MaRTE OS Boot process (x86 architecture)

Daniel Sangorrin

daniel.sangorrin@unican.es

2006-6-20

Revision History

Revision 0.1 2006-6-20 Revised by: dsl
first draft
Revision 0.2 2007-11-23 Revised by: dsl and agc
more details of the GRUB installation

This document shows the boot process of a MaRTE OS. We will show configurations for booting from several devices.

Table of Contents

1. Preface

This document is intented for those who want to know how to boot a MaRTE OS application from several devices. Note that this manual is aimed at the x86 architecture of MaRTE OS. It is desirable that you have read *"HOWTO: Hello MaRTE OS using an emulator (QEMU)"* before reading this tutorial so you have seen at least one demo of a MaRTE OS application.

1.1. Feedback

Feedback is most certainly welcome for this document. Send your additions, comments and criticisms to the following email address : <`daniel.sangorrin@unican.es`>.

2. Inside the MaRTE OS boot process

The boot process of MaRTE OS is quite similar to other Operating Systems. Just in case you don't know anything about booting we are going to start from the very begining. In the next section, we will go more into details of how to boot from a particular device. I'd like to stress one more time that we are going to talk only about the x86 architecture of MaRTE OS. This section is an adaptation to MaRTE OS of the article [1], which explains the boot process for a Linux kernel.

2.1. The BIOS

When a MaRTE OS application is running in our system what we have is just a chunk of code and data (our **mprogram**) loaded in RAM that is being executed (at some point) by the CPU. But, how did the **mprogram** file end up in the RAM? and who gave him the control?.

When a system is first booted, or is reset, the processor executes code at a well-known location. In a PC, this location is in the basic input/output system (BIOS), which is stored in flash memory on the motherboard. The first step of the BIOS is the power-on self test (POST). The job of the POST is to perform a check of the hardware. The second step of the BIOS is local device enumeration and initialization.

The BIOS is made up of two parts: the POST code and runtime services. After the POST is complete, it is flushed from memory, but the BIOS runtime services remain and are available to the target operating system.

To boot an operating system, the BIOS runtime searches for devices that are both active and bootable in the order of preference defined by the complementary metal oxide semiconductor (CMOS) settings. The CMOS is a small memory that mantains its data thanks to a lithium battery. A boot device can be a floppy disk, a CD-ROM, a partition on a hard disk, a device on the network, or even a USB flash memory stick.

This is the **first point** where we can take part, selecting the order of preference of the boot devices. This is typically made by changing some parameters at the BIOS setup program (press "supr" when your PC starting).

For the most typical device for booting, a hard disk, the BIOS will load the first sector (512-bytes). This first sector, called MBR (Master Boot Record) contains the primary boot loader. After the MBR is loaded into RAM, the BIOS yields control to it. Take a look at the code and the raw bytes in your MBR with the following commands:

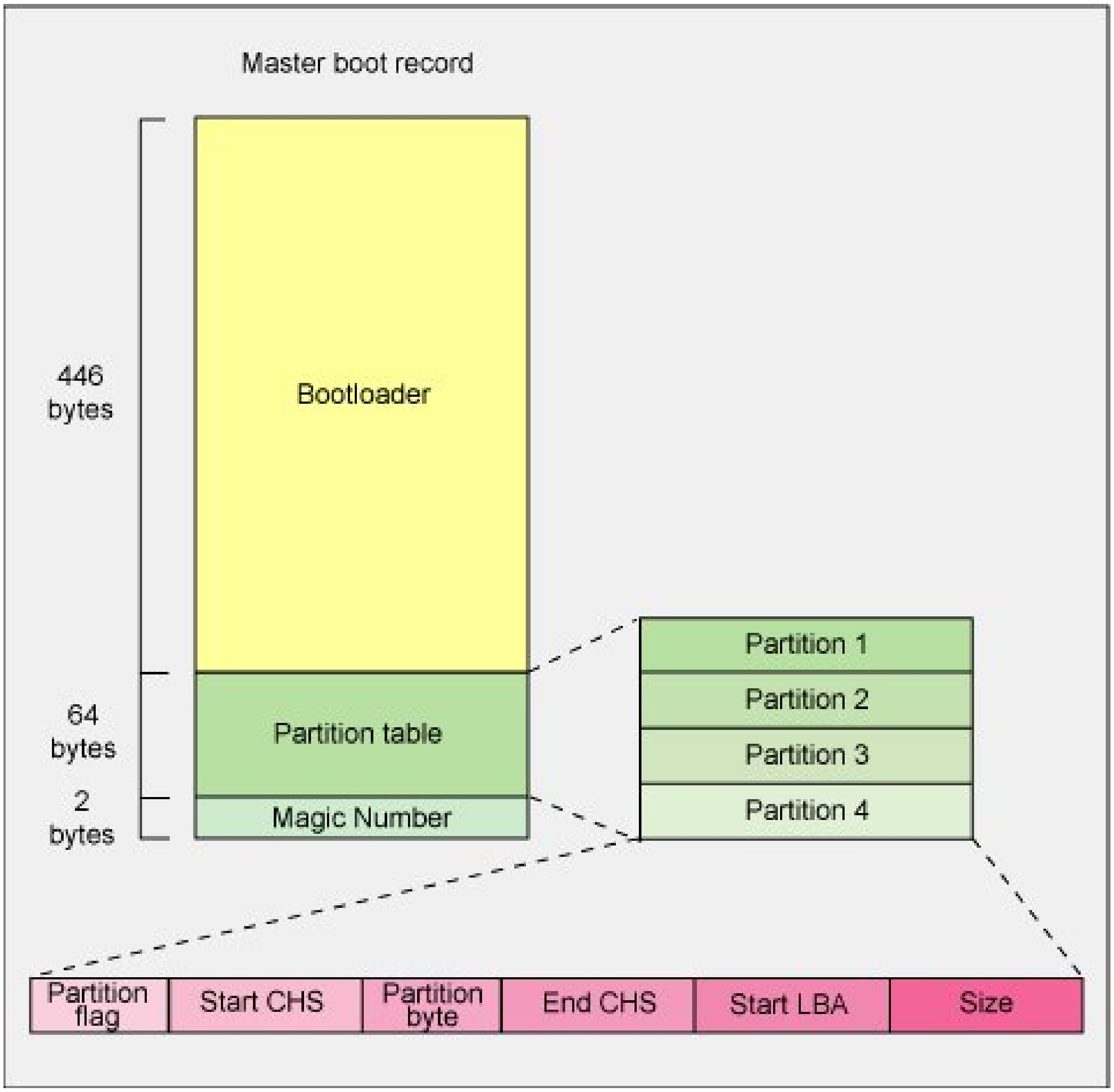
```
$ sudo apt-get install nasm
$ sudo dd if=/dev/hda bs=512 count=1 | ndisasm - | more
$ sudo dd if=/dev/hda bs=512 count=1 | od -t x1
```

This is the **second point** where we can take part, changing the contents of the MBR. We have to store a less than 512-byte chunk of code in the MBR, the *primary boot loader*!.

2.2. The Boot Loader

Ok, now we have this chunk of 512 bytes in RAM being executed. We can't do a lot with just 512 bytes so this chunk of code is in charge of loading a second-stage boot loader. Therefore a boot loader is typically composed of two stages. The job of the primary boot loader is to find and load the secondary boot loader (stage 2). It does this by looking through the partition table for an active partition. When it finds an active partition, it scans the remaining partitions in the table to ensure that they're all inactive. When this is verified, the active partition's boot sector (a boot sector is the first sector of a partition and it looks like the MBR.) is read from the device into RAM and executed.

Figure 1. Layout of the MBR ([1])



The secondary, or second-stage, boot loader could be more aptly called the kernel loader. The task at this stage is to load the MaRTE OS application. MaRTE OS is compliant with the Multiboot Specification which is a protocol between a boot loader and an OS kernel ([3]).

One of the most used boot loaders is the GRand Unified Bootloader (GRUB) boot loader ([2]) which is also multiboot compliant. The great thing about GRUB is that it includes knowledge of a lot of file systems. Instead of using raw sectors on the disk (as LILO boot loader does), GRUB can load a MaRTE OS from several file systems. It does this by making the two-stage boot loader into a three-stage boot loader. Stage 1 (MBR) boots a stage 1.5 boot loader that understands the particular file system containing the MaRTE OS image. Example: **e2fs_stage1_5** (to load from an ext2 or ext3 file system). When the stage 1.5 boot loader is loaded and running, the stage 2 boot loader can be loaded.

With stage 2 loaded, GRUB can, upon request, display a list of available kernels (defined in `/boot/grub/grub.conf` or `/boot/grub/menu.lst`). You can select a kernel and even amend it with additional kernel parameters. Optionally, you can use a command-line shell for greater manual control over the boot process:

```
press 'c'
grub> kernel /mprogram
grub> boot
```

2.3. MaRTE OS

As we said before, the second stage of the boot loader will load our MaRTE OS application, **mprogram**, in RAM and yield the control to it. Ok, but where is it loaded? The file **mprogram** is just an ELF executable file so we can view its properties with the following command:

```
$ readelf -a mprogram
```

You will see from the output that **mprogram** will be loaded at the RAM address 0x00100000. This address is set in linking time by MaRTE OS's script **mld** (which is called when you run **mgcc** and **mgnatmake**). You can change the value at `marte/utils/globasl.pl` ($ARCH_LD_OPTS=" -Ttext 100000 ";)

As any other ELF program MaRTE OS starts and ends at certain points. There is no magic in it. So let's see where MaRTE OS starts. With the following commands we compile an application with debug option and then dump its contents:

```
$ mgnatmake -g marte/examples/hello world.adb
$ objdump -d export/mprogram | more
```

```
      mprogram: file format elf32-i386
      Disassembly of section .text:
      00100000 <_start>:
         100000: eb 0e jmp 100010 <boot_entry>
            100002: 89 f6 mov %esi,%esi
      ...
```

As we said before MaRTE OS starts at 0x00100000. Thanks to the '-g' flag we can see some labels among the code that help us to understand it. As we just want to know the execution flow of MaRTE OS it is enough to know the meaning of a few x86 assembler instructions like **jmp**, **call** or **ret** (check [5] for more info). The first label is 'boot_entry' so let's look for it in the MaRTE OS hierachy:

```
$ grep -R -n -i boot_entry *
```

```
x86_arch/hwi/boot/multiboot.S:43:   jmp     boot_entry
x86_arch/hwi/boot/multiboot.S:59:   .long   boot_entry              /* entry */
x86_arch/hwi/boot/multiboot.S:62:boot_entry:
```

We find out that the first executed code is located at `x86_arch/hwi/boot/multiboot.S`. This file is written in assembler and it calls to the function **multiboot_main** located at `x86_arch/call_main/base_multiboot_main.c`.

This function is in charge of making some initialization stuff by calling other functions (it shows the welcome message of MaRTE OS! Good point to make your first hack!). Most of this code has been taken from the OSKit project ([4]). Finally, this is the last line:

```
exit(wrapper_main(argc, argv, environ));
```

wrapper_main is located either at `x86_arch/call_main/wrapper_main_c.c` or at `x86_arch/call_main/wrapper_main_ada.c` depending on the language of the application. This diference is needed because if the application is written in 'C' language, the Ada packages that export functions to 'C' need to be elaborated. This is made by calling two functions:

```
  adainit();
  ret = main(argc, argv, envp);
  adafinal();
```

From this wrapper it is called the real **main**, that is, our application!. From now on we are the owners of the computer. If our application ends, everything end!. When we create several tasks or threads there is a trick to pass the control to the kernel so it can schedule which thread is going to be executed next. The only way to do that is by means of hardware interrupts. Without them the tasks would execute over and over!. MaRTE OS needs a real-time timer in order to program it for returning the control of the CPU back to the kernel after a few ms. This is one of the most basic functionality of a kernel and in the case of MaRTE OS it is located at `Kernel.Scheduler` and `Kernel.Timed_Events_And_Timer` Ada packages.

When the application ends, it returns a value to ret and after some finalization code a message is shown at `x86_arch/hwi/boot/_exit.c`:

```
  printc(" _exit(%d) called; rebooting...\n", rc);
```

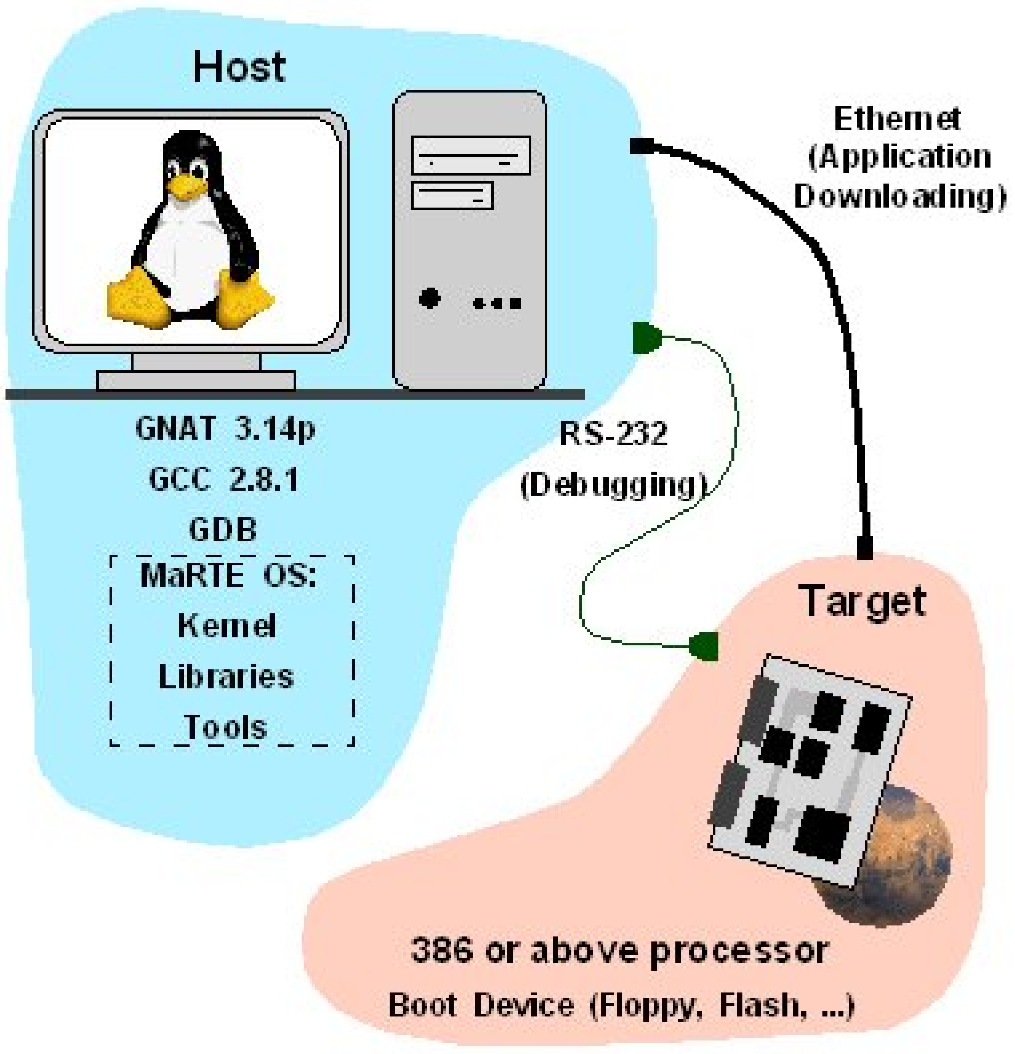
3. Multiboot Loaders

In this section we show how to configure several boot loaders and environments for booting MaRTE OS. We start with environments mostly used at the application implementation phase, and then we show the typical final environments.

3.1. Network environments

This configuration is very used in the development phase of an embedded system. While we are implementing a new application for MaRTE OS we will probably need to do a lot of tests, changes in our application and resets. If we store our **mprogram** in a hard disk we would have to rewrite it every time, then reset... a lot of time wasted. A Network environment consists of two computers, the *host* and the *target*. The *host* is the PC with Linux+GNAT+MaRTE_OS, where we program and generate the **mprogram** application. The *target* is the computer where **mprogram** must be executed. For instance, the CPU of a Robot, a PC-104 computer, or another PC. Therefore, the *target* needs to download **mprogram** from the *host*. This can be done by connecting both computers through an Ethernet network and using some *magic* :). The following picture shows you this environment. The RS-232 line can be used for debugging but we leave this for another tutorial.

Figure 2. Example of MaRTE OS environment at development phase



Ok, suppose you have your *host* with Linux connected to the *target*. You have installed MaRTE OS, compiled an application (i.e. `/marte/examples/hello_world.c`) and generated your **mprogram**. What's left? Oh, yeah, the *magic* I told you about. The *magic* you will need to use depends on the *target* capabilities (i.e. if it has a floppy, a compactflash, a PXE built-in motherboard). When the **target** computer starts or is reset it will do exactly the same steps I mentioned before. The Bios will look for a device to boot from in a certain order. This order is set in the Bios setup program. Depending on the options for booting of your *target* computer the configuration needed is different.

3.1.1. GRUB + Netboot ([6]) on a diskette (or other disk)

This configuration consists of a diskette with GRUB installed and a network loader called Netboot. Netboot loader is included in MaRTE OS (`/marte/netboot`) for several network cards. An script, **mkbootfloppy**, has been also included for recording netboot with grub on a diskette. If you want to install it on other device like a hard disk, a compactflash, etc.. the structure is almost the same. You will have to install GRUB and then add the netboot file.

In this configuration, when the *target* starts it executes GRUB loader. GRUB will load the **netboot** file (with built-in drivers for our specific NIC) in RAM and run it. Netboot will ask the network two questions through the DHCP protocol:

1. What file do I have to download?
2. Where is it?

Therefore, we have to answer those questions from our *host*. This is done by installing a DHCP daemon and configuring it:

```
/etc/dhcpd.conf:
   host mov1 {
      option host-name "mov1.ctr.unican.es";
      hardware ethernet 00:0B:AB:05:BB:B0;
      fixed-address 193.144.198.55;
      server-name "soc1.ctr.unican.es";
      next-server 193.144.198.98;
      filename "/home/user/export/mprogram";
      option root-path "/home/user/export"; }}
```

After that, the Netboot code will start downloading **mprogram** through the NFS protocol. We have to install an NFS server in our Linux and configure it:

```
/etc/exports:
   /home/tucuenta/export 193.144.198.55(ro,sync)
```

3.1.2. Etherboot ([7]) on a diskette (or other disk)

One of the main problems of using Netboot is that there is compatibility only for a few ethernet cards. Etherboot is another network loader that solves this problem because it supports a wide range of ethernet cards. You just have to go to its webpage at [7] and make a floppy bootable ROM image compatible with your card and the DHCP/NFS protocols. Then, record it on your floppy:

```
$ cat eb-5.4.2-3c509.zdsk > /dev/fd0
```

If you want to use another disk (like a hard disk or a compactflash for example) you can make a GRUB compatible ROM image (i.e. eb-5.4.2-3c509.zlilo) instead of a floppy one. The configuration for the *host* is the same as the Netboot case.

3.1.3. PXE ([8]) + Etherboot ([7])

So far, we have used a diskette (or other disk) to load first a network boot loader in charge of loading the **mprogram** file. This can be a problem when our *target* doesn't have a solid state memory available. Nowadays, a lot of computers have a built-in protocol called Preboot Execution Environment (PXE) [8]. When you select the option LAN in the BIOS setup utility, the motherboard will try to download a kernel through the network using this protocol.

PXE will start using DHCP in order to get an answer for the same questions needed to resolve. After that it will download the so called *Network Bootstrap Program* via another protocol, TFTP. In our case, this program will be an Etherboot image in PXE format (you can select it while making your ROM at [7]). Then, Etherboot will be in charge of booting **mprogram** as we have explained before. We need to install a TFTP server, configure it and put our PXE Etherboot image available.

You may have noticed that both PXE and Etherboot use DHCP at the begining of their processes so they can conflict. We need to difference them in our DHCP server in order to tell them the correct files. This is done with this configuration lines:

```
if substring (option vendor-class-identifier,0,9) = "PXEClient"
   {filename "/eb-5.0.8-rtl8139.lzpxe";}
else if substring(option vendor-class-identifier,0,9)="Etherboot"
   {filename "/home/dsl/export/mprogram";
   option root-path "/home/dsl/export";}
```

You can download the configuration files here: /etc/dhcpd.conf (dhcpd.conf), tftp, /etc/exports (exports).

3.2. Disk environments

This is the preferred method for booting in the final embedded system. It consists on installing GRUB in any device in the *target*. For example, a CompacthFlash, USB, hard disk, floppy, etc.. (I haven't tried booting from a cd-rom yet):

1. Create a partition in your device (i.e. CompactFlash) with **fdisk** or **cfdisk**. For instance, if if your device is /dev/hdc (the Secondary master):

   ```
   fdisk /dev/hdc
   ```

   In the menu, create a partition with 'n', toggle the boot flag to true, and set the filesystem to FAT16 (for example).

2. Format the created partition with the desired filesystem with **mkfs**, **mkdosfs**, ... For example:

   ```
   mkdosfs -F 16 /dev/hdc1
   ```

3. Mount the partition ('sudo mount /dev/hdc1 /mnt') and copy the GRUB loader files to your device. You can get them from your own linux at `/boot/grub`. For example:

   ```
   cp -R /boot/grub /mnt
   ```

4. Unmount the partition again and install GRUB stages on the partition with the **grub** command. Note that so far you just copied the files, you need to write the MBR of your partition. WARNING: be careful now because you must do the following operation as root and you could damage your Linux partition if you don't choose the right device. In GRUB the equivalence with Linux naming of disk devices is: hda, hdb, hdc, hdd ==> hd0, hd1, hd2, hd3 And for each partition of the disk the equivalence is: hda1, hda2, hda3.. ==> (hd0,0), (hd0,1), (hd0,2)...

   ```
   # grub
         >> root (hd1,0) <-- click on TAB to see the options
               Now you should get a message indicating the type of filesystem. If you formatted
               the partition with FAT you will see a message saying this is a FAT disk.
         >> setup (hd1) <-- WARNING: be careful you are using your device
         >> quit
   ```

5. Mount the new partition and copy your **mprogram** to it.
6. Finally, edit the GRUB configuration file `/boot/grub/grub.conf`:

   ```
   title MaRTE OS
   root (hd0,0)
   kernel /mprogram
   ```

4. Troubleshooting

1. Problems with Etherboot image

I have experienced problems with newer Etherboot images. Try using older ones.

5. Further Information

1. Inside the Linux boot process (by Tim Jones, Consultant Engineer for Emulex Corp.) http://www-128.ibm.com/developerworks/linux/library/l-linuxboot/?ca=dgr-lnxw09LinuxBoot (http://www-128.ibm.com/developerworks/linux/library/l-linuxboot/?ca=dgr-lnxw09LinuxBoot)
2. GRUB loader http://www.gnu.org/software/grub/ (http://www.gnu.org/software/grub/)
3. Multiboot Specification Manual http://www.gnu.org/software/grub/manual/multiboot/ (http://www.gnu.org/software/grub/manual/multiboot/)

4. The OSkit Project http://www.cs.utah.edu/flux/oskit/ (http://www.cs.utah.edu/flux/oskit/)
5. The Intel x86 Manuals http://www.x86.org/intel.doc/inteldocs.htm (http://www.x86.org/intel.doc/inteldocs.htm)
6. Netboot http://netboot.sourceforge.net/ (http://netboot.sourceforge.net/)
7. Etherboot http://www.etherboot.org/ (http://www.etherboot.org/)
8. Preboot Execution Environment (PXE) http://en.wikipedia.org/wiki/Pre-Boot_Execution_Environment (http://en.wikipedia.org/wiki/Pre-Boot_Execution_Environment)

6. Legal section

6.1. Copyright and License

6.2. Disclaimer