Red Hat Linux 5.1

Alpha Installation Addendum

Red Hat Software, Inc.
Research Triangle Park, North Carolina

ISBN: 1-888172-10-X

Revision: AlphaAdd-5.1-Print-RHS (5/98)

Red Hat Software, Inc.
4201 Research Commons, Suite 100
79 T. W. Alexander Drive
P. O. Box 13588
Research Triangle Park, NC 27709
(919) 547-0012
redhat@redhat.com
http://www.redhat.com

Contents

Preface

The Red Hat Linux 5.1 Installation Guide contains most of the information necessary for the successful installation of Red Hat Linux on an Alpha-based system. However, there is additional information that just wouldn't fit in the Installation Guide; we have made that information available here.

You should start the process of installing Red Hat Linux/Alpha by first reading the Installation Guide. Make sure you take note of the Alpha-specific information contained there. Then, review this document to fill in any gaps left by the Installation Guide.

An Overview of This Manual

This document is geared to give you additional information to make your installation of Red Hat Linux/Alpha as straightforward as possible. Let's take a quick look at each chapter to help you get acclimated:

Chapter 1, *Information Specific to Red Hat Linux/Alpha* contains information necessary to install Red Hat Linux on an Alpha-based computer system.

Chapter 2, *Alpha Miniloader Howto* contains in-depth information concerning the MILO miniloader.

Chapter 3, *SRM Firmware Howto* describes how the SRM firmware may be used to boot Linux.

Chapter 4, *Alpha Howto* contains an overview of the Alpha architecture.

Please Note: While we realize that many people will find the information contained here very useful, please understand that much of it comes from sources outside of Red Hat Software, Inc. Therefore, it may contain descriptions of procedures that are not part of Red Hat Software's installation support for this product. For a description of Red Hat Software's installation support, please refer to the Red Hat Linux Installation Guide, or Red Hat Software's website at `http://www.redhat.com/`.

We Need Feedback!

If you spot any mistakes, or if you've thought of a way to make this manual better, we'd love to hear from you! Please send mail to:

`docs@redhat.com`

Be sure to mention the manual's identifier:

AlphaAdd-5.1-Print-RHS (5/98)

That way we'll know exactly which version of this manual you have. If you have a suggestion, try to be as specific as possible when describing it. If you've found an error, please include the section number and some of the surrounding text so we can find it easily. We may not be able to respond to every message sent to us, but you can be sure that we'll be reading them all!

1

Information specific to Red Hat Linux/Alpha

1.1 Introduction

This chapter describes the differences between Red Hat Linux/Alpha and Red Hat Linux/Intel installations. While it provides a good overview of these differences, you will find it easier to read the Red Hat Linux Installation Guide first. Concentrate on the chapters describing the actual installation. These chapters will refer you to the appropriate parts of this chapter at the appropriate time.

In addition, the Linux/Alpha homepage at `http://www.azstarnet.com/~axplinux/` contains a wealth of information for people considering Red Hat Linux/Alpha.

1.2 Supported Hardware

Red Hat Linux/Alpha supports a variety of hardware based on the Alpha processor and the PCI bus. The most recent list of hardware supported by Red Hat Linux/Alpha can be found at Red Hat Software's World Wide Web site at `http://www.redhat.com/hardware`.

All platforms except the Jensen include an NCR 810 SCSI controller, although BusLogic PCI SCSI controllers (other than the FlashPoint), the Adaptec AHA2940 SCSI controller, and Qlogic 1020 ISP controllers are also supported. The Jensen design uses an AHA1740 SCSI controller, which is supported.

NE2000, DE422, DE4x5, and DE500 (PCI) Ethernet cards are supported, as is the UDB's internal Ethernet hardware. Token ring support is also included in the kernels.

The X Window System should work on any machine with an S3 based video card (except the Jensen machines). S3 Virge cards, the Matrox Millenium, Millenium II, and the Mystique will work, as will most of the cards from Orchid and Number 9. Most Diamond Stealth cards are also supported. Digital TGA cards (based on the DC21030 chip) are supported in 8 bit mode, and an X server for Mach64 cards is also provided.

Please Note: There is an X server available specifically for Jensens with an S3 card. It is available from `ftp://ftp.azstarnet.com/pub/linux/axp/jensen`. Note that this site is mirrored by several of the bigger FTP sites, should you have trouble connecting.

1.3 Installation Overview

Installing Red Hat Linux on an Alpha system is slightly more complex than installing Red Hat Linux/Intel, mostly due to differences in machine architecture, and the variety of different models supported. In general, the main steps to a successful install are:

1. Create MILO, kernel, and ramdisk diskettes from images available on the Red Hat Linux/Alpha CD.
2. Use the MILO diskette to boot the appropriate Linux kernel.

3. Load and run the Red Hat Linux/Alpha installation program.
4. After the installation completes, install MILO on a small disk partition on your machine.

Please Note: While the majority of Alpha systems are supported by MILO, those that are not will need to boot the boot floppy (or CD-ROM) directly from the SRM (System Reference Manual) console . Information on doing this can be found in Chapter 3 on page 41. You should also consult your system documentation for the proper boot command syntax, but in general, the proper command would look like this:

```
boot dva0 -file vmlinux.gz -flags "load_ramdisk=1 prompt_ramdisk=1"
```

1.4 Choosing Diskette Images

The Red Hat Linux/Alpha installation process is diskette-based. The number and contents of the diskettes will vary according to the type of Alpha system you have. Therefore, before creating diskettes, it's necessary to pick the appropriate image files. The folowing table can help. Search the **Family/Model** column for your system. If you know your system's codename or alias, search the **Alias** column instead. The appropriate MILO image (if applicable) and boot image filenames are listed to the right.

Family/Model	**Alias**	**MILO Image** (In `milo/images`)	**Boot Image** (In `images`)
JENSEN		N/A	`jensen-s`
DECpc 150	JENSEN		
DEC 2000 Model 300	JENSEN		
DEC 2000 Model 500	CULZEAN		

Family/Model	Alias	MILO Image (In milo/images)	Boot Image (In images)
ALCOR		xlt-alcor	alcor alcor-s
AS 600	ALCOR		
AS 500 5/3xx	MAVERICK		
AS 500 5/5xx	BRET		
XL-300	XLT		
XL-366	XLT		
XL-433	XLT		
XL		xl	xl
XL-233	XL		
XL-266	XL		
AVANTI		avanti	xl avanti-s
AS 200 4/*	MUSTANG		
AS 205 4/*	LX3		
AS 250 4/*	M3		
AS 255 4/*	LX3+		
AS 300 4/*	MELMAC		
AS 400 4/*	AVANTI		
Personal Workstation	MIATA	miata	miata miata-s
PWS 433a, 433au			
PWS 500a, 500au			
PWS 600a, 600au			
SABLE			
AS 2100 4/xxx	SABLE	N/A	sable-s
AS 2000 4/xxx	DEMI-SABLE	N/A	sable-s
AS 2100 5/xxx	GAMMA-SABLE	N/A	sable-s-5
AS 2100 5/xxx	DEMI-GAMMA-SABLE	N/A	sable-s-5
MIKASA			
AS 1000 4/xxx	MIKASA	N/A	mikasa-s
AS 1000 5/xxx	MIKASA-PRIMO	N/A	mikasa-s-5

Family/Model	**Alias**	**MILO Image** (In milo/images)	**Boot Image** (In images)
NORITAKE			
AS 1000A 4/xxx	NORITAKE	N/A	noritake-s
AS 1000A 5/xxx	NORITAKE-PRIMO	N/A	noritake-s-5
AS 600A 5/xxx	ALCOR-PRIMO	N/A	noritake-s-5
AS 800 5/xxx	CORELLE	N/A	noritake-s-5
RAWHIDE		N/A	rawhide-s
AS 4000	WRANGLER/DURANGO		
AS 4100	DODGE		
AS 1200	TINCUP		
ALPHABOOK (laptop)	ALPHABOOK1	N/A	alphabook1-s
NONAME		udb-noname	noname
AXPpci33	NONAME		
UDB	MULTIA		
EB64+			
EB64+	EB64+	eb64p	eb64p
AlphaPC64	CABRIOLET	cabrio	cab
			cab-s
AlphaPCI-64			
PC164			
AlphaPC164	PC164	pc164	pc164
AlphaPC164-LX	LX164	lx164	lx164
AlphaPC164-SX	SX164	sx164	sx164
EB164	EB164	eb164	eb164
EB66+	EB66+	eb66p	eb66p
EB66	EB66	eb66	eb66
TSUNAMI			
DP264	GOLDRUSH	N/A	dp264-s
TAKARA	TAKARA	takara	takara

Please Note: Boot image filenames containing "-s" are intended for use with the SRM console, while those without a "-s" require a corresponding MILO file.

Creating a MILO Diskette If your system is supported by MILO, you will find the appropriate image in the `milo/images` directory on the Red Hat Linux/Alpha CD-ROM. Refer to the Red Hat Linux Installation Guide for instructions on writing the image file to a diskette. Make sure you label the diskette "MILO Diskette".

Please Note: If you have trouble with MILO, the first thing to try is a newer MILO image. The latest version of MILO is always available at
`ftp://ftp.digital.com/pub/DEC/Linux-Alpha/Miniloader/`.

Creating a Boot Diskette You will find the appropriate boot diskette image in the `images` directory on the Red Hat Linux/Alpha CD-ROM. Refer to the Red Hat Linux Installation Guide for instructions on writing the image file to a diskette. Make sure you label the diskette "Boot Diskette".

Creating a Ramdisk Diskette All Red Hat Linux/Alpha installations use the same ramdisk image. It is called `ramdisk.img`, and can be found in the `images` directory on the Red Hat Linux/Alpha CD-ROM. Refer to the Red Hat Linux Installation Guide for instructions on writing the image file to a diskette. Make sure you label the diskette "Ramdisk Diskette".

1.5 Installing

1.5.1 Booting the MILO Diskette

(If your Alpha system doesn't support MILO, please skip to Section 1.5.2 on page 8.)

Before you can boot your MILO diskette, you'll need to add a new boot selection to your ARC console. You can add a new boot selection for your MILO diskette by following these steps:

1. From the boot menu, select "`Supplementary menu`".
2. From the supplementary menu, select "`Set up the system`".
3. From the boot selections menu, select "`Add a boot selection`".
4. Select "`Floppy Disk 0`".

5. Enter "`\linload.exe`" as the osloader directory and name after erasing any default information.
6. Indicate that the operating system is in the same partition as the osloader by entering "`yes`".
7. Enter "`\`" as the operating system root directory after erasing any default information.
8. Give the boot selection a meaningful name. "`Linux`" is a nice name.
9. Indicate that you do not want to initialize the debugger at boot time by entering "no".
10. From the boot selection menu, select "`Change a boot selection option`". Pick the boot selection you just created.
11. Using the arrow keys, select "`OSLOADFILENAME`", and enter "\milo". Press Enter.
12. Using the arrow keys, select "`OSLOADOPTIONS`", and erase any information that may be there. Press Enter
13. Press Esc to leave the "`Change a boot selection option`" menu.
14. Select "`Setup Menu`". Select "`Supplementary menu, and save changes`".
15. Return to the boot menu.

At the end of this process, your boot selection should look something like this one (Not counting differences in boot selection name and disk identification strings:

```
LOADIDENTIFIER =Linux
SYSTEMPARTITION=multi(0)disk(0)fdisk(0)
OSLOADER=multi(0)disk(0)fdisk(0)\linload.exe
OSLOADERPARTITION=multi(0)disk(0)fdisk(0)
OSLOADFILENAME=\milo
OSLOADOPTIONS=
```

At this point your Alpha system has been set to boot MILO from diskette.

1.5.2 Booting the Kernel Diskette

Now it's time to get things started. We need to start by booting from the kernel diskette you've created. How this is done depends on your Alpha system. If it supports MILO, insert your MILO diskette, and boot from that. At the `MILO>` prompt, insert your kernel diskette, and enter the following boot command:

```
boot floppy
```

MILO should then read the Linux kernel from your boot disk and start running it.

On the other hand, if you cannot use MILO, and must boot using the SRM console, enter the appropriate boot command for your Alpha system, making sure to add the following arguments to be passed to the kernel:

```
"load_ramdisk=1 prompt_ramdisk=1"
```

1.5.3 Starting the Installation

After the kernel boots, it prompts for a ramdisk. Remove the kernel diskette, insert the ramdisk diskette, and press Enter. After the ramdisk is done loading, you should see the first installation screen.

1.5.4 Issues Specific to Red Hat Linux/Alpha

Before you return to the installation, you should know about some differences between Red Hat Linux/Intel and its Alpha counterpart. Let's review them:

Disk Partitioning

Red Hat Linux/Alpha supports two varieties of hard drive partitioning:

- **Standard `fdisk`-style partitioning**: this has the usual oddities (such as extended partitions) and is done in the same way as on Intel platforms (using Linux `fdisk`). This is the partitioning style you must use if you are using the ARC or AlphaBIOS console.

- **BSD-style disk-labelling**: It is normal to make the first partition on a disk-labelled hard drive the swap partition. We don't recommend using BSD disk-labelling unless it is absolutely necessary (such as in the case of sharing a hard drive between Digital Unix and Red Hat Linux/Alpha). Note that you need to use `aboot` if you use BSD-style disk labels. Note that an RPM package of `aboot` comes with Red Hat Linux/Alpha.

Creating a MILO Partition

If your system supports MILO, you should allocate a small (no more than 2 MB) partition to it. The partition type must be DOS. This is where you will copy MILO and `linload.exe` to after the installation completes.

XFree86 and TGA

If you have an Alpha with a TGA card and you choose to install XFree86, the installation program prompts you to choose a video card. Choose **Unlisted Server**, and the installation program will display a list of available servers. Choose the TGA server.

1.5.5 Finishing Up

After the installation is finished and your system is fully configured, you will be asked to reset your computer. This indicates that your system has been successfully installed.

Configuring MILO

In order to boot your newly installed system, you'll need to use your MILO diskette. To boot Red Hat Linux/Alpha from MILO, you must use the `boot` command. The command differs slightly depending on where your root partition is. For example, if your root partition is the second partition on your first SCSI hard drive, you would boot as follows:

```
boot sda2:vmlinux.gz root=/dev/sda2
```

However, if your root partition is the third partition on your second IDE drive, you would use this command:

```
boot hdb3:vmlinux.gz root=/dev/hdb3
```

Boot your Alpha system (using the appropriate root partition name for your system, of course). Once Red Hat Linux/Alpha has finished booting, login, and issue the following command:

```
dd if=/dev/fd0 of=/dev/sda1 bs=1440k
```

This will copy MILO (along with `linload.exe`) to the small MILO partition you created during installation.

Finally, you need to create a boot selection that will look for MILO on your MILO partition. Shutdown your Alpha system, and perform the following steps from the ARC console:

1. From the boot menu, select "`Supplementary menu`".
2. From the supplementary menu, select "`Set up the system`".
3. From the boot selections menu, select "`Add a boot selection`".
4. Select the line that describes your MILO partition, if shown. If it's not there, select "`New System Partition`".
5. Enter the device attributes (ie, SCSI/IDE, bus number, etc.) that describes your MILO partition.
6. Enter "`\linload.exe`" as the osloader directory and name after erasing any default information.
7. Indicate that the operating system is in the same partition as the osloader by entering "`yes`".
8. Enter "`\`" as the operating system root directory after erasing any default information.
9. Give the boot selection a meaningful name. "`Linux`" is a nice name.
10. Indicate that you do not want to initialize the debugger at boot time by entering "no".

11. From the boot selection menu, select "`Change a boot selection option`". Pick the boot selection you just created.
12. Using the arrow keys, select "`OSLOADFILENAME`", and enter "\milo". Press Enter.
13. Using the arrow keys, select "`OSLOADOPTIONS`", and erase any information that may be there. Alternatively, if you would like to have MILO automatically boot your Red Hat Linux/Alpha system, you can enter the boot command that you used to boot Red Hat Linux/Alpha from the MILO floppy. In either case, Press Enter when done.
14. Press Esc to leave the "`Change a boot selection option`" menu.
15. Select "`Setup Menu`". Select "`Supplementary menu, and save changes`".
16. Return to the boot menu.

Here's a sample boot selection:

```
LOADIDENTIFIER =Linux
SYSTEMPARTITION=scsi(0)disk(0)rdisk(0)partition(1)
OSLOADER=scsi(0)disk(0)rdisk(0)partition(1)\linload.exe
OSLOADERPARTITION=scsi(0)disk(0)rdisk(0)partition(1)
OSLOADFILENAME=\milo
OSLOADOPTIONS=boot sda2:vmlinux.gz root=/dev/sda2
```

Note that in this case we are using the `OSLOADOPTIONS` line to automatically direct MILO to boot our kernel.

At this point, reboot your system, the boot selection you just created. Once the `login:` prompt appears, log in as root, and continue with the post-installation tasks outlined in the Red Hat Linux Installation Guide.

2

Alpha Miniloader HOWTO

2.1 Introduction

This document describes the Miniloader for Linux on Alpha AXP (MILO). This firmware is used to initialize Alpha AXP based systems, load and start Linux and, finally, provide PALcode for Linux.

2.1.1 Copyright

Disclaimer. While I have tried to include the most correct and up to date information available to me, I cannot guarantee that usage of information in this document does not result in loss of data or equipment. I provide NO WARRANTY about the information in the HOWTO and I cannot be made liable for any consequences resulting from using the information in this HOWTO.

2.1.2 New Versions of this Document

The latest version of this document can be found in
`ftp://gatekeeper.dec.com/pub/Digital/Linux-Alpha/Miniloader/docs`
and David Mosberger-Tang is kind enough to include the html form of it in his excellent Linux Alpha FAQ site
`http://www.azstarnet.com/ axplinux`.

2.2 What is MILO?

On Intel based PC systems, the BIOS firmware sets up the system and then loads the image to be run from the boot block of a DOS file system. This is more or less what MILO does on an Alpha based system, however there are several interesting differences between BIOS firmware and MILO, not least of which is that MILO includes and uses standard Linux device drivers unmodified. MILO is firmware, unlike LILO, which relies on the BIOS firmware to get itself loaded. The main functional parts of MILO are:

1. PALcode,
2. Memory set up code (builds page tables and turns on virtual addressing),
3. Video code (BIOS emulation code and TGA (21030)),
4. Linux kernel code. This includes real Linux kernel code (for example, the interrupt handling) and ersatz or mock Linux kernel,
5. Linux block device drivers (for example, the floppy driver),
6. File system support (ext2, MS-DOS and ISO9660),
7. User inteface code (MILO),
8. Kernel interface code (sets up the HWRPB and memory map for linux),

9. NVRAM code for managing environment variables.

The following paragraphs describe these functional parts in more detail.

PALcode can be thought of as a tiny software layer that tailors the chip to a particular operating system. It runs in a special mode (PALmode) which has certain restrictions but it uses the standard Alpha instruction set with just five extra instructions. In this way, the Alpha chip can run such diverse operating systems as Windows NT, OpenVMS, Digital Unix and, of course, Linux. The PALcode that MILO uses (and therefore Linux itself) is, like the rest of MILO, freeware. It is derived from Digital's Evaluation Board software example Digital Unix PALcode.. The differences between the different PALcodes are because of differences in address mapping and interrupt handling that exist between the Alpha chips (21066 based systems have a different I/O map to 21064+2107x systems) and different Alpha based systems.

For MILO to operate properly it needs to know what memory is available, where Linux will eventually be running from and it must be able to allocate temporary memory for the Linux device drivers. The code maintains a memory map that has entries for permanent and temporary allocated pages. As it boots, MILO uncompresses itself into the correct place in physical memory. When it passes control to the Linux kernel, it reserves memory for the compressed version of itself, the PALcode (which the kernel needs) and some data structures. This leaves `most` of the memory in the system for Linux itself.

The final act of the memory code is to set up and turn on virtual addressing so that the data structures that Linux expects to see are at the correct place in virtual memory.

MILO contains video code that initialises and uses the video device for the system. It will detect and use a VGA device if there is one, otherwise it will try to use a TGA (21030) video device. Failing that, it will assume that there is no graphics device. The BIOS emulation that the standard, pre-built, images include is Digital's own BIOS emulation which supports most, if not all, of the standard graphics devices available.

Linux device drivers live within the kernel and expect certain services from the kernel. Some of these services are provided directly by Linux kernel code, for example the interrupt handling and some is provided by kernel look-alike routines.

MILO's most powerful feature is that you can embed unaltered Linux device drivers into it. This gives it the potential to support every device that Linux does. MILO includes all of the block devices that are configured into the Linux kernel that it is built against as well as a lot of the block device code (for example, ll_rw_blk()).

MILO loads the Linux kernel from real file systems rather than from boot blocks and other strange places. It understands MSDOS, EXT2 and ISO9660 filesystems. Gzip'd files are supported and these are recommended, particularly if you are loading from floppy which is rather slow. MILO recognises these by their *.gz* suffix.

Built into MILO is a simple keyboard driver which, together with an equally simple video driver allows it to have a simple user interface. That interface allows you to list file systems on configured devices, boot Linux or run flash update utilities and set environment variables that control the system's booting. Like LILO, you can pass arguments to the Kernel.

MILO must tell the Linux kernel what sort of system this is, how much memory there is and which of that memory is free. It does this using the HWRPB (Hardware Restart Parameter Block) data structure and associated memory cluster descriptions. These are placed at the appropriate place in virtual memory just before control is passed to the Linux kernel.

2.3 Pre-Built Standard MILO Images.

If you are planning to run Linux on a standard Alpha based system, then there are pre-built "standard" MILO images that you might use. These (along with the sources and other interesting stuff) can be found in `ftp://gatekeeper.dec.com/pub/Digital/Linux-Alpha/Miniloader`.

The `images` subdirectory contains a directory per standard system (eg AlphaPC64) with MILO images having the following naming convention:

1. `MILO` - Miniloader executable image, this image can be loaded in a variety of ways,
2. `fmu.gz` - Flash management utility,
3. `MILO.dd` - Boot block floppy disk image. These should be written using rawrite.exe or `dd` on Linux.

The `test-images`, like the `images` subdirectory contains a directory per standard system. These images are somewhat experimental but tend to contain all the latest features.

2.4 How To Build MILO

You build MILO separately from the Kernel. As MILO requires parts of the kernel to function (for example interrupt handling) you must first configure and build the kernel that matches with MILO that you want to build. Mostly this means building the kernel with the same version number. So, MILO-2.0.25.tar.gz will build against linux-2.0.25.tar.gz. MILO `may` build against a higher version of the kernel, but there again it may not. Also, now that ELF shared libraries are fully supported, there are two versions of the MILO sources. To build under an ELF system you must first unpack the standard MILO sources and then patch those sources with the same version numbered ELF patch. In the remainder of this discussion, I assume that your kernel sources and object files are stored in the subtree at `/usr/src/linux` and that the linux kernel has been fully built with the command `make boot`

To build MILO, change your working directory to the MILO source directory and invoke `make` with:

```
$    make KSRC=/usr/src/linux config
```

Just like the Linux kernel, you will be asked a series of questions

```
Echo output to the serial port (MINI_SERIAL_ECHO) [y]
```

It's a good idea to echo kernel printk to `/dev/ttyS0` if you can. If you can (and want to), then type "y", otherwise "n". All of the standard, pre-built, MILO images include serial port I/O using COM1.

```
Use Digital's BIOS emulation code (not free) \
                    (MINI_DIGITAL_BIOS_EMU) [y]
```

This code is included as a library which is freely distributable so long as it is used on an Alpha based system. The sources are not available. If you answer `n` then the freeware alternative BIOS emulation will be built. It's sources are included with MILO. Note that you cannot right now build choose Digital's BIOS emulation code in an ELF system (the library is not yet ready) and so you must answer no to this question.

```
Build PALcode from sources (Warning this is dangerous) \
                    (MINI_BUILD_PALCODE_FROM_SOURCES) [n]
```

You should only do this if you have changed the PALcode sources, otherwise use the standard, pre-built PALcode included with MILO.

You are now all set to build the MILO image itself:

```
$    make KSRC=/usr/src/linux
```

When the build has successfully completed, the MILO image is in the file called `milo`. There are a lot of images called `milo.*`, these should be ignored.

2.5 How To Load MILO

The most commonly supported method of loading MILO is from the Windows NT ARC firmware as most shipping systems support this. However, there are a wide variety of loading MILO. It may be loaded from:

- a failsafe boot block floppy,
- Windows NT ARC firmware,
- Windows NT AlphaBIOS,
- Digital's SRM console,
- an Alpha Evaluation Board Debug Monitor,
- flash/ROM.

2.5.1 Loading MILO from the Windows NT ARC firmware

Most, if not all, Alpha AXP based systems include the Windows NT ARC firmware and this is the prefered method of booting MILO and thus Linux. Once the Windows NT firmware is running and you have the correct MILO image for your system, this method is completely generic.

The Windows NT ARC firmware is an environment in which programs can run and make callbacks into the firmware to perform actions. The Windows NT OSLoader is a program that does exactly this. Linload.exe is a much simpler program which does just enough to load and execute MILO. It loads the appropriate image file into memory at 0x00000000 and then makes a swap-PAL PALcall to it. MILO, like Linux, uses a different PALcode to Windows NT which is why the swap has to happen. MILO relocates itself to 0x200000 and continues on through the PALcode reset entry point as before.

Before you add a Linux boot option, you will need to copy linload.exe and the appropriate MILO that you wish to load to someplace that the Windows NT ARC firmware can read from. In the following example, I assume that you are booting from a DOS format floppy disk:

1. At the boot menu, select `"Supplementary menu..."`
2. At the `"Supplementary menu"`, select `"Set up the system..."`
3. At the `"Setup menu"`, select `"Manage boot selection menu..."`
4. In the `"Boot selections menu"`, choose `"Add a boot selection"`
5. Choose `"Floppy Disk 0"`
6. Enter `"linload.exe"` as the osloader directory and name
7. Say `"yes"` to the operating system being on the same partition as the osloader
8. Enter `"\"` as the operating system root directory
9. I usually enter `"Linux"` as the name for this boot selection
10. Say `"No"` you do not want to initialise the debugger at boot time
11. You should now be back in the `"Boot selections menu"`, choose the `"Change a boot selection option"` and pick the selection you just created as the one to edit
12. Use the down arrow to get `"OSLOADFILENAME"` up and then type in the name of the MILO image that you wish to use, for example `"noname.arc"` followed by return.
13. Press ESC to get back to the `"Boot Selections menu"`
14. Choose `"Setup Menu"` (or hit ESC again) and choose `"Supplementary menu, and save changes"` option
15. ESC will get you back to the `"Boot menu"` and you can attempt to boot MILO. If you do not want Linux as the first boot option, then you can alter the order of the boot options in the `"Boot selections menu"`.

At the end of all this, you should have a boot selection that looks something like:

```
LOADIDENTIFIER=Linux
SYSTEMPARTITION=multi(0)disk(0)fdisk(0)
OSLOADER=multi(0)disk(0)fdisk(0)\linload.exe
OSLOADPARTITION=multi(0)disk(0)fdisk(0)
OSLOADFILENAME=\noname.arc
OSLOADOPTIONS=
```

You can now boot MILO (and then Linux). You can load linload.exe and MILO directly from a file system that Windows NT understands such as NTFS or DOS on a hard disk.

The contents `OSLOADOPTIONS` are passed to MILO which interprets it as a command. So, in order to boot Linux directly from Windows NT without pausing in MILO, you could pass the following in `OSLOADOPTIONS`:

```
boot sda2:vmlinux.gz root=/dev/sda2
```

See 2.6 for more information on the commands available.

Another (rather sneaky) way of loading MILO via the WNT ARC firmware is to put MILO onto an MS-DOS floppy and call it `fwupdate.exe` and then choose the "Upgrade Firmware" option.

2.5.2 Loading MILO from the Windows NT AlphaBIOS

With the introduction of the XLT series, Digital changed the system console interface for its NT systems from ARC to AlphaBIOS. AlphaBIOS is a screen-oriented interface which should be more familiar to PC users. This change in console interface necessitates a change in setup procedure for those who wish to run Linux/Alpha on AlphaBIOS-based systems.

The first thing you must do is to install the latest version of AlphaBIOS on your system. This can be obtained from Digital's "System Software and Driver Updates" Web page,
`http://www.windows.digital.com/support/sysoft.htp`. Download the ZIP file, unzip it, and install it as follows:

- Copy the files to a FAT-formatted floppy

- Turn on the system and insert the floppy. At the opening screen, press F2 to go into setup mode
- Select `"Upgrade AlphaBIOS"`
- Follow the directions on the screen

Once your AlphaBIOS is at the latest revision level, you can start bootstrapping your system as follows:

- Create a FAT-formatted floppy with the following files:
  - linload.exe (from this directory)
  - milo (the version appropriate to your system)
- Turn on the system and insert the floppy. At the opening screen, press F2 to go into setup mode
- Select `"Utilities->OS Selection Setup..."`
- Press INSERT to add a new operating system selection
- For `"Boot Name"`, enter something like `"Linux"`. Press TAB to get over to the next field.
- Press down-arrow until the selection for `"Boot File"` is `"A:"`. TAB over to the next field
- Enter `"linload.exe"`. TAB *twice* (i.e. skip the OS Path load device – it's irrelevant)
- Enter `"\"` for the OS Path load file
- Press ENTER to add the selection.

At this point, AlphaBIOS will probably put up a big, unfriendly dialog box labelled "Warning: Operating System Selection not valid!". Ignore this error (it's only a problem for NT) and press ENTER to continue.

- Press F10 to save the changes you just made; press ENTER to confirm the changes.
- Press ESC twice to get back to the opening screen.

- Use the up and down arrows to select the boot selection you just added, and press ENTER to boot it.
- AlphaBIOS will load linload, which will in turn load MILO. When you get to the MILO prompt, proceed as you would for a normal ARC-based system installation.

If you dedicated the first partition of your first disk drive to a small FAT partition for booting (as the installation procedure advises you to), then once Linux is installed you should copy linload.exe and milo to this partition. Once you shut down, you can then modify the Linux menu selection to load MILO from this partition as follows:

- At the opening screen, select F2 to go into setup mode
- Select `"Utilities->OS Selection setup"`
- Highlight the entry for Linux, then press F6 to edit it.
- TAB over to the device portion of the `"Boot File"` line. Use the up and down arrow keys to select the hard-disk partition where linload and milo reside (typically `"Disk 0 Partition 1"` or `"Disk 1 Partition 1"`). Press ENTER to confirm the selection
- If you wish to auto-boot linux after MILO is loaded, then TAB over to the `"OS Options"` line and enter the MILO command to boot the system, e.g. `"boot sda2:vmlinux.gz"` Press ENTER to confirm the selection.
- Press F10 to save the changes. Press ENTER to confirm.

Once you have done this, booting and running Linux on an AlphaBIOS based system should be very similar to doing so on an ARC system.

2.5.3 Loading MILO from the Evaluation Board Debug Monitor

Evaluation boards (and often designs cloned from them) include support for the Alpha Evaluation Board Debug Monitor. Consult your system document before considering this method of booting MILO. The following systems are *known* to include Debug Monitor support:

- AlphaPC64 (Section 2.5.7)

- EB64+ (Section 2.5.7)
- EB66+ (Section 2.5.7)
- EB164 (Section 2.5.7)
- PC164 (Section 2.5.7)

Before you consider this method, you should note that the early versions of the Evaluation Board Debug Monitor did not include video or keyboard drivers and so you must be prepared to connect another system via the serial port so that you can use the Debug Monitor. Its interface is very simple and typing help shows a whole heap of commands. The ones that are most interesting include the word `boot` or `load` in them.

The Evaluation Board Debug Monitor can load an image either via the network (netboot) or via a floppy (flboot). In either case, set the boot address to 0x200000 (> `bootadr 200000`) before booting the image.

If the image is on floppy (and note that only DOS formatted floppies are supported), then you will need to type the following command:

```
AlphaPC64> flboot <MILO-image-name>
```

2.5.4 Loading MILO from a Failsafe Boot Block Floppy

Only the AxpPCI33 is *known* to include failsafe boot block floppy support (Section 2.5.7).

If you do not have a standard pre-built MILO .dd image, then you may need to build an SRM boot block floppy. Once you have built MILO, you need to do the following on Digital Unix box:

```
fddisk -fmt /dev/rfd0a
cat mboot bootm > /dev/rfd0a
disklabel -rw rfd0a 'rx23' mboot bootm
```

Or on a Linux box:

```
cat mboot bootm > /dev/fd0
```

If you have a standard MILO image available (say `MILO.dd`) then you would build a boot block floppy using the following command:

```
dd if=MILO.dd of=/dev/fd0
```

2.5.5 Loading MILO from Flash

There are a number of systems where MILO can be blown into flash and booted directly (instead of via the Windows NT ARC firmware):

- AlphaPC64 (Section 2.5.7)
- Noname (Section 2.5.7)
- EB66+ (Section 2.5.7)
- EB164 (Section 2.5.7)
- PC164 (Section 2.5.7)

2.5.6 Loading MILO from the SRM Console

The SRM (short for System Reference Manual) Console knows nothing about filesystems or disk-partitions, it simply expects that the secondary bootstrap loader occupies a consecutive range of physical disk sectors starting from a given offset. The information describing the secondary bootstrap loader (its size and offset) is given in the first 512 byte block. To load MILO via the SRM you must generate that structure on a device which the SRM can access (such as a floppy disk). This is what `mboot` and `bootm`, `mboot` is the first block (or boot description) and `mboot` is the `MILO` image rounded up to a 512 byte boundary.

To load MILO from a boot block device, either build `mboot` and `bootm` and push them onto the boot device using the following command:

```
$ cat mboot bootm > /dev/fd0
```

Or, grab the appropriate `MILO.dd` from a web site and write it onto the boot device using either `RAWRITE.EXE` or `dd`.

Once you have done that you can boot the SRM console and use one of its many commands to boot MILO. For example, to boot MILO from a boot block floppy you would use the following command:

```
>>>boot dva0
(boot dva0.0.0.0.1 -flags 0)
block 0 of dva0.0.0.0.1 is a valid boot block
reading 621 blocks from dva0.0.0.0.1
bootstrap code read in
base = 112000, image_start = 0, image_bytes = 4da00
initializing HWRPB at 2000
initializing page table at 104000
initializing machine state
setting affinity to the primary CPU
jumping to bootstrap code
MILO Stub: V1.1
Unzipping MILO into position
Allocating memory for unzip
####...
```

The following systems are *known* to have SRM Console support:

- Noname (Section 2.5.7)
- AlphaPC64 (Section 2.5.7)
- EB164 (Section 2.5.7)
- PC164 (Section 2.5.7)

2.5.7 System Specific Information

AxpPCI33 (Noname)

The Noname board can load MILO from the Windows NT ARC firmware (Section 2.5.1), from the SRM Console (Section 2.5.6). and from a failsafe boot block floppy (Section 2.5.4). A flash management utility, runnable from MILO is available so that once MILO is running, it can be blown into flash (Section 2.7). However, be warned that once you have done this you will lose the previous image held there as there is only room for one image.

The way that Noname boots is controlled by a set of jumpers on the board, J29 and J28. These look like:

```
                4
J29         2 x x x 6
            1 x x x 5

J28         2 x x x 6
            1 x x x 5
                3
```

The two options that we're interested in are J28, pins 1-3 which boots the console/loader from flash and J29, pins 1-3 which boots the console/loader from a boot block floppy. The second option is the one that you need to first boot MILO on the Noname board.

Once you've selected the boot from floppy option via the jumpers, put the SRM boot block floppy containing MILO into the floppy and reboot. In a few seconds (after the floppy light goes out) you should see the screen blank to white and MILO telling you what's going on.

If you are really interested in technical stuff, the Noname loads images off of the floppy into physical address 0x104000 and images from flash into 0x100000. For this reason, MILO is built with it's PALcode starting at 0x200000. When it is first loaded, it moves itself to the correct location (see relocate.S).

AlphaPC64 (Cabriolet)

The AlphaPC64 includes the Windows NT ARC firmware (Section 2.5.1), the SRM Console (Section 2.5.6) and the Evaluation Debug Monitor (Section 2.5.3). These images are in flash and there is room to add MILO so that you can boot MILO directly from flash. A flash management utility, runnable from MILO is available so that once MILO is running, it can be blown into flash (Section 2.7). This system supports MILO environment variables.

You select between the boot options (and MILO when it is been put into flash) using a combination of jumpers and a boot option which is saved in the NVRAM of the TOY clock.

The jumper is J2, SP bits 6 and 7 have the following meanings:

- SP bit 6 should always be out. If this jumper is set then the SROM mini-debugger gets booted,

- SP bit 7 in is boot image selected by the boot option byte in the TOY clock,
- SP bit 7 out is boot first image in flash.

So, with bit 7 out, the Debug Monitor will be booted as it is `always` the first image in flash. With bit 7 in, the image selected by the boot option in the TOY clock will be selected. The Debug Monitor, the Windows NT ARC firmware and MILO all support setting this boot option byte but you must be very careful using it. In particular, you cannot set the boot option so that next time the system boots MILO when you are running the Windows NT ARC firmware, it only allows you to set Debug Monitor or Windows NT ARC as boot options.

To get MILO into flash via the Evaluation Board Debug Monitor, you will need a flashable image. The build proceedures make MILO.rom, but you can also make a rom image using the makerom tool in the Debug Monitor software that comes with the board:

```
> makerom -v -i7 -l200000 MILO -o mini.flash
```

(type makerom to find out what the arguments mean, but 7 is a flash image id used by the srom and -l200000 gives the load address for the image as 0x200000).

Load that image into memory (via the Debug Monitor commands flload, netload, and so on) at 0x200000 and then blow the image into flash:

```
AlphaPC64> flash 200000 8
```

(200000 is where the image to be blown is in memory and 8 is the segment number where you put the image. There are 16 1024*64 byte segments in the flash and the Debug Monitor is at seg 0 and the Windows NT ARC firmware is at seg 4).

Set up the image that the srom will boot by writing the number of the image into the TOY clock.

```
AlphaPC64> bootopt 131
```

(131 means boot the 3rd image, 129 = 1st, 130 = 2nd and so on).

Power off, put jumper 7 on and power on and you should see the MILO burst into life. If you don't then take jumper 7 back off and reboot the Debug Monitor.

EB66+

The EB66+, like all of the Alpha Evaluation Boards built by Digital contains the Evaluation Board Debug Monitor and so this is available to load MILO (Section 2.5.3). Quite often (although not always) boards whose design is derived from these include the Debug Monitor also. Usually, these boards include the Windows NT ARC firmware (Section 2.5.1). A flash management utility, runnable from MILO is available so that once MILO is running, it can be blown into flash (Section 2.7). This system supports MILO environment variables.

These systems have several boot images in flash controlled by jumpers. The two jumper banks are J18 and J16 and are located at the bottom of the board in the middle (if the Alpha chip is at the top). You select between the boot options (and MILO when it is been put into flash) using a combination of jumpers and a boot option which is saved in the NVRAM of the TOY clock.

Jumper 7-8 of J18 in means boot the image described by the boot option. Jumper 7-8 of J18 out means boot the Evaluation Board Debug Monitor.

Blowing an image into flash via the Evaluation Board Debug Monitor is exactly the same proceedure as for the AlphaPC64 (Section 2.5.7).

EB64+/Aspen Alpine

This system is quite like the AlphaPC64 except that it does not contain flash which MILO can be loaded from. The EB64+ has two ROMs, one of which contains the Windows NT ARC firmware (Section 2.5.1). and the other contains the Evaluation Board Debug Monitor (Section 2.5.3).

The Aspen Alpine is a little different in that it only has one ROM; this contains the Windows NT ARC firmware.

Universal Desktop Box (Multia)

This is a very compact pre-packaged 21066 based system that includes a TGA (21030) graphics device. Although you can *just* fit a half height PCI graphics card in the box you are better off waiting for full TGA support in XFree86. It includes the Windows NT ARC firmware and so booting from that is the prefered method (Section 2.5.1).

EB164

The EB164, like all of the Alpha Evaluation Boards built by Digital contains the Evaluation Board Debug Monitor and so this is available to load MILO (Section 2.5.3). Quite often (although not always) boards whose design is derived from these include the Debug Monitor also. Usually, these boards include the Windows NT ARC firmware (Section 2.5.1). The SRM console is also available (Section 2.5.6). A flash management utility, runnable from MILO is available so that once MILO is running, it can be blown into flash (Section 2.7). This system supports MILO environment variables.

These systems have several boot images in flash controlled by jumpers. The two jumper bank is J1 and is located at the bottom of the board on the left (if the Alpha chip is at the top). You select between the boot options (and MILO when it is been put into flash) using a combination of jumpers and a boot option which is saved in the NVRAM of the TOY clock.

Jumper SP-11 of J1 in means boot the image described by the boot option. Jumper SP-11 of J1 out means boot the Evaluation Board Debug Monitor.

Blowing an image into flash via the Evaluation Board Debug Monitor is exactly the same proceedure as for the AlphaPC64 (Section 2.5.7).

PC164

The PC164, like all of the Alpha Evaluation Boards built by Digital contains the Evaluation Board Debug Monitor and so this is available to load MILO (Section 2.5.3). Quite often (although not always) boards whose design is derived from these include the Debug Monitor also. Usually, these boards include the Windows NT ARC firmware (Section 2.5.1). The SRM console is also available (Section 2.5.6). A flash management utility, runnable from MILO is available so that once MILO is running, it can be blown into flash (Section 2.7). This system supports MILO environment variables.

These systems have several boot images in flash controlled by jumpers. The main jumper block, J30, contains the system configuration jumpers and jumper CF6 in means that the system will boot the Debug Monitor, the default is out.

Blowing an image into flash via the Evaluation Board Debug Monitor is exactly the same proceedure as for the AlphaPC64 (Section 2.5.7).

XL266

The XL266 is one of a family of systems that are known as Avanti. It has a riser card containing the Alpha chip and cache which plugs into the main board at right angles. This board can replace the equivalent Pentium board.

Some of these systems ship with the SRM console but others, notably the XL266 ship with only the Windows NT ARC firmware (Section 2.5.1).

Here is my list of compatible systems:

- AlphaStation 400 (Avanti),
- AlphaStation 250,
- AlphaStation 200 (Mustang),
- XL. There are two flavours, XL266 and XL233 with the only difference being in processor speed and cache size.

Note The system that I use to develop and test MILO is an XL266 and so this is the only one that I can guarentee will work. However, technically, all of the above systems are equivalent; they have the same support chipsets and the same interrupt handling mechanisms.

Platform2000

This is a 233Mhz 21066 based system.

2.6 MILO's User Interface

Once you have correctly installed/loaded/run MILO you will see the MILO (for MIniLOader) prompt displayed on your screen. There is a very simple interface that you must use in order to boot a particular Linux kernel image. Typing "help" is a good idea as it gives a useful summary of the commands.

2.6.1 The "help" Command

Probably the most useful command that MILO has:

```
MILO> help
MILO command summary:

ls [-t fs] [dev:[dir]]
                 - List files in directory on device
boot [-t fs] [dev:file] [boot string]
                 - Boot Linux from the specified device and file
run [-t fs] dev:file
                 - Run the standalone program dev:file
show             - Display all known devices and file systems
set VAR VALUE    - Set the variable VAR to the specified VALUE
unset VAR        - Delete the specified variable
reset            - Delete all variables
print            - Display current variable settings
help [var]       - Print this help text

Devices are specified as: fd0, hda1, hda2, sda1...
Use the '-t filesystem-name' option if you want to use
  anything but the default filesystem  ('ext2').
Use the 'show' command to show known devices and filesystems.
Type 'help var' for a list of variables.
```

Note that the `bootopt` command only appears on AlphaPC64 (and similar) systems. Refer to the board's dcoumentation to find out just what it means.

Devices. Until you use a command that needs to make use of a device, no device inititalisation will take place. The first `show`, `ls`, `boot` or `run` commands all cause the devices within MILO to be initialised. Devices are named in the same way (exactly) that Linux itself will name them. So, the first IDE disk will be called 'hda' and it's first partition will be 'hda1'. Use the `show` command to show what devices are available.

File Systems. MILO supports three file systems, MSDOS, EXT2 and ISO9660. So long as a device is available to it, MILO can `listboot` or `run` an image stored on one of these file systems. MILO's default file system is `EXT2` and so you have tell MILO that the file system is something other than that. All of the commands that use filenames allow you to pass the file system using the `-t [filesystem]` option. So, if you wanted to list the contents of a SCSI CD ROM, you might type the following:

```
MILO> ls -t iso9660 scd0:
```

Variables. MILO contains some settable variables that help the boot process. If you are loading via the Windows NT ARC firmware, then MILO makes use of the boot option environment variables set up by that firmware. For some systems, MILO (for example, the AlphaPC64) maintains its own set of environment variables that do not change from boot to boot. These variables are:

```
MILO> help var
Variables that MILO cares about:
MEMORY_SIZE  - System memory size in megabytes
BOOT_DEV     - Specifies the default boot device
BOOT_FILE    - Specifies the default boot file
BOOT_STRING  - Specifies the boot string to pass to the kernel
SCSIn_HOSTID - Specifies the host id of the n-th SCSI controller.
PCI_LATENCY  - Specifies the PCI master device latency
AUTOBOOT     - If set, MILO attempts to boot on powerup
               and enters command loop only on failure.
AUTOBOOT_TIMEOUT - Seconds to wait before auto-booting on powerup.
```

BEWARE of setting AUTOBOOT without setting the timeout, you can end up automatically booting after a timemout of 0 seconds which may not be what you want.

PCI_LATENCY is the number of PCI bus cycles that PCI bus mastering device hangs onto the PCI bus for once it has control. The default value for this is 32 and the maximum value is 255. Setting a high value means that each time a device gets control of the PCI bus it can transfer more data. However, setting it high also means that devices may have to wait longer to get control.

2.6.2 Booting Linux

The `boot` command boots a linux kernel from a device. You will need to have a linux kernel image on an EXT2 formated disk (SCSI, IDE or floppy) or an ISO9660 formatted CD available to MILO. The image can be gzip'd and in this case MILO will automatically gunzip it. Early versions of MILO recognised a gzip'd file by the .gz suffix but later MILOs look for magic numbers in the image.

You should note that the version of MILO does not usually have to match the version of the Linux kernel that you are loading. You boot Linux using the following command syntax:

```
MILO> boot [-t file-system] device-name:file-name \
                  [[boot-option] [boot-option] ...]
```

Where `device-name` is the name of the device that you wish to use and `file-name` is the name of the file containing the Linux kernel. All arguments supplied after the file name are passed directly to the Linux kernel.

If you are installing Red Hat, then you will need to specify a root device and so on. So you would use:

```
MILO> boot fd0:vmlinux.gz root=/dev/fd0 load_ramdisk=1
```

MILO will automatically contain the block devices that you configure into your vmlinux. I have tested the floppy driver, the IDE driver and a number of SCSI drivers (for example, the NCR 810), and these work fine. Also, it is important to set the host id of the SCSI controller to a reasonable value. By default, MILO will initialize it to the highest possible value (7) which should normally work just fine. However, if you wish, you can explicitly set the host id of the *n*-th SCSI controller in the system by setting environment variable `SCSI`*n*`_HOSTID` to the appropriate value. For example, to set the hostid of the first SCSI controller to 7, you can issue the following command at the MILO prompt:

```
setenv SCSI0_HOSTID 7
```

2.6.3 Rebooting Linux

You may want to reboot a running Linux system using the `shutdown -r now` command. In this case, the Linux kernel returns control to MILO (via the HALT CallPAL entrypoint). MILO leaves a compressed copy of itself in memory for just this reason and detects that the system is being rebooted from information held in the HWRPB (Hardware Restart Parameter Block). In this case it starts to reboot using exactly the same command that was used to boot the Linux kernel the last time. There is a 30 second timeout that allows you to interrupt this process and boot whatever kernel you wish in whatever way you wish.

2.6.4 The "bootopt" command

For flash based systems such as the AlphaPC64, EB164 and the EB66+, there are a number of possible boot options and these are changed using the `bootopt` command. This has one argument, a decimal number which is the type of the image to be booted the next time the system is power cycled or reset:

0 Boot the Evaluation Board Debug Monitor,

1 Boot the Windows NT ARC firmware.

In order to tell the boot code to boot the MILO firmware from flash then you need a boot option that means boot the N'th image. For this, you need to 128 plus N, so if MILO is the third image, you would use the command:

```
MILO> bootopt 131
```

`Note`: Be very careful with this command. A good rule is never to set bootopt to 0 (the Evaluation Board Debug Monitor), but instead use the system's jumpers to achieve the same thing.

2.7 Running the Flash Management Utility

The `run` command is used to run the flash management utility. Before you start you will need a device available to MILO that contains the updateflash program. This (like vmlinux) can be gzip'd. You need to run the flash management utility program from the MILO using the (`run`) command:

```
MILO> run fd0:fmu.gz
```

Once it has loaded and initialised, the flash management utility will tell you some information about the flash device and give you a command prompt. Again the `help` command is most useful.

```
Linux MILO Flash Management Utility V1.0

Flash device is an Intel 28f008SA
  16 segments, each of 0x10000 (65536) bytes
Scanning Flash blocks for usage
Block 12 contains the environment variables
FMU>
```

`Note` that on systems where environment variables may be stored and where there is more than one flash block (for example, the AlphaPC64) the flash management utility will look for a block to hold MILO's environment variables. If such a block already exists, the flash management utility will tell you where it is. Otherwise, you must use the `environment` command to set a block and initialise it. In the above example, flash block 12 contains MILO's environment variables.

2.7.1 The "help" command

```
FMU> help
FMU command summary:

list        - List the contents of flash
program     - program an image into flash
quit        - Quit
environment - Set which block should contain the environment \
                                                    variables
bootopt num - Select firmware type to use on next power up
help        - Print this help text
FMU>
```

Note that the `environment` and `bootopt` commands are only available on the EB66+, the AlphaPC64, EB164 and PC164 systems (and their clones).

2.7.2 The "list" command

The "list" command shows the current usage of the flash memory. Where there is more than one flash block, the usage of each flash block is shown. In the example below you can see that Windows NT ARC is using blocks 4:7 and block 15.

```
FMU> list
Flash blocks:  0:DBM  1:DBM  2:DBM  3:WNT  4:WNT  5:WNT  6:WNT
       7:WNT  8:MILO  9:MILO 10:MILO 11:MILO 12:MILO 13:U 14:U
15:WNT
Listing flash Images
  Flash image starting at block 0:
    Firmware Id:  0 (Alpha Evaluation Board Debug Monitor)
    Image size is 191248 bytes (3 blocks)
    Executing at 0x300000
```

```
  Flash image starting at block 3:
    Firmware Id:  1 (Windows NT ARC)
    Image size is 277664 bytes (5 blocks)
    Executing at 0x300000
  Flash image starting at block 8:
    Firmware Id:  7 (MILO/Linux)
    Image size is 217896 bytes (4 blocks)
    Executing at 0x200000
FMU>
```

2.7.3 The ”program” command

The flash management utility contains a compressed copy of a flash image of MILO. The ”program” command allows you to blow this image into flash. The command allows you to back out, but before you run it you should use the ”list” command to see where to put MILO. If MILO is already in flash, then the flash management utility will offer to overwrite it.

```
FMU> program
Image is:
    Firmware Id:  7 (MILO/Linux)
    Image size is 217896 bytes (4 blocks)
    Executing at 0x200000
Found existing image at block 8
Overwrite existing image? (N/y)? y
Do you really want to do this (y/N)? y
Deleting blocks ready to program: 8 9 10 11
Programming image into flash
Scanning Flash blocks for usage
FMU>
```

Wait until it has completed before powering off your system.

`Note:` I cannot emphasise just how careful you must be here not to overwrite an existing flash image that you might need or render your system useless. A very good rule is never to overwrite the Debug Monitor.

2.7.4 The ”environment” command

This selects a flash block to contain MILO’s environment variables.

2.7.5 The ”bootopt” command

This is just the same as MILO’s ”bootopt” command, see (Section 2.6.4).

2.7.6 The ”quit” command

This is really pretty meaningless. The only way back to MILO (or anything else) once the flash management utility has run is to reboot the system.

2.8 Restrictions.

Unfortunately this is not a perfect world and there, as always, some restrictions that you should be aware of.

MILO is not meant to load operating systems other than Linux, although it can load images linked to run at the same place in memory as Linux (which is 0xFFFFFC0000310000). This is how the flash management utilities can be run.

The PALcode sources included in `miniboot/palcode/`*blah* are correct, however there are problems when they are built using the latest `gas`. They *do* build if you use the ancient a.out gas that’s supplied in the Alpha Evaluation Board toolset (and that’s how they were built). I’m trying to get someone to fix the new gas. Meanwhile, as a workaround, I have provided pre-built PALcode for the supported boards and David Mosberger-Tang has a fixed gas on his ftp site.

2.9 Problem Solving.

Here are some common problems that people have seen, together with the solutions.

Reading MS-DOS floppies from the Evaluation Board Debug Monitor.

Some of the older versions of the Evaluation Board Debug Monitor (pre-version 2.0) have a problem with DOS format flopies generated from Linux. Usually, the Debug Monitor can load the first few sectors all right, but then goes into an endless loop complaining about ”bad sectors.” Apparently, there is an incompatibility between the DOS file system as expected by the Debug Monitor and the Linux implementation of DOSFS. To make the long story short: if you run into this problem, try using DOS to write the floppy disk. For example, if loading the file `MILO.cab` doesn't work, use a DOS machine, insert the floppy and then do:

```
copy a:MILO.cab c:
copy c:MILO.cab a:
del c:MILO.cab
```

Then try booting from that floppy again. This normally solves the problem.

MILO displays a long sequence of `O>` and does not accept input.

This usually happens when MILO was built to use COM1 as a secondary console device. In such a case, MILO echo output to COM1 and accepts input from there also. This is great for debugging but not so great if you have a device other than a terminal connected. If this happens, disconnect the device or power it down until the Linux kernel has booted. Once Linux is up and running, everything will work as expected.

MILO complains that the kernel image has the wrong magic number

Older versions of MILO did not support the ELF object file format and so could not recognise an ELF image and this might be your problem. If this is reported, upgrade to the latest MILO that you can find. All 2.0.20 and beyond MILOs support ELF. On the other hand it could be that the image is indeed damaged. You should also note that MILO does not yet automatically distinquish between GZIP'd and non-GZIP'd images; you need to add the ".gz" suffix to the file name.

MILO prints "...turning on virtual addressing and jumping to the Linux Kernel" and nothing else happens

One obvious problem is that the kernel image is wrongly built or is built for another Alpha system altogether. Another is that the video board is a TGA (Zlxp) device and the kernel has been built for a VGA device (or vice versa). It is worth building the kernel to echo to COM1 and then connecting a terminal to that serial port or retrying the kernel that came with the Linux distribution that you installed.

MILO does not recognise the SCSI device

The standard MILO images include as many device drivers as are known to be stable for Alpha (as of now that includes the NCR 810, QLOGIC ISP, Buslogic and Adaptec 2940s and 3940 cards). If your card is not included, it may be that the driver is not stable enough on an Alpha system yet. Again, the latest MILO images are worth trying. You can tell which SCSI devices a MILO image has built into it by using the ”show” command.

2.10 Acknowledgements.

I would like to thank:

- Eric Rasmussen and Eilleen Samberg the authors of the PALcode,
- Jim Paradis for the keyboard driver, the original MILO interface and the AlphaBIOS work,
- Jay Estabrook for his help and bugfixes,
- David Mosberger-Tang for the freeware BIOS emulation and his support and encouragement,
- Last (and `not` least) Linus Torvalds for the timer code and his kernel.

There are a number of things that still need doing to MILO, if you want to add something yourself, then do let me (`david.rusling@reo.mts.dec.com`) know so that we do not duplicate our efforts.

Finally, a big thank you to Digital for producing such a wonderful chip (and paying me to do this).

3

SRM Firmware HOWTO

Unless you're interested in technical details, you may want to skip right to Section 3.3 on page 43.

3.1 How Does SRM Boot an OS?

All versions of SRM can boot from SCSI disks and the versions for recent platforms, such as the Noname or AlphaStations can boot from floppy disks as well. Network booting via `bootp` is supported. Note that older SRM versions (notably the one for the Jensen) *cannot* boot from floppy disks. Also, booting from IDE disk drives is unsupported.

Booting Linux with SRM is a two step process: first, SRM loads and transfers control to the secondary bootstrap loader. Then the secondary bootstrap loader sets up the environment for Linux, reads the kernel image from a disk filesystem and finally transfers control to Linux.

Currently, there are two secondary bootstrap loaders for Linux: the *raw* loader that comes with the Linux kernel and `aboot` which is distributed separately. These two loaders are described in more detail below.

3.1.1 Loading The Secondary Bootstrap Loader

SRM knows nothing about filesystems or disk-partitions. It simply expects that the secondary bootstrap loader occupies a consecutive range of physical disk sector, starting from a given offset. The information on the size of the secondary bootstrap loader and the offset of its first disk sector is stored in the first 512 byte sector. Specifically, the long integer at offset 480 stores the *size* of the secondary bootstrap loader (in 512-byte blocks) and the long at offset 488 gives the *sector number* at which the secondary bootstrap loader starts. The first sector also stores a flag-word at offset 496 which is always 0 and a checksum at offset 504. The checksum is simply the sum of the first 63 long integers in the first sector.

If the checksum in the first sector is correct, SRM goes ahead and reads the *size* sectors starting from the sector given in the *sector number* field and places them in *virtual* memory at address `0x20000000`. If the reading completes successfully, SRM performs a jump to address `0x20000000`.

3.2 The Raw Loader

The sources for this loader can be found in directory

```
linux/arch/alpha/boot
```

of the Linux kernel source distribution. It loads the Linux kernel by reading `START_SIZE` bytes starting at disk offset `BOOT_SIZE+512` (also in bytes). The constants `START_SIZE` and `BOOT_SIZE` are defined in `linux/include/asm-alpha/system.h`. `START_SIZE` must be at least as big as the kernel image (i.e., the size of the `.text`, `.data`, and `.bss` segments). Similarly, `BOOT_SIZE` must be at least as big as the image of the raw bootstrap loader. Both constants should be an integer multiple of the sector size, which is 512 bytes. The default values are currently 2MB for `START_SIZE` and 16KB for `BOOT_SIZE`. Note that if you want to boot from a 1.44MB floppy disk, you have to reduce `START_SIZE` to 1400KB and make sure that the kernel you want to boot is no bigger than that.

To build a raw loader, simply type `make rawboot` in `/usr/src/linux`. This should produce the following files in `arch/alpha/boot`:

`tools/lxboot`:

The first sector on the disk. It contains the offset and size of the next file in the format described above.

tools/bootlx:

The raw boot loader that will load the file below.

vmlinux.nh:

The raw kernel image consisting of the `.text`, `.data`, and `.bss` segments of the object file in `/usr/src/linux/vmlinux`. The extension `.nh` indicates that this file has no object-file header.

The concatenation of these three files should be written to the disk from which you want to boot. For example, to boot from a floppy, insert an empty floppy disk in, say, `/dev/fd0` and then type:

```
cat tools/lxboot tools/bootlx vmlinux >/dev/fd0
```

You can then shutdown the system and boot from the floppy by issueing the command `boot dva0`.

3.3 The aboot Loader

When using the SRM firmware, `aboot` is the preferred way of booting Linux. It supports:

- direct booting from various filesystems (`ext2`, `ISO9660`, and `UFS`, the DEC Unix filesystem)
- booting of executable object files (both ELF and ECOFF)
- booting compressed kernels
- network booting (using bootp)
- partition tables in DEC Unix format (which is compatible with BSD Unix partition tables)
- interactive booting and default configurations for SRM consoles that cannot pass long option strings

3.3.1 Getting and Building aboot

The latest sources for `aboot` are available in `ftp://ftp.azstarnet.com/pub/linux/axp/aboot`this ftp directory. The description in this manual applies to `aboot` version 0.5 or newer.

Once you downloaded and extracted the latest tar file, take a look at the `README` and `INSTALL` files for installation hints. In particular, be sure to adjust the variables in `Makefile` and in `include/config.h` to match your environment. Normally, you won't need to change anything when building under Linux, but it is always a good idea to double check. If you're satisfied with the configuration, simply type `make` to build it (if you're not building under Linux, be advised that `aboot` requires GNU `make`).

After running `make`, the `aboot` directory should contain the following files:

aboot

This is the actual `aboot` executable (either an ECOFF or ELF object file).

bootlx

Same as above, but it contains only the text, data and bss segments—that is, this file is not an object file.

sdisklabel/writeboot

Utility to install `aboot` on a hard disk.

tools/e2writeboot

Utility to install `aboot` on an ext2 filesystem (usually used for floppies only).

tools/isomarkboot

Utility to install `aboot` on a iso9660 filesystem (used by CD-ROM distributors).

tools/abootconf

Utility to configure an installed `aboot`.

3.3.2 Floppy Installation

The bootloader can be installed on a floppy using the `e2writeboot` command (note: this can't be done on a Jensen since its firmware does *not* support booting from floppy). This command requires that the disk is not overly fragmented as it needs to find enough contiguous file blocks to store the entire `aboot` image (currently about 90KB). If `e2writeboot` fails because of this, reformat the floppy and try again (e.g., with `fdformat(1)`). For example, the following steps install `aboot` on floppy disk assuming the floppy is in drive `/dev/fd0`:

```
fdformat /dev/fd0
mke2fs /dev/fd0
e2writeboot /dev/fd0 bootlx
```

3.3.3 Harddisk Installation

Since the `e2writeboot` command may fail on highly fragmented disks and since reformatting a harddisk is not without pain, it is generally safer to install `aboot` on a harddisk using the `swriteboot` command. `swriteboot` requires that the first few sectors are reserved for booting purposes. We suggest that the disk be partitioned such that the first partition starts at an offset of 2048 sectors. This leaves 1MB of space for storing `aboot`. On a properly partitioned disk, it is then possible to install `aboot` as follows (assuming the disk is `/dev/sda`):

```
swriteboot /dev/sda bootlx
```

On a Jensen, you will want to leave some more space, since you need to write a kernel to this place, too—2MB should be sufficient when using compressed kernels. Use `swriteboot` as described in Section 3.3.6 on the next page to write `bootlx` together with the Linux kernel.

3.3.4 CD-ROM Installation

To make a CD-ROM bootable by SRM, simply build `aboot` as described above. Then, make sure that the `bootlx` file is present on the iso9660 filesystem (e.g., copy `bootlx` to the directory that is the filesystem master, then run `mkisofs` on that directory). After that, all that remains to be done is to mark the filesystem as SRM bootable. This is achieved with a command of the form:

```
isomarkboot filesystem bootlx
```

The command above assumes that `filesystem` is a file containing the iso9660 filesystem and that `bootlx` has been copied into the root directory of that filesystem. That's it!

3.3.5 Building the Linux Kernel

A bootable Linux kernel can be built with the following steps. During the `make config`, be sure to answer "yes" to the question whether you want to boot the kernel via SRM.

```
cd /usr/src/linux
make config
make dep
make boot
```

The last command will build the file `arch/alpha/boot/vmlinux.gz` which can then be copied to the disk from which you want to boot from. In our floppy disk example above, this would entail:

```
mount /dev/fd0 /mnt
cp arch/alpha/boot/vmlinux.gz /mnt
umount /mnt
```

3.3.6 Booting Linux

With the SRM firmware and `aboot` installed, Linux is generally booted with a command of the form:

boot *devicename* −fi *filename* −fl *flags*

The *filename* and *flags* arguments are optional. If they are not specified, SRM uses the default values stored in environment variables `BOOT_OSFILE` and `BOOT_OSFLAGS`. The syntax and meaning of these two arguments is described in more detail below.

Boot Filename

The filename argument takes the form:

[*n*/]*filename*

n is a single digit in the range 1..8 that gives the partition number from which to boot from. *filename* is the path of the file you want boot. For example to boot from the second partition of SCSI device 6, you would enter:

```
boot dka600 -file 2/vmlinux.gz
```

Or to boot from floppy drive 0, you'd enter:

```
boot dva0 -file vmlinux.gz
```

If a disk has no partition table , `aboot` pretends the disk contains one `ext2` partition starting at the first diskblock. This allows booting from floppy disks.

As a special case, partition number 0 is used to request booting from a disk that does not (yet) contain a file system. When specifying "partition" number 0, `aboot` assumes that the Linux kernel is stored right behind the `aboot` image. Such a layout can be achieved with the `swriteboot` command. For example, to setup a filesystem-less boot from `/dev/sda`, one could use the command:

```
swriteboot /dev/sda bootlx vmlinux.gz
```

Booting a system in this way is not normally necessary. The reason this feature exists is to make it possible to get Linux installed on a systems that can't boot from a floppy disk (e.g., the Jensen).

Boot Flags

A number of bootflags can be specified. The syntax is:

```
-flags "options..."
```

Where "options..." is any combination the following options (separated by blanks). There are many more bootoptions, depending on what drivers your kernel has installed. The options listed below are therefore just examples to illustrate the general idea:

load_ramdisk=1

Copy root file system from a (floppy) disk to the RAM disk before starting the system. The RAM disk will be used in lieu of the root device. This is useful to bootstrap Linux on a system with only one floppy drive.

floppy=*str*

Sets floppy configuration to *str*.

root=*dev*

Select device *dev* as the root-file system. The device can be specified as a major/minor hex number (e.g., 0x802 for /dev/sda2) or one of a few canonical names (e.g., `/dev/fd0`, `/dev/sda2`).

single

Boot system in single user mode.

kgdb

Enable kernel-gdb (works only if `CONFIG_KGDB` is enabled; a second Alpha system needs to be connected over the serial port in order to make this work)

Some SRM implementations (e.g., the one for the Jensen) are handicapped and allow only short option strings (e.g., at most 8 characters). In such a case, `aboot` can be booted with the single-character boot flag "i". With this flag, `aboot` will prompt the user to interacively enter a boot option string of up to 256 characters. For example:

```
boot dka0 -fl i
aboot> 3/vmlinux.gz root=/dev/sda3 single
```

Since booting in that manner quickly becomes tedious, `aboot` allows to define short-hands for frequently used commandlines. In particular, a single digit option (0-9) requests that `aboot` uses the corresponding option string stored in file `/etc/aboot.conf`. A sample `aboot.conf` is shown below:

```
#
# aboot default configurations
#
0:3/vmlinux.gz root=/dev/sda3
1:3/vmlinux.gz root=/dev/sda3 single
2:3/vmlinux.new.gz root=/dev/sda3
3:3/vmlinux root=/dev/sda3
8:- root=/dev/sda3            # fs-less boot of raw kernel
9:0/vmlinux.gz root=/dev/sda3 # fs-less boot of (compressed)
                                                   ECOFF kernel
-
```

With this configuration file, the command

```
boot dka0 -fl 1
```

corresponds exactly to the boot command shown above. It is quite easy to forget what number corresponds to what option string. To alleviate this problem, boot with option "h" and `aboot` will print the contents of `/etc/aboot.conf` before issueing the prompt for the full option string.

Finally, whenever `aboot` prompts for an option string, it is possible to enter one of the single character flags ("i", "h", or "0"-"9") to get the same effect as if that flag had been specified in the boot command line. For example, you could boot with flag "i" and then type "h" (followed by return) to remind yourself of the contents of `/etc/aboot.conf`

Selecting the Partition of /etc/aboot.conf When installed on a harddisk, `aboot` needs to know what partition to search for the `/etc/aboot.conf` file. A newly compiled `aboot` will search the *second* partition (e.g., `/dev/sda2`). Since it would be inconvenient to have to recompile `aboot` just to change the partition number, `abootconf` allows to directly modify an installed `aboot`. Specifically, if you want to change `aboot` to use the *third* partition on disk `/dev/sda`, you'd use the command:

```
abootconf /dev/sda 3
```

You can verify the current setting by simply omitting the partition number. That is: `abootconf /dev/sda` will print the currently selected partition number. Note that aboot does have to be installed already for this command to succeed. Also, when installing a new aboot, the partition number will fall back to the default (i.e., it will be necessary to rerun abootconf).

Since aboot version 0.5, it is also possible to select the aboot.conf partition via the boot command line. This can be done with a command line of the form *a*:*b* where *a* is the partition that holds /etc/aboot.conf and *b* is a single-letter option as described above (0-9, i, or h). For example, if you type boot -fl "3:h" dka100 the system boots from SCSI ID 1, loads /etc/aboot.conf from the third partition, prints its contents on the screen and waits for you to enter the boot options.

3.3.7 Booting Over the Network

Two prelimenary steps are necessary before Linux can be booted via a network. First, you need to set the SRM environment variables to enable booting via the bootp protocol and second you need to setup another machine as the your boot server. Please refer to the SRM documentation that came with your machine for information on how to enable bootp. Setting up the boot server is obviously dependent on what operating system that machine is running, but typically it involves starting the program bootpd in the background after configuring the /etc/bootptab file. The bootptab file has one entry describing each client that is allowed to boot from the server. For example, if you want to boot the machine myhost.cs.arizona.edu, then an entry of the following form would be needed:

```
myhost.cs.arizona.edu:\
        :hd=/remote/:bf=vmlinux.bootp:\
        :ht=ethernet:ha=08012B1C51F8:hn:vm=rfc1048:\
        :ip=192.12.69.254:bs=auto:
```

This entry assumes that the machine's Ethernet address is 08012B1C51F8 and that its IP address is 192.12.69.254. The Ethernet address can be found with the show device command of the SRM console or, if Linux is running, with the ifconfig command. The entry also defines that if the client does not specify otherwise, the file that will be booted is vmlinux.bootp in directory /remote. For more information on configuring bootpd, please refer to its man page.

Next, build aboot with with the command make netboot. Make sure the kernel that you want to boot has been built already. By default, the aboot Makefile uses the kernel in /usr/src/linux/arch/alpha/boot/vmlinux.gz (edit the Makefile if you want to use a different path). The result of make netboot is a file called vmlinux.bootp which contains aboot *and* the Linux kernel, ready for network booting.

Finally, copy vmlinux.bootp to the bootsever's directory. In the example above, you'd copy it into /remote/vmlinux.bootp. Next, power up the client machine and boot it, specifying the Ethernet adapter as the boot device. Typically, SRM calls the first Ethernet adapter ewa0, so to boot from that device, you'd use the command:

```
boot ewa0
```

The -fi and -fl options can be used as usual. In particular, you can ask aboot to prompt for Linux kernel arguments by specifying the option -fl i.

3.4 Sharing a Disk With DEC Unix

Unfortunately, DEC Unix doesn't know anything about Linux, so sharing a single disk between the two OSes is not entirely trivial. However, it is not a difficult task if you heed the tips in this section. The section assumes you are using aboot version 0.5 or newer.

3.4.1 Partitioning the disk

First and foremost: *never* use any of the Linux partitioning programs (minlabel or fdisk) on a disk that is also used by DEC Unix. The Linux minlabel program uses the same partition table format as DEC Unix disklabel, but there are some incompatibilities in the data that minlabel fills in, so DEC Unix will simply refuse to accept a partition table generated by minlabel. To setup a Linux ext2 partition under DEC Unix, you'll have to change the disktab entry for your disk. For the purpose of this discussion, let's assume that you have an rz26 disk (a common 1GB drive) on which you want to install Linux. The disktab entry under DEC Unix v3.2 looks like this (see file /etc/disktab):

```
rz26|RZ26|DEC RZ26 Winchester:\
        :ty=winchester:dt=SCSI:ns#57:nt#14:nc#2570:\
```

```
:oa#0:pa#131072:ba#8192:fa#1024:\
:ob#131072:pb#262144:bb#8192:fb#1024:\
:oc#0:pc#2050860:bc#8192:fc#1024:\
:od#393216:pd#552548:bd#8192:fd#1024:\
:oe#945764:pe#552548:be#8192:fe#1024:\
:of#1498312:pf#552548:bf#8192:ff#1024:\
:og#393216:pg#819200:bg#8192:fg#1024:\
:oh#1212416:ph#838444:bh#8192:fh#1024:
```

The interesting fields here are `o?`, and `p?`, where *?* is a letter in the range `a-h` (first through 8-th partition). The `o` value gives the starting offset of the partition (in sectors) and the `p` value gives the size of the partition (also in sectors). See `disktab(4)` for more info. Note that DEC Unix likes to define overlapping partitions. For the entry above, the partition layout looks like this (you can verify this by adding up the various `o` and `p` values):

```
  a    b        d          e          f
|---|-------|-----------|-----------|-----------|

                     c
|-----------------------------------------------|

                  g               h
            |----------------|----------------|
```

DEC Unix insists that partition `a` starts at offset 0 and that partition `c` spans the entire disk. Other than that, you can setup the partition table any way you like.

Let's suppose you have DEC Unix using partition `g` and want to install Linux on partition `h` with partition `b` being a (largish) swap partition. To get this layout without destroying the existing DEC Unix partition, you need to set the partition types explicitly. You can do this by adding a `t` field for each partition. In our case, we add the following line to the above disktab entry.

```
:ta=unused:tb=swap:tg=4.2BSD:th=resrvd8:
```

Now why do we mark partition `h` as "reservd8" instead of "ext2"? Well, DEC Unix doesn't know about Linux. It so happens that partition type "ext2" corresponds to a numeric value of 8, and DEC Unix uses the string "reservd8" for that value. Thus, in DEC Unix speak, "reservd8" means "ext2". OK, this was the hard part. Now we just need to install the updated disktab entry on the disk. Let's assume the disk has SCSI id 5. In this case, we'd do:

```
disklabel -rw /dev/rrz5c rz26
```

You can verify that everything is all right by reading back the disklabel with `disklabel -r /dev/rrz5c`. At this point, you may want to reboot DEC Unix and make sure the existing DEC Unix partition is still alive and well. If that is the case, you can shut down the machine and start with the Linux installation. Be sure to skip the disk partitioning step during the install. Since we already installed a good partition table, you should be able to proceed and select the 8th partition as the Linux root partition and the 2nd partition as the swap partition. If the disk is, say, the second SCSI disk in the machine, then the device name for these partitions would be `/dev/sdb8` and `/dev/sdb2`, respectively (note that Linux uses letters to name the drives and numbers to name the partitions, which is exactly reversed from what DEC Unix does; the Linux scheme makes more sense, of course ;-).

3.4.2 Installing `aboot`

First big caveat: with the SRM firmware, you can boot one and only one operating system per disk. For this reason, it is generally best to have at least two SCSI disks in a machine that you want to dualboot between Linux and DEC Unix. Of course, you could also boot Linux from a floppy if speed doesn't matter or over the network, if you have a `bootp`-capable server. But in this section we assume you want to boot Linux from a disk that contains one or more DEC Unix partitions.

Second big caveat: installing `aboot` on a disk shared with DEC Unix renders the first and third partition unusable (since those *must* have a starting offset of 0). For this reason, we recommend that you change the size of partition `a` to something that is just big enough to hold `aboot` (1MB should be plenty).

Once these two caveats are taken care of, installing `aboot` is almost as easy as usual: since partition `a` and `c` will overlap with `aboot`, we need to tell `swriteboot` that this is indeed OK. We can do this under Linux with a command line of the following form (again, assuming we're trying to install `aboot` on the second SCSI disk):

```
swriteboot -f1 -f3 /dev/sdb bootlx
```

The `-f1` means that we want to force writing `bootlx` even though it overlaps with partition 1. The corresponding applies for partition 3.

This is it. You should now be able to shutdown the system and boot Linux from the harddisk. In our example, the SRM command line to do this would be:

```
boot dka5 -fi 8/vmlinux.gz -fl root=/dev/sdb8
```

4

Alpha HOWTO

4.1 What is Alpha

”Alpha” is the name given to Digital’s 64-bit RISC architecture. The Alpha project in Digital began in mid-1989, with the goal of providing a high-performance migration path for VAX customers. This was not the first RISC architecture to be produced by Digital, but it was the first to reach the market. When Digital announced Alpha, in March 1992, it made the decision to enter the merchant semicondutor market by selling Alpha microprocessors.

Alpha is also sometimes referred to as Alpha AXP, for obscure and arcane reasons that aren’t worth persuing. Suffice it to say that they are one and the same.

4.2 What is Digital Semiconductor

`http://www.digital.com/info/semiconductor/`Digital Semiconductor (DS) is the business unit within Digital Equipment Corporation (Digital - we don’t like the name DEC) that sells semiconductors on the merchant market. Digital’s products include CPUs, support chipsets, PCI-PCI bridges and PCI peripheral chips for comms and multimedia.

4.3 Alpha CPUs

There are currently 2 generations of CPU core that implement the Alpha architecture:

- EV4
- EV5

Opinions differ as to what ”EV” stands for (Editor’s note: the true answer is of course ”Electro Vlassic” 4.12 ([1])), but the number represents the first generation of Digital’s CMOS technology that the core was implemented in. So, the EV4 was originally implemented in CMOS4. As time goes by, a CPU tends to get a mid-life performance kick by being optically shrunk into the next generation of CMOS process. EV45, then, is the EV4 core implemented in CMOS5 process. There is a big difference between shrinking a design into a particular technology and implementing it from scratch in that technology (but I don’t want to go into that now). There are a few other wildcards in here: there is also a CMOS4S (optical shrink in CMOS4) and a CMOS5L.

True technophiles will be interested to know that CMOS4 is a 0.75 micron process, CMOS5 is a 0.5 micron process and CMOS6 is a 0.35 micron process.

To map these CPU cores to *chips* we get:

21064-150,166

EV4 (originally), EV4S (now)

21064-200

EV4S

21064A-233,275,300

EV45

21066

LCA4S (EV4 core, with EV4 FPU)

21066A-233

LCA45 (EV4 core, but with EV45 FPU)

21164-233,300,333

EV5

21164A-417

EV56

21264

EV6
(`http://www.mdronline.com/report/articles/21264/21264.html`)

The EV4 core is a dual-issue (it can issue 2 instructions per CPU clock) superpipelined core with integer unit, floating point unit and branch prediction. It is fully bypassed and has 64-bit internal data paths and tightly coupled 8Kbyte caches, one each for Instruction and Data. The caches are write-through (they never get dirty).

The EV45 core has a couple of tweaks to the EV4 core: it has a slightly improved floating point unit, and 16KB caches, one each for Instruction and Data (it also has cache parity). (Editor's note: Neal Crook indicated in a separate mail that the changes to the floating point unit (FPU) improve the performance of the divider. The EV4 FPU divider takes 34 cycles for a single-precision divide and 63 cycles for a double-precision divide (non data-dependent). In constrast, the EV45 divider takes typically 19 cycles (34 cycles max) for single-precision and typically 29 cycles (63 cycles max) for a double-precision division (data-dependent).)

The EV5 core is a quad-issue core, also superpipelined, fully bypassed etc etc. It has tightly-coupled 8Kbyte caches, one each for I and D. These caches are write-through. It also has a tightly-coupled 96Kbyte on-chip second-level cache (the Scache) which is 3-way set associative and write-back (it can be dirty). The EV4->EV5 performance increase is better than just the increase achieved by clock speed improvements. As well as the bigger caches and quad issue, there are microarchitectural improvements to reduce producer/consumer latencies in some paths.

The EV56 core is fundamentally the same microarchitecture as the EV5, but it adds some new instructions for 8 and 16-bit loads and stores (see Section 4.8 (Bytes and all that stuff)). These are primarily intended for use by device drivers. The EV56 core is implemented in CMOS6, which is a 2.0V process.

The 21064 was anounced in March 1992. It uses the EV4 core, with a 128-bit bus interface. The bus interface supports the ’easy’ connection of an external second-level cache, with a block size of 256-bits (2 data beats on the bus). The Bcache timing is completely software configurable. The 21064 can also be configured to use a 64-bit external bus, (but I’m not sure if any shipping system uses this mode). The 21064 does not impose any policy on the Bcache, but it is usually configured as a write-back cache. The 21064 does contain hooks to allow external hardware to maintain cache coherence with the Bcache and internal caches, but this is hairy.

The 21066 uses the EV4 core and integrates a memory controller and PCI host bridge. To save pins, the memory controller has a 64-bit data bus (but the internal caches have a block size of 256 bits, just like the 21064, therefore a block fill takes 4 beats on the bus). The memory controller supports an external Bcache and external DRAMs. The timing of the Bcache and DRAMs is completely software configurable, and can be controlled to the resolution of the CPU clock period. Having a 4-beat process to fill a cache block isn’t as bad as it sounds because the DRAM access is done in page mode. Unfortunately, the memory controller doesn’t support any of the new esoteric DRAMs (SDRAM, EDO or BEDO) or synchronous cache RAMs. The PCI bus interface is fully rev2.0 compliant and runs at upto 33MHz.

The 21164 has a 128-bit data bus and supports split reads, with upto 2 reads outstanding at any time (this allows 100% data bus utilisation under best-case dream-on conditions, i.e., you can theoretically transfer 128-bits of data on every bus clock). The 21164 supports easy connection of an external 3-rd level cache (Bcache) and has all the hooks to allow external systems to maintain full cache coherence with all caches. Therefore, symmetric multiprocessor designs are ’easy’.

The 21164A was announced in October, 1995. It uses the EV56 core. It is nominally pin-compatible with the 21164, but requires split power rails; all of the power pins that were +3.3V power on the 21164 have now been split into two groups; one group provided 2.0V power to the CPU core, the other group supplies 3.3V to the I/O cells. Unlike older implementations, the 21164 pins are not 5V-tolerant. The end result of this change is that 21164 systems are, in general, not upgradeable to the 21164A (though note that it would be relatively straightforward to design a 21164A system that could also accommodate a 21164). The 21164A also has a couple of new pins to support the new 8 and 16-bit loads and stores. It also improves the 21164 support for using synchronus SRAMs to implement the external Bcache.

4.4 21064 performance vs 21066 performance

The 21064 and the 21066 have the same (EV4) CPU core. If the same program is run on a 21064 and a 21066, at the same CPU speed, then the difference in performance comes only as a result of system Bcache/memory bandwidth. Any code thread that has a high hit-rate on the *internal* caches will perform the same. There are 2 big performance killers:

1. Code that is write-intensive. Even though the 21064 and the 21066 have write buffers to swallow some of the delays, code that is write-intensive will be throttled by write bandwidth at the system bus. This arises because the on-chip caches are write-through.
2. Code that wants to treat floats as integers. The Alpha architecture does not allow register-register transfers from integer registers to floating point registers. Such a conversion has to be done via memory (And therefore, because the on-chip caches are write-through, via the Bcache). (Editor's note: it seems that both the EV4 and EV45 can perform the conversion through the primary data cache (Dcache), provided that the memory is cached already. In such a case, the store in the conversion sequence will update the Dcache and the subsequent load is, under certain circumstances, able to read the updated d-cache value, thus avoiding a costly roundtrip to the Bcache. In particular, it seems best to execute the stq/ldt or stt/ldq instructions back-to-back, which is somewhat counter-intuitive.)

If you make the same comparison between a 21064A and a 21066A, there is an additional factor due to the different Icache and Dcache sizes between the two chips.

Now, the 21164 solves both these problems: it achieve *much* higher system bus bandwidths (despite having the same number of signal pins - yes, I *know* it's got about twice as many pins as a 21064, but all those extra ones are power and ground! (yes, really!!)) and it has write-back caches. The only remaining problem is the answer to the question "how much does it cost?"

4.5 A Few Notes On Clocking

All of the current Alpha CPUs use high-speed clocks, because their microarchitectures have been designed as so-called short-tick designs. None of the sytem busses have to run at horrendous speeds as a result though:

- on the 21066(A), 21064(A), 21164 the off-chip cache (Bcache) timing is completely programmable, to the resolution of the CPU clock. For example, on a 275MHz CPU, the Bcache read access time can be controller with a resolution of 3.6ns
- on the 21066(A), the DRAM timing is completely programmable, to the resolution of the CPU clock (*not* the PCI clock, the CPU clock).
- on the 21064(A), 21164(A), the system bus frequency is a sub-multiple of the CPU clock frequency. Most of the 21064 motherboards use a 33MHz system bus clock.
- Systems that use the 21066 can run the PCI at any frequency relative to the CPU. Generally, the PCI runs at 33MHz.
- Systems that use the APECs chipset (see Section 4.6 ()) always have their CPU system bus equal to their PCI bus frequency. This means that both busses tends to run at either 25MHz or 33MHz (since these are the frequencies that scale up to match the CPU frequencies). On APECs systems, the DRAM controller timings are software programmable in terms of the CPU system bus frequency

Aside: someone suggested that they were getting bad performance on a 21066 because the 21066 memory controller was only running at 33MHz. Actually, it's the superfast 21064A systems that have memory controllers that 'only' run at 33MHz.

4.6 The chip-sets

DS sells two CPU support chipsets. The 2107x chipset (aka APECS) is a 21064(A) support chiset. The 2117x chipset (aka ALCOR) is a 21164 support chipset. There will also be 2117xA chipset (aka ALCOR 2) as a 21164A support chipset.

Both chipsets provide memory controllers and PCI host bridges for their CPU. APECS provides a 32-bit PCI host bridge, ALCOR provides a 64-bit PCI host bridge which (in accordance with the requirements of the PCI spec) can support both 32-bit and 64-bit PCI devices.

APECS consists of 6, 208-pin chips (4, 32-bit data slices (DECADE), 1 system controller (COMANCHE), 1 PCI controller (EPIC)). It provides a DRAM controller

(128-bit memory bus) and a PCI interface. It also does all the work to maintain memory coherence when a PCI device DMAs into (or out of) memory.

ALCOR consists of 5 chips (4, 64-bit data slices (Data Switch, DSW) - 208-pin PQFP and 1 control (Control, I/O Address, CIA) - a 383 pin plastic PGA). It provides a DRAM controller (256-bit memory bus) and a PCI interface. It also does all the work required to support an external Bcache and to maintain memory coherence when a PCI device DMAs into (or out of) memory.

There is no support chipset for the 21066, since the memory controller and PCI host bridge functionality are integrated onto the chip.

4.7 The Systems

The applications engineering group in DS produces example designs using the CPUs and support chipsets. These are typically PC-AT size motherboards, with all the functionality that you'd typically find on a high-end Pentium motherboard. Originally, these example designs were intended to be used as starting points for third-parties to produce motherboard designs from. These first-generation designs were called Evaluation Boards (EBs). As the amount of engineering required to build a motherboard has increased (due to higher-speed clocks and the need to meet RF emission and susceptibility regulations) the emphasis has shifted towards providing motherboards that are suitable for volume manufacture.

Digital's system groups have produced several generations of machines using Alpha processors. Some of these systems use support logic that is designed by the systems groups, and some use commodity chipsets from DS. In some cases, systems use a combination of both.

Various third-parties build systems using Alpha processors. Some of these companies design systems from scratch, and others use DS support chipsets, clone/modify DS example designs or simply package systems using build and tested boards from DS.

The EB64: Obsolete design using 21064 with memory controller implemented using programmable logic. I/O provided by using programmable logic to interface a 486<->ISA bridge chip. On-board Ethernet, SuperI/O (2S, 1P, FD), Ethernet and ISA. PC-AT size. Runs from standard PC power supply.

The EB64+: Uses 21064 or 21064A and APECs. Has ISA and PCI expansion (3 ISA, 2 PCI, one pair are on a shared slot). Supports 36-bit DRAM SIMs. ISA bus generated by Intel SaturnI/O PCI-ISA bridge. On-board SCSI (NCR 810 on PCI) Ethernet (Digital 21040), KBD, MOUSE (PS2 style), SuperI/O (2S, 1P, FD), RTC/NVRAM. Boot ROM is EPROM. PC-AT size. Runs from standard PC power supply.

The EB66: Uses 21066 or 21066A. I/O sub-system is identical to EB64+. Baby PC-AT size. Runs from standard PC power supply. The EB66 schematic was published as a marketing poster advertising the 21066 as "the first microprocessor in the world with embedded PCI" (for trivia fans: there are actually 2 versions of this poster - I drew the circuits and wrote the spiel for the first version, and some Americans mauled the spiel for the second version)

The EB164: Uses 21164 and ALCOR. Has ISA and PCI expansion (3 ISA slots, 2 64-bit PCI slots (one is shared with an ISA slot) and 2 32-bit PCI slots. Uses plus-in Bcache SIMMs. I/O sub-system provides SuperI/O (2S, 1P, FD), KBD, MOUSE (PS2 style), RTC/NVRAM. Boot ROM is Flash. PC-AT-sized motherboard. Requires power supply with 3.3V output.

The AlphaPC64 (aka Cabriolet): derived from EB64+ but now baby-AT with Flash boot ROM, no on-board SCSI or Ethernet. 3 ISA slots, 4 PCI slots (one pair are on a shared slot), uses plug-in Bcache SIMMs. Requires power supply with 3.3V output.

The AXPpci33 (aka NoName), is based on the EB66. This design is produced by Digital's Technical OEM (TOEM) group. It uses the 21066 processor running at 166MHz or 233MHz. It is a baby-AT size, and runs from a standard PC power supply. It has 5 ISA slots and 3 PCI slots (one pair are a shared slot). There are 2 versions, with either PS/2 or large DIN connectors for the keyboard.

Other 21066-based motherboards: most if not all other 21066-based motherboards on the market are also based on EB66 - there's really not many system options when designing a 21066 system, because all the control is done on-chip.

Multia (aka the Universal Desktop Box): This is a very compact pedestal desktop system based on the 21066. It includes 2 PCMCIA sockets, 21030 (TGA) graphics, 21040 Ethernet and NCR 810 SCSI disk along with floppy, 2 serial ports and a parallel port. It has limited expansion capability (one PCI slot) due to its compact size. (There is some restriction on when you can use the PCI slot, can't remember what) (Note that 21066A-based and Pentium-based Multia's are also available).

DEC PC 150 AXP (aka Jensen): This is a very old Digital system - one of the first-generation Alpha systems. It is only mentioned here because a number of these systems seem to be available on the second-hand market. The Jensen is a floorstanding tower system which used a 150MHz 21064 (later versions used faster CPUs but I'm not sure what speeds). It used programmable logic to interface a 486 EISA I/O bridge to the CPU.

Other 21064(A) systems: There are 3 or 4 motherboard designs around (I'm not including Digital *systems* here) and all the ones I know of are derived from the EB64+ design. These include:

- EB64+ (some vendors package the board and sell it unmodified); AT form-factor.
- Aspen Systems motherboard: EB64+ derivative; baby-AT form-factor.
- Aspen Systems server board: many PCI slots (includes PCI bridge).
- AlphaPC64 (aka Cabriolet), baby AT form-factor.

Other 21164(A) systems: The only one I'm aware of that isn't simply an EB164 clone is a system made by DeskStation. That system is implemented using a memory and I/O controller proprietary to Desk Station. I don't know what their attitude towards Linux is.

4.8 Bytes and all that stuff

When the Alpha architecture was introduced, it was unique amongst RISC architectures for eschewing 8-bit and 16-bit loads and stores. It supported 32-bit and 64-bit loads and stores (longword and quadword, in Digital's nomenclature). The co-architects (Dick Sites, Rich Witek) justified this decision by citing the advantages:

1. Byte support in the cache and memory sub-system tends to slow down accesses for 32-bit and 64-bit quantities.
2. Byte support makes it hard to build high-speed error-correction circuitry into the cache/memory sub-system.

Alpha compensates by providing powerful instructions for manipulating bytes and byte groups within 64-bit registers. Standard benchmarks for string operations (e.g., some of the Byte benchmarks) show that Alpha performs very well on byte manipulation.

The absence of byte loads and stores impacts some software semaphores and impacts the design of I/O sub-systems. Digital's solution to the I/O problem is to use some low-order address lines to specify the data size during I/O transfers, and to decode these as byte enables. This so-called Sparse Addressing wastes address space and has the consequence that I/O space is non-contiguous (more on the intricacies of Sparse Addressing when I get around to writing it). Note that I/O space, in this context, refers to all system resources present on the PCI and therefore includes both PCI memory space and PCI I/O space.

With the 21164A introduction, the Alpha archtecture was ECO'd to include byte addressing. Executing these new instructions on an earlier CPU will cause an OPCDEC PALcode exception, so that the PALcode will handle the access. This will have a performance impact. The ramifications of this are that use of these new instructions (IMO) should be restricted to device drivers rather than applications code.

These new byte load and stores mean that future support chipsets will be able to support contiguous I/O space.

4.9 PALcode and all that stuff

This is a placeholder for a section explaining PALcode. I will write it if there is sufficient interest.

4.10 Porting

The ability of any Alpha-based machine to run Linux is really only limited by your ability to get information on the gory details of its innards. Since there are Linux ports for the E66, EB64+ and EB164 boards, all systems based on the 21066, 21064/APECS or 21164/ALCOR should run Linux with little or no modification. The major thing that is different between any of these motherboards is the way that they route interrupts. There are three sources of interrupts:

- on-board devices
- PCI devices
- ISA devices

All the systems use an Intel System I/O bridge (SIO) to act as a bridge between PCI and ISA (the main I/O bus is PCI, the ISA bus is a secondary bus used to support slow-speed and 'legacy' I/O devices). The SIO contains the traditional pair of daisy-chained 8259s.

Some systems (e.g., the Noname) route all of their interrupts through the SIO and thence to the CPU. Some systems have a separate interrupt controller and route all PCI interrupts plus the SIO interrupt (8259 output) through that, and all ISA interrupts through the SIO.

Other differences between the systems include:

- how many slots they have
- what on-board PCI devices they have
- whether they have Flash or EPROM

4.11 More Information

All of the DS evaluation boards and motherboard designs are license-free and the whole documentation kit for a design costs about \$50. That includes all the schematics, programmable parts sources, data sheets for CPU and support chipset. The doc kits are available from Digital Semiconductor distributors. I'm not suggesting that many people will want to rush out and buy this, but I do want to point out that the information is available.

Hope that was helpful. Comments/updates/suggestions for expansion to `neal.crook@reo.mts.digital.com`Neal Crook.

4.12 References

`http://www.research.digital.com/wrl/publications/abstracts/TN-13.html`
Bill Hamburgen, Jeff Mogul, Brian Reid, Alan Eustace, Richard Swan, Mary Jo Doherty, and Joel Bartlett. *Characterization of Organic Illumination Systems*. DEC WRL, Technical Note 13, April 1989.

Index

E

F

H

I

K

L

M

N

O

P

R

S

T

U

X