1.1. Boot Loader(BLOB)

Boot loader가 하는 일을 보드의 초기화와 저장 매체로부터(혹은 serial이나 ethernet을 통해서) 커널을 읽어서 메모리의 적절한 위치로 적재(load)한 다음, 커널로 제어를 옮기는 역할을 수행한다. 또한 개발중인 커널의 경우에는 네트워크를 통한 커널 image의 다운로드도 제공하며, 현재로서는 일반적으로 serial을 통한 커널의 다운로딩을 제공하는 것들이 많다. 각각의 CPU architecture마다 boot loader가 각기 존재하기 때문에, 커널의 개발자는 자신이 사용하는 CPU의 architecture가 어떤 것인가에 따라서, 선택할 수 있는 boot loader의 종류를 알 수 있을 것이다. 이것도 못마땅하다면, 직접 개발해서 사용할 수도 있지만, 개발기간이 좀더 추가되어야 할 것이다.

우리가 이곳에서 보게될 boot loader로는 blob이며, 각각의 target마다 적절한 boot loader를 선택해 주거나, 혹은 스스로 개발해서 사용해야 할 것이다. ARM에서 사용하는 boot loader의 대표적인 예로 blob를 택했으며, 관련된 site로는 http://www.lart.tudelft.nl/lartware/blob [1]를 보기 바란다. 이하에서는 blob를 기본 boot loader로 사용해서 설명하는 내용이다.

먼저, 본격적으로 source 코드를 보면서 진행하기에 앞서 전체적인 BLOB의 진행 절차에 대해서 간단히 알아보도록 하자.

---

[1] 이 site에서는 ARM toolchain 및 그것들에 대한 patch와 Linux 커널에 대한 patch도 찾을 수 있을 것이다. Cross compiler를 직접 compile하기 싫은 사람들을 위한 binary 파일도 제공하므로, 그냥 가져다가 쓰면 될 것이다. 이 글을 쓰고 있는 현재 BLOB는 2.0.3 버전을 찾을 수 있었기에, 이하의 코드는 이것을 기준으로 하겠다.

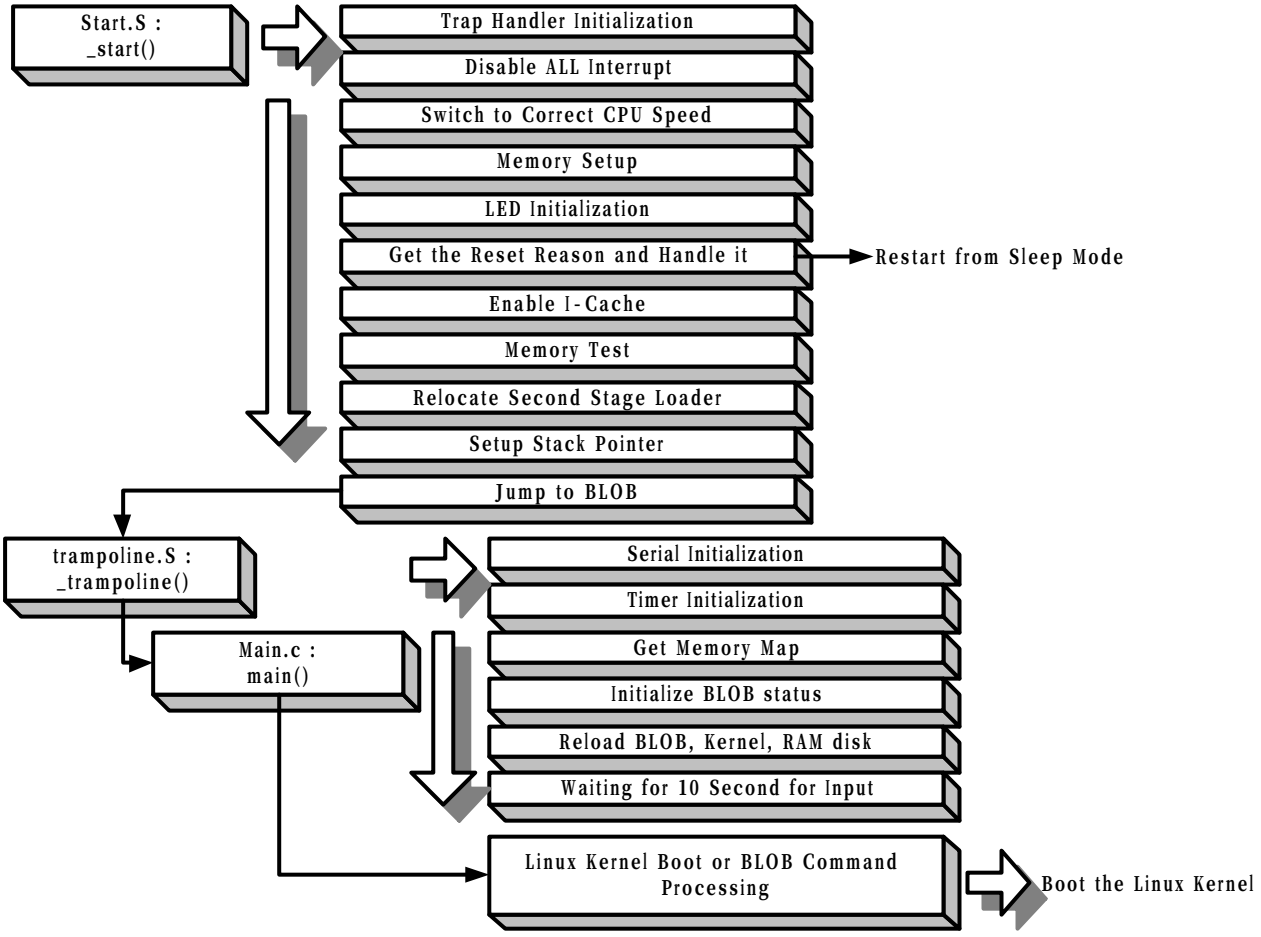

그림 1. BLOB의 실행 순서

[그림 1]은 BLOB의 실행 순서 별로 추적하면서 정리한 것이다. 큰 분류별로 본 것으로 최종적으로 Linux Kernel을 booting하는 것을 목적으로 한다. 먼저 시작하는 곳은 start.S파일에서 시작하게 되고, trampoline.S를 통해서 BLOB의 main() 함수가 정의된 main.c를 실행하게 된다.

이젠 이러한 구조를 가진 BLOB를 앞에서 보여준 순서에 의해서 차례로 설명을 진행하도록 하겠다. 그리고, BLOB의 설명이 끝나면, SA1100 Linux Kernel의 booting으로 넘어가서 실제 Linux 커널의 boot sequence를 추적해 보겠다.

1.1.1. start.S의 분석

먼저, blob의 source tree를 보면, tools와 src, include를 디렉토리를 찾을 수 있을 것이다. 각각은 사용하는 tool 및 utility에 대한 디렉토리와, blob의 source, 그리고, head 파일을 가지고 있다. 여기서는 compile과 설치는 설명하지 않기로하고, 일단 먼저 start-ld-script

라는 파일을 보도록 하자. 이것은 LD(Loader & Linker)의 input으로 주어져서, object 파일을 생성하는데 규칙을 제공할 것이다. 아래와 같은 부분이 있을 것이다.

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
        . = 0x00000000;
        . = ALIGN(4);
        .text : { *(.text) }
        . = ALIGN(4);
        .rodata : { *(.rodata) }
        . = ALIGN(4);
        .data : { *(.data) }
        . = ALIGN(4);
        .got : { *(.got) }
        . = ALIGN(4);
        .bss : { *(.bss) }
}
```

코드 1. start-ld-script 파일

파일에서 OUTPUT_FORMAT은 ELF32의 little endian으로 코드를 생성하겠다는 말이며, OUTPUT_ARCH는 이 binary를 실행할 수 있는 CPU architecture로 ARM을 주겠다는 뜻이다. Entry point가 되는 함수는 _start이다. 또한 SECTIONS의 정의를 보면, text, rodata, data, got, bss라는 section들이 정의되어 있음을 볼 수 있을 것이다. 이중에서 실행할 코드가 들어가는 부분은 text이다. "."은 현재의 address를 가르키는 일종의 포인터로 생각하면 될 것이다. 따라서, 코드는 0x00000000에서 시작해서 4 byte단위로 정렬된 text section이 제일 먼저 나올 것이다.
이젠 blob가 어디서부터 실행될 지를 알게되었다(_start). _start는 start.S에 다시 아래와 같이 정의되어 있으며, 이곳에서부터 설명을 진행하도록 하겠다.

```
.text

/* Jump vector table as in table 3.1 in [1] */
.globl _start
```

```
_start:  b       reset
         b       undefined_instruction
         b       software_interrupt
         b       prefetch_abort
         b       data_abort
         b       not_used
         b       irq
         b       fiq
```

코드 2. start.S 파일

.text로 이곳이 text section에 속함을 알 수 있다. 또한 _start가 .global로 정의되어 export된 symbol임을 짐작할 수 있으며, 앞에서 _start가 entry point가 된다는 것을 알았다. _start의 첫부분은 reset으로 branch하는 연산이다. 따라서, reset으로 제어가 옮겨질 것이다. 나머지는 ARM에서 발생할 수 있는 exception 상황에 대한 처리 routine들로 되어 있으며, 각각은 undefined instruction, software interrupt, prefetch abort, data abort, irq, fiq의 핸들러를 가르킨다.

```
/* the actual reset code */
reset:
         /* First, mask **ALL** interrupts */
         ldr     r0, IC_BASE
         mov     r1, #0x00
         str     r1, [r0, #ICMR]

         /* switch CPU to correct speed */
         ldr     r0, PWR_BASE
         LDR     r1, cpuspeed
         str     r1, [r0, #PPCR]
```

코드 3. start.S 파일(계속)

IC_BASE(=0x90050000)는 interrupt controller register의 기본(base) 주소를 가진다. 이 값을 r0에 가져오고, r1에는 0을 둔다. 다시 r0가 가르키는 값에 ICMR의 offset(=0x04)을 더해서, 이곳에 r1(=0)을 넣는다. 이것으로 모든 interrupt를 masking해서 interrupt가 발생하지 않도록 만든다.

PWR_BASE(=0x90020000)는 power manager register의 기본 기본 주소를 가진다. r0에 PWR_BASE를 가져오고, r1에는 cpuspeed 값을 가져온후 r0에 PPCR(=0x14)을 더한 주소

에 r1을 넣는다. 이것으로 CPU의 clock speed를 조정해준다. 이때 사용되는 cpuspeed의 값은 아래와 같이 정의된다.

```
/* The initial CPU speed. Note that the SA11x0 CPUs can be safely overclocked:
 * 190 MHz CPUs are able to run at 221 MHz, 133 MHz CPUs can do 206 Mhz.
 */
#if (defined ASSABET) || (defined CLART) || (defined LART)  \
        || (defined NESA) || (defined NESA)
cpuspeed:       .long   0x0b    /* 221 MHz */
#elif defined SHANNON
cpuspeed:       .long   0x09    /* 191.7 MHz */
#else
#warning "FIXME: Include code to use the correct clock speed for your board"
cpuspeed:       .long   0x05    /* safe 133 MHz speed */
#endif
```

코드 4. start.S 파일(계속)

즉, ASSABET과 CLART, LART, NESA가 정의된 cpuspeed가 0x0B를 사용하며, SHANNON이 정의된 경우에는 0x09를, 나머지는 0x05로 둔다. 이곳에서 정의된 상수값들은 실제로 PPL(Phase-Locked Loop) 설정하는 PPCR fpwltmxjdml gkdnl 5 bit(CCF<0..5> : Core Clock Configuration Field)에 반영되는 값이다. 이 레지스터에 정의된 값은 reset후에는 남아있지 않기때문에 이곳에서 새로 설정해 주게된다. 상수값이 의미하는 바는 CPU 메뉴얼을 찾아보기 바란다. 각각의 값에 따라서, 3.6864 MHz의 crystal oscillator를 설정하는 것으로, 실제 211.2 MHz, 191.7 MHz, 132.7 MHz라는 동작 속도를 정해준다.[2]

```
        /* setup memory */
        bl      memsetup

        /* init LED */
        bl      ledinit
```

코드 5. start.S 파일(계속)

앞에서 이미 interrupt를 masking했고, CPU의 동작 speed를 결정해 주었다. 이젠 초기

[2] 133MHz는 대략적인 속도를 나타낸 값이다.

booting에서 사용할 메모리를 설정하는 부분이다. BL(Branch with Link)를 사용해서 subroutine으로 memsetup()이라는 함수를 호출한다. 이것을 마치고나면, 다시 ledinit()라는 함수를 호출해서 LED(Light Emitted Device)를 초기화한다. 각각의 함수는 memsetup.S와 ledasm.S에 있으며, 이 파일을 분석하고 계속 진행하도록 하겠다.

```
.globl memsetup
memsetup:
#if defined USE_SA1100
        /* Setup the flash memory */
        ldr     r0, MEM_BASE
        ldr     r1, mcs0
        str     r1, [r0, #MCS0]
        /* Set up the DRAM */
        /* MDCAS0 */
        ldr     r1, mdcas0
        str     r1, [r0, #MDCAS0]
        /* MDCAS1 */
        ldr     r1, mdcas1
        str     r1, [r0, #MDCAS1]
        /* MDCAS2 */
        ldr     r1, mdcas2
        str     r1, [r0, #MDCAS2]
        /* MDCNFG */
        ldr     r1, mdcnfg
        str     r1, [r0, #MDCNFG]
```

코드 6. memsetup.S 파일

memsetup.S 파일에서 하는 일은 초기 메모리를 설정하는 일이다. 프로그램은 주석이 잘 처리가 되어있는 관계로 보기가 편할 것이다. 먼저 SA1100 board를 사용하고 있다면, #if defined USE_SA1100이하에서 실행이 시작될 것이다.

MEM_BASE(=0xA0000000)는 메모리 설정(memory configuration) 레지스터들의 기본(base)주소를 담고 있다. 이것을 r0로 가져온다. mcs0(=0xfff8fff8)[3]를 다시 r1에 가져오고, MCS0(=0x10)과 r0를 더한 값이 가르키는 곳에 r1을 넣도록 한다. MCS0는 static memory control register 0을 의미한다. 이 레지스터와 MCS1은 read/write가 가능한 레지스터들로

[3] SA1100을 사용한다면 Brutus가 정의된 곳을 사용할 것이다.

서 static memory에 대한 설정을 제어한다. Reset후에 가질 수 있는 값은 가능한 가장 늦은 speed의 ROM에 맞춰지며, 각각의 레지스터는 동일한 두개 16 bit의 field를 가진다. 따라서, 전체로는 4개의 필드로 구성된다.

여기서 잠시 SA1100의 메모리와 관련된 register들을 조금 정리해보도록 하겠다. 먼저 메모리 configuration register가 있는 위치는 다음과 같다.

위치	심벌	이름
0xA000 0000	MDCNFG	DRAM Configuration Register
0xA000 0004	MDCAS0	DRAM CAS Waveform Shift Register 0
0xA000 0008	MDCAS1	DRAM CAS Waveform Shift Register 1
0xA000 000C	MDCAS2	DRAM CAS Waveform Shift Register 2
0xA000 0010	MSC0	Static Memory Control Register 0
0xA000 0014	MSC1	Static Memory Control Register 1
0xA000 0018	MECR	Expansion Bus Configuration Register

표 1. SA1100의 메모리 설정 레지스터

MDCNFG 레지스터는 32 bit read/write 레지스터로서 DRAM을 설정하는 control bit을 가진다. 모든 DRAM bank들은 같은 종류의 DRAM device들로 구성되어야 한다. MDCASX의 심벌을 가지는 레지스터는 32 bit의 read/write가 가능하며, CAS(Column Address Selection)의 waveform을 결정해주는 것으로 DRAM을 접근하는데 필요한 CPU cycle을 나타내준다. MSCX의 심벌을 가지는 레지스터는 32 bit의 read/write가 가능하며, ROM이나 flash와 같은 Static 메모리에 대한 제어를 담당한다. MECR 레지스터는 expansion 메모리의 설정을 관리하며, PCMCIA와 같은 것을 위해서 사용되어 진다. 좀더 자세한 것을 알고자 한다면, SA1100의 개발자 메뉴얼을 참고하기 바란다. 현재로서는 이와 같은 레지스터에 특정한 값을 설정해서 BLOB을 load하거나 C 함수를 수행하기 위해서 reset 이후에 올바른 값으로 정해 주어야 한다는 것만 기억하기 바란다.

```
        ldr     r1, MEM_START
.rept   8
        ldr     r0, [r1]
.endr
#elif defined USE_SA1110
...
# Note : 이 부분은 SA1110을 위한 메모리 설정을 다룬다. 거의 비슷한 과정을 거치는 것이므로 생략하도록 하겠다.
```

```
...
#else
#error "Configuration error: CPU not defined!" /* 만약 SA11x0을 사용하지 않는다면 에러로 처리한다. */
#endif
        mov     pc, lr                  /* Sub routine을 복귀한다. (앞에서 BL로 jump했음)*/
```

코드 7. memsetup.S 파일

앞에서는 이미 메모리와 관련된 register들에 대한 설정을 마쳤다. r1에 MEM_START(=0xC0000000)의 값을 읽어오고, r0에 r1이 가르키고 있는 곳에서 값을 읽어 넣는 일을 8번 반복한다. 이것은 disable된 bank가 refresh하지 못하도록 막는 역할을 한다. 이젠 메모리에 대한 설정을 마쳤으므로 pc(program counter register)에 앞에서 branch시에 저장해 두었던 lr(=link register)의 내용을 가져오도록 한다(mov). 이것으로 subroutine의 return이 이루어진다.

```
.globl ledinit
ledinit:
        ldr     r0, GPIO_BASE
        ldr     r1, LED
        str     r1, [r0, #GPDR]         /* LED GPIO is output */
        str     r1, [r0, #GPSR]         /* turn LED on */
        mov     pc, lr
.globl led_on
        /* turn LED on. clobbers r0 and r1 */
led_on:
        ldr     r0, GPIO_BASE
        ldr     r1, LED
        str     r1, [r0, #GPSR]
        mov     pc, lr
.globl led_off
        /* turn LED off. clobbers r0 and r1 */
led_off:
        ldr     r0, GPIO_BASE
        ldr     r1, LED
        str     r1, [r0, #GPCR]
```

```
        mov     pc, lr
```

코드 8. ledasm.S 파일

이젠, ledasm.S를 보기로 하자. 이곳에선 단순히 LED에 대한 초기화만을 담당한다. LED의 GPIO를 초기하하고 , LED를 켜준다. 물론, 여기서 정의된 led_on()과 led_off() 함수는 LED를 켜고 끄는 역할을 하는 함수로 나중에 사용될 것이다. ledinit() 함수를 먼저 보도록 하자. r0에 GPIO_BASE(=0x90040000)를 넣고, r1에는 LED를 넣는다. LED값은 led.h 파일에 아래와 같은 정의를 가진다.

```
/* define the GPIO pin for the LED */
#if defined ASSABET
# define LED_GPIO 0x00020000 /* GPIO 17 */
#elif (defined CLART) || (defined LART) || (defined NESA)
# define LED_GPIO 0x00800000 /* GPIO 23 */
#elif defined PLEB
# define LED_GPIO 0x00010000 /* GPIO 16 */
#else
#warning "FIXME: Include code to turn on one of the LEDs on your board"
# define LED_GPIO 0x00000000 /* safe mode: no GPIO, so no LED */
#endif
```

코드 9. led.h 파일

즉, ASSABET으로 정의된 경우에는 0x00020000(=GPIO17)을, LART나 CLART 및 NESA가 정의된 경우에는 0x00800000(=GPIO23)을, PLEB가 정의된 경우에는 0x00010000(=GPIO16)을 각각 사용한다. 정의된 값이 없다면, 0x00000000이 될 것이다. 우리가 사용하는 정의는 ASSABET인 경우이므로, LED_GPIO는 0x00020000이 될 것이다. 이 값으로 r1을 초기화 하고, GPDR에 output으로 GPIO의 pin direction을 설정하게 되며, GPSR에 해당 bit을 ON시켜서 LED가 켜지도록 만든다. 나머지 LED를 켜는 함수와 끄는 함수는 GPSR과 GPCR 레지스터의 해당 bit을 ON/OFF시키는 동작만 하도록 되어 있다. 이제 메모리와 LED에 대한 설정을 끝냈으므로 다시 start.S로 돌아가서 계속 이야기를 진행하도록 하자.

```
        /* check if this is a wake-up from sleep */
        ldr     r0, RST_BASE
        ldr     r1, [r0, #RCSR]
        and     r1, r1, #0x0f
```

```
        teq     r1, #0x08
        bne     normal_boot     /* no, continue booting */

        /* yes, a wake-up. clear RCSR by writing a 1 (see 9.6.2.1 from [1]) */
        mov     r1, #0x08
        str     r1, [r0, #RCSR]  ;

        /* get the value from the PSPR and jump to it */
        ldr     r0, PWR_BASE
        ldr     r1, [r0, #PSPR]
        mov     pc, r1
```

코드 10. start.S 파일(계속)

RST_BASE(=0x90030000)는 reset controller의 base주소를 가르킨다. SA1100의 reset controller는 다양한 reset을 관리할 목적으로 사용되며, 프로그래머의 관점에서 보면, 크게 두개의 register로 구성된다. 하나는 software reset를 시키는 일을 하며, 다른 하나는 프로세서의 reset 이유가 무엇인가를 나타내는 목적으로 사용된다. 각각은 RSRR(Reset Controller Software Reset Register)와 RCSR(Reset Controller Status Register)로 나뉜다.
먼저 RCSR의 레지스터를 읽어서 r1에 둔다. 이 값을 0x0F로 AND시켜서 하위 4bit만을 살펴보게되며, 이 값이 0x08과 같지 않다면, hardware reset이나, software reset, 혹은 watchdog reset이 들어간 상태임을 보여주게 된다. 나머지 하나는 sleep mode reset이다. 즉, sleep mode에서 깨어나게 되는 reset이 걸린다면, RCSR에 있는 sleep mode reset을 지우기 위해서 RCSR에 0x08을 넣어주며, PSPR(Power Manager Scratch Pad Register)[4]를 읽어서 sleep mode 이전의 프로세서의 설정을 읽어 r1에 넣어준 후, 이 값으로 jump를 한다. 즉, sleep mode에서 깨어나서 계속 연산을 진행해 나가는 것이 된다.
만약 sleep mode에서 reset이 걸린 것이 아니라면, normal_boot으로 제어가 이동해서 진행될 것이다. 우리가 현재 다루고 있는 것은 hardware reset이후의 상황이므로 normal_boot이 될 것이다.

```
normal_boot:
        /* enable I-cache */
        mrc     p15, 0, r1, c1, c0, 0       @ read control reg
        orr     r1, r1, #0x1000             @ set Icache
```

[4] PSPR에 들어있는 값은 프로세서의 설정을 나타내는 어떤 값이거나, ROM과 같은 곳에 있는 routine에 대한 pointer가 될 수 있을 것이다.

```
        mcr     p15, 0, r1, c1, c0, 0       @ write it back

        /* check the first 1MB  in increments of 4k */
        mov     r7, #0x1000
        mov     r6, r7, lsl #8      /* 4k << 2^8 = 1MB */
        ldr     r5, MEM_START
mem_test_loop:
        mov     r0, r5
        bl      testram
        teq     r0, #1
        beq     badram

        add     r5, r5, r7
        subs    r6, r6, r7
        bne     mem_test_loop
```

코드 11. start.S 파일(계속)

이젠 I-cache(Instruction Cache)를 enable시켜준다. 이것은 control register를 읽어서, I cache enable을 나타내는 bit을 설정한 후, 다시 이것으로 원래의 값을 대치시켜주면 된다. 이것을 마치고나면, 이젠 첫 1MBytes 영역에 대한 test에 들어간다. 먼저 r7에 4K(=0x1000)을 넣고, 다시 이를 좌측으로 8 bit shift해서 r6에 넣는다. 이렇게 하고나면, r6는 1 M을 가질 것이다. r5에는 MEM_START(=0xC0000000)로 두어 메인 메모리의 시작 주소를 가르키도록 한다. 메모리 테스트가 일어나는 곳은 mem_test_loop이다. 이곳에서 loop를 돌면서 확인하게 된다. 확인하는 단위는 4KBytes로 1MBytes길이에 대해서 일어난다. 먼저, r0에 메모리의 시작 주소를 가져온다(r5). 테스트는 testram() 함수에서 일어나게 되며(bl), 만약 테스트의 결과 값(r0)이 1이라면, badram으로 제어를 옮기게 되며, 그렇지 않다면, r5에 r7(=4K)를 더하고, r6에서는 r7을 빼서 다시 loop를 돌게된다.

```
.text
.globl testram
        @ r0 = address to test
        @ returns r0 = 0 - ram present, r0 = 1 - no ram
        @ clobbers r1 - r4
testram:
      ldmia   r0, {r1, r2}          @ store current value in r1 and r2
        mov     r3, #0x55           @ write 0x55 to first word
```

```
        mov     r4, #0xaa       @ 0xaa to second
        stmia   r0, {r3, r4}
        ldmia   r0, {r3, r4}    @ read it back
        teq     r3, #0x55       @ do the values match
        teqeq   r4, #0xaa
        bne     bad             @ oops, no
        mov     r3, #0xaa       @ write 0xaa to first word
        mov     r4, #0x55       @ 0x55 to second
        stmia   r0, {r3, r4}
        ldmia   r0, {r3, r4}    @ read it back
        teq     r3, #0xaa       @ do the values match
        teqeq   r4, #0x55
bad:    stmia   r0, {r1, r2}    @ in any case, restore old data
        moveq   r0, #0          @ ok - all values matched
        movne   r0, #1          @ no ram at this location
        mov     pc, lr
```

코드 12. testmem.S 파일

실제 source code를 보면, testmem.S와 testmem2.S 두개의 파일이 있다. 각 화일 하는 일은 동일하며, 차이가 나는 것은 stack을 이용해서 변동이 있게 되는 register의 내용을 저장할 것인가, 아니면, 그냥 수행할 것인가만이 차이가 나며, 나머지는 동일하다. 여기서는 testmem.S를 보도록 하겠다. r0에는 test의 시작 주소가, 그리고, 나중에 test의 결과 같이 r0로 돌아가게 된다. 결과값이 0이라면, RAM이 존재한다는 것을 나타내고, 그렇지 않다면, RAM이 없음을 나타낸다. r1에서 r4까지는 이 subroutine에서 사용되는 register들이다[5].

r0가 가르키는 곳에 원래 저장된 두개의 32 bit 값을 r1과 r2로 읽어드린다(ldmia). r3에는 0x55를 r4에는 0xAA를 두고, 이 값을 r0가 가르키는 곳에 적는다(stmia). 다시 r0가 가르키는 곳에서 r3와 r4를 읽어서, 이 값들디 앞에서 적었던 값과 일치하는지 검사한다(teq, teqeq). 만약 같지 않다면, 쓴 값과 읽은 값이 다르므로 RAM이 존재하지 않는다는 말이 되므로, bad로 제어를 옮기게 될 것이다. r3에는 다시 0xAA를 넣고, r4에 0x55를 넣은 후, r0가 가르키는 곳에 다시 r3와 r4를 저장한다(stmia). 다시 같은 위치에서 r3와 r4를 가져와서(ldmia)이 값이 앞서 쓴 값과 같은 지를 비교한다. r0에는 앞에서 저장해 두었던 r1과 r2를 다시 쓰게되고, r0에는 RAM이 존재하는지 그렇지 않은지에 따라서, 0이나 1값을 주게된다(moveq, movne). 마지막으로 (mov pc, lr)로 subroutine은 복귀한다. 따라서, 결과값은 r0에 들어갈 것이다.

---

[5] 이와 같은 이유로 앞에서 r1에서 r4까지는 사용하지 않았었다. r0와 r5, r6, r7만이 사용되었다.

```
badram:
        b       blinky
...
blinky:
        /* This is test code to blink the LED
        very useful if nothing else works */
        bl      led_on
        bl      wait_loop
        bl      led_off
        bl      wait_loop
        b blinky
wait_loop:
        /* busy wait loop*/
        mov     r2, #0x1000000
wait_loop1:
        subs    r2, r2, #1
        bne     wait_loop1
        mov     pc, lr
```

코드 13. start.S 파일(계속)

앞에서 제대로 RAM을 찾을 수 없었다면, 제어는 badram으로 옮겨지게 될 것이며, badram에서는 blinky로 branch(b)하게 된다. blinky에서는 단순히 LED를 깜빡이는 일을 할 것이다. led_on과 led_off는 앞에서 이미 보았지만, LED를 켜고 끄는 일을 하게되며, wait_loop는 delay를 주기 위해서 사용하고 있다.

```
        /* the first megabyte is OK, so let's clear it */
        mov     r0, #((1024 * 1024) / (8 * 4))          /* 1MB in steps of 32 bytes */
        ldr     r1, MEM_START
        mov     r2, #0
        mov     r3, #0
        mov     r4, #0
        mov     r5, #0
        mov     r6, #0
        mov     r7, #0
        mov     r8, #0
```

```
        mov     r9, #0
clear_loop:
        stmia   r1!, {r2-r9}
        subs    r0, r0, #(8 * 4)
        bne     clear_loop
```

코드 14. start.S 파일(계속)

이젠 첫 1 MBytes에 대한 확인이 끝났다. 이젠 확인된 1MBytes부분을 0으로 clear시키는 일을 하게된다. 먼저 r0에 32 bytes단위로 1 Mbytes를 나눈 값으로 초기화한 후, r1에는 시작 메모리의 주소(MEM_START)를 가르키도록 만든다. r2에서 r9까지는 0으로 초기화한다. stmia(Store and Increment after)를 사용해서 r1이 가르키는 주소에 r2에서 r9까지를 저장한 후, r1을 증가시킨다. r0는 원래의 값에서 32를 감소시키게되며, 1 Mbytes를 clear했는지를 알아본다. Clear를 다하지 못했다면, 계속 clear_loop를 돌면서 수행할 것이다.

```
        /* get a clue where we are running, so we know what to copy */
        and     r0, pc, #0xff000000     /* we don't care about the low bits */

        /* relocate the second stage loader */
        add     r2, r0, #(128 * 1024)   /* blob is 128kB */
        add     r0, r0, #0x400          /* skip first 1024 bytes */
        ldr     r1, MEM_START
        add     r1, r1, #0x400          /* skip over here as well */
```

코드 15. start.S 파일(계속)

이젠 현재 우리가 어디서 진행하고 있는지를 알기 위해서 pc(Program Counter)의 상위 1bytes만을 제외하고 나머지를 0으로 clear시켜서 r0에 넣는다. 이 값에 128K(=128x1024)를 곱해서 r2로 두고, r0에는 다시 0x400만큼을 더한다. r1에는 메모리 시작주소(MEM_START)를 가르키도록 만들고, 여기에 0x400을 더해서 첫 1024(=0x400)을 skip하도록 한다. 이 절차는 rest-ld-script이라는 곳에서 시작 주소를 0xC0000400으로 준 것에 관련되어 있는 것으로 blob가 첫 1024 Bytes이후의 번지로 copy가 될 것이기 때문이다.

```
        /* r0 = source address
         * r1 = target address
         * r2 = source end address
         */
```

```
copy_loop:
        ldmia   r0!, {r3-r10}
        stmia   r1!, {r3-r10}
        cmp     r0, r2
        ble     copy_loop

        /* turn off the LED. if it stays off it is an indication that
         * we didn't make it into the C code
         */
        bl led_off

        /* set up the stack pointer */
        ldr     r0, MEM_START
        add     r1, r0, #(1024 * 1024)
        sub     sp, r1, #0x04

        /* blob is copied to ram, so jump to it */
        add     r0, r0, #0x400
        mov     pc, r0
```

코드 16. start.S 파일 (계속)

r0에는 앞에서 copy할 source의 시작 주소가 들어가도록 하고, r1에는 target이 되는 주소를, r2에는 copy의 대상이 되는 것의 마지막 주소(128Kbytes의 크기)가 들어가도록 해주었다. 이젠 r0가 가르키는 곳에선 r1이 가르키는 곳으로 copy를 한다. 이것은 ldmia와 stmia의 연속적인 것으로 가능하며, r0가 r2와 같게 되었을 때, copy가 끝난다. 그렇지 않다면, copy_loop로 제어를 옮겨서 계속 copy를 할 것이다.
copy가 끝나면, LED를 off시키고(led_off), r0는 다시 메모리의 시작 번지로 만들어주게 되며(MEM_START), r1은 메모리의 시작주소에서 1Mbytes 떨어진 곳을, sp는 다시 r1보다 4작은 곳을 가르키도록 만든다. 이젠 stack pointer를 설정해 주었으므로 C routine을 실행할 준비가 되었다. 이미 앞에서 1024Bytes 정도 MEM_START에서 떨어진 곳에 copy를 해두었으므로, r0에 다시 0x400을 더한후 pc를 그곳으로 옮겨서 실행을 계속 진행하도록 한다(mov pc, r0). 실제 수행이 옮겨지는 곳은 trampoline.S에 정의된 second stage boot loader로서 다음과 같은 정의를 가진다.

```
.text
```

```
.globl _trampoline
_trampoline:
        bl      main
        /* if main ever returns we just call it again */
        b       _trampoline
```

코드 17. trampoline.S 파일

trampoline.S 파일은 단지 main() 함수로 branch with link하는 연산만을 수행할 뿐이다. 만약 main에서 return하게 된다면, 다시 한번 _trampoline을 호출하도록 해서 계속 loop를 돈다. 이젠 main.c로 진행할 준비가 다 끝난 것이며, main.c에 있는 main() 함수가 수행될 것이다.