

工學碩士 學位論文

임베디드 리눅스를 이용한 TCP/IP 基盤의  
遠隔 制御시스템 具現에 관한 研究

A Study on Implementation of TCP/IP based  
Remote Control System using Embedded Linux

指導教授 金 基 文

2003年 2月

韓國海洋大學校 大學院

電子通信工學科

張 晶 允

工學碩士 學位論文

임베디드 리눅스를 이용한 TCP/IP 基盤의  
遠隔 制御시스템 具現에 관한 研究

A Study on Implementation of TCP/IP based  
Remote Control System using Embedded Linux

指導教授 金 基 文

2003年 2月

韓國海洋大學校 大學院

電子通信工學科

張 晶 允

本 論 文 을 張 晶 允 의 工 學 碩 士  
學 位 論 文 으 로 認 准 함 .

委 員 長 梁 圭 植



委 員 林 宰 弘



委 員 金 基 文



2003年 2月

韓國海洋大學校 大學院

電子通信工學科 張 晶 允

# 목 차

Abstract

제 1 장 서 론 .....	1
1.1 연구의 배경 .....	1
1.2 연구의 목적 및 내용 .....	2
제 2 장 임베디드 네트워크 .....	4
2.1 임베디드 네트워크 환경 .....	4
2.2 임베디드 리눅스 커널의 개요 .....	5
2.3 임베디드 리눅스 부팅 코드 분석 및 부팅과정 .....	23
제 3 장 원격 제어시스템의 설계 및 구현 .....	26
3.1 시스템의 개요 .....	26
3.2 임베디드 시스템 포팅을 위한 환경 구성 .....	29
3.3 SA-1110의 분석 및 회로의 구성 .....	36
3.4 시험회로의 평가 .....	43
제 4 장 결 론 .....	50
참 고 문 헌 .....	52
부       록 : 타겟 보드 시스템 회로도 .....	53

## 표 차 례

<표 2-1> 임베디드 운영체제의 종류 .....	5
<표 3-1> 미니콤 환경 설정 값 .....	29
<표 3-2> 타겟 보드 메모리 맵 .....	41

## 그 림 차 례

<그림 2-1> 리눅스 커널의 내부구조 .....	7
<그림 2-2> 리눅스 커널의 소스 트리 구조 .....	8
<그림 2-3> 프로세스 상태 전이도 .....	11
<그림 2-4> 실행 상태 구분 .....	12
<그림 2-5> 가상 메모리 구조 .....	15
<그림 2-6> 물리 메모리 .....	16
<그림 2-7> 리눅스 네트워킹 계층 .....	18
<그림 2-8> 데이터 캡슐화 과정 .....	19
<그림 2-9> 디바이스 드라이버 .....	20
<그림 2-10> main 소스의 구성도 .....	24
<그림 3-1> 시스템 개요 .....	27
<그림 3-2> 시스템 부팅 과정 .....	28

<그림 3-3> 미니콧 실행 .....	30
<그림 3-4> 크로스 컴파일러 환경 설정 .....	31
<그림 3-5> 커널 설치 과정 .....	32
<그림 3-6> Menuconfig 구성 .....	33
<그림 3-7> BOOTP 데몬 .....	34
<그림 3-8> TFTP 전송환경 .....	35
<그림 3-9> SA-1110의 내부 다이어그램 .....	36
<그림 3-10> 평가회로의 SA-1110의 메모리 맵 .....	38
<그림 3-11> CS8900A의 내부 다이어그램 .....	40
<그림 3-12> 평가회로의 블록도 .....	42
<그림 3-13> 평가회로의 SA-1110의 메모리 맵 .....	42
<그림 3-14> 시스템 전송 절차 .....	43
<그림 3-15> 평가회로의 동작 운용도 .....	44
<그림 3-16> Ping 테스트 결과 .....	45
<그림 3-17> 서버 루프백 실험결과 .....	46
<그림 3-18> 클라이언트 루프백 실험결과 .....	46
<그림 3-19> 임베디드 리눅스 포팅 결과 .....	47
<그림 3-20> 평가회로 사진 .....	48
<그림 3-21> 시스템의 평가 절차 .....	48
<그림 3-22> 제어명령 전송 및 응답 .....	49
<그림 3-23> 제어명령 실행 .....	49

## Abstract

Embedded system is defined as the device of executing limited and special function and composed of embedded type on system. Recently, embedded systems are interconnected by means of network. Connecting type of embedded system forms network group into embedded system collection of one division. And, it's layer construction consists of upper network group which is gathered these groups.

Embedded OS is porting operating system in special purpose hardware. In the first stage, system was simple, therefore operating system wasn't needed. But recently system has been complex, and operating system concept appears to be important.

Embedded system must be satisfied with condition of ethernet port, serial and parallel port, small size, low cost. Best operating system of satisfying it in this condition is Linux.

This thesis is focused on network connecting technique of using embedded Linux and implementing remote control system base function using TCP/IP with manufactured target board. Analyzing embedded Linux kernel consisting of process management, file system, network management, device management, and we set up data for the development.

In this study, it implements remote control system base function using TCP/IP and StrongARM processor of INTEL INC. But, it suggested potentiality of system implementation using other processor embedded in Linux porting technique and establishment of control system design.

As this system develops, we can transmit remote data in embedded network environment of efficient and trustable. We consider embedded OS can be easily embedded in 16 bit and 32 bit processor.

# 제 1 장 서 론

## 1.1 연구의 배경

네트워크의 발달로 인하여 많은 기기들이 인터넷에 연결되어 있다. 하지만 아직까지 많은 기기들의 대부분이 RS-232C와 같은 매우 단순한 통신 프로토콜만을 지원하며, 이러한 단순한 통신 프로토콜은 직접 인터넷에 연결되기에는 충분하지 않다.

현재 대부분의 기기들은 PC나 워크스테이션을 경유하여 인터넷에 연결된다. 그 기기는 최소한의 요구되는 정보를 RS-232C와 같은 통신 포트를 통해 PC로 보내고, PC에 있는 웹 서버는 기기 정보를 수집하여, 그런 정보를 요청하는 클라이언트에게 전송한다. 이런 접근 방법은 기기에 새로운 연산 능력과 메모리 사이즈를 요구하지 않는 장점이 있다. 즉, 현관에 있는 센서에 PC를 연결한다면 PC는 너무 비싸고 큰 공간을 차지하게 된다.

따라서, 전용 임베디드 시스템을 개발하여 사용하기 위해서는 적용하고자 하는 어플리케이션의 목적에 따른 성능을 비교하여, 경제성 및 효율성, 향후 업그레이드되어야 하는 서비스의 기능 개발의 용이성 및 개발자원의 가용성 등을 충분히 고려하여야 한다.

한편, 기존의 임베디드 컨트롤러들은 내부의 제어요소가 제어루틴 형태로 구현되어 있다. 이러한 이유로 특정한 상황이 바뀌게 되면 해당되는 제어요소를 전부 바꾸던지, 제어루틴을 모두 바꿔야 하는 특정 상황을 맞게된다.

그러나, 임베디드 시스템은 특정한 전체 시스템에 포함됨으로써 해당 시스템 자체를 특정 상황에 대해 보다 지능적이고 유연하게 대처하게 할 수 있다. 기본적인 시스템의 핵심인 커널(kernel)만 설치가 된다면



제어부분 또는 시스템을 모두 바꿔야 하는 것이 쉽게 조정 가능하다.

기존의 운영체제는 커널 부분의 크기가 상당히 커서 임베디드 시스템이라 불리는 플랫폼에 적재하기에는 무리가 있다. 그리고, 그 크기를 줄이려면 시스템의 커널 부분을 조정 및 재작성을 해야 하는데, 대부분의 상용 운영체제는 커널의 소스를 공개하지 않는다. 따라서, 대부분의 임베디드 시스템의 운영체제는 리눅스를 사용한다.

## 1.2 연구의 목적 및 내용

임베디드 운영체제(embedded OS)는 특수목적용으로 만든 하드웨어에 포팅된 운영체제이다. 임베디드 시스템은 그 종류가 매우 광범위하므로 그 운영체제 역시 다양한 종류가 존재한다. 초기에는 임베디드 시스템이 비교적 단순해서 운영체제가 불필요했으나, 최근에는 멀티미디어 정보를 처리해야 하는 임베디드 시스템이 늘어나면서 그 역할이 매우 많아지고 복잡해졌기 때문에 운영체제 개념의 중요성이 대두되고 있다.

또한, 임베디드 OS중에서도 실시간 운영체제(RTOS : Real-Time Operating System)는 제한된 시간내에 작업이 이루어져야 하는 시스템으로 논리적인 정확성뿐만 아니라 시간적인 정확성이 요구되어진다. 특히, 제어시스템에 RTOS를 적용하여 시스템의 성능을 획기적으로 향상시킬 수 있으며, 이러한 요구조건에 의해 특별히 설계된 VxWorks(Windriver사), pSOS(ISI사), VRTX(Mento Grapics사), OS-9(Microwave System사)과 같은 RTOS도 개발되고 있다.

임베디드 RTOS는 여러 가지 기술이 개발되고 있으나, 현재까지는 고가의 비용과 취급기술의 난이도를 요구하고 있다. 이 중에서 리눅스(Linux)는 소스 공개로 운영되므로 현재 많은 분야에서 기술개발이 적

극 이루어지고 있으며, 내장형 RTOS로서 가능성과 기술적 안정성을 검증하고 있으며 개선이 이루어지고 있다.

16비트 또는 32비트 마이크로 컨트롤러에 적용시킬 수 있는 내장형 운영체제의 개발은 되고 있으나, 일부 기업위주로 이루어지고 있는 실정이므로, 본 논문에서는 공개용으로 사용될 수 있도록 하기 위한 목적과 아울러 기반기술의 확충을 목표로 하였다.

그러므로 본 논문에서는 원격 제어시스템에 적용될 수 있으며, 고가의 비용이 요구되지 않고, 기술적 안정성이 검증된 내장형 실시간 운영체제인 임베디드 리눅스 운영체제를 선택하였다. 통신 프로토콜은 대표적인 프로토콜인 TCP/IP를 사용하여 임베디드 네트워크를 형성하였다.

그리고, 임베디드 리눅스 커널을 분석하고, 타겟 보드를 구성하여 이를 포팅(porting)하였으며, 테스트 환경하에서 사용자코드를 실행시켜 커널의 포팅 완성도를 높이고자 하였다.

본 논문의 구성은 다음과 같다. 제 2 장에서는 임베디드 네트워크 환경 구성을 위해 임베디드 네트워크 환경에 대한 개요를 설명하였으며, 프로세서 관리자, 메모리 관리자, 파일시스템, 네트워크 관리자, 그리고 디바이스 드라이버의 장치관리자 이렇게 5가지 부분으로 구성된 임베디드 리눅스 커널을 살펴보고, 임베디드 리눅스의 부팅 과정 및 부팅 코드를 분석하였다.

제 3 장에서는 전체적인 시스템 구성을 고찰하고, 임베디드 네트워크를 설계하기 위하여 필요한 사항들을 정리하였다. 또한 INTEL 사의 마이크로프로세서인 SA-1110을 사용하여 시스템의 분석 및 이를 응용한 설계기술의 연구를 하였으며, RTOS 포팅을 위한 환경과 평가회로를 구성하여 그 성능을 평가하였다. 마지막으로 제 4 장에서는 결론을 도출하였다.

## 제 2 장 임베디드 네트워크

### 2.1 임베디드 네트워크 환경

임베디드 시스템은 임베디드 소프트웨어 응용 프로그램과 함께 사용되어 전용 동작 또는 제한되고 전문화된 기능을 수행하는 장치로 정의된다. 최근 인터넷을 기반으로 하는 응용들의 급격한 팽창에 의한 정보화 기반구조 흐름 중에 하나는 이러한 임베디드 시스템들이 네트워크를 수단으로 상호 연결되고 협력한다는 점이다.

한편 기술 발전에 힘입어 하드웨어의 가격이 하락하고 미래 정보화 사회 환경에서 그 효용성이 커짐에 따라서 전문화된 기능을 수행하는 임베디드 시스템의 수는 기존의 범용 기능을 수행하는 PC보다 많아질 것이다<sup>[1]</sup>.

임베디드 시스템들의 연결 형태는 일부분의 임베디드 시스템 집합이 네트워크 그룹을 형성하고 이러한 그룹들이 모여서 상위 네트워크 그룹을 구성하는 계층적 구조를 갖게된다.

이때 하위 네트워크 그룹들의 지리적 위치 및 관리자에 따라 그룹단위 노드 기능은 물론 각 네트워크 그룹내에서 사용하는 상호연결 기술은 현실적으로 정적인 구조의 통신방식을 가진다.

이러한 정적인 구조의 통신방식을 개선하기 위하여 지리적 혹은 기능적으로 분산된 수천 혹은 수만 개의 임베디드 시스템들을 이용하고 있으며, 독자적인 동작을 수행하는 임베디드 시스템들은 임베디드 네트워크를 수단으로 상호협력 능력을 가짐으로써 더욱 강력하고 유용한 시스템으로 발전할 것으로 보인다.

## 2.2 임베디드 리눅스 커널의 개요

### 2.2.1 내장형 운영체제

내장형 운영체제로는 팜 과일릿에 기반한 팜 운영체제(Palm OS), 마이크로소프트사의 윈도우즈 CE(Windows CE), 각종 실시간 운영체제, 오랜 기술축적을 통해 상당부분 시장을 점유하고 있는 Windriver System사의 VxWorks 등의 운영체제가 있다.

<표 2-1> 임베디드 운영체제

<Table 2-1> Embedded OS

제품명	회사	특징	개발환경
VxWorks	Windriver	· 확장 가능한 마이크로커널 구조 · 다중프로세서 지원	PC, SUN, HP
pSOS	Intergrated Systems	· 확장 가능한 마이크로커널 구조 · 다중프로세서 지원 · 다중 개체 대기 지원	PC, SUN
OS-9	Microwave	· 제한된 기능에서 고성능을 제공하는 작은 실시간 운영체제	PC, UNIX
VRTX	Microtech	· 다양한 CPU 지원 · 선점적 멀티태스킹 · 모듈화, 라이브러리 기반구조	PC, SUN
Window CE	Microsoft	· 가장 크고 가능 느린 모빌 OS · 다양한 장치 지원 · 윈도우 기능 제공	PC

하지만 이러한 상용 임베디드 OS는 다음과 같은 문제점을 가지고 있다.

- 크기가 너무 크고 커널의 재구성이 쉽지 않다.

- 임베디드 휴대용 장비가 지원하는 자원의 제약을 극복하기 어렵다.
- 다양한 시스템을 구성하기 힘들다.
- 초기 구입비와 로열티가 비싸다.
- 개발 인력이 많지 않다.

최근에는 내장형 프로세서 제작사들이 리눅스의 가격이 저렴하다는 장점 때문에 자사 프로세서를 탑재한 내장형 시스템에 리눅스를 포팅하고 있다. 또한, Linux를 사용하면 커널과 모든 시스템 프로그램의 소스 코드를 자유롭게 읽고 수정할 수 있기 때문에, 서버급·중소형 개인용 컴퓨터뿐만 아니라, 내장형 운영체제로도 각광을 받을 것으로 예상된다<sup>[2]</sup>.

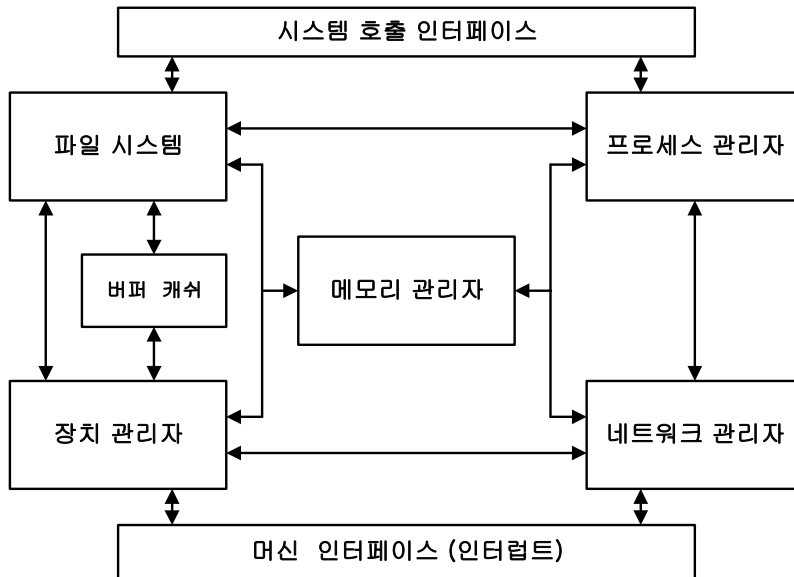
## 2.2.2 임베디드 리눅스 커널

### (1) 리눅스 커널의 구조

리눅스 커널은 크게 프로세서 관리자, 메모리 관리자, 파일시스템, 네트워크 관리자, 그리고 디바이스 드라이버의 장치관리자 이렇게 5가지 부분으로 구성된다. 커널은 자원 관리자이며, 커널이 관리하는 자원에는 물리적인 자원과 추상적인 자원으로 나뉘어 진다<sup>[3]</sup>.

프로세서 관리자는 프로세서의 생성, 실행, 상태 전이(state transition), 스케줄링, 시그널 처리, 프로세스간 통신(IPC : Inter Process Communication) 등의 서비스를 제공한다. 메모리 관리자는 가상 메모리, 주소 변환(address translation), 페이지 부재 결함 처리 등의 서비스를 제공한다. 파일시스템은 파일의 생성, 접근 제어, 인덱스 노드(Inode) 관리, 디렉토리 관리, 슈퍼 블록 관리, 버퍼 캐쉬 관리 등의 서비스를 제공한다. 네트워크 관리자는 소켓(socket) 인터페이스, TCP/IP 같은 통신 프로토콜

등의 서비스를 제공한다. 장치 관리자는 디스크 드라이버, 터미널, CDROM, 네트워크 카드 등과 같은 주변 장치를 구동하는 드라이버들로 구성된다.



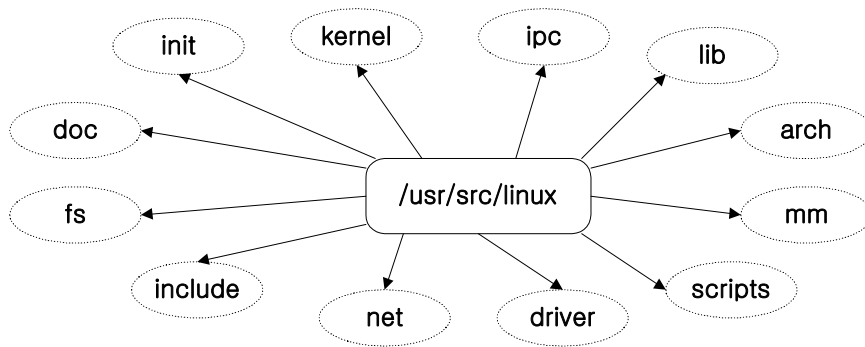
<그림 2-1> 리눅스 커널의 내부 구조

<Fig. 2-1> Internal structure of Linux kernel

리눅스 커널의 내부 구조는 <그림 2-1>에서 보는 바와 같으며, 응용 레벨과 하드웨어 레벨 사이에 위치하게 된다. 그리하여 커널은 기계 인터페이스 및 인터럽트 처리 알고리즘 등을 사용하여 하드웨어와 통신하며, 시스템 호출 인터페이스를 이용하여 응용 레벨과 통신하게 된다.

(2) 리눅스 커널의 소스 트리 구조

아래 <그림 2-2>는 리눅스 커널 소스 트리 구조를 도시화한 것이다.



<그림 2-2> 리눅스 커널의 소스 트리 구조

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

① kernel 디렉토리

프로세스 관리자가 구현된 디렉토리이다. 태스크의 생성과 소멸, 프로그램의 실행, 스케줄링, 시그널 처리 등의 기능이 이 디렉토리에 구현되어 있다.

② arch 디렉토리

리눅스 커널 기능 중 하드웨어 종속적인 부분들이 구현된 디렉토리이다. 이 디렉토리는 CPU 타입 즉, 인텔 처리기, 64비트 알파 AXP 처리기, 32비트 ARM 처리기, 모토롤라 68xxx처리기, Sun Sparc 처리기, Power PC 처리기에 따라 하위 디렉토리가 다시 구분된다.

32비트 ARM 처리기는 linux/arch/arm 하위 디렉토리에 구현되어 있다. arm/boot 디렉토리는 시스템의 초기화에 사용하는 부트스트랩 코드가 구현되어 있으며, arm/kernel 디렉토리에는 프로세스 관리자 중에서 문맥교환이나 쓰레드(thread) 관리 같은 하드웨어 종속적인 부분이 구현되어 있다.

arm/mm에는 메모리 관리자 중에서 페이지 부재 결함 처리 같은 하드웨어 종속적인 부분이 구현되어 있으며, arm/lib에는 커널이 사용하는 라이브러리 함수가 구현되어 있다.

이외에 ARM 코어를 가지고 있는 프로세서를 위한 하드웨어 처리를 위해 하위 디렉토리를 구성하고 있다.

### ③ fs 디렉토리

리눅스에서 지원하는 다양한 파일 시스템과 시스템 호출이 구현된 디렉토리이다. 각 파일 시스템은 하위 디렉토리에 구현되어 있는데, 대표적인 파일 시스템으로는 ext2, nfs, ufs, msdos, ntfs, proc, coda 등이 있다. 그리고, 다양한 파일시스템을 사용자가 일관된 인터페이스로 접근할 수 있도록 하기 위하여 리눅스는 시스템 호출과 각 파일시스템 사이에 추상화된 개념, 즉 가상파일시스템(VFS : Virtual File System)이 구현되어 있다.

### ④ mm 디렉토리

메모리 관리자가 구현된 디렉토리이다. 가상메모리, 태스크마다 할당되는 메모리 객체 관리, 커널 메모리 할당자 등의 기능이 구현되어 있다.

### ⑤ driver 디렉토리

디스크, 터미널, 네트워크 카드 등 주변 장치를 추상화시키고 관리하는 커널 구성요소인 디바이스 드라이버가 구현된 디렉토리이다.

리눅스에서 디바이스 드라이버는 크게 블록(block) 디바이스 드라이버, 문자(character) 디바이스 드라이버, 네트워크 디바이스 드라이버로 구분된다.



#### ⑥ net 디렉토리

리눅스에서 지원하는 통신 프로토콜이 구현된 디렉토리이다. 현재 리눅스에는 대표적인 통신 프로토콜인 TCP/IP 뿐만 아니라 유닉스 도메인 통신 프로토콜, X.25, IEEE 802 통신 프로토콜, IPX, AppleTalk 등이 구현되어 있다. 사용자 인터페이스를 제공하는 소켓은 하위 디렉토리에 구현되어 있다.

#### ⑦ ipc 디렉토리

리눅스 커널이 지원하는 프로세스간 통신기능이 구현된 디렉토리이다. 이 디렉토리에는 메시지 패싱(message passing), 공유 메모리(shared memory), 그리고 세마포어(semaphore)가 구현되어 있다.

#### ⑧ init 디렉토리

커널의 초기화 부분, 즉, 커널의 메인 시작 함수가 구현된 디렉토리이다.

#### ⑨ include 디렉토리

리눅스 커널이 사용하는 헤더 파일이 구현된 디렉토리이다. 헤더 파일 중에서 하드웨어 독립적인 부분은 하위에 /linux 디렉토리에 구현되어 있으며, 하드웨어 종속적인 부분은 처리기 이름으로 구성된 하드웨어 디렉토리에 구현되어 있다.

#### ⑩ other 디렉토리

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 컴파일할

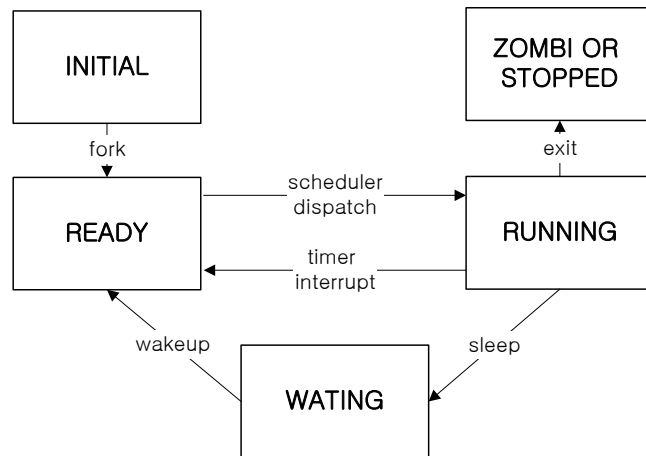
때 이용되는 스크립터들이 존재하는 /scripts 디렉토리 등이 존재한다<sup>[4]</sup>.

### (3) 프로세스 관리자

#### ① 프로세스 상태전이

프로세스(process)는 코드가 프로세서(processor)에 의해 수행되는 과정으로, 메모리의 코드와 데이터를 읽어 이를 순차적으로 수행하는데 이러한 일련의 흐름을 말한다.

<그림 2-3>은 프로세스의 상태 전이 과정을 도시화한 것이다.



<그림 2-3> 프로세스 상태 전이도

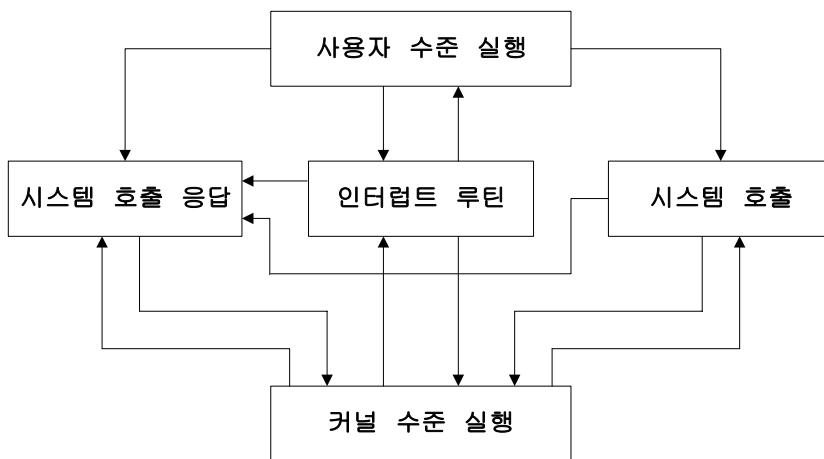
<Fig. 2-3> Process state transition diagram

사용자 프로세스는 프로그램 형태로 보조기억장치에 저장되어 있다가 시스템 호출에 통해서 운영체제의 프로세스 형태로 생성된다. 생성된 프로세스는 TASK\_RUNNING 상태를 가지며 스케줄러에 의해서 스케줄되어 동작(running) 상태가 되고 다시 타이머 인터럽트에 의해 준비(ready)

상태가 되며, 이 과정을 프로세스가 종료될 때까지 반복한다. 만약 동작 상태에서 I/O 관련 작업을 기다리다가 이벤트(event) 상태가 되면 이 때는 TASK\_INTERRUPTIBLE, TASK\_UNINTERRUPTIBLE 중 하나의 상태 값을 가진다.

커널이 다른 상위 프로세스에 의해 생성된 프로세스를 덤프(dump)하거나 중지시킬 수 있고 이때는 TASK\_ZOMBIE, TASK\_STOPPED 상태를 갖는다.

프로세스의 실행 상태를 좀더 자세히 살펴보겠다. <그림 2-4>에서 보는 바와 같이 프로세스의 실행 상태는 사용자 수준 실행(user level running)과 커널 수준 실행(kernel level running) 상태로 구분할 수 있다.



<그림 2-4> 실행 상태 구분

<Fig. 2-4> Running status division

사용자 수준 실행 상태는 프로세스가 프로그래머가 작성한 프로그램이나 라이브러리 함수를 수행하는 상태로, 사용자 수준 권한으로 동작

한다. 하지만, 커널 수준 실행 상태는 프로세스가 커널 프로그램의 일부분을 실행하는 상태로 사용자 수준 권한보다 더욱 강력한 커널 수준 권한으로 동작한다.

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 커널의 시스템 호출 루틴으로 상태가 넘어간다.

인터럽트에 의한 방법은 리눅스 커널에 인터럽트가 걸리게 되면, 이때 실행중이던 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.

## ② 스케줄링

프로세스는 항상 시스템 호출을 하며 따라서 종종 기다리게 된다. 그럼에도 불구하고 어떤 프로세스는 기다리게 될 때까지 너무 많은 CPU 시간을 사용할 수 있으며, 이러한 경우 리눅스는 선점형 스케줄링(preemptive scheduling)을 사용한다.

리눅스에서는 한 프로세스가 정해진 타임 슬라이스(slice)를 초과해서 사용하면, 그 프로세스를 중단시켜 다른 프로세스를 실행하는 선점형 스케줄링을 한다. 하지만, 리눅스의 커널 모드에서는 비선점형(non-preemptive)으로 동작한다. 이는 커널 코드가 재진입 가능하지 않게 만들어졌기 때문이다. 일단 시스템 호출이 되면 시스템 호출이 자발적으로

CPU의 할당을 내 놓지 않는 이상 시스템 호출은 다른 프로세스에 의해 멈추지 않는다.

### ③ 시그널 처리

시그널은 프로세스에게 비동기적인 사건의 발생을 알리는 방법이다.

프로세스가 시그널을 처리하려면 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.

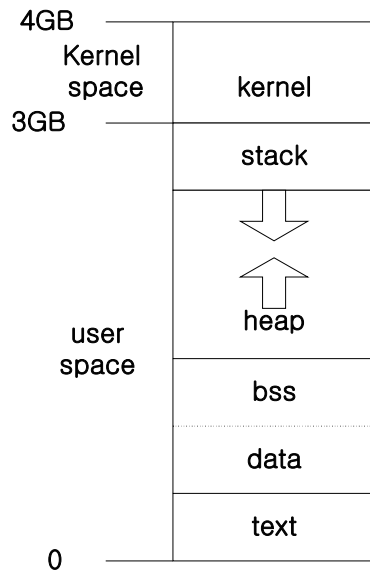
### ④ 프로세스간 통신

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

## (4) 메모리 관리자

### ① 가상메모리

초기 컴퓨터가 개발될 때부터 사용자는 시스템에 물리적으로 존재하는 것보다 더 많은 양의 메모리를 필요로 해 왔다. 물리 메모리의 한계를 극복하기 위한 여러 방법들이 개발되었지만, 그 중에서 가장 성공적이며 지금도 대부분의 시스템에서 사용하는 방법이 가상 메모리(virtual memory)이다. 가상 메모리는 실제 시스템에 존재하는 물리 메모리의 크기에 관계없이, 32비트 처리기의 경우  $2^{32}$  크기(4GB)의 가상 주소 공간을 사용자에게 제공하며, 64비트 처리기의 경우  $2^{64}$  크기의 주소 공간을 사용자에게 제공한다.



<그림 2-5> 가상 메모리 구조

<Fig. 2-5> Virtual memory structure

위의 <그림 2-5>에서 텍스트 세그먼트는 명령어의 부분으로 구성되며, 데이터 세그먼트는 전역변수들로 구성된다. 데이터 세그먼트는 다시 초기화된 데이터 부분과 초기화되지 않은 데이터 부분으로 구분하기도 한다. 이 경우 앞부분을 데이터 영역이라 부르고 뒷부분을 BSS(Block Started by Symbol) 영역이라고 부른다. 스택 세그먼트는 함수를 호출할 때 전달되는 인자들과 리턴 주소 및 함수의 지역 변수들로 구성된다.

사용자 프로그램 중에서 텍스트 세그먼트는 가장 하위 공간을 차지한다. 즉 가상 주소 0x00000000 번지부터 텍스트 세그먼트가 존재한다. 텍스트 세그먼트의 끝 다음부터는 데이터 세그먼트가 위치한다.

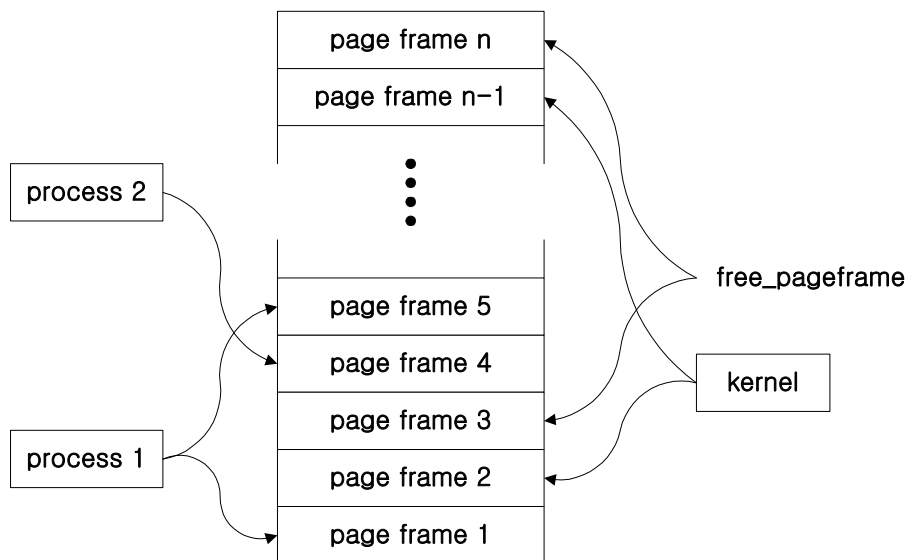
한편 스택 세그먼트는 사용자 프로그램과 커널 프로그램의 경계 위치 (3GB, 가상 주소 0xC0000000)부터 아래 방향으로 공간을 차지한다. 스

택은 프로그램이 수행됨에 따라 동적으로 변한다. 즉, 스택은 함수를 호출할 때 인자나 호출된 함수의 지역 변수를 저장하기 위하여 아래 방향으로 크기가 커진다.

메모리가 할당되는 공간을 힙(heap)이라고 하며, 힙은 데이터 세그먼트의 끝 이후 부분을 차지한다.

## ② 물리 메모리

가상메모리와 달리 물리 메모리는 시스템에 존재하는 메인 메모리의 크기에 의해 그 크기가 결정된다. 그리고 물리 메모리는 페이지 프레임(page frame)이라 불리는 고정된 크기의 기본 단위로 분할된다.



<그림 2-6> 물리 메모리

<Fig. 2-6> Physical memory

가상 메모리의 페이지와 물리 메모리의 페이지 프레임의 크기는 일반적으로 같다. 페이지 프레임의 크기는 처리기에 따라 다르다. 인텔 처리기의 경우 페이지 프레임의 크기는 4 KB이며, ARM 처리기의 경우는 16 KB이다.

#### (5) 리눅스의 파일시스템

리눅스의 가장 중요한 특징 중 하나는 많은 파일 시스템을 지원한다는 것이다. 이렇게 함으로써 리눅스는 유연성을 갖게 되었고 다른 많은 운영체제와 잘 공존할 수 있게 되었다.

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

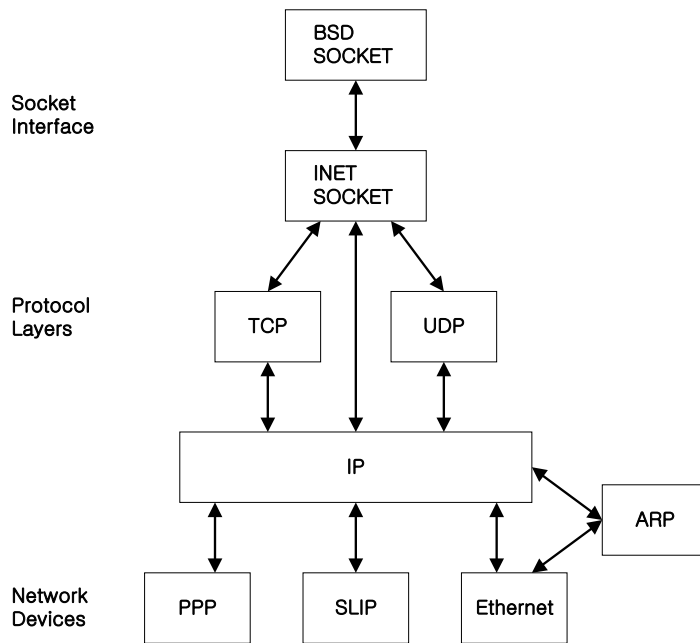
리눅스가 처음 사용했던 미닉스(Minix) 파일 시스템은 제한적이고 성능이 좋지 못했다. 파일 이름이 14자를 넘지 못했고, 파일 크기가 64 MB로 제한되었다. 리눅스 전용으로 설계되었던 첫 번째 파일 시스템은 확장 파일 시스템(extended file system)으로 1992년에 소개되었고 많은 문제점을 해결했지만 아직도 성능개선이 크게 이루어지지 못했다.

그래서, 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.



(6) 네트워크 관리자

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



<그림 2-7> 리눅스 네트워크 계층

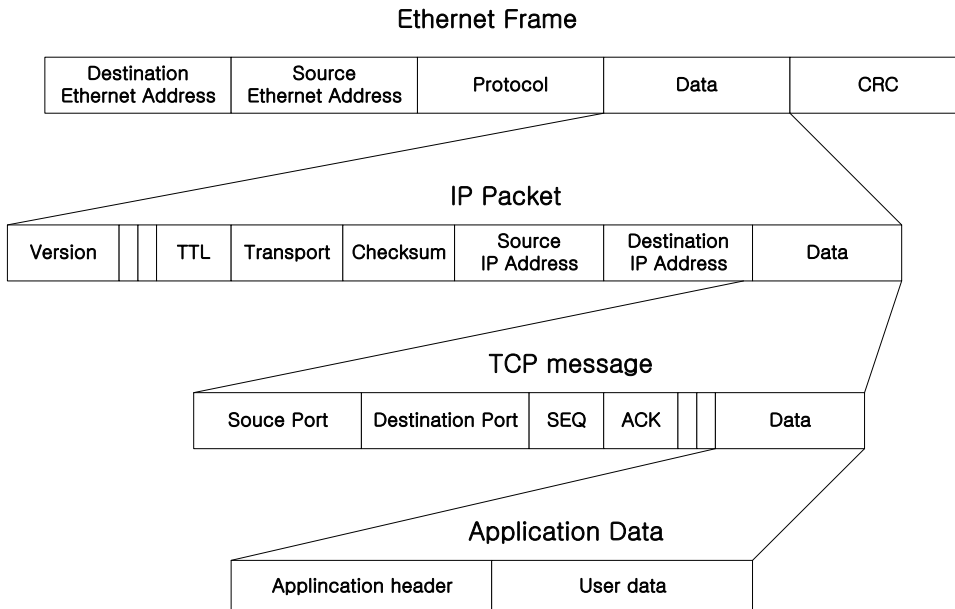
<Fig. 2-7> Linux network layer

BSD 소켓 층에서 사용자는 프로토콜 패밀리를 선택할 수 있다. 리눅스에서 지원되는 대표적인 프로토콜 패밀리는 INET(TCP/IP 지원), 유닉스, IPX, APPLETALK 등이 있다. INET 층에서 일반적으로 사용되는 유형에는 스트림(stream)과 데이터그램(datagram)이 있다.

스트림은 데이터가 전송 중 분실, 오염 또는 중복되지 않는다는 것을 보장하는 신뢰할 수 있는 양방향 순차 데이터 전송을 제공하며, 데이터그램은 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 선택하게 된다면 UDP 층으로 내려가게 된다.

그 아래에는 IP 층이 존재한다. 이 층에서는 IP 주소를 사용해 통신하며 각 호스트마다 자신의 고유한 4 Byte의 IP 주소를 가지며, 자신의 IP 주소와 목적지 IP 주소를 이용해 패킷을 만든다.



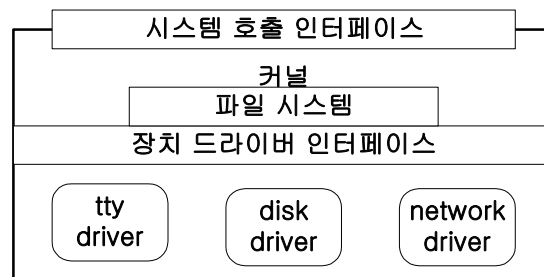
<그림 2-8> 데이터 캡슐화 과정  
<Fig. 2-8> Data encapsulation process

그리고, 에러 처리를 위한 검사합(checksum), 데이터가 너무 클 경우 단편화(fragmentation), 다른 호스트로 재전송(forwarding) 등을 수행한다. 또한, 라우팅 정보와 목적지 IP를 보고 해당 네트워크 드라이버에게 패킷을 전송하며, 전송할 때 패킷 흐름 제어 등의 추가적인 기능을 수행한다.

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

### (7) 장치관리자

Linux에는 디바이스 드라이버가 매우 간결하면서 일관된 구조로 구현되었다. 디바이스 드라이버는 디바이스와 시스템 메모리간에 데이터의 전달을 담당하는 커널 내부 기능이다. 디바이스 드라이버는 일반적으로 위쪽으로는 파일시스템과 인터페이스를 가지며 아래쪽으로는 실제 디바이스 하드웨어와 인터페이스를 갖는다.



<그림 2-9> 디바이스 드라이버

<Fig. 2-9> Device driver

위의 <그림 2-9>에서 보듯이 리눅스는 디바이스 드라이버를 크게 문자 디바이스 드라이버, 블록 디바이스 드라이버, 그리고 네트워크 디바이스 드라이버 등 세 가지로 구분한다.

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 가지의 구분은 커널의 버퍼 캐쉬를 이용하는가 여부에 따라 문자 디바이스 드라이버와 블록 디바이스 드라이버를 구분한다.

또한, 네트워크 디바이스 드라이버는 네트워크에 프레임 전송하거나 받는 드라이버이다.

새로운 디바이스 드라이버를 리눅스에 추가할 때 필요한 과정은 다음과 같이 세단계로 이루어진다.

- 디바이스 드라이버 함수를 구현한다. 디바이스 드라이버는 파일시스템과 인터페이스, 디바이스와 인터페이스, 드라이버 초기화 인터페이스 등 잘 정의된 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 설명하겠다.

#### ① 문자 디바이스 드라이버의 구조

대표적인 문자 디바이스 드라이버에는 터미널 디바이스 드라이버가 있다. 터미널 디바이스 드라이버는 크게 파일시스템과 인터페이스를 갖

는 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 이용하는데, 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 해주는 캐싱기능, 지연 쓰기, 미리 가져오기 등의 부가적인 기능도 제공한다.

## ③ 네트워크 디바이스 드라이버

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

네트워크 드라이버는 문자 디바이스 드라이버나 블록 디바이스 드라이버와 비교해 볼 때 차이점이 많다. 첫째, 문자나 블록 디바이스 드라이버는 파일시스템과 인터페이스를 갖는 데 비해 네트워크 디바이스 드라이버는 통신 프로토콜 스택(일반적으로 IP 층)과 인터페이스를 갖는다. 따라서, read(), write()등에 대한 인터페이스는 없으며, 그 대신 패킷을 네트워크로 전송하는 인터페이스와 네트워크에서 받은 패킷을 프로토콜 스택 층에 전달하는 인터페이스 등의 인터페이스가 존재한다. 둘째, 네트워크 드라이버에는 파일 연산 자료구조가 없는 대신 커널 자료구조가 역할을 대신하다<sup>[5]</sup>.

## 2.3 임베디드 리눅스 부팅 코드 분석 및 부팅과정

### 2.3.1 임베디드 리눅스 부팅 코드 분석

일반적으로 부트로더(boot loader)라 하면 x86계열 리눅스에서 LILO (Linux Loader)를 많이 사용한다. LILO란 리눅스 로더로써 도스나 Windows NT, 리눅스 등 다른 OS를 선택적으로 부팅할 수 있도록 하는 기능을 제공한다. LILO를 하드디스크의 MBR(Master Boot Record)에서 동작이 되는 프로그램으로 OS가 실행할 수 있도록 점프하는 기능을 수행한다.

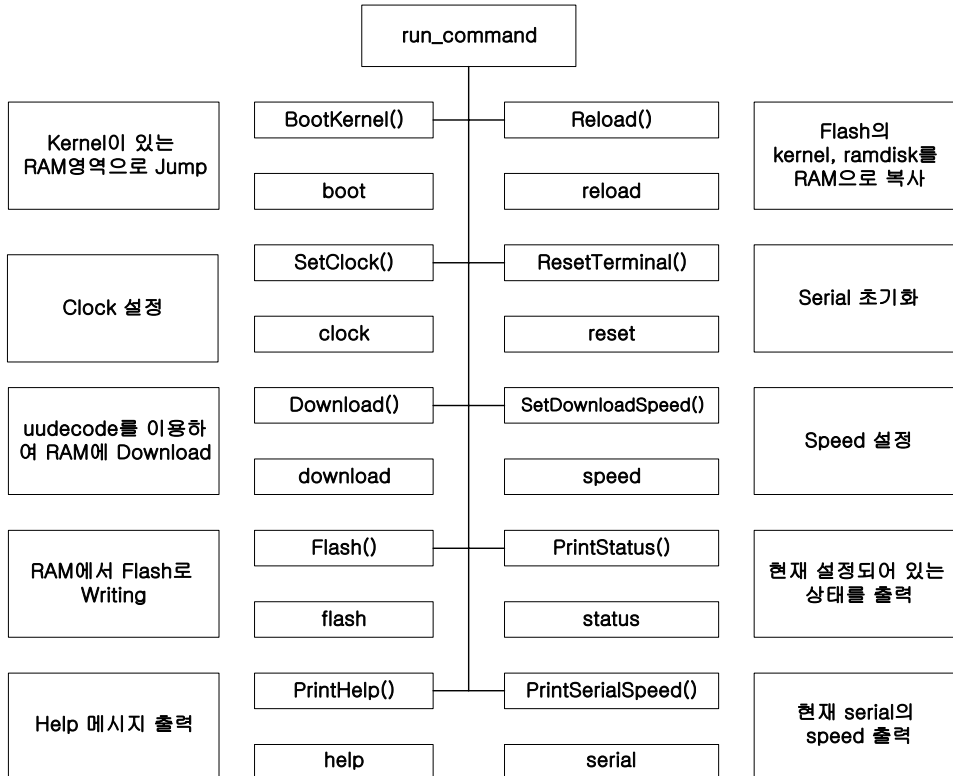
부트로더의 기능은 하드웨어의 초기화, 리눅스 부팅, 커널 또는 ramdisk 이미지를 플래시메모리에 쓰기, TFTP를 통해 SDRAM에 다운로드하는 기능을 수행한다.

<그림 2-10>에서 보는 바와 같이 main 소스에서 bootkernel() 함수는 부팅을 할 수 있게 RAM 영역으로 점프하고 setclock() 함수는 클럭을 설정하며, Download() 함수는 호스트에서 SDRAM 영역으로 다운로드 하는 기능을 수행한다. 다운로드 되는 데이터는 kernel과 ramdisk의 인자 값으로 실행된다. 그리고, flash()는 RAM에서 플래시메모리로 writing을 할 수 있고 PrintHelp()는 Help 내용을 출력해준다<sup>[6]</sup>.

### 2.3.2 임베디드 리눅스 부팅과정

앞 절에서 설명한 바와 같이 임베디드 리눅스는 소형화 시스템에 포팅되어 사용되어진다. 그리하여 호스트 컴퓨터에 연결하여 타겟 보드로써 개발이 이루어져야 하며, 타겟 보드의 리눅스 부팅은 시스템 프롬프

트를 얻기 위해서 리셋 후에 몇 번의 단계들을 통해 진행된다.



<그림 2-10> main 소스의 구성도

<Fig. 2-10> Configuration of main source

ROM 시작 코드와 레지스터 구성 같은 초기 단계들은 마이크로프로세서 하드웨어에 의존한다. 커널 그 자체는 초기화 코드가 처음 실행되어진 마이크로프로세서 구조를 포함한다. 이 초기화 코드는 보호 모드 작동을 위한 마이크로프로세서 레지스터들을 구성하고 그 다음에 start\_kernel이라고 불리는 아키텍처에 독립적인 커널 시작지점을 호출한다.

커널 부트 과정은 모든 구조들에 동등하면, 리눅스 부팅은 다음과 같은 단계를 포함한다<sup>[7]</sup>.

- 프로세스 리셋 후에 ROM 시작 코드를 실행한다.
- ROM 시작 코드는 CPU, 메모리 제어기, 그리고 디바이스를 초기화하고 메모리 맵을 구성한다. ROM 시작 코드 그 다음에는 부트 로더를 실행한다.
- 부트 로더는 플래시메모리 또는 TFTP 서버 전송으로부터 RAM 내의 리눅스 커널에 압축을 해제한다. 그것은 그 다음에 커널의 첫 번째 명령으로 건너뛴을 실행한다. 커널은 처음으로 마이크로프로세서 레지스터들을 구성한 후, 아키텍처에 독립적인 시작점인 `start_kernel`을 시작한다.
- 커널은 캐쉬와 여러 디바이스 드라이버를 초기화한다.
- 커널은 루트 파일시스템을 마운트한다.
- 커널은 `init` 프로세스를 실행한다.
- `init` 프로세스 실행은 공유한 런타임 라이브러리들을 적재하고 사용자 프로세스를 실행한다.



## 제 3 장 원격 제어 시스템의 설계 및 구현

각 단위 임베디드 시스템들은 주위에서 발생하는 사건들을 감지하는 등의 특정 작업을 수행하고 있으며, 통신망의 발달로 인터넷을 통한 임베디드 네트워크 환경에서 원격 제어 및 원격 계측의 기능을 수행할 수 있게 되었다.

본 논문에서는 이러한 작업을 수행하는 최적화된 하드웨어를 구성하고, TCP/IP 프로토콜을 이용하여 각 노드 단위로 유일한 이름을 제공하는 고정된 바인딩을 이용하는 네이밍 기법이 아닌 노드의 네트워크 위상이나 노드의 가용성이 동적으로 변화하는 시스템 상황에서도 사용할 수 있는 동적 할당을 이용하고자 한다.

### 3.1 시스템의 개요

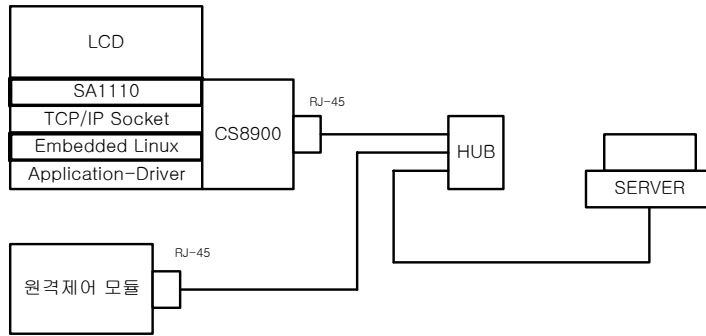
본 절에서는 2장에서 분석·정리된 임베디드 리눅스를 이용하여 원격 제어시스템을 설계하고자 한다. 개발하고자 하는 시스템은 INTEL사의 SA-1110을 MCU로 하여 패킷을 처리하도록 하였으며, 물리 계층에는 CS8900을 사용하였고, 외부 네트워크와의 연결을 위하여 허브를 사용하였다.

원격시스템의 기본 구성도는 <그림 3-1>과 같다.

#### 3.1.1 시스템의 부팅과정

시스템에 전원이 인가되었을 경우 제일먼저 실행되는 것이 부트로더 부분중의 start.S라는 파일이다. 이 파일을 통해 하드웨어에 대한 초기

화와 각종 설정이 이루어지는데, 먼저 인터럽트를 비활성화시키고, CPU 클럭을 설정한 다음 메모리와 시리얼, 스택 포인터를 설정한다. 그 후 메인 루틴으로 점프를 하게 된다.



<그림 3-1> 시스템 개요

<Fig. 3-1> Introduction of System

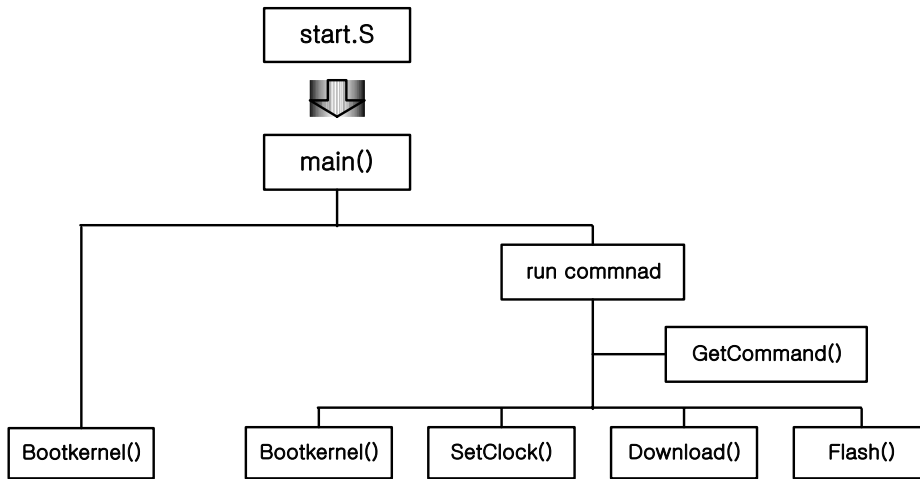
부트로더의 실질적인 시작은 SA1110 타겟 보드의 구동시 0번지로 점프하고, 0번지에는 벡터 테이블이 존재한다. 하드웨어 리셋시 이 벡터 테이블의 setup이 가리키는 주소를 찾아서 실행되는데, 이 벡터 테이블을 start.S에서 만들어준다.

부트로더는 실질적으로 스타트업(start-up) 루틴으로 수행되는데, 스타트업 루틴의 역할은 다음과 같다.

먼저 벡터 테이블을 생성하고, 모든 인터럽트를 막는다. 그리고 CPU의 클럭을 지정하고, SDRAM 및 플래시메모리 영역을 설정한 후 C 코드로 점프한 후 커널을 로딩하게 된다.

<그림 3-2>에서 시스템 부팅과정에 대해 자세히 설명하고 있다.

먼저 start.S를 통해 각종 하드웨어 초기화가 이루어지고 start.S에서 main() 함수로 점프한다. 이 함수는 시리얼과 타이머를 초기화하고 커널과 ramdisk를 SDRAM으로 로딩한 다음 명령어를 기다린다.



<그림 3-2> 시스템 부팅과정

<Fig. 3-2> System booting process

아무런 명령이 없을 경우 bootkernel() 함수를 통해 바로 커널이 수행되고, 명령이 있을 경우 부트로더 명령을 할 수 있도록 프롬프트 상태가 된다.

프롬프트 상태에서 명령이 들어오면 GetCommand() 함수를 통해 명령을 분석하여 해당 함수를 호출하여 명령을 실행한다.

## 3.2 임베디드 시스템 포팅을 위한 환경구성

### 3.2.1 통신 에뮬레이터 환경 설정

리눅스에서는 타겟 보드의 부팅을 확인하기 위해서 미니콤이라는 리눅스용 통신 에뮬레이터 프로그램을 사용하는데, 이것은 타겟 보드와 호스트 사이의 상호 데이터 전송 및 모니터링을 할 수 있다.

타겟 보드와의 직렬 통신을 하기 위하여 아래와 같이 미니콤의 환경을 설정한다.

<표 3-1> 미니콤 환경 설정 값

<Table 3-1> Minicom environment setup value

구 성 환 경	설 정 값
Baudrate	115200bps
Data size	8
Parity	None
Stop bits	1
Hardware flow control	No
software flow control	No

아래 <그림 3-3>에서는 타겟 보드의 부트로더 및 커널 실행을 위한 미니콤의 환경을 설정하고 있는 과정이다.

```

+-----+
| A -   Serial Device       : /dev/ttyS0
| B -   Lockfile Location   :
| C -   Callin Program     :
| D -   Callout Program    :
| E -   Bps/Par/Bits       : 115200 8N1
| F -   Hardware Flow Control : No
| G -   Software Flow Control : No
|
| Change which setting? 
+-----+
| Screen and keyboard
| Save setup as df1
| Save setup as..
| Exit
| Exit from Minicom
+-----+

```

<그림 3-3> 미니콤 실행

<Fig. 3-3> Minicom Running

### 3.2.2 크로스 컴파일러 설정

일반적으로 컴파일러는 자신의 시스템에 맞는 바이너리 코드를 만드는 작업을 수행한다. 타겟 보드가 저장 할 수 있는 메모리 공간이 매우 부족하기 때문에, 타겟 보드에서 직접 응용프로그램이나 커널 컴파일을 할 수가 없다.

그 결과로 타겟용 커널 및 응용프로그램을 개발하기 위하여 호스트 시스템에 타겟용 크로스 컴파일러 환경을 구축하게 된다.

보통 리눅스에서 개발을 한다면 시그너스사의 공개 환경과 GNU의 개발 환경을 사용하게 된다. 이런 공개 프로그램은 지속적인 버그수정과 향상에 따라 버전이 올라가게 되는데, 이에 따라 이미 구축되어 있는 개발 환경이 아니라 지속적인 업데이트가 필요하다<sup>[8]</sup>.

이러한 크로스 컴파일러 구축 환경은 아래와 같다.

```

[root@localhost cross_compiler]# rpm -Uvh gcc-arm-2.95.2-12e4.i386.rpm
Preparing... ##### [100%]
 1: gcc-arm ##### [100%]
[root@localhost cross_compiler]# rpm -Uvh cpp-arm-2.95.2-12e4.i386.rpm
Preparing... ##### [100%]
 1: cpp-arm ##### [100%]
[root@localhost cross_compiler]# rpm -Uvh g++-arm-2.95.2-12e4.i386.rpm
Preparing... ##### [100%]
 1: g++-arm ##### [100%]
[root@localhost cross_compiler]#

```

<그림 3-4> 크로스 컴파일러 환경 설정

<Fig. 3-4> Cross-Compiler environment setup

### 3.2.3 커널 설치 및 컴파일

ARM 리눅스 커널은 I386용(PC용) 리눅스와 전혀 다르게 설계되고 코딩되어 있는 것은 아니며, 리눅스 커널 자체가 상당히 이식하기 좋은 구조로 되어 있기 때문에 이를 버리고 다른 커널을 디자인 할 필요는 없다.

ARM용 리눅스 커널을 구하여야 하는 이유는 아래와 같다.

- C 프로그램 작성을 위해서 ARM에 관련된 헤더 파일을 작성해야 하는데 어려운 점이 많다.
- 부트로더는 PC용 리눅스 커널의 헤더 파일을 참조한다.

그리고, 리눅스 커널을 구성하는 방법은 다음 순서에 따라야 한다.

- ARM용 리눅스 커널을 구성한다.
- ARM 패치를 수행한다.
- StrongArm용 패치를 수행한다.
- 타겟 보드를 위한 패치를 작성하여 수행한다.

아래 <그림 3-5>는 커널의 설치 과정이다.

```

linux/Documentation/SubmittingDrivers
linux/Documentation/dnotify.txt
linux/Documentation/mkdev.cciss
linux/Documentation/SubmittingPatches
linux/Documentation/parisc/
linux/Documentation/parisc/00-INDEX
linux/Documentation/parisc/IODC.txt
linux/Documentation/parisc/debugging
linux/Documentation/parisc/registers
linux/Documentation/cris/
linux/Documentation/cris/README
linux/Documentation/SAK.txt
linux/Documentation/i810_rng.txt
linux/Documentation/mips/
linux/Documentation/mips/GT64120.README
linux/Documentation/mips/time.README
linux/Documentation/mips/pci/
linux/Documentation/mips/pci/pci.README
linux/Documentation/power/
linux/Documentation/power/pci.txt
linux/Documentation/README.nsp_cs.eng
linux/REPORTING-BUGS
[root@telecom kernel18]# 
[영어] [완성] [두벌식]

```

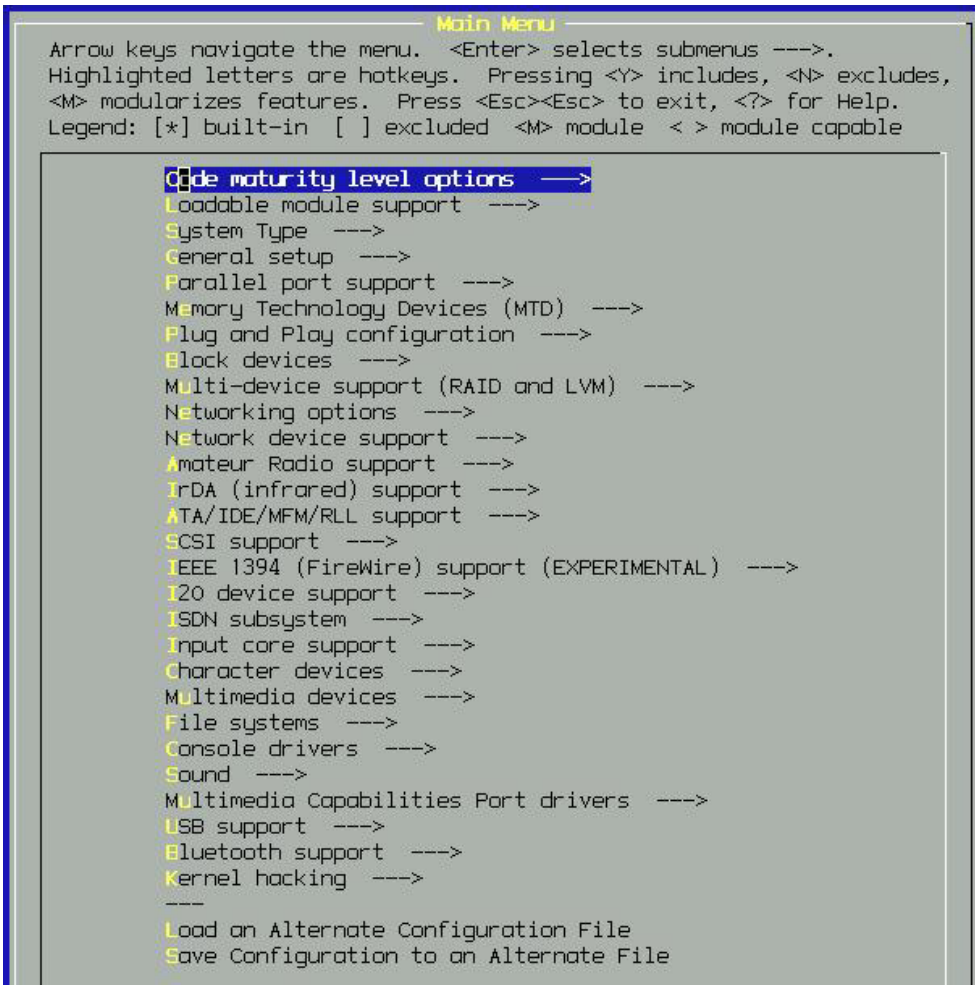
<그림 3-5> 커널 설치 과정

<Fig. 3-5> Kernel setup process

타겟 보드에 맞게 커널을 수정하였다면, 컴파일 옵션이나 환경 설정에 관련한 사항을 수정해 주어야한다. 이 과정은 아래 <그림 3-6>에서 보는 바와 같이 menuconfig 환경에서 이루어진다.

### 3.2.4 다운로드 과정

타겟 보드는 IP 주소가 고정되어 있지 않으므로 이를 수동으로 할당하거나 IP 주소를 자동으로 할당받아야 하는데, IP 주소를 할당받는 것에는 BOOTP(Bootstrap Protocol)와 DHCP(Dynamic Host Configuration Protocol)가 있다. 이중 BOOTP는 매우 간단한 프로토콜 구현으로 수행될 수 있으므로 부트로더와 같은 프로그램은 이것을 사용한다.



<그림 3-6> Menuconfig 구성

<Fig. 3-6> Menuconfig configuration

또한, 리눅스를 다운로드 하는 방식에는 시리얼을 이용하는 방식과 LAN을 이용하는 방식이 있다. 다운 받아야 할 내용이 매우 큰 파일을 전송속도가 느린 시리얼로 다운 받게 되면 개발 속도가 현저히 떨어지므로, LAN을 이용하여 파일을 타겟 보드에 다운로드 하는 방법으로



TFTP를 이용하였다.

#### (1) BOOTP

BOOTP란 이더넷의 MAC 어드레스를 이용하여 IP 주소를 얻어오는 프로토콜로 UDP 방식을 이용한다. 즉, IP 주소를 얻고자 하는 클라이언트는 자신의 이더넷 MAC 어드레스를 브로드캐스트하고, BOOTP 서비스를 하는 호스트들은 이것을 모두 받아 자신이 가지고 있는 이더넷 MAC 어드레스와 일치하는 가를 확인 후 해당 IP 주소를 전송한다.

보통 BOOTP와 TFTP를 조합하여 사용하기 때문에 호스트에서 타겟 보드로 전송되는 정보에는 IP 주소뿐만 아니라 부트 이미지의 파일명도 전송된다.

```
[root@telecom linux]# ps -ax | grep bootpd
1097 pts/1    S      0:00 grep bootpd
[root@telecom linux]#
```

<그림 3-7> BOOTP 데몬

<Fig. 3-7> BOOTP daemon

#### (2) TFTP

TFTP(Trivial File Transfer Protocol)란 이더넷을 이용하여 파일을 다운 받는 프로토콜 로직으로 UDP 방식을 사용한다. TFTP는 FTP와 같은 파일 전송 프로토콜이지만, 매우 간단한 프로토콜로 구성되므로 부트로더와 같은 작은 크기의 프로그램에서 수행할 수 있다.

즉, 클라이언트가 서버에 접속하여 파일을 요청하면 해당 파일을 서버는 전송하고 클라이언트는 응답하는 방식을 취하는데, 하나의 블록 전송이라도 실패하면 전송을 중지한다.

```
EZBOOT>tmk
Receive zImage
CS8900 Init.....
CS8900 Mac Address : [00 D0 CA F1 26 25]
CS8900 DECTECT VALUE : [00003000]
CS8900 INIT OK!!!

Send ARP Packet
ARP PACKET Resive
HOST MAC : [ 00 E0 4B 01 47 A2 ]
HOST IP : [211.242.5.217]
LOCAL IP : [211.242.5.218]
Resive Address : C100-0000
TFTP Request Send
ALL DATA RESIVE OK [ 626128 bytes ]

EZBOOT>
```

<그림 3-8> TFTP 전송 환경

<Fig. 3-8> TFTP transmission environment

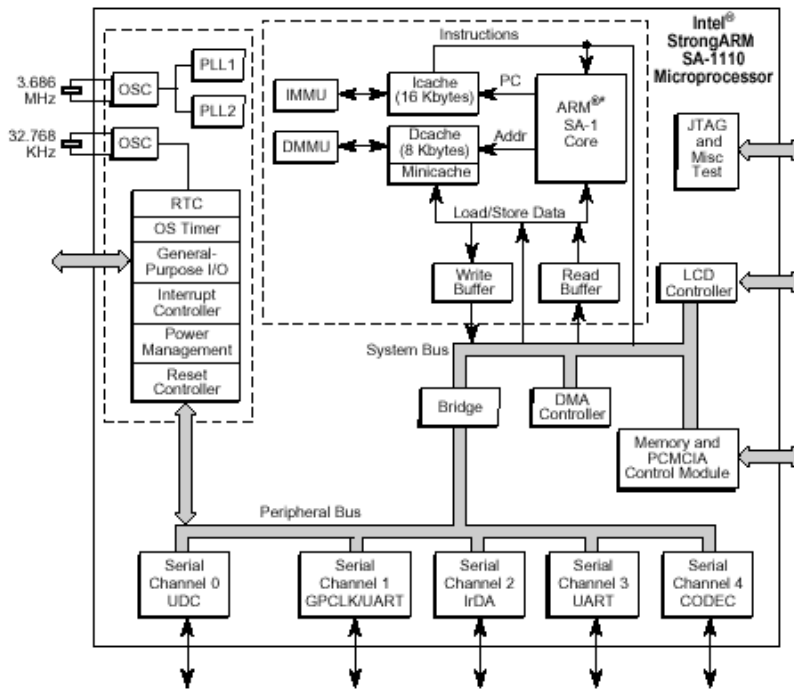
### 3.3 SA-1110의 분석 및 회로의 구성

#### 3.3.1 SA-1110의 분석

##### (1) INTEL SA-1110의 특징

StrongARM은 INTEL사에서 ARM코어를 사용하여 ASIC을 구현한 마이크로컨트롤러로서 SA-110, SA-1100, SA-1110의 종류가 있다.

다음 <그림 3-9>는 SA-1110의 내부 다이어그램이다.



<그림 3-9> SA-1110의 내부 다이어그램

<Fig. 3-9> Internal Diagram of SA-1110

SA-1110은 32비트 마이크로컨트롤러로서 고성능으로 평가받고 있으며, 내부에 ARM 코어를 내장하고 있을 뿐만 아니라, LCD 제어모듈과 메모리와 PCMCIA 제어모듈 및 클럭 발생기를 내장하고 있어 시스템의 소형화 및 저전력화 등 많은 장점을 가지고 있다.

연구개발 과정에 있어서 C 언어뿐만 아니라, 어셈블러와 같이 프로그램을 구현할 수 있는 점을 채택하여 연구에 활용토록 하였다.

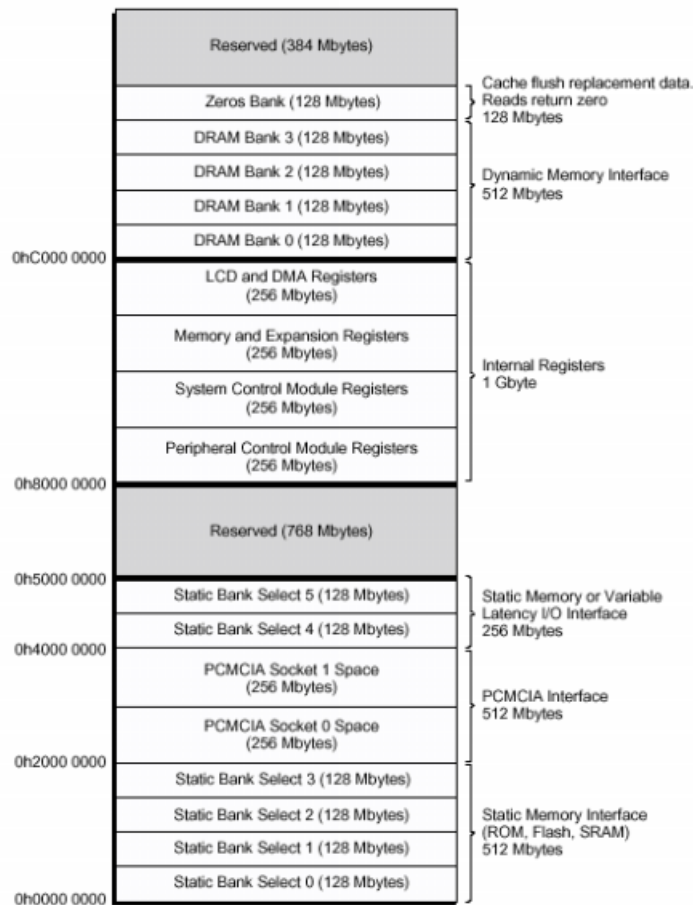
다음은 SA-1110의 특징을 설명한 것이다<sup>[9],[10]</sup>.

- 32-bit RISC processor Core
- Low Power 400mW
- 256 mini-ball grid array(mBGA Type)
- 3.3V I/O Interface
- Integrated Clock generation
- Power-management features
- Big and little endian operating modes
- 32-entry MMUs
- 32-bit command and data cache
- Write and read buffer

## (2) INTEL SA-1110의 메모리 맵 배치

SA-1110의 메모리 공간은 크게 정적 메모리 영역과 PCMCIA 메모리 영역, 내부 레지스터 영역, 다이내믹 메모리 영역, 캐쉬 영역으로 나뉜다. <그림 3-10>은 SA-1110의 메모리 맵을 나타내고 있다.

- 정적 메모리 영역 : ROM, Flash, SRAM 같은 지연 처리가 필요 없는 영역과 지연 처리가 필요한 I/O 영역으로 나뉜다. 이 영역은 0x00000000H~0x20000000H, 0x40000000~0x50000000으로 두



<그림 3-10> SA-1110 메모리 맵  
 <Fig. 3-10> SA-1110 memory map

개의 분리된 영역으로 나누어진다. 0x00000000부터 0x80000000까지는 부팅 처리 소프트웨어가 탑재되는 롬이나 부트 롬 영역에 할당하게 된다. 각 영역의 크기는 nCS 신호선을 이용하여 처리한다면 128Mbyte 단위가 된다. StrongARM에 설계되는 외부 인터페이스는 이 공간에 할당하여야 한다. 특히 느린 디바이스라면

0x18000000 (nCS3), 0x40000000(nCS4), 0x48000000(nCS5) 영역에 설계하여야 한다.

- 내부 레지스터 영역 : StrongARM의 내부에 탑재된 디바이스는 0x80000000~0xC0000000 영역에 할당되어 있다.
- 다이내믹 메모리 영역(DRAM 영역) : 임베디드 시스템에 사용하는 RAM영역으로 0xC0000000~0xE0000000에 할당되어 있다. 이 공간은 다시 4개로 나누어지는데 각각 128 MByte 공간을 갖는다.

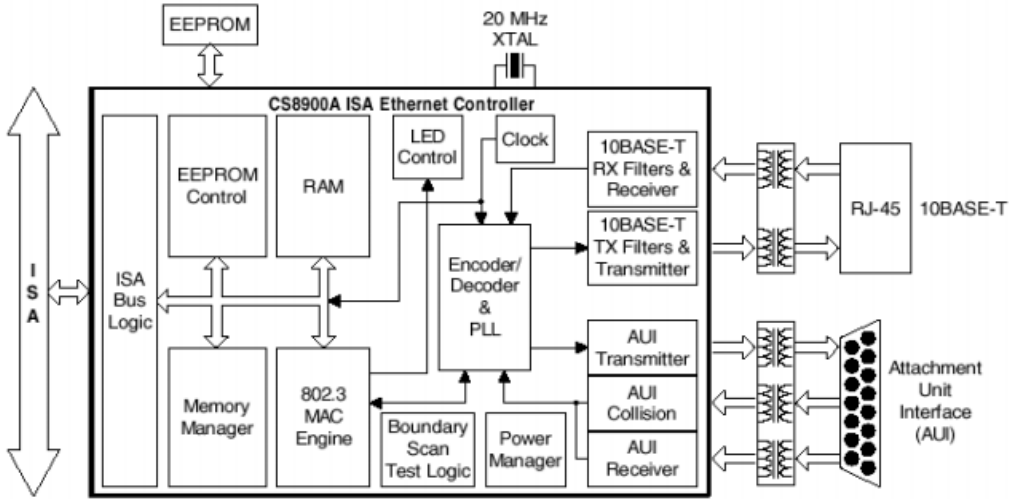
### 3.3.2 CS8900A의 분석

CS8900A는 Cirrus Logic사의 이더넷 컨트롤러이다. CS8900A는 IEEE 802.3 이더넷 표준을 완전하게 적용하고 있으며, 이더넷 프레임 송·수신의 모든 측면을 핸들링할 수 있다. 내부에 모든 아날로그 신호와 인터페이스에 필요한 디지털 회로를 포함하고 있어 아날로그 전송 신호를 디지털적으로 제어가 가능해 설계를 용이하게 할 수 있는 장점을 가진다.

다음은 CS8900A의 특징을 설명한 것이다<sup>[11]</sup>.

- Single-Chip IEEE802.3 Ethernet Controller with Direct ISA-Bus Interface
- 55mA Maximum Current Consumption
- 3V Operation
- On-Chip RAM Buffers Transmit and Receive Frames
- Programmable Transmit and Receive
- Boot PROM Support Diskless Systems
- LED Drives for Link Status and LAN Activity

- Standby and Suspend Sleep Modes



<그림 3-11> CS8900A의 내부 다이어그램

<Fig. 3-11> Internal diagram of CS8900A

CS8900A의 패킷 전송 과정은 두 가지 단계로 나눌 수 있다. 첫째는 호스트가 이더넷 프레임을 CS8900A의 버퍼로 옮기는 과정이고, 둘째는 CS8900A가 이더넷 프레임을 이더넷 패킷으로 변환하고, 네트워크에 전송하는 과정이다.

좀더 자세히 살펴보면 첫 번째 단계는 CS8900A에 프레임이 전송될 것이라는 전송명령을 선언하고, 전송길이를 전송한 후, 요구한 버퍼 공간이 이용가능한지를 검사한다. 그리고 버퍼공간이 이용 가능하면 이더넷 프레임을 전송한다.

두 번째 단계는 프리앰블(Preamble)과 구분문자의 시작을 전송하고, DA, SA, 길이 부분과 DLC(Data Link Control) 데이터, 그리고 CRC (Cyclic Redundancy Checking)를 전송한다.

### 3.3.3 회로의 구성

<그림 3-12>는 평가회로의 블록 다이어그램을 나타낸 것이다. 패킷의 처리를 위하여 StrongARM SA-1110을 사용하였으며, 시스템의 메모리 맵은 <그림 3-13>에서 설명되고 있다.

0xC0000000부터 0xE0000000에 할당된 RAM 영역에 작성한 커널과 ramdisk 이미지를 다운로드하였다.

그리고, 서버에서 클라이언트로 제어를 위하여 데이터를 전송하고, 클라이언트 시스템의 출력장치에 초기화면 메시지 및 전송된 명령의 수행 과정 및 결과를 출력하였다.

물리계층의 CS8900A는 물리적인 주소 0x10000000에 매핑되어 있는 nCS2에 연결하였다. A[0:25]는 0x000를 기본주소로 사용하고 있으므로 BUS로 나가는 물리적인 주소는 0x10000000이다.

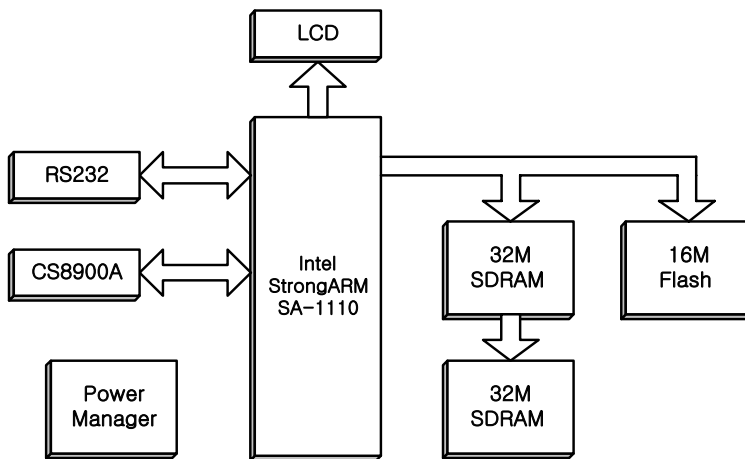
LCD 패널은 물리적인 주소 0x08000000에 매핑되어 있는 nCS1에 연결하였다.

<표 3-2> 타겟 보드 메모리 맵

<Table 3-2> Target board memory map

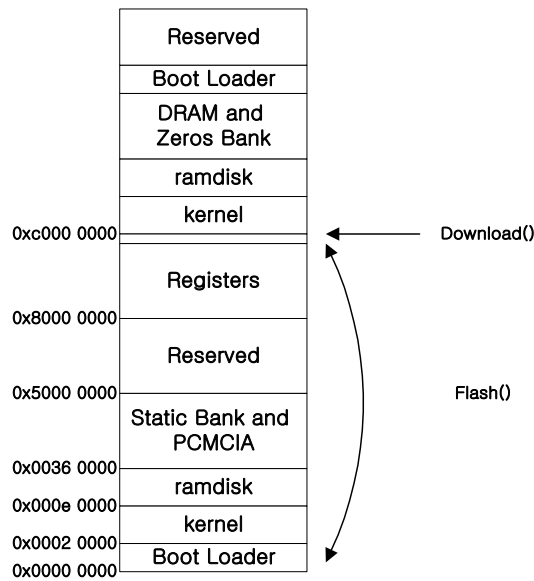
PART	MEMORY AREA	CONTROL PORT	
kernel image	0x00020000	nSDCS0	Flash
ramdisk image	0x000E0000	nSDCS0	
kernel image	0xC0005000	nCS0	Memory
ramdisk image	0xC0100000	nCS0	
CS8900A	0xDC000000	nCS2	
LCD 패널	0xD8000000	nCS1	





<그림 3-12> 평가회로 블록도

<Fig. 3-12> Evaluation circuit block diagram



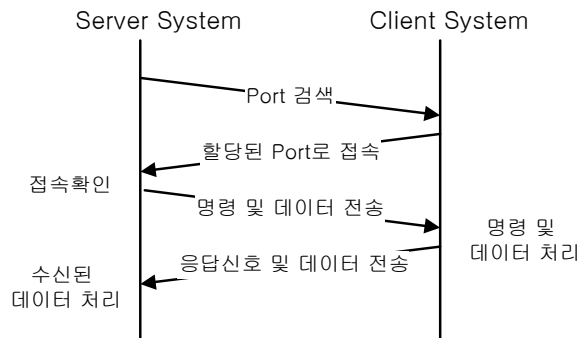
<그림 3-13> 평가회로의 SA-1110 메모리 맵

<Fig. 3-13> SA-1110 memory map of evaluation board

### 3.4 시험회로의 평가

클라이언트 프로그램은 통신을 시작하고, 서버 프로그램은 할당된 포트를 검색하며, 클라이언트의 접속을 기다린다. 클라이언트의 접속을 확인한 후 클라이언트에 명령 및 데이터를 전송하고, 그에 따른 응답신호 및 데이터를 전송 받는다.

그래서, 클라이언트는 서버의 주소와 포트번호를 초기에 알고 있어야 하는데 반해, 서버는 클라이언트의 정보를 알 필요가 없는 점이 서버-클라이언트 시스템에서 중요하다.

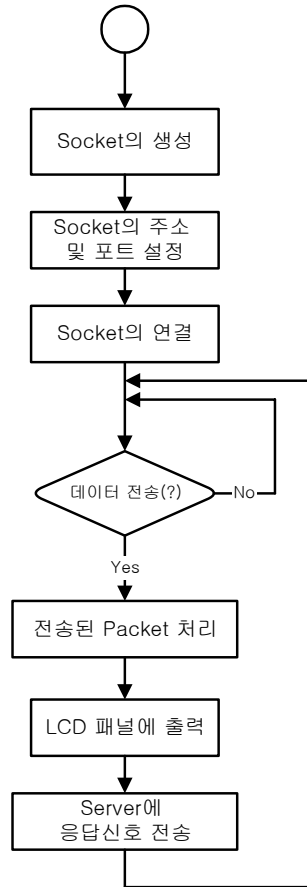


<그림 3-14> 시스템 전송 절차

<Fig. 3-14> System transmission procedure

그리고, 내장형 시스템은 운영체제와 응용 프로그램 및 기타 파일들이 ROM에 저장되고, ROM에 저장된 파일 시스템을 RAM에 로딩하여 사용한다. 그리고, 시리얼 장치를 연결하여 사용하고, 출력 장치를 사용하지 않지만, LCD 패널을 출력장치로 활용하기도 한다.

본 논문에서는 모듈에 LCD 패널을 출력 장치로 연결하여 사용하였다. 아래 <그림 3-15>는 평가회로에서 사용한 시스템 알고리즘이다.



<그림 3-15> 평가회로 동작 흐름도

<Fig. 3-15> Flow chart of evaluation circuit

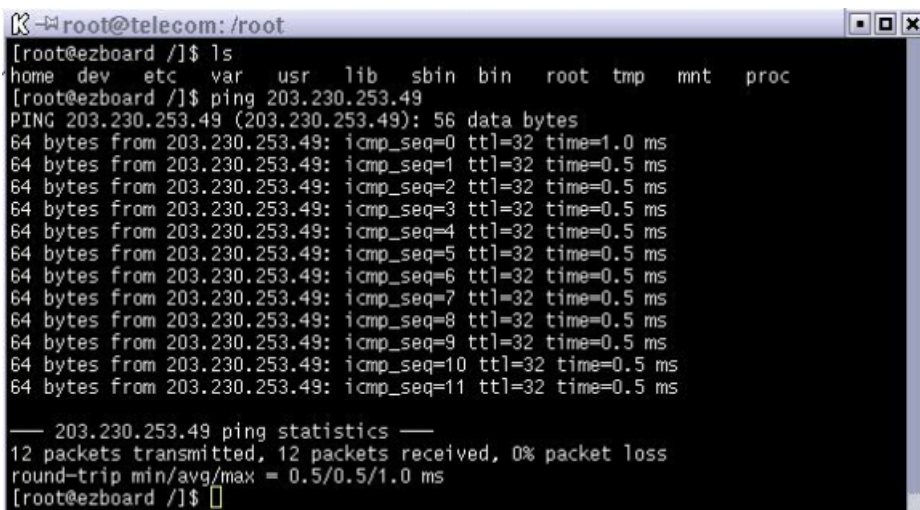
### 3.4.1 소켓 프로그래밍

소켓은 어플리케이션이 데이터를 보내고 받을 수 있는 추상적인 개념이다.

TCP/IP 프로토콜 패밀리에서 소켓은 스트림소켓과 데이터그램소켓으로 구분할 수 있는데, 본 논문에서는 스트림 소켓만을 다루겠다.

스트림 소켓은 IP를 하위에 두고 TCP 프로토콜을 사용하는 신뢰성 있는 바이트-스트림 서비스를 제공하며, 포트 번호와 결합하여 원격 어플리케이션으로부터 메시지를 전송 받는다<sup>[12],[13]</sup>.

<그림 3-16>은 서버 시스템과의 네트워크 연결을 확인하고 있다.



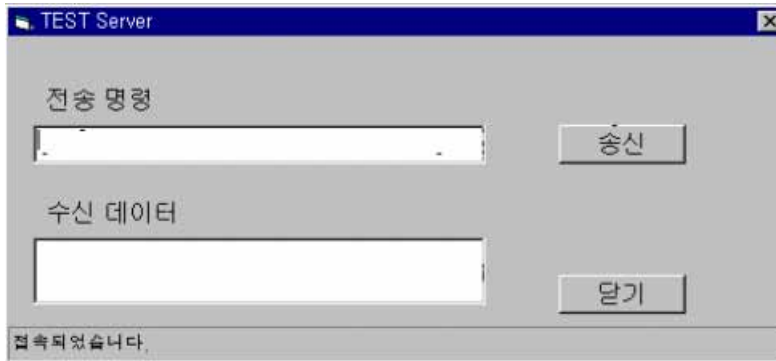
```
[root@ezboard /]$ ls
home dev etc var usr lib sbin bin root tmp mnt proc
[root@ezboard /]$ ping 203.230.253.49
PING 203.230.253.49 (203.230.253.49): 56 data bytes
64 bytes from 203.230.253.49: icmp_seq=0 ttl=32 time=1.0 ms
64 bytes from 203.230.253.49: icmp_seq=1 ttl=32 time=0.5 ms
64 bytes from 203.230.253.49: icmp_seq=2 ttl=32 time=0.5 ms
64 bytes from 203.230.253.49: icmp_seq=3 ttl=32 time=0.5 ms
64 bytes from 203.230.253.49: icmp_seq=4 ttl=32 time=0.5 ms
64 bytes from 203.230.253.49: icmp_seq=5 ttl=32 time=0.5 ms
64 bytes from 203.230.253.49: icmp_seq=6 ttl=32 time=0.5 ms
64 bytes from 203.230.253.49: icmp_seq=7 ttl=32 time=0.5 ms
64 bytes from 203.230.253.49: icmp_seq=8 ttl=32 time=0.5 ms
64 bytes from 203.230.253.49: icmp_seq=9 ttl=32 time=0.5 ms
64 bytes from 203.230.253.49: icmp_seq=10 ttl=32 time=0.5 ms
64 bytes from 203.230.253.49: icmp_seq=11 ttl=32 time=0.5 ms

--- 203.230.253.49 ping statistics ---
12 packets transmitted, 12 packets received, 0% packet loss
round-trip min/avg/max = 0.5/0.5/1.0 ms
[root@ezboard /]$
```

<그림 3-16> Ping 테스트 결과

<Fig. 3-16> Result of ping test

아래 <그림 3-17>, <그림 3-18>은 소켓 프로그램을 작성하여 호스트와 타겟 보드와의 루프백 실험을 한 결과이다<sup>[14]</sup>.



<그림 3-17> 서버 루프백 실험 결과

<Fig. 3-17> Loopback test result of server



<그림 3-18> 클라이언트 루프백 실험 결과

<Fig. 3-18> Loopback test result of client

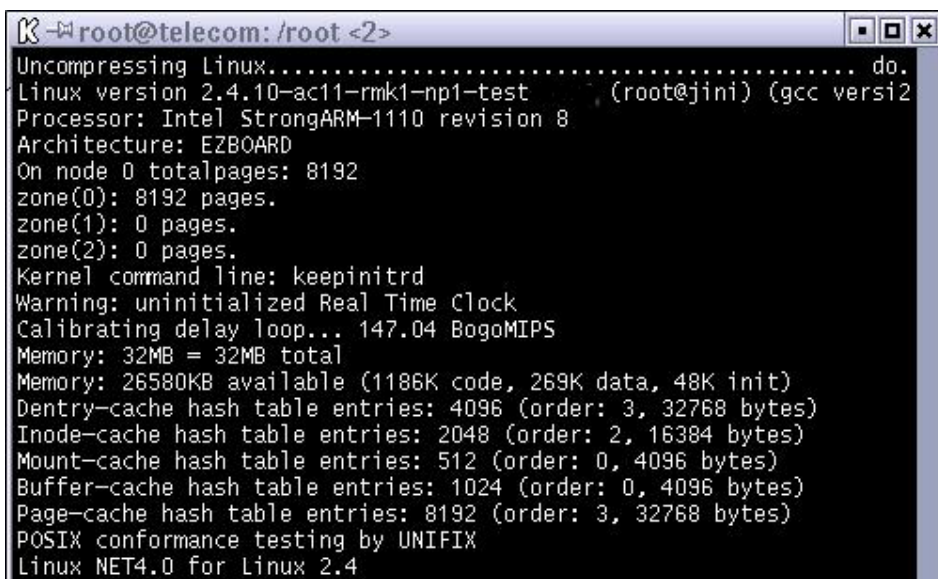
### 3.4.2 회로시험 결과

본 논문에서는 임베디드 운영체제인 임베디드 리눅스를 최적화하여 포팅하고, 응용제품을 설계하기 위한 시험 소프트웨어를 개발하는데 중

점을 두었다.

임베디드 리눅스는 타겟보드를 구성하여 포팅을 하였으며, 포팅의 결과는 다음 <그림 3-19>와 같다. 시험 소프트웨어는 앞 절에서 작성한 TCP/IP 소켓 프로그램을 바탕으로 디바이스 드라이버를 작성하였다.

전체적인 시스템은 서버 시스템과 타겟보드를 허브에 연결하여 네트워크 환경에 접속할 수 있도록 구축한 후 원격 제어를 위한 테스트를 하였다.

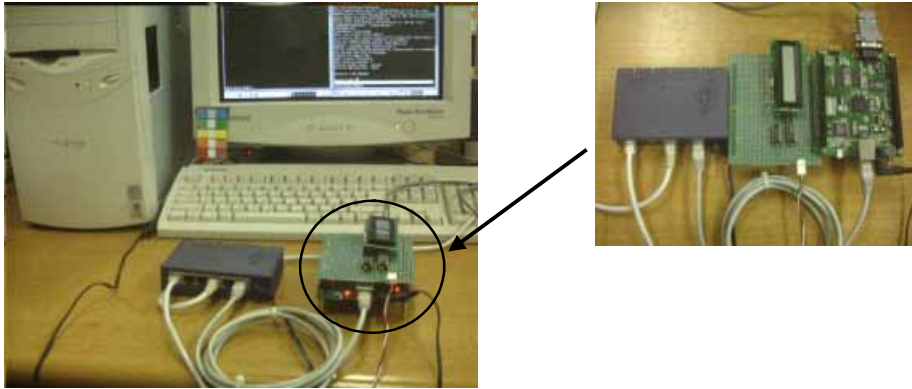


```
root@telecom: /root <2>
Uncompressing Linux..... do.
Linux version 2.4.10-ac11-rmk1-np1-test (root@jini) (gcc versi2
Processor: Intel StrongARM-1110 revision 8
Architecture: EZBOARD
On node 0 totalpages: 8192
zone(0): 8192 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: keepinitrd
Warning: uninitialized Real Time Clock
Calibrating delay loop... 147.04 BogoMIPS
Memory: 32MB = 32MB total
Memory: 26580KB available (1186K code, 269K data, 48K init)
Dentry-cache hash table entries: 4096 (order: 3, 32768 bytes)
Inode-cache hash table entries: 2048 (order: 2, 16384 bytes)
Mount-cache hash table entries: 512 (order: 0, 4096 bytes)
Buffer-cache hash table entries: 1024 (order: 0, 4096 bytes)
Page-cache hash table entries: 8192 (order: 3, 32768 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
```

<그림 3-19> 임베디드 리눅스 포팅 결과

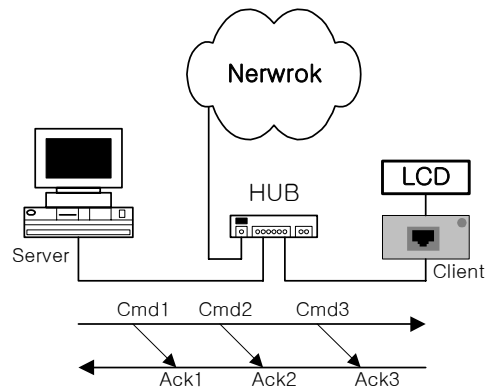
<Fig. 3-19> Result of Embedded Linux

<그림 3-20>는 원격 제어시스템 회로를 구현한 평가회로 사진이며, <그림 3-21>은 구현된 시스템의 평가 절차이다.



<그림 3-20> 평가회로 사진

<Fig. 3-20> Photography of evaluation circuit

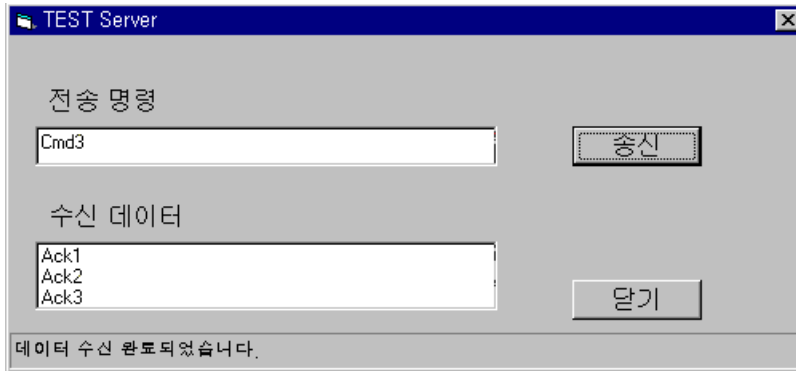


<그림 3-21> 시스템의 평가 절차

<Fig. 3-21> Evaluation procedure of system

먼저 서버 시스템에서 클라이언트 시스템으로 원하는 제어 명령 Cmd1을 전송을 한다. 클라이언트 시스템에서는 전송된 명령을 수행하고, 수행되는 과정 및 결과를 출력장치(LCD 패널)에 출력을 한 후 서

버 시스템에 그 결과를 통보하도록 하였다. 이러한 제어 명령을 Cmd3 까지 수행을 하여 만족한 결과를 얻었으며, 그 수행한 결과는 <그림 3-22>, <그림 3-23>에 나타내었다.



<그림 3-22> 제어명령 전송 및 응답

<Fig. 3-22> Control command transmission and response



<그림 3-23> 제어명령 실행

<Fig. 3-23> Control command execution



## 제 4 장 결 론

임베디드 시스템은 시스템에 내장된 형태로 존재하여 제한되고 전문화된 기능을 수행한다. 또한, 임베디드 OS는 이러한 특수목적용으로 사용되는 하드웨어에 포팅된 운영체제이다.

고가이면서 커널의 재구성이 어렵다는 기존의 사용 운영체제가 가지고 있는 문제점을 해결할 수 있고, 효율성과 신뢰성을 가지고 있는 임베디드 리눅스는 원격 제어시스템 구축에 있어서 최상의 임베디드 OS이다.

본 연구에서는 임베디드 리눅스를 이용한 네트워크 접속 기술과 타겟 보드를 제작하여 TCP/IP를 이용한 원격 제어시스템의 기본 기능을 구현하였다. 임베디드 리눅스 커널을 분석하여 개발을 위한 기초자료를 확보하였고, 부팅 과정과 부팅코드 개발을 위한 자료들을 정립함으로써, 실제 임베디드 네트워크 기반의 다양한 네트워크 서비스가 지원되는 타겟 보드개발을 위한 기초자료로 활용할 수 있으리라 기대되며 원격 제어시스템 구현을 위한 연구를 통하여 다음과 같은 결론을 얻을 수 있었다.

본 연구에서는 Intel 사의 StrongARM 프로세서를 이용하여 TCP/IP를 이용한 원격 제어시스템의 기본적인 기능만 구현하였으나, 임베디드 리눅스 포팅기술과 제어시스템 설계의 정립으로 이 외의 프로세서를 이용한 시스템 구현의 가능성을 확보하였다.

또한, 하드웨어의 기본 회로를 중심으로 구현하였기 때문에, 확장된 디바이스 드라이버 구현 및 응용기술에 대한 점은 제외하였으나, 임베디드 네트워크 환경에서의 적용성을 얻을 수 있었다.

최근에 개발되는 내장형 네트워크 시스템들은 단순한 접속기능과

TCP/IP 기반으로 하는 패킷전송만을 목적으로 하고 있으나, 보다 많은 어플리케이션 기능들을 구현하기 위해 기술적으로 부족한 시점에서 유용한 자료로 활용 될 수 있으리라 판단된다.

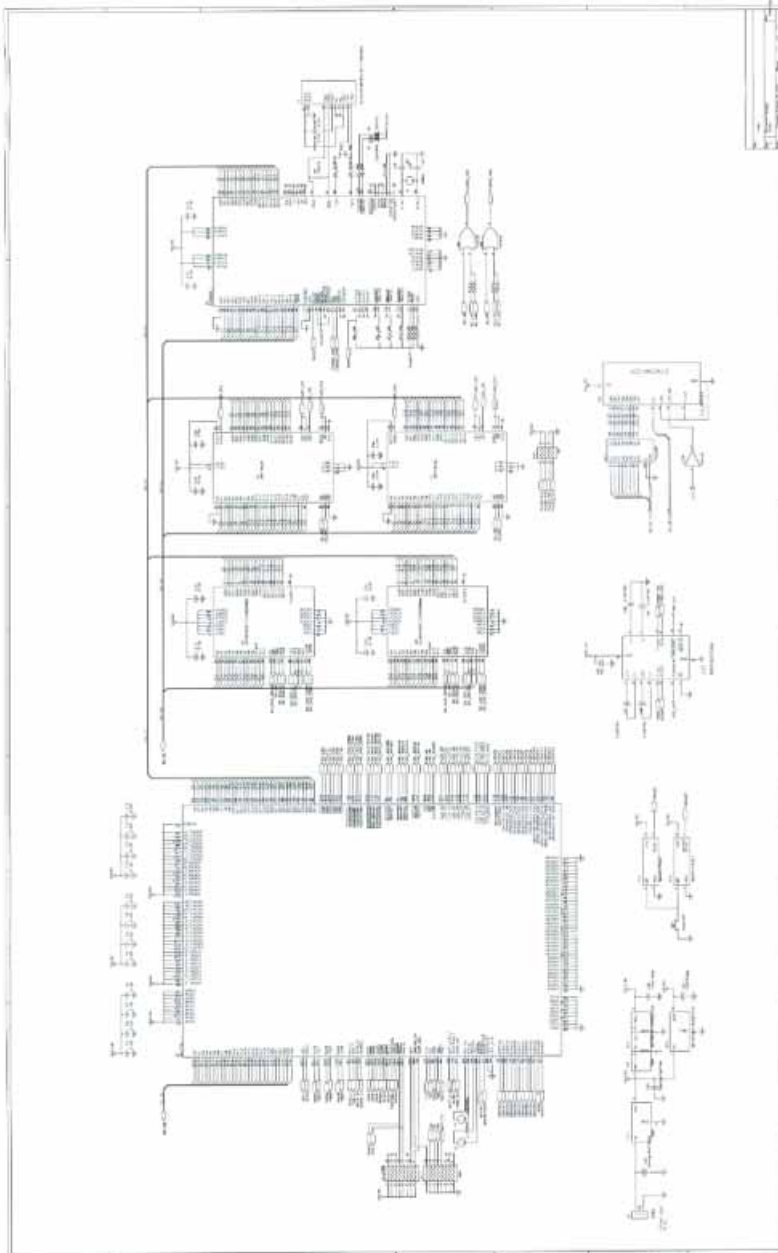
추후 다양한 디바이스 드라이버 인터페이스 및 서비스 기능에 대해 보완을 진행할 것이며, 기존 통신시스템에 사용되는 기능의 고도화에 대해서 더욱 연구할 계획이다.

## 참 고 문 헌

- [1] 김문자와외 2명, “네트워크 기반 대규모 임베디드시스템의 상호협동을 위한 Smart Message 기법”, 정보처리학회지, 제9권, 제1호, pp.60~69. 2002.
- [2] 노영욱외 1명, “임베디드 리눅스 개발도구 기술동향,” 정보처리학회지, 제9권, 제1호, pp.35~42. 2002.
- [3] 이호외 1명 역, 「리눅스 커널의 이해」, 한빛미디어, 2001.
- [4] 조유근외 2명, 「커널 프로그래밍」, 교학사, 2002.
- [5] 임근수, “범용 운영체제 구현을 위한 리눅스 커널 완전 분석”, 연세대학교, 2001.
- [6] J.D&T INC., FALINUX EZ Board Manual, 2002.
- [7] 박재희 역, 「임베디드 리눅스-하드웨어, 소프트웨어, 인터페이스」, 정보문화사, 2002.
- [8] 박제현, 정재원 역, 「리눅스 C 프로그래밍」, 인포북, 2000.
- [9] Steve furber, 「ARM system-on-chip architecture」, Addison-Wesley, 2000.
- [10] INTEL Inc., SA-1110 User-manual, 2000.
- [11] CIRRUS LOGIC INC., CS8900A Datasheet, 2001.
- [12] 박준철 역, 「TCP/IP 소켓 프로그래밍」, 사이텍미디어, 2001.
- [13] Warren W. Gay, 「Linux Socket Programming」, QUE, 2000.
- [14] 썸틀기획 편저, 「비주얼베이직 인터넷 프로그래밍」, 영진출판사, 2001.
- [15] 권덕용외 1명, 「Linux 서버 관리」, 이비컴, 2002.

## 부 록

### 1. Target Board 시스템 회로도



## 감사의 글

지난 2년동안 부족한 저를 배려하고 지도해 주신 김기문 교수님께 진심으로 감사 드리며, 교수님의 지도와 격려에 더욱 열심히 했어야 했다는 아쉬움이 남아 죄송한 마음뿐입니다. 또한 바쁘신 일정에도 틈틈히 격려를 해주시며, 끝까지 논문을 지도해 주시고 심사해 주신 양규식 교수님, 임재홍 교수님께도 감사의 말씀을 올립니다.

그리고, 학교생활을 지도해 주신 홍창희 교수님께 감사드리며, 학과 교수님께도 감사를 드립니다.

아울러, 논문을 완성하기까지 아낌없는 지원을 해주신 유형열 선배님께 감사드리며, 사심없이 도와준 이두석 선배님, 김귀찬씨에도 감사드립니다. 또한 생활을 같이 하며 지내던 박인용, 정창우, 최재영 후배와 저에게 위로와 격려를 아끼지 않았던 통신공학실험실 여러 선배님들에게도 고마움을 전합니다.

항상 곁에서 도움을 준 친구들인 김광덕, 김덕근, 안상준 군에게 고마움을 전하며, 학과 동기들에게도 감사한 마음을 전합니다.

어려운 여건 속에서도 자식들 뒷바라지를 하시며 격려와 용기를 주신 부모님께 감사드리며, 옆에서 지켜봐준 영희, 향희 두 동생의 사랑이 큰 힘이 되었습니다.

많은 분들의 도움으로 이루어진 논문인 만큼 더욱 열심히 살라는 가르침으로 알고 마음에 새기며 항상 노력하여 저를 지켜보고 계시는 주위의 은인들에게 보답하겠습니다.