工學碩士 學位論文

USB 프로토콜 적용을 위한 임베디드 시스템 설계 및 구현

Design and Implementation of the Embedded System for Adapting USB Protocol

指導敎授 林 宰 弘

2004年 2月

韓國海洋大學校 大學院

電子通信工學科

趙 在 晩

本 論文을 趙在晩의 工學碩士

學位論文으로 認准함.

委員長 : 梁 圭 植 (印)

委 員 : 李 尙 培 (印)

委 員 : 林 宰 弘 (印)

2004年 2月

韓國海洋大學校 大學院

電子通信工學科 趙在晩

목 차

그 림 차 례

표 차 례

Abstract

The embedded system is defined as the device which is composed for limited environment and special purpose.

These systems must be changed from existing operating system to suitable platform because they couldn't take enough dealing rate of CPU or memory compared with general desktop computer. Therefore embedded operating system is proposed for these embedded systems.

But existing embedded operating systems have many limitations, for example, size of operating system and price competition. Embedded Linux was suggested for overcome these limitations.

Usually serial or parallel protocol has been used for data communication between embedded system and PC. And IEEE 1394 protocol is used for transfer the moving picture.

But previously stated protocols are debased at data transaction rate and reliability of transaction. Also IEEE 1394 is difficult to implement because of complex constitution.

This thesis suggests USB protocol for overcome these defects. USB is able to high-speed data communication with reliability. In addition, USB is easy to implement because it is composed of serial line.

In this thesis, embedded Linux is used as the embedded operating system and USB protocol is implemented for data communication.

Therefore, it becomes possible stable usage of embedded system and reliably data communication.

제 1 장 서 론

현대 사회는 컴퓨터의 고성능화와 더불어 많은 휴대용 정보통신장치가 사용되고있다. 그 중에서도 젊은층 사이에서 급속도로 확산되고 있는 PDA와 같이 정보를 남들보다 빨리 습득하기 위한 정보화 기기들이 많은 인기를 누리고 있으나 이 시스템들은 우리가 사용하고 있는 데스크탑 컴퓨터에 비해 중앙처리장치의 처리속도, 메모리 등이 충분히 확보되어 있지 못하다. 때문에 개발자들은 이러한 시스템에 적용할 수 있는 플랫폼을 개발하여야 한다. 임베디드 운영체제는 바로 이런 시스템을 위해 개발된 운영체제이다[1].

하지만 기존의 임베디드 운영체제는 운영체제의 크기, 가격 경쟁력 등의 한계를 나타내고 있다. 이러한 한계들을 극복하고자 제안된 운영체제가 리눅스이다. 리눅스는 소스의 공개로 인해 많은 사람들이 개발을 자유롭게 할 수 있으며, 중앙처리장치에 제한되지 않는 플랫폼을 제공하여 각광 받고 있다.

임베디드 시스템과 PC의 데이터 통신을 위해서 기존에 여러 가지 방식이 사용되어 왔다. 그 예로 직렬, 병렬, IEEE 1394(Institute of Electrical and Electronics Engineers 1394) 방식들이 있다. 하지만 기존의 직렬, 병렬과 같은 데이터 통신방식은 속도와 데이터의 신뢰성에서 많은 한계점을 드러내고 있으며, 멀티미디어 통신을 위한 IEEE 1394 프로토콜은 시스템 구성이 너무 복잡하여 그 구현이 어렵다는 단점을 보여주고 있다.

이러한 점을 극복하기 위하여 본 논문에서는 USB 방식을 제안한다. USB 방식은 IEEE 1394의 특성인 고속통신이 가능하며 데이터 라인의 시리얼 구성으로 그 구성이 간단하며 구현이 쉽다. 또한 USB 방식은 핫 플러깅(Hot plugging) 기능을 제공하기 때문에 시스템이 동

작중인 상태에서도 설치와 제거가 가능한 이점을 지닌다[2].

본 논문에서는 임베디드 시스템과 PC 혹은 다른 임베디드 시스템과의 통신을 위한 USB 통신 프로토콜을 구현하였으며, 임베디드 시스템의 임베디드 운영체제는 안정성과 신뢰성이 높은 임베디드 리눅스를 이용하였다. 또한 임베디드 리눅스의 커널을 직접 분석하고 보드를 구성하여 운영체제를 임베디드 시스템에 포팅했다.

본 논문의 구성은 2장에서 USB 프로토콜의 등장배경과, 스펙을 살펴보고, 3장에서 임베디드 시스템에 적재하고자하는 커널인 임베디드 리눅스의 특징을 살펴본 후, 임베디드 리눅스의 구성 요소와 그 기능들을 소개하였다. 4장에서는 실제 시스템을 설계해 보았으며 설계한 타겟보드에 임베디드 리눅스를 포팅하며, USB 드라이버 제작을 위한 준비와 그 제작과정을 살펴본다. 후에 실제로 시스템을 구성하여 시험하였으며, 5장은 결론과 향후 연구계획에 대해 기술한다.

제 2 장 USB(Universal Serial Bus)

2.1 등장 배경

PC 환경의 멀티미디어화로 PC와 주변기기간에 송·수신 데이터 량이 증가하게 되고, 동영상이나 음성 파일을 처리하는데 있어서 기존의 직렬/병렬 포트로는 효율적이지 못하게 되었다.

또한 PC 주변장치 확장을 보다 쉽게 할 수 있고, 높은 전송률을 지원하는 저가격의 해결책을 제시할 수 있으며, 음성과 압축된 비디오에 대한 실시간 데이터에 대한 완벽한 지원, 혼합된 모드의 동기 데이터 전송과 비동기 메시지에 대한 규약상의 유연성과 유용한 장치기술에서의 통합성 그리고 다양한 PC 구성과 형태요소의 이해, 제품으로의 빠른 확산을 가능케 하는 표준환경을 제공하고 PC 성능을 증가시키는 새로운 장치의 개발이 필요하였다.

이러한 목표로 인하여 결국 USB는 마스터/슬레이브 개념과 호스트와 각 주변장치간의 직접적인 연결이나 허브를 통한 연결 하에서 작용하게 되었으며, 이는 기본적으로 직렬버스 형태로 구성된다.

첫 목표였던 PC주변장치 확장의 쉬운 사용은, USB의 중요한 특색 중의 하나로 진정한 플러그 앤 플레이 개념을 사용하는 호스트 PC에 주변장치를 쉽게 연결하거나 구성할 수 있게 된 것이다.

이로 인하여 모든 연결되는 주변장치들은 호스트상의 직접적인 USB 포트나 USB 허브장치를 통해서 외부적으로 연결될 수 있으며, 이러한 주변장치들을 사용할 수 있도록 저속 1.5Mbps, 고속 12Mbps까지 가능하게 되었다.

이러한 USB 규격과 관련하여, 1994년 7개의 USB 코어 회사들이 뭉쳐, 1995년 WinHEC(Windows Hardware Engineering Conference)

와 USB-IF(USB Implementers Forum)를 결성하였다.

그 첫 성과로 1996년 1월15일 USB 1.0 규약이 발표되었으며, USB 첫 제품들이 컴덱스에 소개되기에 이르렀다. 그러던 것이 1.0 규약의 문제점들을 해결하고 실제 사용할 수 있는 주변 장치들을 위해 1998년 9월 23일 USB 1.1 규약이 다시 발표되었다.

이러한 USB 1.1 규약의 발표로 인하여 비로소 USB 포트를 지원하는 칩셋들이 인텔이나 VIA, SiS, ALi 등과 같은 회사에서 출시되었고, 마더보드에 기본적으로 장착되기 시작했다. 하지만, 이러한 하드웨어상의 진전을 마이크로소프트사의 윈도우가 지원하지 못하여 여러 가지 문제점을 낳았고, 윈도우95의 OSR 2.1에서 비로소 제대로 지원이 되기 시작했다. 그러나 좀 더 현실적으로 보면 윈도우98 에서부터 지원되었으며, 윈도우98도 SE 버전에서 USB 주변장치들의 드라이버를 포함하는 주요한 버전 업이 이루어져, 현재에 이르고 있다[2].

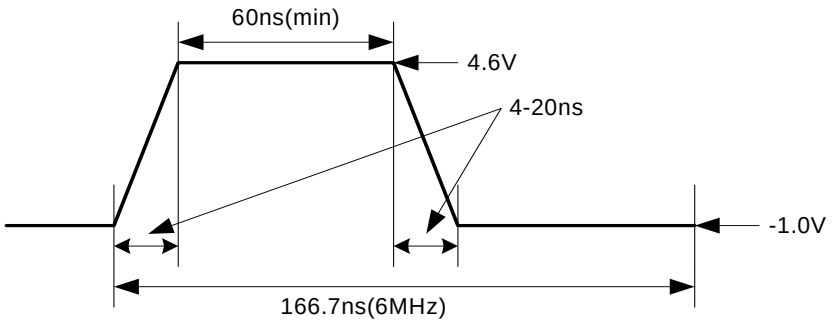
2.2 시스템 구조

USB를 구성하는 요소 중에서 빠질 수 없는 것은 USB 호스트, 허브, 디바이스와 케이블이다. 호스트를 기준으로 하여 허브를 통해 분기되어 케이블의 최대 연결 길이는 30m에 달한다. 또한 USB를 구분하기 위한 어드레스 비트 길이는 7비트이며 이로서 전체 시스템을 거대하게 구성한다고 가정할 때 전체의 접속 디바이스는 허브를 포함해서 최대 127 개가된다.

2.2.1 USB 드라이버

USB 케이블을 통해서 데이터 신호들을 전송하기 위해서는 차동 출력드라이버를 사용한다. 이러한 드라이버의 출력 스윙의 값은 1.5 ㏀의 부하를 가질 때에는 최대 V_{OL}(Low 상태에서의 출력전압)이 최대 0.3V의 값을 가지고, 15 ㏀의 부하를 가지는 경우에는 V_{OH}(High 상태에서의 출력전압)의 값이 최소 2.8V 최대 3.6V를 가지게 된다.

드라이버에서의 슬루율 제어를 잘 수행하여야 신호에서의 노이즈와 크로스 토크를 최소화 할 수 있다. 드라이버의 출력은 양방향 반이중 동작을 달성하기 위해서 3상 동작을 지원 할 수 있어야 한다. USB에서 최대 입력 신호 파형을 다음의 <그림 2-1>에 나타내었다[3], [4].



<그림 2-1> USB 최대 입력 파형

<Fig. 2-1> Maximum input waveform for USB

2.2.2 호스트

USB 시스템에서는 어떠한 경우이던지 오직 한 개의 호스트를 가진다. IEEE 1394의 피어 간의 통신과는 달리 USB는 호스트가 중심이

되는 통신 방식이다. 호스트 컴퓨터 시스템에 대한 인터페이스는 하드웨어, 소프트웨어, 펌웨어로 구성되어진 호스트 컨트롤러가 담당한다. 호스트 시스템에 내재되어 있는 루트 허브는 한 개 또는 그 이상의 접속 포인트를 제공한다.

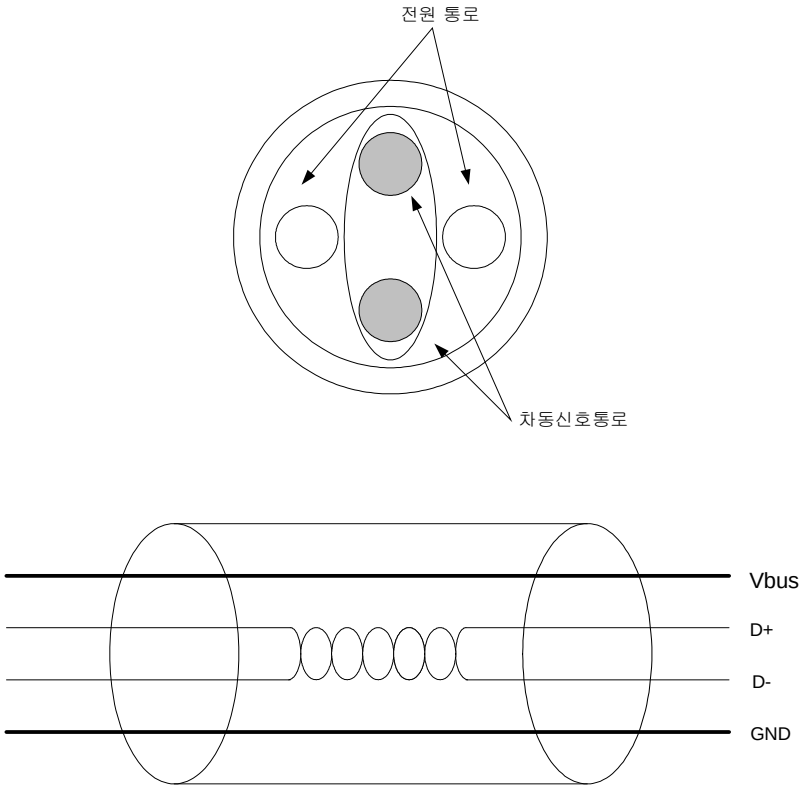
2.2.3 디바이스

디바이스란 허브와 펑션 모두를 의미하는데 허브는 USB에 결합될 수 있는 부가적인 포인트를 제공하며, 펑션은 ISDN 연결, 디지털 조이스틱 또는 스피커와 같은 시스템에 제공할 수 있는 여러 가지를 제공하는 것을 의미한다. 이와 더불어 여기서 USB 디바이스들은 표준 USB 인터페이스를 제공해야 한다.

2.2.4 케이블의 물리적 사양

4개의 신호선(전원용으로 2개, 데이터용으로 2개)으로 구성되어 있으며, 데이터는 기본적으로 차등신호로서 취급되어 1세트의 트위스트페어 선을 사용한다. 잡음을 차단하기 위해 케이블 전체는 접지로부터 쉴드 되어있으며, 최대 5m로 제한되어 있고, 최대 5단 허브까지 허용된다. 따라서 PC와 디바이스의 최대거리는 30m가 된다.

다음 <그림 2-2>에서 D+, D-의 신호 선은 트위스트페어 선으로 구현되며, 외부 노이즈에 대해서 둔감하도록 하기 위해 쉴드 되어있지만 낮은 속도로 사용하기 위한 신호선의 경우는 트위스트페어 선이나 쉴드할 필요는 없고, 대신 3m로 길이가 제한된다. <그림 2-2>에서는 케이블의 구조를 도식화하고 있다[3].



<그림 2-2> 케이블의 구조

<Fig. 2-2> Structure of USB cable

2.2.5 전기적 특성

속도는 1.1버전에서는 최대 속도를 가지는 신호의 전송률이 12Mbps를 가지며, D+에서는 풀업 되도록 구성되어 있고, 낮은 속도를 가지는 속도의 전송률은 1.5Mbps를 가지고 D-에서는 풀업 되도록 구성되어있다. 그러나 버전2.0에서는 고속(Hi-speed)을 지원하며 속도는 480Mbps까지 지원된다. 낮은 속도는 주로 휴먼 인터페이스 디바이스(HID)에 주로 사용된다.

USB 전원은 2가지 관점에서 논할 수 있는데, 첫 번째는 전원의 분

배이다. 버스에서 전원이 공급되는 디바이스 경우 호스트에 의하여 5V의 전원이 공급되며, 키보드와 마우스 등의 예를 들 수 있다. 스스로 전원을 공급할 수 있는 디바이스의 경우 자체 전원을 가지고 있으며, 프린터와 스캐너 등이 예이다. 두 번째로는 전원의 관리로써, 서스펜드, 리쥼 등의 이벤트에 따른 시스템 소프트웨어에 의해 전원이 관리된다[3].

2.3 USB의 데이터 전송 프로토콜

USB의 데이터는 프레임이라는 개념으로 시분할 되고, 프레임단위로 반복하면서 실행된다. 프레임은 1ms 시간 단위로 반복되며, SOF (Start of Frame) 패킷에 의해 개시된다. SOF 전송 후 호스트는 스케쥴링 되어 있던 데이터 전송요구 토큰을 순차 송출함으로서 복수의 펑션과 데이터 전송을 하게된다. 펑션에서 호스트로 데이터 전송시도원 데이터를 프레임 단위로 분할해서 전송해야 하므로 프레임 단위로 데이터 열을 저장해 두는 FIFO가 필요하다.

2.3.1 USB의 전송형태

(1) 등시성(Isochronous) 전송

프레임 분할에 의한 USB 전송의 특징을 가장 잘 이용한 전송방식이며, 일정주기에 일정량의 데이터를 전송하고자 할 때와 같은 실시간 어플리케이션에 주로 사용된다. 1ms 마다 프레임에서 전송 바이트를 설정할 수 있으므로 다른 전송모드와 비교하여 데이터 전송이 우선적으로 실행될 수 있다.

대표적인 특징으로는 첫 번째는 데이터 전송 폭을 보증한다. 등시성 전송을 원하는 펑션은 설정정보를 호스트에 전송할 때 원하는 전송 폭에 대한 정보를 포함한다. 호스트는 1ms의 프레임 시간 중 등시성 전송을 우선적으로 할당한다. 두 번째는 데이터 전송시간을 보증한다. 호스트는 등시성 전송에 필요한 시간을 미리 우선적으로 스케쥴링 하게된다. 세 번째로 데이터 오류 보증을 하지 않는데, CRC(Cyclic Redundancy Checking)체크시 오류가 된 경우에도 재전송 요구를 할 수 없다. 전화기에서의 음성 전송과 같이 실시간에서는 수 바이트의 오류 때문에 데이터 재전송을 요구하기보다 오류를 남긴 채 다음의 데이터 패킷을 받아들인다. 따라서 SOF의 프레임 번호, CRC를 통해서 오류를 검출한다. USB에서는 데이터 전송 에러 율을 10% 이하로 억제하도록 되어 있다.

(2) 인터럽트(Interrupt) 전송

펑션에서 호스트에 주기적으로 소량의 데이터를 입력하는 어플리케이션에 사용하며, 호스트에서 데이터 전송을 요구하는 경우에 해당하는 키보드, 마우스, 조이스틱과 같은 곳에 사용한다.

(3) 제어(Control) 전송

디바이스가 설정 정보 등을 호스트에 전송할 때 사용한다. 호스트와 디바이스 제어 엔드포인트를 사용해서 전송되며, 전송 데이터 패킷 포맷이 USB 규격으로 정해져 있다.

(4) 벌크(Bulk) 전송

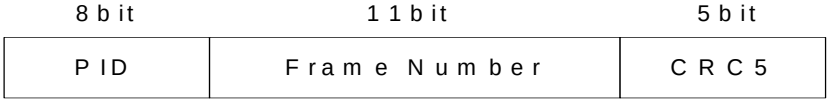
대량의 데이터를 신뢰성을 가지고 고속으로 전송할 경우에 사용하며, 실시간 어플리케이션의 경우보다는 전송 데이터의 질을 보증해야

하는 어플리케이션에 사용하며 등시성, 인터럽트 전송 후 사용하지 않는 시간에 데이터를 전송하는 방식에 사용한다. 데이터 패킷 오류발생 시에는 재전송을 요구하며, 전송속도, 전송 지연시간은 보증되지 않는다.

2.3.2 USB 패킷의 종류

USB에서의 데이터의 전송은 3단계를 통해서 이루어지며, 프레임, 트랜잭션, 패킷이 그에 해당된다. 패킷은 전송의 최소단계이고 이러한 패킷이 모여서 트랜잭션이 되어 트랜잭션은 호스트와 타겟 디바이스 간의 전송대상이 된다. 패킷과 트랜잭션이 모여서 한 프레임이 되는데 이러한 프레임은 1ms마다 발생하고 SOF 패킷, 전송될 트랜잭션을 포함하게 된다. 패킷은 0x80의 값을 가지는 동기 패턴이 제일 먼저 시작되게 되고, 지금 현재의 패킷이 어떤 것인지를 나타내는 PID(Packet ID)가 그 뒤를 따르게 된다.

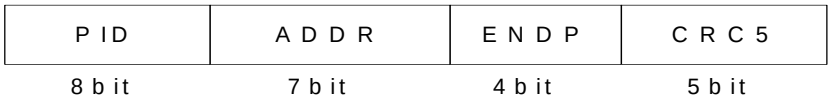
SOF 패킷은 프레임의 시작을 나타내며, 1ms마다 호스트에 의하여 전송된다. PID는 패킷 타입을 지정하는 ID이다. 프레임 번호는 그 명칭이 나타내는 그대로 11비트의 크기를 가지는 프레임의 번호를 나타내기 위한 필드이다. CRC5에서는 X^5+X^2+1을 이용하여 에러의 발생 유무를 판단하게 된다. 다음 <그림 2-3>에서 SOF 패킷의 형태를 보여준다.



<그림 2-3> SOF 패킷의 형태

<Fig. 2-3> SOF packet format

토큰 패킷은 호스트에 의해서만 전송되는 것이며 내부요소는 다음 <그림 2-4>에서 보이는 것과 같이 구성되어 있다. ADDR은 펑션의 주소를 나타내며 최대 개수는 256개이다. ENDP는 엔드포인트의 ID를 나타내며, 펑션당 엔드포인트는 16 개가된다.



<그림 2-4> 토큰 패킷의 형태

<Fig. 2-4> Token packet format

데이터 패킷은 데이터와 더불어 다른 패킷에서는 CRC5를 사용했지만 여기서 16 bit의 CRC를 사용한다. 데이터는 반드시 바이트 단위여야 하며, 전송하려는 데이터는 DATA0, 1의 두 종류가 존재하고 데이터가 여러 개의 패킷으로 분할되어 전송되는 경우에 이 두 가지가 교대로 전송되게 된다. 다음 <그림 2-5>에서 데이터 패킷의 형태를 보여주고 있다.

<그림 2-5> 데이터 패킷의 형태

<Fig. 2-5> Data packet format

핸드쉐이크 패킷은 PID로만 구성되며, 데이터 트랜잭션 결과를 보고하기 위하여 이용된다. PID로서는 ACK, NAK, STALL의 3가지의 종류가 존재한다. ACK의 경우는 데이터의 수신이 정상 종료된 경우에 발생되며, IN 트랜잭션인 경우에는 호스트가 발생하며, OUT 트랜

잭션인 경우는 타겟 디바이스가 발생한다. NAK는 데이터의 전송준비가 아직 되지 않았을 경우 많이 발생하게 된다. 호스트가 NAK를 수신하게 되면, 같은 트랜잭션을 다시 전송하게 된다. STALL의 경우에는 타겟 디바이스만 발생하게 되며, 트랜잭션이 비정상적으로 종료되거나 엔드포인트가 사용불가능이라는 것을 나타내게 된다.

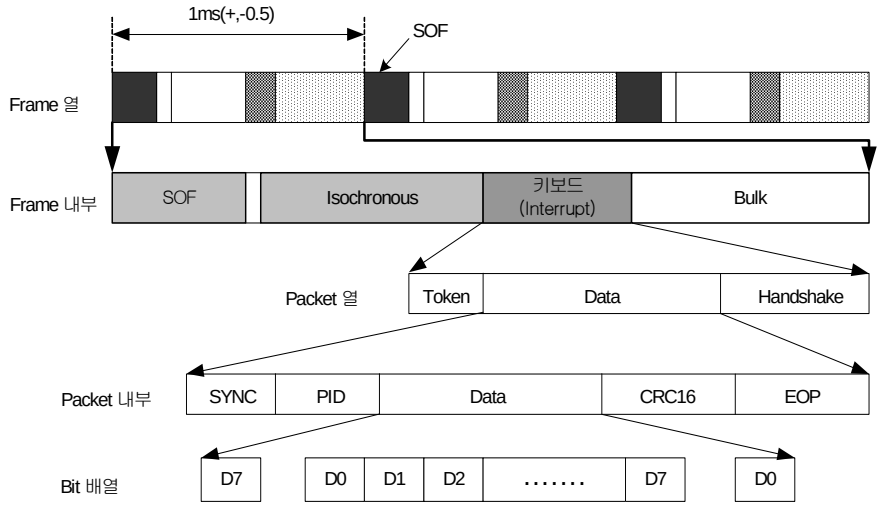
USB에서 전송하고자 하는 최소단위의 여러 가지 패킷에서는 어떠한 패킷이 전송되는지를 알리기 위한 PID를 반드시 포함하게 되며, 이는 ACK, NAK, STALL 3종류가 존재한다.

PID는 8비트로 구성되며 하위 4비트는 정상적인 PID를 나타내기 위한 것이며, 상위 4비트는 하위 4비트를 반전시킨 것이다. 다음 <표 2-1>에서는 PID의 종류에 따른 구성을 보여준다.

<표 2-1> PID의 종류

<Table. 2-1> Classification of PID

그룹명	PID 명	값	정 의
Token	OUT	0001	엔드포인트로의 전송
	IN	1001	호스트로의 전송
	SOF	0101	프레임의 시작
	SETUP	1101	셋업
Data	DATA0	0010	짝수 PID
	DATA1	1011	홀수 PID
Handshake	ACK	0010	데이터 수신성공
	NAK	1010	데이터 전송불가
	STALL	1110	엔드포인트의 stall
특별한 경우	PRE	1100	Low Speed 전송의 기능



<그림 2-6> 프레임 구조

<Fig. 2-6> Frame structure

데이터를 전송하는 경우에 있어서 프레임의 구조를 <그림 2-6>에서 자세히 나타내었다. 열의 시작은 SOF로 이루어지는 것을 볼 수 있으며, 1ms마다 한 개의 프레임이 전송될 수 있음을 나타낸다. 각 프레임의 내부를 볼 때 등시성, 인터럽트, 벌크 모드의 전송이 이루어지는 것을 나타내며, 인터럽트 전송일 경우에 패킷 그 내부가 어떻게 구성되는 지에 대해서 나타내었다[4].

제 3 장 임베디드 리눅스

3.1 리눅스와 임베디드 시스템

리눅스는 초기에 PC에서 사용되는 프로세서인 인텔 i386 프로세서 계열만 지원했다. 그 후에 다른 중앙처리장치에 포팅을 하면서 아키텍처 의존적인 코드들이 정리되기 시작했다. 이런 정리가 이루어진 후 리눅스 해커들은 자신이 사용하는 시스템의 프로세서에 하나 둘씩 이식해 가는 숫자를 늘려 갔고 덕분에 리눅스 커널은 다른 아키텍처에 이식하기 쉬운 구조가 되었다.

그러나 초기에는 고기능의 프로세서에 커널 포팅을 수행했기 때문에 주로 알파 프로세서 같은 고성능 프로세서에만 포팅이 가능했다. 게다가 아무리 이식성이 우수할지라도 8비트 원칩 프로세서와 같은 계열에 포팅 한다는 것은 불가능하다. 16비트 프로세서인 인텔 286에 포팅 하는 프로젝트도 존재하기는 하지만 지금은 진행이 미비한 상태이다. 리눅스 커널이 필요로 하는 최소 사양은 32비트 처리 능력과 최소한 메모리 4MB 이상은 되어야 한다. 하지만 32비트 원칩 프로세서가 등장하면서 리눅스 커널이 동작할 수 있는 성능이 보장되기 시작했다[5].

다음에서 임베디드 시스템 개발자들이 리눅스 커널의 사용을 선호하는 이유에 대해서 간략히 기술한다.

첫째, 리눅스는 소스가 공개되어 있다. 소스가 공개되어 있다는 것은 신뢰성과 연관될 수 있다. 또한 문제가 생기면 그 문제에 대한 제어권이 개발자 자신에게 있다는 것은 개발자가 문제를 해결할 가능성을 내포하고 있음을 의미한다.

둘째, 많은 디바이스 드라이버 소스가 포함되어 있다. 임베디드 시

스템 개발하고자하는 프로그래머는 디바이스 제어 프로그램을 제작해야한다. 이런 디바이스 제어에 필요한 소스가 커널에 존재하고 있기 때문에 리눅스 프로그래머에게 개발의 용이성을 가져준다.

셋째, 응용프로그램이 하드웨어 구조에 영향을 적게 받는다. 운영체제 없이 개발하는 펌웨어 경우에는 새로운 하드웨어를 도입하게되면 다시 작성해야 한다. 하지만 리눅스 커널을 도입하게되면 응용프로그램 소스는 거의 손대지 않아도 된다. 즉, 하드웨어 관련분야는 커널수준에서 수정이 가능하다는 얘기이다. 이는 개발 생산성이라는 측면에서 상당한 이득을 가져다 준다.

넷째, 실제 하드웨어가 없어도 응용프로그램을 구현할 수 있다. 임베디드 시스템 개발자 대부분이 하드웨어가 준비된 후 개발을 진행하는 방식을 택한다. 하드웨어 의존적인 코드가 많기 때문에 미리 하기가 힘들고 시험자체도 검증이 되지 않기 때문이다. 하지만 리눅스 커널이 동작하는 환경이면 데스크탑 컴퓨터와 임베디드 시스템의 응용프로그램 구현 방식이 같다. 필요한 하드웨어 제어는 디바이스 드라이버 접근에 대한 처리를 에뮬레이션 형태로 해주면 대부분의 로직이 검증되기 때문이다.

지금까지 임베디드 시스템에 리눅스를 적용하게 된 이유를 정리해 보았다. 다음 장에서는 임베디드 시스템을 위한 리눅스 커널을 수정한 임베디드 리눅스에 대해서 서술한다.

3.2 임베디드 리눅스의 개요

임베디드 리눅스는 '낮은 성능의 프로세서와 작은 크기의 메모리를 가진 임베디드 시스템용으로 개발된 리눅스' 이다. 따라서 임베디드 리눅스는 다음의 두 조건을 만족해야 한다.

첫째, 임베디드 시스템은 작은 크기의 메모리 밖에 장착할 수 없다는 제약으로 인하여 리눅스 자체의 크기와 기능이 최소화, 경량화, 그리고 맞춤화 되어야한다. 이 조건은 임베디드 리눅스가 가져야 하는 필수조건이다. 둘째, 낮은 성능의 프로세서를 사용하는 제약을 극복하기 위하여 성능이 최적화되어야 한다.

임베디드 리눅스가 부각되고 있는 이유는 최근에 정보기기라고 통칭되는 새로운 종류의 임베디드 시스템들이 붐을 일으키고 있기 때문이다. 인터넷으로 통칭되는 통신 혁명으로 인해서 이러한 결과물을 낳은 것이다. IDC(Internet Data Center)에 따르면, 정보 가전은 2005년에 가면 PC 수요를 앞지를 것이라고 한다.

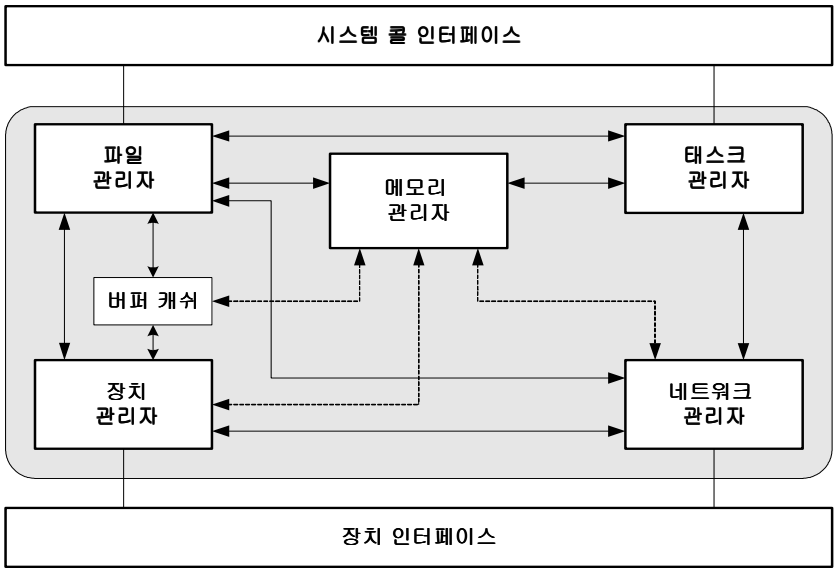
이러한 정보가전 장치들은 제한된 메모리를 부착하고 있으며, 겨우 수백 달러에 팔리게 될 것이므로 개발자들은 윈도우즈와 같은 크고 값비싼 운영체제를 대체할 수 있는 대안을 찾게 되었다. 그 최적의 대안으로 임베디드 리눅스가 떠오르고 있다[1].

3.2.1 리눅스 커널의 구조

리눅스 커널은 크게 태스크 관리자, 메모리 관리자, 파일시스템, 네트워크 관리자, 그리고 디바이스 드라이버의 장치관리자 등 5가지 부분으로 구성된다. 커널은 자원 관리자이며, 커널이 관리하는 자원에는 물리적인 자원과 추상적인 자원으로 나뉘어진다.

태스크 관리자는 태스크의 생성, 실행, 상태 전이, 스케줄링, 시그널 처리, 프로세스간 통신 등의 서비스를 제공한다. 메모리 관리자는 가상 메모리, 주소 변환, 페이지 부재 결합 처리 등의 서비스를 제공한다. 파일시스템은 파일의 생성, 접근 제어, inode 관리, 디렉토리 관리, 슈퍼 블록 관리, 버퍼 캐쉬 관리 등의 서비스를 제공한다. 네트워크

관리자는 소켓 인터페이스, TCP/IP 같은 통신 프로토콜 등의 서비스를 제공한다. 장치 관리자는 디스크 드라이버, 터미널, CDROM, 네트워크 카드 등과 같은 주변 장치를 구동하는 드라이버들로 구성된다. 다음 <그림 3-1>은 커널의 내부구조를 나타내고 있다.



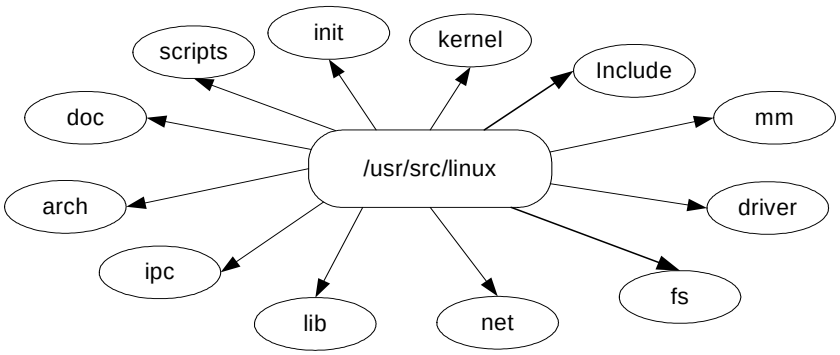
<그림 3-1> 리눅스 커널의 내부구조

<Fig. 3-1> Internal structure of Linux kernel

커널은 사용자 수준 응용과 하드웨어 사이에 존재하며, 하드웨어를 관리하고 이에 대한 서비스를 사용자 수준 응용에게 제공한다. 커널은 디바이스 인터페이스와 인터럽트 처리 매커니즘 등을 이용하여 하드웨어와 통신한다. 또한 커널은 시스템 호출 인터페이스를 이용하여 사용자 수준 응용레벨과 통신한다[6], [7].

3.2.2 커널의 소스 트리 구조

지금까지 커널의 구조와 상호 연동작용에 대해 간단히 살펴보았다. 지금부터는 커널의 실제 구성을 살펴보고자 한다. <그림 3-2>에서 커널의 소스 트리 구조를 도시 하고있다.



<그림 3-2> Linux 커널의 소스 트리 구조

<Fig. 3-2> Source tree structure of Linux kernel

/kernel 디렉토리는 태스크 관리자가 구현된 디렉토리이다. 태스크의 생성과 소멸, 프로그램의 실행, 스케줄링, 시그널 처리 등의 기능이 이 디렉토리에 구현되어 있다.

/arch 디렉토리는 리눅스 커널 기능 중 하드웨어 종속적인 부분들이 구현된 디렉토리이다. 중앙처리장치의 형태 즉, 인텔 처리기, 64 bits 알파 AXP처리기, 32bits ARM(Advanced RISC Machine) 처리기, 모토롤라 68xxx처리기, Sun Sparc 처리기, Power PC 처리기 등에 따라 하위 디렉토리가 다시 구분된다.

/fs 디렉토리는 리눅스에서 지원하는 다양한 파일 시스템들과 파일 조작에 관련된 시스템 호출이 구현된 디렉토리이다. 각 파일 시스템은

하위 디렉토리에 구현되어 있는데, 대표적인 파일 시스템으로는 ext2, nfs, msdos, ntfs, proc, coda등이 있다. 그리고, 다양한 파일시스템을 사용자가 일관된 인터페이스로 접근할 수 있도록 하기 위하여 리눅스는 시스템 호출과 각 파일시스템 사이에 추상화된 개념, 즉 가상파일 시스템(VFS : Virtual File System)이 구현되어 있다.

/mm 디렉토리는 메모리 관리자가 구현된 디렉토리이다. 가상 메모리, 태스크마다 할당되는 메모리 객체 관리, 커널 메모리 할당자 등의 기능이 구현되어 있다.

/driver 디렉토리에는 디스크, 터미널, 네트워크 카드 등 주변 장치를 추상화시키고 관리하는 커널 구성요소인 디바이스 드라이버가 구현된 디렉토리이다. 리눅스에서 디바이스 드라이버는 크게 블록 디바이스 드라이버, 문자 디바이스 드라이버, 네트워크 디바이스 드라이버로 구분된다.

/net 디렉토리에서는 리눅스에서 지원하는 통신 프로토콜이 구현되어 있다. 현재 리눅스에는 대표적인 통신 프로토콜인 TCP/IP 뿐만 아니라 유닉스 도메인 통신 프로토콜, X.25, IEEE 802 통신 프로토콜, IPX (Internetwork Packet Exchange), SUN RPC(Remote Procedure Call), Apple Talk 등이 하위 디렉토리에 구현되어 있다. 한편 사용자 인터페이스를 제공하는 소켓은 net/ 디렉토리에 구현되어 있다.

/ipc 디렉토리는 리눅스 커널이 지원하는 IPC 기능이 구현된 디렉토리이다. 이 디렉토리에는 메시지 패싱, 공유 메모리, 세마포어가 구현되어 있다.

/init 디렉토리는 커널의 초기화 부분, 즉, 커널의 메인 시작 함수가 구현된 디렉토리이다.

/include 디렉토리는 리눅스 커널이 사용하는 헤더 파일이 구현된 디렉토리이다. 헤더 파일 중에서 하드웨어 독립적인 부분은 하위에 /linux 디렉토리에 구현되어 있으며, 하드웨어 종속적인 부분은 처리

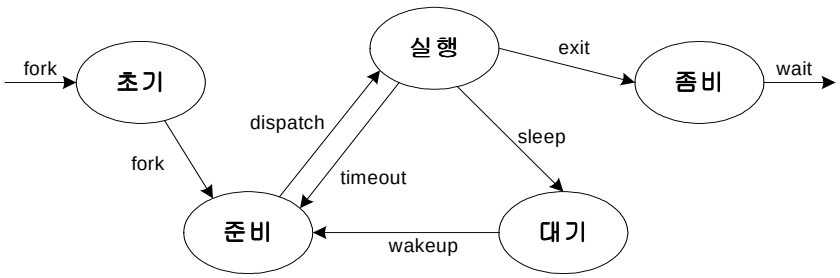
기 이름으로 구성된 하드웨어 디렉토리에 구현되어 있다.

그 외에 리눅스 커널 및 명령어들에 대한 자세한 문서 파일들이 존재하는 /doc 디렉토리, 커널 라이브러리 함수들이 구현된 /lib 디렉토리, 컴파일된 모듈 함수들이 존재하는 /module 디렉토리 그리고 커널 구성 및 컴파일 할 때 이용되는 스크립터들이 존재하는 /scripts 디렉토리 등이 존재한다[6], [7].

3.2.3 태스크 관리자

(1)태스크 상태전이

태스크는 생성되어 소멸되기까지 다양한 상태 전이 과정을 겪는다. <그림 3-3>은 태스크 상태 전이과정을 도시화한 것이다.



<그림 3-3> 태스크의 상태 전이

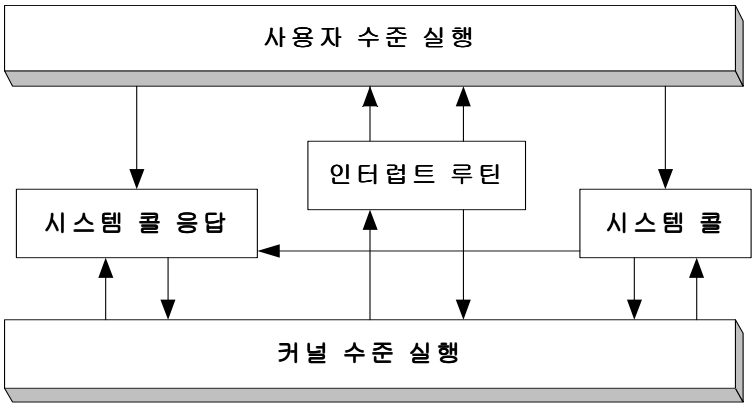
<Fig. 3-3> State transition of task

사용자 프로세스는 프로그램 형태로 보조 기억장치에 저장된 후 시스템 호출을 통해서 운영체제의 프로세스 형태로 생성된다. 생성된 프로세스는 TASK_RUNNING 상태를 가지며 스케줄러에 의해서 스케줄되어 동작상태가 되고 다시 타이머 인터럽트에 의해 준비상태가 되며, 이 과정을 프로세스가 종료될 때까지 반복한다. 만약 동작 상태에

서 I/O 관련 작업을 기다리다가 이벤트 상태가 되면 이때는 TASK_ZOMBIE, TASK_STOPPED 상태를 갖는다.

사용자 수준 실행 상태는 프로세스가 프로그래머가 작성한 프로그램이나 라이브러리 함수를 수행하는 상태로, 사용자 수준 권한으로 동작한다. 하지만, 커널 수준 실행 상태는 프로세스가 커널 프로그램의 일부분을 실행하는 상태로 사용자 수준 권한보다 더욱 강력한 커널 수준 권한으로 동작한다.

프로세스의 실행 상태는 사용자 수준 실행 상태와 커널 수준 상태로 구분 할 수 있다. <그림 3-4>는 두 수준 상태를 구분하여 보여준다.



<그림 3-4> 실행 상태 구분

<Fig. 3-4> Division of running state

사용자 수준 실행 상태에서 커널 수준 상태로 전이할 수 있는 방법은 시스템 호출과 인터럽트의 발생이 있다. 먼저 프로세스가 시스템 호출을 요청하면 리눅스 커널에 트랩이 걸리게 되고 그 결과 태스크의 상태가 커널 수준 실행 상태로 전이되며 커널이 시스템 호출 루틴으로 상태가 넘어간다.

인터럽트에 의한 방법은 리눅스 커널에 인터럽트가 걸리게 되면, 이때 실행 중이던 프로세스가 사용자 수준에서 동작하게 된다. 또한 프로세스는 커널 수준 실행 상태로 전이되고, 커널의 인터럽트 처리 루틴으로 넘어가게 된다.

그리고 커널이 시스템 호출의 서비스를 완료하거나 인터럽트 처리를 완료하면 커널 수준 실행 상태에서 사용자 수준 실행 상태로 전이한다[7].

(2) 스케줄링

프로세스는 항상 시스템 호출을 하므로 종종 기다리게 된다. 그럼에도 불구하고 어떤 프로세스는 기다리게 될 때까지 너무 많은 중앙처리장치의 시간을 사용하게 되므로, 이러한 경우 리눅스는 선점형 스케줄링 기법을 사용한다.

리눅스에서는 한 프로세스가 정해진 타임 슬라이스를 초과해서 사용하면, 그 프로세스를 중단시켜 다른 프로세스를 실행하는 선점형 스케줄링을 한다. 하지만, 리눅스의 커널 모드에서는 비 선점형으로 동작한다. 이는 커널 코드가 재진입이 가능하지 않게 만들어졌기 때문이다. 일단 시스템 호출이 되면 자발적으로 중앙처리장치의 할당을 내놓지 않는 이상 시스템 호출은 다른 프로세스에 의해 멈추지 않는다.

(3) 시그널 처리

시그널은 프로세스에게 비동기적인 사건의 발생을 알리는 방법이다. 프로세스가 시그널을 처리하기 위해서 다른 프로세스에게 시그널을 보낼 수 있는 기능, 자신에게 오는 시그널을 수신할 수 있는 기능, 그리고 자신에게 시그널이 오면 그 시그널을 처리할 수 있는 함수를 호출할 수 있는 기능을 가져야 된다.

(4) 프로세스간 통신

프로세스들은 상호간의 활동을 조정하기 위하여 프로세스간, 그리고 커널과 통신을 한다. 리눅스는 시그널, 파이프, 세마포어 같은 프로세스간 통신 기능을 제공한다.

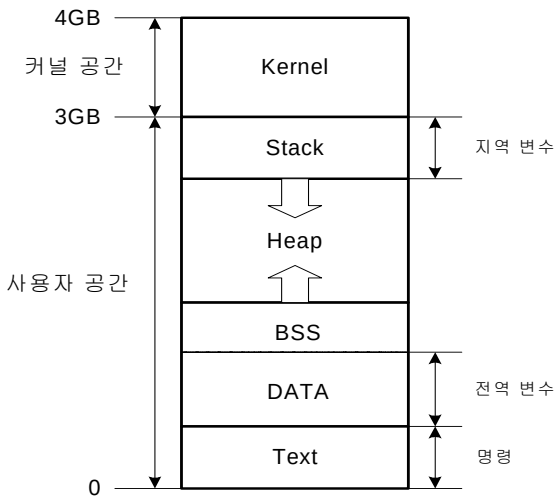
3.2.4 메모리 관리

(1) 가상 메모리

컴퓨터가 개발될 당시부터 사용자들은 시스템에 물리적으로 존재하는 것보다 많은 메모리를 필요로 하였다. 물리 메모리의 한계를 극복하기 위한 다양한 기법들이 개발되었지만, 그 중에서 가장 성공적이며 현재까지 시스템에 적용되고 있는 방식이 가상 메모리이다. 가상 메모리는 실제 시스템에 존재하는 물리 메모리의 크기와 상관없이, 32 bit 처리기의 경우 2^{32} 크기(4GB)의 가상 주소 공간을, 64 bit 경우 2^{64}크기의 주소공간을 사용자에게 제공한다.

다음 <그림 3-5>는 인텔 처리기에서의 가상 메모리의 개념적 구조를 나타내고 있다. 텍스트 세그먼트는 프로그램의 명령어 부분으로 구성되고, 데이터 세그먼트 경우 초기화된 데이터 부분과 초기화되지 않은 데이터 부분으로 구분된다.

이 경우 앞부분은 데이터 영역이라 하며, 뒷부분은 BBS(Block Started by Symbol)이라 한다. 스택 세그먼트는 함수 호출 시 전달되는 인자들과 리턴 주소 그리고 함수의 지역변수들로 구성된다. 텍스트 세그먼트는 가장 하위 공간을 차지한다. 즉 가상주소 0x00번지부터 텍스트 세그먼트가 사용한다. 데이터 세그먼트는 텍스트 세그먼트 끝 이후에 위치하게 된다. 스택 세그먼트는 사용자 공간과 커널 공간의 경계에 위치(3GB, 가상주소 0xC0000000)부터 아래 방향으로 차지한



<그림 3-5> 가상메모리 구조

<Fig. 3-5> Structure of virtual memory

다. 스택은 프로그램이 수행됨에 따라 동적으로 변한다. 즉 스택은 호출함수 인자와 지역변수를 저장하기 위하여 아래 방향으로 크기가 커진다.

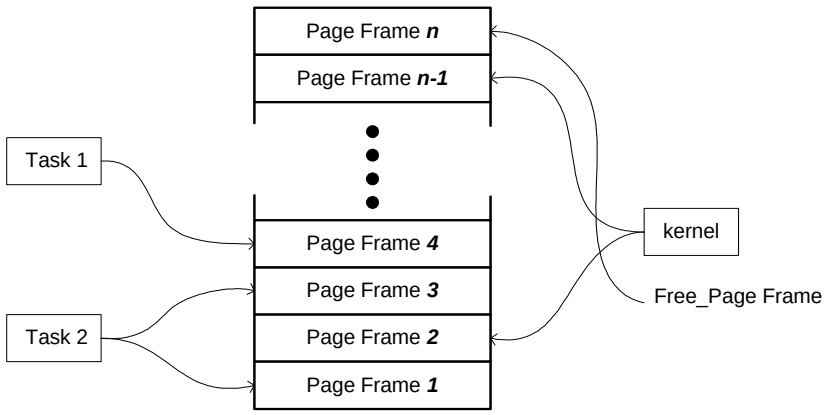
프로그램 수행 중에 동적으로 메모리 공간을 할당받을 수 있다. 이때 메모리가 할당되는 공간을 힙 영역이라 부르며, 데이터 세그먼트 이후부터 시작된다.

(2) 물리 메모리

가상 메모리와는 달리 물리 메모리는 시스템에 존재하는 메인 메모리의 크기에 의해 그 크기가 결정되고 페이지 프레임이라는 고정된 크기의 기본 단위로 분할된다. 여기서 페이지 프레임과 가상메모리에서 사용하는 페이지의 크기는 일반적으로 그 크기가 동일하다.

페이지 프레임의 크기는 처리기에 따라 다르다. 인텔 처리기의 경우

크기는 4KB이며, ARM 처리기 경우 16KB이다. 아래 <그림 3-6>은 물리 메모리의 구조를 보여준다.



<그림 3-6> 물리 메모리의 구조

<Fig. 3-6> Structure of physical memory

3.2.5 리눅스 파일 시스템

리눅스의 가장 중요한 특징 중 하나는 다양한 파일 시스템을 지원한다는 것이다. 이는 리눅스가 타 운영체제들과 공조할 수 있는 유연성을 가지고 있다는 것을 의미한다.

리눅스는 시스템이 사용할 수 있는 각각의 파일 시스템이 장치식별자로 접근되는 것이 아니라 하나의 계층적인 트리 구조로 통합해 들어가 파일 시스템이 마치 하나인 것처럼 보이게 한다.

리눅스가 처음 사용했던 미닉스 파일 시스템은 제한적이고 성능이 좋지 못했다. 파일 이름이 14자를 넘지 못했고, 파일 크기가 64M로 제한되었다. 리눅스 전용으로 설계되었던 첫 번째 파일 시스템은 확장 파일 시스템으로서 1992년에 소개되었고 많은 문제점들을 해결했지만

아직도 성능개선이 크게 이루어지지 못했다.

1993년에 2차 확장 파일 시스템(EXT2 : Second extended file system)이 추가되었다. 2차 확장 파일 시스템은 확장 파일 시스템과는 달리 255문자의 파일명, 최대 2GB의 파일, 4TB의 디스크 용량을 지원한다. 또한, 파일시스템이 깨지거나 지워지는 돌발상황에 대비하여 몇 개의 블록 그룹을 가지고 있는데 이러한 블록의 정보를 중복해서 저장하고 있다[7].

3.2.6 네트워크 관리자

통신 프로토콜은 계층 구조를 갖는다. 가장 상위계층은 소켓 인터페이스를 제공하는 BSD(Berkeley Software Distribution) 소켓층이다. 소켓은 통신 연결의 한쪽 끝으로 생각할 수 있는데, 통신하고 있는 두 태스크는 통신 연결에서 자신쪽 끝에 해당하는 소켓을 가지게 된다.

BSD 소켓층에서 사용자는 프로토콜 패밀리를 선택할 수 있다. 리눅스에서 지원되는 대표적인 프로토콜 패밀리는 INET(TCP/IP), 유닉스, IPX, APPLE TALK 등이 있다. INET 층에서 일반적으로 사용되는 유형에는 스트림과 데이터그램이 있다.

스트림은 데이터가 전송 중 분실, 오염 또는 중복되지 않는다는 것을 보장하는 신뢰할 수 있는 양방향 순차 데이터 전송을 제공하며, 데이터 그램은 양방향 데이터 전송을 제공하지만 스트림 소켓과는 달리 그 메시지가 제대로 도착한다는 것을 보장하지는 않는다.

사용자가 스트림 유형의 소켓을 선택하였다면 TCP 층으로 내려가게 된다. 반면에 데이터그램 유형의 소켓을 선택하였다면 UDP 층으로 내려가게 된다.

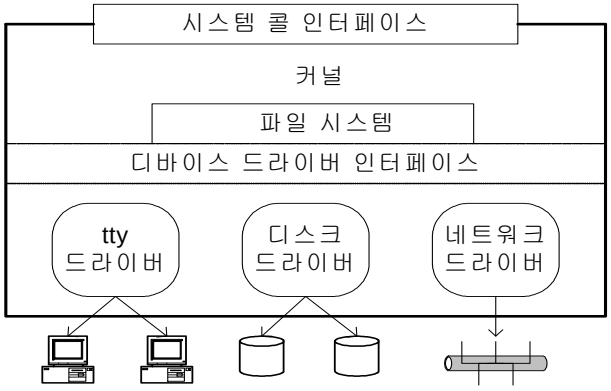
그리고 에러 처리를 위한 검사합, 데이터가 너무 클 경우 단편화, 다

른 호스트로 재 전송 등을 수행한다. 또한, 라우팅 정보와 목적지 IP를 보고 해당 네트워크 드라이버에게 패킷을 전송하며, 전송할 때 패킷 흐름 제어 등의 부가적인 기능을 한다.

IP 계층 아래에는 PPP(Point-to-Point Protocol), SLIP(Serial Line Internet Protocol.) 또는 이더넷과 같은 네트워크 계층이 존재하는데, 이 층은 네트워크 디바이스가 존재하고, 각 디바이스는 드라이버라는 자료구조에 자신의 정보를 저장하여 IP 층에 제공하게 된다.

3.2.7 장치 관리자

리눅스에는 디바이스 드라이버가 매우 간결하면서 일관된 구조로 구현되어 있다. 디바이스 드라이버는 디바이스와 시스템 메모리간에 데이터의 전달을 담당하는 커널 내부 기능이다. 일반적으로 상위레벨의 파일시스템과 인터페이스를 가지며 하위레벨의 실제 디바이스 하드웨어와 인터페이스를 갖는다.



<그림 3-7> 디바이스 드라이버

<Fig. 3-7> Device driver

<그림 3-7>에서 커널내부의 디바이스 드라이버의 역할을 보여주고

있다. 디바이스 드라이버는 크게 문자 디바이스 드라이버, 블록 디바이스 드라이버, 그리고 네트워크 디바이스 드라이버 등으로 구분되어진다[8].

문자 디바이스 드라이버는 순차 접근이 가능하고 임의의 크기로 데이터 전송이 가능한 드라이버이다. 블록 디바이스 드라이버는 임의 접근이 가능하고 고정된 크기의 블록 단위로 데이터를 전송하는 드라이버로서, 이 두 가지 구분은 커널의 버퍼 캐쉬를 이용하는가 여부에 따라 문자 디바이스 드라이버와 블록 디바이스 드라이버를 구분한다.

또한, 네트워크 디바이스 드라이버는 네트워크에 프레임을 전송하거나 받는 드라이버이다.

새로운 디바이스 드라이버를 리눅스에 추가할 때 필요한 과정은 다음과 같은 세 단계로 이루어진다[9].

- 디바이스 드라이버 함수를 구현한다. 디바이스 드라이버는 파일시스템과 인터페이스, 디바이스와 인터페이스, 드라이버 초기화 인터페이스 등 잘 정의 된 인터페이스를 가지고 있으며 따라서 사용자는 각 인터페이스를 위한 함수를 구현해 주어야 한다.
- 디바이스 드라이버를 커널에 등록한다.
- 디바이스 드라이버를 위한 파일을 생성한다.

디바이스 드라이버는 문자인지 블록인지 또는 네트워크인지에 따라 구조가 조금씩 다르다. 아래는 각 디바이스 드라이버의 차이점에 대해 보여준다.

(1) 문자 디바이스 드라이버의 구조

대표적인 문자 디바이스 드라이버에는 터미널 디바이스 드라이버가 있다. 터미널 디바이스 드라이버는 크게 파일 시스템과 인터페이스를

갖는 함수와 하드웨어와 인터페이스를 갖는 함수, 그리고 디바이스를 초기화하는 함수로 구분된다.

(2) 블록 디바이스 드라이버

대표적인 블록 디바이스 드라이버는 IDE 하드디스크 디바이스 드라이버가 있다. 블록 디바이스 드라이버는 버퍼 캐쉬라는 공간을 사용하는데, 이 버퍼 캐쉬는 사용자와 커널, 커널과 블록 디바이스간의 데이터 크기 차이를 해결한다. 뿐만 아니라 버퍼 캐쉬는 다시 참조되는 데이터를 디스크에 접근하지 않고 메모리 자체에서 서비스 해 주는 캐슁 기능, 지연 쓰기, 미리 가져오기 등의 부가적인 기능도 제공한다.

(3) 네트워크 디바이스 드라이버

네트워크 드라이버도 다른 유형의 드라이버처럼 커널 내의 상위층과 인터페이스를 갖는 부분, 네트워크 디바이스에 명령을 내리거나 상태를 읽는 부분, 초기화하는 부분 등 세 부분으로 구성된다.

문자 디바이스 드라이버나 블록 디바이스 드라이버가 파일 시스템과 인터페이스를 갖는데 비해 네트워크 디바이스 드라이버는 통신 프로토콜 스택과 인터페이스를 갖는다. 따라서, read(), write()등에 대한 인터페이스는 없으며, 그 대신 패킷을 네트워크로 전송하는 인터페이스와 네트워크에서 받은 패킷을 프로토콜 스택 층에 전달하는 인터페이스 등이 존재한다. 네트워크 드라이버에는 파일 연산 자료구조가 없는 대신 커널 자료구조가 역할을 대신한다.

제 4 장 평가 보드 설계 및 구현

본 논문에서는 임베디드 시스템과 PC 사이의 USB 통신을 구현해 보고자 한다. 개발하고자 하는 시스템은 ARM 코어의 ARM920T를 ASIC으로 생산된 삼성 반도체의 S3C2410X를 MCU로 사용한 보드이며 USB 호스트 컨트롤러와 디바이스 컨트롤러가 칩 내부에 구현되어 있다. 여기에 임베디드 리눅스를 운영체제로써 포팅하여 평가보드를 구성하였다.

4.1 평가 보드 구성

4.1.1 S3C4210의 구성과 특성

(1) ARM920T의 특성

ARM920T는 ARM에서 개발한 16/32-bit RISC 프로세서로서, ARM 프로세서의 가장 큰 특징은 저 전력 소모라는 점과 비교적 저렴한 가격이다. 또한 각 프로세서간 명령어에 별다른 차이점도 없고 대부분 16 비트/32 비트 명령어를 선택하여 사용할 수 있다.

ARM920T는 가장 최근에 나온 코어로서 16K의 I-캐쉬와 D-캐쉬를 갖고 있으며 최대 220MHz에서 동작 가능하다. 또한 ARM920T는 운영체제를 위한 가상 어드레싱을 지원하는 MMU를 내장하고 있기 때문에 포팅에 있어서 상당한 장점을 가진다[10].

ARM920T 은 EPOC, 리눅스, 윈도우CE 와 같은 실시간 또는 임베디드 시스템 운영체제를 지원하기 위해서 만들어 졌으며 PDA나 스마트 카드 또는 인터넷 응용기기 등의 개발에 적합하다.

또한 고성능 매크로셀과 캐쉬가 있기 때문에 소프트모뎀, 음성인식과 같은 실시간 적인 요소가 운영체제와 더불어 한 중앙처리장치에서

수행될 수 있다.

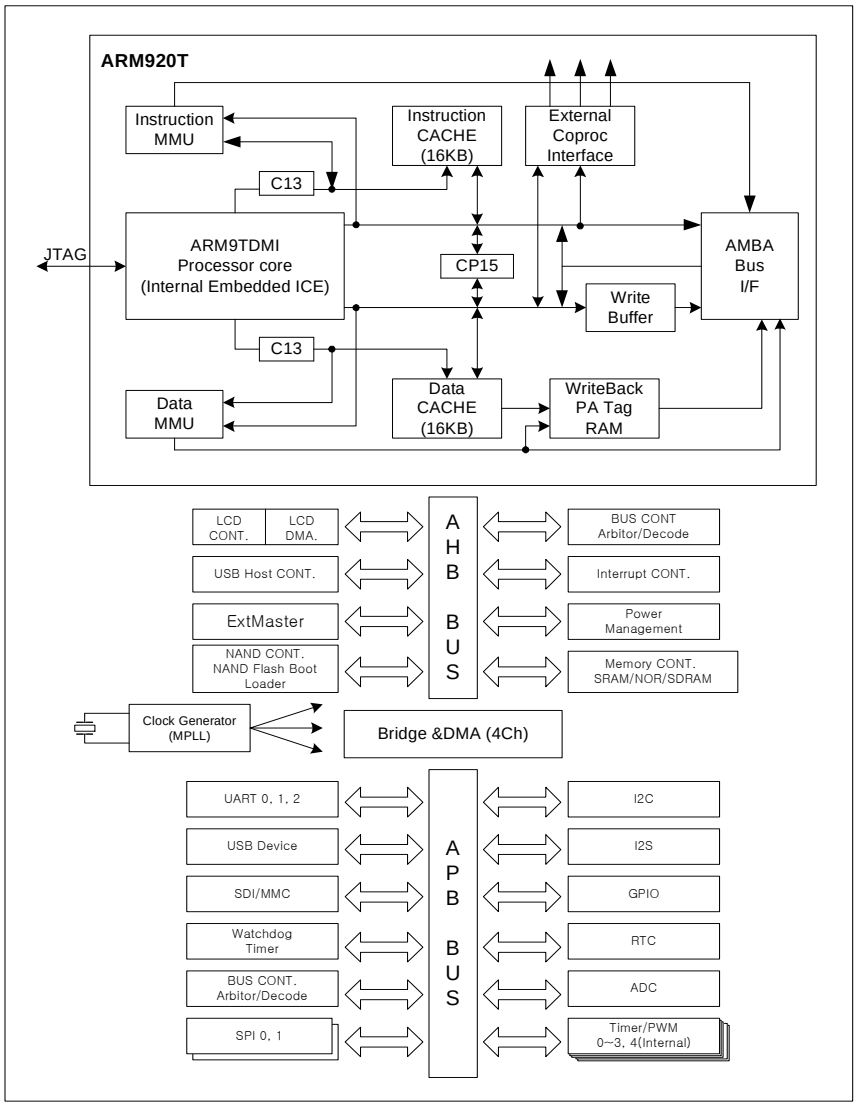
(2) S3C2410X

S3C2410X는 ARM920T 코어를 사용하여 주문형 집적회로로 구현한 마이크로 컨트롤러이다. <그림 4-1>은 S3C2410X의 내부 다이어그램이다.

S3C2410X는 32비트 마이크로컨트롤러로서 내부에 ARM 코어를 내장하고 있으며, LCD 제어 모듈과 PCMCIA 제어모듈 및 클럭 발생기를 내장하고 있어, 시스템의 소형과 및 저 전력화 등의 특성을 가지고 있다[11].

다음은 S3C2410X의 특성을 설명한다.

- 32-bit RISC 프로세서.
- 3.3V 입·출력 인터페이스.
- 저 전력 시스템.
- 3개의 UART 와 2개의 SPI 지원.
- 2포트의 USB 호스트와 1포트의 USB 디바이스 컨트롤러.
- PLL을 포함한 클럭 발생기 내장.
- 전력 관리 특성.
- 리틀/빅 엔디안(Little/Big endian) 지원.
- 부팅을 위한 다양한 롬 타입(Nor/Nand Flash, EEPROM, other) 지원.
- 32엔트리 MMU(윈도우CE, EPOC 와 리눅스) 제공.

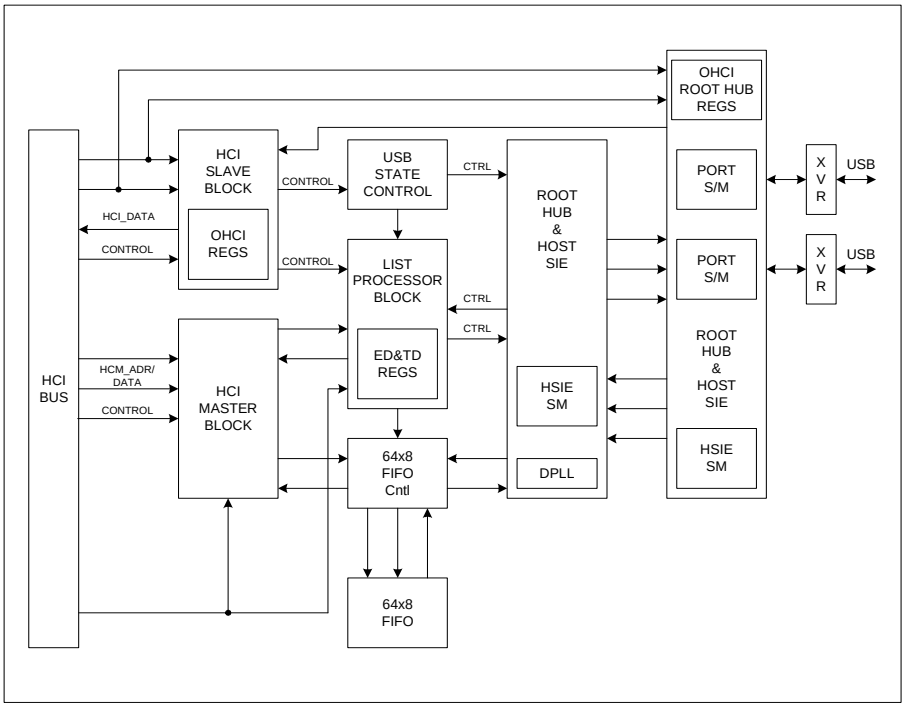


<그림 4-1> S3C2410X의 블록 다이어그램

<Fig. 4-1> S3C2410X block diagram

(3) USB 호스트 및 디바이스 컨트롤러

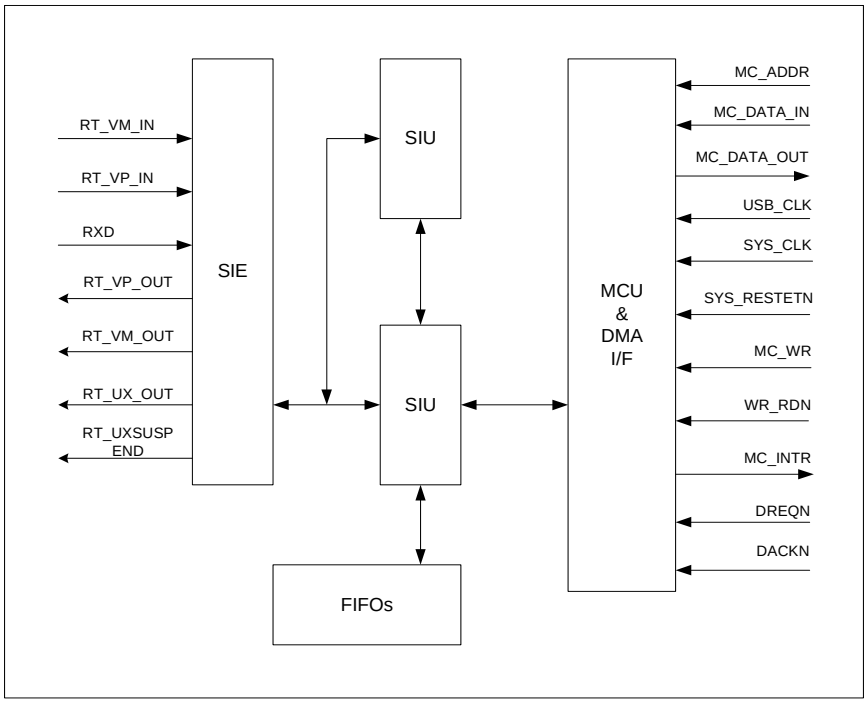
S3C2410X는 2포트의 USB 호스트 인터페이스를 제공한다. <그림 4-2>는 호스트 컨트롤러의 블록 다이어그램을 보여주고 있다.



<그림 4-2> USB 호스트 컨트롤러 블록 다이어그램

<Fig. 4-2> USB host controller block diagram

USB 디바이스 컨트롤러는 DMA 인터페이스를 이용한 제어 방식을 사용하며 컨트롤러는 벌크, 인터럽트, 컨트롤 전송을 모두 지원한다. 다음 <그림 4-3>은 디바이스 컨트롤러의 블록 다이어그램을 나타낸다[11].



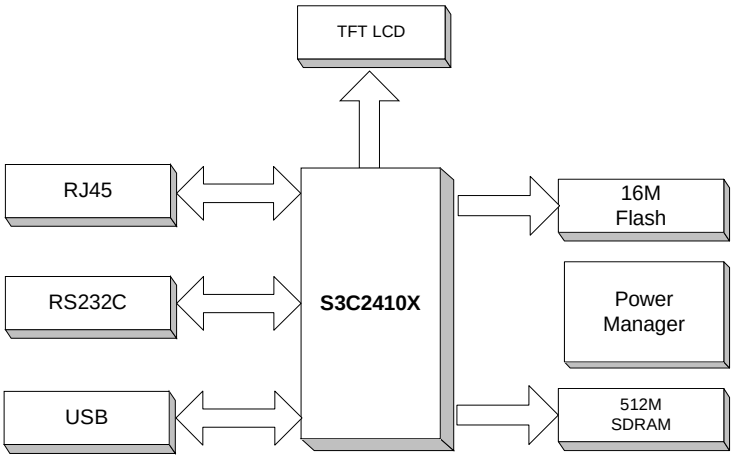
<그림 4-3> USB 디바이스 컨트롤러의 블록 다이어그램

<Fig. 4-3> USB device controller block diagram

4.1.2 평가 보드의 구성

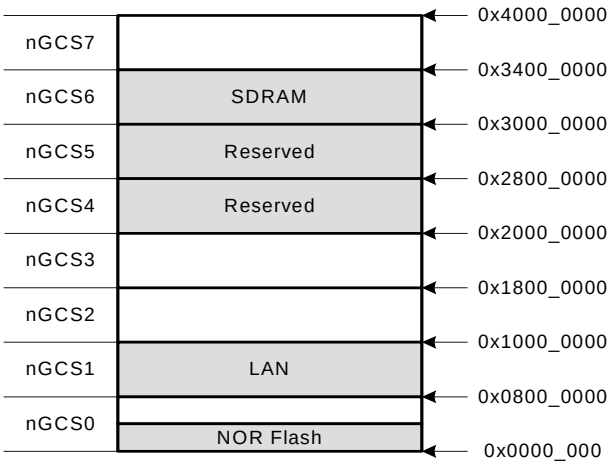
<그림 4-4>는 평가 보드의 블록 다이어그램을 나타낸 것이다. 커널 포팅과 응용프로그램을 위한 MCU는 S3C2410X를 이용하였으며, USB 호스트, 디바이스 컨트롤러는 칩 내부에 내장이 되어 있다. 커널과 실행파일들을 다운로드 받기 위해 RS232C 통신이 가능하게 하였으며 고속의 다운로드를 위하여 랜 포트를 이용할 수 있게 하였다. LCD를 위한 연결 단자도 존재한다.

<그림 4-5>은 NOR 플래쉬 메모리를 이용하였을 경우에 평가보드의 메모리 맵을 보여주고 있다.



<그림 4-4> 평가보드 블록도

<Fig. 4-4> Block diagram of Evaluation board



<그림 4-5> NOR 플래쉬를 이용한 메모리 맵

<Fig. 4-5> Memory map using NOR Flash

4.2 임베디드 리눅스 포팅

일반적으로 임베디드 시스템에 운영체제를 포팅하기 위해서는 먼저 크로스 개발환경을 갖춰야 한다. 즉 임베디드 시스템에서 직접 컴파일과 실행이 가능하지 않기 때문에, 임베디드 시스템과 같은 환경의 호스트 컴퓨터에서 크로스 컴파일러를 이용하여 커널과 응용프로그램을 컴파일 한 후 임베디드 시스템에 다운로드를 하여 확인하는 과정을 거쳐야 한다. 다음은 크로스 개발 환경에 대해 기술하였다.

4.2.1 통신 에뮬레이터 설정

리눅스에서는 평가보드의 부팅과정을 확인하기 위하여 미니콤 이라는 리눅스 통신 에뮬레이터 프로그램을 사용한다. 이 프로그램으로 평가보드와 호스트간의 데이터 전송 및 모니터링이 가능하다.

4.2.2 크로스 컴파일러

일반적으로 컴파일러는 자신의 시스템에 맞는 바이너리 코드를 만드는 작업을 수행한다. 임베디드 시스템은 중앙처리장치의 용량과 메모리 공간이 한정되어 있기 때문에, 타겟보드에서 직접 컴파일을 하여 실행하는 것이 불가능하다.

그 결과로 임베디드 시스템용 커널 및 응용프로그램을 개발하기 위하여 호스트 시스템에 개발하고자 하는 임베디드 시스템의 아키텍쳐에 맞는 크로스 컴파일러 환경을 구축하여 사용한다.

4.2.3 커널 설치 및 크로스 컴파일

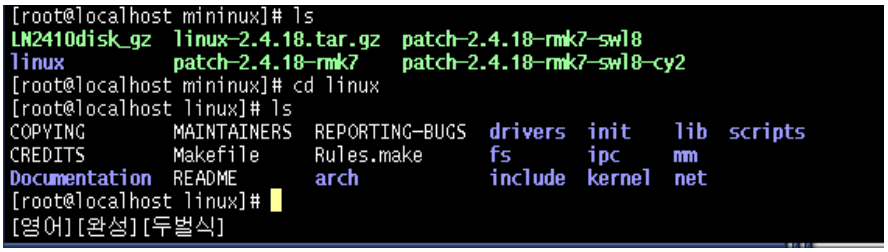
ARM 코어용 리눅스 커널은 i386 리눅스와 전혀 다르게 설계되고 코딩되어 있는 것이 아니며, 리눅스 커널 자체가 이식성이 뛰어나기 때문에 새로운 커널을 디자인 할 필요는 없다. 하지만 대용량의 PC 혹은 웍스테이션에서 사용되는 커널에는 불필요한 것들이 많이 포함되어 있어서 임베디드 시스템에 포팅을 하고자 한다면 이러한 대용량의 커널을 임베디드 시스템에 적재 가능하게 최적화 시켜야 한다.

다음은 리눅스 커널을 구성하는 순서이다[6], [10].

- ARM용 리눅스 커널을 구성한다.
- ARM 패치를 수행한다.
- 타겟 보드를 위한 패치를 작성하여 수행한다.
- 타겟보드에 맞게 커널을 수정하였다면, 컴파일 옵션이나 환경 설정에 관련된 사항을 수정해 주어야 한다. 이 과정은 menu config 에서 이루어진다.

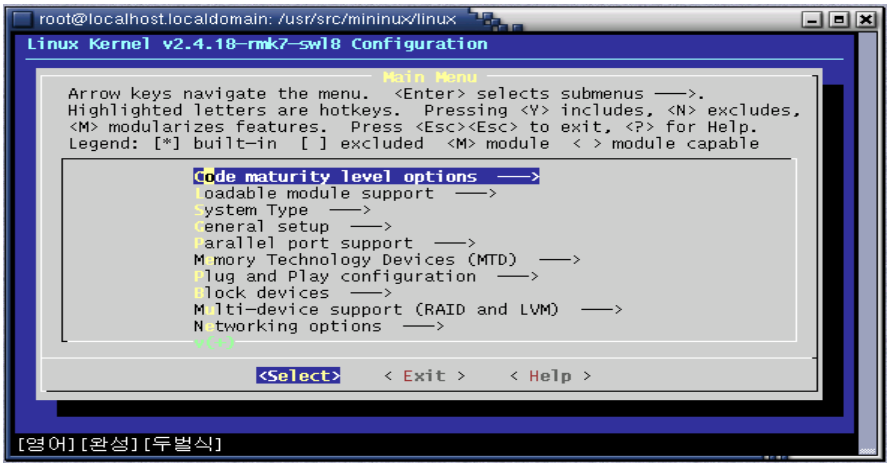
ARM용 리눅스 커널과 패치 및 GCC 컴파일러 등은 인터넷에서 다운받아 사용할 수 있다. 다음 <그림 4-6>은 커널 구성과 패치에 필요한 파일들과 커널 소스 패치후 소스 디렉토리 내부의 모습을 보여주고 있다.

커널 패치가 성공적으로 끝나면 GCC 컴파일러를 이용하여 타겟보드에 적제하기 위하여 커널 이미지를 만들어야 한다. 일반적으로 zImage라고 하며 이미지 생성 후 타겟보드에 맞게 이미지의 환경을 설정 해 주어야 한다. 다음 <그림 4-7>에서 커널 컴파일 후 타겟보드에 적용될 환경 구성 설정을 보여주고 있다.



<그림 4-6> 커널 소스

<Fig. 4-6> Kernel source



<그림 4-7> 커널 환경 구성

<Fig. 4-7> Kernel environment configuration

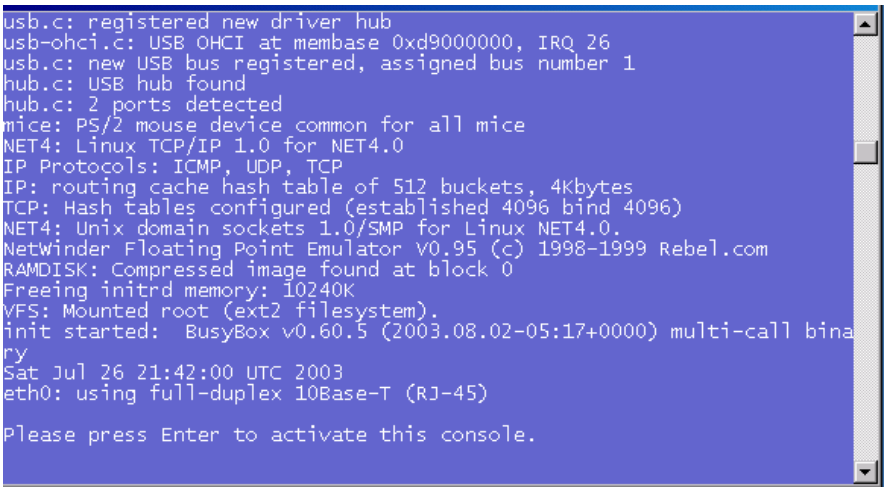
4.2.4 커널 다운로드

타겟보드는 IP 주소가 고정되어 있지 않으므로 이를 수동으로 할당하거나 IP 주소를 자동으로 할당받아야 한다. 이때 IP 주소를 할당받을 수 있는 방식은 BOOTP (Bootstrap Protocol), DHCP (Dynamic

Host Configuration) 이 있다. 이중 BOOTP는 매우 간단한 프로토콜 구현으로 수행될 수 있으므로 부트로더와 같은 프로그램은 이것을 사용한다[6].

또한, 리눅스를 다운로드 하는 방식에는 시리얼을 이용하는 방식과 랜을 이용하는 방식이 있다.

다음 <그림 4-8>은 수정된 커널 이미지를 평가보드에 다운로드 하여 부팅이 성공하였을 때의 모습이다.



<그림 4-8> 커널 포팅

<Fig. 4-8> Kernel porting

4.3 USB 디바이스 드라이버

4.3.1 윈도우즈 환경에서 드라이버 개발

PC와 주변기기간에 통신을 할 수 있는 것은 디바이스 드라이버가 컴퓨터와의 인터페이스를 제공하고 있으며, 사용자가 이러한 디바이

스 드라이버의 내용을 알지 못해도 사용이 가능하도록 제품에서 제공된다. 윈도우를 사용하는 경우 그 내부 소스들이 공개되지 않기 때문에 마이크로 소프트에서는 일정 사용료를 내어 내부 커널들의 동작을 알 수 있는 일반 VxWorks나 다른 임베디드 환경에서 사용할 수 있는 운영체제와는 달리 사용자들이 운영시스템의 세부사항을 알 수 없으므로 보다 손쉽게 디바이스 드라이버를 제작할 수 있도록 DDK(Driver Development Kit)라는 툴을 만들어 제공한다. 이를 사용하면 예제를 이용하여 간단한 디바이스 드라이버를 제작할 수 있다. 특히 마이크로 소프트에서는 표준 드라이버(WDM : Window Driver Manager)들에 대한 필요한 기본적인 기능들을 DDK안에 넣었는데, 대표적으로 USB와 IEEE 1394에 관한 것이 있다[3].

4.3.2 윈도우즈 드라이버 개발 툴

DDK는 윈도우 98, 2000뿐 아니라 최근에 출시된 XP 버전도 지원한다. 임베디드 시스템과 데스크탑 컴퓨터간의 USB 통신을 위해서는 PC에서 사용되는 운영체제인 윈도우즈에서의 USB 드라이버를 개발해야 한다. 이때 드라이버를 손쉽게 구현할 수 있게 해주는 것이 DDK이다[13].

일반적으로 디바이스 드라이버는 VxD와 WDM으로 나뉜다. 전자는 윈도우95, 윈도우98에서 사용될 수 있는 것이며, WDM으로 만든 드라이버는 윈도우98, NT 계열에서 모두 사용 가능하다.

디바이스 드라이버를 구현하기 위해서는 하드웨어 인터럽트를 처리해야 한다는 것을 인지해야 하며, 중앙처리장치의 부하를 줄이기 위해서 DMA를 통한 메모리와 디바이스 사이의 연결을 잘 수행해야 한다. 또한 이와 더불어 핫플러깅에 대한 생각도 해야한다. 이전에는 주변기

기를 연결하였을 경우 다시 부팅 해야하고 윈도우가 가지고 있는 디바이스 드라이버 중에 연결된 드라이버를 찾아서 등록시키곤 했다. 그러나 지금은 다시 부팅 시킬 필요 없이 연결된 그 순간에 모든 것이 해결된다. 연결과 마찬가지로 분리도 그 순간에 모든 것이 해결된다.

4.3.3 INF 파일

(1) 구성

INF 파일은 설치에 관련된 모든 정보를 담고 있는 배치 파일이다. 설치할 파일, 위치, 변경 혹은 추가할 레지스트리까지 하나의 파일에 모두 담는다. 또한 드라이버뿐만 아니라 일반인도 간단히 설치할 수 있도록 구성할 수 있다. 공개 프로그램인 “ezpad”는 INF 파일로 설치할 수 있도록 구성되어 있다. INF 파일은 INI와 마찬가지로 섹션과 하부 엔트리 설정 값으로 구성되어 있다[3], [12].

INF는 최소한 16개 섹션을 갖게 된다. 설치 정보를 윈도우 계열과 NT 계열에 대해 별도로 갖고 있어야 하므로 별로 복잡하지 않다. 다음에서 INF 파일의 각 섹션에 대해 상세히 기술한다.

가) 문자열 섹션

INF 파일 내에서 사용할 문자열을 미리 정의한 부분으로, 별도로 문자열 섹션을 둔 이유는 한가지 언어에 제한되지 않도록 하기 위함이다. 엔트리 이름에는 ‘.’이나 공백을 넣을 수 있으므로 이름을 지을 때 무척 자유롭다. DDK 예제에서는 제품명이나 제작사 이름, 문자열 이름 사이에 점을 넣어 구분해 사용한다. 이 섹션은 보통 INF 제일 마지막에 위치한다. 다음 <표 4-1>는 제작 예를 나타내고 있다.

<표4-1> 문자열 섹션의 실제

<Table 4-1> Exam of String section

```
[Strings]
WdmFrame.DeviceDesc="WDM Frame Device"
WdmFrame.ClassName="WDM Frame Device"
;.....
```

위 표에서 “WdmFrame.ClassNameF” 이 엔트리명이며 그 뒤의 문자열이 정의 할 값이다. 정의된 문자열을 “%문자열 정의명%” 형식으로 사용한다. 문자열 “WDM Frame Device”를 사용하고자하면 다음부터 “%WdmFrame.ClassName%”을 적어주면 된다.

나) 버전 섹션

모든 INF 파일에 공통으로 들어가는 섹션으로, 다음과 같은 하부 엔트리로 구성되어 있다.

- Signature : 설치할 운영체제를 명시한다. $Windows NT$, $Chicago$, $Windows 95$ 중에 하나만 적어야 한다. 대소문자를 가리므로 틀리지 않도록 적어야 한다. 여러 윈도우 버전에서 사용하기 위해서는 $Windows NT$만 사용해도 별 문제가 없다.
- Class : 드라이버 종류를 써줘야 하며 개발하고자 하는 디바이스 USB라고 적는다.
- ClassGUID : GuidGen.exe로 생성한 guid를 기록한다.
- Provider : INF 파일 작성자 이름이나 드라이버 제작사의 이름을 적어준다. 문자열 섹션에 미리 정의하고 해당 엔트리로 대체한다.

다) 제조자 섹션

하위 섹션에 영향을 주는 섹션이다. 여기서부터 설치정보가 시작된다.

```
%WdmFrame.Manufacturer%=WdmFrame
```

라) “SourceDisksNames” 섹션

설치할 때 사용할 타이틀과 파일 위치를 기록한다. 파일위치는 어디나 명시할 수 있지만 굳이 정하지 않아도 문제가 없다. 따라서 타이틀만 기록하고 나머지는 생략한다. 사용한 엔트리 이름은 다음 섹션인 “SouceDisksFiles” 섹션에서 참조한다. 엔트리 이름은 설치할 디스켓의 번호로, 작성자가 임의로 결정할 수 있다.

```
[SourceDisksNames]
1="WDM frame driver installation diskette", "",,
```

마) “SourceDisksFiles” 섹션

복사할 파일을 적는다. 엔트리명은 해당 디스켓에 들어있어야 할 파일명이고 엔트리 값은 “SourceDisksNames” 섹션에서 사용한 엔트리명이다.

```
[SouceDisksFiles]
WdmFrame.sys=1
```

바) “DestinationDirs” 섹션

파일이 복사될 위치를 정한다

```
[destinationDirs]
파일리스트 섹션 = drid [,subdir]
```

이 섹션의 엔트리도 다른 섹션이 참조하는데 파일 리스트 섹션의

이름으로는 Copy Files, Rename Files, Delete Files 중 하나가 올 수 있다. 해당 섹션이 처리할 파일을 여기에 기록하는 것이다. drid에는 미리 정의된 위치 값을 적으며 <표 4-2>에 나오는 값 중하나를 사용한다. 중요한 몇 가지만을 표에 나타내었다.

<표 4-2> "DestinationDirs" 섹션의 값

<Table. 4-2> Value of "DestinationDris" section

값	해당위치
0xFFFF	INF가 위치한 현재 디렉토리
10	윈도우즈 디렉토리
11	%windir%\system32
12	드라이버 디렉토리
30	부팅 파티션의 루트

(2) INF 파일 내부 섹션간의 참조 관계

INF를 이해하기 위해서 무엇이 어디에 포함되는지 이해해야 한다. 다음 <그림 4-9>은 INF 섹션간에 참조 관계에 포함되는 순서이다. 제조자(Manufacturer)가 시작되며 참조 문자열의 시작이다. 참조의 기본 원칙은 상위 섹션의 엔트리 값이 다음 하위 섹션의 이름이 된다.

```
[Version]
[Strings]
[SourceDisksNames]
[SouceDisksFiles]
[DestinationDirs]

[Manufacturer]
        Nobody Make = Nobody.Make
                      [Nobody.Make]
                      Driver 1 = Driver1.Install, ID
                      CopyFiles = Driver1.CopyFiles
                      AddReg =Driver1.AddReg
                                    [Driver1.CopyFiles]
                                    ; …
                                    [Driver1.CopyFiles]
                                    ; …
```

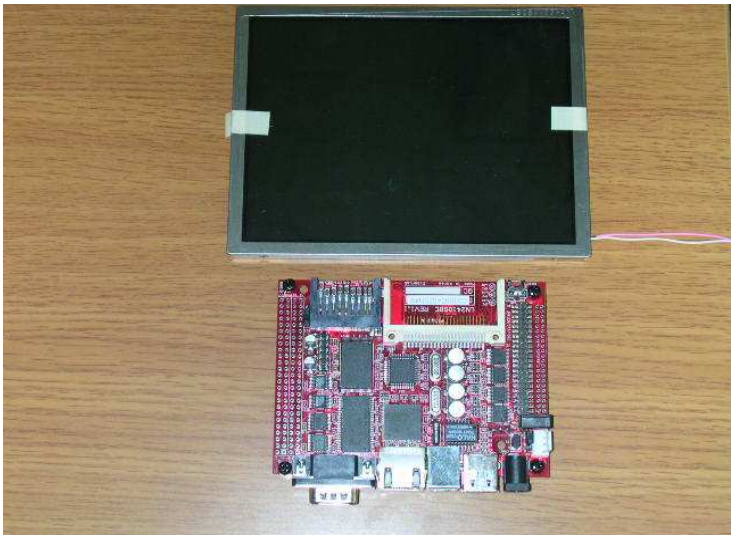
<그림 4-9> INF 참조 관계

<Fig. 4-9> INF reference relationship

4.4 USB 드라이버 제작 및 시험

평가 보드는 4장에서 설명한 스펙을 기준하여 주문 제작하였다. 제작된 평가보드에 3장에서 설명한 과정을 통하여 임베디드 리눅스를 포팅을 한 후 호스트 PC와 USB 케이블을 이용하여 통신을 하고자 하였다. 이때 호스트 PC는 현재 상용되고 있는 윈도우즈2000을 운영체제로 갖춘 PC를 이용하였다. USB 드라이버는 2000 DDK와 VC++을 이용하여 제작하였으며, 제작된 드라이버의 정상작동을 확인하기 위하여 읽고 쓰기가 가능한 예제를 사용하여 평가보드와 호스트 PC간의 USB 통신을 하여 드라이버의 정상작동을 확인하였다.

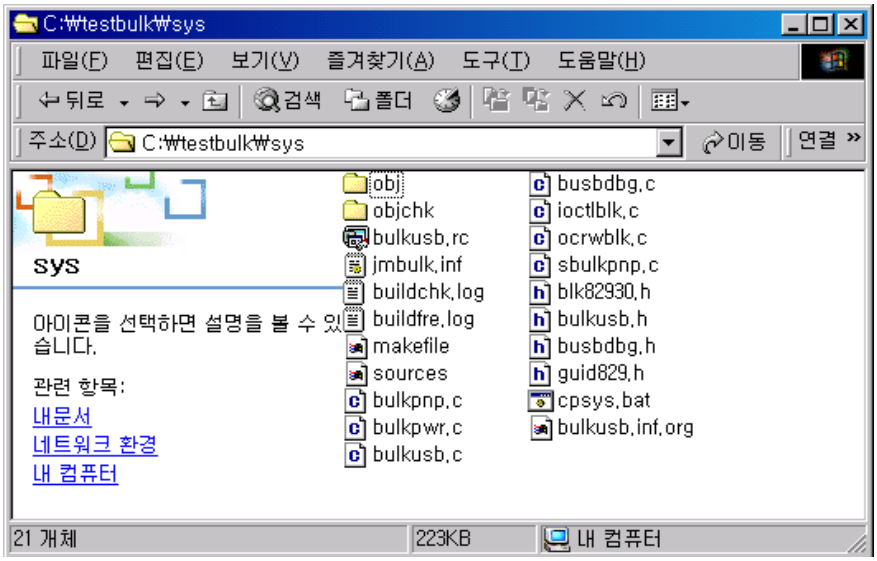
다음 <그림 4-10>은 평가 보드와 TFT LCD의 모습을 보여주고 있다.



<그림 4-10> 평가 보드와 LCD

<Fig. 4-10> Evaluation board & LCD

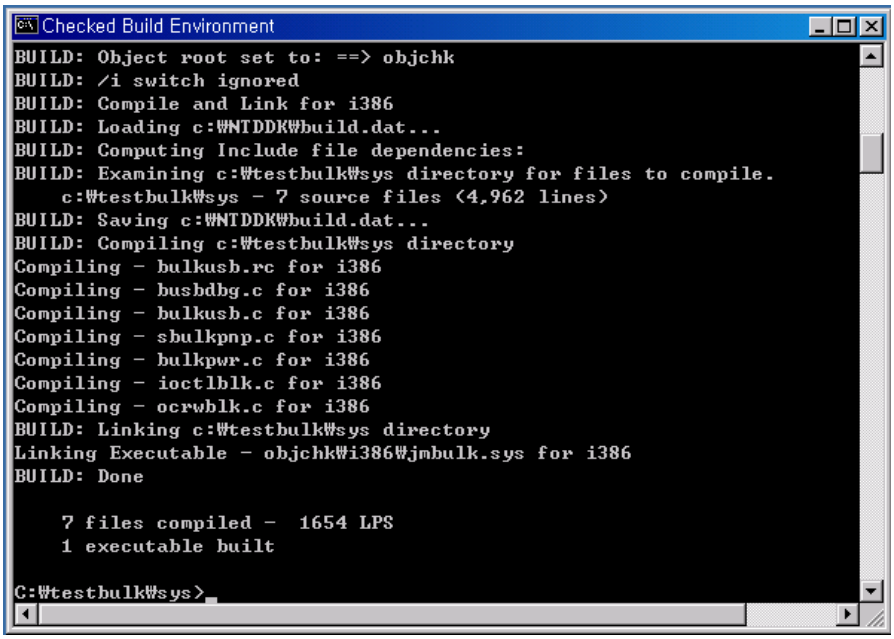
USB를 이용한 통신형태는 벌크, 등시성, 인터럽트, 제어 전송이 가능하다. 본 논문에서는 벌크 전송 방식을 이용한 USB 드라이버를 제작하였다. 벌크 통신을 위한 드라이버는 2000 DDK에서의 기본 예제를 타겟보드에 맞게 변형하여 제작하였다. 다음 <그림 4-11>는 벌크 통신을 위한 드라이버를 만들기 위한 소스 파일들이다. 이 소스들을 DDK를 이용하여 생성하면 sys 파일이 생성된다. 또한 sys 파일을 설치하기 위한 INF 파일은 INF 파일 생성 마법사를 이용하여 작성할 수 있으며, 직접 만들 수도 있다.



<그림 4-11> 벌크 드라이버 소스

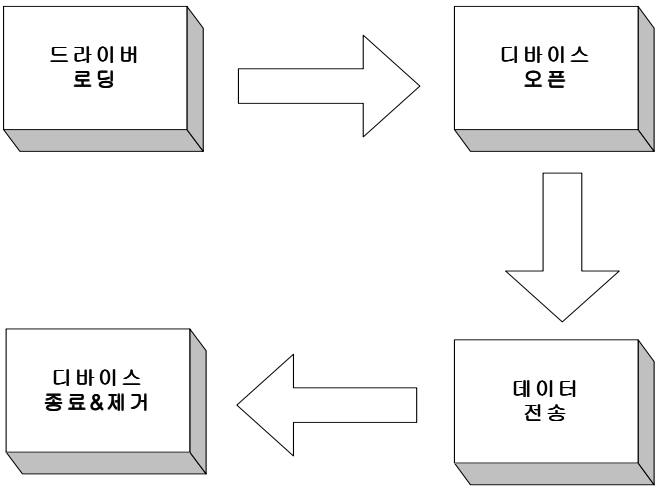
<Fig. 4-11> Bulk driver source

다음 <그림 4-12>은 소스들을 DDK와 VC++를 이용하여 빌드 한 결과이다. 빌드가 성공하면 jmbulk.sys 파일이 생성된다. 이 파일과 먼저 만들어 놓았던 jmbulk.inf 파일을 이용하여 호스트 PC에 제작한 드라이버를 설치한다.



<그림 4-12> 드라이버 파일 생성

<Fig 4-12> Driver file creation

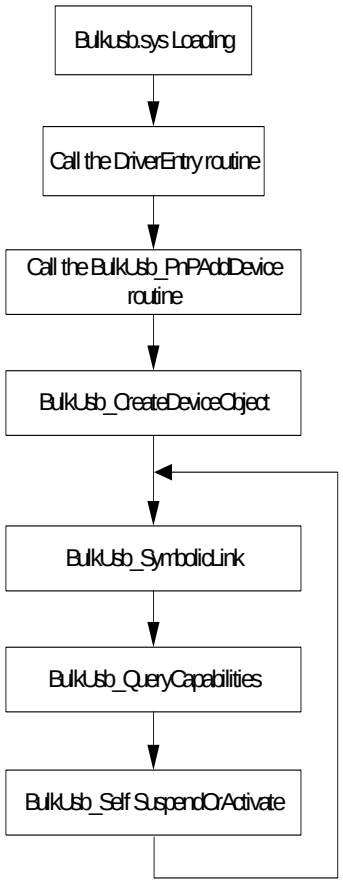


<그림 4-13> 벌크 드라이버 동작

<Fig. 4-13> Bulk driver operation

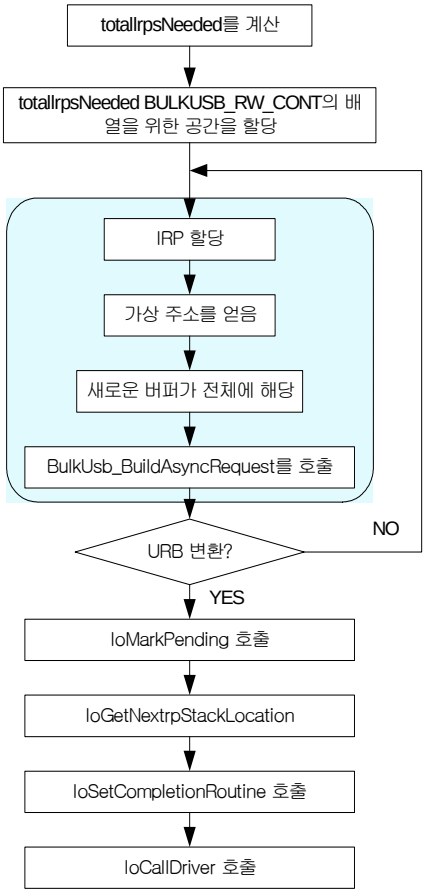
<그림 4-13>는 벌크 드라이버가 동작하는 과정을 나타내고 있다. 이 드라이버는 총 4가지의 동작을 하는데 운영시스템에 의해서 로딩된 후 드라이버에 대한 디바이스를 열어 데이터를 전송한 후 디바이스를 종료시킨 후 운영체제에서 제거하는 동작을 수행한다.

다음 <그림 4-14>는 드라이버의 로딩시 순서도를 나타내고 있으며, <그림 4-15>은 데이터 전송시 순서도를 보여주고 있다.



<그림 4-14> 로딩 순서도

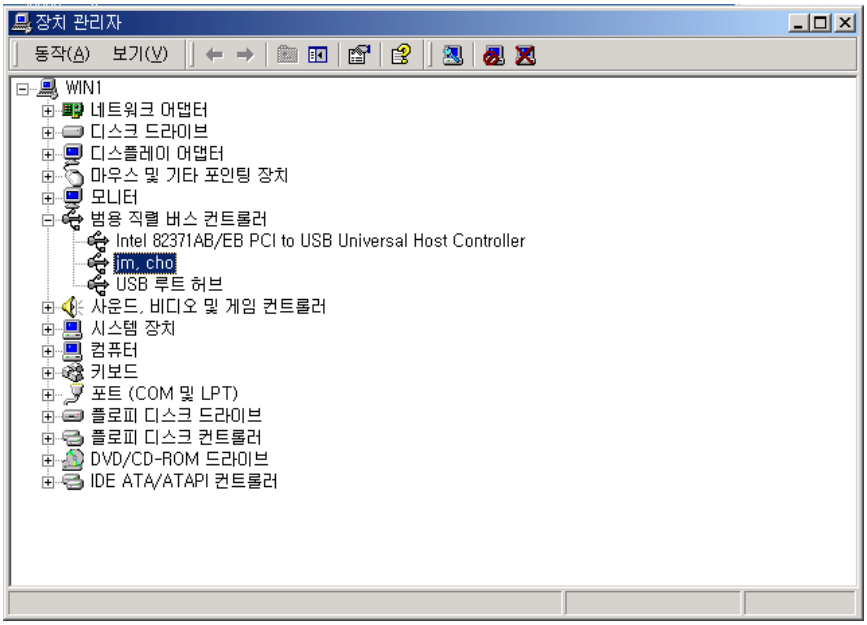
<Fig. 4-14> Loading flowchart



<그림 4-15> 전송 순서도

<Fig. 4-15> Sending flowchart

다음 <그림 4-16>은 실제 윈도우 2000 환경에서 제작한 드라이버를 설치한 후 장치 관리자에서 확인한 화면이다. 여기에서 새로 설치되어 있는 USB 장치 중 “JM, CHO” 드라이버를 확인 할 수 있었다. <그림 4-17>은 새로 설치한 드라이버의 등록정보를 확인하고 있는 화면이다. 새로 설치한 드라이버가 제대로 동작하고 있음을 보여주고 있다.

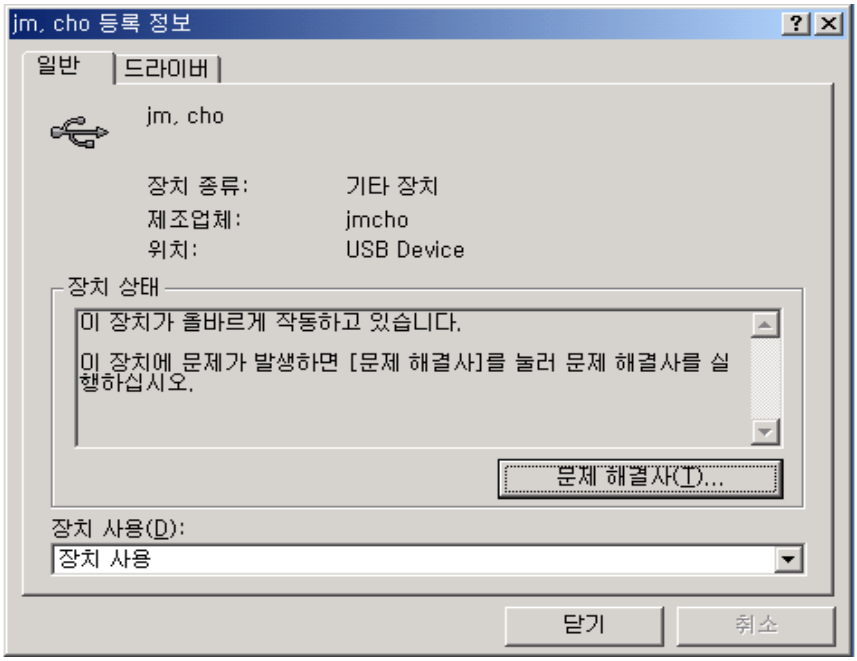


<그림 4-16> 장치관리자에서 확인

<Fig. 4-16> Confirmation at the device manager

USB 벌크 드라이버 설치 후 윈도우 등록정보에서도 확인이 가능하지만 VID(Vender ID), PID(Product ID), 전송속도, 전송방식 등을 확인하기 위하여 “usbviewer” 프로그램을 사용하여 USB 드라이버 설치 정보를 확인해 보았다[13]. 다음 <그림 4-18>은 설치 후 USB 상태를 확인한 모습이다. 여기에 보이는 박스는 현재 드라이버의 VID,

PID, 와 데이터 전송방식, 최대 버퍼 크기, 전송속도와 같은 정보들을 보여주고 있다.

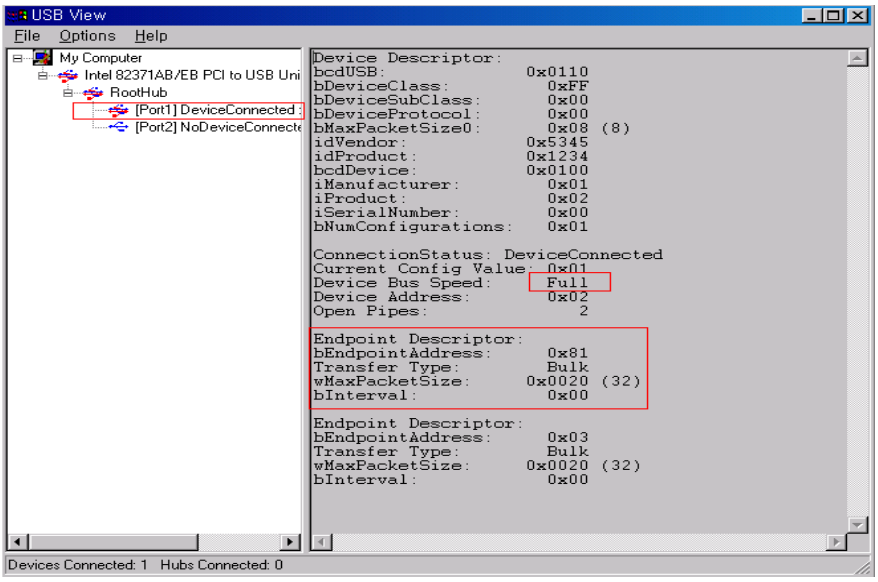


<그림 4-17> USB 등록정보

<Fig. 4-17> Registration information of USB

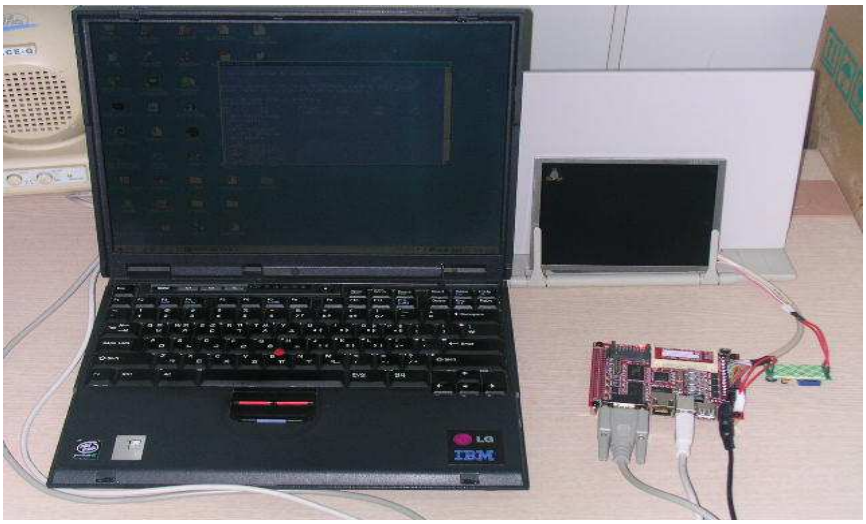
다음 <그림 4-19>는 호스트 PC와 평가보드를 USB 케이블을 통하여 연결하여 통신을 하고 있는 모습을 보여주고 있다.

앞서 개발했던 벌크 통신 드라이버를 통하여 통신을 할 수 있는 예제를 이용하여 호스트 PC와 평가보드 사이에 통신을 해보았다. 다음 <그림 4-20>는 통신예제를 이용하여 USB 케이블을 통해서 평가보드에 데이터를 쓰고, 그 데이터를 다시 평가보드로부터 읽어 들이는 상황을 보여주고 있다.



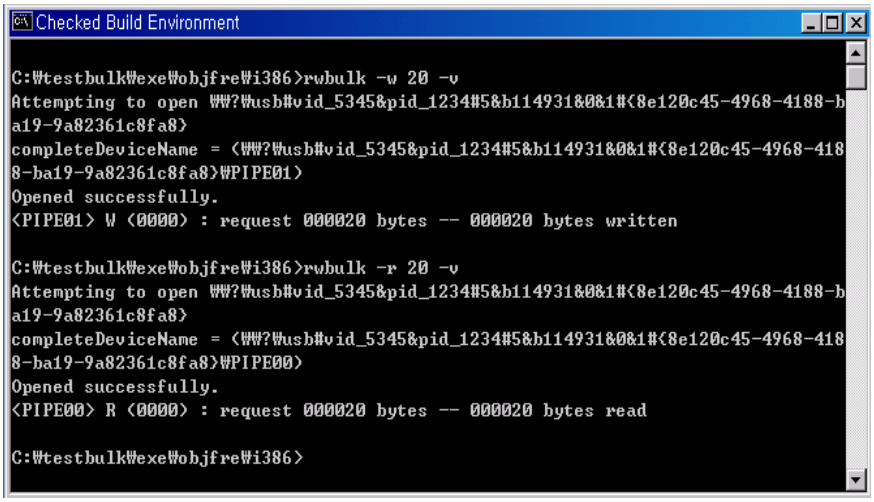
<그림 4-18> USB 드라이버 상태

<Fig. 4-18> USB driver state



<그림 4-19> 평가보드의 통신

<Fig. 4-19> Communication of evaluation board



<그림 4-20> USB 예제 실행

<Fig. 4-20> USB example execution

제 5 장 결 론

기존의 임베디드 운영체제는 고가이면서 커널 소스가 공개되지 않아 프로그래머가 커널을 수정하고자 하여도 재구성이 어려운 문제점을 지니고 있었다. 그러나 커널 소스가 공개되어 있어 효용성과 신뢰성과 확장성을 가지고 있는 임베디드 리눅스는 임베디드 시스템 구축에 있어 최상의 운영체제이다.

임베디드 시스템은 데스크탑 컴퓨터와의 데이터 전송, 수신을 위해서 현재 직렬, 병렬, IEEE 1394등의 방식을 사용해왔다. 하지만 지금의 방식은 많은 한계점들을 보여주고 있다. 앞의 두 방식은 전송속도와 신뢰성으로 인한 동영상, 사운드, 이미지 와 같은 자료들을 전송, 수신하기해서는 많은 시간의 낭비가 초래된다. IEEE 1394 방식은 동영상을 전송하기 위한 특별한 프로토콜로 속도는 빠르나 그 구성이 까다롭고 구현하기가 힘든 단점을 내재하고 있다.

본 논문에서는 이러한 단점들을 극복하고자 현재 급속히 발달되고 있는 USB 프로토콜을 이용하여 시스템간의 통신을 구현해보았다. USB 방식은 속도와 신뢰성을 두루 갖추고 있으며 또한 시리얼 라인으로 구성되어 있기 때문에 구현도 간단하다.

본 연구에서는 ARM MCU를 이용하여 현재 사용되고 있는 임베디드 시스템의 기본 보드를 구성하였으며, 이 평가보드와 PC와의 데이터 통신을 위해서 USB 프로토콜을 사용하였다. 또한 USB 통신을 위해서 평가보드에 적절한 USB 드라이버를 제작 해보았다.

또한 기존의 임베디드 시스템들은 호스트로 사용하기 위해서는 USB 호스트 컨트롤러를 따로 부착하여야 다른 임베디드 시스템과의 USB 연결이 가능하였다. 이렇게되면 시스템의 크기가 커지며, 펌웨어를 제작해야 하기 때문에 효용성이 떨어지나 S3C2410X를 사용하면

자체에 호스트 컨트롤러를 포함하고 있기 때문에 시스템의 확장 없이도 구현이 가능하다.

임베디드 시스템이 호스트 역할을 하게되면 우리가 흔히 사용하고 있는 USB 플래쉬 메모리와 같은 휴대 저장 장치들도 직접 임베디드 시스템에 연결하여 사용이 가능하다.

USB 통신을 위하여 벌크 전송방식의 데이터 전송을 위하여 드라이버를 구현해보았으며, 통신 예제를 통하여 드라이버의 정상동작을 확인 할 수 있었다.

이러한 기술을 이용하여 좀더 대용량, 즉 동영상 같은 고속통신이 가능하며, 또한 신뢰성을 필요로 하는 대량 데이터도 전송의 가능성도 확보하였다.

본 논문에서는 임베디드 시스템은 리눅스를 사용하였으며 USB 통신을 위한 호스트 PC는 윈도우 환경이었다. 향후에는 리눅스 호스트의 환경에서의 드라이버를 개발해보고자 하며, USB 2.0 프로토콜에 대해서 연구해볼 것이다.

참 고 문 헌

[1] 이민석, “모바일 기기를 위한 임베디드 리눅스”, 한국정보처리학회지, 9권, 1호, pp.112～120, 2002.

[2] “USB 기초자료”, http ://esacademy.co.kr/html/USB-KR-ST.htm

[3] 김영훈, 「USB 가이드」, OHM사, pp.46～83, 2002.

[4] “USB specification, ver1.1”, http://www.usb.org, pp.107～142

[5] 노영욱외 1명, “임베디드 리눅스 개발도구 기술동향”, 정보처리학회지, 제9권, 제1호, pp.35～42, 2002.

[6] 장정윤, “임베디드 리눅스를 이용한 TCP/IP 기반의 원격 제어 시스템 구현에 관한 연구”, 석사학위논문, 한국해양대학교, pp.5～18, 2003.

[7] 조유근, 최종무, 홍지만, 「리눅스 매니아를 위한 커널프로그래밍」, (주)교학사, pp.24～110, 2002.

[8] 한성호, “임베디드 시스템에서 리눅스 커널을 위한 부트 로더에 관한 연구”, 석사학위논문, 성균관대학교, 2001.

[9] 김인성외 1명역,「리눅스 디바이스 드라이버」, 한빛 미디어, 2000

[10] 지현묵, 옥정훈 “ARM 기반보드에 리눅스 이식하기”, 마이크로소프트웨어, pp264～273, 2000.

[11] SAMSUNG, S3C2410 user manual, 2002.

[12] “INF 구성”, http://www.devguru.co.kr/

[13] “Kernel-Mode Drivers : Windows 2000 DDK", Microsoft

[14] "CyberLab", http://www.armkorea.com

[15] Steve furber,「ARM system-on-chip architecture」, Addison-Wesley, 2000.

[16] 정보통신 종합 정보 센터, “USB 기술 및 표준화 동향”

[17] “USB specification, ver2.0”, http://www.usb.org

[18] 성원호 역,「임베디드 시스템 펌웨어 분석」, CMPBooks, 2002.

[19] 이연조, 「임베디드 리눅스 프로그래밍」, PCBOOK, 2002.

[20] 성원호역「Embedded Systems Building Blocks」, CMPBooks, 2002.

[21] Epplin. J, "Linux as an Embedded Operating System", Embedded Systems Programing, Vol. 10, No. 10, pp.96~105, 1997.

[22] "USB Design By Example" http://www.usb-by-example.com

[23] "WDM USB Driver Interface", Microsoft

[24] 김선자, 김홍남, 김채규“정보가전용 임베디드 운영체제 기술”, 한국통신학회 논문지 18권, 12호 pp72~82, 2001.

[25] 김용수, “임베디드 시스템에 적합한 저전력 마이크로 컨트롤러 코어의 설계”, 석사학위, 한국과학기술원, 1999.

감사의 글

무언가를 시작해서 이제 그 결실을 보게 되었습니다. 이 시점에서 그 동안의 고마움을 표현하고저 이렇게 글을 올립니다.

우선 석사 2년동안 많이 부족한 저에게 진리탐구의 의미를 일깨워 주신 임재홍 지도교수님, 바쁘신 와중에도 세세히 논문을 심사해주신 양규식 교수님, 이상배 교수님과 다른 전자통신공학과 모든 교수님께 감사 드립니다.

2년간 같이 고락을 해온 DCN 연구실 모든 선배님들과 동기, 후배님들 특히 바로 옆에서 지켜봐 주신 지금은 외국에서 열심히 공부중인 송아늑나, 임신한 몸으로 끝까지 논문을 봐주신 선영늑나, 하드웨어에 대한 지식과 자료들을 주신 희철 선배님께 감사 드립니다.

그리고 많은 걱정과 관심을 보여주신 나이 많은 동기 성훈이형, 학교에 자주는 아니지만 오면 늘 재정적 지원을 아끼지 않았던 용철이, 틱격태격하며 정이든 형석이, 이런 동기들과 선배라는 이유만으로 못난 나를 따라주었던 원희, 재석, 슬기, 지금은 다들 졸업하신 박사 이태오, 진성호 선배님께도 감사 드립니다.

객지에서 서로 외로움을 달랠 수 있었던 오랜 벗인 안테나 실험실의 인용이, 전문적 지식을 많이 알려준 초등학교 친구 형남이, 어쩌다 서울로 찾아가면 기꺼이 맞아주는 고등학교 친구들 영진이와 주영이, 일과후 피터지게 혈전을 치뤘던 지금은 졸업한 정윤형, 학과 사무실로 찾아가 매일 괴롭혀도 싫은 내색하지 않은 조교 현진 누나께도 감사 드립니다.

끝으로 제가 하는 일에 대해서 묵묵히 믿고 지켜봐 주셨던 부모님과 가족들께 감사 드립니다. 이번의 결실을 맺을 수 있었던 것은 이 모든 분들의 믿음과 소중한 마음들 때문이었던 것 같습니다. 앞으로 마음속 깊이 이 소중한 마음들을 간직하며 살아갈 것입니다.

2004년 2월 재만이가 모든 고마운 이들에게...