



설계와 구현을 통한 임베디드 OS의 이해와 응용 _2

개발환경 구성 및 부트로더 작성

이번 호에서는 'cuteOS' 구현을 위한 개발환경 구성 방법을 알아보고, GNU toolchain 설치 및 부트로더를 작성해 보기로 하자. 또한 mmu의 정확한 특성도 이해해 보기로 하겠다.

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

cuteOS 개발환경

• 호스트 시스템

- i386 PC
- 와우리눅스 7.3 Paran R2
- cross-2.95.3.tar.bz2(크로스 컴파일러)

• 타겟 시스템

- s3c2440A SOC가 장착된 보드(ARM920T)
- 1M NOR flash ROM (0x00000000에 위치)
- 64M SDRAM(32M x 2) (0x30000000에 위치)

이외에 실행 이미지를 타겟 시스템으로 다운로드하기 위한 JTAG와 디버깅을 위해서 시리얼을 사용한다.

앞으로 작성할 소스코드는 C와 ARM 어셈블리어를 사용하며, 이 두 가지 언어는 이미 숙지하고 있다고 가정하고 기사를 진행하기로 하겠다. 또한 ARM 프로세서 내의 레지스터의 구조와 용도, 인터럽트의 처리 방식 등, ARM 프로세서의 programmer's model에 대해 잘 이해하고

있어야 하며, 리눅스 사용법에 대해서도 어느 정도 익숙하다는 가정 하에 설명하겠다.

그러면 먼저 i386 PC에 와우리눅스 7.3 Paran R2 OS가 설치되어 있는 환경에서 ARM용 크로스 툴체인을 설치하는 과정을 살펴보기로 하자. 우리가 사용할 ARM용 크로스 툴체인 cross-2.95.3.tar.bz2 파일은 다음 사이트에서 구할 수 있다.

<http://ftp.arm.linux.org.uk/pub/armlinux/toolchain/>

참고로 리눅스를 기반으로 한 ARM과 관련된 개발을 할 경우 다음 사이트를 추천한다.

<http://www.arm.linux.org.uk/>

cross-2.95.3.tar.bz2 파일을 설치하기 위해 리눅스 상에서 먼저 다음과 같은 작업을 한다. 파일의 설치에 반드시 root 사용자 권한으로 해야 한다.

```
# cd /usr/local/
# mkdir arm
# cd arm/
```

즉, /usr/local/ 디렉토리로 들어가 arm 디렉토리를 생성한 후, arm/ 디렉토리로 들어간다. 그리고 cross-2.95.3.tar.bz2 파일을 위의 사이트에서 다운로드 받아 /usr/local/arm/ 디렉토리로 가져온다. 그리고 /usr/local/arm/ 디렉토리에서 tar 명령어를 이용하여 다음과 같이 파일을 설치한다.

```
# tar xvjf cross-2.95.3.tar.bz2
```

이와 같이 하면 /usr/local/arm/2.95.3/ 디렉토리에 크로스 툴체인이 설치된다. 설치한 툴체인을 사용하기 위해 홈 디렉토리 밑에 있는 .bash_profile 파일의 마지막 부분에 다음과 같이 두 문장을 추가해 준다.

```
PATH=/usr/local/arm/2.95.3/bin:$PATH
export PATH
```

참고로 cuteOS의 개발은 일반 사용자의 권한으로도 할 수 있으니, 일반 사용자 계정을 추가한 후 일반 사용자의 홈 디렉토리 밑에 있는 .bash_profile 파일에 위와 같은 내용을 추가하고 개발을 진행하도록 하자. 참고로 필자는 일반 사용자의 계정으로 cuteOS를 개발해 나갈 것이다.

이렇게 .bash_profile 파일을 수정한 후에 이 내용을 적용하기 위해서는 다시 로그인을 하거나, 또는 다음과 같은 리눅스 명령어를 이용하여 .bash_profile 파일의 내용을 현재 셸에 적용시켜야 한다.

```
# source ~/.bash_profile
```

앞으로 우리가 사용할 명령어들은 /usr/local/arm/2.95.3/bin/ 디렉토리 밑에 존재하며 다음과 같은 명령어들을

주로 사용한다.

- arm-linux-gcc
- arm-linux-ld
- arm-linux-objcopy
- arm-linux-nm
- arm-linux-objdump

이러한 툴들에 대한 설명은 실제 개발을 진행해 가면서 적절한 부분에서 설명하기로 하겠다.

이상에서 ARM용 크로스 툴체인을 설치하는 과정을 살펴봐왔다.

부트로더 작성

다음은 부트로더의 앞부분을 어떻게 작성하는지, 최종적으로 실행 이미지를 어떻게 만드는지 보기로 하자. 먼저 지금 우리가 작성할 부트로더는 물리적으로 0x00000000 번지에 위치하며, 다음에 작성할 cuteOS는 물리적으로 0x30000000 번지에 위치하게 된다.

이번 호에서 작성할 예제는 세 파일로 구성된다. 즉, 부트로더의 최초 시작 지점인 start.S, 부팅시 디버깅용으로 사용할 LED를 제어하기 위한 led.S, 그리고 Makefile로 구성된다.

먼저 start.S의 내용은 다음과 같다.

start.S

```
#define PWTCN 0x53000000
#define INTMSK 0x4a000008
#define INTSUBMSK 0x4a00001c

.globl _start
_start:
    b reset
```

```

reset:
    mrs r0,cpsr
    bic r0,r0,#0x1f
    orr r0,r0,#0xd3
    msr cpsr,r0

    ldr r0,=pWTCON
    mov r1,#0x0
    str r1,[r0]

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

    ldr r1,=0x7fff
    ldr r0,=INTSUBMSK
    str r1,[r0]

    bl led_init

    bl led_off

    mov r0,#0x00000010
    bl led_on

1: b 1b
    
```

start.S 파일에서 하는 일은 다음과 같다.

1. CPU를 SVC32 mode로 전환한다.
2. watchdog timer를 끈다.
3. 인터럽트 컨트롤러의 모든 인터럽트 소스를 mask 처리한다.

그럼 구체적으로 start.S 파일의 내용을 들여다보자.

```

.globl _start
_start:
    b reset
    
```

이 부분은 파일의 시작 지점으로 물리적으로 0x00000000 번지에 놓이게 된다. 0x00000000 번지에는 1MB 크기의 플래시 ROM이 놓여 있다. 즉, _start 지점이 0x00000000 번지에 놓이게 되며, <b reset> 명령어는 0x00000000 번지에 놓이는 최초의 명령어가 된다. 이 명령어에 의해 CPU는 reset 위치로 쏜다.

<.globl _start>에서 .globl은 전체 목적 파일을 링크할 때 _start 심벌을 링커 ld가 볼 수 있도록 하는 역할을 한다. _start 지점이 0x00000000 번지에 놓이는 지는 뒤에서 arm-linux-nm 이라는 명령어를 이용하여 확인해 보기로 한다.

```

.balignl 16,0xdeadbeef
    
```

이 부분은 다음에 올 코드를 16바이트 경계에 놓고 중간에 비게 되는 부분을 4바이트 크기의 0xdeadbeef 값으로 채우라는 어셈블러 지시어이다. 즉, reset의 위치는 0x00000010 번지가 되며, <b reset> 명령어와 reset 번지의 사이는 0xdeadbeef로 채워진다. 이 부분 역시 뒤에서 'arm-linux-objdump' 라는 명령어를 이용하여 확인하기로 한다.

```

reset:
    mrs r0,cpsr
    bic r0,r0,#0x1f
    orr r0,r0,#0xd3
    msr cpsr,r0
    
```

여기는 cpsr 레지스터의 값을 r0 레지스터로 읽어온 후, r0 레지스터의 하위 5비트를 이진수 00000으로 채우고, 다시 하위 5비트와 5, 6, 7번 비트를 이진수 11010011

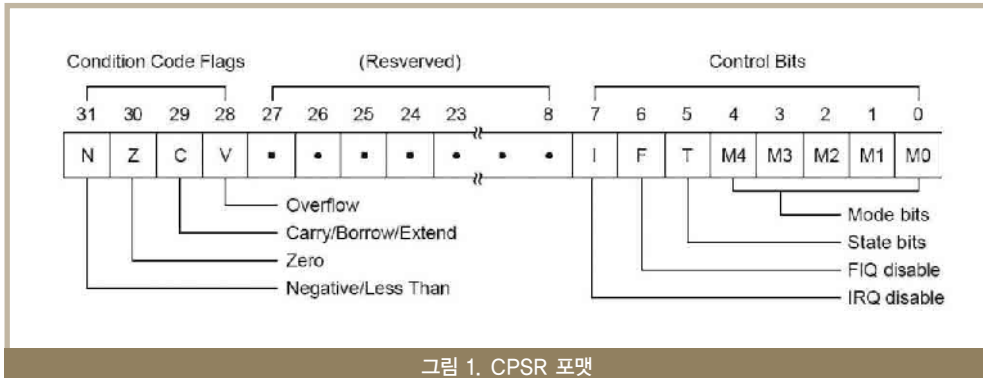


그림 1. CPSR 포맷

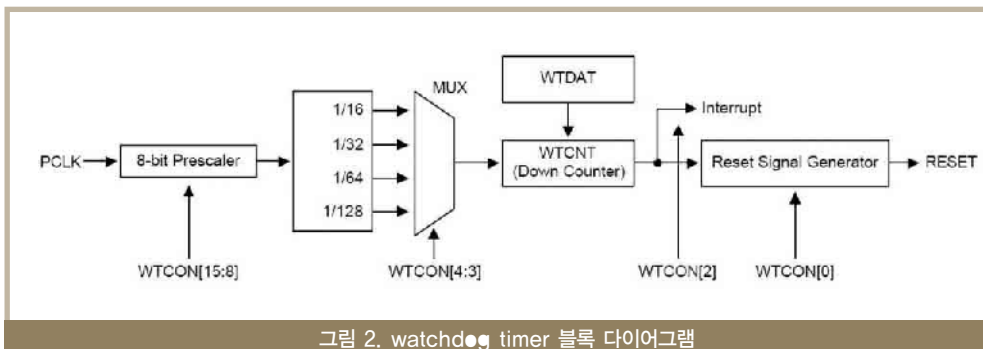


그림 2. watchdog timer 블록 다이어그램

값으로 채운 후, 수정한 r0 레지스터의 값을 cpsr 레지스터로 쓰는 부분이다. 이 부분은 CPU를 SVC32 모드로 전환을 하며, IRQ와 FIQ를 끄는 역할을 한다. 참고로 CPSR의 포맷은 그림 1과 같다.

```
ldr r0,=pWTCN
mov r1,#0x0
str r1,[r0]
```

이 부분은 watchdog timer를 끄는 역할을 한다. watchdog timer의 위치는 다음 문장에서 나타내는 것처럼 0x53000000 번지에 있다.

```
#define pWTCN 0x53000000
```

watchdog timer의 주된 역할은, 예를 들어 OS가 deadlock 등에 의해 정상적으로 동작하지 않을 경우 CPU에 reset 신호를 보내주어, 시스템을 재부팅하는 역

0으로 만듦으로써 앞에서 설명한 것과 같은 동작을 수행한다.

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

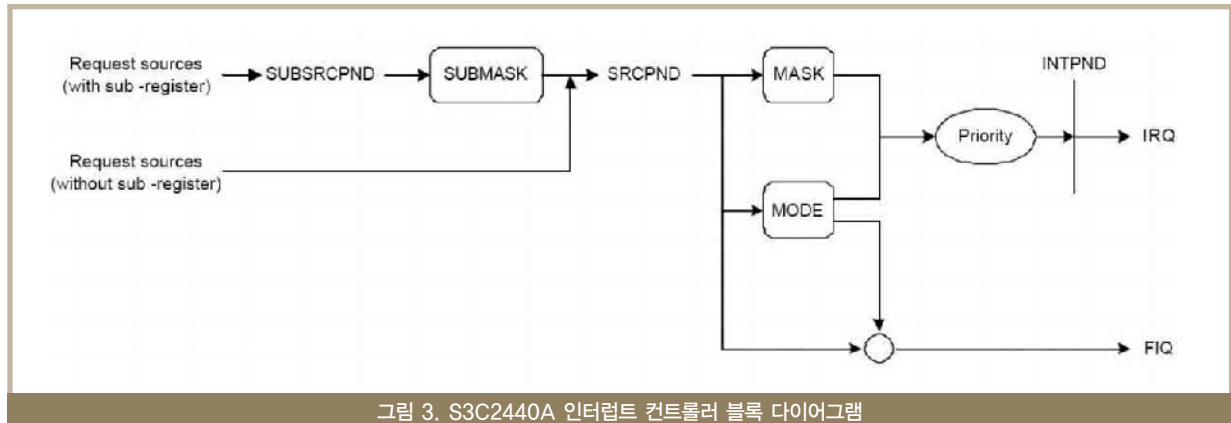
ldr r1,=0x7fff
ldr r0,=INTSUBMSK
str r1,[r0]
```

이 부분은 인터럽트 컨트롤러에 연결되어 있는 모든 인터럽트 소스를 mask 처리하는 부분이다. S3C2440A 프로세서에는 60개의 인터럽트 소스가 존재한다. S3C2440A의 인터럽트 컨트롤러 블록 다이어그램은 다음과 같다.

그림 3에서 인터럽트 컨트롤러의 SUBMASK 레지스

할을 한다. 좀 더 자세히 살펴보면, 여기서는 WTCN 레지스터의 0번 비트와 2번 비트를 0으로 만들어 줌으로써 watchdog timer의 reset 신호를 발생시키는 기능과 인터럽트를 발생시키는 기능을 비활성화시킨다. watchdog timer의 블록 다이어그램은 그림 2와 같다.

그림 2에서 WTCN[0]과 WTCN[2]에 해당하는 부분을



터와 MASK 레지스터를 위의 소스와 같이 처리함으로써 프로세서 내부 모듈과 외부에서 오는 모든 인터럽트 소스를 mask 처리한다.

```
bl led_init

bl led_off

mov r0,#0x00000010
bl led_on
```

이 예제를 수행하는 타깃 보드에는 4개의 LED가 있으며, 부팅시에 디버깅용으로 사용한다. 여기서는 각각 LED를 초기화하고 OFF 시킨 후에 첫 번째 LED를 켜다.

‘mov r0,#0x00000010’ 명령어는 켜고자 하는 해당 LED의 번호를 r0 레지스터로 넘겨주는 부분이다. LED에 대한 구체적인 설명은 led.S 파일에서 다루겠다.

```
1: b 1b
```

이 부분을 C로 표현하면 논리적으로 다음과 같다.

```
while(1);
```

위의 명령어의 1b에서 b는 backward의 약자로 명령어 앞에 있는 숫자 1의 위치를 의미한다. 즉, 무한히 1의 위

```
led.S
#define rGPFCON 0x56000050
#define rGPFDAT 0x56000054
#define rGPFUP 0x56000058

#define dGPFCON 0x000055aa
```

```
.globl led_init
led_init:
    ldr r0,=rGPFCON
    ldr r1,=dGPFCON
    str r1,[r0]

    ldr r0,=rGPFUP
    mov r1,#0x000000ff
    str r1,[r0]

    mov pc,lr
```

```
.globl led_off
led_off:
    ldr r0,=rGPFDAT
    ldr r1,[r0]
    orr r1,r1,#0x000000f0
```

```

str r1,[r0]

b delay

.globl led_on
led_on:
    ldr r1,=rGPFDAT
    ldr r2,[r1]
    bic r2,r2,r0
    str r2,[r1]
delay:
    mov r0,#0x8000
1:
    subs r0,r0,#1
    bne 1b

    mov pc,lr
    
```

led.S 파일에는 세 가지 함수가 있다. led_init, led_off, led_on의 세 함수가 그것들이며, 각각 LED를 초기화하고, 끄고 켜는 역할을 한다.

그럼 led.S 파일의 내용을 자세히 들여다보자.

```

led_init:
    ldr r0,=rGPFCON
    ldr r1,=dGPFCON
    str r1,[r0]

    ldr r0,=rGPFUP
    mov r1,#0x000000ff
    str r1,[r0]

    mov pc,lr
    
```

이 부분은 LED를 초기화 하는 함수이다. 필자가 사용하는 보드에는 LED가 GPF(F port)의 4, 5, 6, 7번 비트

에 물려있다. GPF는 총 8개의 외부 핀으로 연결되며, 각각의 핀을 input, output 또는 외부 인터럽트를 받을 수 있는 용도 중 하나로 사용할 수 있다.

이 중 필자가 사용하는 보드에서는 GPF의 4, 5, 6, 7번 비트에 LED를 연결하고 output용으로 사용한다. GPF를 제어하기 위해서는 GPFCON, GPFDAT, GPFUP의 세 레지스터를 사용한다. 각각의 레지스터는 다음에서 정의하는 것처럼 0x56000050, 0x56000054, 0x56000058 번지에 있다.

```

#define rGPFCON 0x56000050
#define rGPFDAT 0x56000054
#define rGPFUP 0x56000058
    
```

위의 코드에서는 GPFCON 레지스터를 이용하여 GPF의 4, 5, 6, 7번 비트를 output용으로 설정하고, GPFUP 레지스터를 이용하여 풀업 저항을 비활성화시키고 있다. 풀업 저항은 GPIO 핀을 input용으로 사용할 경우, input 값이 불안정하게 들어오는 것을 막는 역할을 한다.

즉, input 값이 0과 1사이의 어중간한 값으로 들어올 때 0 또는 1로 고정시켜 주는 역할을 한다. 따라서 GPIO 핀을 output용으로 사용할 경우에는 풀업 저항을 사용할 필요가 없다. 그래서 여기서는 풀업 저항을 비활성화시키고 있다.

led_off 함수는 GPFDAT 레지스터의 4, 5, 6, 7번 비트에 이진수 1111을 써줌으로써 LED를 끄는 역할을 한다.

led_on 함수는 r0 레지스터로 넘어온 값에 해당하는 LED를 켜주는 역할을 한다. 즉, r0의 값이 0x00000010일 경우 GPF의 4번 비트에 해당하는 LED를 켜주고, r0의 값이 0x00000020일 경우 GPF의 5번 비트에 해당하는 LED를 켜주는 역할을 한다. delay 루틴은 LED가 꺼져 있거나 켜져 있는 시간을 일정하게 유지시켜 주는 역할을 한다.

마지막으로 Makefile의 내용을 들여다보자.



Makefile

```
cute-boot: start.o led.o
    arm-linux-ld start.o led.o -o cute-boot -
Ttext 0x00000000
    arm-linux-objcopy cute-boot cute-boot.bin
-O binary

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

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

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

먼저 Makefile의 작성 요령을 다음 예를 통해서 간단하게 알아보자.

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

여기서 start.o: start.S와 같이 콜론 “:”을 포함하고 있는 부분을 dependency line이라 한다. 콜론을 중심으로 왼쪽 부분을 타깃이라 하고, 오른쪽 부분을 타깃을 만들기 위한 소스라고 한다. dependency line은 다음과 같은 의미를 갖는다.

- target 부분은 source에 의존한다.

즉, 위에서 start.o는 start.S에 의존한다. make 명령어를 이용하여 Makefile의 내용을 수행할 경우 make 명령어는 start.o 파일이 바뀐 시간을 start.S 파일이 바뀐 시간과 비교한다. 만약 start.S 파일이 start.o 파일보다 더 최근에 바뀌었다면, make는 arm-linux-gcc

start.S -c 부분을 수행한다.

즉, start.o 파일을 start.S 파일의 내용에 맞게 새롭게 컴파일을 수행한다. 주의할 점은 arm-linux-gcc start.S -c와 같은 명령어 부분은 반드시 탭 키로 한 칸 띄어 주어야 한다. 즉, 다음과 같이 명령어 부분을 작성해야 한다.

```
<탭>arm-linux-gcc start.S -c
```

탭이 아닌 스페이스 키 등으로 띄어 줄 땐 에러가 발생하니 주의하기 바란다. 위의 Makefile을 이용하여 make 명령어를 수행할 경우 다음과 같이 나타난다.

```
$ make
arm-linux-gcc start.S -c
arm-linux-gcc led.S -c
arm-linux-ld start.o led.o -o cute-boot -
Ttext 0x00000000
arm-linux-objcopy cute-boot cute-boot.bin -O
binary
```

여기서는 arm-linux-gcc를 이용하여 start.S와 led.S 파일을 각각 start.o와 led.o 파일로 만든다.

다음으로 arm-linux-ld를 이용하여 이 두 개의 목적 파일로 cute-boot 결과 파일을 만든다. arm-linux-ld 명령어의 -o 옵션은 output의 약자이다. 결과 파일의 맨 앞부분이 놓일 메모리 위치는 0x00000000 번지가 된다. -Ttext 옵션은 “text” 세그먼트의 시작위치를 링커에게 알려주는 옵션이다. cute-boot 파일은 elf 포맷의 파일이며, 다음과 같이 file이란 명령어를 이용하여 cute-boot 파일의 형식을 알 수 있다.

```
$ file cute-boot
cute-boot: ELF 32-bit LSB executable, ARM,
version 1 (ARM), statically linked, not stripped
```

여기서 cute-boot 파일이 elf 포맷의 32비트 ARM용

실행 파일임을 알 수 있다.

마지막으로 arm-linux-objcopy 명령어는 파일의 포맷을 바꾸는 역할을 하며, 여기서는 elf 포맷의 cute-boot 파일을 순수 바이너리 이미지로 바꾸는 역할을 한다. 왜냐하면, 우리가 작성한 프로그램을 타겟 보드상에서 수행하기 위해서는 elf 포맷의 파일을 순수 바이너리 이미지 파일로 만들어야 한다.

즉, arm-linux-objcopy 명령어는 입력 파일로 cute-boot를 받아 출력 파일로 cute-boot.bin을 만들어 내는데 그 포맷은 -O binary 옵션에 의해 순수 바이너리 이미지 파일이 된다.

참고로 다음과 같이 명령어를 수행해 보자.

```
$ make clean
rm -f *.o
rm -f cute-boot
rm -f cute-boot.bin
```

이렇게 명령어를 사용할 경우 make는 Makefile에서 clean에 해당하는 타겟을 찾는다. 여기서는 *.o, cute-boot, cute-boot.bin 등 make를 수행하는 과정에서 생성된 파일을 모두 제거하는 역할을 한다.

이상에서 Makefile의 내용을 알아보았다.

최종적으로 생성된 cute-boot.bin 이미지 파일을 JTAG를 이용하여 보드상에 다운로드 받아 실행해 보고 의도된 대로 동작하는지 확인해 본다.

다음은 arm-linux-nm 명령어와 arm-linux-objdump 명령어를 이용하여 cute-boot 파일의 내용을 좀 더 자세히 살펴보자.

이러한 명령어들은 실제 개발하는 과정에서 디버깅용으로 유용하게 사용할 수 있는 명령어들이며, 이 후 기사에서도 종종 사용할 것이다.

먼저 arm-linux-nm 명령어는 실행 파일로부터 심벌에 대한 정보를 보여 주는 역할을 한다. 참고로 순수 바이너리 이미지에는 심벌에 대한 정보가 존재하지 않는다. 따라서 여기서는 elf 포맷의 cute-boot 파일을 이용하여 심벌의 정보를 본다.

다음과 같이 arm-linux-nm 명령어를 수행해 본다.

```
$ arm-linux-nm cute-boot -n
00000000 T _start
00000010 t reset
00000070 T led_init
0000008c T led_off
000000a0 T led_on
000000b0 t delay
000080d0 A __bss_end__
000080d0 A __bss_start
000080d0 A __bss_start__
000080d0 D __data_start
000080d0 A __end__
000080d0 A __bss_end__
000080d0 A __edata
000080d0 A __end
```

여기서 우리는 _start가 0x00000000 번지에 놓여 있는 것을 알 수 있다. 또한 reset이 0x00000010 번지에 놓여 있는 것을 볼 수 있다. 참고로 __bss_end__, __bss_start__, __bss_start__, __data_start__, __end__, __bss_end__, __edata, __end 등의 심벌들은 ld가 자체적으로 정의해서 사용하는 심벌들이다. 궁금하면 다음과 같이 명령어를 수행해 보고 나온 결과를 자세히 살펴보기 바란다.

```
$ arm-linux-ld --verbose
```

다음은 arm-linux-objdump 명령어에 -D 옵션을 이용하여 cute-boot 파일을 역어셈블 해 보기로 하자. 다음과 같이 명령어를 수행한다.

```
$ arm-linux-objdump cute-boot -D

cute-boot: file format elf32-littlearm
```




Disassembly of section .text:

00000000 <start>:

```
0: ea000002 b 10<reset>
4: deadbeef cdple 14, 10, cr11, cr13, cr15, {7}
8: deadbeef cdple 14, 10, cr11, cr13, cr15, {7}
c: deadbeef cdple 14, 10, cr11, cr13, cr15, {7}
00000010 <reset>:
10: e10f0000 mrs r0, CPSR
14: e3c0001f bic r0, r0, #31; 0x1f
18: e38000d3 orr r0, r0, #211; 0xd3
...
```

그러면 역어셈블한 내용을 좀 더 자세히 들여다보자.

cute-boot: file format elf32-littlearm

이 부분에서 우리는 cute-boot 파일이 little endian 방식의 ARM용 32비트 elf 포맷이라는 정보를 얻을 수 있다.

Disassembly of section .text:

이 부분은 .text 세션에 대한 역어셈블을 의미하며 아래에 실제 역어셈블한 내용이 나온다. .text 세션은 일반적으로 함수 부분이 컴파일 되어 위치하게 되는 부분이다.

00000000 <start>:

이 부분은 0x00000000 번지에 _start가 위치한다는 의미이다.

```
0: ea000002 b 10<reset>
```

역어셈블한 결과는 세 부분으로 나뉜다. 먼저 맨 왼쪽에 주소부분, 중간에 기계어, 맨 오른쪽에 기계어에 대응

하는 어셈블리어로 나타난다. 이 부분은 0번지에 'ea000002' 라는 기계어가 있으며, 이 기계어를 역어셈블한 결과가 다음과 같다는 의미이다.

```
b 10<reset>

4: deadbeef cdple 14, 10, cr11, cr13, cr15, {7}
8: deadbeef cdple 14, 10, cr11, cr13, cr15, {7}
c: deadbeef cdple 14, 10, cr11, cr13, cr15, {7}
```

이 부분은 다음 부분을 컴파일하고 링크하여 나온 결과이다.

```
.balignl 16,0xdeadbeef
```

앞에서 이 부분에 대한 설명을 하였으니 다시 한번 확인하기 바란다.

00000010 <reset>:

이와 같이 0x00000010 번지에 reset이 위치하는 점도 참고하기 바란다.

이상에서 개발환경을 구성하는 방법과, 부트로더의 앞부분을 작성해 보았다. 참고로 기사에서 작성하던 예제를 기사와 함께 <Real-Time EmbeddedWorld> 홈페이지 자료실에 올리기로 하겠다. 참고하기 바란다.

다음 호에서는 cache와 MMU 설정을 포함하여 부트로더의 나머지 부분을 완성해 보기로 하겠다. ^{Real}Time

참고 자료

<http://sourceforge.net/projects/u-boot>

u-boot-1.1.2.tar.bz2

<http://emlinux.co.kr/index.php>

u-boot-1.0.0-emlinux.tgz

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