Firmware in Linux

(제 4회 리눅스 세미나 강의록)

삼성 소프트웨어 멤버쉽

리눅스 사용자 모임

ssmLUG

정 병 수

(bsjung@alpha.secsm.org)

2000년 3월 9일 수정

2000년 3월 26일 발표

내용 목차

Firmware in Linux

(제 4회 리눅스 세미나 강의록)

이번에 다루게 될 내용은 리눅스로 펌웨어라는 시스템 영역에 해당하는 부분을 프로그램하는 방법을 다룬다. 펌웨어란 시스템의 BIOS나 ROM 모니터 프로그램등 매우 시스템 깊숙한 부분이다. 일반적으로 이러한 펌웨어를 다루는 사람들을 펌웨어 엔지니어라고 부르며, 펌웨어 엔지니어라고 하는 사람들은 시스템 커널과 디바이스 드라이버를 제외한 모든 시스템 프로그램을 만들게 된다.

이번 강좌에서는 3가지 Firmware를 다르게 되는데, 첫 번째는 Alpha MILO와 OpenBIOS 프로젝트의 BIOS에 대한 것이고, 두 번째는 PMON라는 시스템 ROM 모니터 프로그램을 다루며, 세 번째는 ELKS라는 내장형 리눅스 커널과 디바이스 드라이버를 다룬다.

I. BIOS : Alpha MILO와 OpenBIOS 프로젝트

BIOS는 영어로 Basic Input/Output System을 준인 말이다. 즉, 시스템의 가장 기본적인 입출력을 다루는 부분으로, 시스템 커널이 디바이스를 제어할 때, 기본적인 입출력 루틴을 제공한다.

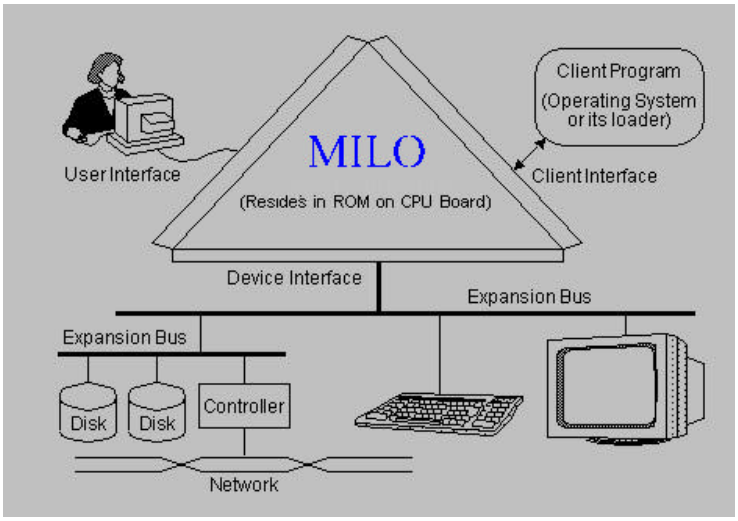
여러분이 컴퓨터를 맨처음 켜면, 보이는 화면이 이러한 BIOS의 시스템 체크하는 화면으로, IBM PC에서는 Phoenix의 BIOS가 유명하다. 맨처음 IBM PC가 나왔을 때, IBM에서는 IBM BIOS라는 것을 만들고, 나머지 시스템 운영 체제는 마이크로 소프트에서 만들게 되었다.

Phoenix BIOS와 AmiBIOS등 여러 업체에게 이러한 IBM BIOS와 호환이 되며, 조금 더 성능을 개선하여 만들기 시작하여, 이제 대부분의 PC에서는 이러한 IBM BIOS 호환 BIOS가 시장을 장악하고 있다.

현재 PC의 BIOS도 GNU 정신에 의해 개발되고 있는 것이 있는데, 대표적인 것이 바로 Alpha Linux에서의 MILO와 OpenBIOS 프로젝트가 있겠다.

I-1. Alpha MILO

MILO란 Miniloader의 약자로 알파용 리눅스에서 기존의 LILO와 비슷한 역할을 하는 리눅스 커널을 올리는 역할을 한다. 하지만 기존의 간단한 LILO와는 많이 틀리며, 거의 BIOS의 역할을 할수 있는 매우 방대한 펌웨어라고 할 수 있다.



[그림 1] MILO의 개념도

1. MILO의 소개

간단히 MILO의 역사를 뒤집어 보면 다음과 같다.

1. ftp://gatekeeper.dec.com/pub/Digital/Linux-Alpha/Miniloader/에 있는 READ.ME 파일에 의하면 Alpha Miniloader는 David A Rusling (david.rusling@reo.mts.dec.com)이 유지관리하고 있으며, v1.3 디렉트리는 초창기 리눅스 커널 버젼 1.3에서 만들어진 MILO를 말하며, v2.0은 커널 2.0로 안정된 커널에 기반한 MILO이며, v2.1은 개발 커널 2.1에 바탕을 둔 MILO를 말한다.

2. 또 http://www.piren.epita.fr/~dom/AXP-port-FAQ/history.html.en 에 의하면, 리눅스 커널의 알파로의 포팅은 1993년 커널 1.0대에 Jim Paradis가 이끄는 Alpha Migration Tools Group에 의해서 행해졌다고 한다. 그 뒤 Linus Torvals가 Digital의 senior manager인 John Hall로부터 Jensen기종의 알파 머신을 빌려서 자신이 직접 커널을 포팅하여 1994년 10월 24일 리눅스 커널 1.1.60에 include/asm-alpha라는 부 디렉토리가 생겼다.

3. 그 뒤, Dave Rusling(rusling@linux.reo.dec.com, 지금은 ARM에서 일하고 있으며, E-mail은 david.rusling@arm.com이다.)에 의해 MILO bootloader가 만들어지게 되고, Jay Eastabrook(jestabro@amt.tay1.dec.com)에 의해 XFree86이 64비트 Liux에 포팅되었다.

2. MILO 구성 및 역할

알파 리눅스의 경우 기존의 인텔 기반의 리눅스에 비해서 커널 이외에도 바이오스 역할을 하는 MILO라는 것이 필요하다.

이러한 알파 리눅스의 경우 바이오스 역할을 하는 것으로 이미 알파 시스템에 설치되어 있는 시스템 ROM에 있는 바이오스와 바이오스 비슷한 역할을 하는 MILO가 있다.

이 MILO에 대해서는 이미 리눅스 HOWTO가 있어서 많은 도움을 받을수 있을 것이다. 하지만 이 HOTWO라는 것이 주로 MILO 소스 코드 자체보다는 어떻게 MILO가 알파 리눅스에서 리눅스 커널을 올리느냐하는데 초점이 맞추어져 있어서 소스 코드 자체에 대한 자세한 설명이 부족한 편이다. 하지만 MILO를 이해하는데 좋은 자료가 되고 있다.

이 MILO HOWTO에서 설명하고 있는 MILO의 구성은 다음과 같다.

1. PALcode (주 : PALcode에 대해서는 다음에 자세히 설명할 것임.)
2. 메모리 셋업 코드 (페이지 테이블과 가상 메모리를 생성)
3. 비디오 코드 (BIOS 흉내내기 코드와 TGA (21030))
4. 리눅스 커널 코드. (인터럽트 핸들러등)
5. 리눅스 블록 디바이스 드라이버 (예를 들어 플로피 디스크용)
6. 파일 시스템 관련 코드 (ext2, MS-DOS 와 ISO9660),
7. 사용자 인터페이스 코드 (MILO)
8. 커널 인터페이스 코드 (HWRPB를 셋팅과 리눅스용 메모리 맵)
9. 환경변수를 위한 NVRAM 코드

MILO는 주로 시스템에 관련된 분야는 어셈블리어로 된 PALcode에서 다루고 나머지 부분은 일반적인 C 언어로 기술되고 있다.

3. MILO 로딩 방법

MILO를 시스템에 로딩하는 방법은 크게 2가지 방법이 있는데 이미 시스템에 설치되어 있는 펌웨어를 이용하는 방법과 아에 MILO 자체를 시스템의 펌웨어로 이용하는 방법이 있다.

다음은 알파 시스템 LX에서 MILO를 로딩하는 방법을 정리한 것이다.

시스템에 설치되어 있는 펌웨어를 이용하는 방법

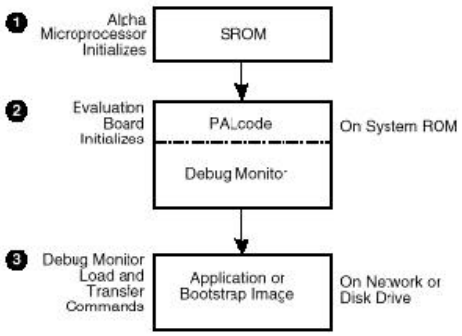
1. Windows NT ARC firmware : Windows NT용, 가장 쉽고 일반적
2. SRM Console : Digital UNIX 용 SRM 콘솔을 이용

MILO 자체를 시스템의 펌웨어로 이용하는 방법

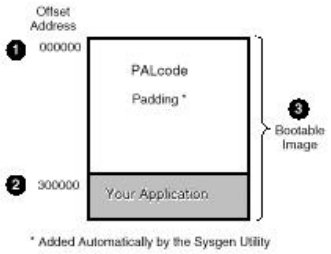
1. Evalutaion Board Debug Monitor : 평가보드의 디버그 모니터에서 로딩

2. Failsafe Boot Block Floppy : 부팅 디스켓을 사용하는 방법
3. Flash ROM : MILO를 아에 시스템의 Flash ROM에 넣는 방법

MILO 로딩하는 과정



[그림 2] MILO의 로딩 과정



[그림 3] MILO의 이미지 파일의 구조

4. PALcode의 배경

PALcode가 사실 MILO의 핵심 부분이라고 할 수 있으며, PALcode의 배경 지식을 간단하게 쌓아놓고 시작하기로 하자. 먼저 알파 CPU와 PALcode의 생성 배경을 알아보면 다음과 같다.

알파 CPU는 1988년 가을에 디지탈(현재는 컴팩)의 Dick Sites와 Rich Witek이 속한 컴퓨터 아키텍처 팀에서 시작된 프로젝트이다. 컴퓨터의 발전 속도를 예측해본 결과, 32비트 CPU로 앞으로 다가올 멀티미디어 시대의 대용량 파일을 처리하기에는 힘들 것이라 생각하고 10년에서 25년의 디자인 기간을 갖고 64비트 CPU를 만들기로 계획했다(이들은 어쩌면 20년 앞은 바라볼 수 있었지만 10년안에 회사가 망해 컴팩에 인수된다는 안목은 없었나 보다).

이렇듯 그들이 가장 자랑스럽게 여기는 기술 중 하나인 PALcode는 바로 10년 이상의 긴 프로젝트 기간을 거쳐 만들어진 것이었다. 알파 CPU에 OpenVMS, 유닉스, 윈도우 NT와 같은 다양한 운영체제가 간섭받지 않고 공존하게 된 것은 모두 PALcode 덕택으로 봐도 된

다. 이렇듯 CPU가 바뀜에 따라 애플리케이션을 다시 제작하는 엄청난 비용을 줄이기 위해 알파 아키텍처 엔지니어는 CPU 코어에 기존 다른 CPU 코어의 명령을 첨가하는 형태가 아니라 PALcode 명령을 이용한 소프트웨어 변환이라는 방법을 제안한 것이다.

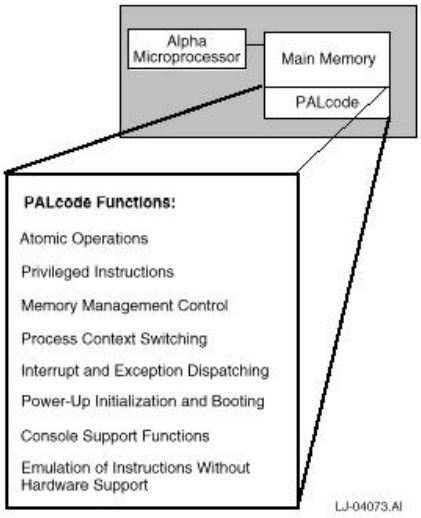
PALcode는 다양한 운영체제를 지원하기 위한 역할 외에도 다른 CPU 명령으로 작성된 애플리케이션을 쉽게 알파 명령으로 바꿔주는 번역기 기능에서도 중요한 역할을 하고 있다. 이와 같은 번역기로는 인텔용 윈도우 NT 명령을 알파용 윈도우 NT 명령으로 바꿔주는 FX32와 SPARC 명령으로 작성된 것을 알파용으로 바꿔주는 프리포트 익스프레스(Freeport Express)가 있다.

이처럼 알파 CPU는 단순히 몇 년내에 완성된 것이 아니라 적어도 10여년 간의 많은 노력이 결집된 것이다. 현재 초창기의 21064를 이용한 알파 AXP에서 21164를 이용한 알파 PC 164, 알파 164LX, 알파 164UX, 알파 164RX 등의 알파 시스템이 있다. 현재 삼성에서도 알파 시스템 생산에 나서 21164까지는 주로 디지탈 엔지니어가 설계한 메인보드를 이용했지만 21264부터는 삼성전자에서 자체적으로 설계한 알파 264DP라는 메인보드를 만들고 있다.

5. PALcode의 구조

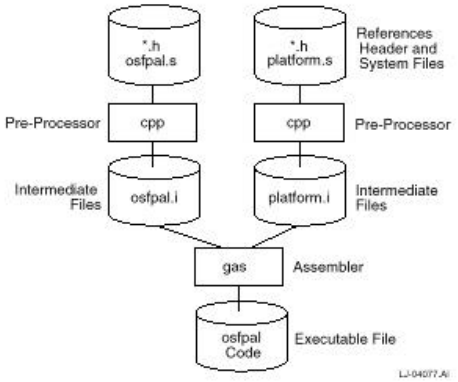
우선 PALcode의 역할을 다시 정리하면 다음과 같다.

1. Power-Up 초기화
2. memory 관리 제어
3. interrupt와 예외 처리
4. Privileged 명령 수행
5. x86 instruction emulation 지원



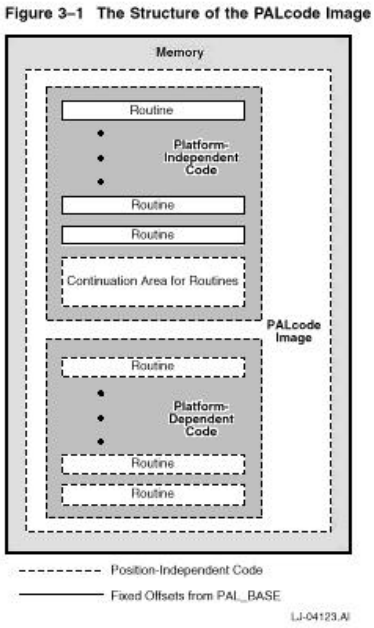
[그림 4] PALcode의 기능

이러한 기능을 하기 위해 PALcode는 다시 크게 비교적 시스템에 직접적으로 관계되지 않는 osfpal.S와 시스템에 밀접히 관련된 platform.S라는 파일로 이루어져 있다.



[그림 5] PALcode 만드는 과정

위의 그림에서와 같이 osfpal.S 와 platform.S가 PALcode의 핵심이라고 할 수 있다. 이렇게 만들어진 osfpal은 앞서 만든 milo와 같이 합쳐져서 실제 부팅할수 있는 이미지가 만들어진다.



[그림 6] PALcode 이미지

I-2. OpenBIOS 프로젝트

[그림 7] OpenBIOS 로고 (주석 : openbios.jpg)

리눅스 시스템을 가만히 보면 커널은 펜티엄, 스팍, 알파, ARM, MIPS등 많은 CPU등을 지원하고 막강한 성능을 가지고, 수많은 응용 프로그램들이 소스 코드 레벨에서 호환되어, 소스 코드만 공개되어 있으면 어떤 CPU에서도 지원이 되는 만능 운영체제라는 느낌이 든다. 하지만 가만히 보면 리눅스 시스템도 하드웨어 업체에서 제공하는 바이오스 위에서 돌아가는 하나의 프로그램일 뿐이라는 사실이다.

바이오스는 사실 CPU를 생산하는 하드웨어 생산 업체나 이를 뒷받침하는 전문업체에서만 다루는 베일에 싸인 처녀림(?)인 듯 싶다. 즉, 관심은 있으나 함부로 다가서기에는 두려운 그런 느낌인 것이다. 하지만 걱정하지 말자. 요즘 리눅스의 성공으로 공개 소프트웨어 정신이 퍼져서 인지, 이러한 처녀림에 뛰어 들겠다는 사람들이 있으니 말이다.

1. OpenBIOS 유사 프로젝트

이러한 OpenBIOS 프로젝트 비슷한 일을 하고 있는 사람들을 정리하면 다음과 같다.

1. OpenBIOS 프로젝트 : http://www.freiburg.linux.de/openbios/

OpenBIOS 프로젝트는 독일의 Stefan Reinaur에 의해서 추진중에 있으면 아직 버전 0.0.1과 0.0.2pre만이 있는 초보수준이다. 하지만, GPL 라이센스의 GNU 프로젝트로 추진중이어서 많은 개발자들이 참여하여 참신한 의견이 도출되고 있다.

2. OpenFirmware 프로젝트 : http://playground.Sun.COM/1275/home.html

이 프로젝트는 실제 썬(SUN) 사의 스팍 머신에서 1989년부터 쓰고 있는 Openboot라는 펌웨어에 기반을 두고 공개 프로젝트로 OpenFirmware를 만드는 프로젝트이다.
OpenFirmware는 PowerPC와 ARM 프로세서에 포팅되어 있어서 실제 애플의 파워 맥킨토쉬에서 사용중인 펌웨어이다. 하지만 OpenFirmware는 스펙은 공개되어 있으나, 현재 소스는 공개하고 있지 않으며, FirmWorks라는 상업회사와 썬, 애플, IBM등 대기업의 주도로 추진중에 있다.

3. GRUB 프로젝트 : http://www.uruk.org/grub/

이 프로젝트는 Erich Boleyn이라는 인텔에서 근무하는 시스템 엔지니어에 의해서 시작되었으며, 원래는 Mach커널을 올리는 멀티 부트로더를 만들려는 데에서 유래되었다. 이 프로젝트의 약자가 GRUB(GRand Unified Bootloader)라는 점이 알려주듯이 기존의 부트로더들이 매우 엉성하게 어느 한 CPU만 지원하도록 설계되었지만, GRUB은 매우 유연하게 설계되어서 여러 가지 운영체제의 커널을 올리는 부트로더로써 유용하게 사용되고 있다.

위에 여러 가지 펌웨어에 대해서 나열을 했지만, 아직은 모든 프로젝트들이 시작단계나 혹은 이제막 준비단계인 것들이 많다는 것을 알수 있을 것이다. 즉, 이제 막 걸음마를 하고 있는 중이다. 개인적으로 이러한 때에 우리나라의 프로그래머들의 활발한 참여가 아쉽다고 생각한다.

2. OpenBIOS 의 구조

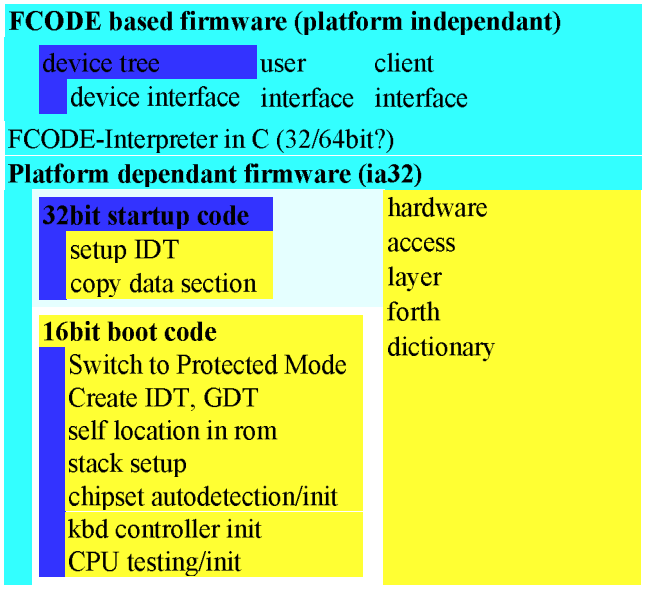
우선 OpenBIOS가 중요한 점은 기존의 리눅스에 있는 시스템 프로그램들이 모두 소프트웨어 중심으로 이루워져왔으나, 앞의 MILO와 OpenBIOS는 모두 ROM화를 목적으로 만들어진 진정한 펌웨어라고 할수 있다.

우선 ftp://alpha.secsm.org/pub/OpenBIOS나 /http://www.freiburg.linux.de/OpenBIOS/에서 OpenBIOS-0.0.2pre1.tar.gz를 받아다가 소스 코드를 보면 다음과 같다.

```
alpha [18]/home/bsjung/os/BIOS % tar zxvf OpenBIOS-0.0.2pre1.tar.gz
alpha [19]/home/bsjung/os/BIOS % cd OpenBIOS-0.0.2
alpha [20]/home/bsjung/os/BIOS/OpenBIOS-0.0.2 % ls -l
total 103
-rw-r--r--   1 bsjung   200         17982 Oct 28  1998 COPYING
-rw-r--r--   1 bsjung   200           571 Nov 13  1998 CREDITS
-rw-r--r--   1 bsjung   200          1425 Nov 27  1998 Config.mak
drwxr-xr-x   2 bsjung   200          1024 Nov 25  1998 Documentation/
-rw-r--r--   1 bsjung   200          5640 Nov 26  1998 Make.log
-rw-r--r--   1 bsjung   200          1899 Nov 27  1998 Makefile
-rw-r--r--   1 bsjung   200           335 Nov 13  1998 README
-rw-r--r--   1 bsjung   200           998 Nov 21  1998 Rules.mak
drwxr-xr-x   5 bsjung   200          1024 Nov 27  1998 boot/
drwxr-xr-x   5 bsjung   200          1024 Nov 21  1998 drivers/
drwxr-xr-x   5 bsjung   200          1024 Nov 27  1998 firmware/
drwxr-xr-x   2 bsjung   200          1024 Nov 12  1998 include/
-rw-r--r--   1 bsjung   200         65536 Nov 27  1998 rom.bin
drwxr-xr-x   4 bsjung   200          1024 Nov 27  1998 scripts/
alpha [21]/home/bsjung/os/BIOS/OpenBIOS-0.0.2 %
```

위 디렉토리중 필자에서 가장 눈에 띄는 것은 rom.bin과 boot이라는 디렉토리인데, rom.bin은 이미 ROM에 구울수있게 만들어진 이진 파일이고, boot 디렉토리는 boot16과 boot32와 몇가지 툴들로 구성되어 있어서, 처음 시스템이 부팅할 때 CPU가 16비트인 Real Mode에서 동작할 때 실행되는 boot16 코드와 다음 Protected Mode에서 실행되는 32비트 boot32 코드가 있다.

OpenBIOS 프로젝트의 디자인 원칙은 현재 진행중인 OpenFirmware 스펙을 따르고 있으므로, OpenFirmware의 구조를 보면 이해가 되는데 OpenFirmware의 구조는 다음과 같다.



[그림 8] OpenFirmware의 구조

위의 그림에서와 같이 OpenFirmware는 플렛폼에 관계없는 부분은 Fcode로 작성되고, 나머지 플렛폼에 관계되는 부분은 16비트 부트 코드와 32비트 시작 코드와 몇가지 하드웨어 드라이버들로 구성되어 있다.

현재 OpenBIOS도 이와같은 구조로 다시 디자인되고 있으며, Fcode의 전신인 forth를 이용하여 플렛폼에 관계없는 부분은 forth를 사용하고, 플렛폼에 관계있는 부분은 어셈블리어와 C언어로 작성될 예정이다.

3. OpenBIOS 의 동작

우선 OpenBIOS의 동작으로 보면 boot16에서 부팅을 한다음, boot32에서 시스템 셋업을 하고, firmware에 있는 init 디렉토리에서 각종 PCI 디바이스들과 IDE 콘트롤러를 초기화한다.

이것을 이해하기 위해서는 일반적으로 Pentium 매뉴얼을 보면 쉽게 알수 있는데, 다음은 인텔의 CPU들이 RESET될때의 각 레지스터들의 초기값을 나타내는 것이다.

REGISTER	REGISTER NAME	INITIALIZED CONTENTS	COMMENTS
EAX	Accumulator	xxxx xxxxh	0000 0000h indicates self-test passed.
EBX	Base	xxxx xxxxh	
ECX	Count	xxxx xxxxh	
EDX	Data	06 + Device ID	Device ID = 51h or 59h (2X clock) Device ID = 55h or 5Ah (2.5X clock) Device ID = 53h or 5Bh (3X clock) Device ID = 54h or 5Ch (3.5X clock)
EBP	Base Pointer	xxxx xxxxh	
ESI	Source Index	xxxx xxxxh	
EDI	Destination Index	xxxx xxxxh	
ESP	Stack Pointer	xxxx xxxxh	
EFLAGS	Flag Word	0000 0002h	
EIP	Instruction Pointer	0000 FFF0h	
ES	Extra Segment	0000h	Base address set to 0000 0000h. Limit set to FFFFh.
CS	Code Segment	F000h	Base address set to FFFF 0000h. Limit set to FFFFh.
SS	Stack Segment	0000h	Base address set to 0000 0000h. Limit set to FFFFh.
DS	Data Segment	0000h	Base address set to 0000 0000h. Limit set to FFFFh.
FS	Extra Segment	0000h	Base address set to 0000 0000h. Limit set to FFFFh.
GS	Extra Segment	0000h	Base address set to 0000 0000h. Limit set to FFFFh.
IDTR	Interrupt Descriptor Table Register	Base = 0, Limit = 3FFh	
GDTR	Global Descriptor Table Register	xxxx xxxxh, xxxxh	
LDTR	Local Descriptor Table Register	xxxx xxxxh, xxxxh	
TR	Task Register	xxxxh	
CR0	Machine Status Word	6000 0010h	
CR2	Control Register 2	xxxx xxxxh	
CR3	Control Register 3	xxxx xxxxh	
CR4	Control Register 4	0000 0000h	
CCR (0-6)	Configuration Control (0-6)	00h CCR(0-3, 5-6) 80h CCR4	
ARR (0-7)	Address Region Registers (0-7)	00h	
RCR (0-7)	Region Control Registers (0-7)	00h	
DR7	Debug Register 7	0000 0400h	

Note: x = Undefined value

[표 1] 인텔 펜티엄 레지스터 초기화

위의 그림에서 보는바와 같이 CPU가 RESET신호로 초기화될 때 가르키는 메모리는 코드 세그먼트를 가르키는 CS 레지스터와 인스트럭션 포인터 레지스트인 EIP로 결정되는데, CS 레지스터는 0xFFFF 0000h 번지에서 0xF000h을 더한값으로 셋팅되어 있고, EIP는 0x0000 FFF0h로 지정된 것을 알수 있다. 즉 이 둘의 조합인 0xFFFF FFF0h번지에 ROM BIOS의 초기루틴이 있게 된다.

이러한 CPU 초기루틴이 있는 곳은 boot16코드안에 있게 된다.

```
SEGMENT .text
BITS 16

; entry point into startup code - the bootstrap will vector
; here with a near JMP generated by the builder. This
; label must be in the top 64K of linear memory.
global          _main
_main:
                jmp startup

sig:            db 0xde, 0xad, 0xbe, 0xef

startup:

        mov     bx, cs
        mov     ds, bx
        xor     bx, bx          ; es:bx == 0000:0000
        mov     es, bx

        mov     fs, ax          ; Might contain BIST result
        mov     gs, dx          ; CPU model/rev info

      CPU_TEST

      PORT_80 0x01
```

[리스트 1] boot16/start.asm 중에서 처음 시작부분

즉 위의 0xFFFF FFF0에 있는 ROM BIOS 시작부분은 jmp 인스트럭션으로 실제 ROM BIOS가 시작되는 startup에 위치하게 된다.

다음은 위 boot16에 있는 초기 부팅 코드를 ROM BIOS 영역의 초기 주소를 0xf0000으로 잡고, 16비트 어셈블리어로 된 start.asm을 컴파일하는 과정이다.

```
alpha [38]/home/bsjung/os/BIOS/OpenBIOS-0.0.2/boot/boot16 % ll
total 23
-rw-r--r--    1 bsjung    200             1047 Mar  5 03:56 Makefile
-rw-r--r--    1 bsjung    200             2043 Mar  5 03:56 boot.inc
-rw-r--r--    1 bsjung    200             5083 Mar  5 03:56 generic.inc
-rw-r--r--    1 bsjung    200             1522 Mar  5 03:56 ia32.inc
-rw-r--r--    1 bsjung    200             5493 Mar  5 03:56 start.asm
-rw-r--r--    1 bsjung    200             5493 Mar  5 03:56 start.asm.bak
alpha [39]/home/bsjung/os/BIOS/OpenBIOS-0.0.2/boot/boot16 % make
nasm -f as86
```

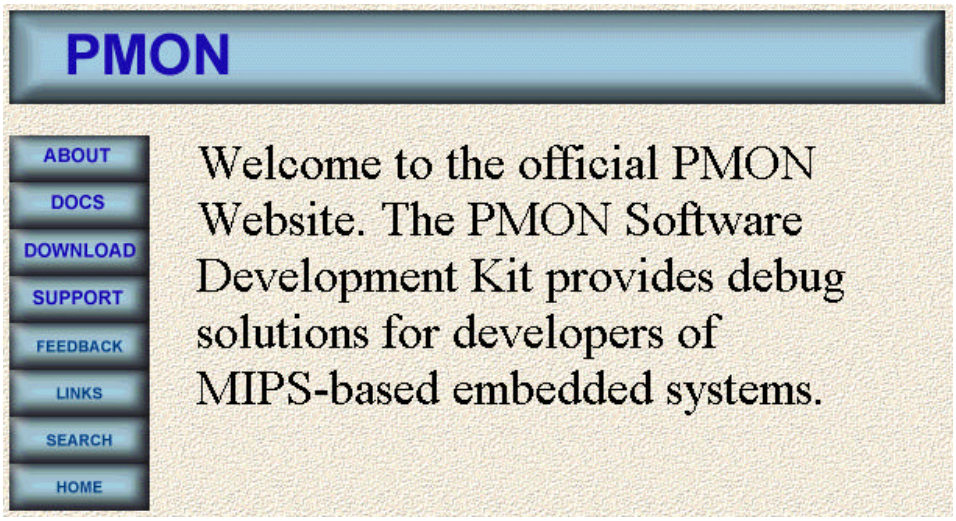
-i/home/bsjung/os/BIOS/OpenBIOS-0.0.2/drivers/chipset/acc1287/
-dCPU_486=1 -dROMBASE=0xf0000 start.asm
cp -f start.o code16.o
alpha [40]/home/bsjung/os/BIOS/OpenBIOS-0.0.2/boot/boot16 %

4. OpenBIOS 의 활용

실제 OpenBIOS가 활용되는 분야는 클러스트링(Clustering) 분야로 현재 미국의 LANL(Los Alamos National Laboratory) 에 있는 ACL(Advanced Computing Laboratory) 에서 Bewolf 프로젝트에 Linuxbios라는 OpenBIOS와 리눅스 커널을 합쳐놓은 것으로 인텔과 알파로 된 Bewolf에 적용하고 있다.

즉, OpenBIOS를 활용하여, 지금까지 DOS 1.0을 지원하기 위해서 BIOS에 쓸데없이 들어가 있던 각종 테스트들을 없애고, 클러스터링에 맞는 네트워크 부팅을 지원하기 위해서, OpenBIOS 코드와 리눅스 커널 코드를 이용하고 있는 것이다.

II. PROM 모니터 프로그램 : PMON



[그림 9] PMON 초기 홈페이지 (http://www.carmel.com/pmon)

위에 있는 PMON는 PROM Monitor의 줄임말로, MIPS용 모니터 프로그램이다. 모니터 프로그램이기보다는 ICE(Incircuit Emulator)에 가까울정도로 그 기능이 막강하다. 시리얼 프로그램 다운로딩을 비롯하여, Ethernet을 이용한 프로그램 다운로딩, 프로그램 제어등 어떠한 모니터 프로그램보다 다양한 기능을 가지고 있다. 게다가 PMON의 소스코드도 공개되어 있어, 마음껏 수정할수도 있다.

PMON는 LSI Logic에서 공식적으로 MIPS 평가판 보드에 쓰고 있는 실정이며, 이것을

만든 전 LSI Logic의 엔지니어에 의해 소스 코드를 볼수 있도록 인터넷에 공개하고 있다. 또 그 소스 코드를 어떠한 저작권의 제한없이 쓸수 있다는 장점이 있다. 심지어 GNU 저작권에도 제한을 받지 않는다.

1. PMON 의 소개

PMON는 Phil Bunce에 의해 MIPS용으로 만들어진 PROM 모니터 프로그램이다.

PMON는 다음의 3가지 다른 디버깅 환경으로 되어 있다.

1. PMON : 가장 일반적인 어셈블리 레벨의 디버깅 모니터이다. 하지만 단점은 약 300KB의 용량이 필요하다는 것이고, 시리얼 포트나 Ethernet으로 프로그램을 다운로딩할 수 있다.

2. SerialICE-1A : PMON와 기능을 같으며, 보드에 약 1KB의 메모리 용량밖에 필요치 않는다는 장점이 있다. 하지만, 보드에 특별한 SerialICE 컨트롤 보드가 필요하다는 점이다.

3. SerialICE-1B : SerialICE-1A와 기능을 같으며, SerialICE 컨트롤 보드가 필요없는 장점이 있다. 하지만 Win95/NT만 지원을 한다.

PMON은 http://www.carmel.com/pmon/pmon5.html 이나, ftp://alpha.secsm.org/pub/PMON에 가면 있다. pmon5.tar.gz을 풀어보면 다음과 같은 디렉토리가 있다.

```
alpha [21]/home/bsjung/os/PMON % tar zxvf pmon5.tar.gz
alpha [22]/home/bsjung/os/PMON % cd PMON
alpha [23]/home/bsjung/os/PMON/PMON % ls -l
total 141
-rw-r--r--   1 bsjung   200          1033 Apr 13  1999 Install
-rw-r--r--   1 bsjung   200          6039 Apr 13  1999 Makefile
-rw-r--r--   1 bsjung   200          1485 Apr 13  1999 README
drwxr-xr-x   2 bsjung   200          1024 Feb 27 03:42 bsps/
drwxr-xr-x   2 bsjung   200          1024 Feb 27 03:42 examples/
drwxr-xr-x   4 bsjung   200          3072 Feb 27 03:47 html/
drwxr-xr-x   2 bsjung   200          1024 Feb 27 03:42 imon/
drwxr-xr-x   2 bsjung   200          1024 Feb 27 03:42 include/
-rw-r--r--   1 bsjung   200          1982 Apr 13  1999 index.htm
-rw-r--r--   1 bsjung   200        112882 Apr 13  1999 install.c
drwxr-xr-x   2 bsjung   200          5120 Feb 27 03:42 lib/
drwxr-xr-x   2 bsjung   200          1024 Feb 27 03:42 libsa/
drwxr-xr-x   2 bsjung   200          1024 Feb 27 03:42 mon/
drwxr-xr-x   2 bsjung   200          1024 Feb 27 03:42 pmon/
```

```
drwxr-xr-x   2 bsjung   200          1024 Feb 27 03:42 tools/
-rw-r--r--   1 bsjung   200             8 Apr 13  1999 version
alpha [24]/home/bsjung/os/PMON/PMON %
```

위 디렉토리를 보면, PMON5는 다음과 같이 이루어져 있음을 알 수 있다.

imon : SerialICE 모니터에 관련된 소스코드들이 있다.
pmon : PMON에 관계된 소스코드들이 있다.
mon : 여기는 PMON와 SerialICE 모두에 관계되는 메인 소스 코드들이 있다.
lib : 여기는 C 런타임 라이브러리가 있다.
bsps : 여기는 SerialICE-1에 필요한 커널과 드라이버들이 있다.
include : 여기는 이 패키지에 필요한 여러 가지 헤더 파일들이 있다.
examples : 여기는 PMON와 SerialICE 아래에서 실행가능한 예제 파일들이 있다.
tools : 여기는 호스트 컴퓨터에서 실행되는 여러 가지 툴들이 있다.

2. PMON의 사용법

PMON을 쓰는 방법은 다음의 3가지 스텝을 따른다.

1. 목적 보드에 맞는 프로그램을 호스트에서 크로스 컴파일러로 작성한다.

2. 목적 보드와 호스트 컴퓨터를 RS232나 Ethernet으로 연결한다.

3. 호스트 컴퓨터에서, 시리얼 제어 프로그램를 실행하여, PMON를 이용하여,
   작성된 프로그램을 다운로딩하여 실행한다.

PMON의 명령어들은 다음의 4가지 부류가 있다.

1. 실행 컨트롤
   - b : breakpoint를 설정할 때 사용한다.
   - when : 복잡한 breakpoint할 때 사용한다.
   - c : breakpoint로 멈춘후 프로그램을 다시 계속 수행하게 한다.
   - call : function을 실행한다.
   - db : breakpoint를 보이거나 지운다.
   - g : 프로그램을 실행한다.
   - t : trace 프로시져를 초기화한다.
   - to : trace 프로시져를 초기화한다.
     (t와 다른점은 프로시져을 하나의 스텝으로 본다.)

2. 보기와 수정하기
   - copy : 지정된 주소의 바이트들을 복사한다.

d : 메모리에 있는 내용을 16진수나 ASCII형식으로 출력한다.
dump : 모토롤라 S코드로 된 것을 호스트포트에 업로드한다.
fill : 메모리에 16진수를 써넣는다.
l : 메모리에 있는 내용을 역어셈브리해서 보인다.
load : 프로그램을 다운로딩한다.
m : 메모리에 있는 내용을 보이거나 수정한다.
r : 레지스터의 내용을 보이거나 수정한다.
search : 메모리에서 특정 패턴을 찾는다.

3. 환경설정

hi : 지금까지 친 명령어들을 보인다.
ls : 현재의 symbol 테이블에서 symbol들을 리스트한다.
more : 사용자의 입력에 screen-at-a-time 컨트롤
set : 환경 변수를 셋팅한다.
sh : 명령어 쉘를 실행한다.
stty : 터미널 옵션을 셋팅하거나 보인다.

4. 그외의 것들

debug : 모니터의 dbx 모드를 초기화한다.
flush : 데이터와 인스트럭션 캐쉬를 초기화한다.
h : 도움말
mt : 메모리를 테스트한다.
tr : transparent 모드를 선택한다.

다운로딩은 직렬 포트로 하는 경우와 Ethernet 포트를 이용하는 방법이 있다.

다음으로 실제로 직렬 포트를 이용하여 파일을 다운로딩하는 과정이다.

1. 호스트에서 직렬 포트를 제어하는 프로그램을 실행한다.
% tip -19200 hardwire

2. 타겟 보드의 포트를 초기화한다.
PMON> set hostport tty0

3. echo 시그널을 다운로딩하는 옵션을 off로 해 놓는다.
PMON> set dlecho off

4. 다운로딩할 때 프로토콜을 사용하지 않는다는 의미이다.
PMON> set dlproto none

5. bubble.bin 이라는 미리 컴파일된 바이너리 파일을 다운로딩한다.

PMON> load -c "cat bubble.bin"

6. bubble 이라는 프로그램을 실행한다.
PMON> g Run the downloaded program.

다음으로 Ethernet 포트를 이용하여 파일을 다운로딩하는 과정이다.

1. /etc/hosts에 목적 보드의 IP 주소를 셋팅한다.
210.118.74.129 beta

2. 하드웨어 Ethernet 주소를 셋팅한다.
PMON> **set etheraddr 08:00:69:03:00:00**

3. arp 명령어로 목적 보드의 Ethernet 주소를 호스트 컴퓨터의 arp 테이블에 추가한다.
 # arp -s beta 08:00:69:03:00:00

4. 이제 호스트포트를 Ethernet으로 설정한다.
PMON> set hostport ethernet

5. load를 실해하여, 목적 보드가 프로그램을 다운로딩할 준비를 한다.
PMON> load

6. 이제 호스트 컴퓨터에서 tftp를 사용하여 파일을 다운로딩한다.
% tftp beta
tftp> put bubble.bin
tftp> quit

7. 프로그램을 실행한다.
PMON> g

III. 내장형 커널 : ELKS

1.ELKS에 대하여

[그림 10] ELKS 로고

XT에서 리눅스를 쓰자!

ELKS를 만드는 계기는 XT(처음 나온 IBM 컴퓨터)나 AT에서 리눅스를 쓰려는 사람들을 위해서 이다. 일명 작은 리눅스인데, 이러한 작은 리눅스는 내장형 시스템에 알맞아서 많은 사람들이 XT뿐만 아니라, PDA나 라우터나 인터넷 셋탑 박스등에 응용하려고 하고 있다.

내장형 시스템을 위한 전용 리눅스 커널

ELKS는 내장형 시스템을 위한 전용 리눅스 커널이다. 이 커널은 기존의 리눅스 커널이 8MB이상의 메모리를 요구하던 것을 500KB 이내로 줄인 것이다. 아주 작은 크기의 리눅스 커널이기 때문에 내장형 시스템에 알맞지만, 현재 프로젝트 초기 단계이기 때문에 i86계열에만 포팅되어 있으며, 아직 16비트에만 머물러 있다. 그래서, 현재 32비트 위주의 내장형 시스템 시장에 적용하려면 많은 노력이 요구되고 있다.

2.ELKS커널구조

ELKS 커널을 보면, 리누스 토발즈가 처음 MINIX를 가지고 리눅스 커널 1.0을 만들때의 상황을 상상할 수 있게 한다. 즉, 기본적인 것 이외에는 아무것도 없다.

하지만, 리누스가 처음 리눅스를 포팅할 때 i386를 기본으로 하여, 32비트 CPU를 목표로 삼았지만, ELKS는 이보다 더 작은 CPU인 8086이상의 16비트 CPU를 목표로 삼고 있다.

ELKS가 있는 곳

일단 ELKS 공식 버전은 다음의 URL에서 구할 수 있다.

1. ftp://linux.mit.edu/pub/ELKS/
2. ftp://alpha.secsm.org/pub/ELKS/

위의 URL에서 구할수 있는 테스트 가능한 ELKS는 버전이 0.0.66까지 있으며, 현재 최신 버전은 CVS 서버에 있다. 2000년 1월 8일의 최신 버전은

ftp://alpha.secsm.org/pub/ELKS/CVS

에 있으며, 이것을 받아 보기 바란다.

ELKS의 커널 디렉토리

위의 CVS에 있는 elks-0_0_74-pre1.tar.gz을 받아서 ELKS 커널 구조를 보면 다음과 같은 구조로 되어 있다.

1. arch/

2. kernel/
3. fs/
4. lib/
5. scripts/
6. include/
7. modules/
8. Documentation/
9. init/
10. net/

위의 arch 디렉토리에는 현재 i86 밖에 없다. (즉, 16비트의 인텔 CPU 밖에 지원하지 않는다. 하지만, 현재 내부적으로 M68K와 MIPS용으로 ELKS를 포팅중에 있으니, 관심 있는 분은 연락바란다.)

ELKS의 커널에 있는 것들

간단히 ELKS에 있는 주요 요소를 보면 다음과 같다.

1. 커널
   i86 디렉토리에 8086을 부팅하는 코드가 있다.

2. 파일 시스템
   minix : MINIX 파일 시스템을 지원한다.
   elksfs : ELKS 자체 파일 시스템을 말한다.
   romfs : 내장형 시스템에 적합한 롬 파일 시스템이다.

3. 디바이스 드라이버
   serial : 시리얼 드라이버를 지원한다.
   parallel : 패러렐 드라이버를 지원한다.

3.ELKS부팅과정

ELKS의 부팅은 16비트 도스의 부팅과정과 거의 흡사하다.

ELKS의 부팅 과정

ELKS가 부팅이 되면 80x86 CPU가 RESET이 되어 리얼 모드에서 0xFFFF0에 있는 ROM-BIOS를 실행하게 된다. 이 ROM-BIOS에는 시스템이 잘 작동되는 가를 체크하고 (Power on Test), 부팅할 수 있는 디바이스의 첫 번째 섹터(일명 부팅 섹터)를 0x7C00으로 로딩한다.

그러면 0x7C00에 있던 arch/i86/boot/bootsect.S 가 하는 일은 자기 자신을 0x90000으로 옮기고, 0x90000으로 점프한다. 그리고, 0x90200에 arch/i86/boot/setup.S를 로딩한다. 그리고, 나머지 커널은 0x10000에 로딩한다. 그 다음에 모든 권한을 setup.S에게 넘긴다.

bootsect.S가 하는 일

이 과정을 정리하면 다음과 같다.

arch/i86/boot/bootsect.S (부팅 섹터에 있는 처음 512 바이트)

1. 자신을 0x90000으로 옮기고, 거기로 점프한다.
2. 0x90200에 setup.S를 로딩한다.
3. 나머지 커널을 0x10000으로 로딩한다.
4. 이제 setup.S이 있는 곳으로 점프한다.

이제 bootsect.S 소스를 보면서 앞의 과정을 살펴보자.

```
#include <linuxmt/config.h>   ! 시스템에 대한 정보가 있다.
```

위 config.h에는 다음과 같이 정의가 되어 있다.

```
#define DEF_INITSEG     0x0100
#define DEF_SYSSEG      0x1000
#define DEF_SETUPSEG 0x0120
#define DEF_SYSSIZE     0x2F00

/* internal svga startup constants */
#define NORMAL_VGA      0xffff          /* 80x25 mode */
#define EXTENDED_VGA 0xfffe             /* 80x50 mode */
#define ASK_VGA 0xfffd          /* ask for it at bootup */
```

```
.text

SETUPSECS = 4
BOOTSEG   = 0x07C0  ! PC-BIOS가 부르는 Bootsector의 주소가 있다.
INITSEG   = DEF_INITSEG     ! 즉 0x0100에서 we move boot here - out of the way
SETUPSEG  = DEF_SETUPSEG
SYSSEG    = DEF_SYSSEG      ! 시스템이  0x10000 로 로딩된다.
                            ! 커널이 있는 곳이다.
SYSSEGB   = DEF_SYSSEG + 2
```

생략

```
! ld86 requires an entry symbol. This may as well be the usual one.
.globl  _main     ! 프로그램의 시작이다.
_main:
#if 0 /* hook for debugger, harmless unless BIOS is fussy (old HP) */
    int 3
#endif
    mov     ax,#BOOTSEG
    mov     ds,ax
    mov     ax,#INITSEG
    mov     es,ax
    mov     cx,#256
    xor si,si
    xor di,di
    cld
    rep
    movsw
    jmpi    go,INITSEG

! ax and es already contain INITSEG

go: mov     di,#0x4000-12
! 0x4000 is arbitrary value >= length of
! bootsect + length of setup + room for stack
! 12 is disk parm size

! bde - changed 0xff00 to 0x4000 to use debugger at 0x6400 up (bde).  We
! wouldn't have to worry about this if we checked the top of memory.  Also
! my BIOS can be configured to put the wini drive tables in high memory
! instead of in the vector table.  The old stack might have clobbered the
! drive table.

    mov     ds,ax
    mov     ss,ax           ! put stack at INITSEG:0x4000-12.
    mov     sp,di
/*
 *  Many BIOS's default disk parameter tables will not
 *  recognize multi-sector reads beyond the maximum sector number
 *  specified in the default diskette parameter tables - this may
 *  mean 7 sectors in some cases.
```

```
 *
 *  Since single sector reads are slow and out of the question,
 *  we must take care of this by creating new parameter tables
 *  (for the first disk) in RAM.  We will set the maximum sector
 *  count to 36 - the most we will encounter on an ED 2.88.
 *
 *  High doesn't hurt.  Low does.
 *
 *  Segments are as follows: ds=es=ss=cs - INITSEG,
 */

    mov     bx,#0x78
! 0:bx is parameter table address
    push    ds
    push    es
    xor     ax,ax
    mov     es,ax                   !
    seg es
    lds si,[bx]
    pop es

! ds:si is source

    mov     cl,#6
! copy 12 bytes
    cld
    push    di

    rep
    movsw

    pop di
    pop ds
                                    ! what kind of instruction is this?
    movb    4[di],*36               ! patch sector count

    push    ds
!   xor ax,ax                       ax still 0
    mov     ds,ax
    mov     [bx],di
    mov     2[bx],es
```

```
	pop ds

! load the setup-sectors directly after the bootblock.
! Note that 'es' is already set up.
! Also cx is 0 from rep movsw above.

load_setup:                     ! Setup 섹터를 로딩한다.
	xor ah,ah               ! reset FDC
	xor dl,dl
	int  0x13

	xor dx, dx              ! drive 0, head 0
	mov     cx,#0x0002              ! sector 2, track 0
	mov     bx,#0x0200              ! address = 512, in INITSEG
	mov     ah,#0x02        ! service 2, nr of sectors
	mov     al,setup_sects          ! (assume all on head 0, track 0)
	int  0x13               ! read it
	jnc ok_load_setup       ! ok - continue
                        ! Setup 섹터를 모두 로딩하였다면 ok_load_setup으로 점프한다.

	push    ax                      ! dump error code
	call print_nl
	mov     bp, sp
	call print_hex
	pop ax

	jmp  load_setup
```

ok_load_setup :

```
! Get disk drive parameters, specifically nr of sectors/track

#if 0

! bde - the Phoenix BIOS manual says function 0x08 only works for fixed
! disks.  It doesn't work for one of my BIOS's (1987 Award).  It was
! fatal not to check the error code.

	xor dl,dl
	mov     ah,#0x08        ! AH=8 is get drive parameters
	int  0x13
```

```
    xor ch,ch
#else

! It seems that there is no BIOS call to get the number of sectors.  Guess
! 36 sectors if sector 36 can be read, 18 sectors if sector 18 can be read,
! 15 if sector 15 can be read.  Otherwise guess 9.

    mov     si,#disksizes           ! table of sizes to try

probe_loop:
    lodsb
    cbw                         ! extend to word
    mov     sectors, ax
    cmpsi,#disksizes_end
    jae got_sectors             ! if all else fails, try 9
    xchg    ax, cx                  ! cx = track and sector
    xor dx, dx                  ! drive 0, head 0
    xor bl, bl
    mov     bh,setup_sects
    inc bh
    shl bh,#1                   ! address after setup (es = cs)
    mov     ax,#0x0201              ! service 2, 1 sector
    int 0x13
    jc  probe_loop              ! try next value

#endif

    mov     cx,#msg1end-msg1
    mov     si,#msg1
nxt_chr:
    lodsb
    call print_chr
    loop nxt_chr


! ok, we've written the message, now
! we want to load the system (at 0x10000)

    mov     ax,#SYSSEG
    mov     es,ax           ! segment of 0x10000
    call read_it        ! 시스템(커널)을 0x10000에 올린다.
```

```
call kill_motor
mov     al,#':
call    print_chr
call print_nl

jmpi    0,SETUPSEG ! setup.S 가 있는 곳으로 점프한다.
```

생략

[리스트 1] bootsect.S 분석

setup.S가 하는 일

일단 bootsect.S에서 setup.S로 넘어오면, setup.S가 하는 일은 시스템을 초기화 하는 일인데, 메모리나 디스크, VGA 타입, CPU 종류등을 BIOS로부터 알아내어, 0x90000-0x901FF에 저장한다. 그리고, 0x10020 번지로 점프한다. 왜냐하면, 커널은 0x10000 번지에 있지만, 거기에는 커널 COFF 이미지의 헤더파일이 있는데, 헤더 파일 크기가 32바이트이기 때문에, 0x10020번지부터 실질적인 커널이 시작하는 것이다.

이곳이 ELKS와 기존의 리눅스 커널과의 차이점이 있는데, 리눅스 커널은 setup.S에서 하는 일이 조금 다른데, 초기화후에 곧장 0x10000번지에 있는 리눅스 커널로 점프하지 않고, 커널 전체를 0x1000번지로 옮긴다음에, real 모드에서 protected 모드로 전환한다. 이때 0x1000번지에 있는 커널은 zBoot/head.S 라는 것으로 시작하는데, 이것은 레지스터들을 초기화하고, decompress_kernel()을 호출하여, 압축이 해제된 커널 전체를 0x100000 번지에 옮겨놓는다. 0x100000은 1MB이상으로 즉, 전체 시스템의 메모리가 1MB 이상이 되어서 우리가 원하는 내장형 시스템에는 적합치 않는 리눅스 커널이 된다.

그럼 앞의 setup.S 가 하는 일을 간단히 정리하면 다음과 같다.

arch/i86/boot/setup.S

1. 시스템 초기화하여 그 파라미터들을 0x90000-0x901FF에 저장한다.
2. ELKS 커널이 있는 0x10020으로 점프한다.

2.ELKS커널

ELKS 커널을 자세히 보려면, 일단은 ELKS 커널이 있는 /usr/src/elks의 kernel 디렉토리를 보면 된다.

일단 부팅이 끝나면 ELKS 커널의 주도권은 다음의 start_kernel()이라는 함수에 의해서 장악된다. 즉, 이제부터는 지겨운 어셈블리어가 아닌 순수 C 코드가 나타나게 되는 것이다. 암

흑과도 같은 Chaos의 시대에서 질서의 시대로 가는 것이다.

이 start_kernel()은 init/main.c에 있으며, 그 내용은 다음과 같다.

```
/*
 *      For the moment this routine _MUST_ come first.
 */

void start_kernel()
{
        __u16 a;
        __pregisters set;
        seg_t base,end;

/*      We set the scheduler up as task #0, and this as task #1 */

(1)     setup_arch(&base, &end);
(2)     mm_init(base, end);
(3)     init_IRQ();
(4)     init_console();
/*      calibrate_delay(); */
(5)     setup_mm();             /* Architecture specifics */
(6)     tty_init();
#ifndef CONFIG_NOFS
(7)     buffer_init();
#endif
#ifdef CONFIG_SOCKET
(8)     sock_init();
#endif
(9)     device_setup();
        inode_init();
#ifndef CONFIG_NOFS
(10)    fs_init();
#endif
(11)    sched_init();
        printk("ELKS version %s\n", system_utsname.release );
        task[0].t_kstackm = KSTACK_MAGIC;
        task[0].next_run = task[0].prev_run = &task[0];
        kfork_proc(&task[1],init_task);

        /*
```

```
         * We are now the idle task. We won't run unless no other
         * process can run
         */
(12)     while (1) {
                schedule();
         }
}
```

[리스트 1] start_kernel() 함수

앞의 내용을 살펴보면 다음과 같다.

(1) setup_arch() : 각 아키텍쳐에 맞는 초기화 루틴이 있으며, 아키텍쳐에 관계된 부분이므로, arch/i86/kernel/system.c에 구현이 되어 있다.

(2) mm_init() : 메모리 메니저 초기화 루틴으로, arch/i86/mm/malloc.c에 구현이 되어 있다.

(3) init_IRQ() : IRQ 초기화 루틴으로, arch/i86/kernel/irq.c에 구현이 되어 있다.

(4) init_console() : 콘솔 초기화 루틴으로, arch/i86/drivers/char에 구현이 되어 있다.

(5) setup_mm() : 메모리 메니저 셋업 루틴으로, arch/i86/mm/init.c에 구현이 되어 있다.

(6) tty_init() : 터미널 디바이스 초기화 루틴으로, arch/i86/drivers/char/tty.c에 구현이 되어 있다.

(7) buffer_init() : 파일 시스템의 버퍼를 초기화하는 루틴으로, fs/buffer.c에 구현이 되어 있다.

(8) sock_init() : 소켓을 초기화하는 루틴으로, net/socket.c에 구현이 되어 있다.

(9) device_setup() : 블록 디바이스 초기화하는 루틴으로, arch/i86/drivers/block/genhd.c에 구현이 되어 있다.
inode_init() : 파일 시스템의 i-node를 초기화하는 루틴으로, fs/inode.c에 구현이 되어 있다.

(10) fs_init() : 파일 시스템을 초기화 하는 루틴으로, fs/filesystems.c에 구현이 되어 있다.

(11) sched_init() : 프로세스 스케쥴링 데몬을 초기화 하는 루틴으로, kernel/sched.c에 구현이 되어 있다.

(12) schdule() : 실제 프로세스들을 스케쥴링하는 루틴으로, kernel/sched.c에 구현이

되어 있다.

위와 같이 start_kernel()은 아키텍쳐에 대한 초기화와 메모리 관리자 초기화, 파일 시스템 및 디바이스들의 초기화를 마치고, 무한히 프로세스들을 스케줄링하면서, 컴퓨터 전원이 꺼질 때까지 살아 있게 된다.

맺음말

이상으로 리눅스에서 현재 진행중인 펌웨어 프로젝트에 대해서 잠시 알아보았다. 펌웨어는 CPU 메인 보드에 들어가 있어서, 선뜩 프로그래머들이 다가서기가 어려운 분야지만, 그 분야도 역시 오픈 소스(Open Source) 운동의 여파가 거세게 일어나고 있다.

그동안 펌웨어는 CPU 메인 보드를 만드는 시스템 엔지니어들이 그 소스 코드를 공개하기를 꺼려왔지만(회사의 정책적인 문제겠지만) 그들도 이제 이러한 소스 코드를 공개하여, 기업체의 퀘퀘묵은 팬이 돌아가는 연구실의 동료들이 아닌, 집에서 열심히 키보드나 치는 젊은 시스템 해커(어쩌면 진정한 해커들일지도... , 해커들의 공통점은 어셈블리어를 즐긴다는 것이리라.)들의 도움을 받아 프로젝트를 성공적으로 끝맺고 있는 추세이다.

우리도 이제 이러한 펌웨어 프로젝트에 적극적으로 참여하여, 우리나라에 부족한 시스템 프로그램의 영역에서 돋보이는 활동을 했으면 한다.

참고문헌

1. 삼성전자 RTOS 팀 채종원 수석 연구원과의 개인적인 대화

2. 삼성전자 Alpha 보드팀 성종수 과장님과의 개인적인 대화

3. 삼성전자 Alpha 보드팀 서형민 연구원과의 개인적인 대화

4. Alan Cox와의 ELKS에 관한 개인적인 E-mail

5. ELKS 한글 FAQ : http://alpha.secsm.org/elks/korean-elks-faq.html

6. ssmLUG ELKS 홈페이지 : http://alpha.secsm.org/elks

7. Linux Kernel Hacker's Guide :
   http://www.linuxdoc.org/LDP/khg/HyperNews/get/khg.html

8. Linux Kernel book :

Remy Card, Eric Dumas, Franck Mevel, John Wiley & Sons Ltd, 1998

9. Linux Device Drivers :
Allessandro Rubini, O'Reilly Asso., 1998

10. UNIX Internals :
Steve D Pate, Addison-Wesley Pub., 1996

11. Operating Systems Design and Implementation :
Andrew S. Tanenbaum, Prentice-Hall International, Inc., 1987