4장 u-boot의 startup코드 분석

어제 드뎌 5일동안 나를 절라 고생시켰던 문제가 잡혀서 기분이 절라 좋습니다.
그러나 앞으로는 커널이라는 큰 산이 있어서 막막하긴 하지만요.

어제 후배만나서 술먹고 늦잠까정 자고...--; 하여간 오늘은 프로그램은 글렀고
글이나 쓸랍니다....^^

이것을 보기전에 봐야할 문서들이 있습니다. 물론 arm assembly에 대해 잘 아신다면야
뭐 이것을 볼 필요도 없고, 이 코드 분석 자체도 큰 의미가 없겠지만요...^^

제가 전에 올렸던 자료 리스트랑 인터넷 혹은 다른데서라도 해당 자료를 구해서 보시기 바랍니다.

1. arm7강좌.hwp(자료 리스트에 링크걸어놨음)
2. ADS(ARM Development Suite)의 assembly guide
3. GNU ld manual
4. S3C2410 manual
5. DRAM manual(dram종류는 저번강좌중 hw쪽 간단한 브리핑때 설명했으니 알아서 보시고용..^^)

요거는 있어야 보기가 좀 쉬울겁니다.

그리고, 분석을 올리기에 앞서서....

1. 저는 assembly 절라 못합니다. 즉, 보고 틀린게 있음 욕하지 마시고용...^^
2. u-boot-1.1.1을 기본으로 재작성하였습니다(원래는 0.4.0때 분석했던건데...)
이유는 코드가 조금 바뀌었고 좀 낫더군요...^^(relocation code부분)

그럼 startup code들어갑니다.

봐야할 파일들을 나열하면

board/smdk2410/u-boot.lds
board/smdk2410/config.mk
board/smdk2410/memsetup.S
cpu/arm920t/start.S

주로 start.S를 볼 것이고, 나머지는 그냥 referece정도입니다.

큰 코드의 흐름은

1> cpu초기화
2> dram초기화
3> flash에 있는 코드를 작업공간인(relocation area) dram으로 relocate
4> C쪽 코드의 메인인 board/lib_arm/board.c의 start_armboot()로 점프(여기서부터
dram에서 코드가 돌기 시작하겠지요...맞나?....^^)

입니다.

간단하죠...코드도 간단합니다만 약간의 괴리가 있는 부분이 있습니다.

5> 원래 코드가 돌아야할 위치는 config.mk에서 TEXT BASE로 지정을 합니다.
하지만 cpu가 처음 부팅되었을 때는 부트로더를 flash에서 읽어옵니다.
그래서 flash에서 2>까지의 코드를 돌리고 3>이 끝난후에 dram에서 작업을 합니다.

왜 괴리가 발생을 하느냐 하면 모든 연산을 offset개념으로만 하게 되어 있습니다.
왜냐하면 현재 코드가 도는 번지랑 실제 동작해야할 번지랑 일치가 안되기 때문이죠.
(relocation전에는)

큰 흐름을 유념하시면 별 어려움 없이 보실 수 있을겁니다.

board/smdk2410/config.mk

내용
```
========================================
#
# SMDK2410 has 1 bank of 64 MB DRAM
#
# 3000'0000 to 3400'0000
#
# Linux-Kernel is expected to be at 3000'8000, entry 3000'8000
# optionally with a ramdisk at 3080'0000
#
# we load ourself to 33F8'0000
```

```
#
# download area is 3300'0000
#


TEXT_BASE = 0x33F80000
=======================================
```

아주 설명이 잘되어 있습니다. 일단 bank6에 64메가 dram이 붙어있고요(see s3c2410 manual's memory controller part).
리눅스 커널은 physical ram start address+0x8000에 로딩이 됩니다. 그래서 entry point가 3000'8000입니다. 여기에 ramdisk가 들어가는 주소가 3080'0000인데 뭐 개발시 nfs를 쓸때는 큰 의미가 없고요.

u-boot의 프롬프트에서 작업할때 다운로드하는 작업공간의 시작 주소를 3300'0000로 하고요
(커널 다운로드 받을때등등.... ex> tftp 03000000 kernel_img)

TEXT_BASE가 중요한 넘인데, 부트로더가 올라가야할 address를 지정합니다. 즉, relocation시 destination address가 되고, 부트로더(모니터코드등등으로 불리는)의 실제 위치가 되는것이죠.

이 TEXT_BASE는 linker(gnu ld)옵션으로 들어가서 linker script(u-boot.lds)에 지정된 text address 에 더하기가 됩니다. 즉, 컴파일된 실행코드인 text의 base가 0일 경우 실제 로딩되고 배치되어야 할 번지는 0+TEXT_BASE입니다. 즉, 시작점을 지정하는 것이죠...(내가 읽어봐도 말 어렵넹...^^)

u-boot.lds랑 ld의 -Ttext 옵션을 보시면 크게 어려울것은 없습니다.

※ ld의 -Ttext org 옵션
-Ttext org
org를 출력 파일의 bss, data, 또는 text 세그먼트---각각에--- 대해서 시작 주소로 사용하도록 한다. org는 반드시 16진수 정수이어야 한다. 다른 링커들과의 호환성을 위하여 여러분은 16진수 값 앞에 보통 붙이는 `0x'를 생략할 수도 있다.

자 그럼 이제는 u-boot.lds를 보겠습니다.(링커한테 코드배치를 이렇게 하라고 설명하는 파일입니다)

board/smdk2410/u-boot.lds
내용

```
====================================
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
/*OUTPUT_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")*/
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
        . = 0x00000000;

        . = ALIGN(4);
        .text      :
        {
          cpu/arm920t/start.o   (.text)
          *(.text)
        }

        . = ALIGN(4);
        .rodata : { *(.rodata) }

        . = ALIGN(4);
        .data : { *(.data) }

        . = ALIGN(4);
        .got : { *(.got) }

        __u_boot_cmd_start = .;
        .u_boot_cmd : { *(.u_boot_cmd) }
        __u_boot_cmd_end = .;

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

제가 여기에 대한 전문적인 지식이 없어서리...^^, 하지만 ld manual을 보시면 자세한 것은 나와 있고요. 대략 설명을 드리면요...

1> little endian으로 잡는것(arm이 두가지 모드로 되시는거 아시죠?)

2> ENTRY(_start)는 이 u-boot binary의 시작을 _start란 녀석부터 하겠다는 얘기입니다.
보통 C에서 코드짜듯이 설명드리면 main()이란 얘기랑 비슷합니다.

3> SECTIONS를 이용해서 각 섹션의 위치와 정렬을 지정하게 되는데 .으로 시작하는 어쩌구저쩌구
(ex> .text)가 각 setction에 관한 내용입니다. 이 부분은 코드의 배치를 이렇게 하란 설명입니다.
text section은 한마디로 실행코드란 얘기입니다.

4> . = 0x00000000; 는 이 섹션의 시작이 0번지부터 시작하라는 얘기입니다. 위에서 설명한
ld 옵션인 -Ttext org를 넣게 되면(org는 실제 컴파일시 보면 config.mk의 TEXT_BASE입니다)
실제는 0x0000000이 아니라, TEXT_BASE서부터 map이 잡히게 되겠지요.

5> . = ALIGN(4); 는 4byte align하라는 말이고, text section의 맨앞에 cpu/arm920t/start.o
를 배열하라고 되어 있습니다. 당근 startup코드를 여기다 배열해야 되겠지요?

6> data랑 got section등은 넘어가고요(GOT의 경우는 ppc에서 절라 사용이 많이 됩니다. 뭐
여기서는 별루 사용하는것 같지는 않지만 함 찾아보세요...제가 그냥 넘어가는 것일수도 있으니)

7> __u_boot_cmd_start와 __u_boot_cmd_end사이의 .u_boot_cmd는 u-boot에서 사용되는 user
interface command structure를 이 사이에 배열하라는 얘기이고요.

8> bss section 정의합니다. bss section이란 넘은 간단하게 말씀드리면 C에서 global 변수로
변수들을 선언하시죠? 그 선언한 녀석들을 명시적으로 0으로 해주거나 안해주게 되는데 안해준
변수들을 uninitialized data영역이라고 부릅니다(나만 일케 부르남?...^^) 그 부분을 bss영역
이라고 얘길합니다.(bss의 약자가 뭔지는 옛날엔 알았는데....지금은 몰라용....)
보통 이부분은 C코드가 활성화 되기 전에 0으로 clear해주고 넘어가야 변수 초기화를 안해도
오작동을 안 일으키도록 만들 수 있습니다. 초기화 안하면 쓰레기값일테니깐요.

9> __u_boot_cmd_start, __u_boot_cmd_end, __bss_start, _end
요 녀녀적은 global변수를 선언한거라고 보지면 됩니다. 나중에 C코드, startup코드에서
이것을 가지고 위치계산들을 마니 합니다.

이제 사전작업은 했으니 startup code 분석으로 들어갑니다.
cpu/arm920t/start.S랑
board/smdk2410/memsetup.S요.

제가 이부분을 분석할때는 0.4.0이었는데 벌써 1.1.1이 되었습니다. 코드가 틀린 부분이 있지만
기본개념은 같습니다. 대충 넘어가는 부분이 있더라도 이해하시고요.

실제 코드에 주석달듯이 설명을 해 놔서리...보기가 불편하더라도 쫓아가기는 이게 나을겁니다.
(한글로 된건 제가 다 주석단겁니다. 원 저자가 한글을 몰라서 제가 약간 지원을 했지용...ㅋㅋ)

그리고, 컴파일하면 생겨나는 System.map파일을 참조하시면 좋습니다.

참고로

※ objdump를 이용하시면 disassemble된 걸 보실수 있을겁니다. 그걸 참조하세요.
u-boot의 Makefile에 보시면 해당 명령어도 있읍니다.
make u-boot.dis를 하면 disassemble된 걸 보실수 있습니다. System.map이랑 u-boot.dis를
참조하시면서 보시면 두배 편할겁니다...ㅎㅎ

※ 참, arm에서 쓰는 register의 symbol표기입니다. 쓰이는 이름도 있으니 보는데 좀
편할겁니다.

Assembler Guide 문서를 참조하시면 됩니다.
Variable Register(v1~v8) 이 r4~r11로 맵핑이 되네요
==> ADS(ARM developer suite v1.1)의 Assembly guide, page 3-9
3.3 Predefined registers and coprocessor names 부분을 볼 것
(ghcstop)

ARM에서는 Register들을 알아보기 쉽도록 미리 정의해 놓은 것들이 있습니다.
A1-A4 (R0-R3, Argument)
V1-V8 (R4-R11, Variable)
SB (R9, Stack Base)
SL (R10, Stack Limit)
FP (R11, Frame Pointer)
IP (R12, Inter-procedure-call scratch register)
SP (R13, Stack Point)
LR (R14, Link Register)
PC (R15, Program Counter)

fp는 R11을 의미합니다.

아래는 각 레지스터의 또다른 이름과 용도를 말합니다.
용도를 지키거나 말거나 그런 사용자 맘이죠. 권고 사항입니다.

```
R0 a1 argumect 1 / interger result /scratch register
R1 a2 argumect 2 / scratch register
R2 a3 argumect 3 / scratch register
R3 a4 argumect 4 / scratch register
R4 v1 register variable 1
R5 v2 register variable 2
R6 v3 register variable 3
R7 v4 register variable 4
R8 v5 register variable 5
R9 sb/v6 static base / register variable 6
R10 sl/v7 stack limit / register variable 7
R11 fp frame pointer
r12 ip scratch reg. / new sb in inter-link-unit calls
R13 sp Lower end of current stack frame
R14 lr link address / scratch register
R15 pc program counter

p0-p15(coprocessors 0-15)
c0-c15(coprocessor registers 0-15)


start.S
==========================================================================
#include <config.h>
#include <version.h>


/*
 *************************************************************************
 *
 * Jump vector table as in table 3.1 in [1]
 *
 *************************************************************************
 */


// 코드 start
.globl _start

// 이 부분은 exception vector의 점프루틴이다. 실제적으로는 이것만 해주면 되는거 같더군요...^^
// 즉, ARM은 exception이 걸리면 각 예외에 따라 무조건 정해진 해당 번지로 점프하니깐
// 여기서 reset은 실제 cpu입장에서 보면 0x00000000이고, 두번째넘은 0x00000004...이런식으로
// 벡터맵에 따라 정의해주는 것이다.
_start: b       reset // exception의 처음인 0x00000000번지이다. reset 함수로 분기
        ldr     pc, _undefined_instruction //exception vector 설정, ARM
        ldr     pc, _software_interrupt
        ldr     pc, _prefetch_abort
        ldr     pc, _data_abort
        ldr     pc, _not_used
        ldr     pc, _irq
        ldr     pc, _fiq

// exception vector의 위치 정의 word(4byte)로 정의한것
_undefined_instruction: .word undefined_instruction
_software_interrupt:    .word software_interrupt
_prefetch_abort:        .word prefetch_abort
_data_abort:            .word data_abort
_not_used:              .word not_used
_irq:                   .word irq
_fiq:                   .word fiq

        .balignl 16,0xdeadbeef


/*
 *************************************************************************
 *
 * Startup Code (reset vector)
 *
 * do important init only if we don't start from memory!
 * relocate armboot to ram
```

```
 * setup stack
 * jump to second stage
 *
 - 메모리로부터 부팅을 하지 않을 경우에만 중요한 초기화 작업을 한다(밑의 램 초기화부분 때문에)
 ==> 이건 rom emulator를 사용해서 개발할 때를 얘기하는 것이다.
 include/configs/smdk2410.h의 #define CONFIG_INIT_CRITICAL 1
 부분을 볼 것.

 - armboot를 ram에 relocate한다.
 - stack을 설정한다.
 - second stage로 점프한다.( lib_arm/board.c start_armboot() )

 ************************************************************************
 */

//  - 각 보드에 대한 config.mk, 여기서는 board/smdk2410/config.mk에 선언되어 있다.
//    TEXT_BASE = 0x33F80000
_TEXT_BASE:
        .word   TEXT_BASE

// _start라는 위치로 _armboot_start란 것을 설정, 위의 reset vector위치이다.
.globl _armboot_start
_armboot_start:
        .word _start

/*
 * These are defined in the board-specific linker script.
 이 부분은 linker script쪽을 볼 것 board/smdk2410/u-boot.lds
 */
.globl _bss_start
_bss_start:
        .word __bss_start

.globl _bss_end
_bss_end:
        .word _end

#ifdef CONFIG_USE_IRQ
/* IRQ stack memory (calculated at run-time) */
.globl IRQ_STACK_START
IRQ_STACK_START:
        .word   0x0badc0de // 그냥 나중에 계산된다는 얘기다...

/* IRQ stack memory (calculated at run-time) */
.globl FIQ_STACK_START
FIQ_STACK_START:
        .word 0x0badc0de
#endif


/*
 * the actual reset code
 */

/*
         * set the cpu to SVC32 mode, 현재상태는 previledged 모드이다.user모드에서는 이딴짓 못한다.
         */
        // arm architecture reference manual p.82
        mrs     r0,cpsr     // cpsr을 r0로 읽는다.
        bic     r0,r0,#0x1f // 0x1f를 !해서 r0와 and한 것을 r0로 넣는다.
                            // 즉 r0와 0xffffffe0와 and한 것을 r0로 넣는다.
                            // 하위 5비트의 mode 비트를 clear한다.
        orr     r0,r0,#0xd3 // r0와 0xd3(10011)를 or해서 supervisor모드로 바꾼다.
        msr     cpsr,r0     // r0의 값을 cpsr로 로딩한다.

// 각 cpu 메뉴얼의 레지스터 맵을 보면 된다.
#if defined(CONFIG_S3C2400) // 2400일 경우
        #define pWTCON          0x15300000
        /* Interupt-Controller base addresses */
        #define INTMSK          0x14400008
        /* clock divisor register */
        #define CLKDIVN         0x14800014
#elif defined(CONFIG_S3C2410) // 2410일 경우
        // 매뉴얼 1-32, 18-3을 볼 것, watchdog control register
        #define pWTCON          0x53000000
        /*
```

```
         * Interupt-Controller base addresses
         * 1-26, 14-10, 14-18을 볼 것
        - 메뉴얼 내용
        interrupt mask register: 이 레지스터는 각각의 인터럽트 소스에 연관이 된 32비트
        를 갖고 있다. 만일 이 비트들 중 특정 비트가 1로 설정이 되면, cpu는 이것과 일치하는
        인터럽트 소스로부터의 인터럽트 요청에 대해 서비스 처리를 하지 않는다(참고러 이러한
        경우가 발생하더라도 SRCPND 레지스터의 해당 비트는 1로 설정된다). 만일 mask bit가 0
        이면 인터럽트 요청에 대해 서비스 처리는 이루어진다.

        interrupt submask register: 이 레지스터는 각각의 인터럽트 소스에 연관이 된 11비트
        를 갖고 있다. 만일 이 비트들 중 특정 비트가 1로 설정이 되면, cpu는 이것과 일치하는
        인터럽트 소스로부터의 인터럽트 요청에 대해 서비스 처리를 하지 않는다(참고러 이러한
        경우가 발생하더라도 SUBSRCPND 레지스터의 해당 비트는 1로 설정된다). 만일 mask bit가 0
        이면 인터럽트 요청에 대해 서비스 처리는 이루어진다.
        */

        #define INTMSK          0x4A000008
        #define INTSUBMSK       0x4A00001C
        /*
         *
         메뉴얼 7장의 clock & power management

         앞부분 clock관련 약간의 요약(17-1 page): 2410의 클럭 제어 로직은 cpu를 위한 fclk,
         ahb bus 주변장치들을 위한 hclk와 apb bus 주변장치들을 위한 pclk를 포함한 요구되는 클럭
         신호를 생성할 수 있다.
         2410은 두개의 PLL을 가지고 있는데 하나는 fclk, hclk, pclk를 위한 것이고 다른 녀석은
         usb block(48Mhz)를 위한 넘이다. clock control logic은 PLL없이(말 뜻이 모호한데
         아마도 PLL의 조작없이란 뜻 같음) 클럭들을 느리게 만들 수 있고, 전력소모를 줄이기 위하여
         소프트웨어를 이용해서 각 주변장치 블럭에 대해 clock을 연결하거나 끊을 수 있다.

         7-22 page요약(자세한것은 메뉴얼 볼 것):
         FCLK: ARM920T에서 사용된다.
         HCLK: ARM920T에서 사용되는 AHB bus에서 사용됨, memory controller, interrupt controller,
               LCD controller, DMA, USB host block.
         PCLK: WDT, IIS, I2C, PWM timer, MMC interface, ADC, UART, GPIO, RTC & SPI 와 같은
               주변장치에 의해 사용되는 APB 버스에서 사용된다.
         S3C2410X은 FCLK, HLCK, PCLK사이의 클럭 분할비를 선택할 수 있다.
         이 비율은 CLKDIVN control register의 HDIVN, PDIVN에 의해 결정이 된다.

         clock divisor register: 메뉴얼 1-28, 7-22, FCLK의 clock에 맞춰서 HCLK랑 HCLK에 맞춰
         PCLK를 어케 세팅할것인지   결정하는 넘.
         bit 1: 0이면 hclk는 fclk랑 같고, 아님 hclk = fclk/2
         bit 0: 0이면 pclk는 hclk랑 같고, 아님 pclk = hclk/2
         */
        #define CLKDIVN         0x4C000014
#endif

        // watchdog을 끈다. disable
        ldr     r0, =pWTCON // r0로 pWTCON값을 immediate로 로딩한다.(즉, pWTCON이 들어있는 번지로딩)
        mov     r1, #0x0    // r1에 0을 넣고
        str     r1, [r0]    // r1의 값을 r0가 가리키는 곳의 내용으로 넣는다.
                            // C로 표현하면 *r0 = r1이 된다.

        /*
         * mask all IRQs by setting all bits in the INTMR - default

         - INTMR의 모든 비트를 세팅함으로써 모든 IRQ들을 mask해버린다(무시한다...)

         에구, 여기서는 모든 인터럽트들 source에 대해 mask를 씌우는것 같은데
         명령이 내가 공부한 거랑 틀리다야...arm7강좌에 보면 mov로 0xffffffff란
         값을 immediate를 넣으면 에러가 발생한다고 되어 있는데, 여기서는 그냥 넣네
         arm9은 틀려졌나?...하여간 arm7에서는 mvn r1, #0 로 해야된다고 되어있네용
         mvn은 not move 명령으로 #0를 not해서 r1으로 넣게 되어 있다. 이렇게 되면
         r1에는 -1이 들어간다....이건 arm7강좌.hwp 28 페이지를 볼 것

         근데 여기서는 다이렉트로 넣어버렸네요...^^(compiler에서 처리해주는걸로 알고 있슴다)
         u-boot.dis파일을 참조하세요.(저도 신경써서는 안봤음...ㅎㅎ)
         */
        mov     r1, #0xffffffff
        ldr     r0, =INTMSK // INTMSK가 있는 녀석의 번지로딩
        str     r1, [r0]

#if defined(CONFIG_S3C2410) // 2410일 경우 interrupt submask register를 masking한다.
        ldr     r1, =0x3ff  // 11 bit이므로
        ldr     r0, =INTSUBMSK
        str     r1, [r0]
```

```
#endif

        /*
         * FCLK:HCLK:PCLK = 1:2:4
         bit 1: 0이면 hclk는 fclk랑 같고, 아님 hclk = fclk/2
         bit 0: 0이면 pclk는 hclk랑 같고, 아님 pclk = hclk/2
         여기서 집어넣는 값이 0x00000003이니 당근 1:2:4가 되겠지?
         */
        /* default FCLK is 120 MHz ! */
        ldr     r0, =CLKDIVN
        mov     r1, #3
        str     r1, [r0]

        /*
        /*
         * we do sys-critical inits only at reboot,
         * not when booting from ram!
         리부팅시 시스템에 중요한 초기화 작업을 한다. 램에서 부팅일때는 하지 않는다
         (이 루틴이 ram을 초기화하는 루틴이기 땜에 ram에서 부팅할때 손대면 안좋지롱...^^)
         include/configs/smdk2410.h의 #define CONFIG_INIT_CRITICAL 1
         부분을 볼 것.
         */
#ifdef CONFIG_INIT_CRITICAL // smdk2410.h에 정의되어 있음
        bl      cpu_init_crit // 요 밑에 있는 녀석으로 jump, sdram세팅(dram초기화)
#endif

// cpu_init_crit()에서 부팅을 다 한 후(sdram살리고) 메모리에 모니터 코드들을 relocation을 한다.
relocate:                               /* relocate U-Boot to RAM           */
        // _start의 위치를 pc와의 offset으로 계산한다. arm assemble manual을 보세요.
        // 아니면 kelp에서 ADR로 검색해보면 해당 글이 있을겁니다. 실제로는 r0에 0이 들어갑니다
        adr     r0, _start              /* r0 <- current position of code   */
        // TEXT_BASE: 코드가 relocation될 주소
        ldr     r1, _TEXT_BASE          /* test if we run from flash or RAM */
        // 만일 두녀석이 같은 위치면 relocation할 필요 없지용...이것은 아마도 nand boot
        // 때문에 새로 추가된 코드 같습니다.
        cmp     r0, r1                  /* don't reloc during debug         */
        // relocation할필요 없으니 이거 건너뛰고 바로 stack setup을 한다.
        beq     stack_setup


        // relocation이 필요한 경우
        ldr     r2, _armboot_start     //_armboot_start의 코드상에서의 위치, 실제적으로는
_start이다.
        ldr     r3, _bss_start          // u-boot.lds를 보면 실제 코드의 끝입니다.
        // 두 위치를 빼면 당근 복사해야할 text(코드)의 크기가 나오겠지요
        sub     r2, r3, r2               /* r2 <- size of armboot           */
        add     r2, r0, r2               /* r2 <- source end address, 즉
                                            r0가 시작점이니깐 거기다가 코드의
                                            크기를 더하면 end address가 나온다. */

        /*
         * r0 = source address
         * r1 = target address
         * r2 = source end address
         여기서 하는일은 플래쉬에 있는 코드를 ram영역의 _TEXT_BASE로 복사하는 것이다.
         실제로는 relocation이라고 봐야겠지요.
         즉 r0: _start서부터 pointer를 하나씩 증가시키면서 코드를 r3~r10 레지스터로
         ldmia명령을 이용해서 복사한 후, 이것을 다시 stmia명령을 이용해서 r1: _TEXT_BASE로
         (r1의 번지도 !(write-back)을 이용해서 하나씩 증가시키면서) 복사하는 것이다.

         ldmia: post increment load
         stmia: post increment store
         */
copy_loop:
        ldmia   r0!, {r3-r10}           /* copy from source address [r0]    */
        stmia   r1!, {r3-r10}           /* copy to   target address [r1]    */
        cmp     r0, r2                  /* until source end addreee [r2]    */
        ble     copy_loop

        // 이부분은 옛날과 코드가 틀려졌는데, 대충 TEXT_BASE 밑으로 malloc영역을잡고
        // 그 밑에 bdinfo(아마도 bd_info structure일 것이라는...^^)를 잡고
        // 또 그 밑에 stack을 잡습니다.
        // stack point의 맨위서부터 12byte(3 word)를 reserved하는 것은 abort exception
        // 이 발생했을 경우 exception이 일어나기전의 pc값과 cpse값을 저장하여 debug정보에
        // 사용하기 위해서랍니다...^^
        // 이부분은 자세히 안봐서리....알아서 보세용...^^
```

```
        /* Set up the stack                                             */
stack_setup:
        ldr     r0, _TEXT_BASE          /* upper 128 KiB: relocated uboot   */
        sub     r0, r0, #CFG_MALLOC_LEN /* malloc area                      */
        sub     r0, r0, #CFG_GBL_DATA_SIZE /* bdinfo                        */
#ifdef CONFIG_USE_IRQ
        sub     r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
#endif
        sub     sp, r0, #12             /* leave 3 words for abort-stack    */

       /* bss영역 clear
       add      r0, r0, #4
       코드가 아무리 봐도 이상한 넘인데 이렇게 한다면 bss영역의 처음 변수는 0으로
       되지 않거든요...--;
       구래서 mailing list를 봤더니... 아마도 bug인거 같다는 얘기가 있슴다.

       원문:
Hi folks,

 on the smdk2410 I found out that the BSS segment is initialized at offset 0x4
 and not with offset 0x0, leaving the first entry undefined (in my case
 timer_load_val), at least with gcc-3.3. Cause of pain is
 arm920t/start.S:clear_bss

 clear_bss:
        ldr     r0, _bss_start          /* find start of bss segment        */
        add     r0, r0, #4              /* start at first byte of bss       */
        ldr     r1, _bss_end            /* stop here                        */

 I think the add has to be removed. This also seems to be the case on arm720t
 and arm926ejs.

 Bye,
 --
 Markus Pietrek
 Kueferstrasse 8, D-79206 Breisach, Germany
 FS Forth-Systeme GmbH
 Phone: +49 (7667) 908 145,  FAX +49 (7667) 908 245


<답>
Dear Markus,

 in message <200405041112.56047.maillist@fs...> you wrote:
 >
 > on the smdk2410 I found out that the BSS segment is initialized at offset 0x4
 > and not with offset 0x0, leaving the first entry undefined (in my case
 > timer_load_val), at least with gcc-3.3. Cause of pain is
 > arm920t/start.S:clear_bss

 I think you are right. Thanks for pointing out.

 > clear_bss:
 >      ldr     r0, _bss_start          /* find start of bss segment        */
 >      add     r0, r0, #4              /* start at first byte of bss       */
 >      ldr     r1, _bss_end            /* stop here                        */
 >
 > I think the add has to be removed. This also seems to be the case on arm720t
 > and arm926ejs.

 Actually _all_ ARM  systems  are  affected  -  except  s3c44b0  which
 doesn"t clear BSS at all?


 Andrea, am I missing something, or is the clear_bss code  missing  in
 your port for the s3c44b0 processor?

 Best regards,

 Wolfgang Denk
*/
clear_bss:
        ldr     r0, _bss_start          /* find start of bss segment        */
        add     r0, r0, #4              /* start at first byte of bss       */
        ldr     r1, _bss_end            /* stop here                        */
        mov     r2, #0x00000000         /* clear                            */
```

```
// 0으로 clear
clbss_l:str     r2, [r0]                /* clear loop...                    */
        add     r0, r0, #4
        cmp     r0, r1
        bne     clbss_l

#if 0
        /* try doing this stuff after the relocation */
        ldr     r0, =pWTCON
        mov     r1, #0x0
        str     r1, [r0]

        /*
         * mask all IRQs by setting all bits in the INTMR - default
         */
        mov     r1, #0xffffffff
        ldr     r0, =INTMR
        str     r1, [r0]

        /* FCLK:HCLK:PCLK = 1:2:4 */
        /* default FCLK is 120 MHz ! */
        ldr     r0, =CLKDIVN
        mov     r1, #3
        str     r1, [r0]
        /* END stuff after relocation */
#endif

        ldr     pc, _start_armboot // ==> lib_arm/board.c의 start_armboot()로 jump

_start_armboot: .word start_armboot


/*
 ************************************************************************
 *
 * CPU_init_critical registers
 *
 * setup important registers
 * setup memory timing
 *
 ************************************************************************
 */


cpu_init_crit:
        /*
         여기서 p15는 CP15(coprocessor 15)를 말하고, c0~c15은 해당 coprocessor의
         register를 말하는 것이다.

         Manual Appendix A2 Programmer's Model page 2-1의 내용:
         arm920t는 두개의 코프로세서를 포함하고 있다.
         - CP14: 디버그 통신채널에의 sw적인 접근을 허용하는, MCR과 MRC 명령을 사용한다.
         - CP15: system control processor. cache, MMU, protection system, clocking mode,
                 big 혹은 little endian operation과 같은 arm920t의 다른 시스템 옵션들을
                 설정하고 제어하는데 사용된다. MCR과 MRC 명령으로 접근할 수 있고, 해당
                 레지스터 맵은 page 2-4의 summary에서 볼 수 있다.

         coprocessor에 대한 자세한 내용은 S3C2410 메뉴얼 Programmer's Model part와
         이것에 대한 번역본인 S3C2800 programmer's model의 2-18 page서부터 참조할 것
         */

        /*
         * flush v4 I/D caches

         여기 사용되는 명령들은 메뉴얼에 아예 있다. 자기가 만들어서 쓰는게 아닌것 같음...^^
         mcr: register to coprocessor
         mrc: coprocessor to register
         */
        mov     r0, #0 // 밑의 명령을 수행하기 위해서는 반드시 이게 0으로 되어 있어야 한다.
        mcr     p15, 0, r0, c7, c7, 0   /* flush v3/v4 cache: ICache와 DCache 무효화 */
        mcr     p15, 0, r0, c8, c7, 0   /* flush v4 TLB: TLB(Translation Lookaside
Buffer-MMU관련캐쉬) 무효화 */

        /*
         * disable MMU stuff and caches
```

```
         */
        mrc     p15, 0, r0, c1, c0, 0   // control register로부터 r0로 값을 읽어온다.

        /*
         0x00002300 = 0000 0000 0000 0000 0010 0011 0000 0000 이것을 bic의 값으로 넣는다면
         이것에의 !값이 and가 되니깐 해당 비트만 clear되는 것이다.
         control register의 V, S, R 비트를 clear한다.
         V: 예외 레지스터들의 베이스 location을 0x0000000으로 맞춘다.
         R, S: MMU세팅에서의 Domain Access Control 부분을 봐야한다, Appendix page 3-20의 테이블
               3-6을 볼 것. 아니면 S3C2800 메뉴얼 번역본 MMU부분을 볼 것.
               이 값과 Domain Access Control register의 permission에 따라 설정이 틀려진다.
         */
        bic     r0, r0, #0x00002300     @ clear bits 13, 9:8 (--V- --RS)

        /*
         B: Little-endian operation
         CAM: Cache disable, Alignment Fault checking disable, MMU disable
         */
        bic     r0, r0, #0x00000087     @ clear bits 7, 2:0 (B--- -CAM)

        /* alignment fault checking enable, Instruction Cache enable */
        orr     r0, r0, #0x00000002     @ set bit 2 (A) Align
        orr     r0, r0, #0x00001000     @ set bit 12 (I) I-Cache

        /* write control register */
        mcr     p15, 0, r0, c1, c0, 0


        /*
         * before relocating, we have to setup RAM timing
         * because memory timing is board-dependend, you will
         * find a memsetup.S in your board directory.
         relacation전에 RAM timing을 설정해야 한다. 왜냐하면 memory timing은 board에
         의존적이기 때문이다(보드따라 메모리가 틀리다는 얘기지롱), 해당 보드 디렉토리에 보면
         memsetup.S라는 넘이 있으니 이걸 볼 것.
         */

        /*
         ip는 r12를 얘기한다. IP (R12, Inter-procedure-call scratch register)
         보통 사용하는 것은 asm에서 함수 호출을 하고, 다시 그함수에서 다른 호출을 해서
         호출이 nesting될 경우 첫번째 호출이야 lr을 써서 return할 pc를 저장하나
         거기서 다시 호출을 할 경우는 그 return 위치를 stack에 저장해야 한다.
         근데 여기서 보면 속도 문제도 있지만 stack을 현재 사용하지 못하는 상태이기 때문에
         (왜냐 아직 ram이 초기화가 안되어 있잖냐) ip(r12)에다가 처음 호출했던 코드의 리턴할
         주소인 lr을 저장하고 함수를 호출하면 리턴할 pc가 lr로 저장이 된다(만일 lr을 stack(여기서는
         없으니깐 안되지만)이나 다른 레지스터에 저장을 하지 않음 덮어씌우기 되어서 난리난다).
         그리고 nesting된 함수에서 mov pc,lr로 리턴하고 리턴된 함수(cpu_init_crit)에서는
         저장되었던 lr을 복귀시키고 이것을 가지고 return을 시킨다.

         reset -> cpu_init_crit -> memsetup
         */
        mov     ip, lr
        bl      memsetup //board/smdk2410/memsetup.S로 점프
        mov     lr, ip

        mov     pc, lr // return

/*
 *************************************************************************
 *
 * Interrupt handling
 *
 *************************************************************************
 */

@
@ IRQ stack frame.
@
#define S_FRAME_SIZE    72

#define S_OLD_R0        68
#define S_PSR           64
#define S_PC            60
#define S_LR            56
#define S_SP            52
```

```
#define S_IP            48
#define S_FP            44
#define S_R10           40
#define S_R9            36
#define S_R8            32
#define S_R7            28
#define S_R6            24
#define S_R5            20
#define S_R4            16
#define S_R3            12
#define S_R2            8
#define S_R1            4
#define S_R0            0

#define MODE_SVC 0x13
#define I_BIT    0x80

/*
 * use bad_save_user_regs for abort/prefetch/undef/swi ...
 * use irq_save_user_regs / irq_restore_user_regs for IRQ/FIQ handling
 */

        .macro  bad_save_user_regs
        sub     sp, sp, #S_FRAME_SIZE
        stmia   sp, {r0 - r12}                  @ Calling r0-r12
        ldr     r2, _armboot_start
        sub     r2, r2, #(CONFIG_STACKSIZE+CFG_MALLOC_LEN)
        sub     r2, r2, #(CFG_GBL_DATA_SIZE+8)  @ set base 2 words into abort stack
        ldmia   r2, {r2 - r3}                   @ get pc, cpsr
        add     r0, sp, #S_FRAME_SIZE           @ restore sp_SVC

        add     r5, sp, #S_SP
        mov     r1, lr
        stmia   r5, {r0 - r3}                   @ save sp_SVC, lr_SVC, pc, cpsr
        mov     r0, sp
        .endm

        .macro  irq_save_user_regs
        sub     sp, sp, #S_FRAME_SIZE
        stmia   sp, {r0 - r12}                  @ Calling r0-r12
        add     r8, sp, #S_PC
        stmdb   r8, {sp, lr}^                   @ Calling SP, LR
        str     lr, [r8, #0]                    @ Save calling PC
        mrs     r6, spsr
        str     r6, [r8, #4]                    @ Save CPSR
        str     r0, [r8, #8]                    @ Save OLD_R0
        mov     r0, sp
        .endm

        .macro  irq_restore_user_regs
        ldmia   sp, {r0 - lr}^                  @ Calling r0 - lr
        mov     r0, r0
        ldr     lr, [sp, #S_PC]                 @ Get PC
        add     sp, sp, #S_FRAME_SIZE
        subs    pc, lr, #4                      @ return & move spsr_svc into cpsr
        .endm

        .macro get_bad_stack
        ldr     r13, _armboot_start             @ setup our mode stack
        sub     r13, r13, #(CONFIG_STACKSIZE+CFG_MALLOC_LEN)
        sub     r13, r13, #(CFG_GBL_DATA_SIZE+8) @ reserved a couple spots in abort stack

        str     lr, [r13]                       @ save caller lr / spsr
        mrs     lr, spsr
        str     lr, [r13, #4]

        mov     r13, #MODE_SVC                  @ prepare SVC-Mode
        @ msr   spsr_c, r13
        msr     spsr, r13
        mov     lr, pc
        movs    pc, lr
        .endm

        .macro get_irq_stack                    @ setup IRQ stack
        ldr     sp, IRQ_STACK_START
        .endm
```

```
        .macro get_fiq_stack                    @ setup FIQ stack
        ldr     sp, FIQ_STACK_START
        .endm

/*
 * exception handlers
 */
        .align  5
undefined_instruction:
        get_bad_stack
        bad_save_user_regs
        bl      do_undefined_instruction

        .align  5
software_interrupt:
        get_bad_stack
        bad_save_user_regs
        bl      do_software_interrupt

        .align  5
prefetch_abort:
        get_bad_stack
        bad_save_user_regs
        bl      do_prefetch_abort

        .align  5
data_abort:
        get_bad_stack
        bad_save_user_regs
        bl      do_data_abort

        .align  5
not_used:
        get_bad_stack
        bad_save_user_regs
        bl      do_not_used

#ifdef CONFIG_USE_IRQ

        .align  5
irq:
        get_irq_stack
        irq_save_user_regs
        bl      do_irq
        irq_restore_user_regs

        .align  5
fiq:
        get_fiq_stack
        /* someone ought to write a more effiction fiq_save_user_regs */
        irq_save_user_regs
        bl      do_fiq
        irq_restore_user_regs

#else

        .align  5
irq:
        get_bad_stack
        bad_save_user_regs
        bl      do_irq

        .align  5
fiq:
        get_bad_stack
        bad_save_user_regs
        bl      do_fiq

#endif

        .align  5
.globl reset_cpu
reset_cpu:
#ifdef CONFIG_S3C2400
        bl      disable_interrupts
```

```
# ifdef CONFIG_TRAB
        bl      disable_vfd
# endif
        ldr     r1, _rWTCON
        ldr     r2, _rWTCNT
        /* Disable watchdog */
        mov     r3, #0x0000
        str     r3, [r1]
        /* Initialize watchdog timer count register */
        mov     r3, #0x0001
        str     r3, [r2]
        /* Enable watchdog timer; assert reset at timer timeout */
        mov     r3, #0x0021
        str     r3, [r1]
_loop_forever:
        b       _loop_forever
_rWTCON:
        .word   0x15300000
_rWTCNT:
        .word   0x15300008
#else /* ! CONFIG_S3C2400 */
        mov     ip, #0
        mcr     p15, 0, ip, c7, c7, 0           @ invalidate cache
        mcr     p15, 0, ip, c8, c7, 0           @ flush TLB (v4)
        mrc     p15, 0, ip, c1, c0, 0           @ get ctrl register
        bic     ip, ip, #0x000f                 @ ............wcam
        bic     ip, ip, #0x2100                 @ ..v....s........
        mcr     p15, 0, ip, c1, c0, 0           @ ctrl register
        mov     pc, r0 /* 이건 뭐냐, 혹시 reset번지 아닌감?, 번지 같다.
                        C에서 asm call할때 r0로 인자가 넘어오는지 검사할 것
                        : 0번지로 뛰라는 얘기다...즉, reset vector번지 */
#endif /* CONFIG_S3C2400 */
```

==========

이제는 memsetup.S의 dram이랑 CS쪽을 초기화 하는 루틴입니다. 글을 쓰다보니
넘 힘들어서 이 부분은 0.4.0때 썼던거를 그대로 복사할겁니다.

틀린 부분도 있을지 모르지만 성질참고 욕하지 마시고....직접 고쳐서 보세용...ㅎㅎ

memsetup.S
==========

```
#include <config.h>
#include <version.h>


/* some parameters for the board */

/*
 *
 * Taken from linux/arch/arm/boot/compressed/head-s3c2410.S
 *
 * Copyright (C) 2002 Samsung Electronics SW.LEE  <hitchcar@sec.samsung.com>
 *
 */

#define BWSCON  0x48000000

/* BWSCON */
#define DW8                     (0x0)
#define DW16                    (0x1)
#define DW32                    (0x2)
#define WAIT                    (0x1<<2)
#define UBLB                    (0x1<<3)

#define B1_BWSCON               (DW32)
#define B2_BWSCON               (DW16)
#define B3_BWSCON               (DW16 + WAIT + UBLB)
#define B4_BWSCON               (DW16)
#define B5_BWSCON               (DW16)
#define B6_BWSCON               (DW32)
#define B7_BWSCON               (DW32)

/* BANK0CON */
#define B0_Tacs                 0x0     /*  0clk */
#define B0_Tcos                 0x0     /*  0clk */
```

```
#define B0_Tacc                 0x7     /* 14clk */
#define B0_Tcoh                 0x0     /*  0clk */
#define B0_Tah                  0x0     /*  0clk */
#define B0_Tacp                 0x0
#define B0_PMC                  0x0     /* normal */

/* BANK1CON */
#define B1_Tacs                 0x0     /*  0clk */
#define B1_Tcos                 0x0     /*  0clk */
#define B1_Tacc                 0x7     /* 14clk */
#define B1_Tcoh                 0x0     /*  0clk */
#define B1_Tah                  0x0     /*  0clk */
#define B1_Tacp                 0x0
#define B1_PMC                  0x0

#define B2_Tacs                 0x0
#define B2_Tcos                 0x0
#define B2_Tacc                 0x7
#define B2_Tcoh                 0x0
#define B2_Tah                  0x0
#define B2_Tacp                 0x0
#define B2_PMC                  0x0

#define B3_Tacs                 0x0     /*  0clk */
#define B3_Tcos                 0x3     /*  4clk */
#define B3_Tacc                 0x7     /* 14clk */
#define B3_Tcoh                 0x1     /*  1clk */
#define B3_Tah                  0x0     /*  0clk */
#define B3_Tacp                 0x3     /*  6clk */
#define B3_PMC                  0x0     /* normal */

#define B4_Tacs                 0x0     /*  0clk */
#define B4_Tcos                 0x0     /*  0clk */
#define B4_Tacc                 0x7     /* 14clk */
#define B4_Tcoh                 0x0     /*  0clk */
#define B4_Tah                  0x0     /*  0clk */
#define B4_Tacp                 0x0
#define B4_PMC                  0x0     /* normal */

#define B5_Tacs                 0x0     /*  0clk */
#define B5_Tcos                 0x0     /*  0clk */
#define B5_Tacc                 0x7     /* 14clk */
#define B5_Tcoh                 0x0     /*  0clk */
#define B5_Tah                  0x0     /*  0clk */
#define B5_Tacp                 0x0
#define B5_PMC                  0x0     /* normal */

/* bank control register setting for sdram ====================================*/
/* 삼성: K4S561632C-TC75 , cpu manual 5-16과 SDRAM manual의 Operating AC Parameter부분을 볼 것 */
#define B6_MT                   0x3     /* SDRAM */
/*
 3clock = TC75일 경우 RAS to CAS delay가 20ns인데 이것을 clock으로 계산하면
          동작 clock이 133MHz일 경우 1hz가 7.5ns정도니깐 20/7.5 = 2.6...를
          반올림 하면 3clock이 된다.
 */
#define B6_Trcd                 0x1
#define B6_SCAN                 0x1     /* 9bit: culumn address number(13 x 9 짜리니깐 9다) */

#define B7_MT                   0x3     /* SDRAM */
#define B7_Trcd                 0x1     /* 3clk */
#define B7_SCAN                 0x1     /* 9bit */


/* refresh control register setting ===================================*/
/* REFRESH parameter: 요건 cpu메뉴얼 5-17 */
#define REFEN                   0x1     /* Refresh enable */
#define TREFMD                  0x0     /* CBR(CAS before RAS)/Auto refresh */
/*
※ 고쳐야할지도 모르는 부분
B6_Trcd 값이 0x1니깐 3clock이거든요 sdram(k4s561632c-tc75)
메듀얼을 보니 20ns더군요. 밑에 나와있는대로 133MHz로 동작시켰을때 CAS Latency가 3일때
1hz가 7.51ns가 나오니깐 20을 나누기해서 반올림하면 3clock이 나오는데

그래서 refresh쪽의 Trp(row precharge time)도 같은 20ns니까.
3 clock이 되어야 할것 같은데 2 clock이네요.
이것에 따라서 Row Cycle Time(Trc) = Tsrc + Trp니깐 Row Cycle Time는 위의 숫자대로
```

하면 Tsrc가 7이고 Trp가 2니깐 당연히 9가 나오는것은 맞습니다.

그런데 Row Cycle Time를 메뉴얼에서 찾아보니 65 ns더군요.
9clock이 맞습니다(65/7.51에 반올림).

그래서

Trp(row precharge time)가 3이 나와야되고, Trp가 3이면 Tsrc가 6clock
이 되야 하지 않나 생각이 드는데 어떻게 된것인지 찰 모르겠습니다.
*/
#define Trp   0x0   /* 2clk */
#define Tsrc   0x3   /* 7clk */

/* ※ 고쳐야할지도 모르는 부분 */
#define Tchr   0x2   /* 3clk: 이건 2410에서 reserved된 값이니 없어도 될거 같음
*/
/*
※ 고쳐야할지도 모르는 부분
K4S561632C-TC75(256MBit = 32MByte)의 경우는 64ms / 8K 니깐 7.8 us인 것 같다.
여기의 REFCNT 계산이 틀려져야 할것 같은 느낌이 팍팍온다. 즉 period를 7.8us로 계산해야 하는거 아닌감?
refresh rate 15.6 us (= 64 ms / 4K = 62.4 / quad bursts) for <= 128 MBit
refresh rate 7.8 us (= 64 ms / 8K = 31.2 / quad bursts) for 256 MBit
*/
#define REFCNT   1113   /* period=15.6us, HCLK=60Mhz, (2048+1-15.6*60) */
/************************************/


```
_TEXT_BASE:
        .word   TEXT_BASE

.globl memsetup
memsetup:
        /* memory control configuration */
        /* make r0 relative the current location so that it */
        /* reads SMRDATA out of FLASH rather than memory !, 메모리가 아니라 flash로부터 읽어온다?
*/
        // 여기서 읽어오는 주소는 실제 메모리안에 로딩이 되었을때의 주소이다.
        ldr     r0, =SMRDATA    // 주소 로딩
        ldr     r1, _TEXT_BASE  // text base의 주소를 r1에 로딩

        /* 아아...relocation전이므로, flash에서 읽어오므로 r0의 값은 0번지부터
           읽어오는 것으로 생각하면된다. 즉, 위의 코드인 ldr은 텍스트 베이스로부터 시작된 코드들중
           literal pool인 SMRDATA의 텍스트에서의 offset을 가져오기 위한 로딩이다.
           실제로는 이 로딩된 번지는 사용되지 않는다(offset계산 이외에는)

           밑의 sub계산을 하면 r0에 smrdata의 offset만 들어가게 되는데 밑의 loop를 보면
           offset번지에서 읽어다가 해당 레지스터로 바로 때려댄다. 이게 말이 되는가? 실제 SMRDATA가
           있는 번지가 아닌 0번지서부터의 offset을 읽어다가 넣어도?
           말이된다. 실제 코드가 flash에 있을때는 무조건 그 시작번지가 0이 되기 때문이다.
           원래 메모리 맵에는 틀리게 잡혀있을지라도(이건 ppc일 때는 0번지로 지정 안된다.
           설정하기 나름이다.하지만 arm일때는 0번지로 고정되어 있기는 하네...arm이 이런건 낫다...ㅋㅋㅋ)
           보드를 켜면 CS0를 억세스 하므로 거기 있는 코드를 무조건 0번지라고 인식을 하고
           들어갈 것이다.
           그러므로, 여기서는 r0에다가 offset만 넣어서 그걸 가지고 연산을 해도 된다.
           하여간, 이해 못하면 flash는 0번지서부터고 text base를 쓰는 것은 나중에 relocation한거서
           부터다....

           첨부되어 있는 memsetup.S.ppt 파일을 볼 것 - 넘 그지 같애서 거따 그림 그려놨음
           */

        sub     r0, r0, r1      // 두개의 차이만큼 r0에 저장, 즉, base로부터 literal pool의
offset을 구한다.
        ldr     r1, =BWSCON     /* Bus Width Status Controller의 주소를 r1으로 로딩 */
        add     r2, r0, #13*4   /* r0의 주소에다 13*4 만큼 더한 값을 r2에다 세팅해서 loop의 끝지점을
지정한다 */
0:
        ldr     r3, [r0], #4 /* post indexed addressing mode, r0가 가리키는 곳의 내용을 r3로 넣고,
r0의 주소를 4만큼 증가시킨다. r3 = *r0; r0 += 4;*/
        str     r3, [r1], #4 /* r3의 내용을 r1(bus width & wait control register)의 내용으로
집어넣는다. 그리고, r1의 주소를 4만큼 증가, 즉, 다음레지스터의 위치로, r1 += 4 */
        cmp     r2, r0       /* r2(loop의 end)랑 r0(변수)가 같은지 비교 */
        bne     0b           /* 그 비교값이 0이 아니면 label 0으로 jump, 라벨좀 바꾸지...원 */

        /* everything is fine now */
```

```
        mov     pc, lr

/* 이걸 함 C코드로 짜볼까?

uint offset = SMRDATA - _TEXT_BASE;
uint *write_pointer = BWSCON;
uint *read_pointer = 0 + offset; // 0번지 + offset서부터
int i=0;

// 원래한다면 while문인데 for문이 편해서리...^^
for( i=0; i<13; i++)
{
   *write_pointer = *read_pointer;
}

*/



/*
 ltorig: directive로서 이전에 생성된 literal(constant선언시 label을 부여하는 과정을 생략한 것,
         constant의 연산으로 이루어질 수 있음)의 pool을 만들라는 것(지시자)

 literal pool:
     - assembler collects all the literals into one or more literal pools
     - default location is at the end of the program
            for better code reading
     - programmer can declare a place (LTORG)
            to use PC-relative addressing
            to keep data close to instruction

 literal:
   ● constant에 의한 연산 명령
     . constant를 선언하고 label을 부여
     . instruction에서는 label 값으로 연산을 하여야 함
          → label을 부여하는 과정을 생략하는 기능 → literal
     . prefixed with " = '
   ● literal과 immediate operand의 차이를 이해하는 것이 중요하다.
     LDA =X'05'
     LDA #5

  . LTORG directive 다음에
    이전에 선언된 literal에 대한 pool이 생성됨
        → programmer의 명령어에 의해 literal의 상대 주소를 조정 가능


 The literal pool is used to store the field names pointed to by each field node
 and the constant values pointed to by each constant node.

각 레지스터의 이름
SMRDATA:
    .word bus width & wait control register
    .word bank control register 0
    .word bank control register 1
    .word bank control register 2
    .word bank control register 3
    .word bank control register 4
    .word bank control register 5
    .word bank control register 6
    .word bank control register 7
    .word Refresh control register
    .word bank size register
    .word 0x30
    .word 0x30
 */



        .ltorg
/* the literal pools origin */
SMRDATA:
    .word
(0+(B1_BWSCON<<4)+(B2_BWSCON<<8)+(B3_BWSCON<<12)+(B4_BWSCON<<16)+(B5_BWSCON<<20)+(B6_BWSCON<<
24)+(B7_BWSCON<<28))
    .word
((B0_Tacs<<13)+(B0_Tcos<<11)+(B0_Tacc<<8)+(B0_Tcoh<<6)+(B0_Tah<<4)+(B0_Tacp<<2)+(B0_PMC))
```

```
    .word
((B1_Tacs<<13)+(B1_Tcos<<11)+(B1_Tacc<<8)+(B1_Tcoh<<6)+(B1_Tah<<4)+(B1_Tacp<<2)+(B1_PMC))
    .word
((B2_Tacs<<13)+(B2_Tcos<<11)+(B2_Tacc<<8)+(B2_Tcoh<<6)+(B2_Tah<<4)+(B2_Tacp<<2)+(B2_PMC))
    .word
((B3_Tacs<<13)+(B3_Tcos<<11)+(B3_Tacc<<8)+(B3_Tcoh<<6)+(B3_Tah<<4)+(B3_Tacp<<2)+(B3_PMC))
    .word
((B4_Tacs<<13)+(B4_Tcos<<11)+(B4_Tacc<<8)+(B4_Tcoh<<6)+(B4_Tah<<4)+(B4_Tacp<<2)+(B4_PMC))
    .word
((B5_Tacs<<13)+(B5_Tcos<<11)+(B5_Tacc<<8)+(B5_Tcoh<<6)+(B5_Tah<<4)+(B5_Tacp<<2)+(B5_PMC))
    .word ((B6_MT<<15)+(B6_Trcd<<2)+(B6_SCAN))
    .word ((B7_MT<<15)+(B7_Trcd<<2)+(B7_SCAN))
    .word ((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Tsrc<<18)+(Tchr<<16)+REFCNT)
    .word 0x32
    .word 0x30
    .word 0x30
=================================================================================
```

자 그럼 startup코드는 이쯤에서 끝난걸로 하고요.

다음서부터는 실제 main()인 start_armboot()입니다.

u-boot-1.1.1로 가면서 코드들이 조금씩 바뀌는 통에 옛날걸 갖색하려고 해도 절라 신경쓰여서
좀 템포를 늦춰야하지 않을까 하는 생각이 듭니다.

그럼...수고들 하세요.