@node Internals @appendix Hacking GRUB

This chapter documents the user-invisible aspect of GRUB.

As a general rule of software development, it is impossible to keep the descriptions of the internals up-to-date, and it is quite hard to document everything. So refer to the source code, whenever you are not satisfied with this documentation. Please assume that this gives just hints to you.

@menu * Memory map:: The memory map of various components * Embedded data:: Embedded variables in GRUB * Filesystem interface:: The generic interface for filesystems * Command interface:: The generic interface for built-ins * Bootstrap tricks:: The bootstrap mechanism used in GRUB * I/O ports detection:: How to probe I/O ports used by INT 13H * Memory detection:: How to detect all installed RAM * Low-level disk I/O:: INT 13H disk I/O interrupts * MBR:: The structure of Master Boot Record * Partition table:: The format of partition tables * Submitting patches:: Where and how you should send patches @end menu

@node Memory map @section The memory map of various components

GRUB consists of two distinct components, called @dfnstages, which are loaded at different times in the boot process. Because they run mutual-exclusively, sometimes a memory area overlaps with another memory area. And, even in one stage, a single memory area can be used for various purposes, because their usages are mutually exclusive.

Here is the memory map of the various components:

@table @asis @item 0 to 4K-1 BIOS and real mode interrupts

@item 0x07BE to 0x07FF Partition table passed to another boot loader

@item down from 8K-1 Real mode stack

@item 0x2000 to ? The optional Stage 1.5 is loaded here

@item 0x2000 to 0x7FFF Command-line buffer for Multiboot kernels and modules

@item 0x7C00 to 0x7DFF Stage 1 is loaded here by BIOS or another boot loader

@item 0x7F00 to 0x7F42 LBA drive parameters

@item 0x8000 to ? Stage2 is loaded here

@item The end of Stage 2 to 416K-1 Heap, in particular used for the menu

@item down from 416K-1 Protected mode stack

@item 416K to 448K-1 Filesystem buffer

@item 448K to 479.5K-1 Raw device buffer

@item 479.5K to 480K-1 512-byte scratch area

@item 480K to 512K-1 Buffers for various functions, such as password, command-line, cut and paste, and completion.

@item The last 1K of lower memory Disk swapping code and data @end table

See the file @filestage2/shared.h, for more information.

@node Embedded data @section Embedded variables in GRUB

Stage 1 and Stage 2 have embedded variables whose locations are well-defined, so that the installation can patch the binary file directly without recompilation of the stages.

In Stage 1, these are defined:

@table @code @item 0x3E The version number (not GRUB's, but the installation mechanism's).

@item 0x40 The boot drive. If it is 0xFF, use a drive passed by BIOS.

@item 0x41 The flag for if forcing LBA.

@item 0x42 The starting address of Stage 2.

@item 0x44 The first sector of Stage 2.

@item 0x48 The starting segment of Stage 2.

@item 0x1FE The signature (@code0xAA55). @end table

See the file @filestage1/stage1.S, for more information.

In the first sector of Stage 1.5 and Stage 2, the block lists are recorded between @codefirstlist and @codelastlist. The address of @codelastlist is determined when assembling the file @filestage2/start.S.

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

The format of a block list can be seen from the example in the code just before the @codefirstlist label. Note that it is always from the beginning of the disk, but @emphnot relative to the partition boundaries.

In the second sector of Stage 1.5 and Stage 2, these are defined:

@table @asis @item @code0x6 The version number (likewise, the installation mechanism's).

@item @code0x8 The installed partition.

@item @code0xC The saved entry number.

@item @code0x10 The identifier.

@item @code0x11 The flag for if forcing LBA.

@item @code0x12 The version string (GRUB's).

@item @code0x12 + @dfnthe length of the version string The name of a configuration file. @end table

See the file @filestage2/asm.S, for more information.

@node Filesystem interface @section The generic interface for filesystems

For any particular partition, it is presumed that only one of the @dfnnormal filesystems such as FAT, FFS, or ext2fs can be used, so there is a switch table managed by the functions in @filedisk$_{i}o.c.Thenotationisthatyoucanon$

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

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

@vtable @code @item current$_{d}riveThecurrentBIOSdrivenumber(numberedfrom0,ifafloppy,andnumberedfrom0$

@item current$_{p}artitionThecurrentpartitionnumber.$

@item current$_{s}liceThecurrentpartitiontype.$

@item saved$_{d}riveThe@dfndrivepartoftherootdevice.$

@item saved$_{p}artitionThe@dfnpartitionpartoftherootdevice.$

@item part$_{s}tartThecurrentpartitionstartingaddress,insectors.$

@item part$_{l}engthThecurrentpartitionlength,insectors.$

@item print$_{p}ossibilitiesTruewhenthe@codedirfunctionshouldprintthepossiblecompletionsofafile,andfalsewhenit$

@item FSYS$_{B}UFFilesystembufferwhichis32Kinsize,touseinanywaywhichthefilesystembackenddesires.@endvt$

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

@vtable @code @item filepos The current position in the file, in sectors.

@strongCaution: the value of @varfilepos 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 backend code!

@item filemax The length of the file.

@item disk$_{r}ead_{f}uncThevalueof@vardisk_{r}ead_{h}ook@emphonlyduringreadingofdataforthefile,notanyotherfsdata$

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

@ftable @code @item devread Only read sectors from within a partition. Sector 0 is the first sector in the partition.

@item grub$_{r}eadIfthebackendusestheblocklistcode,then@codegrub_{r}eadcanbeused,aftersetting@varblock_{f}ileto1.$

@item print$_{a}c$$ompletionIf@varprint_{p}ossibilitiesistrue,call@codeprint_{a}c$$ompletionforeachpossiblefilename.Other$

The functions expected to be defined by the filesystem backend are described at least moderately in the file @filefilesys.h. Their usage is fairly evident from their use in the functions in @filedisk$_{i}o.c,lookfortheuseofthe@varfsys$

@strongCaution: The semantics are such that then @sampmounting the filesystem, presume the filesystem buffer @codeFSYS$_{B}UFiscorrupted,and(re-)loadallimportantcontents.Whenopeningandreadingafile,presumetha$

@node Command interface @section The generic interface for built-ins

GRUB built-in commands are defined in a uniformal interface, whether they are menu-specific or can be used anywhere. The definition of a builtin command consists of two parts: the code itself and the table of the information.

The code must be a function which takes two arguments, a command-line string and flags, and returns an @sampint value. The @dfnflags argument specifies how the function is called, using a bit mask. The return value must be zero if successful, otherwise non-zero. So it is normally enough to return @varerrnum.

The table of the information is represented by the structure @codestruct builtin, which contains the name of the command, a pointer to the function, flags, a short description of the command and a long description of the command. Since the descriptions are used only for help messages interactively, you don't have to define them, if the command may not be called interactively (such as @commandtitle).

The table is finally registered in the table @varbuiltin$_{t}able, sothat@coderun_{s}criptand@codeenter_{c}mdlinecanfindthec$

@node Bootstrap tricks @section The bootstrap mechanism used in GRUB

The disk space can be used in a boot loader is very restricted because a MBR (@pxrefMBR) is only 512 bytes but it also contains a partition table (@pxrefPartition table) 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, @dfnStage 1 and @dfnStage 2 (and optionally @dfnStage 1.5). @xrefMemory map, for more information.

We embed Stage 1 in a MBR or in the boot sector of a partition, and place Stage 2 in a filesystem. The optional Stage 1.5 can be installed in a filesystem, in the @dfnboot loader area in a FFS or a ReiserFS, and in the sectors right after a MBR, because Stage 1.5 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 Stage1 must do is just load Stage2 or Stage1.5. But even if Stage 1 needs not to support the user interface or the filesystem interface, it is impossible to make Stage 1 less than 400 bytes, because GRUB should support both the CHS mode and the LBA mode (@pxrefLow-level disk I/O).

The solution used by GRUB is that Stage 1 loads only the first sector of Stage 2 (or Stage 1.5) and Stage 2 itself loads the rest. The flow of Stage 1 is:

@enumerate @item Initialize the system briefly.

@item Detect the geometry and the accessing mode of the @dfnloading drive.

@item Load the first sector of Stage 2.

@item Jump to the starting address of the Stage 2. @end enumerate

The flow of Stage 2 (and Stage 1.5) is:

@enumerate @item Load the rest of itself to the real starting address, that is, the starting address plus 512 bytes. The block lists are stored in the last part of the first sector.

@item Long jump to the real starting address. @end enumerate

Note that Stage 2 (or Stage 1.5) does not probe the geometry or the accessing mode of the @dfnloading drive, since Stage 1 has already probed them.

@node I/O ports detection @section How to probe I/O ports used by INT 13H

FIXME: I will write this chapter after implementing the new technique.

@node Memory detection @section How to detect all installed RAM

FIXME: I doubt if Erich didn't write this chapter only himself wholly, so I will rewrite this chapter.

@node Low-level disk I/O @section INT 13H disk I/O interrupts

FIXME: I'm not sure where some part of the original chapter is derived, so I will rewrite this chapter.

@node MBR @section The structure of Master Boot Record

FIXME: Likewise.

@node Partition table @section The format of partition tables

FIXME: Probably the original chapter is derived from "How It Works", so I will rewrite this chapter.

@node Submitting patches @section Where and how you should send patches

When you write patches for GRUB, please send them to the mailing list @emailbug-grub@@gnu.org. Here is the list of items of which you should take care:

@itemize @bullet @item Please make your patch as small as possible. Generally, it is not a good thing to make one big patch which changes many things. Instead, segregate features and produce many patches.

@item Use as late code as possible, for the original code. The CVS repository always has the current version (@pxrefObtaining and Building GRUB).

@item Write ChangeLog entries. @xrefChange Logs, , Change Logs, standards, GNU Coding Standards, if you don't know how to write ChangeLog.

@item Make patches in unified diff format. @sampdiff -urN is appropriate in most cases.

@item Don't make patches reversely. Reverse patches are difficult to read and use.

@item Be careful enough of the license term and the copyright. Because GRUB is under GNU General Public License, you may not steal code from software whose license is incompatible against GPL. And, if you copy code written by others, you must not ignore their copyrights. Feel free to ask GRUB maintainers, whenever you are not sure what you should do.

@item If your patch is too large to send in e-mail, put it at somewhere we can see. Usually, you shouldn't send e-mail over 20K. @end itemize