 번

의 누른 것처럼 인식되는 채터링(chattering) 현상으로 인해 하나의 버튼을 처리하고 나서 일정한 시간동안 버튼을 처리하지 못하도록 했다.

소스 파일을 저장한 다음 Makefile과 Kconfig 파일에 관련 내용을 추가한다.

drivers/input/keyboard/Makefile

```
> obj-$(CONFIG_KEYPAD_MBA2410) += mba2410_kpad.o
```

drivers/input/keyboard/Kconfig

```
> config KEYPAD_MBA2410
>         tristate "MBA2410 Keypad"
>         select MACH_MBA2410
```

커널 환경설정에서 KEYPAD_MBA2410를 선택하고 다시 컴파일 한 뒤 커널 부팅해보자. 비록 숫자에 한정되었지만 키 매트릭스로 키 입력을 할 수 있다.

```
Device Drivers > Input device support > Keyboards > MBA2410 keypad (KEYPAD_MBA2410)
```

- The Linux USB Input Subsystem, Part I (http://www.linuxjournal.com/article/6396)
- Linux Device Drivers, 3rd edition - (O'Reilly, http://lwn.net/Kernel/LDD3/)
- Programming input drivers (Documentation/input/input-programming.txt)
- cli()/sti() removal guide (Documentation/cli-sti-removal.txt)

■ References

- **ARM** (http://arm.com)
- **The ARM Linux Project** (http://www.arm.linux.org.uk)
- **Linux Cross Reference** (http://www.linux-m32r.org/lxr/http/source)
- **LWN** (http://lwn.net/)
- **KernelTrap** (http://kerneltrap.org)
- **Linux Journal** (http://www.linuxjournal.com)
- **Korea Embedded Linux Project** (http://kelp.or.kr)

- **ARM System Developer's Guide** - Sloss, Symes, Wright (사이텍미디어)
- **Linux Kernel Development, 2nd edition** - Robert Love (Novell Press)
- **The Linux Kernel Primer** - Claudia Salzberg Rodriguez 외 2인 (Prentice Hall PTR)
- **Understanding the Linux Kernel, 3ed** - Daniel P. Bovet and Marco Cesati (O'Reilly)
- **Linux Device Drivers, 3rd edition** - Jonathan Corbet 외 2인 (O'Reilly)

- **Porting the Linux Kernel to a New ARM Platform** - Intel PCA Developers Network volume. 4, summer 2002
- **ARM 부트로더 제작기 #1~#9** - 월간 마이크로소프트웨어 2003년 10월호 ~ 2004년 7월호

- **MBA2410 Manual** (http://www.digital-kid.co.kr/down/evboard/MBA-2410_manual_no_chapter_4_5.pdf)
- **S3C2410 User's Manual Rev 1.2** (http://www.samsung.com/Products/Semiconductor/SystemLSI/MobileSolutions/MobileASSP/MobileComputing/S3C2410X/um_s3c2410s_rev12_030428.pdf)
- **CS8900A Product Data Sheet** (http://www.cirrus.com/en/pubs/proDatasheet/CS8900A_F3.pdf)

- **The DENX U-Boot and Linux Guide (DULG) for TQM8xxL** (http://www.denx.de/wiki/DULG/Manual)