A Project
entitled

Porting the NetBSD Operating System to the NCD HMX Thin Client

by

David Gajewski

As partial fulfillment of the requirements for the
Masters of Science Degree in Engineering

____
Advisor: Dr. Lawrence Miller

____
Graduate School

The University of Toledo
December 2002

An Abstract of

Porting the NetBSD Operating System to the NCD HMX Thin Client

David Gajewski

Submitted in partial fulfillment
of the requirements for the
Masters of Science Degree in Engineering

The University of Toledo
December 2002

This document describes the experience of porting the NetBSD Operating System to the NCD HMX Thin client. While the HMX deviates slightly from standard MIPS convention with regards to memory, the NetBSD kernel can still be configured to work as desired. A partial description of the architecture of the HMX is given; including information on the bus, the devices, and the layout of these things in memory-mapped I/O space. To purposefully make this project more difficult, information about how the HMX works was derived by reverse engineering and disassembling program code. Care was taken so that this work did not violate the DMCA.

Contents

Abstract ii

Contents iii

List of Tables iv

1 Background Information 1

1.1 NetBSD 2

1.2 HMX 3

1.2.1 BootROM 4

1.2.2 Prompt Capabilities 4

1.3 MIPS 5

1.3.1 Memory Map 6

1.3.2 Interrupt/Exception Handling 8

1.4 Additional Definitions 9

2 Project Goals 11

2.1 Reverse Engineering & Disassembling the HMX BootROM 11

2.2 Getting Custom Code to run on the HMX 13
2.3 A Booter 13
2.4 The NetBSD Kernel 14

3 Results: An In Depth View of the HMX 15

3.1 Types of Expansion Cards Supported 16
3.2 The Bus 18
3.3 The Hardware 20
3.3.1 Serial Ports 20
3.3.2 Parallel Port(s) 21
3.3.3 Keyboard 21
3.3.4 Sound 22
3.3.5 Video 22
3.3.6 Lance Ethernet 23
3.3.7 PCMCIA 24
3.3.8 NVRam 24
3.4 Prompt Abilities and Other Goodies 25

4 NetBSD on the HMX 29

5 Future Work 32

6 Concluding Remarks 34

List of Tables

3.1 Expansion Card List 17

3.2 Understood Bus Registers 18

3.3 Example for how Byte sized registers are arranged in memory 19

3.4 Example for how Halfword sized registers are arranged in memory . . 19

3.5 The Keyboard Device Registers 22

3.6 Known Sound Device Registers 22

3.7 Lance Registers 23

3.8 Undocumented Prompt Commands 26

3.9 BootROM provided Syscalls 26

3.10 Magic 40-byte structure in .text, entries are 32-bit 27

Chapter 1

Background Information

This project has taken shape because of the circumstances which have lead up to it. The author was first introduced to the HMX (and other products from NCD) years ago and learned to be proficient in their use while administering them. As I began to find interest in other CPU architectures and wished to experiment with them, it appeared that Linux treated them as second-class citizens (such as needing to rewrite entire drivers for common hardware and not having one's work in the main kernel source tree.[1]) Seeking out a new software base, NetBSD was soon discovered. On one web page at the NetBSD website, a suggestion was made that NetBSD could be made to run on the HMX. When the time came to decide on an approachable project, relevant knowledge and ability were primed for attempting such a port.

[1]For instance see how each platform has its own Lance ethernet driver and how such ports as to the Dreamcast, SGIs, VAXen, and HP PA-RISC machines are housed and maintained by various groups. Not that I don't think GNU/Linux is a killer Operating System — in fact, this report, along with all code and compiling, was achieved on a PlayStation2 running Linux!

1.1 NetBSD

NetBSD takes pride in being the most portable operating system in the world today. It currently supports 53 platforms across 11 distinct CPU architectures. A UNIX-type operating system, NetBSD is derived from the BSD research operating system written at Berkeley. The operating system, from the kernel to the userland programs, is given away freely under the BSD software license (and a few needed programs under the GPL). These licenses are Open Source licenses which permit anyone the ability to read and modify the source code as well as distribute the changes. This open-ness continues the BSD heritage in making NetBSD a wonderful research operating system.

By following an ideal of doing things more correctly and abstractly than other Operating Systems, NetBSD is able to work around the ideosynchronies of differing hardware conventions. By being portable, this means that one is able to make localized changes to the code to grant the resulting software the ability to run on a machine it has not seen before. In being a unified codebase, any improvement made to portions of the source code can benefit some or all of the various ports. Not only is the OS designed to run on a large number of target machines, it is designed in such a way that can be developed on a large number of host machines as well.[2] Using a program called `build.sh`, an entire cross-compiling environment can be setup. With this environment, one is able to build a kernel, or even compile and layout the entire Operating System!

[2]Did I mention that I did the port on a PlayStation2 running Linux?

1.2 HMX

Network Computing Devices, Inc. (NCD) make thin clients for both UNIX and Windows environments. Thin clients can be thought of as a intermediary step between having a single machine with two monitors and having two separate machines. A thin client is a device that can support input from a keyboard and mouse with output to a monitor, yet has no hard drive by which to load user applications or store user data. It is a dumb device which coordinates all of the needed input and output with a remote machine across a network; either via a UNIX X session or a Windows Terminal Server session. The usefulness of a thin client is that it can be very cheap to add another user station to an installation.

NCD has made a number of different thin clients over the years. Currently they market the ThinSTAR and NCx00, but it was their previous group of devices, which ran software known as NCDWare, which are NCD's claim to fame. NCDWare is highly configurable, and in part contains: a windowing environment, remote configuration and diagnostics via SNMP or telnet-able ports, OpenGL, and network audio. PowerPC based Explora's and ExploraPro's ran NCDWare as well as MIPS based 15r's and HMX's. The HMX in particular is interesting as it is the only device which gives the illusion of a complete computer, has the best audio, and is upgrade-able via interchangeable cards. The HMX has 8 MB of main memory (configurable up to 136 MB) with 2 MB of on-board video memory. There are updated versions of the HMX, the HMXpro and HMXpro24, which may have 8 MB of video ram.

1.2.1 BootROM

The BootROM is the software responsible for initializing all of the hardware and downloading and running NCDWare. In PC terms it would be called the Bios. Just like a PC Bios, one is able to configure parameters. Yet it also has other abilities which will be examined shortly. The BootROM is a relatively small program, weighing in at less than 256 KB. It is important as it provides us with the initial environment by which to investigate the HMX. It is physically located on the expansion card, and updating the firmware revision is accomplished by replacing two PROMs. NCD calls these chips and their functionality the “Boot Monitor.”

1.2.2 Prompt Capabilities

The prompt on all of the NCD devices offer the user an amazing array of diagnostic tools by which we may explore the inner workings of the machine. A classic Chevron (the ’>’ symbol) may be obtained by canceling the download of the NCDWare software. Here one can enter any of the possible commands seen below (as seen in a version 2.7.2 BootROM). A few of these commands have proven vital in the understanding the operation of, and experimentation with, the HMX and all of its devices. Long term settings are accessed via navigatable menus available by entering ’SE’. Here is what a user sees when entering a ’?’ at the prompt:

```
BD[file] boot via MOP
BL[file] boot locally
BN[file][local-IP host-IP][gateway-IP][subnet-mask] boot via nfs
BT[file][local-IP host-IP][gateway-IP][subnet-mask] boot via tftp
```

```
DA display addresses

DM[adr][len] display memory

DR[d f s] display registers

DS display booting statistics

EX extended tests

KM keyboard mapper

KS keyboard statistics

NV NVRAM utility

PI[timeout][local-IP host-IP][gateway-IP][subnet-mask] ping host

RS reset system

SE NVRAM setup

SM show memory configuration

UD upload to host via MOP

UP[file][local-IP host-IP][gateway-IP][subnet-mask] upload via tftp

ZK zero keyboard statistics

ZS zero boot statistics
```

1.3 MIPS

MIPS is a 32-bit RISC CPU. It was born out of a research project at Stanford University where people wondered if a processor could be made more efficient while stripping down its abilities to a near bear minimum. A reduced instruction set computer (RISC) typically has a lot of registers and can only access memory with simple loads and stores. RISC chips such as the MIPS, PowerPC, SPARC, and Alpha were able to use hardware optimizations such as pipelining long before their complex instruction set (CISC) counterparts (such as the x86 and Motorola 68k lines); new CPU

architectures tend to be a hybrid of the two.

MIPS processors are produced by various manufacturers and are often found in embedded products (such as the HMX). One is able to find MIPS processors within SGI machines, HP LaserJets, Cisco routers, and the PlayStation2, PlayStation, and Nintendo64 game consoles. Since 1991, MIPS processors have been 64-bit as well as 32-bit, and the various manufacturers have expanded the base instruction sets to facilitate such things as multimedia. The reader is invited to read the empowering book, *See MIPS Run.* [1]

1.3.1 Memory Map

To be able to understand how one can explore the HMX, one needs to know how memory is laid out on a MIPS system. With an address size of 32-bits, a machine can address more than 4 billion memory addresses (which could be 4 Gigabytes worth of memory), yet MIPS defines that spans of these addresses are reserved for hardware and mirrored data. Even under the most restricted view, the chip is assured that 256 MB of memory may exist. Base memory occurs from address 0x0000_0000 to 0x0FFF_FFFF. The various devices[3] and the bootrom will be found from 0x1000_0000 to 0x1FFF_FFFF (with the bootrom beginning at 0x1FC0_0000). Consider the range from 0x2000_0000 to 0x7FFF_FFFF to be undefined. The addresses where the high bit is set is only accessible from an operating system kernel.[4] KSEG0, the range

[3] There are two ways that different types of machines use to communicate with devices: specialized instructions and memory mapped I/O. x86 based PC's use special instructions, almost everything else uses memory mapped I/O. Under memory mapped I/O, device registers and data are addressed at memory-like locations.

[4] More specifically, any program running in kernel mode

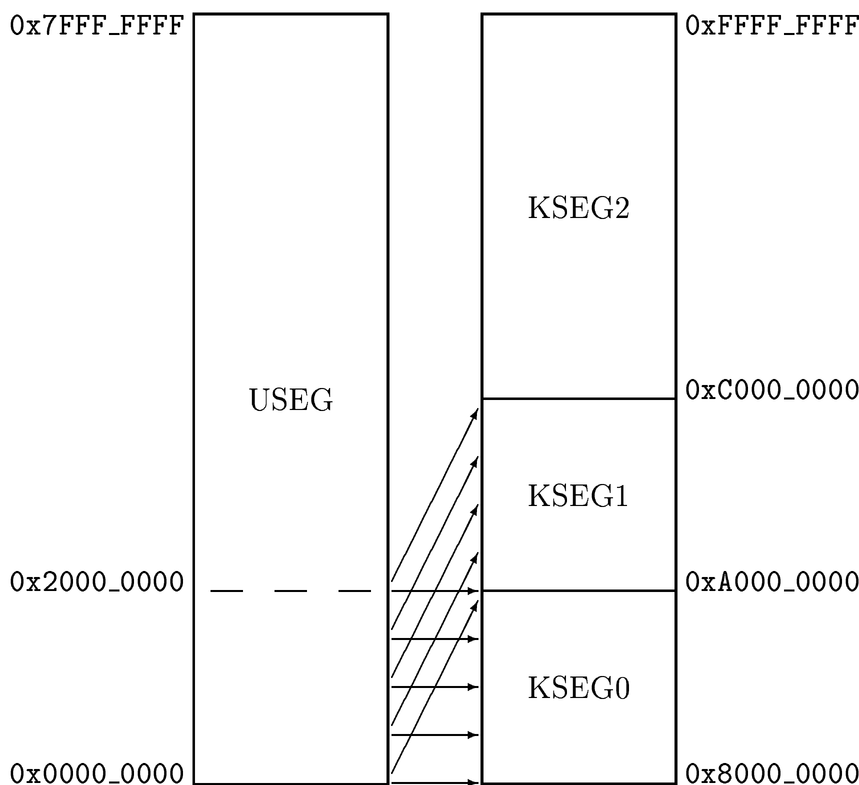

Figure 1-1: MIPS Address Space

from 0x8000_0000 to 0x9FFF_FFFF, is mapped to the same area as user memory and I/O by removing the highest bit. KSEG1, which occurs from 0xA000_0000 to 0xBFFF_FFFF, is mapped to the same place in memory that KSEG0 is, except all access to this region is uncached.

Since the kernel has access to the addresses with the high bit set, it is fitting to call the space which is always user addressable USEG. Whenever we access data from KSEG0 or KSEG1, we get it directly. However, if we access it via USEG or KSEG2 (which is the remainder of kernel space), the address is translated via a MMU (memory mapping unit). A MMU is a highly important piece of hardware which is required by all modern operating systems. It gives us the ability to use such things as virtual memory and have the option of executing the same program multiple times

concurrently. These multiple address segments may seem somewhat complicated, but they turn out to be quite a useful setup for an operating system. We will need them when describing some of the hardware, and the issues involved with porting NetBSD.

1.3.2 Interrupt/Exception Handling

Like all CPUs, MIPS processors are set up to take care of unexpected and external events. Examples include such things as data arriving at an I/O device or a user program executing an illegal instruction. When an exception occurs, a number of things happen. The address of the last completed instruction is stored in a special register, another special register holds a number corresponding to the type of exception which occurred (when applicable), and the CPU starts executing instructions at a special location in memory. When this exception handler has completed, the program counter is restored — and whatever task was currently running has no knowledge that anything even occurred (except if it did a kernel system call).

To be able to properly drive the hardware, we will need to know how interrupt signals propagate to the CPU. When a device needs attention, it serves an interrupt to an interrupt controller. This could be a system bus, another device, or the processor itself. It may occur that an interrupt is masked (and therefor not propagated) on its way to the processor. It is useful to mask off interrupt signals which we do not know how to service. When an interrupt is serviced, meaning that the kernel attends to the reason that the interrupt is raised, then that specific interrupt is no longer sent. For the MIPS processors in particular, the CPU can be wired to 6 hardware interrupt

lines (and 2 software interrupts), and can mask each of them.

1.4 Additional Definitions

This section exists to define additional concepts and to solidify certain terms used up to this point. The purpose it serves is to help bring the reader up to a certain level of understanding to be able to comprehend the rest of this paper. While this paper is designed for a more technical audience, interested parties should still be able to approach and understand its contents.

Assembly Language A low level programming language. Controls every aspect of the CPU directly.

Cache Stores recently used data and instructions. It makes memory access efficient by not sending data to/from memory unless forced to.

Disassemble Take apart a program by converting it to its assembly language equivalent.

Endian, Big/Little Describes how sequential bytes of a CPU register are stored in memory. This only matters when data is read back at a length different than it was stored.

NVRam Non-Volatile RAM. Memory which does not loose its contents when the machine is powered down.

Reverse Engineer Take knowledge of how one thing works, and make something

else have the same effect. How the new object achieves this will most likely be different than how the original does.

Chapter 2

Project Goals

2.1 Reverse Engineering & Disassembling the HMX BootROM

The BootROM is the gateway into the HMX. One is only capable of doing to the HMX what the BootROM allows. The commands available at the prompt give us an amazing ability to navigate address space in search of devices. One can also see items which the BootROM program leaves at fixed memory addresses. While this may be enough for the casual user, we would like to know the full range of detail we can learn from the BootROM. For instance, one can plainly see that the commands are each two letters in length. If one tries every two letter pair from 'AA' to 'ZZ' at the prompt, could new, undocumented commands be found? Would they be useful? The answer to both questions is "Yes."

Furthermore, there is knowledge within the BootROM program itself that would

be useful to us. The BootROM knows how to find, configure, and communicate with the hardware devices. To be able to do the same, it would be beneficial to see how it does these things. Since NCD probably doesn't want to disclose how the machine works,[1] it would have to be derived from the code of the BootROM. Disassembling the BootROM will not give any form of annotated source code, and determining the intent of the different subroutines can become tedious. Yet there is a quickly earned reward for looking through the code: starting at the special addresses holding the exception handlers, one should encounter the device interrupt handlers just a few subroutines deep.

Notice that we are using the BootROM, and not the NCDWare software, to understand the HMX hardware. The reason is mostly a legal one (the DMCA). The NCDWare software most likely comes with a *"Do not disassemble or reverse engineer this software."* type of clause.[2] I believe that one is legally allowed to disassemble/reverse engineer the BootROM code as a means of providing inter-operable software.[3] But I am no lawyer. If this turns out to not be the case, the work done here cannot be disseminated back into the general public. The other reason that the BootROM code was chosen was that it is smaller in size, has less configuration options, and is the program that sets up the items at certain fixed memory addresses.

[1]They might have given out information, or they might have delivered a "No" which might have dissuaded me from starting this project. Besides, the extra leg-work makes the project seem more like an adventure.

[2]I was unable to locate the install media and cannot be sure. However, the individual who installed this software on the server copied it directly from the CD, and did not use their installer...

[3]See Linux on the XBox for a contemporary example

2.2 Getting Custom Code to run on the HMX

Before even considering if a full operating system kernel can be run, we must be certain that custom code can be made to run on the HMX. The BootROM is designed to boot NCDWare from across the network or from a local flash card. One cannot just point the HMX to another file to boot from, as the BootROM performs checks on the downloaded binary before it is run. With the **Xncdhmx** and **Xncdhmx.bl** files to inspect, which are booted from across the network and from a flash card respectively, the required program layout can be determined. Simple tests can be done by changing portions of these files and seeing what happens.

2.3 A Booter

It is inconceivable that the kernel image will follow the strict requirements of the BootROM, so an intermediate kernel booter will be required. This booter can be forged specifically to the BootROMs specification of a bootable image. It would be the job of the booter to actually load the kernel and boot to it, passing on any important information as arguments. This also can be used to work around any limitations that the BootROM may have in its program loading code.

NetBSD can already be of some use at this stage: it comes with a stand-alone library which is used to create booters. It has the ability to boot from across the network, a hard drive, CD-ROM, etc. and can read compressed programs and file systems. The booter code used by other NetBSD ports can be easily mimicked or copied.

2.4 The NetBSD Kernel

The final piece of the puzzle will be to have a NetBSD kernel specific to the HMX. With many MIPS based NetBSD ports, much of the work needed can be copied from these other sources. The kernel needs to be aware of things such as the interrupt tree on the HMX, and any other issues specific to the HMX. It has been written that having a simple driver for a serial port is very beneficial for initial debugging of the kernel. [2] Thus an early goal should be finding and controlling the serial port. Since the HMX has no local storage, the files for the operating system must reside on a remote machine. The NetBSD website has a step by step guide on how to set up NFS mounting the file system at http://www.netbsd.org/Documentation/network/netboot/, but to do this support for the ethernet port is needed. These two things give the bear minimum driver support needed to drive a kernel to support an operating system.[4]

[4]Well there is a timer that is needed to interrupt the system every $1/100^{th}$ of a second, but we get such a timer with every MIPS processor.

Chapter 3

Results: An In Depth View of the HMX

The results found here are a union of all data retrieved from various NCD manuals and help files found on *www.ncd.com*, identified chips on the HMX mainboard and add-on card, disassembling and reverse engineering information in the BootROM, and a little luck. They mainly represent the specific HMX and add-on card available to the author. In certain regards the following information is highly lacking; however it is just enough to create a useful operating system.

The processor in the HMX is the R4600 (v2.0) produced by IDT (also known as QED). It seems to run at 82.2 MHz, although speeds of 100 MHz and up were standard. It has split cache with both the instruction and data cache being 16 KB in size and 2-way set associative. It also comes with a built-in floating point unit (FPU). Instructions and data are in big endian format.

The HMX differs (slightly) from standard MIPS convention in that it is video

memory which occurs at the base address of 0x0000_0000. This means that the kernel and its data structures can be placed so that they are visible on a monitor.[1] With only 2 MB of video memory on an HMX, there will not be much room for a kernel and programs, especially since "average" size kernels weigh in at nearly a megabyte. Normal ram begins at address 0x2000_0000, and can end at from 0x2080_0000 (8 MB) to 0x2880_0000 (136 MB = on board 8 MB + 128 MB in expansion memory). A file on the NCD website describes what type of memory can be used in the HMX, and what configurations are possible. [6]

Interrupts on the HMX are set up quite simply. All of the devices send their interrupt to the bus, which sends a lone interrupt signal to the processor. This would be 0x04 in the interrupt portion of the MIPS status register, also known as hardware interrupt 0. There is a standard MIPS timer attached at hardware interrupt 5 (0x80 on the status register). Both the bus and the processor are able to mask out individual interrupt signals.

3.1 Types of Expansion Cards Supported

The HMX (and the the other MIPS based NCD thin clients) have a slot for a mandatory expansion card. This expansion card holds the BootROM, some sort of ethernet, and possibly more. An expansion card is required, since that is where the BootROM is located. The NVRam may also be in this area as the MAC address of the card should be unique to the transceiver. The expansion card is held in place by

[1] For instance I can literally "see" ethernet frames as they come and go, and can watch what seems to be the page-freeing daemon scan a list of pages.

Type	*Contains*
Type 1	DECnet Ethernet
Type 2	Token Ring Ethernet, PCMCIA expansion
Type 3	Ethernet, ESPO RS-232 port, PCMCIA expansion
Type 4	N/A
Type 5	LANCE Ethernet, Parallel port, PCMCIA expansion

Table 3.1: Expansion Card List

two screws and is easily removed. One can upgrade their machine by switching cards and/or replacing the BootROM chips.

Internally, the BootROM numbers the different types of cards from 1 to 5 (on a version 2.7.2 chip). Type number 4 is neither referenced nor set and may be an expansion card that never made it beyond a prototype stage. Ethernet connection options have been seen to be a combination of 10 Base-T, BNC, and AUI. This information comes from the earliest of the HMX BootROMs (version 2.7.2), other expansion cards may have been produced.

Every board type except the first has a 12 volt PCMCIA controller. The only cards which will physically fit in the controller are those which would not protrude outside of the slot. This means that such cards as wireless ethernet adaptors cannot be used because of the extra length that the antenna contributes. However, later versions of the BootROM are able to boot across a network using a Proxim RangeLAN wireless ethernet card. Whether this means that there is another board type, that the card passes the physical requirements, or that only members of the Explora line (which have no PCMCIA card restriction) can do this, is unknown.

Address	*Size*	*Purpose*
0xB000_0029	Byte	Lance Ethernet High Byte (see Section 3.3.6)
0xB000_002F	Byte	VGA output control (see Section 3.3.5)
0xB000_0030	Byte	8-bit Interrupt Cause List
0xB000_0031	Byte	8-bit Interrupt Mask
0xB000_0032	Byte	Spurious Interrupt Silencer ???
0xB000_0037	Byte	Latch Register

Table 3.2: Understood Bus Registers

3.2 The Bus

The bus on the HMX is little endian and the size of the device and bus registers are limited to being 8-bit and 16-bit in size (we will follow the MIPS convention and call these sizes byte and halfword). There are three major areas on the bus: the bus registers, the keyboard/parallel/video/sound area, and everything else. Many of these areas are repeated throughout the bus address space. Although the base addresses for these items occur from 0x1000_0000 to 0x1FC0_0000, the KSEG1 equivalents will be used, as uncached access is the proper way to communicate with devices.

The bus registers occur from 0xB000_0000 to 0xB000_003F, and its 64 byte contents are mirrored all the way through 0xB7FF_FFFF. These registers give information about the bus and ways to control it. The latch register controls whether certain bus registers are read-only or read/write. 0x96 and 0x69 can be sent to the latch register to achieve the respective effect. Each of the devices on the bus capable of issuing an interrupt sends it to one of the 8 pins in the interrupt cause list. If the cause list, when logically AND'ed with the interrupt mask, is non-zero, a single interrupt is sent to the processor at MIPS hardware interrupt 0 (or 0x04 on the interrupt portion of the MIPS status register). It appears that one can silence a spurious interrupt

Address	+0	+1	+2	+3	+4	+5	+6	+7
base+0x00	XX	reg0	XX	XX	XX	XX	XX	XX
base+0x08	XX	reg1	XX	XX	XX	XX	XX	XX
base+0x10	XX	reg2	XX	XX	XX	XX	XX	XX
base+0x18	XX	reg3	XX	XX	XX	XX	XX	XX
base+0x20	XX	reg4	XX	XX	XX	XX	XX	XX
base+0x28	XX	reg5	XX	XX	XX	XX	XX	XX
base+0x30	XX	reg6	XX	XX	XX	XX	XX	XX
base+0x38	XX	reg7	XX	XX	XX	XX	XX	XX

Table 3.3: Example for how Byte sized registers are arranged in memory

Address	+0	+1	+2	+3	+4	+5	+6	+7
base+0x00	'CD'	'AB'	XX	XX	XX	XX	XX	XX
base+0x08	'GH'	'EF'	XX	XX	XX	XX	XX	XX
base+0x10	'KL'	'IJ'	XX	XX	XX	XX	XX	XX
base+0x18	'OP'	'MN'	XX	XX	XX	XX	XX	XX
base+0x20	'ST'	'QR'	XX	XX	XX	XX	XX	XX
base+0x28	'WX'	'UV'	XX	XX	XX	XX	XX	XX
base+0x30	'01'	'YZ'	XX	XX	XX	XX	XX	XX
base+0x38	'45'	'23'	XX	XX	XX	XX	XX	XX

Table 3.4: Example for how Halfword sized registers are arranged in memory

(an interrupt the bus has but a device did not send) by sending its mask to the bus register at offset 0x32.

Normal devices live from 0xB800_0000 to 0xBFC0_0000 (which is the start of the BootROM), but their registers are laid out strangely. First of all, the device registers can only exist in the first two bytes following an address which is a multiple of 8. I.e., device registers can exist at the base address +0, +8, +16, +24, etc. Secondly, we have a little endian bus with a big endian chip. This means that the 16-bit data will appear byte-swapped, and that 8-bit data registers will occur at an address which is 1 more than a multiple of 8. In the example, consider the contents of the halfword sized registers to be "ABCD", "EFGH", etc.

Now comes the exception to the rule: the devices which live at 0xBB00_0000.

While other devices get their own address space, the video, audio, parallel, and keyboard device registers are jumbled together. Also, this area contains a mix of word and halfword sized data ports. Finally, it breaks convention with byte sized register arrangement by allowing addressable bytes at consecutive addresses (i.e. base +0 and +1, +8 and +9).

3.3 The Hardware

With the exception of what appears on the add-on card, the base hardware of an HMX remains constant. There is a PS/2 keyboard port, 2 serial ports (one of which is meant for a serial mouse), a parallel port, audio in and out, monitor connection, power and power pass-through, and a system fan. Here, all that is known about these devices (and those on the expansion cards) will be given. Interestingly absent from the on board device list is a system clock. NCD explicitly states in the NCDWare user manual that there is no system clock, and that the correct time will be retrieved from a time server.

3.3.1 Serial Ports

The serial ports on the main board are driven by a somewhat well known, and easily programmable DUART. The MC2681, which is a clone of the more well known SCN2681, controls the two ports through 16 byte wide registers. The base hardware address is 0xB800_0000, and it follows the byte register layout described in Section 3.2. Channel A is labeled “Mouse” and channel B “Auxiliary” on the HMX. The interrupt

for this controller appears as 0x20 in the bus interrupt register. It is also the means by which the NVRam is read. See Section 3.3.8 for more on that.

The serial port on the type 3 expansion card is an ESPO RS-232 port. There is no information on this controller nor how it appears under the HMX.

3.3.2 Parallel Port(s)

The parallel port lives in a halfword at 0xBB00_0000. After byte-swapping, we see that byte 2 is used for config/status, and byte 1 for data. The controller is unknown, and no known parallel port seems to behave similarly. The interrupt for this controller appears on the bus interrupt register as one of: 0x08, 0x04, 0x02, 0x01.

The type 5 expansion card has the same parallel port controller as is on the main HMX bus. It appears at the address location 0xBC0C_0000. The interrupt for this controller is 0x80 on the bus interrupt register. It may also serve another interrupt in the low 4 bits of the bus interrupt register (just as its main bus cousin).

3.3.3 Keyboard

The keyboard controller seems to be a standard controller for the PS/2 type keyboard. However, it does not come with the standard PS/2 mouse port. In the accompanying table, one of the first 2 registers may be incorrect. The keyboard controller interrupts at 0x10 on the bus interrupt register.

Address	*Size*	*Purpose*
0xBB00_0009	Byte	Data or config ???
0xBB00_0010	Byte	Data or config ???
0xBB00_0011	Byte	Status (bit field)

Table 3.5: The Keyboard Device Registers

3.3.4 Sound

The sound device is driven by an unknown controller. It is marketed as being 16-bit sound and of better quality than that of other NCD thin clients. It has audio out and in. The data input format for the sound data register is currently unknown. This device serves no interrupts. To test the audio on the HMX (and other NCD machines) type 'BE' at the prompt. 'BE' is an undocumented prompt command — learn more about it and other goodies in Section 3.4.

Address	*Size*	*Purpose*
0xBB00_0018	Byte	Sound Data
0xBB00_0019	Byte	Sound Config
0xBB00_0029	Byte	Volume

Table 3.6: Known Sound Device Registers

3.3.5 Video

The video display is powered by an AT&T PrecisionDAC. This RamDAC is labeled 21C505, which is a newer version of the 20C505, which is found in S3 and ATI Mach accelerated graphics cards[7]. The display output begins at 0x0001_0000 and continues as is needed by the video resolution. Controlling the RamDAC has not been investigated. It uses registers in the area of 0xBB00_0000. One thing noticed is that writing values to the bus register 0xB000_002F causes the video output to cease.

Writing to this bus register is most likely part of the process of changing the video display resolution.

3.3.6 Lance Ethernet

The ethernet controller on a type 5 expansion card is the Am79c960. This is a member of the widely used LANCE family, and more properly, the PCnet family. These groups of chips are very well known and are easy to program. One just writes the needed internal register number to the address port, and then the mapped data port can be read from or written to. This construction is somewhat different than other invocations in that our MAC address is not nearby. In fact, it is stored in NVRam. The Lance chip delivers its interrupt to 0x40 on the bus interrupt register.

Address	*Size*	*Purpose*
0xBC00_0DC0	Halfword	Data Port
0xBC00_0DC8	Halfword	Address Port
0xBC00_0DD0	Halfword	Reset
0xBC00_0DD8	Halfword	Bus Interface

Table 3.7: Lance Registers

The chip is built to work in either 32-bit or 16-bit mode. Recall that the NCD bus can only do 8- and 16-bit reads/writes, therefor the chip is in 16-bit mode. Normal ram is used to load/store configuration information and ethernet frames. This presents two issues. First, because of the little endian bus, the configuration data has to be read/written byte-swapped. Thankfully the ethernet frames are written a byte at a time. Secondly, 16-bit mode is also known as 24-bit mode in that all memory addresses used by the chip are 24-bits in size. This is what the bus register

0xB000_0029 is used for. It is the high byte that completes the 32-bit address. The reader may be delighted to know that the chip can be driven to store its ethernet frames in visible video memory.

3.3.7 PCMCIA

Each type of expansion card except the oldest has a 12 volt PCMCIA port (some NCD clients can handle 5 volts). The controller is unknown and the location of the controller's registers in address space seems to vary for each board. An NCD document states that 16-bit reads are done with an access time of 300 nS. [5] The interrupt for the controller is unknown; in fact their may not be an interrupt. The PCMCIA port was designed to hold a flash card, to store files so that the HMX would not need a network to boot from. Later versions of the BootROM were able to boot off of a Proxim RangeLAN card, but it is unknown if this was meant only for the Explora line (PPC based) of thin clients. Because of the layout of the PCMCIA slot, only short cards can fit.

3.3.8 NVRam

The NVRam on the HMX is 512 bytes in size and holds important configuration information. A user may browse this data by entering 'NV' at the prompt. To access the NVRam in software, one goes through the serial port controller. This controller has extra input and output lines which are connected to the NVRam controller. To write data to the output pins, one writes the mask for the bits to be set in a certain

register, and the mask for the bits to be cleared in another. There are 3 output pins, and one input pin connected to the NVRam controller. It may be some time before it is understood.

3.4 Prompt Abilities and Other Goodies

In learning about how the HMX works internally, there were a few fun surprises in how the BootROM and other related things operate. For instance, it was unexpected to see that a C switch statement could be converted into a jump table or a binary search tree by a circa 1995 compiler. One major mistake was noticed in the version 2.7.2 BootROM. After a program returns by calling exit() (see later), the code does not restore the TLB (MMU) mappings, instead it saves the TLB's contents to its own list! This is bad in that a kernel's TLB entries are used by a program which knows nothing about them.

Seeing that all of the prompt's commands are 2 characters long, it seems feasible that hidden, diagnostic commands can be easily found by checking "AA" through "ZZ". Both "CB" and "CW" were instrumental in understanding how the HMX works by altering memory and hand configuring devices (also using "DM", or display memory). By far my favorite is the boing effect "BE". This command is found on all of the NCD thin clients, although some just beep. It also takes a volume parameter! "BE 00" is no volume and "BE FF" is at the highest volume.

For more fun, the "CW" command can be used to mirror the display across the serial port! By entering 0001 into 0x4000_1508 (yes this is being mapped by the

Command	*My Naming*	*What it does*
BE	Boing Effect	Plays sound (takes a hex parameter for volume)
CB	Change Byte	Changes consecutive bytes
CK	Reboot	Reboots, may be different than "RS"
CW	Change Halfword	Works like "CB" but on halfwords
DF	Display Font	Displays a representative sentence
MI	Monitor ID	Displays information on monitor id pins
TL	TLB List	Displays the current wired TLB entries

Table 3.8: Undocumented Prompt Commands

MMU) on at least a version 2.7.2 BootROM, input and output can be accepted from the serial port labeled "Auxiliary" at 9600 baud. It cannot navigate the graphical menus when "SE" is entered.

Function	`a0`	`a1`	Returns
putc(`a1`)	1	a character	—
putc(getc())	2	—	getc()
TFTP Boot	3	string pointer	—
Ready to send char ???	4	—	0 or 1
exit()	8	—	—

Table 3.9: BootROM provided Syscalls

The BootROM is setup to handle 5 MIPS system calls. Syscalls are a standard way for user programs to communicate with the kernel. The `syscall` assembly instruction causes an exception, which hands control to the kernel. The number of the system call is placed in `a0` and any optional argument in `a1` before calling `syscall` in an assembly program. Some of these syscalls proved invaluable in the early stages of programming the HMX. The argument for syscall number 3 takes a space or newline terminated string. All of these also work with the serial port display trick mentioned above.

To be able to write a program which will boot on the HMX one needs a hand-

crafted ELF file. NCDWare software contains the ELF sections: .text, .rdata, .data, .sdata, and .bss. An loadable executable may have more. There are minimum sizes for each of these sections, but any program which is not too simple can achieve them. One also needs a magic 40 byte header at the beginning of the .text section, just after a branch and nop. If one satisfies these simple requirements, any custom code can be made to run. The BootROM code will not allow loading and booting programs from KSEG0, thus a booting program is required.

Value	*Notes*
0	
Magic	one byte magic number
"Xncd"	
"HMX"	Type of the system
CRC-1	
CRC-2	
function Address	Location of a local CRC function
CRC-3	
0	
Magic	one byte magic number

Table 3.10: Magic 40-byte structure in .text, entries are 32-bit

The 40 byte structure contains a string for the system type, CRC (cyclic redundancy check) values, and some extra magic. The entries are 32-bit in size. The system type string for the HMX is "XncdHMX", the other types of NCD thin clients have other strings. There are 3 entries of CRC data (probably for 3 ELF sections), and the address of a function which calculates the CRC on the data. That function is stored in the downloaded code, and its results are not verified by the BootROM. Set the CRC entries to 0, and create a dummy CRC function which always returns 0. This ensures that there are no steps that have to be taken after any program is compiled, making the image already suitable for booting on the HMX. The "magic" entries

seem to be bit-fields, and were not investigated. Here is what NCD themselves have to say about the structure [4]:

> "The boot monitor does CRC checking mainly to catch instances where users might have attempted to "patch" an NCD server image and therefore possibly damage the overall quality of the server." ...
>
> "When the NCD server image is built at the factory, a CRC checksum is calculated and then inserted in a data structure in the first "section" of the server image."

Chapter 4

NetBSD on the HMX

Here's what you've all been waiting for — the **dmesg** output from the HMX:

```
Copyright (c) 1996, 1997, 1998, 1999, 2000, 2001, 2002
    The NetBSD Foundation, Inc.  All rights reserved.
Copyright (c) 1982, 1986, 1989, 1991, 1993
    The Regents of the University of California.  All rights reserved.

NetBSD 1.6I (GENERIC) #94: Sat Nov 30 16:49:58 EST 2002
    dgajews@ps2dev:/home/dgajews/netbsd/1.6F/src/sys/arch/ncdmips/compile/GENERIC
1108 KB memory, 788 KB free, 108 KB in 27 buffers
mainbus0 (root)
le0 at mainbus0: address 00:00:a7:15:6a:dd
le0: 4 receive buffers, 1 transmit buffers
cpu0 at mainbus0: QED R4600 Orion CPU (0x2020) Rev. 2.0 with built-in FPU Rev. 2.0
cpu0: 16KB/32B 2-way set-associative L1 Instruction cache, 48 TLB entries
cpu0: 16KB/32B 2-way set-associative write-back L1 Data cache
scn0 at mainbus0: scn2681
```

```
root device: le0
dump device:
file system (default generic):
root on le0
nfs_boot: trying DHCP/BOOTP
nfs_boot: DHCP next-server: 10.0.0.4
nfs_boot: my_addr=10.0.0.5
nfs_boot: my_mask=255.0.0.0
root on 10.0.0.4:/home/netbsd
root file system type: nfs
init path (default /sbin/init):
init: trying /sbin/init
Enter pathname for shell or RETURN for /bin/sh:
```

It stops here as the current tty driver does not yet serve the serial port interrupt correctly. Recall that only a simple console driver is needed to debug the kernel. For testing purposes, the kernel is set up to boot into single user mode — only spawning init and a shell. The limiting factor of this port currently is the lack of addressable memory. See how there is less than 800 KB to work with. If one removes **/dev/console**, then **init** attempts to mount a memory file system over top of **/dev**. As one might guess, it runs out of memory in this case. Yet **init** does not die; it perpetually tries to fork off a shell and enter single user mode. Because **init** can run, the port of the NetBSD *kernel* is complete. With just a bit more work, the whole operating system can run.

There are tricks which can be attempted to boot the rest of the operating system

at this point (to work around the < 800 KB memory issue). First, one could write a small, fake shell. It's purpose would be to first initialize swap over NFS, and then fork off a real shell. Assuming there is enough room left to do this. Secondly, one could add support for these non-kernel mappable memory locations into the NetBSD kernel. This is the desirable solution, and if the work is not done by this author, there are other NetBSD developers who might eventually add this support themselves.

Chapter 5

Future Work

It has been fun getting this far, but don't think that all of the work is done yet. With time, the following can be done to give the NetBSD port both incremental and monumental improvement:[1]

- Provide a means to access memory at and above 0x2000_0000. This would benefit any MIPS based NetBSD port which has memory in that area.

- Speak with Network Computing Devices, Inc. They might like the idea of seeing their hardware in use as a general purpose computing device. Would they provide hardware documentation — or would they sue?

- Find a means by which to determine memory size and location. (Currently the sizes are fixed at compile-time.)

- Obtain other HMX add-on cards and support the hardware found there.

[1] Some things will be attempted as soon as this paper is finished...

- Find and add support for these devices on the main HMX bus:
  - NVRAM (will be difficult)
  - Parallel port
  - Sound
  - System fan, and
  - PC Display if there is enough room in KSEG0 for video
- Find and reverse engineer the PCMCIA port on the type 5 add-on card. This could open the HMX up to becoming a real computer by having a flash card or micro drive attached to a PCMCIA card.
- Take what was learned and apply it to the other MIPS based NCD thin clients (which is why the port is called "ncdmips"). This includes the other older NCDWare machines and newer NCD lines: the NC200, NC400 and NC900, which hold an NEC 4300, NEC 4310, and QED RM5231 respectively.
- Provide support to all thin clients. This is mentioned as some vastly different manufacturers produce machines which are oddly alike.

Chapter 6

Concluding Remarks

The port of the full NetBSD operating system is but a few bug-fixes away. When the serial console tty driver is corrected, and possibly before the memory issue is addressed, the kernel should be able to boot into a multiuser environment with little hassle. NetBSD's code was extremely easy to mold for this port: both the booter and the kernel. Userland never had to be touched — the same binaries work across all platforms with the same processor. The HMX is as easy to understand and control as one could have hoped. The roadblocks thus far have been few and short-lived. The future looks bright for this port of the NetBSD operating system, some day it may even become an officially supported platform. This author is grateful to have been given a chance to work on a project that has been both fun and educational.

In conclusion: *"Of course it runs NetBSD."*

References

[1] Sweetman, Dominic. *See MIPS Run.* Morgan Kaufmann Publishers, San Francisco, 1999.

[2] Kesteloot, Lawrence. *Porting BSD UNIX to a New Platform.* 2001.

[3] NCDWare Manuals. http://www.ncd.com/support/docs_ncdware.html

[4] CRC Checking ftp://ftp.ncd.com/pub/ncd/Archive/NCD-Articles/NCD_X_Terminals/boot_monitor_CRC_checking

[5] Supported Flash Cards. ftp://ftp.ncd.com/pub/ncd/Archive/NCD-Articles/NCD_X_Terminals/supported_flash_cards

[6] HMX Memory Specs. ftp://ftp.ncd.com/pub/ncd/Archive/NCD-Articles/NCD_X_Terminals/Memory_specs/HMX_memory_specs

[7] Hardware-HOWTO. /usr/doc/HOWTO/Hardware-HOWTO under Linux.