Alpha Miniloader Howto

David A. Rusling, david.rusling@reo.mts.dec.com v0.81, 17th April 1996

This document describes the Miniloader, a program for Alpha based systems that can be used to initialize the machine and load Linux. The Alpha Linux Miniloader (to give it it's full name) is also known as Milo.

Contents

1 Introduction **2**

1.1 Copyright 2

1.2 New Versions of this Document 3

2 What is Milo? **3**

3 Pre-Built Standard Milo Images. **4**

4 How To Build Milo **5**

5 How To Load Milo **5**

5.1 Loading Milo from the Windows NT ARC firmware 6

5.2 Loading Milo from the Evaluation Board Debug Monitor 7

5.3 Loading Milo from a Failsafe Boot Block Floppy 8

5.4 Loading Milo from Flash 8

5.5 Loading Milo from the SRM Console 8

5.6 System Specific Information 9

5.6.1 AxpPCI33 (Noname) 9

5.6.2 AlphaPC64 (Cabriolet) 10

5.6.3 EB66+ 11

5.6.4 EB64+/Aspen Alpine 11

5.6.5 Universal Desktop Box (Multia) 11

5.6.6 EB164 12

5.6.7 XL266 12

5.6.8 Platform2000 13

6 Milo's User Interface **13**

6.1 The "help" Command 13

6.2 Booting Linux 14

6.3 Rebooting Linux 15

6.4 The "bootopt" command 15

7 Running the Flash Management Utility **15**

7.1 The "help" command 16

7.2 The "list" command 16

7.3 The "program" command 17

7.4 The "environment" command 17

7.5 The "bootopt" command 17

7.6 The "quit" command 18

8 Restrictions. **18**

9 Problem Solving. **18**

10 Acknowledgements. **19**

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.

1.1 Copyright

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 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.

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.

4 How To Build Milo

You build Milo seperately 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-1.3.76.tgz will build against linux-1.3.76.tar.gz. Milo `may` build against a higher version of the kernel, but there again it may not. In the remainder of this discussion, I assume that your kernel sources and object files are stored in the subtree at `/usr/src/linux`.

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".

```
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.

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,
- the Windows NT ARC firmware,
- Digital's SRM console,
- an Alpha Evaluation Board Debug Monitor,
- flash/ROM.

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 "äs 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 6 for more information on the commands available.

Another (rather sneaky) way of loading 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.

5.2 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 5.6.2)
- EB64+ (Section 5.6.4)
- EB66+ (Section 5.6.3)
- EB164 (Section 5.6.6)

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>
```

5.3 Loading Milo from a Failsafe Boot Block Floppy

Only the AxpPCI33 is *known* to include failsafe boot block floppy support (Section ¡ id=”noname-section” name=”Noname”¿).

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
```

5.4 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 5.6.2)
- Noname (Section 5.6.1)
- EB66+ (Section 5.6.3)
- EB164 (Section 5.6.6)

5.5 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 5.6.1)
- AlphaPC64 (Section 5.6.2)
- EB164 (Section 5.6.6)

5.6 System Specific Information

5.6.1 AxpPCI33 (Noname)

The Noname board can load Milo from the Windows NT ARC firmware (Section 5.1), from the SRM Console (Section 5.5). and from a failsafe boot block floppy (Section 5.3). A flash management utility, runnable from Milo is available so that once Milo is running, it can be blown into flash (Section 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).

5.6.2 AlphaPC64 (Cabriolet)

The AlphaPC64 includes the Windows NT ARC firmware (Section 5.1), the SRM Console (Section 5.5) and the Evaluation Debug Monitor (Section 5.2). 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 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.

5.6.3 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 5.2). 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 5.1). A flash management utility, runnable from Milo is available so that once Milo is running, it can be blown into flash (Section 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 5.6.2).

5.6.4 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 5.1). and the other contains the Evaluation Board Debug Monitor (Section 5.2).

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

5.6.5 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 5.1).

5.6.6 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 5.2). 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 5.1). The SRM console is also available (Section 5.5). A flash management utility, runnable from Milo is available so that once Milo is running, it can be blown into flash (Section 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 5.6.2).

5.6.7 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 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.

5.6.8 Platform2000

This is a 233Mhz 21066 based system.

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.

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.
  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.
```

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 recognise that it is gzip'd by the .gz suffix.

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 the NCR SCSI driver, 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 `SCSIn_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
```

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.

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.

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.

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 and the EB164 systems (and their clones).

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>
```

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.

7.4 The "environment" command

This selects a flash block to contain Milo's environment variables.

7.5 The "bootopt" command

This is just the same as Milo's "bootopt" command, see (Section 6.4).

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.

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.

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.

Several people reported having problems reading files from the floppy disk using the Evaluation Board Debug Monitor supplied with Digital Alpha Evaluation Boards such as the AlphaPC64. 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 `0>` 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.

10 Acknowledgements.

I would like to thank:

- Eric Rasmussen and Eilleen Samberg the authors of the PALcode,
- Jim Paradis for the keyboard driver and the original MILO interface,
- 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 know *david.rusling@reo.mts.dec.com* (`mailto:david.rusling@reo.mts.dec.com`) 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).