Etherboot Developers Manual

Ken Yap

Georg Baum

This document explains the internals of the Etherboot package. The information here applies to version 5.2 of Etherboot.

1. About this Developers Manual

1.1. Obtaining the most recent version of this document

This document and related documents are also kept online at the Etherboot Home Page (http://etherboot.sourceforge.net/). This will in general have the latest source distributions and documentation.

1.2. Feedback

Comments on and corrections for this Developers Manual may be directed to the primary author (mailto:ken_yapATusersPERIODsourceforgePERIODnet).

1.3. Copyrights and Trademarks

1.4. Acknowledgements and Thanks

Thanks to all the people who have contributed information and corrections to this document.

2. Characteristics of Etherboot

For an introduction to Etherboot from the user perspective you may wish to read the User Manual (../userman/t1.html) first.

In order to understand Etherboot development, it is necessary to first understand how Etherboot differs from normal application programs.

1. Etherboot runs as a standalone program, not under an operating system. The only services it relies on are those provided by the BIOS of the motherboard, or LinuxBIOS.
2. It has complete access to the "bare metal" of the machine.
3. It needs to have as small a memory footprint as practicable, in order to maximise the amount of memory available to the code that it loads.

3. The execution environment of Etherboot

3.1. The network booting process

Since this is the part that the user sees first, let us first demystify how network booting works.

From time immemorial, well actually since the IBM XT appeared on the market, the PC architecture has a mechanism for invoking "extension BIOSes". The original reason for this mechanism was to allow adaptor cards that the main BIOS didn't know how to deal with to carry ROMs with initialisation code or drivers. An early example was the XT hard disk controller. The main BIOS of XTs only knew how to boot from floppies. When an XT hard disk controller is added, the code in the ROM on the controller appears in the memory space of the PC and is called as part of the machine initialisation. Another example is the BIOSes on VGA video adaptor cards, although strictly speaking that is a special case in terms of ROM address. When network adaptors were made for the PC, it was a natural step to put ROMs on them that could contact a server for network booting.

How does the main BIOS know that the code in the ROM is to be executed and why does it not execute some random code by accident? The ROM code has several conditions placed on it.

- The ROM must start on a 2kB boundary in the memory space, between 0xC8000 and 0xEE000, although some main BIOSes scan outside these limits.

- The first two bytes of the ROM must be 55 AA hex.
- The third byte of the ROM should contain the number of bytes in the ROM code divided by 512. So if the ROM code is 16kB long, then this byte would hold 20 hex (32 decimal).
- All the bytes in the ROM (specified by the length byte just mentioned) must checksum to 8 bits of binary zero. The sum is formed by 8 bit addition of all the bytes, throwing away the carry. Note that there is not a particular location designated as the "checksum byte". Normally the ROM building process alters an unused byte somewhere to fulfil the checksum condition.

If such a ROM is detected and validated by a scan, then the main BIOS does a far call to ROMSEG:3, where ROMSEG is the segment of the ROM and 3 is the offset to transfer control to the discovered extension BIOS. Typically a network boot ROM does not take full control at this point. Instead the normal procedure to do some initialisation or probing of the hardware and then plant a vector that will be called when the BIOS is ready to boot the OS. The vector used for this purpose is normally interrupt 0x19 although interrupt 0x18 is sometimes used.

For PCI plug and play ROMs things are more complicated. For the full story, you need to get the specifications from Phoenix and Intel. Here is a quick summary.

- The boot ROM must satisfy the requirements for ROMs listed above (called legacy ROMs).
- There are two additional structures in the ROM, the PCIR structure and the PnP structure. These structures are pointed to by offsets in two 16-bit words at 0x18 and 0x1A bytes respectively from the beginning of the ROM. As a double check, the structures each begin with 4 magic bytes, PCIR and $PnP respectively.
- The PCIR structure contains the vendor and device IDs of the network adaptor, and these must match the IDs that is stored in the adaptor's PCI configuration memory, or the ROM will be ignored.
- The PnP structure contains various vectors. The one of interest to us is the Boot Execution Vector (BEV). This points to the starting point of the boot ROM code. The first time the ROM is detected, it is called at the ROMSEG+3 entry point as for legacy ROMs. This entry point must indicate, by returning 0x20 in register AX, that it is a network boot device. When the BIOS is ready to boot, it calls the BEV. Note that the BIOS only calls the BEV if the BIOS configuration specifies the device in the boot sequence.
- There is a checksum for the PnP structure in addition to the overall checksum in legacy ROMs.

The network boot process then works like this:

1. The main BIOS detects the Etherboot ROM as an extension BIOS and passes control to it with a far call.
2. For legacy ROMs, the Etherboot code hooks itself to interrupt 0x19 and returns control to the main BIOS. For PnP ROMs the Etherboot code indicates that it is a bootable device.
3. The main BIOS finishes initialising other devices and boots the operating system by calling interrupt 0x19.
4. The Etherboot code gains control.

5. It initialises the network hardware so that it is ready to send and receive packets.
6. It sends a Boot Protocol (BOOTP (http://www.ietf.org/rfc/rfc951.txt)) or Dynamic Host Configuration Protocol (DHCP (http://www.ietf.org/rfc/rfc2131.txt)) broadcast query packet. An alternative is Reverse Address Resolution Protocol (RARP (http://www.ietf.org/rfc/rfc903.txt))
7. Assuming a reply is received, the Etherboot code decodes the fields of the reply, sets its IP address and other parameters, and sends a Trivial File Transfer Protocol (TFTP (http://www.ietf.org/rfc/rfc1350.txt)) request to download the file. Be aware that the 16-bit counter field of TFTP may limit transfers to 16 MB (signed interpretation), 32 MB (unsigned interpretation) or 90 MB (unsigned, large block size option active). This is a rollover bug in many TFTP servers, but quite prevalent. An alternative loading protocol is Network File System (NFS (http://www.ietf.org/rfc/rfc1094.txt)) protocol. In this instance a mount of the remote filesystem is done (with the bare minimum of features) and the boot file is read off the filesystem.
8. The file to be loaded is in a special format, it contains a "directory" in the first block that specifies where in memory the various pieces of the file are to be loaded. Formats that are supported are tagged Appendix A or Execution and Loader Format (ELF) (http://www.acm.uiuc.edu/sigops/rsrc/pfmt11.pdf). One small extension to ELF has been made, the top bit in the e_flags longword of the ELF header has been used as ELF_PROGRAM_RETURNS_BIT, meaning that the program transferred to intends to return to Etherboot, e.g. a menu program, and that Etherboot should not disable the network interface yet. See the file `core/osloader.c`.

   Eric Biederman adds: The ELF format is documented in the SysV Generic ABI doc (http://www.sco.com/developer/devspecs/gabi41.pdf). http://www.sco.com/developer/devspecs/abi386-4.pdf Also check out the linux standard base (http://www.linuxbase.org/spec/), it has good links to all of these documents, in its related documents section.
9. Etherboot transfers control to the loaded image.

Notice no assumption was made that the image is a Linux kernel. Even though loading Linux kernels is the most common use of Etherboot, there is nothing in the procedure above that is Linux specific. By creating the loaded file appropriately, different operating systems, e.g. FreeBSD, DOS, can be loaded.

In the case of a Linux kernel, there is some additional work to be done before the kernel can be called, so the segment of the file that Etherboot transfers to is not the startup segment of the kernel, but an initial stub, whose code is in `first32.c` in the mknbi package. This stub has several tasks, which either cannot be done by Etherboot, or should not be done by Etherboot because they are Linux specific. These are: to append kernel arguments from option 129 of the BOOTP or DHCP reply; to copy and expand special kernel parameters, in particular the vga= and the ip= parameters and then to point the kernel to the location of the parameter area; to move the RAMdisk, if there is one, to the top of memory (this last cannot be done at image creation time since the size of the RAM of the machine is not known then).

The kernel parameters are passed to the kernel as a pointer to the string written in a certain location in the original bootblock (`boot.S` from the Linux kernel sources). This is a 16-bit pointer and is the offset of the parameter area from the base of the bootblock. This is one reason why the parameter area must be in the same 64kB as the bootblock. If the components of Etherboot are to be relocated elsewhere, e.g. 0x80000 upwards, then they should be relocated together. In version 0x0202 and above of the Linux setup segment, this can be passed instead as a absolute 32-bit pointer in a certain location in the setup segment. This eases the relocation requirements. The address of the RAM disk, if it exists, is passed to the kernel as a 32-bit address in a particular location in the setup segment (`setup.S` from the Linux kernel sources). This is filled in with the final location of the RAM disk after it has been moved to the top of memory.

3.2. To compress or not to compress, that is the question

We simplified things a little when we talked about how the main BIOS detects the Etherboot ROM and passes control to it. At this point the code is executing from ROM. There are two problems with executing from ROM where x86 PCs are concerned.

1. The x86 architecture does not easily support Position Independent Code (PIC). The main drawback when executing C code is referencing global entities (global and static variables). Since the ROM address is not known when building the image, addresses cannot be assigned to global entities. More advanced environments have a dynamic loader for adjusting references just before use. Etherboot has no such help. If the code were written in assembler, we could use a convention like always referencing global entities as offsets from a particular register. But we don't want to write in assembler and we don't have control over what the C compiler generates.
2. C code assumes that data locations are writable. ROM locations are not writable, by definition. One could remedy this by locating the writable entities in a RAM segment, but this causes more complication.

For the reasons above, Etherboot copies itself to the top of memory or 4 GB, whichever is the lesser, and executes there. However that is the physical address. The logical address for execution is set in the Makefile and currently is 0x20000 for the x86 version. The translation is done using the address translation facility of the MMU. This allows Etherboot global variables and functions to be assigned known addresses at the linking step.

(Actually top of memory isn't completely true. On the x86 platform it executes in an even megabyte just below the top of memory. This is normally the second last megabyte. The reason has to do with the setting of the A20 gate. If Etherboot executes in an odd megabyte it will get gated out of existence when the A20 gate is disabled. By running in an even megabyte this is avoided. Note that we still have to disable the A20 gate because some operating systems react strangely if they find it enabled when they are expecting it not to be, but now we can do it from C instead of from real mode asm running in the lower 640kB.)

This gives us about 1 MB for code, data, and stack. There is no heap as is normally understood; Etherboot does not have dynamic storage allocation. This keeps things simple, makes it a bit less prone to programming errors, and also acts as a check against unthinking use of memory by programmers. There is however a small heap which can be used to provide small allocations of memory below 640 kB.

The ROM loader copies the payload to a temporary location starting at 0x80000 and continues execution from there. The reason for this first relocation is so that execution goes faster. Typically ROM has longer access times than RAM and is often accessed 8 bits at a time. The glue chipsets used in motherboards add wait states so that the CPU can interface with relatively slow ROM. This is to reduce ROM costs, as execution speed is not important at boot time. By copying itself to RAM first, the decompression goes faster. After relocation and resumption of execution, for the x86 platform, the loader then jumps to start16, which switches into protected mode, then this jumps to start32. On other platforms, there is no need to switch modes so it jumps to the beginning of the code directly. Other loaders are used when the code is not run from ROM but they also have the ultimate action of jumping to start16.

We usually want to minimise the size of the ROM used to hold Etherboot. Even if the network adapter accommodates large ROMs, there are many claims on the area between 0xC8000 to 0xEE000, by other extension BIOSes, by peripherals that use shared memory, and so forth. Compressed images are suffixed with .z*. The payload is a .zimg file. The first part of a .zimg file is the decompressor which decompresses its own payload to the intended execution location. The decompressor is assembled from `arch/i386/prefix/unnrv2b.S`. The second part is the .img result of a compilation. In non-compressed images, the .img is the one and only part in the payload. Compressed .zrom images should work just the same as .rom images, but are smaller (but this saving may or may not bring it below a particular ROM chip capacity which are powers of two). The compressor is a C program called `util/nrv2b.c`. This is run on the host platform when building images.

Here is a diagram of a full prefix stack for the x86 architecture. Other architectures will not have the real mode to protected mode transition and will have different runtime routines and decompressor:

```
+-----------------------+
|                       |
| Payload               | *.img
+-----------------------+
|                       |
| Decompressor          | unnrv2b.S
+-----------------------+
|                       |
| PM runtime routines   | start32.S
+-----------------------+
|                       |
| RM to PM transition   | start16.S
+-----------------------+
|                       |
| Media specific loader | loader.S, *prefix.S
+-----------------------+
```

3.3. Real and protected mode

One of the complications of the x86 architecture is the existence of the real and protected modes of execution. To simplify a long story, if we want to execute 32-bit code as generated by the C compiler, we need to be in protected mode. However the processor boots up in real mode, which is 16-bit. So the loaders must execute in real mode. In addition, the main BIOS calls the extension BIOS in real mode. So at least the prologue of the main code must be in real mode. BIOS calls must also be in real mode. In Etherboot this is handled by a pair of functions (written in assembler of course) called `prot_to_real` and `real_to_prot` that do the switching. The C code doesn't call this directly. They are implicitly called from routines that use BIOS services, such as printing characters to the screen or reading characters from the keyboard. In Etherboot 5.2 and above, the 16-bit code is executed inside special segments of memory created on the fly for running real-mode code fragments, as opposed to previous versions, where they ran inline with the 32-bit code. This allows the 32-bit stack to be separate from the 16-bit one.

The file `first32` establishes a 32-bit stack which is separate from the 16-bit one and uses that thereafter.

4. The architecture of Etherboot

Leaving aside the peripheral code for the loaders, the rest of Etherboot can be divided into the core, the drivers and miscellaneous (catch-all category).

```
+-----------------+-----------------+
| Etherboot core  |  Miscellaneous  |
+-----------------+-----------------+
|          Driver support           |
+-----------------------------------+
|          Hardware drivers         |
+-----------------------------------+
|             Hardware              |
+-----------------------------------+
```

These parts are mapped on the directory structure in the `src` directory as follows:

- `arch`: Architecture specific files. Each architecture has an own subdirectory, currently we have `i386` and `ia64`.
- `bin`: The directory where all binary files are built.
- `core`: Etherboot core.
- `drivers/disk`: Drivers for floppies and hard disks.
- `drivers/net`: Drivers for NICs.
- `firmware`: BIOS-specific files.
- `include`: Include files.
- `util`: Utility programs that are needed for building, but not included in the resulting roms.

These directories are duplicated under `arch` where appropriate.

4.1. Etherboot core

The core of Etherboot handles the protocol for obtaining a network identity and for loading data over the network. This comprises the files `main.c` (main program) `config.c` (probe routines), `osloader.c` (support for various load image formats), `nfs.c, proto*.c` (support for network I/O) and `vsprintf.c` (printing routines). (More here.)

4.2. Drivers

Generally each file represents a family of network adapters. For example `tulip.c` handles all adapters that use a Tulip compatible network controller, even if they are from different manufacturers. `3c90x.c` handles a whole family of related adapters from 3Com.

The interface between the core and the drivers is well-defined and explained in Section 7.

The files `timer.h` and `timer.c` provide routines for access to the second hardware timer of the PC. This is used for implementing microsecond timeouts.

The files `pci.h` and `pci.c` provide a PCI initialisation subsystem that is executed for PCI adapters.

`skel.c` is a skeleton driver that an aspiring driver writer can use as a starting point.

4.3. Miscellaneous

In this category are all the files that don't fit into the first two categories.

4.4. External auxiliary programs

In Etherboot 5.0 and later there is the facility to load and run external auxiliary programs. As Etherboot runs in high memory. All the memory below that and above 0x10000 is fair game for loading. What if we did not load the target operating system but instead loaded an external program, one that returned to Etherboot after doing something. This something could be a fancy menu system. The advantage of this approach is that the menu system does not have to be compiled into Etherboot, which means it can be changed without changing ROMs, and can be much larger.

How does such a menu program indicate to Etherboot which image is to be loaded next? Several new features of Etherboot make this possible. Firstly, a bit in the header of the loaded image is used to indicate that the loaded program intends to return to Etherboot, as opposed to never returning, in the case of an operating system. Next, the external program is passed a pointer to the BOOTP/DHCP structure obtained by Etherboot, which includes the filename field. Finally, the external program can return one longword of status to Etherboot to indicate various conditions.

So here's how the external menu program would work. Etherboot is told to load an image which is a menu program. The program would present the user with a list of images to choose from. The presentation can be as simple as a numbered menu or as fancy as a point and click interface (if you can arrange to interface to the pointer). When a particular image has been chosen, the menu program alters the filename field in the BOOTP/DHCP structure and returns a longword of status indicating that Etherboot should retry the loading. Since the filename has been altered, Etherboot will end up loading the desired image instead of the menu this time around. Think of it as a program chaining facility.

5. The development environment of Etherboot

Etherboot is written in C, with certain routines written in x86 assembler for access to machine resources or to do things not possible in C. The tools used are the GNU C compiler and GNU x86 assembler. **ld** is used to link the object files. An auxiliary program called **makerom.pl** converts this binary into a form suitable for writing into a ROM.

A Makefile coordinates all the building procedures. However there are many network adapters that differ only in the PCI vendor and device IDs. This information must be programmed into the ROM header using **makerom**. An auxiliary script called **genrules.pl** scans all driver source files for the device names and IDs and outputs make rules to a file called `Roms` that is included by the main Makefile to generate all the ROM images. Another file included by Makefile is `Config`, which contains user configurable build parameters.

Let us take an example of a ROM image that we could build, `bin/mx98715.zrom`. According to this line in the file `tulip.c`:

```
PCI_ROM(0x10d9, 0x0531, "mx98715", "Macronix MX987x5"),
```

the driver code is in `tulip.c` and the vendor ID is 0x10d9 and the device ID is 0x0531. If one were to say **make bin/mx98715.zrom**, the following actions happen:

- Any needed utilities such as `bin/nrv2b` are built.
- `drivers/net/tulip.c` is compiled to `bin/tulip.o`.
- The startup routine, `arch/i386/core/start32.S` is assembled to `bin/start32.o`.
- All the core Etherboot files, e.g. `core/main.c`, `osloader.c`, etc, are compiled to corresponding object files in `bin/*.o` and combined into an ar archive, `bin/bootlib.a`, for convenience.
- The startup routine (which must be first in the list of objects), the driver object, `bootlib.a`, the platform specific objects are linked to produce an ELF binary, linked to run at the relocation address of Etherboot, currently 0x20000. The ELF binary is converted to an Etherboot binary using GNU utilities. This binary is can be on non-x86 platforms by conversions. (To fill in later.) For the x86 platform, we need to prepend loaders for the chosen load method.
- The prepended ROM loaders, rloader, and prloader, are produced by assembling `arch/i386/prefix/loader.S` with different defines. The meaning of the prefixes are: rloader = basic ROM loader, and prloader = PCI header basic ROM loader. The loaders contain the needed headers for the BIOS to recognise the code as an extension ROM. See Section 3 on the extension BIOS mechanism.
- For example, in the case of `bin/mx987x5.zrom`, this is for a PCI adaptor and we want a compressed ROM, so the loader is prloader. This is prepended to the Etherboot image to generate a ROM image.
- **makerom.pl** is run over this ROM image to pad it to the size of a standard EPROM. (8 kB, 16 kB, 32 kB, etc.) The PCI IDs are written into the image into the PnP structure at the right spots. A string identifying the device and Etherboot version is written into the unprogrammed bytes at the end of the ROM image, if space is available, and a pointer to that written into the PnP structure. The vendor string and the device string are normally printed by the PCI PnP BIOS at boot time. Makerom also calculates the checksum (both for the PnP structure, if present and for the whole image) and alters designated spare bytes in the image so that the checksums come out right. The image is then ready to be written into an EPROM.
- By prepending alternate loaders to the Etherboot image, we can load the image from floppy, LILO/SYSLINUX, PXE or DOS. In the case of floppy loading, the preloader is called `floppyload.bin`. It is one sector (512 bytes) long and loaded from the floppy, starting at the

beginning of the floppy. It then loads the blocks following itself, which it assumes to be an Etherboot image, into memory, then jumps to the start of it.

In the case of LILO/SYSLINUX booting, the preloader is `bin/liloprefix.bin`. It contains what looks like a floppy sector and a startup segment to LILO/SYSLINUX and makes LILO/SYSLINUX "think" that this is a Linux kernel.

- The loader for chaining from PXE is in `bin/pxeprefix.bin`, and when prepended to an image, produces a file that can be downloaded and executed by a PXE conforming boot rom as a first stage loader. Etherboot then takes over the remainder of the load process.
- Yet another preloader is not for a device but for an OS environment. This is `bin/comprefix.bin`, and when prepended to an image, makes it look like a DOS .com executable, including the peculiarity of starting at COM_SEGMENT:0x100. All it does however is jump to the Etherboot image.
- All the preceeding special preloaders have associated Makefile rules and are created by asking for images with the appropriate suffix. For example, if one wanted an image for writing onto a floppy one would say:

  ```
  make bin/mx98715.zdsk
  ```

6. Frequently asked questions

6.1. How portable is Etherboot?

Work leading up to 5.2 made Etherboot much more portable. The platforms on which Etherboot runs, other than the x86, are the Itanium and the Hammer. The portability issues can be looked at in two categories:

Support routines needed for platforms need to be written. Platform independent and dependent code have been separated and it should be straightforward to identify the support that is needed for running Etherboot on a new platform.

Byte order, operand size and alignment issues in the core routines and PCI drivers have been mostly dealt with, although there are some known places where the code has not been made portable, e.g. longword access of registers in the eepro100 driver. We hope to find these bugs as time goes by. In some cases, e.g. ISA bus NICs, there is no need to fix the problem as the ISA bus only exists on the PC platform (unless somebody attaches a NE2000 to a big-endian microcontroller and plans to run Etherboot on it).

Here are the low-level services that need to provided: writing and reading from the console, determining the size of memory, obtaining the an elapsed time, inserting a boot vector to be called, and a microsecond timer for short delays. Depending on the target environment, these services may be provided in different ways.

6.2. How can I get Etherboot to boot a Cardbus (PCMCIA) or a USB NIC?

Cardbus (PCMCIA) and USB NICs are interfaced to the CPU through their respective bus controllers. Before Etherboot can address the registers and memory on these NICs, it must initialise the bus controller appropriately. In Linux this is done by the PCMCIA and USB subsystems. In Etherboot, all this must be done by the code in the ROM. What needs to be written is a subsystem like the PCI subsystem in Etherboot. Volunteers are most welcome.

7. Writing an Etherboot Driver

7.1. Preliminaries

So Etherboot does not have a driver for your network adapter and you want to write one. You should have a good grasp of C, especially with respect to bit operations. You should also understand hardware interfacing concepts, such as the fact that the x86 architecture has a separate I/O space and that peripherals are commanded with ‘out’ instructions and their status read with ‘in’ instructions. A microprocessor course such as those taught in engineering or computer science curricula would have given you the fundamentals. (Note to educators and students in computer engineering: An Etherboot driver should be feasible as a term project for a final year undergraduate student. I estimate about 40 hours of work is required. I am willing to be a source of technical advice.)

Next you need a development machine. This can be your normal Linux machine. You need another test machine, networked to the development machine. This should be a machine you will not feel upset rebooting very often. So the reset button should be in working condition. :-) It should have a floppy drive on it but does not need a hard disk, and in fact a hard disk will slow down rebooting. Alternatively, it should have another network adapter which can netboot; see discussion further down. Needless to say, you need a unit of the adapter you are trying to write a driver for. You should gather all the documentation you can find for the hardware, from the manufacturer and other sources.

7.2. Background information

There are several types of network adapter architecture. The simplest to understand is probably programmed I/O. This where the controller reads incoming packets into memory that resides on the adapter and the driver uses ‘in’ instructions to extract the packet data, word by word, or sometimes byte by byte. Similarly, packets are readied for transmission by writing the data into the adapter’s memory using ‘out’ instructions. This architecture is used on the NE2000 and 3C509. The disadvantage of this architecture is the load on the CPU imposed by the I/O. However this is of no import to Etherboot (who cares how loaded the CPU is during booting), but will be to Linux. Next in the sophistication scale are shared memory adapters such as the Western Digital or SMC series, of which the WD8013 is a good example. Here the adapter’s memory is also accessible in the memory space of the main CPU. Transferring data between the driver and the adapter is done with memory copy instructions. Load on the CPU is light. Adapters in this category are some of the best performers for the ISA bus. Finally there are bus mastering cards such as the Lance series for the ISA bus and practically all good PCI adapters (but not the NE2000 PCI). Here the data is transferred between the main memory and the adapter controller using Direct Memory Access. Setting up the transfers usually involves a sequence of operations with the registers of the controller.

7.3. Structure of the code

Examine the file `skel.c`, in the src directory, which is a template for a driver. You may also want to examine a working driver. You will see that an Etherboot driver requires 4 functions to be provided:

- A probe routine, that determines if the network adapter is present on the machine. This is passed a pointer to a ‘nic’ struct, possibly a list of candidate addresses to probe, and possibly a pointer to a ‘pci’ struct. This routine should initialise the network adapter if present. If a network adapter of the type the driver can handle is found, it should save the I/O address at which it was found for use by the other routines. In the case of ISA adapters, it may be passed a list of addresses to try, or if no list is passed in, it may use an internal list of candidate addresses. In the case of PCI adapters, the address has already been found by the PCI support routines. Then it should determine the Ethernet (MAC) address of the adapter and save it in nic->node_addr. It should then initialise the adapter. Finally it should fill in the function pointers for the other routines, and return the ‘nic’ pointer. If it fails to find an adapter, it should return 0.

  The field rom_info in the ‘nic’ struct contains, on entry to the probe routine, a pointer to the structure rom_info, defined in `etherboot.h`. This contains the segment address and length in shorts of the ROM as detected by the BIOS. These can be used by the driver to discriminate between multiple instances of the same network adaptor on the bus, and choose to activate only the one the ROM is installed on. For example, the 3c509 hardware has a register indicating what address the ROM is mapped at. Note that the driver should not reject "ROM segment addresses" outside 0xC000 to 0xEE00 as this indicates booting from floppy disk or other non-ROM media.

  Initialising the adapter means programming the registers so that the chip is ready to send and receive packets. This includes enabling the appropriate hardware interface (10B2, 10BT) in the case of adapters with more than one interface, and setting the right speed (10Mb, 100Mb) if the hardware does not autosense and set it. It also includes setting up any memory buffers needed by the hardware, along with any necessary pointers.

  Note that you should program the receiver registers to allow broadcast Ethernet packets to be received. This is needed because other IP hosts will do an ARP request on the diskless computer when it boots.

- A disable routine, which puts the adapter into a disabled state. This is passed a pointer to a ‘nic’ struct. This is needed to leave the adapter in a suitable state for use by the operating system which will be run after Etherboot. Some adapters, if left in an active state, may crash the operating system at boot time, or cannot be found by the operating system. This can be called from the probe routine.
- A transmit routine, to send an Ethernet packet. This is passed a pointer to a ‘nic’ struct, the 6 byte Ethernet address of the destination, a packet type (IP, ARP, etc), the size of the data payload in bytes, and a pointer to the data payload. Remember the packet type and length fields are in x86 byte order (little-endian) and the adapter’s byte order may be the reverse (big-endian). Note that the routine knows nothing about IP (or any other type) packets, the data payload is assumed to be a filled in packet, ready to transmit.

- A poll routine, to check if a packet has been received and ready for processing. This is passed a pointer to a 'nic' struct. If a packet is available, it should copy the packet from the adapter into the data area pointed to by nic->packet, and set nic->packetlen to the length of the data, and return 1, otherwise 0.

  A few Ethernet controller chips will receive packets from itself, as detected by having a source address of itself. You can throw these out immediately on reception and not bother the upper layer with them.

No routine needs to be public, all routines should be static and private to the driver module. Similarly all global data in the driver should be static and private.

If the NIC is a PCI adapter, create a struct pci_driver (as in `skel.c`) and an array of struct pci_id:

```
static struct pci_id skel_nics[] = {
PCI_ROM(0x0000, 0x0000, "skel-pci", "Skeleton PCI Adaptor"),
};
```

Fill the pci_id array with one entry for each combination of pci vendor id and pci device id that your driver can handle. PCI_ROM is a a macro defined in `include/pci.h`. The arguments have the following meaning: vendor id, device id, rom name and short description. Since this information is also used to build the Makefile rules, you must use the PCI_ROM macro and can't fill in the values directly. Both the pci vendor and device id must be given in hex form, no define is allowed. Additionally PCI_ROM must occur only once in a line and one macro call must not span more than one line. You can obtain the vendor and device ids from the file `/usr/include/linux/pci.h`. It is also displayed by PCI BIOSes on bootup, or you can use the **lspci** program from the pciutils package to discover the ids.

If the NIC is an ISA adapter, create a struct isa_driver (as in `skel.c`) and one line like the following:

```
ISA_ROM("skel-isa", "Skeleton ISA driver")
```

in your driver source. The ISA_ROM macro is like the PCI_ROM without the vendor/device ids. The same rules about formatting as in the PCI case apply.

Only for special cases where the automatic generation of build rules via the PCI_ROM and ISA_ROM entries does not work, add an entry to the here document in `genrules.pl` so that the build process will create Makefile rules for it in the file `bin/Roms`.

The above mentioned structs and macros hold all information that etherboot needs about your driver. In case you wonder how this works at all although everything is declared static: The special build process that is used by Etherboot (including linker scripts and some Perl magic) packs the necessary information into public segments.

The Etherboot build process places a few restrictions on your driver: If you need more than one .c file, the main file (that will contain the PCI_ROM or ISA_ROM macro call) must contain #include directives for the other files. They must not contain a PCI_ROM or ISA_ROM call. See `drivers/net/prism*` for an example.

7.3.1. Rom naming rules

Currently there is no official rom naming convention in etherboot. Use some descriptive name, but note that two ore more consecutive hyphens (like in "my--rom") are not allowed, since "--" is the delimiter sign for multiple-driver-roms. Sometimes it is difficult to find a sensible name, for example for "NICs" that are built in motherboard chipsets or if you don't know the model name. In this case we choose to name the corresponding roms "driver-deviceid", like "eepro100-1035". Of course you have to make sure that your rom name is unique in etherboot.

7.4. Booting the code from a floppy

Use the rule for bin/driver.fd0 to write another instance of the driver to the floppy for testing. Use lots of printf statements to track where execution has reached and to display the status of various variables and registers in the code. You should expect to do this dance with the development machine, floppy disk and target machine many many times.

7.5. Booting the test code with another Etherboot ROM

There is another method of testing ROM images that does not involve walking a floppy disk between the machines and is much nicer. Set up a supported NIC with a boot ROM. Put the target NIC on the same machine but at a non-conflicting I/O location. That is to say, your test machine has two NICs and two connections to the LAN. Then when you are ready to test a boot image, use the utility mknbi-rom (../mknbi.html) to create a network bootable image from the ROM image, and set up bootpd/DHCPD and tftpd to send this over the when the machine netboots. Using Etherboot to boot another version of itself is rather mind-boggling I know.

7.6. Writing the code

First set up the various required services, i.e. BOOTP/DHCP, tftp, etc. on the development machine. You should go through the setup process with a supported adapter card on a test machine so that you know that the network services are working and what to expect to see on the test machine.

If you are starting from a Linux driver, usually the hardest part is filtering out all the things you do not need from the Linux driver. Here is a non-exhaustive list: You do not use interrupts. You do not need more than one transmit buffer. You do not need to use the most efficient method of data transfer. You do not need to implement statistics counting. In general it is a good idea to use the latest Linux driver as base, because it usually supports newer cards and has more bugs fixed than older versions. If the driver is written by Donald Becker, it is probably best to start from the version available on this page (http://www.scyld.com/page/support/network/), since the driver in the official kernel may be behind. See also Section 7.9.

Generally speaking, the probe routine is relatively easy to translate from the Linux driver. The exception is when you need to handle media and speed switching. The transmit is usually straightforward, and the receive a bit more difficult. The main problem is that in the Linux driver, the work is split between routines called from the kernel and routines triggered by hardware interrupts. As mentioned before, Etherboot does not use interrupts so you have to bring the work of transmitting and receiving back into the main routines. The disable routine is straightforward if you have the hardware commands.

When coding, first get the probe routine working. You will need to refer to the programmer's guide to the adapter when you do this. You can also get some information by reading a Linux or FreeBSD driver. You may also need to get the disable routine working at this time.

Next, get the transmit routine working. To check that packets are going out on the wire, you can use tcpdump or ethereal on the development machine to snoop on the Ethernet. The first packet to be sent out by Etherboot will be a broadcast query packet, on UDP port 67. Note that you do not need interrupts at all. You should ensure the packet is fully transmitted before returning from this routine. You may also wish to implement a timeout to make sure the driver doesn't get stuck inside transmit if it fails to complete. A couple of timer routines are available for implementing the timeout, see `timer.h`. You use them like this (in pseudo-code):

```
for (load_timer2(TIMEOUT_VALUE);
 transmitter_busy && (status = timer2_running()); )
 ;
if (status == 0)
 transmitter_timed_out;
```

The timeout value should be 1193 per millisecond of wait. The maximum value is 65535, which is about 54 milliseconds of timeout. If you just need to delay a short time without needing to do other checks during the timeout, you can call waiton_timer2(TIMEOUT_VALUE) which will load, then poll the timer, and return control on timeout.

Please do not use counting loops to implement time-sensitive delays. Such loops are CPU speed dependent and can fail to give the right delay period when run on a faster machine.

Next, get the receive routine working. If you already have the transmit routine working correctly you should be getting a reply from the BOOTP/DHCP server. Again, you do not need interrupts, unlike drivers from Linux and other operating systems. This means you just have to read the right register on the adapter to see if a packet has arrived. Note that you should NOT loop in the receive routine until a packet is received. Etherboot needs to do other things so if you loop in the poll routine, it will not get a chance to do those tasks. Just return 0 if there is no packet awaiting and the main line will call the poll routine again later.

Finally, get the disable routine working. This may simply be a matter of turning off something in the adapter.

7.7. Things to watch out for

Things that may complicate your coding are constraints imposed by the hardware. Some adapters require buffers to be on word or doubleword boundaries. See `rtl8139.c` for an example of this. Some adapters need a wait after certain operations.

7.8. Tidying up

When you get something more or less working, release early. People on the mailing lists can help you find problems or improve the code. Besides you don't want to get run over by a bus and then the world never gets to see your masterpiece, do you? :-)

Your opus should be released under GPL, BSD or a similar Open Source license, otherwise people will have problems using your code, as most of the rest of Etherboot is GPLed.

7.9. Keeping your driver in sync with the Linux version

Most Etherboot drivers are derived from their Linux counterparts. Sooner or later the Linux driver where you started from will get updated. This may include bugfixes and support for new NICs. Unfortunately there is no way to keep the Etherboot driver automatically up to date, but if you keep the following rules in mind while porting it can be made easier:

- State the version of the Linux driver in a comment in your Etherboot driver. This makes it easy to check to which Linux driver the Etherboot driver corresponds
- Etherboot drivers should be small. Often a Linux driver contains way too much code, but in many cases you can reuse some utility functions from the Linux driver without blowing up the code size too much. In this case, keep the order in which they are defined where possible, and avoid code reformatting in general. (re-indenting is ok, because **diff** can ignore it with the -b switch). This makes it easy to compare later versions with utilities like **mgdiff**.
- You can get the pci ids of all NIC drivers of a particular Linux kernel with the utility **get-pci-ids** in `util` and compare them to the entries in `bin/NIC` in order to find out wether the Linux kernel supports some new NICs.

8. A potted history of Etherboot

In Linux circles Netboot appeared first. According to the docs Jamie Honan was the person who coded up the first version and specified the format of the tagged image files. This version used assembler code taken from packet drivers (http://www.crynwr.com) to interface to the hardware, only Western Digital (now SMC) NICs in the first instance. It also required a DOS environment to compile. Later on Gero Kuhlmann took over the development of Netboot and made tremendous improvements to it. Among other things he created a harness that would simulate just enough of a DOS environment so that unmodified packet driver binaries could be used in a boot ROM. This allows any NIC on the market that has a packet driver to be used immediately. He also migrated the development to a Linux (Unix) platform.

Etherboot was ported from FreeBSD by Markus Gutschke. He made it compile under Linux and added code to support tagged images in addition to NFS boot. Since tagged images are a more general mechanism and requires less boot rom code, this has become the preferred loading method. Markus has also coded most of the additional features between 2.0 and 3.0, such as additional bootp tags, ANSI screen escapes, etc. Many of the features common to Etherboot and Netboot, such as the tagged image format, the support programs such as mknbi, and support in the Linux kernel for diskless booting, are by Gero or Markus.

Ken Yap (http://www.geocities.com/ken_yap_aus/) came to Etherboot a bit later. His original objective was to produce a 16 bit version that could be used to netboot ELKS and other OSes on older CPUs. As these things happen, he got enticed into improving the code, doing structural rearrangement, and merging contributions from others, and is now the primary maintainer of Etherboot.

In the early days, the Etherboot web page was hosted by the Sydney Linux Users Group (http://www.slug.org.au/) web site. Quite coincidentally and unrelatedly, Jamie Honan is one of the founding members of SLUG, so the story has come a full circle here. Since April 2000, Etherboot has been hosted at Sourceforge (http://sourceforge.net/). Sourceforge has provided superb facilities for hosting community Open Source development.

A. Draft Net Boot Image Proposal 0.3, June 15, 1997

Jamie Honan and Gero Kuhlmann, gero AT minix PERIOD han PERIOD de

In order to provide more functionality to the boot rom code Jamie's draft has been changed a little bit. All of Gero Kuhmnann's changes are preceded and followed by *(gk)*. All of Ken Yap's changes are preceded and followed by *(ky)*.

A.1. Preamble - the why

Whilst researching what other boot proms do (at least those implementing TCP/IP protocols) it is clear that each 'does their own thing' in terms of what they expect in a boot image.

If we could all agree on working toward an open standard, O/S suppliers and boot rom suppliers can build their products to this norm, and be confident that they will work with each other.

This is a description of how I will implement the boot rom for Linux. I believe it to be flexible enough for any OS that will be loaded when a PC boots from a network in the TCP/IP environment.

It would be good if this could be turned into some form of standard.

This is very much a first draft. I am inviting comment.

The ideas presented here should be independant of any implementation. In the end, where there is a conflict between the final of this draft, and an implementation, this description should prevail.

The terms I use are defined at the end.

*(gk)*IMPORTANT NOTE: The scope of this document starts at the point where the net boot process gains control from the BIOS, to where the booted image reaches a state from which there is no return to the net boot program possible.*(gk)*

A.2. The target

The target is to have a PC retrieve a boot image from a network in the TCP/IP environment.

*(gk)*The boot may take place from a network adaptor rom, from a boot floppy.*(gk)*

A.3. Net Boot Process Description

*(gk)*The net boot process is started as a result of the PC boot process. The net boot program can reside on a rom, e.g. on an adaptor card, or in ram as a result of reading off disk.*(gk)*

The boot process may execute in any mode (e.g. 8086, 80386) it desires. When it jumps to the start location in the boot image, it must be in 8086 mode and be capable of going into any mode supported by the underlying processor.

The image cannot be loaded into address spaces below 10000h, or between A0000h through FFFFFh, or between 98000h through 9FFFFh. *(gk)*Only when the image is not going to return to the boot process, all the memory is available to it once it has been started, so it can relocate parts of itself to these areas.*(gk)*

The boot process must be capable of loading the image into all other memory locations. Specifically, where the machine supports this, this means memory over 100000h.

The net boot process must execute the bootp protocol, followed by the tftp protocol, as defined in the relevant rfc's.

The file name used in the tftp protocol must be that given by the bootp record.

If less than 512 bytes are loaded, the net boot process attempts to display on the screen any ascii data at the start of the image. The net boot process then exits in the normal manner. For a boot prom, this will allow normal disk booting. *(gk)*Reference to DOS deleted.*(gk)*

When the first 512 bytes have been loaded, the boot process checks for an initial magic number, which is defined later. If this number is present, the net process continues loading under the control of the image format. The image, which is described later, tells the net boot process where to put this record and all subsequent data.

If no initial magic number is present the net boot process checks for a second magic number at offset 510. If the magic number 510 = 55h, 511 = AAh, then the net process continues. If this second magic number is not present, then the net boot process terminates the tftp protocol, displays an error message and exits in the normal manner.

If no initial magic number is present and the second one is, the net boot process relocates the 512 bytes to location 7c00h. The net boot process continues to load any further image data to 10000h up. This data can overwrite the first 512 boot bytes. If the image reaches 98000h, then any further data is continued to be loaded above 100000h. When all the data has been loaded, the net boot process jumps to location 0:7c00.

*(gk)*When the net boot program calls the image, it places 2 far pointers onto the stack, in standard intel order (e.g. segment:offset representation). The first far pointer which immediately follows the return address on the stack, points to the loaded boot image header. The second far pointer which is placed above the first one, shows to the memory area where the net boot process saved the bootp reply.

If the boot image is flagged as being returnable to the boot process, the boot program has to provide the boot image with interrupt vector 78h. It's an interface to services provided by the net boot program (see below for further description).

If the boot image is not flagged as being returnable to the boot process, before the boot image is called, the boot program has to set the system into a state in which it was before the net boot process has started.*(gk)*

A.4. Image Format with Initial Magic Number.

The first 512 bytes of the image file contain the image header, and image loading information records. This contains all the information needed by the net boot process as to where data is to be loaded.

The magic number (in time-honoured tradition (well why not?)) is:

```
0 = 36h
1 = 13h
2 = 03h
3 = 1Bh
```

Apart from the two magic numbers, all words and double words are in PC native endian.

Including the initial magic number the header record is:

```
+---------------------+
|                     |
| Initial Magic No.   |  4 bytes
+---------------------+
|                     |
| Flags and length    |  double word
+---------------------+
|                     |
| Location Address    |  double word in ds:bx format
+---------------------+
|                     |
| Execute Address     |  double word in cs:ip format
+---------------------+
```

The Location address is where to place the 512 bytes. The net boot process does this before loading the rest of the image. The location address cannot be one of the reserved locations mentioned above, but must be an address lower than 100000h.

The rest of the image must not overwrite these initial 512 bytes, placed at the required location. The writing of data by the net boot process into these 512 bytes is deprecated. These 512 bytes must be available for the image to interogate once it is loaded and running.

The execute address is the location in cs:ip of the initial instruction once the full image has been loaded. This must be lower than 100000h, since the initial instructions will be executed in 8086 mode. When the jump (actually a far call) is made to the boot image, the stack contains a far return address, with a far pointer parameter above that, pointing to the location of this header.

(ky) If bit 31 in the flags is set, then the execute address is interpreted as a linear 32-bit address, and a call is made to this address. There is no restriction on the range of the execute address. The arguments to the routine are: a pointer to an Etherboot specific header, a pointer to the tagged image header, and a pointer to the bootp reply. The called routine may return to the boot loader.*(ky)*

The flags and length field is broken up in the following way:

Bits 0 to 3 (lowest 4 bits) define the length of the non-vendor header in double words. Currently the value is 4.

Bits 4 to 7 define the length required by the vendor extra information in double words. A value of zero indicates no extra vendor information.

*(gk)*Bit 8 is set if the boot image can return to the net boot process after execution. If this bit is not set the boot image does never return to the net boot process, and the net boot program has to set the system into a clean state before calling the boot image.*(gk)*

*(ky)*Bit 31 is set if the execute address of the boot image is a linear 32-bit address to be called. The boot image may return to the boot loader.*(ky)*

*(gk+ky)*Bits 9 to 30 are reserved for future use and must be set to zero.*(gk+ky)*

After this header, and any vendor header, come the image loading information records. These specify where data is to be loaded, how long it is, and communicates to the loaded image what sort of data it is.

The format of each image loading information record is :

```
+---------------------+
| Flags, tags and     |  double word
| lengths             |
+---------------------+
|                     |
| Load Address        |  double word
+---------------------+
|                     |
| Image Length        |  double word
+---------------------+
|                     |
| Memory Length       |  double word
+---------------------+
```

Each image loading information record follows the previous, or the header.

The memory length, image length and load address fields are unsigned 32 numbers. They do not have the segment:offset format used by the 8086.

The flags, tags and lengths field is broken up as follows:

Bits 0 to 3 (lowest 4 bits) are the length of the non vendor part of this header in double words. Currently this value is 4.

Bits 4 to 7 indicate the length of any vendor information, in double words.

Bits 8 to 15 are for vendor's tags. The vendor tag is a private number that the loaded image can use to determine what sort of image is at this particular location.

Bits 16 to 23 are for future expansion and should be set to zero.

Bits 24 to 31 are for flags, which are defined later.

Vendors may place further information after this information record, and before the next. Each information record may have a different vendor length.

There are two restrictions on vendor information.

One is that the header and all information records that the net boot process is to use fall within the first 512 bytes.

The second restriction is that the net boot process must ignore all vendor additions. The net boot process may not overwrite vendor supplied information, or other undefined data in the initial 512 bytes.

The flags are used to modify the load address field, and to indicate that this is the last information record that the net boot process should use.

Bit 24 works in conjunction with bit 25 to specify the meaning of the load address.

```
  B24    B25

   0      0    load address is an absolute 32 number

   1      0    add the load address to the location one past the last byte
       of the memory area required by the last image loaded.
       If the first image, then add to 512 plus the location
       where the 512 bytes were placed

   0      1    subtract the load address from the one past the
       last writeable location in memory. Thus 1 would
       be the last location one could write in memory.

   1      1    load address is subtracted from the start of
       the last image loaded. If the first image, then
       subtract from the start of where the 512 bytes were
       placed
```

(For convenience bit 24 is byte 0 of the flag field)

Bit 26 is the end marker for the net boot process. It is set when this is the last information record the net boot process should look at. More records may be present, but the net boot process will not look at them. (Vendors can continue information records out past the 512 boundary for private use in this manner).

The image length tells the net boot process how many bytes are to be loaded. Zero is a valid value. This can be used to mark memory areas such as shared memory for interprocessor communication, flash eproms, data in eproms.

The image length can also be different from the memory length. This allows decompression programs to fluff up the kernel image. It also allows a file system to be larger then the loaded file system image.

Bits 27 through 31 are not defined as yet and must be set to zero until they are.

A.5. Boot prom entry points.

*(gk)*As mentioned above the net boot process has to provide interrupt 78h as an entry point in case, the returnable flag (bit 9 of the flags field in the image header) of the boot image has been set. When calling this interface interrupt, the caller has to load the AH register with a value indicating the type of operation requested:

```
00h  -  Installation check
       Input:  none
```

```
       Output: AX  -  returns the value 474Bh
        BX  -  flags indicating what further services are
        provided by the net boot program:
         Bit 0 - packet driver interface (see below)
         Bits 1 to 15 are unused and have to be zero

01h  -  Cleanup and terminate the boot process services. This will
       also remove the services provided by interrupt 87h.
       Input:  none
       Output: none
```

Further functions are not yet defined. These functions are only available to boot images which have the first magic number at the beginning of the image header, and have the returnable flag set in the flags field.

In order to provide compatibility with net boot programs written to match an earlier version of this document, the loaded image should check for the existence of interrupt 78h by looking at it's vector. If that's 0:0, or if it does not return a proper magic ID after calling the installation check function, the boot image has to assume that the net boot program does not support this services interrupt.

If the bit 0 of register BX of function 00h is set, the boot program has to provide a packet driver (http://www.crynwr.com) interface at interrupt 79h as described in the packet driver interface standard, version 1.09, published by FTP Software, Inc., which is not repeated here. It serves as an interface to the system's network card. It is important to note that the net boot process has to provide a clean packet driver interface without any handles being defined when the boot image gets started. It is expected that the boot image sets up it's own TCP/IP or other network's stack on top of this packet driver interface. When the boot image returns to the net boot process, it has to return a clean packet driver interface as well, without any handles being defined.*(gk)*

A.6. Example of a boot image.

Here is an example of how the boot image would look for Linux:

```
0x1B031336,  /* magic number */
0x4,         /* length of header is 16 bytes, no vendor info */
0x90000000,  /* location in ds:bx format */
0x90000200,  /* execute address in cs:ip format */

      /* 2048 setup.S bytes */
0x4,        /* flags, not end, absolute address, 16 bytes this
  record, no vendor info */
0x90200,     /* load address - note format */
0x800,       /* 4 8 512 byte blocks for linux */
0x800,

      /* kernel image */
0x4,        /* flags, not end, absolute address, 16 bytes this
  record, no vendor info */
0x10000,     /* load address - note format */
0x80000,     /* 512K (this could be shorter */
0x80000,
```

```
      /* ramdisk for root file system */
0x04000004,  /* flags = last, absolute address, 16 bytes this
  record, no vendor info *//
0x100000,    /* load address - in extended memory */
0x80000,     /* 512K for instance */
0x80000,
      /* Then follows linux specific information */
```

A.7. Terms

When I say 'the net boot process', I mean the act of loading the image into memory, setting up any tables, up until the jump to the required location in the image.

The net booting program executes the net boot process. The net boot program may be a rom, but not neccassarily. It is a set of instructions and data residing on the booting machine.

The image, or boot image, consists of the data loaded by the net boot process.

When I say 'the PC boot process', I mean the general PC rom bios boot process, the setting up of hardware, the scanning for adaptor roms, the execution of adaptor roms, the loading in of the initial boot track. The PC boot process will include the net boot process, if one is present.

When I say client, I mean the PC booting up.

When I say 'image host', I mean the host where the boot image is comming from. This may not have the same architecture as the client.

The bootp protocol is defined in RFC951 and RFC1084. The tftp protocol is defined in RFC783. These are available on many sites.

A bootp server is the machine that answers the bootp request. It is not neccessarily the image host.

"Can" and "may" means doesn't have to, but is allowed to and might. "Must" means just that. "Cannot" means must not.

A.8. References

Stevens, W.R 1990, Unix Network Programming, Prentice Hall, Englewood Cliffs, N.J., 1990

B. Compression algorithm, 1 July 1997

Markus Gutschke, gutschk AT math PERIOD uni-muenster PERIOD de

The compressor achieves an average compression rate of 60% of the original size which is on par with "gzip". It seems that you cannot do much better for compressing compiled binaries. This means that the break even point for using compressed images is reached, once the uncompressed size approaches 1.5kB.

We can stuff more than 12kB into an 8kB EPROM and more than 25kB into an 16kB EPROM. As there is only 32kB of RAM for both the uncompressed image and its BSS area, this means that 32kB EPROMs will hardly ever be required.

The compression algorithm uses a 4kB ring buffer for buffering the uncompressed data. Before compression starts, the ring buffer is filled with spaces (ASCII character 0x20). The algorithm tries to find repeated input sequences of a maximum length of 60 bytes. All 256 different input bytes plus the 58 (60 minus a threshold of 2) possible repeat lengths form a set of 314 symbols. These symbols are adaptively Huffman encoded. The algorithm starts out with a Huffmann tree that assigns equal code lengths to each of the 314 symbols (slightly favoring the repeat symbols over symbols for regular input characters), but it will be changed whenever the frequency of any of the symbols changes. Frequency counts are kept in 16bit words until the total number of compressed codes totals 2^15. Then, all frequency counts will be halfed (rounding to the bigger number). For unrepeated characters (symbols 0..255) the Huffman code is written to the output stream. For repeated characters the Huffmann code, which denotes the length of the repeated character sequence, is written out and then the index in the ring buffer is computed. From this index, the algorithm computes the offset relative to the current index into the ring buffer. Thus, for typical input data, one would expect that short to medium range offsets are more frequent than extremely short or medium range to long range offsets. Thus the 12bit (for a 4kB buffer) offset value is statically Huffman encoded using a precomputed Huffman tree that favors those offset values that are deemed to be more frequent. The Huffman encoded offset is written to the output data stream, directly following the code that determines the length of repeated characters.

This algorithm, as implemented in the C example code, looks very good and its operating parameters are already well optimized. This also explains why it achieves compression ratios comparable with "gzip". Depending on the input data, it sometimes excells considerably beyond what "gzip -9" does, but this phenomenon does not appear to be typical. There are some flaws with the algorithm, such as the limited buffer sizes, the adaptive Huffman tree which takes very long to change, if the input characters experience a sudden change in distribution, and the static Huffman tree for encoding offsets into the buffer. The slow changes of the adaptive Huffman tree are partially counteracted by artifically keeping a 16bit precision for the frequency counts, but this does not come into play until 32kB of compressed data is output, so it does not have any impact on our use for "etherboot", because the BOOT Prom does not support uncompressed data of more then 32kB (c.f. doc/spec.doc).

Nonetheless, these problems do not seem to affect compression of compiled programs very much. Mixing object code with English text, would not work too well though, and the algorithm should be reset in between. Actually, we might gain a little improvement, if text and data segments were compressed individually, but I have not experimented with this option, yet.

C. Copying conditions of the compression code, 1 July 1997

Markus Gutschke, gutschk AT math PERIOD uni-muenster PERIOD de

The compression code as implemented in "lzhuf.c" was taken from a BBS program written by Joachim Schurig <jschurig AT zedat PERIOD fu-berlin PERIOD de>. He states that the code can be used freely for programs that are covered by a "freeware" license. This probably includes both BSD style licenses and the GPL.

The code in "zloader.asm" is a reimplementation of the uncompressor. It has been written from scratch and is hereby placed under the conditions of the GNU General Public License (GPL). The algorithm is outlined in Appendix B.

Thus, there are no copyright problems with using this code, but there still might be difficulties with software patents. These patents are not legal in most parts of the world, but if you live in a country that honors software patents then you should verify that using these algorithms is legally permitted. Unless you are absolutely sure, that there are no legal obstacles, you should use the code for educational purposes only (this assumes that your educational institution is exempted from patent laws). The author cannot be held responsible for using the program code in violation of applicable local laws.

If you are aware of patents that might affect the legality of using the code in some parts of the world, please let me know.