Running ARMLinux on the EBSA-285

ERIC ENGSTROM
Computer Science Department
College of Engineering
California Polytechnic State University,
San Luis Obispo
11th December 2001
Copyright ©2001, Eric Engstrom.
Abstract

This paper documents the implementation of a LINUX system on a SA-110/21285 Evaluation Board (EBSA-285), as part of the Cal Poly Intelligent NIC project, by the Cal Poly Network Performance Research Group. The Cal Poly Intelligent NIC project, or CiNIC, seeks to remove the IP network stack from a host’s CPU, and place it on a dedicated co-processor board. The project seeks to explore performance advantages of an Intelligent Network Architecture.

The CiNIC hardware consists of a 21554 PCI-to-PCI Non-Transparent Bridge Evaluation Board (which provides a PCI bus, and a non-transparent bridge to a host’s PCI bus), the EBSA-285 (a single-board computer on a PCI card), and a 3Com 3c905C network interface card. The purpose of this Senior Project was to provide a LINUX system to run on the CiNIC.

Any Linux system requires a LINUX kernel and a root filesystem. Using an i386-to-ARM cross-compiler, the LINUX kernel and a customized root filesystem were prepared for the CiNIC.

Upon booting the EBSA-285, the LINUX kernel can be uploaded via a serial connection, or served over the network via BOOTP and TFTP. Once the kernel is executing, it mounts its root filesystem from a server on the network via NFS.

The system detailed in this Senior Project is a complete LINUX system, and is capable of performing any task that a workstation or server running LINUX is capable of. This provides a solid foundation for the research group to implement the Intelligent Network Architecture concept.
# Contents

1 Introduction 4  
1.1 The Cal Poly/3Com Project in Brief 4  
1.2 Intelligent Network Architecture 5  
1.3 CiNIC: The Cal Poly Intelligent Network Interface Card 6  
1.3.1 The Host 8  
1.3.2 The Card 8  
1.3.3 LINUX on the EBSA-285 9  

2 Hardware 10  
2.1 EBSA-285 STRONARM 110/21285 Evaluation PCI Board 10  
2.1.1 Flash ROM 11  
2.1.2 Boot Sequence 12  
2.2 21554 PCI-to-PCI Non-Transparent Bridge Evaluation Board 13  
2.3 Network Interface Card 14  

3 Building ARMLINUX 15  
3.1 Cross-Compilers 16  
3.2 Building the ARMLINUX Kernel 18  
3.2.1 Obtaining the Source 18  
3.2.2 Files to Edit 19
6.2.2 Network Services ........................................ 50

7 Appendices ......................................................... 52
  7.1 Source Listings ............................................. 52
  7.2 Useful man pages .......................................... 55
Chapter 1

Introduction

This Senior Project was completed as part of the Cal Poly Intelligent NIC project, by the Cal Poly Network Performance Research Group. This paper documents the implementation of a Linux system on a SA-110/21285 Evaluation Board (EBSA-285).

1.1 The Cal Poly/3Com Project in Brief

Since 1996, the 3Com corporation has funded a research group at Cal Poly, San Luis Obispo. Typically, the research group consists of one or more professors from the Electrical Engineering department, the Computer Science department, and the Computer Engineering program, and anywhere from half a dozen to a dozen masters and undergraduate students from those disciplines.

The objective of the research group is to investigate the maximization of network performance, from a user’s perspective, on client PCs. In general, the performance of a network, from the user’s perspective, centers on how much time elapses between a user’s request (for example, clicking a hyperlink in their WWW browser) and when they receive a result (the requested page is dis-
played). Some primary factors impacting network performance include available bandwidth, utilization of available bandwidth, and the utilization of a client’s CPU in processing a request.[10]

CPU utilization is an interesting aspect of network performance. While most CPU cycles spent processing a user’s network request are spent in user-level applications (i.e. WWW browsers, etc.), a significant fraction of it is spent in kernel space, executing code in the network stack. Even in a very fast network stack, the overhead necessary to move a request down the stack, and move the response from the network back up the stack, can be significant. And time spent in the network stack is time not spent in the user’s application.

Reducing the CPU utilization by a client’s network stack, and quantifying the perceived performance benefit to the user, is the primary interest of the research group.

There is little improvement left to be done in the design and implementation of the network stacks themselves. The TCP/IP stacks in modern operating systems such as Linux and Windows NT/2000 are already mature and quite fast. The next logical step to reduce CPU utilization by the network stack is to take the same approach as PC video and sound: namely, a dedicated network co-processor.

To this end, the research group has designed, and is currently implementing, an intelligent network architecture. This Senior Project is a part of that effort.

1.2 Intelligent Network Architecture

What is an intelligent network architecture? Rather than simply moving bits to and from a network, and depending upon the CPU and OS to handle the high-level network functions (as provided by the network stack: i.e. encapsulation, error checking, reliability, etc.), as current network interfaces (typically, in the
form of a PCI Ethernet card) do, an intelligent network interface will contain and run its own network stack. By running its own network stack, an intelligent network interface removes the burden of network communication from the CPU, thus freeing the CPU for other tasks.

While the concept of a network co-processor itself seems promising, an intelligent network interface—being a high-level (i.e. Class 3) network device in its own right—is capable of doing much more than network stack grunt work. An intelligent network interface may provide all manner of network services to its host PC. Other Theses and Senior Projects by members of the research group investigate some of these possibilities, including network data caching\(^1\), security and firewalling, network address translation, and quality-of-service (QoS)\(^2\).

Once the intelligent network architecture is fully realized, the research group will begin to quantify its performance from a user perspective, as compared to a conventional system (i.e. with a “dumb” NIC and all network services running on the CPU). It is hoped that this research will demonstrate the intelligent network architecture concept as a viable means to improving network performance.

### 1.3 CiNIC: The Cal Poly Intelligent Network Interface Card

The CiNIC—Cal Poly Intelligent Network Interface Card—will be the research group’s implementation of the intelligent network architecture concept.

\(^1\)Caching frequently-accessed network data will reduce latency to the user, as well as network load.

\(^2\)QoS deals with the management of different types of network traffic, based on application type and priority.\[20\]
DOWS NT/2000, with the network stack bypassed in favor of a device driver for communicating with the intelligent card.

The “card” itself consists of several components:

**21554 PCI-to-PCI Non-Transparent Bridge Evaluation Board** A non-transparent PCI bridge connects two PCI busses together, but separates the two busses into different processor domains, so that a CPU on one side of the bridge cannot interfere with a CPU or resources on the other side of the bridge. The 21554 Evaluation Board features a 21554 and four PCI slots on the secondary PCI backplane.[12] The 21554 Evaluation Board is connected to the host PC’s PCI bus. See Section 2.2.

**SA-110/21285 Evaluation Board (EBSA-285)** This is essentially a computer on a PCI card. It features a STRONGARM-110 processor, RAM, and Flash-ROM.[13] The EBSA-285 is connected to the 21554’s secondary PCI backplane, and serves as the secondary backplane’s primary controller. See Section 2.1.

**ARMLINUX** A port of the LINUX Operating System to STRONGARM processors. For the CiNIC, ARMLINUX runs on the EBSA-285. ARMLINUX is the primary concern of this Senior Project. See [15].

**NIC** A 3Com 100BaseT Ethernet card is connected to the 21554’s secondary PCI backplane, and is accessible by the EBSA-285 (but not the host). See Section 2.3.

**SCSI/IDE Interface and Hard Disk** Future work (Section 6.2.1) will include support for a SCSI or IDE hard disk to store ARMLINUX and its filesystem, cache files, and other software.
1.3.1 The Host

The host is a typical PC running Linux or Windows NT/2000. Connected in one of the host’s PCI slots will be the CiNIC. Because the 21554 is a non-transparent bridge, the entire CiNIC system will appear to the host as a single device.

The host requires a device driver to communicate to the CiNIC. The device driver implements a communication protocol, which enables it to communicate to a similar device driver running on the EBSA-285, across the 21554 non-transparent bridge.

Since the CiNIC (and, specifically, the EBSA-285) will be running the network stack, the host’s network stack is no longer necessary. However, to enable user applications to use the CiNIC without modification, the interface to the host’s network stack is replaced with an interface which directs application calls to the network stack to the CiNIC’s device driver.

Host functionality is being implemented by other members of the research group, and is the subject of other Senior Projects and Theses.

1.3.2 The Card

The 21554 Evaluation Board serves as the “base” for the CiNIC system. The 21554 board plugs into the host’s PCI bus, and provides four PCI slots on a secondary PCI backplane. The secondary backplane is separated from the host’s PCI backplane by the 21554. Since the 21554 is non-transparent, the entire host will appear to the CiNIC system as a single PCI device (and vice versa).

The EBSA-285 is the “heart” of the CiNIC system, and serves as the network co-processor. The EBSA-285 is a PCI card, and contains a StrongARM-110 RISC processor, 16MB (expandable) of RAM, and 4MB of Flash-ROM for boot software and non-volatile storage.[14] The EBSA-285 is plugged into the
21554’s secondary PCI backplane, and serves as its primary controller.

The EBSA-285 runs ARMLinux, the port of Linux to the StrongARM line of processors. This Linux system is responsible for receiving network requests from the host (via the 21554) and sending them out to the network (via its network stack), and receiving traffic from the network (via its network stack) and forwarding it to the host (via the 21554). Like the host, ARMLinux requires a device driver and communication protocol to communicate with the host via the 21554 bridge. Additionally, the ARMLinux system will host network services provided by the CiNIC; see Section 6.2.2.

Also plugged into the 21554’s secondary PCI backplane is a 3Com 100BaseT Ethernet NIC. This card is the interface to the network. (Future work may include the addition of additional NICs, thus allowing the CiNIC system to provide firewall, routing, and network address translation services for an entire LAN; see Section 6.2.2.)

Future work includes the addition of a SCSI or IDE interface to the secondary PCI backplane, and a local hard disk for use by ARMLinux on the EBSA-285. Uses for a local hard disk include the storage of ARMLinux and its filesystem, storage for other software, and storage for network data caches. See Section 6.2.1.

### 1.3.3 Linux on the EBSA-285

The CiNIC system runs ARMLinux (the StrongARM port of Linux) on the EBSA-285. ARMLinux provides a robust TCP/IP stack, and, being a full Linux system, provides a platform on which to develop other network tools, such as network data caching, QoS, security, etc.; see Section 6.2.2.

As part of the effort of implementing the CiNIC system, building and running a Linux system for the EBSA-285 is the purpose of this Senior Project.
Chapter 2

Hardware

The CiNIC consists of two major hardware components: the EBSA-285, which is essentially a STRONGARM-based computer on a PCI card, and the 21554 PCI-to-PCI Non-Transparent Bridge Evaluation Board, with provides a secondary PCI backplane with four PCI slots.

2.1 EBSA-285 STRONGARM 110/21285 Evaluation PCI Board

The EBSA-285 is single-board computer manufactured by Intel, and is supplied as an add-in PCI card. It is an “evaluation board” for Intel’s STRONGARM 110 microprocessor and 21285 Core Logic chip (collectively known as the FootBridge architecture), and is intended to provide a reference platform for developers to test and develop systems that use the STRONGARM 110 microprocessor and the 21285 core logic chip.[14]

The STRONGARM 110 microprocessor is a RISC design, and is compliant
with the ARM architecture\textsuperscript{1}. On the EBSA-285, the StrongARM-110 processor can be configured (via jumpers) to run at one of 16 clock frequencies, ranging from 88.3MHz to 287MHz.[14]

The 21285 is the Core Logic chip for the StrongARM-110 processor.[14] A core logic chip provides basic services to the CPU, including access to RAM and ROM, the PCI bus, and a standard serial port, located on the card’s mounting bracket.

The EBSA-285 provides two SDRAM sockets. Typically, the EBSA-285 ships with one 16MB DIMM. Additionally, the EBSA-285 provides 4MB of Flash ROM.

The EBSA-285 is provided as a PCI board. It can be configured as the Host Bridge of the PCI bus (that is, it is the primary device on the PCI bus, and receives interrupts from other devices on the PCI bus), or as an add-in card (that is, it sends interrupts to another device on the PCI bus). For the CiNIC, the EBSA-285 is always configured as a Host Bridge.

2.1.1 Flash ROM

The EBSA-285 provides 4MB of Flash ROM (non-volatile memory that can be erased and reprogrammed) for non-volatile storage. The Flash ROM’s primary purpose is to provide boot code for the EBSA-285. The Flash ROM is divided into 16 blocks (or images), numbered 0 through F, which can be erased and programmed independently.

As shipped from Intel, image 0 contains the bootstrap code for the EBSA-285 (called the Primary Boot Loader, or PBL). Image 1 contains a diagnostics program, which reports its results over the serial port. Images 2 through F are available for user programs.

\textsuperscript{1}For more information on the ARM architecture, see \url{http://www.arm.com}. 
The Flash ROM is programmed using the Flash Management Utility (FMU), which is shipped with the EBSA-285 as source, and as a DOS executable (additionally, an Angel debugger plug in is available). To flash the EBSA-285, the card must be configured as an add-in card, held in blank-programming mode by placing a jumper on J15, pins 5-6, and installed in a PC running DOS.

The FMU has a command-line interface, and provides the following commands: help, list, listblocks, testblock, delete, deleteblock, deleteall, program, quit. For information on the FMU, see Chapter 7 in [14]; also see [8].

2.1.2 Boot Sequence

When the EBSA-285 first comes out of reset, it begins to execute the code at location 0, which is image 0 in the Flash ROM. Typically, image 0 contains the Primary Boot Loader (PBL). The primary purpose of the PBL is to determine which of the 15 other images to execute, to load that image into RAM (if necessary), and then branch to it.

To determine which ROM image to execute, the PBL examines the flash image selector switch, a 16-position rotary switch fitted to the EBSA-285's mounting bracket. If the switch is in position 0, the PBL will branch to the Angel debugging stubs (used in conjunction with the Angel debugging environment, which was not used for this project). If the switch is in any position from 1 to F inclusive, the PBL will perform the following tasks (as relevant to the booting of the ARMLINUX kernel; other boot modes are not covered here):

1. Switch the memory map.

2. Verify the checksum of the selected image.

3. Initialize the DRAM.
4. Load the image into DRAM.

5. Jump to the image’s entry point.

At this point, the EBSA-285 will be executing the user program indicated by the flash image selector switch from RAM.

### 2.2 21554 PCI-to-PCI Non-Transparent Bridge Evaluation Board

The 21554 is a PCI-to-PCI Non-Transparent Bridge, manufactured by Intel. A non-transparent bridge separates two PCI busses into separate processor domains, but provides a method for devices on the two busses to communicate. To the Host Bridge on a PCI bus, the 21554—and the PCI bus behind it—appear as a single PCI device, accessible with a device driver (such a device driver was under development by the research group when this Senior Project was completed).

The 21554 Evaluation Board is a board manufactured by Intel, and is intended for developers to test and develop for the 21554. This board contains its own PCI backplane with 4 PCI slots, and a PCI connector so the board can be plugged into a host’s PCI slot. The 21554 on the Evaluation Board bridges the PCI bus that the board is plugged into, and its own PCI backplane.

For the project, the 21554 Evaluation Board was typically plugged into the PCI slot on a host PC. Plugged into the slots on the 21554 Evaluation Board were the EBSA-285, and the 3Com 3c905C NIC.

For a discussion of moving data between the two PCI busses via the 21554, see [5].
2.3 Network Interface Card

To place the CiNIC on the network, a 3Com 3c905C Network Interface Card (a 10/100BaseT Ethernet card manufactured by 3Com) was installed in one of the PCI slots on the 21554 evaluation board. In this configuration, the NIC was on the secondary PCI bus, and thus available to the EBSA-285 as a PCI device.

The hardware address of the NIC must be known by the BOOTP server; see Section 4.3.2.
Chapter 3

Building ARMLINUX

LINUX is a clone of the UNIX Operating System, developed by Linus Torvalds and a loose collection of programmers from around the world. It aims toward POSIX compliance.[23] ARMLINUX is the port of LINUX to ARM-based machines, and is maintained by Russell King. ARMLINUX currently runs on more than 50 different ARM-based machines, including the EBSA-285.[15]

Any LINUX system is comprised of two primary components: the kernel, and the root filesystem.

The kernel is the actual program that is responsible for controlling the computer and all other software. The kernel bootstraps (brings up) the machine, handles errors, traps, and interrupts, and provides system services (memory management, process scheduling, etc.) to other software.

The root filesystem is typically located on a disk, and contains all the rest of the files, executables, directories, etc. that make up a LINUX system. The root filesystem is mounted by the kernel during bootstrapping. If the kernel cannot find or cannot mount a root filesystem during boot, it will panic\(^1\) and halt.

\(^1\)Panic: when execution in the kernel cannot continue due to some fatal error, issue an error message and halt.
The first challenge faced by the research group was building an ARM-Linux kernel. The next challenge was assembling a root filesystem.

### 3.1 Cross-Compilers

Building software from source (most free software, including Linux, is distributed as source) requires a compiler and a set of associated utilities collectively referred to as a toolchain. Together, the compiler and toolchain translate the human-readable instructions in source files to a machine-language file called an executable.

Unfortunately, compilers are platform-dependent—that is, a given compiler and toolchain can only produce executables for one type of computer and operating system (i.e. Linux on an i386\(^2\)). And because Linux on the i386 is a different platform than ARM-Linux on the StrongARM-110, the standard compilers present on the research group’s Linux-based PCs did not suffice for building ARM-Linux.

Further clouding the issue is the fact that compilers typically build for the same platform on which they run. That is to say, a compiler running on an i386/Linux platform will generate executables for i386/Linux, and a compiler running on a StrongARM/ARMLinux platform will generate executables for StrongARM/ARMLinux. This raises an interesting problem: since the research group didn’t (at this point) have a StrongARM/ARMLinux system on which to compile ARMLinux, how did the research group build an ARMLinux system?

The answer lies with cross-compilers. A cross-compiler is a compiler and toolchain that runs on one type of platform (say, i386/Linux), but generates executables for another type of platform (say, StrongARM/ARMLinux). A

---

\(^2\)i386: Intel 80386, 486, Pentium, etc., and their clones (i.e. AMD and Cyrix).
i386-to-ARM cross-compiler would facilitate the building of an ARMLINUX system from i386-based PCs running LINUX. Fortunately, the standard gcc compilers from the Free Software Foundation can be configured as cross-compilers.

In the event, building a gcc cross-compiler from source proved to be difficult. It was decided, in the interest of moving the project forward, that the research group would split its effort. The author moved ahead with building an ARMLINUX system using a pre-built i386-to-ARM cross-compiler maintained by ARMLINUX developer Rod Stewart, and available from ftp://ftp.netwinder.org/users/s/stewart/cross/[21]. Meanwhile, Jim Fischer of the research group worked to build a cross-compiler from the gcc source. See Fischer’s Thesis [8] for details.

Stewart’s cross-compiler and toolchain, once installed, resided in the /usr/armv4l-linux directory on the author’s workstation. The name of every program in the package, including gcc itself, was named with the “armv4l-unknown-linux-” prefix, to differentiate the i386-to-ARM cross-compiler and chain from the workstation’s own native compilers. Thus, to compile a source file with the cross-compiler, the author would type the full pathname to the compiler:

```
$ /usr/armv4l-linux/bin/armv4l-unknown-linux-
gcc foo.c
```

Since most software is compiled with special build scripts called makefiles, the author rarely used this command directly. Rather, before compiling a piece of software with the cross-compiler, the author would modify the software’s makefiles to use the cross-compiler rather than the system’s native compiler. Some software packages (such as the LINUX kernel) handled cross-compiling elegantly and simply, typically requiring no more than a few lines of modifications to the makefiles. Other software (such as bash) was never designed

---

4 bash: the most common shell on LINUX systems.
with cross-compiling in mind, and presented continuing headaches.

### 3.2 Building the ARMLinux Kernel

Once the i386-to-ARM cross-compiler is in place, compiling the Linux kernel to run on ARM hardware is relatively straightforward. Be sure to read [16] before proceeding.

#### 3.2.1 Obtaining the Source

First, obtain the Linux kernel source, available at [http://www.kernel.org](http://www.kernel.org). For this project, the kernel used was linux-2.3.99-pre9. Decompress the source, preferably somewhere like `/usr/local/src` (do *not* overwrite your normal source tree!):

```bash
# mv linux-2.3.99-pre9.tar.gz /usr/local/src
# cd /usr/local/src
# tar -xzf linux-2.3.99-pre9.tar.gz
```

Then, obtain the latest ARMLinux patch from the ARMLinux FTP server: [ftp://ftp.arm.linux.org.uk](ftp://ftp.arm.linux.org.uk). Place the patch file in `/usr/local/src`, and apply it to the Linux kernel tree:

```bash
# mv patch-2.3.99-pre9-rmk1.gz /usr/local/src
# cd /usr/local/src/linux
# gunzip -c ../patch-2.3.99-pre9-rmk1.gz | patch -p1
```

The source tree is now in place.
3.2.2 Files to Edit

A couple of files in the LINUX source tree need to be edited before configuring the kernel.

First, the top-level makefile must be edited. Load the top-level makefile (linux/Makefile) in your favorite editor, and make the following changes:

1. Change the line that begins with ARCH to read ARCH = arm.

2. Set the CROSS_COMPILE variable to point at the cross-compiler’s location and prefix (that is, the entire path minus “gcc”) i.e.:

```
    CROSS_COMPILE = /usr/armv4l-linux/armv4l-
                    unknown-linux-
```

Next, the serial port driver must be modified to use the same serial port speed as the ARMLINUX BIOS, so output from the BIOS and the kernel are both visible on the EBSA-285 serial port (see Section 4.1). Open linux/drivers/char/serial_21285.c in your favorite editor. In the functions rs285_init() and rs285_console_setup(), modify the case statements to make 38400bps the default speed.

3.2.3 Configuration

Configuring the ARM kernel is straightforward, but be sure to read linux/README and linux/Documentation/arm/README before proceeding. First, make sure the tree is clean:

```
    # cd /usr/local/src/linux
    # make mrproper
```

Then, invoke your favorite config program; the author prefers xconfig⁵:

⁵Besides xconfig, config, menuconfig, and oldconfig are available.
# make xconfig

Choose the following options:

- ARM system type: **FootBridge**
- EBSA285 host mode
- Networking, TCP/IP support
- IP: kernel level autoconfiguration
- IP: BOOTP support
- Support for 3Com 3c590/3c900 series Ethernet cards
- Virtual terminal
- Non-standard serial port support
- DC21285 serial port support
- Console on DC21285 serial port
- NFS file system support
- Root file system on NFS

Most other options and drivers can simply be turned off.

Save the configuration and exit config. Finally, make sure kernel dependencies are set up correctly:

    # make dep
3.2.4 Compiling

Now, the kernel can be compiled normally. The makefile will use the cross-compiler, as specified in Section 3.2.2. Start make:

```
# make zImage
```

This will build a compressed ARMLINUX kernel. The compiled kernel will be located at arch/arm/boot/zImage.

If any kernel drivers were configured as modules, compile them next (for simplicity, no modules were compiled in the course of this project):

```
# make modules
```

The compiled modules need to be copied into a temporary directory, so they can be moved to the root filesystem later (see Section 3.3):

```
# make modules_install \
   INSTALL_MOD_PATH=/usr/local/src/arm/
```

3.3 The Root Filesystem

A LINUX system consists of much more than just the kernel. The kernel, upon booting, requires a root filesystem (the filesystem that is mounted on `/`). The root filesystem contains programs, startup scripts, the password file, and other software necessary to boot the system, and make it usable once it is running. For this project, the root filesystem was served via NFS (see Section 4.4) from another machine on the network. (Future work includes putting the root filesystem on a local hard disk; see Section 6.2.1.)

Because the ARMLINUX root filesystem is being served via NFS, it will be organized in a directory on the server’s filesystem. The author placed the ARMLINUX root filesystem in a directory called `/ebsa285fs` on the server.
Note: the permissions on the files and directories of the root filesystem should be appropriate for the users that will log into the CiNIC (as specified by the /etc/passwd file).

Much of the information in this section is from [4].

3.3.1 Directories

The root filesystem is divided into several standard directories:

/bin: basic programs
/dev: device special files
/etc: configuration files
/etc/rc.d: rc (startup) scripts
/home: user home directories
/lib: basic library files (for dynamic linking)
/mnt: mount point for mounting other filesystems
/proc: mount point for proc filesystem
/root: root home directory
/sbin: system and administration programs
/usr: user-installed programs and files
/usr/bin: user-installed programs
/usr/lib: user-installed libraries
/usr/sbin: user-installed system and administration programs
/var: logs, mail spools, etc.

These directories were placed on the root filesystem, /ebsa285fs.
3.3.2 Files

The following files were prepared by the author and placed on the ARMLINUX root filesystem. See Section 7.1 for source listings.

/dev

The /dev directory contains device special files, which represent hardware and virtual devices available on the system. These files can be created by hand with the `mknod`\(^6\) command, but it is easier to simply copy them from an existing Linux installation using `cp -R` (note: do not copy a device special file with `cp -r`!). A minimal ARMLINUX/EBSA-285 /dev directory contains the following files:

- **console**: the console
- **cua0-cua3, ttyS0-ttyS3**: serial port ttys
- **tty0-tty3**: standard ttys
- **pty0-ptyf, ttyp0-ttypf**: telnet ttys

/etc

The /etc directory contains the system’s configuration files. Each configuration file is a text file with a specific format.

- **fstab**: mountable filesystems
- **gettydefs**: getty setup file (for serial getts)
- **group**: login groups

\(^6\)See `man mknod` on a LINUX system.
**inetd.conf**: inetd superserver configuration file (for telnetd and other network services)

**inittab**: the script run by init on boot; this is the first startup script to execute

**ld.so.cache**: cache for dynamic linker (this must be generated with ld-config; see Section 3.3.3)

**ld.so.conf**: library paths for dynamic linker

**passwd**: username/password file

**shells**: list of available shells

**/etc/rc.d**

This directory contains the system’s startup and shutdown scripts. These scripts are executed by init, as specified by the /etc/inittab file. When the system is booted, startup will execute first, followed by either multi or single. When halting, halt will be executed. When rebooting, reboot will be executed.

**halt**: runs when the system is halted. Unmount filesystems (and release other resources) and call the command halt\(^7\).

**multi**: runs when booting into multiuser mode (i.e. multiple users can login), typically runlevel 3 in /etc/inittab. Starts the inetd superserver daemon and flushes /var/run/utmp.

**reboot**: runs when the system is being rebooted (rather than halted). Unmount filesystems and call reboot\(^8\).

\(^7\)See man halt on a LINUX system.

\(^8\)See man reboot on a LINUX system.
**single**: runs when booting into *single-user mode* (i.e. only a single user can login), typically runlevel 1 in `/etc/inittab`. Usually this script does nothing, as single-user mode is only entered to repair a damaged system.

**startup**: this is the first script run by `init`. Mount filesystems, set the system clock, and perform other initializations.

/var

The `/var` directory contains temporary system files, log files, mail spools, and the like. There are two required subdirectories, `/var/log` and `/var/run`.

/var/log/wtmp: this file is a log of all logins. When the filesystem is first assembled, this is an empty file, and it accumulates information over time. To create the initial wtmp file:

```bash
# cd /ebsa285fs/var/log
# echo > wtmp
```

/var/run/utmp: a listing of users currently logged in. This file needs to be flushed at every boot (typically by `/etc/rc.d/multi`), or stale logins from previous sessions will confuse the system.

3.3.3 Executables

Finally, the following programs were compiled with the i386-to-ARM cross-compiler (see Section 3.1) by the author, and placed on the root filesystem.

**glibc 2.1**

Most programs on a *Linux* system link dynamically—that is, standard calls to the C library (such as `printf()`, etc.) and other libraries are not compiled
into the executable, but rather are linked at runtime by the dynamic linker. Because of this, a LINUX system requires, at a minimum, a C library for dynamic linking.

The GNU C Library, known as glibc, is the standard C library used by most LINUX systems. glibc is available at http://www.gnu.org/software/libc/libc.html; see [9]. For the ARMLINUX filesystem, glibc 2.1 was used. Additionally, several add-on packages (such as crypt) were included.

Before attempting to compile glibc, read the included documentation. This section assumes that the glibc tarball was decompressed to /usr/local/src. First, create a build directory in /usr/local/src (the glibc documentation warns against building in the source directory [9]):

```bash
# mkdir /usr/local/src/glibc-build
# cd /usr/local/src/glibc-build
```

glibc supports a number of platforms, including ARM, and the build scripts were designed with cross-compiling in mind. From within the build directory, run the configure script (this assumes that the cross-compiler executable is in the $PATH):

```bash
# ../glibc-2.1/configure \
   --host=arm-unknown-linux \
   --build=i386-redhat-linux \
   --prefix=/ebsa285fs --enable-add-ons
```

The host argument determines what platform the library will run on (ARM), while the build argument determines what platform the library is being compiled on (i386). That the two don’t match indicates that the library is being cross-compiled. The prefix argument determines where the lib directory
the library will be installed to is located (i.e. a value of /ebsa285fs means to install in /ebsa285fs/lib; see Section 3.3).

Then, run make. Note: this takes several hours, even on fast hardware.

    # make

If no errors were generated (make didn’t die with lines of output beginning with “***”), install to the directory specified to configure earlier:

    # make install

glibc should now be installed in /ebsa285fs/lib.

(Note: if size is a concern, only the installed library files fitting a ’*.so’ pattern are really necessary for glibc to function. Further, these files can be stripped of symbols and debugging information with the cross-compiler’s strip command (typically armv4l-unknown-linux-strip).)

The dynamic linker (/lib/ld*.so) looks up the locations of all library files from a library cache file; if this cache does not exist, the linker will not be able to find any of the libraries. To build the cache for the ARMLinux root filesystem, run ldconfig:

    # ldconfig -r /ebsa285fs

glibc is now installed on the ARMLINUX root filesystem.

ash

ash is a shell that provides most of the functionality of bash, but is much smaller. ash is used by several Linux distributions (including RedHat and

---

9See man strip on a LINUX system.
10See man ldconfig on a LINUX system.
11A shell is the program that displays the command line to the user and executes commands.
12bash is the most popular shell for Linux. See man bash on a LINUX system.
to make rescue diskettes. This shell was chosen over bash due to the difficulty encountered in cross-compiling bash.

There is no homepage for ash, but RPMs and LINUX patches are available from RedHat, http://www.redhat.com, and RPMFind, http://rpmfind.net/. For this project, ash-0.2, with supplied LINUX patches applied, was used.

Unfortunately, the ash makefiles build intermediate executables, which are used later in the compile process. This made cross-compiling a bit of a chore, as intermediate executables need to be in the native (i386) format, or they won’t execute during the build.

To cross compile ash, first do a native build. This will build the intermediate executables needed:

```
# cd ash-linux-0.2
# make
```

Next, open the makefile in your favorite editor, and modify the ‘clean:’ target so that a make clean will not remove mkinit, mknodes, and mksyntax. Next, comment out all occurrences of ‘-lbsd’ so that the compiler will not try to link to BSD libraries (doing so will cause the link to fail). Finally, set the $CC variable to point to the cross-compiler; this can either be done in the environment, or by adding the line

```
CC=/usr/armv4l-linux/armv4l-unknown-linux-gcc
```

at the top of the makefile.

Then, recompile the package:

```
# make clean; make
```
This produces the executable `sh`. The author prefers to rename the executable `ash` and provide a symbolic link `sh` when installing on the ARMLINUX filesystem:

```bash
# cp sh /ebsa285fs/bin/ash
# cd /ebsa285fs/bin
# ln -s ash sh
```

**TinyLogin**

TinyLogin, originally written for embedded systems and rescue disks, is a small suite of utilities that handles logins, authentication, and management of users, groups, and passwords. It duplicates the functionality of the following commands in a single executable: `adduser`, `addgroup`, `deluser`, `delgroup`, `login`, `su`, `sulogin`, `passwd`, and `getty`. TinyLogin is maintained by Erik Andersen, and is available from http://tinylogin.lineo.com; see [2]. Supported commands and usage information is documented at http://tinylogin.lineo.com/TinyLogin.html.

Using TinyLogin rather than compiling each of those commands separately reduced development time considerably. For the project, TinyLogin v.0.77 was used.

Compiling TinyLogin is straightforward. First, set the `$CC` variable to point to the cross-compiler, either in the makefile or the environment. Then, the makefile must be edited to use the cross-compiler’s `strip` command; replace the occurrence of `strip` with `arm4l-unknown-linux-strip` (assuming the cross-compiler executables are in the `$PATH`).

Next, open the `tinylogin.def.h` file in your editor, and comment out the line that reads “`#define TLG_SULOGIN`”. This will deactivate the `sulogin` command, which will be supplied by the sysvinit package.
Then make:

```bash
# make
```

The makefile will install the TinyLogin executable, and symbolic links that repre-
sent each supported command, organized by directory, in the ./_install
directory. Finally, the executable and links need to be copied to the ARMLINUX
filesystem:

```bash
# cd _install
# cp -d bin/* /ebsa285fs/bin
# cp -d sbin/* /ebsa285fs/sbin
# cp -d usr/bin/* /ebsa285fs/usr/bin
```

**BusyBox**

BusyBox is a companion to TinyLogin, and is also maintained by Erik An-
derersen. It is available from http://busybox.lineo.com/; see [1]. All
supported commands and usage information is documented at ftp://oss.
lineo.com/busybox/BusyBox.html. For the project, BusyBox v.0.43 was
used.

First, the makefile must be edited to use the cross-compiler. Change the
line:

```bash
CC = gcc
```

to

```bash
CC = armv4l-unknown-linux-gcc
```

and change the line:

```bash
STRIPTOOL = strip
```
to

\texttt{STRIPTOOL = armv4l-unknown-linux-strip}

Then, edit the file, \texttt{busybox.def.h}. Deactivate \texttt{halt}, \texttt{init}, \texttt{reboot}, and \texttt{sh} by commenting out their respective lines (these commands are provided by \texttt{SysVinit} and \texttt{ash}).

Compilation and installation of BusyBox is the same as TinyLogin; see above.

\textbf{SysVinit}

The \texttt{SysVinit} package, written by Miquel van Smoorenburg, provides \texttt{init}\footnote{See \texttt{man init} on a LINUX system.} (the first process started by the kernel, and parent to all processes\cite{19}) and related programs (\texttt{halt, last, killall5, reboot, shutdown}, etc.). \texttt{SysVinit} is available from \texttt{ftp://ftp.cistron.nl/pub/people/miquels/sysvinit/}; see \cite{19}. For the project, \texttt{SysVinit} v.2.78 was used.

First, edit the \texttt{"CC = cc"} line in \texttt{src/Makefile} to point to the cross-compiler. Then, run make:

\begin{verbatim}
# cd src; make
\end{verbatim}

Unfortunately, the makefile does not have a satisfactory \texttt{install} target; the executables and symbolic links must be installed by hand. This is well documented in the \texttt{doc/Install} file.

\textbf{NetKit}

NetKit is a collection of packages maintained by David A. Holland. NetKit provides various basic network daemons and utilities, including \texttt{inetd, ping, telnet, and telnetd}. The NetKit packages are available from \texttt{ftp://ftp.uk.linux.org/pub/linux/Networking/netkit/}; see \cite{11}. For the project,
only the netkit-base-0.17 and netkit-telnet-0.17 packages were necessary. netkit-base provides the inetd superserver and ping\(^4\), while netkit-telnet provides telnet (the client) and telnetd (the server daemon).

netkit-base was built and installed first, followed by netkit-telnet.

The first step in building netkit-base is modifying its configure script. Unfortunately, the configure script wasn’t written with cross-compiling in mind, as it depends on compiling and running a small test program (which, if compiled to ARM code, will not work). As a hack to ensure that configure can run, comment out all occurrences of “__conftest” in the configure script. Then run configure, followed by make and make install:

```
  # ./configure --prefix=/ebsa285fs/usr \
  --with-c-compiler=armv4l-unknown-linux-gcc
  # make
  # make install
```

That will compile and install inetd and ping on the ARMLINUX filesystem.

Now, move on to the netkit-telnet package.

netkit-telnet was written to use the termcap.h header file. However, glibc supplies the termios.h header file instead. As a result, the occurrences of “#include <termcap.h>” in telnet/telnet.cc and telnetd/telnetd.c need to be changed to “#include <termios.h>”. Further, netkit-telnet’s configure has the same problem with __conftest as netkit-base, and requires similar modifications.

Once editing in the source tree is complete, run configure, followed by make and make install:

```
  # ./configure --prefix=/ebsa285fs/usr \
  # make
  # make install
```

\(^4\)ping is a standard network diagnostic tool. See man ping on a LINUX system.
--with-c-compiler=armv4l-unknown-linux-gcc \
--with-c++-compiler=armv4l-unknown-linux-g++

# make
# make install

Finally, inetd needs its configuration file, /etc/inetd.conf (see Section 3.3.2). A good example file is included with inetd-base in netkit-base-0.17/etc.sample/inetd.conf. Copy this file to the ARMLINUX filesystem:

    # cp etc.sample/inetd.conf /ebsa285fs/etc/inetd.conf

The only line that should be uncommented in /ebsa285fs/etc/inetd.conf is this line:

    telnet stream tcp nowait root /usr/sbin/telnetd telnetd
Chapter 4

Booting ARMLINUX

Compared to building the ARMLINUX system, booting it is straightforward.

The ARMLINUX project [15] provides a BIOS/kernel loader for the EBSA-285, which can fetch and boot an ARMLINUX kernel via a Y-modem download on the serial port or over the network with the TFTP protocol. Once the kernel is running on the EBSA-285, it will mount a root filesystem via NFS (as configured in Section 3.2.3), and execute its startup scripts.

4.1 EBSA-285 BIOS

The ARMLINUX kernel isn’t capable of booting the EBSA-285 by itself; a kernel bootloader is required to fetch, load into RAM, and start the kernel (much as an i386 machine requires LILO to start a LINUX kernel). To fill this need, Russell King of the ARMLINUX project [15] has made available an EBSA-285 BIOS, which is a BIOS and kernel bootloader for the EBSA-285. The EBSA-285 BIOS is available via anonymous FTP from the ARMLINUX FTP server [17]. For the project, version 1.07 was used.

The BIOS is capable of fetching an ARMLINUX kernel (or, for that matter,
any ARM executable) from three different locations:

- Y-modem download via the EBSA-285’s serial port (Section 4.2)
- Over a local network with the BOOTP and TFTP protocols (Section 4.3)
- From a locally attached hard disk (future work; see Section 6.2.1)

### 4.1.1 Building the BIOS

Like all other ARM executables at this stage, the BIOS must be cross-compiled. Before building, consult the README file included with the BIOS source [17].

The BIOS can run alongside the EBSA-285’s PBL (see Section 2.1.1), or can completely replace it. For the project, it was decided to not replace the PBL. As such, the BIOS was typically placed in image 5 in the EBSA-285’s Flash ROM (see Section 2.1.1), and the flash image selector was set to 5.

For this setup, the TEXTADDR variable in the makefile was set as follows (per the README file, building ELF):

```make
TEXTADDR = 0x411400C0
```

The CROSS_COMPILE line was set to point to the cross-compiler:

```make
CROSS_COMPILE = /usr/armv4l-linux/armv4l-unknown-linux-
```

The BIOS was then compiled:

```bash
# make dep
# make
```

and programmed into image 5 in the EBSA-285’s Flash ROM (see Section 2.1.1).
4.1.2 The BIOS in Operation

The BIOS prints its output to the EBSA-285’s serial port. To view the output, the serial port on a PC (other than the host) should be connected to the EBSA-285’s serial port with a standard null-modem cable (referred to in this paper as the console PC). The console PC should be running a serial communication program (i.e. Telix or Hyperterminal on Windows, or Minicom on LINUX), configured to read the serial port to which the EBSA-285 is connected (typically, COM1, COM2, COM3, or COM4 under Windows, or /dev/ttyS0, /dev/ttyS1, /dev/ttyS2, or /dev/ttyS3 under LINUX). The serial connection should be configured as 38400, 8n1, no flow control (38400 baud, 8 bits, no parity, one stop bit, no flow control).

When the EBSA-285 is powered on (i.e. the host PC that the CiNIC is installed in is powered on), the PBL will start the BIOS. The BIOS will first perform a memory test. Once the memory test is completed, the BIOS will attempt its autoboot sequence. The autoboot sequence is undesirable, because it typically tries to boot from an IDE hard disk, which isn’t part of this project (see Section 6.2.1). Therefore, strike the “s” key twice to abort the autoboot sequence.

Once the autoboot sequence is aborted, the Manual Boot menu appears. This menu provides several different commands, including: Arguments, which allows command-line arguments to be passed to the ARMLINUX kernel\(^1\); Boot method, which selects the method the BIOS will use to fetch the kernel (including serial and net); pci config, which prints information about the PCI bus (helpful for debugging); and boot, which fetches and boots the kernel as specified by the other commands.

Typically, the Boot method command is used to select the desired method

---
\(^1\)Unfortunately, this was broken in v1.07. However, arguments are not necessary for this project.
for fetching the kernel (either serial or net), followed by the boot command.

4.2 Uploading the Kernel via Serial Port

The BIOS provides three methods of fetching an ARMLinux kernel, two of which were utilized for the project. The simplest method is to upload the kernel to the EBSA-285’s serial port from another computer. Being the simplest method, this method was utilized early on in the project; it was later superseded by the method described in Section 4.3.

The CiNIC (the 21554 non-transparent bridge, with the EBSA-285 and a 3Com NIC attached) must be installed in a PCI slot on a powered-down host PC. Meanwhile, the EBSA-285’s serial port should be connected to another PC (the console PC) running a serial communication program, as described in Section 4.1.2.

First, ensure that the ARMLinux kernel (the zImage file) as built in Section 3.2 is available on the console PC, and that the ARMLinux root filesystem is available via NFS on the network (see Section 4.4). Also, a BOOTP server must be available on the network (see Section 4.3.2). Although the kernel is being uploaded to the EBSA-285 via the serial port rather than being served via TFTP, BOOTP is required so the kernel can find its IP address and root filesystem.

When the host PC (the PC that the CiNIC is connected to) is powered up, the CiNIC will power up as well. At this point, the ARMLinux BIOS on the EBSA-285 will begin its memory test, and then attempt its autoboot sequence (note: the output from the EBSA-285 will be visible on the console PC). At this point, strike the “s” key twice on the console PC (running the serial communication program) to enter the Manual Boot menu.

Use the Boot method command to select the serial method, and then
choose the boot command.

The BIOS will then prompt:

Now send file with y-modem (^X to abort) ...

In the serial communication program running on the console PC, send the zImage kernel executable with the Y-modem protocol. The BIOS will receive the file (note that, at 38400 baud, the download may take a while). When the Y-modem transfer is complete, the BIOS will boot the ARMLinux kernel. The kernel’s output will appear in the serial communication program on the console PC (recall, the kernel was configured to use the serial port for its console in Section 3.2.3).

4.3 Serving the Kernel: BOOTP and TFTP

A more sophisticated method of fetching an ARMLinux kernel is to make it available from a server on the network (presumably, from the same machine that serves the root filesystem). To enable the fetching of an ARMLinux kernel over the network, the BIOS implements a small UDP/IP protocol stack, as well as clients for the BOOTP and TFTP protocols. (Much of this method is documented in [7, 18].) (Note: the console PC (Section 4.1.2) is still necessary to view and interact with the BIOS and the kernel.)

4.3.1 The Protocols

A network bootstrap (where a diskless client, such as the CiNIC, fetches its kernel from a server on the network) typically consists of two phases. The first phase, the discovery of the client’s IP (and other information), is implemented with the BOOTP protocol, while the second phase, the fetching of a bootable file (i.e. a kernel) by the client, is implemented with the TFTP protocol.
The **BOOTP protocol** (Bootstrap protocol) allows a diskless client to “discover” its own IP address, the IP of a server machine, the name of a file to fetch and/or execute, and other information [6]. The client essentially sends a BOOTP broadcast to the local network, which asks, “who am I?”. Any BOOTP server on the local network can respond with the client’s IP and hostname, the filename of an executable (typically, a kernel) to fetch in the second phase, the IP of the machine that executable can be found on, and other information.

The **TFTP protocol** (Trivial File Transfer protocol) is an extremely basic protocol for transferring files over the network [22]. TFTP is a “stop-and-wait” protocol, does not provide authentication, and is typically implemented using datagrams. In fact, TFTP is missing most of the features of a typical FTP. TFTP is designed this way so it can be implemented very simply—ideal for a kernel bootloader running from the ROM of a diskless client.

Once the client has obtained its IP address, the filename of a kernel, and the IP address of the server the kernel is stored on (via BOOTP), the client downloads the specified kernel from the specified server using TFTP.

### 4.3.2 Configuring the BOOTP Server

The machine that is going to be the BOOTP server for the local network (which will probably serve as the TFTP server and NFS root filesystem server as well) must first have the `bootpd` daemon installed—many distributions don’t include it by default. Once installed, configure the server’s `inetd` superserver to respond to BOOTP requests by adding (or uncommenting) the following line in `/etc/inetd.conf`:

```bash
bootps dgram udp wait root /usr/sbin/bootpd bootpd
```
Then, reset **inetd** by sending it the `-HUP` signal (using **kill**).  

Next, **bootpd**’s configuration file, `/etc/bootptab`, needs to be written. The project’s **bootptab** file contained this entry, with tags separated by colons:

```
ebsa285:  
 :ht=ether:  
 :ha=0050da26b054:  
 :rp=/ebsa285fs:  
 :hd=/tftpboot:  
 :bf=zImage:  
 :sm=255.255.255.0:  
 :hn:  
```

The first line, **ebsa285**, is the **hostname** for this entry—that is, whatever hardware device “belongs” to this entry will be called “ebsa285”. **ht=ether** defines the host’s hardware type (Ethernet). **ha=0050da26b054** specifies the hardware address of the device that “belongs” to this entry (the hardware address of the 3Com NIC on the CiNIC; see Section 2.3). **rp=/ebsa285fs** specifies the root filesystem path (the root filesystem to be mounted via NFS later; see Section 4.4 and Section 3.3). **hd=/tftpboot** specifies the boot file directory (the directory the kernel(s) is/are stored in; see Section 4.3.3). **bf=zImage** specifies the boot file (the kernel filename; see Section 4.3.3). **sm=255.255.255.0** specifies the network subnet mask (a Class C network). **hn** is a boolean that, when included, will send the hostname (**ebsa285**) to the client (the CiNIC).

Notice that there is no IP entry. It is possible to specify the IP address in **bootptab** using the **ip** tag, like so: **ip=192.168.1.100**. However, if there is no **IP** tag, **bootpd** will look up the specified hostname (**ebsa285**) using

---

2See man **kill** on a LINUX system.
3See man **bootptab** on a LINUX system.
gethostbyname()⁴. This is convenient because it allows the IP address of the entry to be specified by the server’s /etc/hosts file, or by the network’s DNS server.

4.3.3 Configuring the TFTP Server

Configuring tftpd, the TFTP daemon, is straightforward. First, ensure that tftpd is installed on the server (preferably the same server as bootpd and the ARMLINUX root filesystem). Then, add/uncomment the following line in /etc/inetd.conf:

```
tftp dgram udp wait nobody /usr/sbin/tftpd tftpd
```

and send inetd the -HUP signal.

Because the TFTP protocol doesn’t include authentication, tftpd will only serve files from one directory, typically /tftpboot on the server. Note that /tftpboot is the directory specified by the hd tag in /etc/bootptab (see Section 4.3.2). The kernel specified by the bf tag in /etc/bootptab (as compiled in Section 3.2) must be placed in the /tftpboot directory.

4.4 Serving the Root Filesystem: NFS

The ARMLINUX root filesystem (see Section 3.3) is served to the CiNIC via NFS (Network File System, common on most UNIX systems) over the network. (Note that this is necessary regardless of how the BIOS obtains the kernel.) The directory containing the root filesystem is specified by the rp tag in /etc/bootptab (see Section 4.3.2), which for the project was /ebsa285fs on the server. For more information regarding NFS, refer to [3], and the manpages, man exports and man nfsd, on a LINUX system.

⁴See man gethostbyname on a LINUX system.
First, the server’s Linux kernel must have been compiled with NFS support; if it was not, then either load the NFS filesystem module, or recompile the server’s kernel with NFS support included.

Next, ensure that the NFS daemon, nfsd, is being started by the bootscripts in /etc/rc.d. (Note that while it’s possible to configure the inetd super-server to run nfsd, in practice nfsd is usually started and runs separately from inetd.) Unfortunately, every distribution’s /etc/rc.d is somewhat different, so this may take a little detective work.

With nfsd running on the server, the directory containing the ARMLinux root filesystem needs to be exported—that is, made available for mounting over the network. To export the directory /ebsa285fs, add the following line to /etc/exports:

```
/ebsa285fs (rw,no_root_squash)
```

Finally, send the –HUP signal to nfsd; this will cause nfsd to reload its configuration files, including /etc/exports.

4.5 Booting the System

At this point, the CiNIC is ready to boot. But first, a quick review of the entire system before powering up is in order. Then, the boot process will be examined in detail.

4.5.1 CiNIC Before Power Up

The CiNIC, consisting of the EBSA-285 (with the ARMLinux BIOS in flash image 5, and the flash image selector switch set to 5) and the 3Com NIC, plugged into the 21554 Evaluation Board, is plugged into a host PC. The host PC (and, thus, the CiNIC) is powered off. The serial port on the EBSA-285 is connected
to the serial port of another (running) PC—the console PC—with a null-modem
cable; the console PC is running a serial communication program configured
(38400, 8n1, no flow control) to read and write the serial port to which the
EBSA-285 is connected.

The 3Com NIC on the CiNIC is connected to the local network. On the
local network is a machine—the server—that is running a BOOTP server, and
exporting the ARMLINUX root filesystem via NFS.

If the kernel is to be uploaded to the EBSA-285 via the serial connection,
then the ARMLINUX kernel executable resides on the console PC (to be sent
using the Y-modem protocol over the serial connection). If the kernel is to be
served over the network, then it resides on the server, which is running a TFTP
server.

4.5.2 Powering Up

Power up the host PC. This will also power up the CiNIC. The following ac-
tions will occur:

1. The EBSA-285 PBL will start the ARMLINUX BIOS.

2. The BIOS will run its memory test.

3. The BIOS will attempt its autoboot sequence. Strike the “s” key twice (on
the console PC) to abort the autoboot sequence.


5. Choose the desired method for fetching the kernel: serial or net, using
the Boot method command.

6. Choose the boot command.

7. The BIOS will fetch the kernel.
(a) If the serial method was chosen:

i. The BIOS will prompt for a Y-modem transfer of the kernel executable.

ii. Upload the kernel with the Y-modem protocol on the console machine.

(b) If the net method was chosen:

i. The BIOS will broadcast a BOOTP request.

ii. The BOOTP daemon on the server will respond to the BOOTP request.

iii. The BIOS now knows where to find the kernel.

iv. The BIOS will fetch the kernel executable from the server using the TFTP protocol.

8. The BIOS will boot the kernel.

9. The kernel will decompress itself and begin executing.

10. The kernel will begin its autoconfiguration by broadcasting a BOOTP request.

11. The BOOTP daemon on the server will respond to the BOOTP request.

12. The kernel now knows its hostname, IP address, and location of its root filesystem.

13. The kernel will mount its root filesystem via NFS. (If this fails, the kernel will panic.)

14. The kernel starts its first process, /sbin/init.

15. init reads /etc/inittab.
16. `init` enters runlevel 3 (the default).

17. `init` executes `/etc/rc.d/startup`.
   
   (a) Set the path.
   
   (b) Mount all filesystems specified in `/etc/fstab`.

18. `init` executes `/etc/rc.d/multi`.
   
   (a) Start the `inetd` superserver
   
   (b) Flush `/var/run/utmp`.

19. `init` starts `/sbin/getty` on the console (which is visible on the console PC, over the serial port).

The CiNIC is now up and running. A login prompt should be displayed on the console PC.
Chapter 5

Logging In

Once the CiNIC is up and running, it can take logins from its console (over the serial port), and over the network via telnet. The /etc/passwd file contains two users by default: root (the superuser) and luser. Neither user was originally assigned a password.

5.1 Serial Console

The console, displayed over the serial port (/dev/ttyS0), is running the getty program by default. getty is responsible for opening a terminal connection, and initiating a login. When running, getty displays the “login:” prompt. Once the user has entered a username, getty will start the login program. login is responsible for prompting for the user’s password by printing the “Password:” prompt and receiving the user’s password. login then compares the login name (from getty) and password to those stored in the /etc/passwd file. If the name and password match an entry in the /etc/passwd file, login will start the user’s shell as specified by the /etc/passwd file (typically, /bin/sh).
Once logged in, the user is free to interact with the CiNIC system, just like any other LINUX system.

5.2 Telnet

The CiNIC will also accept telnet connections over the network. When the user telnets to the CiNIC (for the project, the hostname of the CiNIC was “ebsa285”), the inetd superserver will start the telnet daemon, /usr/sbin/telnetd. telnetd will then start getty on the telnet connection. getty will then handle the telnet login process the same way it handles logins on the console.

Logging in on a telnet connection requires the device files, pty0 through ptyf, and ttyp0 through ttypf to be present in the /dev directory. These files are paired: i.e., pty0 is paired with ttyp0, pty1 is paired with ttyp1, and so on. Each telnet login requires one pair, so with pairs 0 through F, 16 telnet users can login simultaneously.
Chapter 6

Conclusions

Implementing an ARMLINUX system to run on the EBSA-285, as part of the CiNIC effort, has been an interesting and challenging experience.

6.1 Summary

The CiNIC project seeks, in short, to move the burden of processing network requests, and executing the network stack, from the CPU to a dedicated co-processor—an intelligent network card. There are many aspects to this undertaking:

- the CiNIC hardware itself
- a way to run software on the CiNIC hardware
- establishing communication between the host and the CiNIC over the PCI bridge
- writing device drivers
• developing services for the CiNIC: firewalling, network address translation, routing, data caching, QoS, etc.

• performance testing

• and so on.

This particular Senior Project was concerned primarily with the second aspect: providing a way to run software on the CiNIC hardware. Without a way to run software on the CiNIC, it is simply fancy but useless hardware. As such, providing a way to run software on the CiNIC—specifically, the EBSA-285—was important for the CiNIC effort.

By providing an entire Linux system on the EBSA-285, this Senior Project is successful in its aim: ARMLinux on the EBSA-285, the heart of the CiNIC hardware, provides a platform on which other software can be developed and executed. ARMLinux on the EBSA-285 provides a foundation for the rest of the CiNIC project to build upon.

### 6.2 Future Work

There are many improvements that can, and will, be made to this implementation of ARMLinux on the EBSA-285. These improvements include more software (a native ARM compiler, additional shells, etc.), more sophisticated startup scripts (such as a complete SystemV or BSD setup), and more hardware support. These improvements will be made incrementally as the CiNIC system evolves.

However, one improvement must be made to make CiNIC practical: a local hard drive. Another improvement lies at the heart of CiNIC’s future: providing network services.
6.2.1 SCSI/IDE Disk Support

As implemented in this Senior Project, the CiNIC is not “self-contained”—it requires the help of another machine, the server, on the network. While this is acceptable for development work, the CiNIC, to become practical, needs to be able to operate without any help from the outside. In short, it needs to be self-contained.

The server provides two things to the CiNIC: a kernel, and a root filesystem. On a typical Linux system, these are both stored on a local hard disk. Similarly, the CiNIC, to become fully self-contained, needs a local hard disk. If a kernel can be loaded from a local hard disk, and a root filesystem mounted from a local hard disk, then the need for a server running BOOTP, TFTP, and NFS will be eliminated.

In addition to making the CiNIC self-contained, a local hard disk will run much faster, and provide more room for storage (for data caches, etc.), than an NFS mount.

Both SCSI and IDE drive controllers were being investigated by the research group when this Senior Project was completed. The ARMLINUX BIOS supports a couple of different IDE controllers, and may support SCSI controllers in the future.

6.2.2 Network Services

Network services is the future of the CiNIC project, and will make for many other Senior Projects and Masters Theses. The CiNIC, being a full Linux system, can provide any service that can run on, or be written for, a Linux PC. Network services under investigation by other members of the research group at the time of this Senior Project’s completion included:

Network Data Caching the caching of data retrieved by the CiNIC from the
network. This will improve the speed of subsequent data retrievals and reduce network traffic.

**Firewalling and Security** the CiNIC can provide a firewall and other security measures to the host, or even to an entire LAN.

**Routing** with multiple NICs on the CiNIC, the ARMLINUX kernel can route network traffic between different networks (say, the Internet on /dev/eth0, and a local LAN on /dev/eth1).

**Network Address Translation** in conjunction with routing, the CiNIC can allow an entire LAN to share a single Internet connection (known as masquerading in LINUX) by translating the local IP addresses on the LAN to valid Internet IP addresses and back again.

**Quality of Service (QoS)** the management of different types of network traffic, based on application type and priority. QoS seeks to provide guaranteed network performance for certain types of traffic (i.e. streaming video, voice, etc.).

Many possibilities exist, with the CiNIC.
Chapter 7

Appendices

7.1 Source Listings

/etc/fstab

/dev/root / ext2 defaults 1 1
/proc /proc proc defaults 0 0

/etc/gettydefs

38400# B38400 CS8 # B38400 SANE CS8 #login: #38400

/etc/group

root::0:root
users::500:1user

/etc/inetd.conf

Uncomment the line for telnet:
telnet stream tcp nowait root /usr/sbin/telnetd telnetd

/etc/inittab

# EBSA 285 Linux inittab file, for sysvinit.
# Author: Eric Engstrom
# Copyright (C) 2000 Eric Engstrom.
# Written for 3Com/EBSA 285/Linux project.
# v0.1 5-29-2000

# Runlevel structure
# 0: halt
# 1: single user mode
# 2-5: multiuser (3 is default)
# 6: reboot

# Default runlevel
id:3:initdefault:

# System initialization before
# anything else happens
si::sysinit:/etc/rc.d/startup

# Scripts to run at a given runlevel
10:0:wait:/etc/rc.d/halt
11:1:wait:/etc/rc.d/single
13:2345:wait:/etc/rc.d/multi
16:6:wait:/etc/rc.d/reboot

# Serial Gettys (tinylogin)
g0:12345:respawn:/sbin/getty 38400 console
#g0:12345:respawn:/sbin/getty 38400 ttyS0
#g1:2345:respawn:/sbin/getty 38400 ttyS1

/etc/passwd

root::0:0:root:/root:/bin/sh
luser::500:500:/home:/bin/sh

/etc/rc.d/halt

    echo "Halt script."
    action "Unmounting filesystems" umount -a
    action "Halting system" halt

/etc/rc.d/multi

    # Start the inetd daemon
    action "Starting INET services" /usr/sbin/inetd
    # Create an empty 'utmp' file
    action "Clearing utmp file" cat /dev/null > /var/run/utmp

/etc/rc.d/reboot

    action "Unmounting filesystems." umount -a
    action "Rebooting." reboot -i
/etc/rc.d/single

This file was left blank.

/etc/rc.d/startup

    PATH=/bin:/sbin:/usr/bin:/usr/sbin
    export PATH
    # Mount all filesystems in fstab
    /bin/mount -av

7.2 Useful man pages

Many commands and configuration files are mentioned throughout this paper. Information on most of these is available in the LINUX online manuals. To access the manual page (manpage) for a command or file, type `man <command-name>` on a LINUX system. Useful manpages include the following:

- bootptab
- exports
- fstab
- gethostbyname
- getty
- halt
- inetd
- init
- inittab
- kill
- ldconfig
- login
- minicom
- mknod
- nfsd
- passwd
- ping
- reboot
- strip
- telnet
- telnetd
- tftpd
Bibliography


