The GRUB Manual

The GRand Unified Bootloader, version 0.5.97, 22 October 2000.

Gordon Matzigkeit
OKUJI Yoshinori

Part I: The Tutorial Manual

1 Overview

Briefly, a *boot loader* is the first software program that runs when a computer starts. It is responsible for loading and transferring control to an operating system *kernel* software (such as the Linux or GNU Hurd kernel). The kernel, in turn, initializes the rest of the operating system (e.g. a GNU system).

GNU GRUB is a very powerful boot loader, which can load a wide variety of free operating systems, as well as proprietary operating systems with chain-loading[1]. GRUB is designed to address the complexity of booting a personal computer; both the program and this manual are tightly bound to that computer platform, although porting to other platforms may be addressed in the future.

One of the important features in GRUB is flexibility; GRUB understands filesystems and kernel executable formats, so you can load an arbitrary operating system the way you like, without recording the physical position of your kernel on the disk.

Thus you can load the kernel just by specifying its filename and the drive (and the partition) where the kernel resides. To let GRUB know the drive and the filename, you can either type in them manually via the command-line interface (see Section 12.1 [Command line], page 19), or use the nice menu interface (see Section 12.2 [Menu interface], page 20) through which you can easily select which OS it boots. To allow you to customize the menu, GRUB will load a preexisting configuration file (see Chapter 5 [Configuration], page 8). Note that you can not only enter the command-line interface whenever you like, but also you can edit specific menu entries prior to using them.

In the following chapters, you will learn how to specify a drive or a partition, and a file name (see Chapter 2 [Naming convention], page 1) to GRUB, how to install GRUB on your drive (see Chapter 3 [Installation], page 2), and how to boot your OSes (see Chapter 4 [Booting], page 5), step by step.

2 Naming convention

The device syntax used in GRUB is a wee bit different from what you may have seen before in your operating system(s), and you need to know it so that you can specify a drive/partition.

Look at the following examples and explanations:

```
(fd0)
```

First of all, GRUB requires that the device name is enclosed with '`(`' and '`)`'. The '`fd`' part means that it is a floppy disk. The number '`0`' is the drive number, which is counted from *zero*. This expression means that GRUB will use the whole floppy disk.

---

[1] *chain-load* is the mechanism for loading unsupported operating systems by loading another boot loader. It is typically used for loading DOS or Windows.

```
(hd0,1)
```

Here, '`hd`' means it is a hard disk drive. The first integer '`0`' indicates the drive number, that is, the first hard disk, while the second integer, '`1`', indicates the partition number (or the PC slice number in the BSD terminology). Once again, please note that the partition numbers are counted from *zero*, not from one. This expression means the second partition of the first hard disk drive. In this case, GRUB uses one partition of the disk, instead of the whole disk.

```
(hd0,4)
```

This specifies the first *extended partition* of the first hard disk drive. Note that the partition numbers for extended partitions are counted from '`4`', regardless of the actual number of primary partitions on your hard disk.

```
(hd1,a)
```

This means the BSD '`a`' partition of the second hard disk. If you need to specify which PC slice number should be used, use something like this: '`(hd1,0,a)`'. If the PC slice number is omitted, GRUB searches for the first PC slice which has a BSD '`a`' partition.

Of course, to actually access the disks or partitions with GRUB, you need to use the device specification in a command, like '`root (fd0)`' or '`unhide (hd0,2)`'. To help you find out which number is a partition you want, the GRUB command-line (see Section 12.1 [Command line], page 19) options have argument completion. That means that, for example, you only need to type '`root (`', followed by a ⟨TAB⟩, and GRUB will display the list of drives, partitions, or filenames, so it should be quite easy to determine the name of your target partition, even with minimal knowledge of the syntax.

Note that GRUB does *not* distinguish IDE from SCSI - it simply counts the drive numbers from zero, regardless of their type. Normally, any IDE drive number is less than any SCSI drive number, although that is not true if you change the boot sequence by swapping IDE and SCSI drives in your BIOS.

Now the question is, how to specify a file? Again, see this example:

```
(hd0,0)/vmlinuz
```

This specifies the file named '`vmlinuz`', found on the first partition of the first hard disk drive. Note that the argument completion works with file names, too.

That was easy, admit it. Do read the next chapter, to find out how to actually install GRUB on your drive.

3 Installation

First, you need to have GRUB itself properly installed on your system, (see Appendix B [Obtaining and Building GRUB], page 65) either from the source tarball, or as a package for your OS.

To use GRUB, you need to install it on your drive. There are two ways of doing that - either using the utility `grub-install` (see Chapter 16 [Invoking grub-install], page 37) on a UNIX-like OS, or by using the native Stage 2. These are quite similar, however, the utility might probe a wrong BIOS drive, so better be careful.

Also, if you install GRUB on a UNIX-like OS, please make sure that you have an emergency boot disk ready, so that you can rescue your computer if, by any chance, your hard drive becomes unusable (unbootable).

GRUB comes with boot images, which are normally installed in the '`/usr/share/grub/i386-pc`' directory. You need to copy the files '`stage1`', '`stage2`', and '`*stage1_5`' to the directory '`/boot/grub`'.

3.1 Creating a GRUB boot floppy

To create a GRUB boot floppy, you need to take the files '`stage1`' and '`stage2`' from '`/boot/grub`' directory, and write them to the first and the second block of the floppy disk, respectively.

> **Caution:** This procedure will destroy any data currently stored on the floppy.

On a UNIX-like operating system, that is done with the following commands:

```
# cd /boot/grub
# dd if=stage1 of=/dev/fd0 bs=512 count=1
1+0 records in
1+0 records out
# dd if=stage2 of=/dev/fd0 bs=512 seek=1
153+1 records in
153+1 records out
#
```

The device filename may be different. Consult the manual for your OS.

3.2 Installing GRUB natively

> **Caution:** Installing GRUB's stage1 in this manner will erase the normal boot-sector used by an OS.

GRUB can currently boot GNU Mach, Linux, FreeBSD, NetBSD, and OpenBSD directly, so using it on a boot sector should be okay. But generally, it would be a good idea to back up the first sector of the partition on which you are installing GRUB's stage1. This isn't as important if you are installing GRUB on the first sector of a hard disk, since it's easy to reinitialize it (e.g. by running '`FDISK /MBR`' from DOS).

If you decide to install GRUB in the native environment, which is definitely desirable, you'll need to create the GRUB boot disk, and reboot your computer with it. Otherwise, see Section 3.3 [Installing GRUB using grub-install], page 4, for more details.

Once started, GRUB will show the command-line interface (see Section 12.1 [Command line], page 19). First, set the GRUB's *root device*[2] to the partition which has your GRUB images, like this:

```
grub> root (hd0,0)
```

If you are not sure which partition actually holds these files, use the `find` command, like this:

---

[2] Note that GRUB's root device doesn't necessarily mean your OS's root partition; if you need to specify a root partition for your OS, add the argument into the command `kernel`.

```
grub> find /boot/grub/stage1
```

This will search for the filename '`/boot/grub/stage1`' and show the devices which contain the file.

Once you've set the root device correctly, run the command `setup`:

```
grub> setup (hd0)
```

This command will install GRUB on the MBR (see Chapter 27 [MBR], page 56) in the first drive. If you want to install GRUB into the *boot sector* of a partition instead of the MBR, specify a partition into which you want to install GRUB:

```
grub> setup (hd0,0)
```

If you install GRUB into a partition or a drive other than the first one, you must chain-load GRUB from another boot loader. Refer to the manual for the boot loader to know how to chain-load GRUB.

Now you can boot GRUB without a GRUB floppy. See the chapter Chapter 4 [Booting], page 5 to find out how to boot your operating systems from GRUB.

3.3 Installing GRUB using grub-install

Unfortunately, if you do want to install GRUB under a UNIX-like OS (such as GNU), invoke the program `grub-install` as the superuser (*root*).

The usage is basically very easy. You only need to specify one argument to the program, namely, where to install GRUB. The argument can be either of a device file or a GRUB's drive/partition. So, this will install GRUB into the MBR of the first IDE disk under Linux:

```
# grub-install /dev/hda
```

Likewise, under Hurd, this has the same effect:

```
# grub-install /dev/hd0
```

If it is the first BIOS drive, this is the same as well:

```
# grub-install '(hd0)'
```

But all the above examples assume that you use GRUB images under the root directory. If you want GRUB to use images under a directory other than the root directory, you need to specify the option '`--root-directory`'. The typical usage is that you create a GRUB boot floppy with a filesystem. Here is an example:

```
# mke2fs /dev/fd0
# mount -t ext2 /dev/fd0 /mnt
# grub-install --root-directory=/mnt '(fd0)'
# umount /mnt
```

Another example is in case that you have a separate boot partition which is mounted at '`/boot`'. Since GRUB is a boot loader, it doesn't know anything about mountpoints at all. Thus, you need to run `grub-install` like this:

```
# grub-install --root-directory=/boot /dev/hda
```

By the way, as you may already know, it is quite difficult to guess BIOS drives correctly under a UNIX-like OS. Thus, `grub-install` will prompt you to check if it could really guess the correct mappings, after the installation. The format is defined in Section 15.3 [Device map], page 36. Please be careful enough. If the output is wrong, it is unlikely that your computer can boot with no problem.

Note that `grub-install` is actually just a shell script and the real task is done by the grub shell `grub` (see Chapter 15 [Invoking the grub shell], page 35). Therefore, you may run `grub` directly to install GRUB, without using `grub-install`. Don't do that, however, unless you are very familiar with the internals of GRUB. Installing a boot loader on a running OS may be dangerous.

4 Booting

For Multiboot-compliant kernels, GRUB can load them in a consistent way, but, for some free operating systems, you need to use some OS-specific magic.

4.1 How to boot operating systems

GRUB has two distinct boot methods. One of the two is to load an operating system directly, and the other is to chain-load another boot loader which then will load an operating system actually. Generally speaking, the former is desirable, because you don't need to install or maintain other boot loaders and GRUB is flexible enough to load an operating system from an arbitrary disk/partition. However, the latter is sometimes required, since GRUB doesn't support all the existing operating systems natively.

4.1.1 How to boot an OS directly by GRUB

Multiboot (see section "Motivation" in *The Multiboot Specification*) is the native format supported by GRUB. For the sake of convenience, there are also support for Linux, FreeBSD, NetBSD and OpenBSD. If you want to boot other operating systems, you will have to chain-load them (see Section 4.1.2 [Chain-loading], page 5).

Generally, GRUB can boot any Multiboot-compliant OS in the following steps:

1. Set GRUB's root device to the drive where the OS images are stored by the command `root`.
2. Load the kernel image by the command `kernel`.
3. If you need modules, load them with the command `module` or `modulenounzip`.
4. Run the command `boot`.

Linux, FreeBSD, NetBSD and OpenBSD can be booted in a similar manner. You can load a kernel image by the command `kernel` and then run the command `boot`. If the kernel requires some parameters, just append the parameters to `kernel`, after the filename of the kernel. Also, please refer to Section 4.2 [OS-specific notes], page 6, for the information on your OS-specific issues.

4.1.2 Load another boot loader to boot unsupported operating systems

If you want to boot an unsupported operating system (e.g. Windows 95), chain-load a boot loader for the operating system. Normally, the boot loader is embedded in the *boot sector* of the partition on which the operating system is installed.

First, set GRUB's root device to the partition by the command `rootnoverify`:

```
grub> rootnoverify (hd0,0)
```

Second, set the *active* flag in the partition by the command `makeactive`[3]:

```
grub> makeactive
```

Third, load the boot loader by the command `chainloader`:

```
grub> chainloader +1
```

'`+1`' indicates that GRUB should read one sector from the start of the partition. The complete description about this syntax can be found in Chapter 11 [Filesystem], page 18. If this succeeds, run the command `boot`.

However, DOS and Windows have a deficiency, so you might have to use more complicated instructions. See Section 4.2 [OS-specific notes], page 6, for more information.

4.2 Some caveats on OS-specific issues

Here, we describe some caveats on several operating systems.

4.2.1 GNU/Hurd

Since GNU/Hurd is Multiboot-compliant, it is easy to boot it; there is nothing special about it. But do not forget that you have to specify a root partition to the kernel.

1. Set GRUB's root device to the same drive as GNU/Hurd's. Probably the command `find /boot/gnumach` or similar can help you.
2. Load the kernel and the module, like this:
   ```
   grub> kernel /boot/gnumach root=hd0s1
   grub> module /boot/serverboot
   ```
3. Run the command `boot`.

4.2.2 GNU/Linux

It is relatively easy to boot GNU/Linux from GRUB, because it somewhat resembles to boot a Multiboot-compliant OS.

1. Set GRUB's root device to the same drive as GNU/Linux's. Probably the command `find /vmlinuz` or similar can help you.
2. Load the kernel:
   ```
   grub> kernel /vmlinuz root=/dev/hda1
   ```
   If you need to specify some kernel parameters, just append them to the command. For example, to set '`vga`' to '`ext`', do this:
   ```
   grub> kernel /vmlinuz root=/dev/hda1 vga=ext
   ```
   See the documentation in the Linux source tree for the complete information on the available options.
3. If you use an initrd, execute the command `initrd` after `kernel`:
   ```
   grub> initrd /initrd
   ```
4. Finally, run the command `boot`.

---

[3] This is not necessary for most of the modern operating systems.

Caution: If you use an initrd and specify the '`mem=`' option to the kernel, to let it use less than actual memory size, you will also have to specify the same memory size to GRUB. To let GRUB know the size, run the command `uppermem` *before* loading the kernel. See Section 13.3 [Command-line and menu entry commands], page 26, for more information.

4.2.3 FreeBSD

GRUB can load the kernel directly, either in ELF or a.out format. But this is not recommended, since FreeBSD's bootstrap interface sometimes changes heavily, so GRUB can't guarantee to pass kernel parameters correctly.

Thus, we'd recommend loading the very flexible loader '`/boot/loader`' instead. See this example:

```
grub> root (hd0,a)
grub> kernel /boot/loader
grub> boot
```

4.2.4 NetBSD

GRUB can load NetBSD a.out and ELF directly, follow these steps:

1. Set GRUB's root device with `root`.
2. Load the kernel with `kernel`. You should append the ugly option '`--type=netbsd`', if you want to load an ELF kernel, like this:

   ```
   grub> kernel --type=netbsd /netbsd-elf
   ```
3. Run `boot`.

For now, however, GRUB doesn't allow you to pass kernel parameters, so it may be better to chain-load it instead, for more information please see Section 4.1.2 [Chain-loading], page 5.

4.2.5 OpenBSD

The booting instruction is exactly the same as for NetBSD (see Section 4.2.4 [NetBSD], page 7).

4.2.6 DOS/Windows

GRUB cannot boot DOS or Windows directly, so you must chain-load them (see Section 4.1.2 [Chain-loading], page 5). However, their boot loaders have some critical deficiencies, so it may not work to just chain-load them. To overcome the problems, GRUB provides you with two helper functions.

If you have installed DOS (or Windows) on a non-first hard disk, you have to use the disk swapping technique, because that OS cannot boot from any disks but the first one. The workaround used in GRUB is the command `map` (see Chapter 13 [Commands], page 21), like this:

```
grub> map (hd0) (hd1)
grub> map (hd1) (hd0)
```

This performs a *virtual* swap between your first and second hard drive.

Caution: This is effective only if DOS (or Windows) uses BIOS to access the swapped disks. If that OS uses a special driver for the disks, this probably won't work.

Another problem arises if you installed more than one set of DOS/Windows onto one disk, because they could be confused if there are more than one primary partitions for DOS/Windows. Certainly you should avoid doing this, but there is a solution if you do want to do so. Use the partition hiding/unhiding technique.

If GRUB *hides* a DOS (or Windows) partition, it ignores the partition. If GRUB *unhides* a DOS (or Windows) partition, it detects the partition. Thus, if you have installed DOS (or Windows) on the first partition and the second of the first hard disk, and boot the first copy, do the following:

```
grub> unhide (hd0,0)
grub> hide (hd0,1)
grub> rootnoverify (hd0,0)
grub> chainloader +1
grub> makeactive
grub> boot
```

4.2.7 SCO UnixWare

It is known that the signature in the boot loader for SCO UnixWare is wrong, so you will have to specify the option '`--force`' to `chainloader`, like this:

```
grub> rootnoverify (hd1,0)
grub> chainloader --force +1
grub> makeactive
grub> boot
```

5 Configuration

You probably noticed that you need to type several commands to boot your OS. There's a solution to that - GRUB provides a menu interface (see Section 12.2 [Menu interface], page 20) from which you can select an item (using arrow keys) that will do everything to boot an OS.

To enable the menu, you need a configuration file, '`/boot/grub/menu.lst`'. We'll analyze an example file.

The file first contains some general settings, the menu interface related options. You can put these commands (see Section 13.1 [Menu-specific commands], page 21) before any of the items (starting with `title`).

```
#
# Sample boot menu configuration file
#
```

As you may have guessed, these lines are comments. Lines starting with a hash character ('`#`'), and blank lines, are ignored by GRUB.

```
# By default, boot the first entry.
default 0
```

The first entry (here, counting starts with number zero, not one!) will be the default choice.

```
# Boot automatically after 30 secs.
timeout 30
```

As the comment says, GRUB will boot automatically in 30 seconds, unless interrupted with a keypress.

```
# Fallback to the second entry.
fallback 1
```

If, for any reason, the default entry doesn't work, fall back to the second one (this is rarely used, for obvious reasons).

Note that the complete descriptions of these commands, which are menu interface specific, can be found in see Section 13.1 [Menu-specific commands], page 21. Other descriptions can be found in Chapter 13 [Commands], page 21.

Now, on to the actual OS definitions. You will see that each entry begins with a special command, `title`, and the action is described after it. Note that there is no command `boot` at the end of each item. That is because GRUB automatically executes `boot` if it loads other commands successfully.

The argument for the command `title` is used to display a short title/description of the entry in the menu. Since `title` displays the argument as is, you can write basically anything in there.

```
# For booting the GNU Hurd
title  GNU/Hurd
root   (hd0,0)
kernel /boot/gnumach.gz root=hd0s1
module /boot/serverboot.gz
```

This boots GNU/Hurd from the first hard disk.

```
# For booting Linux
title  GNU/Linux
kernel (hd1,0)/vmlinuz root=/dev/hdb1
```

This boots GNU/Linux, but from the second hard disk.

```
# For booting Mach (getting kernel from floppy)
title  Utah Mach4 multiboot
root   (hd0,2)
pause  Insert the diskette now^G!!
kernel (fd0)/boot/kernel root=hd0s3
module (fd0)/boot/bootstrap
```

This boots Mach with a kernel on a floppy, but the root filesystem at hd0s3. It also contains a `pause` line, which will cause GRUB to display a prompt and delay, before actually executing the rest of the commands and booting.

```
# For booting FreeBSD
title  FreeBSD
root   (hd0,2,a)
kernel /boot/loader
```

This item will boot FreeBSD kernel loaded from the '`a`' partition of the third PC slice of the first hard disk.

```
# For booting OS/2
title OS/2
root  (hd0,1)
makeactive
# chainload OS/2 bootloader from the first sector
chainloader +1
# This is similar to "chainload", but loads a specific file
#chainloader /boot/chain.os2
```

This will boot OS/2, using a chain-loader.

```
# For booting Windows NT or Windows95
title Windows NT / Windows 95 boot menu
root        (hd0,0)
makeactive
chainloader +1
# For loading DOS if Windows NT is installed
# chainload /bootsect.dos
```

The same as the above, but for Windows.

```
# For installing GRUB into the hard disk
title Install GRUB into the hard disk
root    (hd0,0)
setup   (hd0)
```

This will just (re)install GRUB onto the hard disk.

```
# Change the colors.
title Change the colors
color light-green/brown blink-red/blue
```

In the last entry, the command `color` is used, to change the menu colors (try it!). This command is somewhat special, because it can be used both in the command-line and in the menu (see Section 13.2 [Command-line and menu commands], page 22). GRUB has several such commands, see the user reference.

We hope that you now understand how to use the basic features of GRUB. To learn more about GRUB, see the following chapters and the User Reference Manual (see Chapter 9 [Introduction], page 14).

6 Downloading OS images from a network

Although GRUB is a disk-based boot loader, it does provide network support. To use the network support, you need to enable at least one network driver in the GRUB build process. For more information please see '`netboot/README.netboot`' in the source distribution.

6.1 How to set up your network

GRUB requires a server that will assign an IP address to the machine on which GRUB is running, and a file server. The former is either BOOTP, DHCP or a RARP server (RARP is deprecated, since it cannot serve much information). For the latter, only TFTP is supported at the moment. It is not necessary to run both the servers on one computer.

How to configure these servers is beyond the scope of this document, so please refer to the manuals specific to those protocols/servers.

Once you have set up the servers, run `bootp`, `dhcp` or `rarp` for BOOTP, DHCP or RARP, respectively. Each command will show an assigned IP address, a netmask, an IP address for your TFTP server and a gateway. If any of the addresses is wrong or it causes an error, probably the configuration of your servers isn't set up properly.

Finally, download your OS images from your network. The network can be accessed using the network drive '`(nd)`'. Everything else is very similar to the normal instructions (see Chapter 4 [Booting], page 5).

Here is an example:

```
grub> bootp
Probing... [NE*000]
NE2000 base ...
Address: 192.168.110.23    Netmask: 255.255.255.0
Server: 192.168.110.14     Gateway: 192.168.110.1

grub> root (nd)
grub> kernel /tftproot/gnumach.gz root=sd0s1
grub> module /tftproot/serverboot.gz
grub> boot
```

6.2 Booting from a network

It is sometimes very useful to boot from a network, especially, when you use a machine which has no local disk. In this case, you need to obtain a kind of Net Boot ROM, such as a PXE ROM or a free software package like Etherboot. Such a Boot ROM first boots the machine, sets up the network card installed into the machine, and downloads a second stage boot image from the network. Then, the second image will try to boot an operating system from the network actually.

GRUB provides two second stage images, '`nbgrub`' and '`pxegrub`'. Those images are the same as the normal Stage 2, except that they set up a network automatically, and try to load a configuration file from the network, if specified. The usage is very simple: If the machine has a PXE ROM, use '`pxegrub`'. If the machine has a NBI loader such as Etherboot, use '`nbgrub`'. There is no difference between them but their formats. As how to load a second stage image you want to use should be described in the manual on your Net Boot ROM, please refer to the manual, for more details. The topic is beyond the scope of this documentation.

However, there is one thing specific to GRUB. Namely, how to specify a configuration file in a BOOTP/DHCP server. For now, GRUB uses the tag '`150`', to get the name of a configuration file. This below is an example about a BOOTP configuration:

```
.allhost:hd=/tmp:bf=null:\
        :ds=145.71.35.1 145.71.32.1:\
        :sm=255.255.254.0:\
        :gw=145.71.35.1:\
        :sa=145.71.35.5:

foo:ht=1:ha=63655d0334a7:ip=145.71.35.127:\
        :bf=/nbgrub:\
        :tc=.allhost:\
        :T150="/tftpboot/menu.lst.foo":
```

See the manual about your BOOTP/DHCP server, for more information. The exact syntax should differ from the example, more or less.

7 Using GRUB via a serial line

This chapter describes how to use the serial terminal support in GRUB.

If you have many computers or computers with no display/keyboard, it would be very useful to control the computers with serial communications. To connect a computer with another via a serial line, you need to prepare a null-modem (cross) serial cable, and you may need to have multiport serial boards, if your computer doesn't have extra serial ports. In addition, a terminal emulator is also required, such as minicom. Refer to a manual of your operating system, for more information.

As for GRUB, the instruction to set up a serial terminal is quite simple. First of all, make sure that you haven't specified the option '`--disable-serial`' to the configure script when you built your GRUB images. If you get them in binary form, probably they have serial terminal support already.

Then, initialize your serial terminal after GRUB starts up. Here is an example:

```
grub> serial --unit=0 --speed=9600
grub> terminal serial
```

The command `serial` initializes the serial unit 0 with the speed 9600bps. The serial unit 0 is usually called '`COM1`', so, if you want to use COM2, you must specify '`--unit=1`' instead. This command accepts many other options, so please refer to Chapter 13 [Commands], page 21, for more details.

The command `terminal` chooses which type of terminal you want to use. In that case above, the terminal will be a serial terminal, but you can also pass `console` to the command, like '`terminal serial console`'. In this case, a terminal in which you press any key will be selected as a GRUB terminal.

However, note that GRUB assumes that your terminal emulator is compatible with VT100 by default. This is true for most terminal emulators nowadays, but you should pass the option '`--dumb`' to the command, if your terminal emulator is not VT100-compatible or implements few VT100 escape sequences. If you specify the option, then GRUB doesn't provide you with the menu interface, because the menu requires several fancy features for your terminal. Instead, GRUB only gives you the hidden menu interface and the command-line interface.

8 Security

You may be interested in how to prevent ordinary users from doing whatever they like, if you share your computer with other people. So this chapter describes how to improve the security of GRUB.

One thing which could be a security hole is that the user can do too many things with GRUB, because GRUB allows to modify its configuration and run arbitrary commands at run-time. For example, the user can read even '`/etc/passwd`' in the command-line interface by the command `cat`. So it is necessary to disable all the interactive operations.

Thus, GRUB provides *password* feature, so that only administrators can start the interactive operations (i.e. editing menu entries and entering the command-line interface). To use this feature, you need to run the command `password` in your configuration file, like this:

```
password --md5 PASSWORD
```

If this is specified, GRUB disallows any interactive control, until you press the key ⟨p⟩ and enter a correct password. The option '`--md5`' tells GRUB that '`PASSWORD`' is in MD5 format. If it is omitted, GRUB assumes the '`PASSWORD`' is in clear text.

You can encrypt your password with the command `md5crypt`. For example, run the grub shell (see Chapter 15 [Invoking the grub shell], page 35), and enter your password:

```
grub> md5crypt
Password: **********
Encrypted: $1$U$JK7xFegdxWH6VuppCUSIb.
```

Then, cut and paste the encrypted password to your configuration file.

Also, you can specify an optional argument to `password`. See this example:

```
password PASSWORD /boot/grub/menu-admin.lst
```

In this case, GRUB will load '`/boot/grub/menu-admin.lst`' as the configuration file when you enter the valid password.

Another thing which may be dangerous is that any user can choose any menu entry. Usually, this wouldn't be problematic, but you might want to permit only administrators to run some of your menu entries, such as an entry for booting an insecure OS like DOS.

GRUB provides the command `lock`. This command always fails until you enter a valid password, so you can use it, like this:

```
title Boot DOS
lock
rootnoverify (hd0,1)
makeactive
chainload +1
```

You should insert `lock` right after `title`, because any user can execute commands in an entry, until GRUB encounters `lock`.

You can also use the command `password` instead of `lock`. In this case the boot process will ask for the password and stop if it was entered incorrectly. Since the `password` takes its own *PASSWORD* argument this is useful if you want different passwords for different entries.

Part II: The User Reference Manual

9 Introduction

This part documents the user-visible aspect of GRUB. If you are looking for the information on the internals, see the Programmer Reference Manual (see Chapter 19 [Hacking], page 39).

9.1 History of GRUB

GRUB originated in 1995 when Erich Boleyn was trying to boot the GNU Hurd with the University of Utah's Mach 4 microkernel (now known as GNU Mach). Erich and Brian Ford designed the Multiboot Specification (see section "Motivation" in *The Multiboot Specification*), because they were determined not to add to the large number of mutually-incompatible PC boot methods.

Erich then began modifying the FreeBSD boot loader so that it would understand Multiboot. He soon realized that it would be a lot easier to write his own boot loader from scratch than to keep working on the FreeBSD boot loader, and so GRUB was born.

Erich added many features to GRUB, but other priorities prevented him from keeping up with the demands of its quickly-expanding user base. In 1999, Gordon Matzigkeit and OKUJI Yoshinori adopted GRUB as an official GNU package, and opened its development by making the latest sources available via anonymous CVS. See Appendix B [Obtaining and Building GRUB], page 65, for more information.

9.2 GRUB features

The primary requirement for GRUB is that it be compliant with the *Multiboot Specification*, which is described in section "Motivation" in *The Multiboot Specification*.

The other goals, listed in approximate order of importance, are:

- Basic functions must be straightforward for end-users.
- Rich functionality to support kernel experts and designers.
- Backward compatibility for booting FreeBSD, NetBSD, OpenBSD, and Linux. Proprietary kernels (such as DOS, Windows NT, and OS/2) are supported via a chain-loading function.

Except for specific compatibility modes (chain-loading and the Linux *piggyback* format), all kernels will be started in much the same state as in the Multiboot Specification. Only kernels loaded at 1 megabyte or above are presently supported. Any attempt to load below that boundary will simply result in immediate failure and an error message reporting the problem.

In addition to the requirements above, GRUB has the following features (note that the Multiboot Specification doesn't require all the features that GRUB supports):

Multiple Executable Formats

Supports many of the *a.out* variants plus *ELF*. Symbol tables are also loaded.

Support Non-Multiboot Kernels

Supports many of the various free 32-bit kernels that lack Multiboot compliance (primarily FreeBSD, NetBSD, OpenBSD, and Linux). Chain-loading of other boot loaders is also supported.

Load Multiples Modules

GRUB fully supports the Multiboot feature of loading multiple modules.

Configuration File

Supports a human-readable text configuration file with preset boot commands. The list of commands (see Chapter 13 [Commands], page 21) are a superset of those supported on the command line. An example configuration file is provided in Chapter 5 [Configuration], page 8.

Menu Interface

A menu interface listing the preset boot commands, with a programmable timeout, is available. There is no fixed limit on the number of boot entries, and the current implementation has space for several hundred.

Flexible Command Line Interface

A fairly flexible command line interface, accessible from the menu, is available to edit any preset commands, or write a new boot command set from scratch. If no configuration file is present, GRUB drops to the command line.

The list of commands (see Chapter 13 [Commands], page 21) are a subset of those supported for configuration files. Editing commands closely resembles the Bash command line (see section "Command Line Editing" in *Bash Features*), with ⟨TAB⟩-completion of commands, devices, partitions, and files in a directory depending on context.

Multiple Filesystem Types

Supports multiple filesystem types transparently, plus a useful explicit blocklist notation. The currently supported filesystem types are *BSD FFS*, *DOS FAT16 and FAT32*, *Minix fs*, *Linux ext2fs*, and *ReiserFS*. See Chapter 11 [Filesystem], page 18, for more information.

Decompression Support

Can decompress files which were compressed by `gzip`. This function is both automatic and transparent to the user (i.e. all functions operate upon the uncompressed contents of the specified files). This greatly reduces file size and loading time, a particularly major benefit for floppies.[1]

It is conceivable that some kernel modules should be loaded in a compressed state, so a different module-loading command can be specified to avoid uncompressing the modules.

Access Data on Any Installed Device

Supports reading data from any or all floppy or hard disk(s) recognized by the BIOS, independent of the setting of the root device.

[1] There are a few pathological cases where loading a very badly organized ELF kernel might take longer, but in practice this never happen.

Independent of Drive Geometry Translation
: Unlike many other boot loaders, GRUB makes the particular drive translation irrelevant. A drive installed and running with one translation may be converted to another translation without any adverse effects or changes in GRUB's configuration.

Detect All Installed RAM
: GRUB can generally find all the installed RAM on a PC-compatible machine. It uses an advanced BIOS query technique for finding all memory regions (see Chapter 25 [Memory detection], page 44). As described on the Multiboot Specification (see section "Motivation" in *The Multiboot Specification*), not all kernels make use of this information, but GRUB provides it for those who do.

Support Logical Block Address Mode
: In traditional disk calls (called *CHS mode*), there is a geometry translation problem, that is, the BIOS cannot access over 1024 cylinders, so the accessible space is limited to at least 508 MB and to at most 8GB. GRUB can't universally solve this problem, as there is no standard interface used in all machines. However, several newer machines have the new interface, Logical Block Address (*LBA*) mode. GRUB automatically detects if LBA mode is available and uses it if available. In LBA mode, GRUB can access the entire disk.

Network support
: GRUB is a disk-based boot loader but also has network support. You can load OS images from a network by using the *TFTP* protocol.

Remote terminal support
: To support computers with no console, GRUB provides remote terminal support, so that you can control GRUB from a remote host. Only serial terminal support is implemented at the moment.

Future directions might include an internal programming language for supporting richer sets of boot options with control statements (which would make GRUB its own kind of kernel). Support for non-PC hardware architectures is also planned.[2]

9.3 The role of a boot loader

The following is a quotation from Gordon Matzigkeit, a GRUB fanatic:

> Some people like to acknowledge both the operating system and kernel when they talk about their computers, so they might say they use "GNU/Linux" or "GNU/Hurd". Other people seem to think that the kernel is the most important part of the system, so they like to call their GNU operating systems "Linux systems."
>
> I, personally, believe that this is a grave injustice, because the *boot loader* is the most important software of all. I used to refer to the above systems as either "LILO"[3] or "GRUB" systems.

---

[2] There is already a port to the NEC PC-98xx series. See `http://www.kmc.kyoto-u.ac.jp/proj/linux98/arch/i386/boot/grub98/`, for more information.

[3] The LInux LOader, a boot loader that everybody uses, but nobody likes.

> Unfortunately, nobody ever understood what I was talking about; now I just use the word "GNU" as a pseudonym for GRUB.
>
> So, if you ever hear people talking about their alleged "GNU" systems, remember that they are actually paying homage to the best boot loader around... GRUB!

We, the GRUB maintainers, do not (usually) encourage Gordon's level of fanaticism, but it helps to remember that boot loaders deserve recognition. We hope that you enjoy using GNU GRUB as much as we did writing it.

10 GRUB images

GRUB consists of several images: two essential stages, optional stages called *Stage 1.5*, and two network boot images. Here is a short overview of them. See Chapter 19 [Hacking], page 39, for more details.

'`stage1`'
: This is an essential image used for booting up GRUB. Usually, this is embedded in a MBR or the boot sector of a partition. Because a PC boot sector is 512 bytes, the size of this image is exactly 512 bytes.

  All '`stage1`' must do is to load Stage 2 or Stage 1.5 from a local disk. Because of the size restriction, '`stage1`' encodes the location of Stage 2 (or Stage 1.5) in a block list format, so it never understand any filesystem structure.

'`stage2`'
: This is the core image of GRUB. This does all things but booting up itself. Usually, this is put in a filesystem, but that is not required.

'`e2fs_stage1_5`'
'`ffs_stage1_5`'
'`reiserfs_stage1_5`'
'`fat_stage1_5`'
'`minix_stage1_5`'
: These are called *Stage 1.5*, because the purpose is a bridge between '`stage1`' and '`stage2`', that is to say, Stage 1.5 is loaded by Stage 1 and Stage 1.5 loads Stage 2. The difference between '`stage1`' and '`*_stage1_5`' is that the former doesn't understand any filesystem but the latter does an filesystem (e.g. '`e2fs_stage1_5`' understands ext2fs). So you can move the location of Stage 2 to another safely, even after GRUB has been installed.

  While Stage 2 cannot generally be embedded in a fixed area as the size is so large, Stage 1.5 can be installed into the area right after a MBR, or the boot loader area of a ReiserFS or a FFS.

'`nbgrub`'
: This is a network boot image for the Network Image Proposal used by some network boot loaders, such as Etherboot. This is mostly the same as Stage 2, but this also sets up a network and loads a configuration file from the network.

'`pxegrub`'
: This is another network boot image for the Preboot Execution Environment used by several Netboot ROMs. This is identical to '`nbgrub`', except for the format.

11 Filesystem syntax and semantics

GRUB uses a special syntax for specifying disk drives which can be accessed by BIOS. Because of BIOS limitations, GRUB cannot distinguish between IDE, ESDI, SCSI, or others. You must know yourself which BIOS device is equivalent to which OS device. Normally, that will be clear if you see the files in a device or use the command `find` (see Chapter 13 [Commands], page 21).

11.1 How to specify devices

The device syntax is like this:

`(`*bios-device*`[,`*part-num*`][,`*bsd-subpart-letter*`])`

'`[]`' means the parameter is optional. *bios-device* should be either '`fd`' or '`hd`' followed by a digit, like '`fd0`'. But you can also set *bios-device* to a hexadecimal or a decimal, which is a BIOS drive number, so the following are equivalent:

```
(hd0)
(0x80)
(128)
```

part-num represents the partition number of *bios-device*, starting from zero for primary partitions and from four for extended partitions, and *bsd-subpart-letter* represents the BSD disklabel subpartition, such as '`a`' or '`e`'.

A shortcut for specifying BSD subpartitions is `(`*bios-device*`,`*bsd-subpart-letter*`)`, in this case, GRUB searches for the first PC partition containing a BSD disklabel, then finds the subpartition *bsd-subpart-letter*. Here is an example:

```
(hd0,a)
```

The syntax like '`(hd0)`' represents using the entire disk (or the MBR when installing GRUB), while the syntax like '`(hd0,0)`' represents using the partition of the disk (or the boot sector of the partition when installing GRUB).

If you enabled the network support, the special drive, '`(nd)`', is also available. Before using the network drive, you must initialize the network. See Chapter 6 [Network], page 10, for more information.

11.2 How to specify files

There are two ways to specify files, by *absolute filename* and by *blocklist*.

An absolute filename resembles a Unix absolute filename, using '`/`' for the directory separator (not '\' as in DOS). One example is '`(hd0,0)/boot/grub/menu.lst`'. This means the file '`/boot/grub/menu.lst`' in the first partition of the first hard disk. If you omit the device name in an absolute filename, GRUB uses GRUB's *root device* implicitly. So if you set the root device to, say, '`(hd1,0)`' by the command `root`, then `/boot/kernel` is the same as `(hd1,0)/boot/kernel`.

11.3 How to specify blocklists

A blocklist is used for specifying a file that doesn't appear in the filesystem, like a chainloader. The syntax is `[`*offset*`]+`*length*`[,[`*offset*`]+`*length*`]...`. Here is an example:

```
0+100,200+1,300+300
```

This represents that GRUB should read blocks 0 through 99, block 200, and blocks 300 through 599. If you omit an offset, then GRUB assumes the offset is zero.

Like the filename syntax (see Section 11.2 [Filename syntax], page 18), if a blocklist does not contain a device name, then GRUB uses GRUB's *root device*. So `(hd0,1)+1` is the same as `+1` when the root device is '`(hd0,1)`'.

12 GRUB's user interface

GRUB has both a simple menu interface for choosing preset entries from a configuration file, and a highly flexible command line for performing any desired combination of boot commands.

GRUB looks for its configuration file as soon as it is loaded. If one is found, then the full menu interface is activated using whatever entries were found in the file. If you choose the *command line* menu option, or if the configuration file was not found, then GRUB drops to the command line interface.

12.1 The flexible command line interface

The command line interface provides a prompt and after it an editable text area much like a command line in Unix or DOS. Each command is immediately executed after it is entered[4]. The commands (see Chapter 13 [Commands], page 21) are a subset of those available in the configuration file, used with exactly the same syntax.

Cursor movement and editing of the text on the line can be done via a subset of the functions available in the Bash shell:

⟨C-f⟩
⟨PC right key⟩
Move forward one character.

⟨C-b⟩
⟨PC left key⟩ Move back one character.

⟨C-a⟩
⟨HOME⟩ Move to the start of the line.

⟨C-e⟩
⟨END⟩ Move the the end of the line.

⟨C-d⟩
⟨DEL⟩ Delete the character underneath the cursor.

[4] However, this behavior will be changed in the future version, in a user-invisible way.

⟨C-h⟩
⟨BS⟩ Delete the character to the left of the cursor.

⟨C-k⟩ Kill the text from the current cursor position to the end of the line.

⟨C-u⟩ Kill backward from the cursor to the beginning of the line.

⟨C-y⟩ Yank the killed text back into the buffer at the cursor.

⟨C-p⟩
⟨PC up key⟩ Move up through the history list.

⟨C-n⟩
⟨PC down key⟩
Move down through the history list.

When typing commands interactively, if the cursor is within or before the first word in the command-line, pressing the ⟨TAB⟩ key (or ⟨C-i⟩) will display a listing of the available commands, and if the cursor is after the first word, the ⟨TAB⟩ will provide a completion listing of disks, partitions, and filenames depending on the context.

Note that you cannot use the completion functionality in the TFTP filesystem. This is because TFTP doesn't support filename listing for the security.

12.2 The simple menu interface

The menu interface is quite easy to use. Its commands are both reasonably intuitive and described on screen.

Basically, the menu interface provides a list of *boot entries* to the user to choose from. Use the arrow keys to select the entry of choice, then press ⟨RET⟩ to run it. An optional timeout is available to boot the default entry (the first one if not set), which is aborted by pressing any key.

Commands are available to enter a bare command line by pressing ⟨c⟩ (which operates exactly like the non-config-file version of GRUB, but allows one to return to the menu if desired by pressing ⟨ESC⟩) or to edit any of the *boot entries* by pressing ⟨e⟩.

12.3 Editing a menu entry

The menu entry editor looks much like the main menu interface, but the lines in the menu are individual commands in the selected entry instead of entry names.

If an ⟨ESC⟩ is pressed in the editor, it aborts all the changes made to the configuration entry and returns to the main menu interface.

When a particular line is selected, the editor places the user at a special version of the GRUB command line to edit that line. When the user hits ⟨RET⟩, GRUB replaces the line in question in the boot entry with the changes (unless it was aborted via ⟨ESC⟩, in which case the changes are thrown away).

If you want to add a new line to the menu entry, press ⟨o⟩ if adding a line after the current line or press ⟨O⟩ if before the current line.

To delete a line, hit the key ⟨d⟩. Although GRUB does not support *undo* unfortunately, you can do almost the same thing by just returning to the main menu.

13 The list of available commands

In this chapter, we list all commands that are available in GRUB.

Commands belong to different groups. A few can only be used in the global section of the configuration file (or "menu"); most of them can be entered on the command line and can be either used in the menu or in the menu entries.

13.1 The list of commands for the menu only

The semantics used in parsing the configuration file are the following:

- The menu-specific commands have to be used before any others.
- The files *must* be in plain-text format.
- '`#`' at the beginning of a line in a configuration file means it is only a comment.
- Options are separated by spaces.
- All numbers can be either decimal or hexadecimal. A hexadecimal number must be preceded by '`0x`', and is case-insensitive.
- Extra options or text at the end of the line is ignored unless otherwise specified.
- Unrecognized commands are added to the current entry, except before entries start, where they are ignored.

These commands can only be used in the menu:

default *num* Command

Set the default entry to the entry number *num*. Numbering starts from 0, and entry number 0 is the default if the command is not used.

fallback *num* Command

Go into unattended boot mode: if the default boot entry has any errors, instead of waiting for the user to do anything, immediately start over using the *num* entry (same numbering as the `default` command). This obviously won't help if the machine was rebooted by a kernel that GRUB loaded.

hiddenmenu Command

Don't display the menu. If the command is used, no menu will be displayed on the control terminal, and the default entry will be booted after the timeout expired. The user can still request the menu to be displayed by pressing ⟨ESC⟩ before the timeout expires.

timeout *sec* Command

Set a timeout, in *sec* seconds, before automatically booting the default entry (normally the first entry defined).

title *name* ... Command

Start a new boot entry, and set its name to the contents of the rest of the line, starting with the first non-space character.

13.2 The list of command-line and menu commands

Commands usable both in the menu and in the command line.

bootp Command

Initialize a network device via the *BOOTP* protocol. This command is only available if GRUB is compiled with netboot support.

color *normal* [*highlight*] Command

Change the menu colors. The color *normal* is used for most lines in the menu, and the color *highlight* is used to highlight the line where the cursor points. If you omit *highlight*, then the inverted color of *normal* is used for the highlighted line. The format of a color is *foreground/background*. *foreground* and *background* are symbolic color names. A symbolic color name must be one of these:

- black
- blue
- green
- cyan
- red
- magenta
- brown
- light-gray

These below can be specified only for the foreground.

- dark-gray
- light-blue
- light-green
- light-cyan
- light-red
- light-magenta
- yellow
- white

But only the first eight names can be used for *background*. You can prefix `blink-` to *foreground* if you want a blinking foreground color.

This command can be used in the configuration file and on the command line, so you may write something like this in your configuration file:

```
# Set default colors.
color light-gray/blue black/light-gray

# Change the colors.
title OS-BS like
color magenta/blue black/magenta
```

device *drive file* Command

In the grub shell, specify the file *file* as the actual drive for a BIOS drive *drive*. You can use this command to create a disk image, and/or to fix the drives guessed by GRUB when GRUB fails to determine them correctly, like this:

```
grub> device (fd0) /floppy-image
grub> device (hd0) /dev/sd0
```

This command can be used only in the grub shell (see Chapter 15 [Invoking the grub shell], page 35).

dhcp Command

Initialize a network device via the *DHCP* protocol. Currently, this command is just an alias for `bootp`, since the two protocols are very similar. This command is only available if GRUB is compiled with netboot support.

hide *partition* Command

Hide *partition* by setting the *hidden* bit in its partition type code. This is useful only when booting DOS or Windows and multiple primary FAT partitions exist in one disk.

partnew *part type from to* Command

Create a new primary partition. *part* is a partition specification in GRUB syntax (see Chapter 2 [Naming convention], page 1); *type* is the partition type and must be a number in the range `0-0xff`; *from* and *to* are the starting and ending sectors, expressed as an absolute sector number.

parttype *part type* Command

Change the type of an existing partition. *part* is a partition specification in GRUB syntax (see Chapter 2 [Naming convention], page 1); *type* is the new partition type and must be a number in the range 0-0xff.

password [‘`--md5`’] *passwd* [*new-config-file*] Command

If used in the first section of a menu file, disable all interactive editing control (menu entry editor and command line) and entries protected by the command `lock`. If the password *passwd* is entered, it loads the *new-config-file* as a new config file and restarts the GRUB Stage 2, if *new-config-file* is specified. Otherwise, GRUB will just unlock the privileged instructions. You can also use this command in the script section, in which case it will ask for the password, before continueing. The option ‘`--md5`’ tells GRUB that *passwd* is encrypted with md5crypt.

rarp Command

Initialize a network device via the *RARP* protocol. This command is only available if GRUB is compiled with netboot support.

serial [‘`--unit=unit`’] [‘`--port=port`’] [‘`--speed=speed`’] [‘`--word=word`’] [‘`--parity=parity`’] [‘`--stop=stop`’] [‘`--device=dev`’] Command

Initialize a serial device. *unit* is a number in the range 0-3 specifying which serial port to use; default is 0, that corresponds the port often called COM1. *port* is the

I/O port where the UART is to be found; if specified it takes precedence over *unit*. *speed* is the transmission speed; default is 9600. *word* and *stop* are the number of data bits and stop bits. Data bits must be in the range 5-8 and stop bits are 1 or 2. Default is 8 data bits and one stop bit. *parity* is one of '`no`', '`odd`', '`even`' and defaults to '`no`'. The option '`--device`' can only be used in the GRUB shell and is used to specify the tty device to be used in the host Operating System.

The serial port is not used as a communication channel unless the `terminal` command is used.

This command is only available if GRUB is compiled with serial support.

setkey *to_key from_key* Command

Change the keyboard map. The key *from_key* is mapped to the key *to_key*. Note that this command *does not* exchange the keys. If you want to exchange the keys, run this command again with the arguments exchanged, like this:

```
grub> setkey capslock control
grub> setkey control capslock
```

A key must be an alphabet, a digit, or one of these symbols: '`escape`', '`exclam`', '`at`', '`numbersign`', '`dollar`', '`percent`', '`caret`', '`ampersand`', '`asterisk`', '`parenleft`', '`parenright`', '`minus`', '`underscore`', '`equal`', '`plus`', '`backspace`', '`tab`', '`bracketleft`', '`braceleft`', '`bracketright`', '`braceright`', '`enter`', '`control`', '`semicolon`', '`colon`', '`quote`', '`doublequote`', '`backquote`', '`tilde`', '`shift`', '`backslash`', '`bar`', '`comma`', '`less`', '`period`', '`greater`', '`slash`', '`question`', '`alt`', '`space`', '`capslock`', '`FX`' ('`X`' is a digit), and '`delete`'. This table describes to which character each of the symbols corresponds:

'`exclam`'	'`!`'
'`at`'	'`@`'
'`numbersign`'	'`#`'
'`dollar`'	'`$`'
'`percent`'	'`%`'
'`caret`'	'`^`'
'`ampersand`'	'`&`'
'`asterisk`'	'`*`'
'`parenleft`'	'`(`'
'`parenright`'	'`)`'
'`minus`'	'`-`'

‘underscore’
‘_’

‘equal’ ‘=’

‘plus’ ‘+’

‘bracketleft’
‘[’

‘braceleft’
‘{’

‘bracketright’
‘]’

‘braceright’
‘}’

‘semicolon’
‘;’

‘colon’ ‘:’

‘quote’ ‘'’

‘doublequote’
‘"’

‘backquote’
‘`’

‘tilde’ ‘~’

‘backslash’
‘\’

‘bar’ ‘|’

‘comma’ ‘,’

‘less’ ‘<’

‘period’ ‘.’

‘greater’ ‘>’

‘slash’ ‘/’

‘question’
‘?’

‘space’ ‘ ’

terminal [‘--dumb’] [‘--timeout=secs’] [‘console’] [‘serial’] Command

Select a terminal for user interaction. The terminal is assumed to be vt100 compatible unless ‘--dumb’ is specified. If both ‘console’ and ‘serial’ are specified, then GRUB will use the one where a key is entered first or the first when the timeout expires. If neither are specified, the current setting is reported. Default is ‘console’. This command is only available if GRUB is compiled with serial support.

tftpserver *ipaddr* Command

Override a TFTP server address returned by a BOOTP/DHCP/RARP server. The argument *ipaddr* must be in dotted decimal format, like '`192.168.0.15`'. This command is only available if GRUB is compiled with netboot support.

unhide *partition* Command

Unhide *partition* by clearing the *hidden* bit in its partition type code. This is useful only when booting DOS or Windows and multiple primary partitions exist in one disk.

13.3 The list of command-line and menu entry commands

These commands are usable in the command line and in menu entries. If you forget a command, you can run the command `help`.

blocklist *file* Command

Print the blocklist notation of the file *file* (see Section 11.3 [Blocklist syntax], page 19).

boot Command

Boot the OS/chain-loader which has been loaded. Only necessary if running the fully interactive command line (it is implicit at the end of a menu entry).

cat *file* Command

Display the contents of the file *file*. This command may be useful to remind you of your OS's root partition:

```
grub> cat /etc/fstab
```

chainloader ['`--force`'] *file* Command

Load *file* as a chain-loader. Like any other file loaded by the filesystem code, it can use the blocklist notation to grab the first sector of the current partition with '`+1`'. If you specify the option '`--force`', then load *file* forcibly, whether it has a correct signature or not. This is required when you want to load a defective boot loader, such as SCO Unixware 7.1.

cmp *file1 file2* Command

Compare the file *file1* with the file *file2*. If they differ in size, print the sizes like this:

```
Differ in size: 0x1234 [foo], 0x4321 [bar]
```

If the sizes are equal but the bytes at an offset differ, then print the bytes like this:

```
Differ at the offset 777: 0xbe [foo], 0xef [bar]
```

If they are completely identical, nothing will be printed.

configfile *file* Command

Load *file* as a configuration file.

debug Command

Toggle debug mode (by default it is off). When debug mode is on, some extra messages are printed to show disk activity. This global debug flag is mainly useful for GRUB developers when testing new code.

displaymem Command

Display what GRUB thinks the system address space map of the machine is, including all regions of physical RAM installed. GRUB's *upper/lower memory* display uses the standard BIOS interface for the available memory in the first megabyte, or *lower memory*, and a synthesized number from various BIOS interfaces of the memory starting at 1MB and going up to the first chipset hole for *upper memory* (the standard PC *upper memory* interface is limited to reporting a maximum of 64MB).

embed *stage1_5 device* Command

Embed the Stage 1.5 *stage1_5* in the sectors after the MBR if *device* is a drive, or in the *boot loader* area if *device* is a FFS partition or a ReiserFS partition.[5] Print the number of sectors which *stage1_5* occupies, if successful.

find *filename* Command

Search for the filename *filename* in all of partitions and print the list of the devices which contain the file. The filename *filename* should be an absolute filename like `/boot/grub/stage1`.

fstest Command

Toggle filesystem test mode. Filesystem test mode, when turned on, prints out data corresponding to all the device reads and what values are being sent to the low-level routines. The format is '<*partition-offset-sector*, *byte-offset*, *byte-length*>' for high-level reads inside a partition, and '[*disk-offset-sector*]' for low-level sector requests from the disk. Filesystem test mode is turned off by any use of the `install` or `testload` commands.

geometry *drive* [*cylinder head sector* [*total_sector*]] Command

Print the information for the drive *drive*. In the GRUB shell, you can set the geometry of the drive arbitrarily. The number of the cylinders, the one of the heads, the one of the sectors and the one of the total sectors are set to CYLINDER, HEAD, SECTOR and TOTAL_SECTOR, respectively. If you omit TOTAL_SECTOR, then it will be calculated based on the C/H/S values automatically.

halt '`--no-apm`' Command

The command halts the computer. If the '`--no-apm`' option is specified, no APM BIOS call is performed. Otherwise, the computer is shut down using APM.

help [*pattern* ...] Command

Display helpful information about builtin commands. If you do not specify *pattern*, this command shows short descriptions of all available commands. If you specify any *patterns*, it displays longer information about each of the commands which match those *patterns*.

[5] The latter feature has not been implemented yet.

impsprobe Command

Probe the Intel Multiprocessor Specification 1.1 or 1.4 configuration table and boot the various CPUs which are found into a tight loop. This command can be used only in the Stage 2.

initrd *file ...* Command

Load an initial ramdisk for a Linux format boot image and set the appropriate parameters in the Linux setup area in memory.

install [‘`--force-lba`’] [‘`--stage2=os_stage2_file`’] *stage1_file* [‘`d`’] *dest_dev stage2_file* [*addr*] [‘`p`’] [*config_file*] [*real_config_file*] Command

This command is fairly complex, and you should not use this command unless you are familiar with GRUB. In short, it will perform a full install presuming the Stage 2 or Stage 1.5[6] is in its final install location.

In slightly more detail, it will load *stage1_file*, validate that it is a GRUB Stage 1 of the right version number, install a blocklist for loading *stage2_file* as a Stage 2. If the option ‘`d`’ is present, the Stage 1 will always look for the actual disk *stage2_file* was installed on, rather than using the booting drive. The Stage 2 will be loaded at address *addr*, which must be ‘`0x8000`’ for a true Stage 2, and ‘`0x2000`’ for a Stage 1.5. If *addr* is not present, GRUB will determine the address automatically. It then writes the completed Stage 1 to the first block of the device *dest_dev*. If the options ‘`p`’ or *config_file* are present, then it reads the first block of stage2, modifies it with the values of the partition *stage2_file* was found on (for ‘`p`’) or places the string *config_file* into the area telling the stage2 where to look for a configuration file at boot time. Likewise, if *real_config_file* is present and *stage2_file* is a Stage 1.5, then the Stage 2 *config_file* is patched with the configuration filename *real_config_file*. This command preserves the DOS BPB (and for hard disks, the partition table) of the sector the Stage 1 is to be installed into.

Caution: Several buggy BIOSes don’t pass a booting drive properly when booting from a hard disk drive. Therefore, you will have to specify the option ‘`d`’, whether your Stage2 resides at the booting drive or not, if you have such a BIOS unfortunately. We know these are defective in that:

Fujitsu LifeBook 400 BIOS version 31J0103A

HP Vectra XU 6/200 BIOS version GG.06.11

Caution2: A number of BIOSes don’t return a correct LBA support bitmap even if they do have the support. So GRUB provides a solution to ignore the wrong bitmap, that is, the option ‘`--force-lba`’. Don’t use this option if you know that your BIOS doesn’t have LBA support.

Caution3: You must specify the option ‘`--stage2`’ in the grub shell, if you cannot unmount the filesystem where your stage2 file resides. The argument should be the file name in your operating system.

---

[6] They’re loaded the same way, so we will refer to the Stage 1.5 as a Stage 2 from now on.

ioprobe *drive* Command

Probe I/O ports used for the drive *drive*. This command will list the I/O ports on the screen. For technical information, See Chapter 24 [I/O ports detection], page 44.

kernel [`--type=type`] *file* ... Command

Attempt to load the primary boot image (Multiboot a.out or ELF, Linux zImage or bzImage, FreeBSD a.out, NetBSD a.out, etc.) from *file*. The rest of the line is passed verbatim as the *kernel command line*. Any modules must be reloaded after using this command.

This command also accepts the option '`--type`' so that you can specify the kernel type of *file* explicitly. The argument *type* must be one of these: '`netbsd`', '`freebsd`', '`openbsd`', '`linux`', '`biglinux`', and '`multiboot`'. However, you need to specify it only if you want to load a NetBSD ELF kernel, because GRUB can automatically determine a kernel type in the other cases, quite safely.

lock Command

Prevent normal users from executing arbitrary menu entries. You must use the command `password` if you really want this command to be useful.

This command is used in a menu, as shown in this this example:

```
title This entry is too dangerous to be executed by normal users
lock
root (hd0,a)
kernel /no-security-os
```

makeactive Command

Set the active partition on the root disk to GRUB's root device. This command is limited to *primary* PC partitions on a hard disk.

map *to_drive from_drive* Command

Map the drive *from_drive* to the drive *to_drive*. This is necessary when you chain-load some operating systems, such as DOS, if such an OS resides at a non-first drive. Here is an example:

```
grub> map (hd0) (hd1)
grub> map (hd1) (hd0)
```

The example exchanges the order between the first hard disk and the second hard disk.

module *file* ... Command

Load a boot module *file* for a Multiboot format boot image (no interpretation of the file contents are made, so that user of this command must know what the kernel in question expects). The rest of the line is passed as the *module command line*, like the `kernel` command. You must load a Multiboot kernel image before loading any module.

modulenounzip *file* ... Command

The same as `module`, except that automatic decompression is disabled.

pause *message* . . . Command

Print the *message*, then wait until a key is pressed. Note that placing ⟨^G⟩ (ASCII code 7) in the message will cause the speaker to emit the standard beep sound, which is useful when prompting the user to change floppies.

quit Command

Exit from the GRUB shell `grub` (see Chapter 15 [Invoking the grub shell], page 35). This command can be used only in the GRUB shell.

reboot Command

Reboot the computer.

read *addr* Command

Read a 32-bit value from memory at address *addr* and display it in hex format.

root *device* [*hdbias*] Command

Set the current *root device* to the device *device*, then attempt to mount it to get the partition size (for passing the partition descriptor in `ES:ESI`, used by some chain-loaded boot loaders), the BSD drive-type (for booting BSD kernels using their native boot format), and correctly determine the PC partition where a BSD sub-partition is located. The optional *hdbias* parameter is a number to tell a BSD kernel how many BIOS drive numbers are on controllers before the current one. For example, if there is an IDE disk and a SCSI disk, and your FreeBSD root partition is on the SCSI disk, then use a '`1`' for *hdbias*.

rootnoverify *device* [*hdbias*] Command

Similar to `root`, but don't attempt to mount the partition. This is useful for when an OS is outside of the area of the disk that GRUB can read, but setting the correct root device is still desired. Note that the items mentioned in `root` above which derived from attempting the mount will *not* work correctly.

savedefault Command

Save the current menu entry as a default entry.

setup ['`--force-lba`'] ['`--stage2=os_stage2_file`'] ['`--prefix=dir`'] *install_device* [*image_device*] Command

Set up the installation of GRUB automatically. This command uses the more flexible command `install` in the backend and installs GRUB into the device *install_device*. If *image_device* is specified, then find the GRUB images in the device *image_device*, otherwise use the current *root device*, which can be set by the command `root`. If *install_device* is a hard disk, then embed a Stage 1.5 in the disk if possible.

The option '`--prefix`' specifies the directory under which GRUB images are put. If it is not specified, GRUB automatically searches them in '`/boot/grub`' and '`/grub`'.

The options '`--force-lba`' and '`--stage2`' are just passed to `install` if specified. See the description on `install`, for more information.

testload *file* Command

Read the entire contents of *file* in several different ways and compares them, to test the filesystem code. The output is somewhat cryptic , but if no errors are reported and the final '`i=`*X*`, filepos=`*Y*' reading has *X* and *Y* equal, then it is definitely consistent, and very likely works correctly subject to a consistent offset error. If this test succeeds, then a good next step is to try loading a kernel.

uppermem *kbytes* Command

Force GRUB to assume that only *kbytes* kilobytes of upper memory are installed. Any system address range maps are discarded.

Caution: This should be used with great caution, and should only be necessary on some old machines. GRUB's BIOS probe can pick up all RAM on all new machines the author has ever heard of. It can also be used for debugging purposes to lie to an OS.

14 Error messages reported by GRUB

This chapter describes error messages reported by GRUB when you encounter trouble. See Chapter 15 [Invoking the grub shell], page 35, if your problem is specific to the grub shell.

14.1 Errors reported by the Stage 1

The general way that the Stage 1 handles errors is to print an error string and then halt. Pressing ⟨CTRL⟩-⟨ALT⟩-⟨DEL⟩ will reboot.

The following is a comprehensive list of error messages for the Stage 1:

Hard Disk Error
: The stage2 or stage1.5 is being read from a hard disk, and the attempt to determine the size and geometry of the hard disk failed.

Floppy Error
: The stage2 or stage1.5 is being read from a floppy disk, and the attempt to determine the size and geometry of the floppy disk failed. It's listed as a separate error since the probe sequence is different than for hard disks.

Read Error
: A disk read error happened while trying to read the stage2 or stage1.5.

Geom Error
: The location of the stage2 or stage1.5 is not in the portion of the disk supported directly by the BIOS read calls. This could occur because the BIOS translated geometry has been changed by the user or the disk is moved to another machine or controller after installation, or GRUB was not installed using itself (if it was, the Stage 2 version of this error would have been seen during that process and it would not have completed the install).

14.2 Errors reported by the Stage 1.5

The general way that the Stage 1.5 handles errors is to print an error number in the form `Error` *num* and then halt. Pressing ⟨CTRL⟩-⟨ALT⟩-⟨DEL⟩ will reboot.

The error numbers correspond to the errors reported by Stage 2. See Section 14.3 [Stage2 errors], page 32.

14.3 Errors reported by the Stage 2

The general way that the Stage 2 handles errors is to abort the operation in question, print an error string, then (if possible) either continue based on the fact that an error occurred or wait for the user to deal with the error.

The following is a comprehensive list of error messages for the Stage 2 (error numbers for the Stage 1.5 are listed before the colon in each description):

1 : Filename must be either an absolute filename or blocklist
: This error is returned if a filename is requested which doesn't fit the syntax/rules listed in the Chapter 11 [Filesystem], page 18.

2 : Bad file or directory type
: This error is returned if a file requested is not a regular file, but something like a symbolic link, directory, or FIFO.

3 : Bad or corrupt data while decompressing file
: This error is returned the run-length decompression code gets an internal error. This is usually from a corrupt file.

4 : Bad or incompatible header in compressed file
: This error is returned if the file header for a supposedly compressed file is bad.

5 : Partition table invalid or corrupt
: This error s returned if the sanity checks on the integrity of the partition table fail. This is a bad sign.

6 : Mismatched or corrupt version of stage1/stage2
: This error is returned if the install command is pointed to incompatible or corrupt versions of the stage1 or stage2. It can't detect corruption in general, but this is a sanity check on the version numbers, which should be correct.

7 : Loading below 1MB is not supported
: This error is returned if the lowest address in a kernel is below the 1MB boundary. The Linux zImage format is a special case and can be handled since it has a fixed loading address and maximum size.

8 : Kernel must be loaded before booting
: This error is returned if GRUB is told to execute the boot sequence without having a kernel to start.

9 : Unknown boot failure
: This error is returned if the boot attempt did not succeed for reasons which are unknown.

10 : Unsupported Multiboot features requested
: This error is returned when the Multiboot features word in the Multiboot header requires a feature that is not recognized. The point of this is that the kernel requires special handling which GRUB is likely unable to provide.

11 : Unrecognized device string
: This error is returned if a device string was expected, and the string encountered didn't fit the syntax/rules listed in the Chapter 11 [Filesystem], page 18.

12 : Invalid device requested
: This error is returned if a device string is recognizable but does not fall under the other device errors.

13 : Invalid or unsupported executable format
: This error is returned if the kernel image being loaded is not recognized as Multiboot or one of the supported native formats (Linux zImage or bzImage, FreeBSD, or NetBSD).

14 : Filesystem compatibility error, cannot read whole file
: Some of the filesystem reading code in GRUB has limits on the length of the files it can read. This error is returned when the user runs into such a limit.

15 : File not found
: This error is returned if the specified filename cannot be found, but everything else (like the disk/partition info) is OK.

16 : Inconsistent filesystem structure
: This error is returned by the filesystem code to denote an internal error caused by the sanity checks of the filesystem structure on disk not matching what it expects. This is usually caused by a corrupt filesystem or bugs in the code handling it in GRUB.

17 : Cannot mount selected partition
: This error is returned if the partition requested exists, but the filesystem type cannot be recognized by GRUB.

18 : Selected cylinder exceeds maximum supported by BIOS
: This error is returned when a read is attempted at a linear block address beyond the end of the BIOS translated area. This generally happens if your disk is larger than the BIOS can handle (512MB for (E)IDE disks on older machines or larger than 8GB in general).

19 : Linux kernel must be loaded before initrd
: This error is returned if the initrd command is used before loading a Linux kernel. Similar to the above error, it only makes sense in that case anyway.

20 : Multiboot kernel must be loaded before modules
: This error is returned if the module load command is used before loading a Multiboot kernel. It only makes sense in this case anyway, as GRUB has no idea how to communicate the presence of location of such modules to a non-Multiboot-aware kernel.

21 : Selected disk does not exist
: This error is returned if the device part of a device- or full filename refers to a disk or BIOS device that is not present or not recognized by the BIOS in the system.

22 : No such partition
: This error is returned if a partition is requested in the device part of a device- or full filename which isn't on the selected disk.

23 : Error while parsing number
: This error is returned if GRUB was expecting to read a number and encountered bad data.

24 : Attempt to access block outside partition
: This error is returned if a linear block address is outside of the disk partition. This generally happens because of a corrupt filesystem on the disk or a bug in the code handling it in GRUB (it's a great debugging tool).

25 : Disk read error
: This error is returned if there is a disk read error when trying to probe or read data from a particular disk.

26 : Too many symbolic links
: This error is returned if the link count is beyond the maximum (currently 5), possibly the symbolic links are looped.

27 : Unrecognized command
: This error is returned if an unrecognized command is entered into the command line or in a boot sequence section of a configuration file and that entry is selected.

28 : Selected item cannot fit into memory
: This error is returned if a kernel, module, or raw file load command is either trying to load its data such that it won't fit into memory or it is simply too big.

29 : Disk write error
: This error is returned if there is a disk write error when trying to write to a particular disk. This would generally only occur during an install of set active partition command.

30 : Invalid argument
: This error is returned if an argument specified to a command is invalid.

31 : File is not sector aligned
: This error may occur only when you access a ReiserFS partition by block-lists (e.g. the command `install`). In this case, you should mount the partition with the '`-o notail`' option.

32 : Must be authenticated
: This error is returned if you try to run a locked entry. You should enter a correct password before running such an entry.

15 Invoking the grub shell

This chapter documents the grub shell `grub`. Note that the grub shell is an emulator; it doesn't run under the native environment, so it sometimes does something wrong. Therefore, you shouldn't trust it too much. If there is anything wrong with it, don't hesitate to try the native GRUB environment, especially when it guesses a wrong map between BIOS drives and OS devices.

15.1 Introduction into the grub shell

You can use the command `grub` for installing GRUB under your operating systems and for a testbed when you add a new feature into GRUB or when fix a bug. `grub` is almost the same as the Stage 2, and, in fact, it shares the source code with the Stage 2 and you can use the same commands (see Chapter 13 [Commands], page 21) in `grub`. It is emulated by replacing BIOS calls with UNIX system calls and libc functions.

The command `grub` accepts the following options:

'`--help`' Print a summary of the command line options and exit.

'`--version`'
: Print the version number of GRUB and exit.

'`--verbose`'
: Print some verbose messages for debugging purpose.

'`--device-map=`*file*'
: Use the device map file *file*. The format is described in Section 15.3 [Device map], page 36.

'`--no-floppy`'
: Do not probe any floppy drive. This option has no effect if the option '`--device-map`' is specified (see Section 15.3 [Device map], page 36).

'`--probe-second-floppy`'
: Probe the second floppy drive. If this option is not specified, the grub shell does not probe it, as that sometimes takes a long time. If you specify the device map file (see Section 15.3 [Device map], page 36), the grub shell just ignores this option.

'`--config-file=`*file*'
: Read the configuration file *file* instead of '`/boot/grub/menu.lst`'. The format is the same as the normal GRUB syntax. See Chapter 11 [Filesystem], page 18, for more information.

'`--boot-drive=`*drive*'
: Set the stage2 *boot_drive* to *drive*. This argument should be an integer (decimal, octal or hexadecimal).

'`--install-partition=`*par*'
: Set the stage2 *install_partition* to *par*. This argument should be an integer (decimal, octal or hexadecimal).

'--no-config-file'
: Do not use the configuration file even if it can be read.

'--no-curses'
: Do not use the curses interface even if it is available.

'--batch'
: This option has the same meaning as '--no-config-file --no-curses'.

'--read-only'
: Disable writing to any disk.

'--hold'
: Wait until a debugger will attach. This option is useful when you want to debug the startup code.

15.2 How to install GRUB via grub

The installation procedure is the same as under the *native* Stage 2. See Chapter 3 [Installation], page 2, for more information. The command grub-specific information is described here.

What you should be careful about is *buffer cache*. grub makes use of raw devices instead of filesystems that your operating systems serve, so there exists a potential problem that some cache inconsistency may corrupt your filesystems. What we recommend is:

- If you can unmount drives to which GRUB may write any amount of data, unmount them before running grub.
- If a drive cannot be unmounted but can be mounted with the read-only flag, mount it in read-only mode. That should be secure.
- If a drive must be mounted with the read-write flag, make sure that any activity is not being done on it during running the command grub.
- Reboot your operating system as soon as possible. Probably that is not required if you follow these rules above, but reboot is the most secure way.

In addition, enter the command quit when you finish the installation. That is *very important* because quit makes the buffer cache consistent. Do not push ⟨C-c⟩.

If you want to install GRUB non-interactively, specify '--batch' option in the command line. This is a simple example:

```
#!/bin/sh

# Use /usr/sbin/grub if you are on an older system.
/sbin/grub --batch <<EOT 1>/dev/null 2>/dev/null
root (hd0,0)
setup (hd0)
quit
EOT
```

15.3 The map between BIOS drives and OS devices

When you specify the option '--device-map' (see Section 15.1 [Basic usage], page 35), the grub shell creates the *device map file* automatically unless it already exists. The filename '/boot/grub/device.map' is preferred.

If the device map file exists, the grub shell reads it to map BIOS drives to OS devices. This file consists of lines like this:

device file

device is a drive, which syntax is the same as the one in GRUB (see Section 11.1 [Device syntax], page 18), and *file* is an OS's file, which is normally a device file.

The reason why the grub shell gives you the device map file is that it cannot guess the map between BIOS drives and OS devices correctly in some environments. For example, if you exchange the boot sequence between IDE and SCSI in your BIOS, it mistakes the order.

Thus, edit the file if the grub shell makes a mistake. You can put any comments in the file if needed, as the grub shell assumes that a line is just a comment if the first character is '`#`'.

16 Invoking grub-install

The program `grub-install` installs GRUB on your drive by the grub shell (see Chapter 15 [Invoking the grub shell], page 35). You must specify the device name on which you want to install GRUB, like this:

`grub-install` *install_device*

The device name *install_device* is an OS device name or a GRUB device name.

`grub-install` accepts the following options:

'`--help`' Print a summary of the command line options and exit.

'`--version`'
: Print the version number of GRUB and exit.

'`--force-lba`'
: Force GRUB to use LBA mode even for a buggy BIOS. Use this option only if your BIOS doesn't work in LBA mode even though it supports LBA mode.

'`--root-directory=`*dir*'
: Install GRUB images under the directory *dir* instead of the root directory. This option is useful when you want to install GRUB into a separate partition or a removable disk. Here is an example when you have a separate *boot* partition which is mounted on '`/boot`':

```
grub-install --root-directory=/boot '(hd0)'
```

'`--grub-shell=`*file*'
: Use *file* as the grub shell. You can append arbitrary options to *file* after the filename, like this:

```
grub-install --grub-shell="grub --read-only" /dev/fd0
```

17 Invoking grub-md5-crypt

The program `grub-md5-crypt` encrypts a password in MD5 format. This is just a frontend of the grub shell (see Chapter 15 [Invoking the grub shell], page 35). Passwords encrypted by this program can be used with the command `password` (see Section 13.2 [Command-line and menu commands], page 22).

`grub-md5-crypt` accepts the following options:

'`--help`' Print a summary of the command line options and exit.

'`--version`'
: Print the version information and exit.

'`--grub-shell=`*file*'
: Use *file* as the grub shell.

18 Invoking mbchk

The program `mbchk` checks for the format of a Multiboot kernel. We recommend using this program before booting your own kernel by GRUB.

`mbchk` accepts the following options:

'`--help`' Print a summary of the command line options and exit.

'`--version`'
: Print the version number of GRUB and exit.

'`--quiet`' Suppress all normal output.

Part III: The Programmer Reference Manual

19 Implementation details

This part describes the GRUB internals so that developers can understand the implementation and start to hack GRUB. Of course, the source code has the complete information, so refer to it when you are not satisfied with this documentation.

20 The memory map of various components

GRUB is broken into 2 distinct components, or *stages*, which are loaded at different times in the boot process. The Stage 1 has to know where to find Stage 2, and the Stage 2 has to know where to find its configuration file (if Stage 2 doesn't have a configuration file, it drops into the command line interface and waits for a user command).

Here is the memory map of the various components[1]:

0 to 4K-1
: Interrupt & BIOS area

down from 8K-1
: 16-bit stack area

8K to (ebss1.5)
: Stage 1.5 (optionally) loaded here by Stage 1

0x7c00 to 0x7dff
: Stage 1 loaded here by the BIOS

0x7e00 to 0x7e08
: Scratch space used by Stage 1

32K to (ebss2)
: Stage 2 loaded here by Stage 1.5 or Stage 1

(middle area)
: Heap used for random memory allocation

down from 416K-1
: 32-bit stack area

416K to 448K-1
: Filesystem info buffer (when reading a filesystem)

448K to 479.5K-1
: BIOS track read buffer

479.5K to 480K-1
: 512 byte fixed SCRATCH area

---

[1] Currently GRUB does not use the extended memory for itself, since it is used to load an operating system. But we are planning to use it for GRUB itself in the future by *lazy loading*. Ask okuji for more information.

480K to 511K-1
: General storage heap

See the file '`stage2/shared.h`', for more information.

21 Embedded variables in GRUB

GRUB's *stage1* and *stage2* have embedded variables whose locations are well-defined, so that the installation can patch the binary file directly without recompilation of the modules.

In *stage1*, these are defined (The number in the parenthesis of each entry is an offset number):

stage1 version (0x3e)
: This is the version bytes (should 03:00).

loading drive (0x40)
: This is the BIOS drive number to load the block from. If the number is 0xff, then load from the booting drive.

stage2 sector (0x41)
: This is the location of the first sector of the *stage2*.

stage2 address (0x45)
: This is the data for the `jmp` command to the starting address of the component loaded by the stage1.

  A *stage1.5* should be loaded at address 0x2000, and a *stage2* should be loaded at address 0x8000. Both use a CS of 0.

stage2 segment (0x47)
: This is the segment of the starting address of the component loaded by the *stage1*.

In the first sector of *stage1.5* and *stage2*, the blocklists are recorded between *firstlist* (0x200) and *lastlist* (determined when assembling the file '`stage2/start.S`').

The trick here is that it is actually read backward, and the first 8-byte blocklist is not read here, but after the pointer is decremented 8 bytes, then after reading it, it decrements again, reads, decrements, reads, etc. until it is finished. The terminating condition is when the number of sectors to be read in the next blocklist is 0.

The format of a blocklist can be seen from the example in the code just before the `firstlist` label. Note that it is always from the beginning of the disk, and *not* relative to the partition boundaries.

In *stage1.5* and *stage2* (these are all defined at the beginning of '`shared_src/asm.S`'):

major version (0x6)
: This is the major version byte (should be 3).

minor version (0x7)
: This is the minor version byte (should be 0).

install_partition (0x8)
: This is an unsigned long representing the partition on the currently booted disk which GRUB should expect to find it's data files and treat as the default root partition.

  The format of is exactly the same as the *partition* part (the *disk* part is ignored) of the data passed to an OS by a Multiboot-compliant boot loader in the *boot_device* data element, with one exception.

  The exception is that if the first level of disk partitioning is left as 0xFF (decimal 255, which is marked as no partitioning being used), but the second level does have a partition number, it looks for the first BSD-style PC partition, and finds the numbered BSD sub-partition in it. The default *install_partition* 0xFF00FF, would then find the first BSD-style PC partition, and use the '`a`' partition in it, and 0xFF01FF would use the '`b`' partition, etc.

  If an explicit first-level partition is given, then no search is performed, and it will expect that the BSD-style PC partition is in the appropriate location, else a '`no such partition`' error will be returned.

  If a *stage1.5* is being used, it will pass its own *install_partition* to any *stage2* it loads, therefore overwriting the one present in the *stage2*.

stage2_id (0xc)
: This is the *stage1.5* or *stage2* identifier.

version_string (0xd)
: This is the *stage1.5* or *stage2* version string. It isn't meant to be changed, simply easy to find.

config_file (after the terminating zero of *version_string*)
: This is the location, using the GRUB filesystem syntax, of the config file. It will, by default, look in the *install_partition* of the disk GRUB was loaded from, though one can use any valid GRUB filesystem string, up to and including making it look on other disks.

  The boot loader itself doesn't search for the end of *version_string*, it simply knows where *config_file* is, so the beginning of the string cannot be moved after compile-time. This should be OK, since the *version_string* is meant to be static.

  The code of *stage2* starts again at offset 0x70, so *config_file* string obviously can't go past there. Also, remember to terminate the string with a 0.

  Note that *stage1.5* uses a tricky internal representation for *config_file*, which is the format of *device*`:`*filename* ('`:`' is not present actually). *device* is an unsigned long like *install_partition*, and *filename* is an absolute filename or a blocklist. If *device* is disabled, that is, the drive number is 0xff, then *stage1.5* uses the *boot drive* and the *install partition* instead.

22 The generic interface for the fs code

For any particular partition, it is presumed that only one of the *normal* filesystems such as FAT, FFS, or ext2fs can be used, so there is a switch table managed by the functions in '`disk_io.c`'. The notation is that you can only *mount* one at a time.

The blocklist filesystem has a special place in the system. In addition to the *normal* filesystem (or even without one mounted), you can access disk blocks directly (in the indicated partition) via the blocklist notation. Using the blocklist filesystem doesn't effect any other filesystem mounts.

The variables which can be read by the filesystem backend are:

`current_drive`
: Contain the current BIOS drive number (numbered from 0, if a floppy, and numbered from 0x80, if a hard disk).

`current_partition`
: Contain the current partition number.

`current_slice`
: Contain the current partition type.

`saved_drive`
: Contain the *drive* part of the root device.

`saved_partition`
: Contain the *partition* part of the root device.

`part_start`
: Contain the current partition starting address.

`part_length`
: Contain the current partition length, in sectors.

`print_possibilities`
: True when the `dir` function should print the possible completions of a file, and false when it should try to actually open a file of that name.

`FSYS_BUF`
: Point to a filesystem buffer which is 32K in size, to use in any way which the filesystem backend desires.

The variables which need to be written by a filesystem backend are:

`filepos`
: Should be the current position in the file.

  Caution: the value of *filepos* can be changed out from under the filesystem code in the current implementation. Don't depend on it being the same for later calls into the back-end code!

`filemax`
: Should be the length of the file.

`disk_read_func`
: Should be set to the value of '`disk_read_hook`' *only* during reading of data for the file, not any other fs data, inodes, FAT tables, whatever, then set to `NULL` at all other times (it will be `NULL` by default). If this isn't done correctly, then the `testload` and `install` commands won't work correctly.

The functions expected to be used by the filesystem backend are:

`devread` Only read sectors from within a partition. Sector 0 is the first sector in the partition.

`grub_read`
If the backend uses the blocklist code (like the FAT filesystem backend does), then `grub_read` can be used, after setting *block_file* to 1.

The functions expected to be defined by the filesystem backend are described at least moderately in the file '`filesys.h`'. Their usage is fairly evident from their use in the functions in '`disk_io.c`', look for the use of the *fsys_table* array.

Caution: The semantics are such that then '`mount`'ing the filesystem, presume the filesystem buffer *FSYS_BUF* is corrupted, and (re-)load all important contents. When opening and reading a file, presume that the data from the '`mount`' is available, and doesn't get corrupted by the open/read (i.e. multiple opens and/or reads will be done with only one mount if in the same filesystem).

23 The bootstrap mechanism used in GRUB

The disk space can be used in a boot loader is very restricted because a MBR (see Chapter 27 [MBR], page 56) is only 512 bytes but it also contains a partition table (see Chapter 28 [Partition table], page 57) and a BPB. So the question is how to make a boot loader code enough small to be fit in a MBR.

However, GRUB is a very large program, so we break GRUB into 2 (or 3) distinct components, *stage1* and *stage2* (and optionally *stage1.5*). See Chapter 20 [Memory map], page 39, for more information.

We embed *stage1* in a MBR or in the boot sector of a partition , and place *stage2* in a filesystem. The optional *stage1.5* can be installed in a filesystem, in the *boot loader* area in a FFS, and in the sectors right after a MBR, because *stage1.5* is enough small and the sectors right after a MBR is normally an unused region. The size of this region is the number of sectors per head minus 1.

Thus, all the *stage1* must do is just load a *stage2* or *stage1.5*. But even if *stage1* needs not to support the user interface or the filesystem interface, it is impossible to make *stage1* less than 400 bytes, because GRUB should support both the CHS mode and the LBA mode (see Chapter 26 [Low-level disk I/O], page 49).

The solution used by GRUB is that *stage1* loads only the first sector of a *stage2* (or a *stage1.5*) and *stage2* itself loads the rest. The flow of *stage1* is:

1. Initialize the system briefly.
2. Detect the geometry and the accessing mode of the *loading drive*.
3. Load the first sector of the *stage2*.
4. Jump to the starting address of the *stage2*.

The flow of *stage2* (and *stage1.5*) is:

1. Load the rest of itself to the real starting address, that is, the starting address plus 512 bytes. The blocklists are stored in the last part of the first sector.
2. Long jump to the real starting address.

Note that *stage2* (or *stage1.5*) does not probe the geometry or the accessing mode of the *loading drive*, since *stage1* has already probed them.

24 How to detect I/O ports used for a BIOS drive

In the PC world, BIOS cannot detect if a hard disk drive is SCSI or IDE, generally speaking. Thus, it is not trivial to know which BIOS drive corresponds to an OS device. So the Multiboot Specification describes some techniques on how to guess mappings (see section "BIOS device mapping techniques" in *The Multiboot Specification*).

However, the techniques described are unreliable or difficult to be implemented, so we use a different technique from them in GRUB. Our technique is *INT 13H tracking technique*. More precisely, it runs the INT 13 call (see Chapter 26 [Low-level disk I/O], page 49) in single-step mode just like a debugger and parses the instructions.

To execute the call one instruction at a time, set the TF (trap flag) flag in the register *FLAGS*. By this, your CPU generates *Break Point Trap* after each instruction is executed and call INT 1. In the stack in the interrupt handler, callee's FLAGS and the far pointer which points to the next instruction to be executed are pushed, so we can know what instruction will be executed in the next time and the current contents of all the registers. If the next instruction is an I/O operation, the interrupt handler adds the I/O port into the *I/O map*.

If the INT 13 handler returns, the TF flag is cleared automatically by the instruction `iret`, and then output the I/O map on the screen. See the source code for the command `ioprobe` (see Section 13.3 [Command-line and menu entry commands], page 26), for more information.

25 How to detect all installed RAM

There are three BIOS calls which return the information of installed RAM. GRUB uses these calls to detect all installed RAM and which address range should be treated by operating systems.

25.1 INT 15H, AX=E820h interrupt call

Real mode only.

This call returns a memory map of all the installed RAM, and of physical memory ranges reserved by the BIOS. The address map is returned by making successive calls to this API, each returning one "run" of physical address information. Each run has a type which dictates how this run of physical address range should be treated by the operating system.

If the information returned from INT 15h, AX=E820h in some way differs from INT 15h, AX=E801h (see Section 25.2 [Get Large Memory Size], page 48) or INT 15h AH=88h (see Section 25.3 [Get Extended Memory Size], page 49), then the information returned from E820h supersedes what is returned from these older interfaces. This allows the BIOS to return whatever information it wishes to for compatibility reasons.

Input:

`EAX`	Function Code	E820h
`EBX`	Continuation	Contains the *continuation value* to get the next run of physical memory. This is the value returned by a previous call to this routine. If this is the first call, `EBX` must contain zero.
`ES:DI`	Buffer Pointer	Pointer to an Address Range Descriptor structure which the BIOS is to fill in.
`ECX`	Buffer Size	The length in bytes of the structure passed to the BIOS. The BIOS will fill in at most `ECX` bytes of the structure or however much of the structure the BIOS implements. The minimum size which must be supported by both the BIOS and the caller is 20 bytes. Future implementations may extend this structure.
`EDX`	Signature	'`SMAP`' - Used by the BIOS to verify the caller is requesting the system map information to be returned in `ES:DI`.

Output:

`CF`	Carry Flag	Non-Carry - indicates no error
`EAX`	Signature	'`SMAP`' - Signature to verify correct BIOS revision.
`ES:DI`	Buffer Pointer	Returned Address Range Descriptor pointer. Same value as on input.
`ECX`	Buffer Size	Number of bytes returned by the BIOS in the address range descriptor. The minimum size structure returned by the BIOS is 20 bytes.
`EBX`	Continuation	Contains the continuation value to get the next address descriptor. The actual significance of the continuation value is up to the discretion of the BIOS. The caller must pass the continuation value unchanged as input to the next iteration of the E820h call in order to get the next Address Range Descriptor. A return value of zero means that this is the last descriptor. Note that the BIOS indicates that the last valid descriptor has been returned by either returning a zero as the continuation value, or by returning carry.

The Address Range Descriptor Structure is:

Offset in Bytes	Name	Description
0	*BaseAddrLow*	Low 32 Bits of Base Address
4	*BaseAddrHigh*	High 32 Bits of Base Address

8	*LengthLow*	Low 32 Bits of Length in Bytes
12	*LengthHigh*	High 32 Bits of Length in Bytes
16	*Type*	Address type of this range

The *BaseAddrLow* and *BaseAddrHigh* together are the 64 bit *BaseAddress* of this range. The *BaseAddress* is the physical address of the start of the range being specified.

The *LengthLow* and *LengthHigh* together are the 64 bit *Length* of this range. The *Length* is the physical contiguous length in bytes of a range being specified.

The *Type* field describes the usage of the described address range as defined in the table below:

Value	Mnemonic	Description
1	*AddressRangeMemory*	This run is available RAM usable by the operating system.
2	*AddressRangeReserved*	This run of addresses is in use or reserved by the system, and must not be used by the operating system.
Other	*Undefined*	Undefined - Reserved for future use. Any range of this type must be treated by the OS as if the type returned was *AddressRangeReserved*.

The BIOS can use the *AddressRangeReserved* address range type to block out various addresses as *not suitable* for use by a programmable device.

Some of the reasons a BIOS would do this are:

- The address range contains system ROM.
- The address range contains RAM in use by the ROM.
- The address range is in use by a memory mapped system device.
- The address range is for whatever reason are unsuitable for a standard device to use as a device memory space.

Here is the list of assumptions and limitations:

1. The BIOS will return address ranges describing system memory and ISA or PCI memory that is contiguous with that system memory.
2. The BIOS *will not* return a range description for the memory mapping of PCI devices. ISA Option ROM's, and ISA plug & play cards. This is because the OS has mechanisms available to detect them.
3. The BIOS will return chipset defined address holes that are not being used by devices as reserved.
4. Address ranges defined for memory mapped I/O devices (for example APICs) will be returned as reserved.
5. All occurrences of the system BIOS will be mapped as reserved. This includes the area below 1 MB, at 16 MB (if present) and at end of the address space (4 GB).
6. Standard I/O address ranges will not be reported. Example video memory at A0000 to BFFFF physical will not be described by this function. The range from E0000 to EFFFF is motherboard-specific and will be reported differently on different computers.

7. All of lower memory is reported as normal memory. It is OS's responsibility to handle standard RAM locations reserved for specific uses, for example: the interrupt vector table (0:0) and the BIOS data area (40:0).

Here we explain an example address map. This sample address map describes a machine which has 128 MB RAM, 640K of base memory and 127 MB extended. The base memory has 639K available for the user and 1K for an extended BIOS data area. There is a 4 MB Linear Frame Buffer (LFB) based at 12 MB. The memory hole created by the chipset is from 8 M to 16 M. There are memory mapped APIC devices in the system. The IO Unit is at FEC00000 and the Local Unit is at FEE00000. The system BIOS is remapped to 4G - 64K.

Note that the 639K endpoint of the first memory range is also the base memory size reported in the BIOS data segment at 40:13.

Key to types: *ARM* is AddressRangeMemory, *ARR* is AddressRangeReserved.

Base (Hex)	Length	Type	Description
0000 0000	639K	ARM	Available Base memory - typically the same value as is returned via the INT 12 function.
0009 FC00	1K	ARR	Memory reserved for use by the BIOS(s). This area typically includes the Extended BIOS data area.
000F 0000	64K	ARR	System BIOS.
0010 0000	7M	ARM	Extended memory, this is not limited to the 64MB address range.
0080 0000	8M	ARR	Chipset memory hole required to support the LFB mapping at 12 MB.
0100 0000	120M	ARM	Base board RAM relocated above a chipset memory hole.
FEC0 0000	4K	ARR	IO APIC memory mapped I/O at FEC00000. Note the range of addresses required for an APIC device may vary from one motherboard manufacturer to another
FEE0 0000	4K	ARR	Local APIC memory mapped I/O at FEE00000.
FFFF 0000	64K	ARR	Remapped System BIOS at end of address space.

The following code segment is intended to describe the algorithm needed when calling the Query System Address Map function. It is an implementation example and uses non standard mechanisms.

```
E820Present = FALSE;
Regs.ebx = 0;
do
  {
    Regs.eax = 0xE820;
    Regs.es = SEGMENT (&Descriptor);
    Regs.di = OFFSET (&Descriptor);
    Regs.ecx = sizeof (Descriptor);
    Regs.edx = 'SMAP';
```

```
        _int (0x15, Regs);

        if ((Regs.eflags & EFLAGS_CARRY) || Regs.eax != 'SMAP')
          {
            break;
          }

        if (Regs.ecx < 20 || Regs.ecx > sizeof (Descriptor))
          {
            /* bug in bios - all returned descriptors must be at
               least 20 bytes long, and can not be larger than
               the input buffer.  */
            break;
          }

        E820Present = TRUE;
        .
        .
        .
        Add address range Descriptor.BaseAddress through
        Descriptor.BaseAddress + Descriptor.Length
        as type Descriptor.Type
        .
        .
        .
      }
    while (Regs.ebx != 0);

    if (! E820Present)
      {
        .
        .
        .
        call INT 15H, AX E801h and/or INT 15H, AH=88h to obtain old style
        memory information
        .
        .
        .
      }
```

25.2 INT 15H, AX=E801h interrupt call

Real mode only.

Originally defined for EISA servers, this interface is capable of reporting up to 4 GB of RAM. While not nearly as flexible as E820h, it is present in many more systems.

Input:

`AX`	Function Code	E801h.

Output:

`CF`	Carry Flag	Non-Carry - indicates no error.
`AX`	Extended 1	Number of contiguous KB between 1 and 16 MB, maximum 0x3C00 = 15 MB.
`BX`	Extended 2	Number of contiguous 64KB blocks between 16 MB and 4GB.
`CX`	Configured 1	Number of contiguous KB between 1 and 16 MB, maximum 0x3c00 = 15 MB.
`DX`	Configured 2	Number of contiguous 64KB blocks between 16 MB and 4 GB.

Not sure what this difference between the *Extended* and *Configured* numbers are, but they appear to be identical, as reported from the BIOS.

It is possible for a machine using this interface to report a memory hole just under 16 MB (Count 1 is less than 15 MB, but Count 2 is non-zero).

25.3 INT 15H, AX=88h interrupt call

Real mode only.

This interface is quite primitive. It returns a single value for contiguous memory above 1 MB. The biggest limitation is that the value returned is a 16-bit value, in KB, so it has a maximum saturation of just under 64 MB even presuming it returns as much as it can. On some systems, it won't return anything above the 16 MB boundary.

The one useful point is that it works on every PC available.

Input:

`AH`	Function Code	88h

Output:

`CF`	Carry Flag	Non-Carry - indicates no error.
`AX`	Memory Count	Number of contiguous KB above 1 MB.

26 INT 13H disk I/O interrupts

In the PC world, living with the BIOS disk interface is definitely a nightmare. This section documents how awful the chaos is and how GRUB deals with the BIOS disks.

26.1 CHS addressing and LBA addressing

CHS — Cylinder/Head/Sector — is the traditional way to address sectors on a disk. There are at least two types of CHS addressing; the CHS that is used at the INT 13H interface and the CHS that is used at the ATA device interface. In the MFM/RLL/ESDI and early ATA days the CHS used at the INT 13H interface was the same as the CHS used at the device interface.

Today we have CHS translating BIOS types that can use one CHS at the INT 13H interface and a different CHS at the device interface. These two types of CHS will be called the logical CHS or *L-CHS* and the physical CHS or *P-CHS* in this section. L-CHS is the CHS used at the INT 13H interface and P-CHS is the CHS used at the device interface.

The L-CHS used at the INT 13 interface allows up to 256 heads, up to 1024 cylinders and up to 63 sectors. This allows support of up to 8GB drives. This scheme started with either ESDI or SCSI adapters many years ago.

The P-CHS used at the device interface allows up to 16 heads up to 65535 cylinders, and up to 63 sectors. This allows access to about 2^26 sectors (32GB) on an ATA device. When a P-CHS is used at the INT 13H interface it is limited to 1024 cylinders, 16 heads and 63 sectors. This is where the old 528MB limit originated.

LBA — Logical Block Address — is another way of addressing sectors that uses a simple numbering scheme starting with zero as the address of the first sector on a device. The ATA standard requires that cylinder 0, head 0, sector 1 address the same sector as addressed by LBA 0. LBA addressing can be used at the ATA interface if the ATA device supports it. LBA addressing is also used at the INT 13H interface by the AH=4xH read/write calls.

ATA devices may also support LBA at the device interface. LBA allows access to approximately 2^28 sectors (137GB) on an ATA device.

A SCSI host adapter can convert a L-CHS directly to an LBA used in the SCSI read/write commands. On a PC today, SCSI is also limited to 8GB when CHS addressing is used at the INT 13H interface.

First, all OS's that want to be co-resident with another OS (and that is all of the PC based OS's that we know of) *must* use INT 13H to determine the capacity of a hard disk. And that capacity information *must* be determined in L-CHS mode. Why is this? Because:

1. FDISK and the partition tables are really L-CHS based.
2. MS/PC DOS uses INT 13H AH=02H and AH=03H to read and write the disk and these BIOS calls are L-CHS based.
3. The boot processing done by the BIOS is all L-CHS based.

During the boot processing, all of the disk read accesses are done in L-CHS mode via INT 13H and this includes loading the first of the OS's kernel code or boot manager's code.

Second, because there can be multiple BIOS types in any one system, each drive may be under the control of a different type of BIOS. For example, drive 80H (the first hard drive) could be controlled by the original system BIOS, drive 81H (the second drive) could be controlled by a option ROM BIOS and drive 82H (the third drive) could be controlled by a software driver. Also, be aware that each drive could be a different type, for example, drive 80H could be an MFM drive, drive 81H could be an ATA drive, drive 82H could be a SCSI drive.

Third, not all OS's understand or use BIOS drive numbers greater than 81H. Even if there is INT 13H support for drives 82H or greater, the OS may not use that support.

Fourth, the BIOS INT 13H configuration calls are:

AH=08H, Get Drive Parameters

This call is restricted to drives up to 528MB without CHS translation and to drives up to 8GB with CHS translation. For older BIOS with no support for

>1024 cylinders or >528MB, this call returns the same CHS as is used at the ATA interface (the P-CHS). For newer BIOS's that do support >1024 cylinders or >528MB, this call returns a translated CHS (the L-CHS). The CHS returned by this call is used by FDISK to build partition records.

AH=41H, Get BIOS Extensions Support
: This call is used to determine if the IBM/Microsoft Extensions or if the Phoenix Enhanced INT 13H calls are supported for the BIOS drive number.

AH=48H, Extended Get Drive Parameters
: This call is used to determine the CHS geometries, LBA information and other data about the BIOS drive number.

An ATA disk must implement both CHS and LBA addressing and must at any given time support only one P-CHS at the device interface. And, the drive must maintain a strict relationship between the sector addressing in CHS mode and LBA mode. Quoting *the ATA-2 document*:

```
LBA = ( (cylinder * heads_per_cylinder + heads )
        * sectors_per_track ) + sector - 1

where heads_per_cylinder and sectors_per_track are the current
translation mode values.
```

This algorithm can also be used by a BIOS or an OS to convert a L-CHS to an LBA.

This algorithm can be reversed such that an LBA can be converted to a CHS:

```
cylinder = LBA / (heads_per_cylinder * sectors_per_track)
    temp = LBA % (heads_per_cylinder * sectors_per_track)
    head = temp / sectors_per_track
  sector = temp % sectors_per_track + 1
```

While most OS's compute disk addresses in an LBA scheme, an OS like DOS must convert that LBA to a CHS in order to call INT 13H.

The basic problem is that there is no requirement that a CHS translating BIOS followed these rules. There are many other algorithms that can be implemented to perform a similar function. Today, there are at least two popular implementations: the Phoenix implementation (described above) and the non-Phoenix implementations. Because a protected mode OS that does not want to use INT 13H must implement the same CHS translation algorithm. If it doesn't, your data gets scrambled.

In the perfect world of tomorrow, maybe only LBA will be used. But today we are faced with the following problems:

- Some drives >528MB don't implement LBA.
- Some drives are optimized for CHS and may have lower performance when given commands in LBA mode. Don't forget that LBA is something new for the ATA disk designers who have worked very hard for many years to optimize CHS address handling. And not all drive designs require the use of LBA internally.
- The L-CHS to LBA conversion is more complex and slower than the bit shifting L-CHS to P-CHS conversion.
- DOS, FDISK and the MBR are still CHS based — they use the CHS returned by INT 13H AH=08H. Any OS that can be installed on the same disk with DOS must understand CHS addressing.

- The BIOS boot processing and loading of the first OS kernel code is done in CHS mode — the CHS returned by INT 13H AH=08H is used.
- Microsoft has said that their OS's will not use any disk capacity that can not also be accessed by INT 13H AH=0xH.

These are difficult problems to overcome in today's industry environment. The result: chaos.

26.2 INT 13H, AH=0xh interrupt call

Real mode only. These functions are the traditional CHS mode disk interface. GRUB calls them only if LBA mode is not available.

INT 13H, AH=02h reads sectors into memory.

Input:

`AH` 02h

`AL` The number of sectors to read (must be non-zero).

`CH` Low 8 bits of cylinder number.

`CL` Sector number in bits 0-5, and high 2 bits of cylinder number in bits 6-7.

`DH` Head number.

`DL` Drive number (bit 7 set for hard disk).

`ES:BX` Data buffer.

Output:

`CF` Set on error.

`AH` Status.

`AL` The number of sectors transferred (only valid if CF set for some BIOSes).

INT 13H, AH=03h writes disk sectors.

Input:

`AH` 03h

`AL` The number of sectors to write (must be non-zero).

`CH` Low 8 bits of cylinder number.

`CL` Sector number in bits 0-5, and high 2 bits of cylinder number in bits 6-7.

`DH` Head number.

`DL` Drive number (bit 7 set for hard disk).

`ES:BX` Data buffer.

Output:

`CF` Set on error.

`AH` Status.

`AL` The number of sectors transferred (only valid if CF set for some BIOSes).

INT 13H, AH=08h returns drive parameters. For systems predating the IBM PC/AT, this call is only valid for hard disks.

Input:

`AH` 08h

`DL` Drive number (bit 7 set for hard disk).

Output:

`CF` Set on error.

`AH` 0.

`AL` 0 on at least some BIOSes.

`BL` Drive type (AT/PS2 floppies only).

`CH` Low 8 bits of maximum cylinder number.

`CL` Maximum sector number in bits 0-5, and high 2 bits of maximum cylinder number in bits 6-7.

`DH` Maximum head number.

`DL` The number of drives.

`ES:DI` Drive parameter table (floppies only).

26.3 INT 13H, AH=4xh interrupt call

Real mode only. These functions are IBM/MS INT 13 Extensions to support LBA mode. GRUB uses them if available so that it can read/write over 8GB area.

INT 13, AH=41h checks if LBA is supported.

Input:

`AH` 41h.

`BX` 55AAh.

`DL` Drive number.

Output:

`CF` Set on error.

`AH` Major version of extensions (10h for 1.x, 20h for 2.0 / EDD-1.0, 21h for 2.1 / EDD-1.1 and 30h for EDD-3.0) if successful, otherwise 01h (the error code of *invalid function*).

`BX` AA55h if installed.

`AL` Internal use.

`CX`	API subset support bitmap (see below).
`DH`	Extension version.

The bitfields for the API subset support bitmap are[2]:

Bit(s)	Description
0	Extended disk access functions (AH=42h-44h, 47h, 48h) supported.
1	Removable drive controller functions (AH=45h, 46h, 48h, 49h, INT 15H, AH=52h) supported.
2	Enhanced disk drive (EDD) functions (AH=48h, 4Eh) supported.
3-15	Reserved (0).

INT 13, AH=42h reads sectors into memory.

Input:

`AH`	42h.
`DL`	Drive number.
`DS:SI`	Disk Address Packet (see below).

Output:

`CF`	Set on error.
`AH`	0 if successful, otherwise error code.

The format of *Disk Address Packet* is:

Offset (hex)	Size (byte)	Description
00	1	10h (The size of packet).
01	1	Reserved (0).
02	2	The number of blocks to transfer (max 007F for Phoenix EDD).
04	4	Transfer buffer (SEGMENT:OFFSET).
08	8	Starting absolute block number.

INT 13, AH=43h writes disk sectors.

Input:

`AH`	43h.
`AL`	Write flags (In version 1.0 and 2.0, bit 0 is the flag for *verify write* and other bits are reserved (0). In version 2.1, 00h and 01h indicates *write without verify*, and 02h indicates *write with verify*.
`DL`	Drive number.

[2] It is known that (at least) the AMI BIOS in SuperMicro P6SBA motherboard (AMIBIOSC0631) does *not* return the bitfields correctly.

`DS:SI` Disk Address Packet (see above).

Output:

`CF` Set on error.

`AH` 0 if successful, otherwise error code.

INT 13, AH=48h returns drive parameters. GRUB only makes use of the total number of sectors, and ignore the CHS information, because only L-CHS makes sense. See Section 26.1 [CHS Translation], page 49, for more information.

Input:

`AH` 48h.

`DL` Drive number.

`DS:SI` Buffer for drive parameters (see below).

Output:

`CF` Set on error.

`AH` 0 if successful, otherwise error code.

The format of drive parameters is:

Offset (hex)	Size (byte)	Description
00	2	The size of buffer. Before calling this function, set to the maximum buffer size, at least 1Ah. The size actually filled is returned (1Ah for version 1.0, 1Eh for 2.x and 42h for 3.0).
02	2	Information flags (see below).
04	4	The number of physical cylinders.
08	4	The number of physical heads.
0C	4	The number of physical sectors per track.
10	8	The total number of sectors.
18	2	The bytes per sector.
v2.0 and later		
1A	4	EDD configuration parameters.
v3.0		
1E	2	Signature BEDD to indicate presence of Device Path information.
20	1	The length of Device Path information, including signature and this byte (24h for version 3.0).

21	3	Reserved (0).
24	4	ASCIZ name of host bus (‘ISA’ or ‘PCI’).
28	8	ASCIZ name of interface type (‘ATA’, ‘ATAPI’, ‘SCSI’, ‘USB’, ‘1394’ or ‘FIBRE’).
30	8	Interface Path.
38	8	Device Path.
40	1	Reserved (0).
41	1	Checksum of bytes 1Eh-40h (2’s complement of sum, which makes the 8 bit sum of bytes 1Eh-41h equal to 00h).

The information flags are:

Bit(s)	Description
0	DMA boundary errors handles transparently.
1	CHS information is valid.
2	Removable drive.
3	Write with verify supported.
4	Drive has change-line support (required if drive is removable).
5	Drive can be locked (required if drive is removable).
6	CHS information set to maximum supported values, not current media.
7-15	Reserved (0).

27 The structure of Master Boot Record

A Master Boot Record (*MBR*) is the sector at cylinder 0, head 0, sector 1 of a hard disk. A MBR-like structure must be created in each of partitions by the FDISK program.

At the completion of your system’s Power On Self Test (*POST*), INT 19H is called. Usually INT 19 tries to read a boot sector from the first floppy drive[3]. If a boot sector is found on the floppy disk, that boot sector is read into memory at location 0000:7C00 and INT 19H jumps to memory location 0000:7C00. However, if no boot sector is found on the first floppy drive, INT 19H tries to read the MBR from the first hard drive. If an MBR is found it is read into memory at location 0000:7C00 and INT 19H jumps to memory location 0000:7C00. The small program in the MBR will attempt to locate an active (bootable) partition in its partition table[4]. The small program in the boot sector must locate the first part of the operating system’s kernel loader program (or perhaps the kernel itself or perhaps a *boot manager program*) and read that into memory.

[3] Which drive is read first depends on your BIOS settings.

[4] This behavior is DOS MBR’s, and GRUB ignores the active flag.

INT 19H is also called when the ⟨CTRL⟩-⟨ALT⟩-⟨DEL⟩ keys are used. On most systems, ⟨CTRL⟩-⟨ALT⟩-⟨DEL⟩ causes an short version of the POST to be executed before INT 19H is called.

The stuff is:

Offset 0000
: The address where the MBR code starts.

Offset 01BE
: The address where the partition table starts (see Chapter 28 [Partition table], page 57).

Offset 01FE
: The signature, AA55.

However, the first 62 bytes of a boot sector are known as the BIOS Parameter Block (*BPB*), so GRUB cannot use these bytes for its own purpose.

If an active partition is found, that partition's boot record is read into 0000:7C00 and the MBR code jumps to 0000:7C00 with `SI` pointing to the partition table entry that describes the partition being booted. The boot record program uses this data to determine the drive being booted from and the location of the partition on the disk.

The first byte of an active partition table entry is 80. This byte is loaded into the `DL` register before INT 13H is called to read the boot sector. When INT 13H is called, `DL` is the BIOS device number. Because of this, the boot sector read by this MBR program can only be read from BIOS device number 80 (the first hard disk). This is one of the reasons why it is usually not possible to boot from any other hard disk.

28 The format of partition table

28.1 Overview the partition table

FDISK creates all partition records (sectors). The primary purpose of a partition record is to hold a partition table. The rules for how FDISK works are unwritten but so far most FDISK programs seem to follow the same basic idea.

First, all partition table records (sectors) have the same format. This includes the partition table record at cylinder 0, head 0, sector 1 – what is known as the Master Boot Record (MBR). The last 66 bytes of a partition table record contain a partition table and a 2 byte signature. The first 446 bytes of these sectors usually contain a program but only the program in the MBR is ever executed (so extended partition table records could contain something other than a program in the first 466 bytes). For more information, see Chapter 27 [MBR], page 56.

Second, extended partitions are *nested* inside one another and extended partition table records form a *linked list*. We will attempt to show this in a diagram at Section 28.3 [Partition entry format], page 60.

Each partition table entry is 16 bytes and contains things like the start and end location of a partition in CHS, the start in LBA, the size in sectors, the partition *type* and the *active* flag. Older versions of FDISK may compute incorrect LBA or size values. And when your computer boots itself, only the CHS fields of the partition table entries are used (another reason LBA doesn't solve the >528MB problem). The CHS fields in the partition tables are in L-CHS format, see Section 26.1 [CHS Translation], page 49.

28.2 The list of the *type* code

There is no central clearing house to assign the codes used in the one byte *type* field. But codes are assigned (or used) to define most every type of file system that anyone has ever implemented on the x86 PC: 12-bit FAT, 16-bit FAT, HPFS, NTFS, etc. Plus, an extended partition also has a unique type code.

In the FDISK program '`sfdisk`', the following list is assumed:

00	Empty
01	DOS 12-bit FAT
02	XENIX /
03	XENIX /usr
04	DOS 16-bit FAT <32M
05	DOS Extended
06	DOS 16-bit FAT >=32M
07	HPFS / NTFS
08	AIX boot or SplitDrive
09	AIX data or Coherent
0A	OS/2 Boot Manager
0B	Windows95 FAT32
0C	Windows95 FAT32 (LBA)
0E	Windows95 FAT16 (LBA)
0F	Windows95 Extended (LBA)
10	OPUS
11	Hidden DOS FAT12
12	Compaq diagnostics
14	Hidden DOS FAT16
16	Hidden DOS FAT16 (big)
17	Hidden HPFS/NTFS
18	AST Windows swapfile
24	NEC DOS
3C	PartitionMagic recovery

40	Venix 80286
41	Linux/MINIX (sharing disk with DRDOS)
42	SFS or Linux swap (sharing disk with DRDOS)
43	Linux native (sharing disk with DRDOS)
50	DM (disk manager)
51	DM6 Aux1 (or Novell)
52	CP/M or Microsoft SysV/AT
53	DM6 Aux3
54	DM6
55	EZ-Drive (disk manager)
56	Golden Bow (disk manager)
5C	Priam Edisk (disk manager)
61	SpeedStor
63	GNU Hurd or Mach or Sys V/386 (such as ISC UNIX)[5]
64	Novell Netware 286
65	Novell Netware 386
70	DiskSecure Multi-Boot
75	PC/IX
77	QNX4.x
78	QNX4.x 2nd part
79	QNX4.x 3rd part
80	MINIX until 1.4a
81	MINIX / old Linux
82	Linux swap
83	Linux native[6]
84	OS/2 hidden C: drive
85	Linux extended
86	NTFS volume set
87	NTFS volume set
93	Amoeba

[5] But the reason why they decided that 63 means GNU Hurd is not known. Do not use 63 for GNU Hurd.

[6] This is not true. Use 83 for ext2fs even if the owner OS is GNU/Hurd.

94	Amoeba BBT
A0	IBM Thinkpad hibernation
A5	BSD/386
A7	NeXTSTEP 486
B7	BSDI fs
B8	BSDI swap
C1	DRDOS/sec (FAT-12)
C4	DRDOS/sec (FAT-16, < 32M)
C6	DRDOS/sec (FAT-16, >= 32M)
C7	Syrinx
DB	CP/M or Concurrent CP/M or Concurrent DOS or CTOS
E1	DOS access or SpeedStor 12-bit FAT extended partition
E3	DOS R/O or SpeedStor
E4	SpeedStor 16-bit FAT extended partition < 1024 cyl.
F1	SpeedStor
F2	DOS 3.3+ secondary
F4	SpeedStor large partition
FE	SpeedStor >1024 cyl. or LANstep
FF	Xenix Bad Block Table

28.3 The format of the table entry

The 16 bytes of a partition table entry are used as follows:

```
+--- Bit 7 is the active partition flag, bits 6-0 are zero.
|
|    +--- Starting CHS in INT 13 call format.
|    |
|    |      +--- Partition type byte.
|    |      |
|    |      |    +--- Ending CHS in INT 13 call format.
|    |      |    |
|    |      |    |        +-- Starting LBA.
|    |      |    |        |
|    |      |    |        |       +-- Size in sectors.
|    |      |    |        |       |
v <--+--->  v <--+-->     v       v

0  1  2  3  4  5  6  7  8 9 A B  C D E F
DH DL CH CL TB DL CH CL LBA..... SIZE....

80 01 01 00 06 0e be 94 3e000000 0c610900  1st entry

00 00 81 95 05 0e fe 7d 4a610900 724e0300  2nd entry

00 00 00 00 00 00 00 00 00000000 00000000  3rd entry

00 00 00 00 00 00 00 00 00000000 00000000  4th entry
```

Bytes 0-3 are used by the small program in the Master Boot Record to read the first sector of an active partition into memory. The *DH*, *DL*, *CH* and *CL* above show which x86 register is loaded when the MBR program calls INT 13H AH=02h to read the active partition's boot sector. For more information, see Chapter 27 [MBR], page 56.

These entries define the following partitions:

1. The first partition, a primary partition DOS FAT, starts at CHS 0H,1H,1H (LBA 3EH) and ends at CHS 294H,EH,3EH with a size of 9610CH sectors.
2. The second partition, an extended partition, starts at CHS 295H,0H,1H (LBA 9614AH) and ends at CHS 37DH,EH,3EH with a size of 34E72H sectors.
3. The third and fourth table entries are unused.

28.4 Some basic rules for partition table

Keep in mind that there are *no* written rules and *no* industry standards on how FDISK should work but here are some basic rules that seem to be followed by most versions of FDISK:

1. In the MBR there can be 0-4 *primary* partitions, OR, 0-3 primary partitions and 0-1 extended partition entry.
2. In an extended partition there can be 0-1 *secondary* partition entries and 0-1 extended partition entries.
3. Only 1 primary partition in the MBR can be marked *active* at any given time.

4. In most versions of FDISK, the first sector of a partition will be aligned such that it is at head 0, sector 1 of a cylinder. This means that there may be unused sectors on the track(s) prior to the first sector of a partition and that there may be unused sectors following a partition table sector.

   For example, most new versions of FDISK start the first partition (primary or extended) at cylinder 0, head 1, sector 0. This leaves the sectors at cylinder 0, head 0, sectors 2...n as unused sectors. This same layout may be seen on the first track of an extended partition. See example 2 below.

   Also note that software drivers like Ontrack's Disk Manager depend on these unused sectors because these drivers will *hide* their code there (in cylinder 0, head 0, sectors 2...n). This is also a good place for boot sector virus programs to hang out.
5. The partition table entries (slots) can be used in any order. Some versions of FDISK fill the table from the bottom up and some versions of FDISK fill the table from the top down. Deleting a partition can leave an unused entry (slot) in the middle of a table.
6. And then there is the *hack* that some newer OS's (OS/2 and Linux) use in order to place a partition spanning or passed cylinder 1024 on a system that does not have a CHS translating BIOS. These systems create a partition table entry with the partition's starting and ending CHS information set to all FFH. The starting and ending LBA information is used to describe the location of the partition. The LBA can be converted back to a CHS — most likely a CHS with more than 1024 cylinders. Since such a CHS can't be used by the system BIOS, these partitions can not be booted or accessed until the OS's kernel and hard disk device drivers are loaded. It is not known if the systems using this *hack* follow the same rules for the creation of these type of partitions.

There are *no* written rules as to how an OS scans the partition table entries so each OS can have a different method. For DOS, this means that different versions could assign different drive letters to the same FAT file system partitions.

Appendices and Indices

Appendix A Frequently asked questions

How does GNU GRUB differ from Erich's original GRUB?

GNU GRUB is the successor of Erich's great GRUB. He couldn't work on GRUB because of some other tasks, so the current maintainers OKUJI Yoshinori and Gordon Matzigkeit took over the maintainership, and opened the development in order for everybody to participate it.

Technically speaking, GNU GRUB has many features that are not seen in the original GRUB. For example, GNU GRUB can be installed on UNIX-like operating system (such as GNU/Hurd) via the grub shell '`/sbin/grub`' (or '`/usr/sbin/grub`' on older systems), it supports Logical Block Address (LBA) mode that solves the 1024 cylinders problem, and ⟨TAB⟩ completes a filename when it's unique. Of course, many bug fixes are done as well, so it is recommended to use GNU GRUB.

Can GRUB boot my operating system from over 8GB hard disks?

That depends on your BIOS and your operating system. You must make sure that your drive is accessible in LBA mode. Generally, that is configurable in BIOS setting utility. Read the manual for your BIOS for more information.

Furthermore, some operating systems (i.e. DOS) cannot access any large disk, so the problem is not solved by any kind of boot loader. GNU/Hurd and GNU/Linux can surely boot from such a large disk.

Can I put Stage2 into a partition which is over 1024 cylinders?

Yes, if your BIOS supports the LBA mode.

How to create a GRUB boot floppy with the menu interface?

The easiest way is:

1. Create filesystem in your floppy disk. For example:
   ```
   $ mke2fs /dev/fd0
   ```
2. Mount it on somewhere, say, '`/mnt`'.
3. Copy the GRUB images to '`/mnt/boot/grub`'. Only '`stage1`', '`stage2`' and '`menu.lst`' are necessary. You may not copy *stage1.5*s.
4. Run the following command (substitute '`/usr/sbin/grub`' for '`/sbin/grub`' if you are using an older system):
   ```
   $ /sbin/grub --batch <<EOT
   root (fd0)
   setup (fd0)
   quit
   EOT
   ```

How to specify a partition?

See Section 11.1 [Device syntax], page 18.

GRUB does not recognize my GNU/Hurd partition.

I don't know why, but the authors of FDISK programs have assigned the partition type '0x63' to GNU Hurd incorrectly. A partition type should mean what format is used in the partition, such as filesystem and BSD slices, and should not be used to represent what operating system owns the partition. So use '0x83' if the partition contains ext2fs filesystem, and use '0xA5' if the partition contains ffs filesystem, whether the partition owner is Hurd or not. We will use '0x63' for GNU Hurd filesystem that has not been implemented yet.

I've installed a recent version of binutils, but GRUB still crashes.

Please check for the version of your binutils by this command:

```
$ ld -v
```

This will show two versions, but only the latter is important. If the version is identical with what you have installed, the installation was not bad.

Well, please try:

```
$ gcc -Wl,-v 2>&1 | grep "GNU ld"
```

If this is not identical with the result above, you should specify the directory where you have installed binutils for the script configure, like this:

```
$ ./configure --with-binutils=/usr/local/bin
```

If you follow the instructions above but GRUB still crashes, probably there is a serious bug in GRUB. See Appendix C [Reporting bugs], page 66.

GRUB hangs up when accessing my SCSI disk.

Check if you have turned on the support for INT 13 extension (LBA). If so, disable the support and see if GRUB can now access your SCSI disk. This will make it clear that your SCSI BIOS sucks.

For now, we know the following doesn't provide working LBA mode:

Adaptec AIC-7880

In the case where you have such a SCSI controller unfortunately, you cannot use the LBA mode, though GRUB still works fine in the CHS mode (so the well-known 1024 cylinders problem comes again to you).

Caution: Actually it has not been verified yet if this bug is due to the SCSI BIOS or GRUB itself, frankly speaking. Because the developers haven't seen it by their own eyes. This is why it is desirable that you investigate the cause seriously if you have the skill.

How can I specify an arbitrary memory size to Linux?

Pass a 'mem=' option to your Linux kernel, like this:

```
grub> kernel /vmlinuz mem=128M
```

You may pass other options in the same way. See See Section 4.2.2 [GNU/Linux], page 6, for more details.

I have a separate boot partition and GRUB doesn't recognize it.

This is often reported as a *bug*, but this is not a bug really. This is a feature.

Because GRUB is a boot loader and it normally runs under no operating system, it doesn't know where a partition is mounted under your operating systems. So, if you have the partition '`/boot`' and you install GRUB images into the directory '`/boot/grub`', GRUB recognizes that the images lies under the directory '`/grub`' but not '`/boot/grub`'. That's fine, since there is no guarantee that all of your operating systems mount the same partition as '`/boot`'.

There are several solutions for this situation.

1. Install GRUB into the directory '`/boot/boot/grub`' instead of '`/boot/grub`'. This may sound ugly but should work fine.
2. Create a symbolic link before installing GRUB, like '`cd /boot && ln -s . boot`'. This works only if the filesystem of the boot partition supports symbolic links and GRUB supports the feature as well.
3. Install GRUB with the command `install`, to specify the paths of GRUB images explicitly. Here is an example:

```
grub> root (hd0,1)
grub> install /grub/stage1 d (hd0) /grub/stage2 p /grub/menu.lst
```

How to uninstall GRUB from my hard disk drive?

There is no concept *uninstall* in boot loaders, because if you *uninstall* a boot loader, an unbootable machine would simply remain. So all you need to do is overwrite another boot loader you like to your disk, that is, install the boot loader without uninstalling GRUB.

For example, if you want to install the boot loader for Windows, just run `FDISK /MBR` on Windows. If you want to install LILO[1], run `/sbin/lilo` on GNU/Linux.

GRUB hangs when accessing my large IDE disk.

If your disk is bigger than 32GB, probably updating your mainboard BIOS will solve your problem. This bug is well-known and most vendors should provide fixed versions. For example, if you have ASUS-P3BF, upgrading the BIOS to V1007beta1 or later can fix it. Please ask your vendor, for more information.

Why don't Linux, FreeBSD, NetBSD, etc. become Multiboot-compliant?

Please ask the relevant maintainers. If all free kernels were Multiboot-compliant (see section "Motivation" in *The Multiboot Specification*), the world would be an utopia. . .

Appendix B How to obtain and build GRUB

> **Caution:** GRUB requires binutils-2.9.1.0.23 or later because the GNU assembler has been changed so that it can produce real 16bits machine code between 2.9.1 and 2.9.1.0.x. See `http://sourceware.cygnus.com/binutils/`, to obtain information on how to get the latest version.

GRUB is available from the GNU alpha archive site `alpha.gnu.org:/gnu/grub` or any of its mirrors. The file will be named grub-version.tar.gz. The current version is 0.5.97, so the file you should grab is:

`ftp://alpha.gnu.org/gnu/grub/grub-0.5.97.tar.gz`

[1] I can't imagine why you want to do such a thing, though

To unbundle GRUB use the instruction:

```
zcat grub-0.5.97.tar.gz | tar xvf -
```

which will create a directory called 'grub-0.5.97' with all the sources. You can look at the file 'INSTALL' for detailed instructions on how to build and install GRUB, but you should be able to just do:

```
$ cd grub-0.5.97
$ ./configure
$ make install
```

This will install the grub shell 'grub' (see Chapter 15 [Invoking the grub shell], page 35), the Multiboot checker 'mbchk' (see Chapter 18 [Invoking mbchk], page 38), and the GRUB images It will also install the GRUB manual.

Also, the latest version is available from the CVS. The repository is:

`:pserver:anoncvs@subversions.gnu.org:/home/cvs`

and the module is:

`grub`

The password for anoncvs is empty. So the instruction is:

```
$ cvs -d :pserver:anoncvs@subversions.gnu.org:/home/cvs \
    login
Password: ⟨ENTER⟩
$ cvs -d :pserver:anoncvs@subversions.gnu.org:/home/cvs \
    checkout grub
```

Get the recent version of GNU Automake from the CVS to regenerate 'Makefile.in's. See http://sourceware.cygnus.com/automake/, for more information.

Appendix C Reporting bugs

When you encounter any problem or bug, please submit it to bug-grub@gnu.org with information about your computer and what you did *as much as possible*. Take a look at this list before you send e-mail to the address:

- Write what you did and what messages were printed on the screen in detail. Don't paraphrase them. Please describe them as they were.
- Explain what you wanted to do. It is very useful to know your purpose and your wish, and how GRUB didn't satisfy you.
- Inform us of the information about your GRUB. What version were you using? Which were you using the grub shell or the boot images? If using the grub shell, tell us what operating system was used to run it. And, if you ran `configure` with some options when building GRUB, it would be a good thing to let us know how to build it.
- The information on your hardware is also essential. These are especially important: the geometries and the partition tables of your hard disk drives and your BIOS.
- Write down anything that you think might be related. If you are not sure whether to state a fact or leave it out, state it! Reporting too many things is quite better than omitting an important thing.

Index

B

C

D

E

F

G

H

I

K

L

M

P

Q

R

S

T

U

Table of Contents