osFree Bootsequenz (Entwurf II)

Hinweis:

- Dieses Dokument befindet sich in der Entwicklungsphase.
- Die Multiboot-Spezifikation wird in begrenztem Rahmen unterstützt. Wir denken, dass nicht alle Features notwendig sind. Z.B. der Grafikmodus ist kein Problem des Bootvorganges. Der Kernelloader muss den Kernel laden, weiter nichts 
- Dieses Dokument verwendet einen Teil der Multiboot-Spezifikation Version 0.6.93

Dieses Dokument wurde erstellt von Yuri Prokushev, Valery Sedletski und Sascha Schmidt.

Einleitung

Die osFree Boot-Seqenz verwendet keine klassische Lösung wie GRUB oder LILO. GRUB ist ohne Zweifel ein gutes Programm, aber es hat einige Nachteile, die wir nicht in Kauf nehmen wollen. Der entscheidendste dieser Nachteile ist, dass der **Kernel-Loader die Dateisystem-Struktur kennen muss**. Das heißt, das man - wenn man ein neues Dateisystem verwenden will - (als Endnutzer) GRUB updaten muss, um das neue Dateisystem zu unterstützen (sofern es dafür GRUB-Unterstützung gibt). Als Entwickler muss ein neuer Dateisystem-Treiber zum Kernel und zum Kernel-Loader hinzugefügt werden. In den meisten Fällen handelt es sich dabei um verschiedene Architekturen, Programmier-Stile und Entwicklungsumgebungen. Wir wollen aber nicht das ganze System aktualisieren wegen - im Vergleich zum Gesamtsystem - kleinen Verbesserungen. Um es überspitzt darzustellen - wir wollen nicht den größten Teil der System-Komponenten neu installieren oder aktualisieren, um einen Mauszeiger mit Schatten zu haben. Daher haben wir den Ansatz der installierbaren Dateisysteme (engl.: **installable filesystems, IFS**) von OS/2 wiederverwendet. Wir gehen hier nicht in die Details von MicoFSD und MiniFSD, da das die Aufgabe eines Dokuments zu IFS ist und nicht der Dokumentation zu Kernel-Loader Details und Schnittstellen (die hier vorliegt).

Underground information (Informative)

In this text we will try to discuss some problems about the way osFree must load from disk and the system initialization process. osFree is an OS/2 clone, so it must follow the way of doing things of original OS/2. Also we must borrow the good ideas from OS/2 Warp Connect PowerPC Edition (aka Workplace OS), as it was the 1st example of microkernel OS/2, and had some essential features for microkernel OS/2 system.

<h3>OS/2 Boot sequence</h3>

<p>

At the end of POST procedure the ROM BIOS initializes devices and gives control to int 19h interrupt routine, which loads 1st sector of the 1st boot device (a floopy, HDD or another). If the device was the HDD, then the Master boot record (MBR) is loaded from the 1st sector. The ROM BIOS loads it at

address 0x7c0:0×0. The MBR has a Non-System Bootstrap (NSB) in it, and the Partition Table (PT). The NSB code relocates MBR to 0×60:0×0 and loads the bootsector of boot HDD partition at the same place (0x7c0:0×0) MBR was loaded first. The boot partition is searched in the Partition Table (PT), which is embedded in the MBR sector at the end of it. The PT contains four partition descriptors, each of which has an active flag in the 1st byte of descriptor structure. If this byte is equal to 80h, the partition is active, if it is =00h, then partition is not active. MBR transfers control to the bootsector and the bootsector loads the operating system. This part of boot sequence is the same in different PC OSes including Windows, <acronym title="Operating System">OS</acronym>/2 and Linux when loading from the active primary partition.

 In <acronym title="Operating System">OS</acronym>/2, to freely choose operating system, different from <acronym title="Operating System">OS</acronym>/2, IBM included the <acronym title="Operating System">OS</acronym>/2 Boot manager. The boot manager is installed in a small primary partition, which is marked active in PT. So, the MBR sector loads the bootsector of the <acronym title="Operating System">OS</acronym>/2 boot manager, instead of the bootsector of a corresponding <acronym title="Operating System">OS</acronym>. The boot manager gives a menu to user, from which he or she can choose a partition to continue booting from. So, the boot manager then loads a bootsector of <acronym title="Operating System">OS</acronym> boot partition at the same address 0x7c0:0×0.

 The bootsector has a BPB (Boot Parameters Block) structure in it, which describes some important parameters, needed to properly boot from the partition. The majority of them is specific for diskettes and the FAT filesystem, but some are essential for <acronym title="Operating System">OS</acronym>/2 to load properly. There are three essential parameters in the BPB for the <acronym title="Operating System">OS</acronym>/2 boot sequence. They are HiddenSectors value, the physical boot device and the logical boot device. The latter is equivalent to the boot device drive letter and is important to properly define the drive letter of the boot partition. The physical boot drive is the number of the boot physical device in the format of BIOS int 13h: The value of 00h is for 1st diskette drive, 01h for 2nd diskette, 80h for the 1st hard disk, 81h – the 2nd hard disk and so on. The HiddenSectors is important for booting <acronym title="Operating System">OS</acronym>/2 from logical disk in extended partition. For primary partitions, it is equal to the offset of the partition from the beginning of the HDD, but for logical disks it is for some reason equal to the number of sectors per track (63 for the modern hard drives). The HiddenSectors value is used to convert local sector number (from the beginning of the partition) to the global sector address (from the beginning of the HDD). It is essential for booting <acronym title="Operating System">OS</acronym> from the partition. For this reason, most OSes can load only from primary partition, like Windoze or FreeBSD. But IBM made <acronym title="Operating System">OS</acronym>/2 to be loadable from logical partitions, as well as primary ones. For this, IBM bootmanager fixes the three above mentioned values in bootsector BPB, and only after that gives control to the bootsector. (This feature is required from boot manager to properly load <acronym title="Operating System">OS</acronym>/2 from the logical partition. At present, it is available only from three bootmanagers: IBM Bootmanager, VPart from Veit Kannegieser and AirBoot from Martin Kiewitz.)

 After receiving control from MBR code or boot manager, the bootsector loads the so called blackbox code from the rest of the bootblock (the 15 sectors after the bootsector in HPFS, and about 30 sectors in bootJFS) or from the root directory from os2boot file in FAT. The blackbox is the Micro File System Driver (MicroFSD) and contains several functions to open, read and close files from the root directory of the boot drive. Also it has initialization code and cleanup code. Only one file can be opened at one time, and in IBM blackboxes, only files in the root directory can be read.

 The blackbox (aka MicroFSD) in its initialization part, loads two files from disk: os2ldr and os2boot. (os2boot in FAT contains the MicroFSD, but in other filesystems, it contains the MiniFSD

code). The os2ldr is the <acronym title=“Operating System”>OS</acronym>/2 kernel loader. It is independent from the filesystem, and only for FAT contains a special code (the above mentioned functions to read files from the FAT partition, and some code to support memory dumping and hybernating from/to the FAT partition). Besides that, os2ldr implements DosHlp functions (helpers for the <acronym title=“Operating System”>OS</acronym>/2 kernel), so, it serves like some sort of microkernel – it implements some functions the kernel depends on. Also, os2ldr contains implementation of the OEMHLP$ device driver. (Yes, OEMHLP$ resides in the loader!). So, os2ldr is more than just <acronym title=“Operating System”>OS</acronym> loader :).

 The blackbox code transfers control to the os2ldr, and gives it the info about memory layout and exports its filesystem functions (it gives the loader a FileTable structure, a pointer to the BPB, a physical boot device and some flags). The loader then relocates itself to the top of low memory, loads os2krnl from disk and applies required fixups (it does LX format parsing and placing segments to required places and corrects addresses of functions and variables for proper linkage).

 After that, control is given to os2krnl, and memory info along with os2boot MiniFSD image in memory is given to the kernel by the loader.

 For disk reading, os2krnl uses MiniFSD, not MicroFSD. MicroFSD is intended for loader to work in real mode, and MiniFSD is for the kernel to work in protect mode. MiniFSD has the format of 16-bit NE dll. Its size is limited to 62 Kbytes. The system initialization sysinit routine of os2krnl loads MiniFSD from its in-memory image. It is used at the 1st stage in system initialization process to read config.sys and load base device drivers (BASEDEV and PSD).

 Before the kernel loads <acronym title=“Operating System”>OS</acronym>/2 disk subsystem drivers (they are: ibm1flpy.add, ibm1s506.add, – disk drivers, an ATAPI filter to support CDROMs, DASD manager (os2dasd.dmd), volume manager (os2lvm.dmd)), the disk reading is performed by switching to real mode and calling int 13h BIOS disk read functions. When disk subsystem drivers are loaded and initialized, the kernel uses them to read the disk, and for filesystem access it continues to use the MiniFSD. It is the 1st phase of the system initialization process, as it called in ifs.inf documentation from IBM. At this phase the kernel uses MFS_* MiniFSD functions for filesystem access. They are very similiar to the MicroFSD functions, but unlike them, they work in protect mode. At the 2nd phase the kernel links MiniFSD into the IFS chain (it is the only IFS in chain), calls MFS_TERM function to complete phase 1 and in phase 2 it uses FS_* functions from the MiniFSD, so at this moment, MiniFSD is like ordinary IFS, but it supports only a minimum of FS_* funtions.

 After that, phase 3 begins. At this phase, the boot IFS is loaded, which replaces MiniFSD in the IFS chain. After that begins a full-featured filesystem access. The system now can read from and write to files from any disk, it supports drive letters and can have many open files at the same time. At this phase, the kernel loads the required DLLs and can load ordinary device drivers (“device=”).

 After that, the system continue to load “device=” drivers, then process “run=” and “call=” statements, and then “protshell=” to load pmshell.

<h3>Some info about OS/2 PPC load process</h3>

I have no PowerPC box, but I have IBM's redbook “<acronym title=“Operating

System”>OS</acronym>/2 Warp (PowerPC Edition) A first look”, there is a little paragraph about <acronym title=“Operating System”>OS</acronym>/2 PPC load sequence in this book. This paragraph is very small and contains only a small piece of information. Also I have <acronym title=“Operating System”>OS</acronym>/2 PPC config files for the loader (boot.cfg) and for the <acronym title=“Operating System”>OS</acronym>/2 server (config.sys). The most part of drivers, servers and libraries is loaded from boot.cfg, and the config.sys is specific to the <acronym title=“Operating System”>OS</acronym>/2 personality. The config.sys file is very small. I can show these files in forum, if you interested.

 As written in IBM's redbook, the loader (bl_auto file in the root of the disk) is loaded by the PowerPC ROM directly from boot partition, there is no a bootsector, the ROM loads the loader file immediately. The loader has a configuration file boot.cfg. In the config file, there are the microkernel file, an initial task and other files, which loader must load from disk into memory. These sevices loaded from boot.cfg are called Personality neutral (PN) services, they are independent from <acronym title=“Operating System”>OS</acronym>/2 Personality and include device drivers.

 I have no information about how the bootloader access files on disk, and if there is an equivalent to the MicroFSD in <acronym title=“Operating System”>OS</acronym>/2 PPC, but it seems that there must be something like MicroFSD, and it uses ROM functions to read the disk (analogous to the int 13h PC BIOS functions)

 The bootloader loads a number of files into memory, then starts the microkernel and the initial task. The initial task is called the bootstrap. The bootloader passes the bootstrap some information along with the locations of files it have loaded in memory. The bootstrap acts as a file server for other servers. In other words, it gives access to the files the bootloader had loaded into memory. The bootstrap task has no device drivers in it, instead it has access to the files the bootloader has loaded into memory. Then the bootstrap do the following (I will quote the text from the redbook): <pre class=“code”>[====================begin quote=======================] 1) It loads the Root Name Server 2) Starts the default pager 3) Starts the task manager 4) Provides the file services, which will be used by Task Server 5) Directs the Task Manager to start the personality neutral (PN) servers required to bring up the dominant personality. PN servers include Message Logger, Hardware Resource Manager (HRM), Bus Walkers, and Device Drivers. 6) Starts the Personality.</pre> <pre class=“code”>The bootstrap task continues to behave as a file server until it terminates. [====================end quote=========================]</pre>

Then <acronym title=“Operating System”>OS</acronym>/2 personality starts. The <acronym title=“Operating System”>OS</acronym>/2 personality server parses config.sys and loads the <acronym title=“Operating System”>OS</acronym>/2 personality specific servers. Device drivers are not specific to the <acronym title=“Operating System”>OS</acronym>/2 personality, so they are started by bootstrap and are in bootloader config file (boot.cfg), not in the config.sys file.

<h3>L4 microkernel load process. GNU GRUB bootloader and Multiboot specification</h3>

The L4 microkernel can be started either in real or protect mode. If started in real mode, it switches to protect mode by itself. The exact load procedure is described in the L4 <acronym title="Application Programming Interface">API</acronym>, version X.2 reference manual. For loading the L4 microkernel, the GNU GRUB bootloader is commonly used. The GRUB defines the Multiboot specification, which is intended to be the common protocol between the <acronym title="Operating System">OS</acronym> kernel and the bootloader. At this time only GRUB supports the multiboot specification, but it is possible to create a compatible bootloader. The multiboot compliant kernels include The HURD Mach kernel (The GNU GRUB is the official GNU project bootloader, and it was created specifically for the GNU HURD project. But it also suits for Linux, FreeBSD, OpenBSD, NetBSD, MacOS X and L4). But L4 itself is not a multiboot kernel, instead, it uses its own loader/bootstrapper which in L4Ka::Pistachio is kickstart, and in L4/Fiasco is rmgr.

 The Multiboot specification requires from the kernel to have in its first 8192 bytes the multiboot header. This header defines requirements for the loader from the kernel, such as: the load addresses of various segments of a kernel, initial video mode and kernel entry point. The kernel executable file format may be any, the only requitement is to have the multiboot header. But GRUB also directly supports the ELF and a.out formats. (But <acronym title="Operating System">OS</acronym>/2 (intel) uses LX format, and <acronym title="Operating System">OS</acronym>/2 PPC used ELF format).

 The bootloader loads a kernel (kickstart in our case) and a number of additional modules. The bootloader loads the kernel and applies fixups to it, but modules remain untouched, the loader starts the kernel and passes to it the pointer to Multiboot structure. The GRUB leaves the kernel in a simple protected mode environment with paging disabled, A20 line enabled and Interrupt Controller remains uninitialized. Also, the initial video mode is set.

 The Multiboot structure contains info about memory layout and modules. For the kernel and modules there are strings associated with them. These strings can be used as command lines for kernel and modules, or just as a labels to identify modules.

 In L4Ka:Pistachio, the kickstart bootstrapper plays the role of the multiboot kernel. It receives Multiboot structure from GRUB, and ELF-loads the L4 kernel and initial servers. Then it searches the Kernel Interface Page (KIP) inside the L4 image. It passes the multiboot info in Bootinfo structure, pointed by the field in KIP. Then kickstart fills the fields for initial servers location in the KIP (initial servers are sigma0 and roottask, which were passed by GRUB as multiboot modules). Then kickstart calls the entry point in L4 kernel.

 In the case of L4/Fiasco the role of kickstart plays the resource manager (rmgr). It consists of two stages. Stage 1 is analogous to kickstart. It parses the config and loads L4 and servers. Stage 1 passes the configuration to stage 2. Stage 2 is started by L4 and serves as a root server. But at present, rmgr is divided into 2 parts – bootstrap and roottask, which are loaded as separate multiboot modules.

 Then L4 starts, relocates itself to the proper place in memory, and then starts sigma0 and roottask. After that, the roottask can initialize the rest of the system.

<h3>The FreeLdr project history and the proposed roadmap</h3>

The FreeLdr project was started in 1999 by David Zimmerli. He wrote an article in EDM/2 (The Project to replace OS2LDR: http://www.edm2.com/0705/freeldr/freeldr.html) about os2ldr and started a project to replace the os2ldr. It was a little COM format executable which was loaded by blackbox. It accepts information from the blackbox (FileTable structure, BPB etc.), and uses the blackbox to test its filesystem access functions. FreeLdr writes the received information to the screen and COM port. It calls Open function from the blackbox to show the size of os2krnl file. So, it was for test purposes only. Then, D. Zimmerli proposed to implement the part of os2ldr functionality, but did not finish it. But sources were available from the EDM/2 article page.

 Now the osFree project took these sources, and used them as a starting point of an osFree loader implementation. The name of the loader remains the same, as it corresponds the osFree project name. The original sources were written in Turbo C/Turbo Assembler. The osFree project uses Watcom C and assembler, so the sources were ported to OpenWatcom. The porters are Sascha Schmidt and Yuri Prokushev. Now they combined the FreeLdr sources with a minimal set of routines from GRUB needed to load and start kickstart. At present, the code links with GRUB functions, but doesn't work. The problem is with calling MicroFSD functions – the mu_Open() function is called, but there is some problem with parameters passing and calling conventions. So far, the code is being debugged. We use Bochs PC emulator and its embedded debugger to debug code. (The problem appears to be simple, but we had no developers having skills in assembly language. And the program was being debugged by debug messages only. Now we have a debugger, so it's possible that the problem will be resolved soon <img src="/web/20100304104831im_/http://www.osfree.org/wiki/lib/images/smileys/icon_wink.gif" class="middle" alt="

">).

 Now I (Valery Sedletski, aka valerius) have some ideas about osFree boot sequence and bootloader design. The ideas were inspired by <acronym title="Operating System">OS</acronym>/2 PowerPC boot sequence and GRUB bootloader. But the main source of ideas is, of course, the <acronym title="Operating System">OS</acronym>/2 (intel) boot sequence. Also, some ideas were borrowed from Veit Kannegieser's os2csm config.sys editor and preprocessor.

<h3>A note about OS2CSM</h3>

Veit Kannegieser wrote a program, called os2csm. It was inspired by some DOS program, which modifies config.sys in memory. Os2csm installs itself in place of os2ldr, and renames os2ldr to os2ldr.bin. Os2csm hooks onto int 13h interrupt handler and analyzes the information read through

int 13 routine. The config.sys file contains directives, analogous to C preprocessor directives, and each 512 bytes of config.sys file (each disk sector of config.sys) has a special comment with a special signature in it. The procedure, which hooks onto int 13h interrupt handler, checks each sector for these signatures, and if it finds them, it assumes that the config.sys file is being read. So, it substitutes preprocessor defines and patches config.sys on the fly.

 The config.sys preprocessor is useful to substitute some parts of config.sys by some variables (aka preprocessor symbols). These variables can be obtained from menus, which are presented to user by os2csm. The os2csm then loads os2ldr from the file os2ldr.bin; the loader starts the kernel. When the kernel reads config.sys, os2csm changes it on the fly, and substitutes variables. So, settings set by user substitute variables in config.sys. This way, the set of parameters is passed to the kernel through config.sys preprocessor.

 OS2CSM is now used in eCS (in eCS demo disk and in eCS installation disk). You can download the demo disk from http://ecomstation.com.

 We can use this idea in our bootloader, so, we may implement a simple preprocessor with syntax similiar to the C preprocessor (or, even, PPWizard syntax). I propose to embed this preprocessor into loader and instead of hooking onto int 13 routines, we can hook onto blackbox routines. The configs are read through blackbox (not minifsd, like in present <acronym title="Operating System">OS</acronym>/2!). I suggest to explicitly mark a number of files as configs in the loader config. So, the signatures in each config.sys file sector are not needed. And also we can use more attractive and beautiful looking syntax instead of ugly os2csm syntax.

 The preprocessor directives may include analogues to "#define", "#include", "#ifdef" etc., so, it will be possible to define symbols, include one config file to another, and to conditionally include files or define symbols. So, the resulting config file can be flexibly constructed from parts and variables can be substituted in it.

 The preprocessor idea complements the GRUB idea of passing command lines to a kernel and modules. That is, we suggest to use passing parameters to the kernel as multiboot command lines, as well as through substituting variables in config files. So, we combine these two approaches.

<h3>Ideas about FreeLdr design</h3>

1) First, I suggest to combine the functionality of <acronym title="Operating System">OS</acronym>/2 bootmanager and os2ldr in one program. I.e., the boot sequence must be like this: The MBR loads an active partition, or the partition with a given number. (I already wrote such a MBR sector, it can load a bootsector from selected primary or logical partition (yes, logical partitions are supported too!)). The bootable hard disk number and the number of partition on it are written inside the MBR of the first hard disk. (The bootable partition can reside on the same HDD as well as on the different HDD, than the 1st HDD we read MBR from).

 So, the MBR loads a boot sector from bootable partition. The boot sector loads the blackbox. The blackbox loads our loader. Then, the loader starts. The loader combines the functionality of a bootloader with functionality of bootmanager: after having been called from the blackbox, the loader shows a menu to the user. The user selects a menu item from it, each menu item defines an

<acronym title="Operating System">OS</acronym> to be loaded along with parameters, which are then passed to the <acronym title="Operating System">OS</acronym> kernel. From this point, the loader/bootmanager is capable of executing the bootsectors of OSes, not supported directly by our loader, like windoze. The loader only loads a corresponding bootsector and executes it. But if an <acronym title="Operating System">OS</acronym> kernel is supported directly, then the loader can also pass some parameters to the kernel, through a config file or a command line parameters. The advantage of such an approach is that we can choose an <acronym title="Operating System">OS</acronym> and its parameters from the same place, it is a combined loader/bootmanager. An example: in present <acronym title="Operating System">OS</acronym>/2 the <acronym title="Operating System">OS</acronym>/2 bootmanager allows to choose an <acronym title="Operating System">OS</acronym>, and os2ldr allows to choose additional parameters – it allows to press a hotkey to bring up a Recovery choices menu. There are also many other parameters avaliabe by pressing Alt-F1[F2,F3,...]. In our case, there is only one menu from which an <acronym title="Operating System">OS</acronym> and its parameters can be choosen, and additional menus, like Recovery choices, are avaliable from the same place: the bootloader/bootmanager menu. So, the advantage is an integration. Also, our bootmanager resides on ordinary <acronym title="Operating System">OS</acronym>/2 partition, not special bootmanager partition. For reading files it uses a microfsd. And all settings of the bootmanager/bootloader can be stored in ordinary text config files.

 And finally, our idea of hybrid loader/bootmanager allows us to load different <acronym title="Operating System">OS</acronym>/2 kernel versions from the same partition. And not only kernel, any system component version can be selected from the same place – they can be selected from within the bootmanager menu.

 For more details, read on. 2) The loader present a menu to the user. Each menu item corresponds to the boot script. The script contains commands to ask additional info from user (i.e., this command shows a menu, user changes parameters, and parameters are returned to the loader. Then parameters constitute the loader "environment". The environment strings can substitute variables in command lines and config files), to change current partition, to define variables etc. Also the boot script contains definition of files, loaded by the bootloader. The loader distinguishes between executable files (the loader performs executable format parsing), files that are only loaded by the loader, but its format is not parsed, and config files. The files marked as configs are preprocessed by the preprocessor.

 So, there are following config files: i) The loader menu definition file, it contains a definition of menu items. Each menu item has a loader script associated with it. This config file is similar to the menu.lst file in GRUB. ii) the boot scripts. Each script is referenced or included by menu definition config. Each script is similar to boot.cfg file in <acronym title="Operating System">OS</acronym>/2 PPC. iii) config.sys file. It is specific for <acronym title="Operating System">OS</acronym>/2 personality. These configs are read through microfsd calls and can be preprocessed. There may be additional config files for individual servers. They also use the loader config preprocessing facilities, so parameters defined in the loader script or the ones asked from user may substitute variables in these config files. So we can flexibly set parameters of each system component from the bootmanager menu.

 Also, for better flexibility, we can make a small config for the blackbox. When the blackbox is started by the bootsector, it may read its config file, from which it knows, what files it must load as the minifsd (it is optional, we may not use minifsd, so it may be not necessary to load it) and freeldr main module. (See the next paragraph: The idea of modules).

 3) The idea of modules.

 At present time, the loader is a COM file, so its size is limited to 64 Kb. To include more functionality, it may be necessary to make it a multi-segment program, so, a better EXE format must be used. Because it is 16-bit real mode program, the DOS EXE format and, probably, <acronym title="Operating System">OS</acronym>/2 NE format are suitable for that. The DOS EXE format seems to be the most simple, so it is simpler to implement FreeLdr as a DOS EXE file. To keep the loader modular (to have possibility to load only needed parts of it, and the possibility to load/unload parts of it at any time), I suggest to implement it as a set of modules. A module must work in real mode and can be implemented as a DOS EXE file. (We can't use DLL's in real mode, so we must design a simple mechanism based on DOS EXE files). I propose a module to be a DOS EXE file with additional header. The header helps to locate functions inside the EXE file. The header begins with a pointer to the ASCIIZ string which contains a module name. After this pointer follows a size of the header, then a size of the DOS executable after the header and then follows a table of structures, which can be described as:

 <pre class="code">struct {

```
char *FuncName;
unsigned long EntryPoint;
```

} *pFuncTable;</pre>

i.e., each structure defines a function in EXE file, it links a function name with its offset in the EXE file. This array of structures is followed by a string table, which contains all the function names and a module name. Each FuncName pointer points to the sting in this string table. The header helps to locate each function in the EXE file. The function table can be generated from linker map file.

 The main loader module is loaded by microfsd. It has a DOS EXE format, so, to execute it, we must load it by some executable format loader. The DOS EXE loader can be implemented as a DOS COM file. I suggest to concatenate the DOS EXE loader with the FreeLdr main module (the main module is glued to the tail of the EXE loader, this idea is borrowed from the <acronym title="Microsoft">MS</acronym> NTLDR: the NTLDR consists of the startup COM file glued with the PE format executable). The FreeLdr startup receives info from the blackbox, loads and relocates the main module from its tail, executes and passes it an info received from the blackbox.

 The main module contains a mechanism to load other modules from files on disk. It loads a module as a DOS EXE file, and links its header to the headers list. When performing relocations to the module, the DOS EXE format loader corrects the addresses in headers, which are linked to the list. From this list, the main module can locate any function from any module. For that purpose, the main module supplies a function to be called from other modules. This function takes a module name and a name of a function in it as parameters, and returns an entry point to this function. This way, any module can find an entry point to any function with given name in any other module with given name. So, we have a kind of name service, which can convert a function name to its address. Any module can call any function in another module.

 The microfsd's, loaders for different file formats (DOS EXE, NE, LX, ELF), config file preprocessor etc. can be implemented as separate modules.

 4) We can implement additional executable formats loaders for formats, other than ELF. (For example, LX, NE, PE(?)...). They can be implemented as separate modules

 5) To be possible to read files from other partition than a boot one, it is possible to implement a blackbox switching. For that, the loader can have a command, executed from the boot script, to change the current drive, like “root” command in GRUB. For this, the loader loads a new microfsd for the changed partition filesystem. It updates the FileTable structure by pointers to functions in the new microfsd. It also loads BPB from the new partition bootsector (and patches the HiddenSectors value, if needed). So, this feature can give the loader possibility to read files from several partitions, switching them.

 6) I propose to make the loader capable of loading standard multiboot kernels, L4 kernel (as a kind of a multiboot kernel), and custom kernels, like <acronym title=“Operating System”>OS</acronym>/2 kernel. Before, in this text the idea of modules was described. The idea is to implement a loader as a set of loadable modules. Custom <acronym title=“Operating System”>OS</acronym> kernels, not compatible with multiboot specification, can be supported by custom loader module. The multiboot support can also be implemented as a separate module. The module is loaded by the Freeldr main module. By writing support module for custom kernel type, the developers from outside can extend our loader to load their kernels. There may be, our loader will suit not only unix or <acronym title=“Operating System”>OS</acronym>/2 or L4 kernels, but windoze and ReactOS kernels too. We can't expect uncle Billy to make support for loading windoze kernel from our loader, but it is possible that ReactOS guys can make their kernel loadable from it. <img src=“/web/20100304104831im_/http://www.osfree.org/wiki/lib/images/smileys/icon_wink.gif” class=“middle” alt=“ ”> The part of the loader loaded by the blackbox is called the general part of the loader. The general part contains only support functions for loading modules, locating functions in them, the DOS EXE format loader, and some more functions. It passes control to the custom part along with interfaces to module loader, microfsd's, and the info obtained from the blackbox of a bootable partition. The custom loader part implements support for loading specific kind of kernel.

 a) For loading multiboot kernels, the multiboot specific part of a loader can be implemented as a separate module. Through it, we can load L4 kernel, as well as most unix kernels.

 b) For loading ordinary <acronym title=“Operating System”>OS</acronym>/2 kernels, there must be a specific custom part. This custom part takes from general part the info, obtained from the blackbox. It loads os2ldr file from disk and passes this info to os2ldr. Then the boot process continues as usual. In future, the custom part can be extended, so it will fully replace os2ldr functionality as David Zimmerli wanted – his idea was exactly to replace os2ldr functionality. c) For loading unsupported kernels, the loader can only load corresponding <acronym title=“Operating System”>OS</acronym> bootsector, and give control to it. It may be implemented like GRUB “chainloader” command.

<h3>Suggested boot sequence</h3>

1) Do we need a MiniFSD? <pre class=“code”> If we look at the OS/2 PPC and L4 boot sequence, we may conclude that OS/2 PPC bootloader and GRUB do similar things. They load a kernel and a set of files into memory and start the kernel. Then the bootstrap task, in 1st case, and a root task in 2nd case, will start other tasks; the FreeLdr also must do equivalent tasks. As at this moment the filesystem is not yet initialized, the bootstrap or root task can only access files that were already

loaded by the bootloader.</pre>

To use a filesystem, we must first load the disk driver (ibm1s506.add or ibm1flpy.add), dasd manager, volume manager and filesystem driver (or their equivalents in our microkernel system). Let assume that we can use a minifsd as a filesystem driver. With L4, we can't use 16-bit programs as easy as in present <acronym title="Operating System">OS</acronym>/2. So, our minifsd must be 32-bit. And 62 Kb limit for its size is not applicable here, as we use 32-bit programs.

 As written in ifs.inf file from IBM, a minifsd has two modes of operation – at phase 1 and at phase 2 of boot process. Before phase 1, when minifsd initializes, it can not call any dynalink calls (in MFS_INIT initialization routine). Ordinary IFS in its FS_INIT routine, can call external dlls. The only external functions a minifsd can call are MFSH_* helpers called from the kernel. The full-featured IFS's can call a much wider number of external calls. They are FSH_* IFS helpers. (but, as I understood, the IFS can't call other external functions, besides FSH_* calls in routines other than its FS_INIT routine). For the IFS to be possible to call external DLLs at IFS init time, the dynamic loading support must be working and operational, and the DLLs itself must be available. They can be only loaded by minifsd (because IFS is not initialized yet) or they may be passed by the bootloader.

 The reason why minifsd is used by the <acronym title="Operating System">OS</acronym>/2 (intel) boot process is to have a limited filesystem access after the kernel switched into protected mode. Before the <acronym title="Operating System">OS</acronym>/2 disk subsystem drivers are loaded, the disk read is performed by temporarily switching into real mode and calling int 13h disk read functions. After the disk subsystem is loaded, the disk read is performed through <acronym title="Operating System">OS</acronym>/2 disk driver.

 In L4 or in <acronym title="Operating System">OS</acronym>/2 PPC, before disk drivers are loaded, we can't switch to real mode to call int 13 routines. Consequently, the disk drivers must be read through the microfsd by the bootloader, and then they must be passed to roottask through memory. So, the disk drivers can be started from their memory images by the roottask. But then why we must use a minifsd? We can load the full-featured boot IFS immediately, and before that, we can start ELF format dynamic loader support servers and load all the DLLs boot IFS needs. All these files can be passed by the bootloader along with disk subsystem drivers.

 So, it has not much sense to use minifsd in L4 boot sequence, only microfsd is needed. And if we look at <acronym title="Operating System">OS</acronym>/2 PPC, then we will see that it has not a minifsd and basedev's loading phase. Instead, all required services are passed to the bootstrap task by the bootloader. In our case, FreeLdr and kickstart play the role of <acronym title="Operating System">OS</acronym>/2 PPC bootloader, and roottask is analogous to the bootstrap task. So, we must to make similar design solutions.

 2) In case of loading L4, the loader loads kickstart L4 bootstrapper. Kickstart is given a set of modules by the bootloader through the multiboot structure. The kickstart then passes this info in bootinfo structure, pointed by the field in KIP. When L4 is started, it starts sigma0 and roottask. The roottask can obtain info from bootinfo structure, which can be reached from the KIP. The KIP address can be obtained through the KernelInterface() L4 system call. So, the roottask can obtain info about modules, passed by kickstart. Then the roottask can find the needed modules in the bootinfo structure. The purpose of each module can be defined through strings associated with modules, each module can be marked by special tag, which defines its purpose (e.g., the module contains a library, a config, a root name server, an executable files loader server. etc.). The tag can be contained in string along with command line for this module. This way, an info about modules is passed from loader

through kickstart to roottask, and roottask can find needed servers and load them in proper order. By that, the roottask can implement a funcionality of <acronym title="Operating System">OS</acronym>/2 PPC bootstrap task. The roottask first starts the personality neutral (PN) services, then brings up the <acronym title="Operating System">OS</acronym> personalities (<acronym title="Operating System">OS</acronym>/2, L4Linux, etc.). </p>

Kurzüberblick über die osFree Boot-Sequenz (Normativ)

Wenn der Computer eingeschaltet oder neu gestartet wird, dann wird als Erstes Programm das BIOS ausgeführt. Es gibt verschiedene Arten von BIOSen die unterschiedlich ablaufen. Wir müssen an dieser Stelle aber nur eine Sache wissen: Das BIOS (welcher Art auch immer) lädt einen Bootsektor (unter diesem Begriff verstehen wir nicht nur den eigentlichen Bootsektor auf der Festplatte oder in einem Festplattenabbild sondern jeglichen ersten Code der vom BIOS geladen und ausgeführt wird) und übergibt die Kontrolle über den Computer an den dort positionierten Code. Und hier fängt unsere Boot-Sequenz an. Alles, was vor dieser Stelle passiert packen wir in eine BlackBox. Wir wissen nicht, wie es funktioniert. Wir wissen nur, dass unser Code die Kontrolle erhält. Unser Bootsektor ist Speicherabhängiger 16-Bit-Code. Der Bootsektor lädt die erste Stufe des Loaders, MicroFSD (Micro File System Driver), MiniFSD (Mini File System Driver) und den Kernel-Loader. Der Bootsektor füllt Daten-Strukturen mit Informationen über den derzeitigen Zustand des Arbeitsspeichers und die Einsprungpunkte des MicroFSD und übergibt sie an den Kernel-Loader. Der Kernel-Loader selbst besteht aus einer Mischung aus 16-Bit und 32-Bit-Code. Er lädt einen Multiboot-Kompatiblen Kernel (osFree Kernel), positioniert ihn und MiniFSD im Arbeitsspeicher, linkt die Einsprungpunkte, schaltet die CPU in den Protected Mode um und übergibt die Kontrolle an den Kernel. Der Kernel und MiniFSD sind schon reiner 32-Bit-Code.

MicroFSD/KernelLoader interface (Normative)

The MicroFSD/KernelLoader interface is the same as for OS/2. After MicroFSD has loaded all required code (MiniFSD image, OS3LDR image), it passes control to KernelLoader (OS3LDR). The CPU must be in real mode and the CPU registers must be filled like in the following table.

When initially transferring control to OS3LDR from a "black box", the following interface is defined:

<pre class="code">Register Contains Description DH Boot mode flags bit 0 (NOVOLIO) on indicates that the mini- FSD does not use MFSH_DOVOLIO. bit 1 (RIPL) on indicates that boot volume is not local (RIPL boot) bit 2 (MINIFSD) on indicates that a mini-FSD ispresent. bit 3 (RESERVED) bit 4 (MICROFSD) on indicates that a micro-FSD is present. bits 5-7 are reserved and MUST be zero. DL Boot disk drive number This parameter is ignored if either the NOVOLIO or MINIFSD bits are zero. DS:SI pointer to the BOOT Media's BPB This parameter is ignored if either the NOVOLIO or MINIFSD bits are zero. ES:DI pointer to a filetable structure struct FileTable { /* # of entries in this table */ unsigned short ft_cfiles; /* paragraph # where OS2LDR is loaded */ unsigned short ft_ldrseg; /* length of OS2LDR in bytes */ unsigned long ft_ldrlen; /* paragraph # where microFSD is loaded */ unsigned short ft_museg; /* length of microFSD in bytes */ unsigned long ft_mulen; /* paragraph # where miniFSD is loaded */ unsigned short ft_mfsseg; /* length of miniFSD in bytes */ unsigned long ft_mfslen; /* paragraph # where RIPL data is loaded */ unsigned short ft_ripseg; /* length of RIPL data in bytes. */ unsigned long ft_riplen; </pre> <pre class="code"> /* The next four elements are pointers to

```
  microFSD entry points     */
```

unsigned short(far *ft_muOpen)(char far *pName,

```
                unsigned long far *pulFileSize);
```

unsigned long (far *ft_muRead)(long loffseek,

```
           char far *pBuf, unsigned long cbBuf);
```

unsigned long (far *ft_muClose)(void); unsigned long (far *ft_muTerminate)(void);</pre>

 }

The microFSD entry points interface is defined as follows:

  <li class=“level1”> mu_Open is passed a far pointer to the name of the file to be opened and a far pointer to a ULONG to return the file size. The re-turned value (in AX) indicates success(0) or failure (non-0). <li class=“level1”> mu_Read is passed a seek offset, a far pointer to a data buffer, and the size of the data buffer. The returned value(in DX:AX) indicates the number of bytes actually read. <li class=“level1”> mu_Close has no parameters and expects no return value. It is a signal to the micro-FSD that the loader is done reading the current file. <li class=“level1”> mu_Terminate has no parameters and expects no return value. It is a signal to the micro-FSD that the loader has finished reading the boot drive.

The loader will call the micro-FSD in a Open-Read-Read-….-Read-Close sequence for each file read in from the boot drive. After all files are loaded, mu_Terminate must be called.

<h2>KernelLoader/Kernel interface (Normative)</h2>

 The KernelLoader/Kernel interface is not <acronym title=“Operating System”>OS</acronym>/2 compatible but multiboot compatible. This means you can load different kernels, for example a Linux kernel.

 There are three main aspects of a Kernel loader/Kernel image interface:

  <li class=“level1”> The format of an Kernel image as seen by a Kernel loader. <li class=“level1”> The state of a machine when a Kernel loader starts an operating system. <li class=“level1”> The format of information passed by a Kernel loader to an operating system.

<h3>Kernel image format</h3>

A Kernel image may be an ordinary 32-bit executable file in the standard format for that particular operating system, except that it may be linked at a non-default load address to avoid loading on top of the PC's I/O region or other reserved areas, and of course it should not use shared libraries or other fancy features.

 A Kernel image must contain an additional header, called Multiboot header, besides the headers of the format used by the Kernel image. The Multiboot header must be contained completely within the first 8192 bytes of the Kernel image, and must be longword (32-bit) aligned. In general, it should come as early as possible and may be embedded in the beginning of the text segment after the real executable header.

<h3>The layout of the Multiboot header</h3>

The layout of the Multiboot header must be as follows:

 <pre class="code">Offset Type Field Name Note 0 u32 Magic required 4 u32 Flags required 8 u32 checksum required 12 u32 header_addr if flags[16] is set 16 u32 load_addr if flags[16] is set 20 u32 load_end_addr if flags[16] is set 24 u32 bss_end_addr if flags[16] is set 28 u32 entry_addr if flags[16] is set 32 u32 mode_type must be ignored 36 u32 width must be ignored 40 u32 height must be ignored 44 u32 depth must be ignored </pre>

The fields magic, flags and checksum are defined in Header magic fields, the fields header_addr, load_addr, load_end_addr, bss_end_addr and entry_addr are defined in Header address fields, and the fields mode_type, width, height and depth are defind in Header graphics fields.

 The magic fields of Multiboot header magic The field magic is the magic number identifying the header, which must be the hexadecimal value 0x1BADB002.

 flags The field flags specifies features that the Kernel image requests or requires of an Kernel loader. Bits 0-15 indicate requirements; if the kernel loader sees any of these bits set but doesn't understand the flag or can't fulfill the requirements it indicates for some reason, it must notify the user and fail to load the Kernel image. Bits 16-31 indicate optional features; if any bits in this range are set but the kernel loader doesn't understand them, it may simply ignore them and proceed as usual. Naturally, all as-yet-undefined bits in the flags word must be set to zero in Kernel images. This way, the flags fields serves for version control as well as simple feature selection.

 If bit 0 in the flags word is set, then all boot modules loaded along with the operating system must be aligned on page (4KB) boundaries. Some operating systems expect to be able to map the pages containing boot modules directly into a paged address space during startup, and thus need the boot modules to be page-aligned. If bit 1 in the flags word is set, then information on available memory via at least the mem_* fields of the Multiboot information structure (see Boot information format) must be included. If the kernel loader is capable of passing a memory map (the mmap_* fields) and one exists, then it may be included as well. The bit 2 in the flags word must be ignored by

Kernel loader. If bit 16 in the flags word is set, then the fields at offsets 8-24 in the Multiboot header are valid, and the kernel loader should use them instead of the fields in the actual executable header to calculate where to load the Kernel image. This information does not need to be provided if the kernel image is in ELF format, but it must be provided if the image is in a.out format or in some other format. Compliant kernel loaders must be able to load images that either are in ELF format or contain the load address information embedded in the Multiboot header; they may also directly support other executable formats, such as particular a.out variants, but are not required to.

 checksum The field checksum is a 32-bit unsigned value which, when added to the other magic fields (i.e. magic and flags), must have a 32-bit unsigned sum of zero.

 The address fields of Multiboot header All of the address fields enabled by flag bit 16 are physical addresses. The meaning of each is as follows:

 header_addr Contains the address corresponding to the beginning of the Multiboot header – the physical memory location at which the magic value is supposed to be loaded. This field serves to synchronize the mapping between Kernel image offsets and physical memory addresses. load_addr Contains the physical address of the beginning of the text segment. The offset in the Kernel image file at which to start loading is defined by the offset at which the header was found, minus (header_addr - load_addr). load_addr must be less than or equal to header_addr. load_end_addr Contains the physical address of the end of the data segment. (load_end_addr - load_addr) specifies how much data to load. This implies that the text and data segments must be consecutive in the Kernel image; this is true for existing a.out executable formats. If this field is zero, the kernel loader assumes that the text and data segments occupy the whole Kernel image file. bss_end_addr Contains the physical address of the end of the bss segment. The kernel loader initializes this area to zero, and reserves the memory it occupies to avoid placing boot modules and other data relevant to the operating system in that area. If this field is zero, the kernel loader assumes that no bss segment is present. entry_addr The physical address to which the kernel loader should jump in order to start running the operating system. Machine state When the kernel loader invokes the 32-bit operating system, the machine must have the following state:

EAX Must contain the magic value 0x2BADB002; the presence of this value indicates to the operating system that it was loaded by a Multiboot-compliant kernel loader (e.g. as opposed to another type of kernel loader that the operating system can also be loaded from). EBX Must contain the 32-bit physical address of the Multiboot information structure provided by the kernel loader (see Boot information format). CS Must be a 32-bit read/execute code segment with an offset of 0 and a limit of 0xFFFFFFFF. The exact value is undefined. DS/ES/FS/GS/SS Must be 32-bit read/write data segments with an offset of 0 and a limit of 0xFFFFFFFF. The exact values are all undefined. A20 gate Must be enabled. CR0 Bit 31 (PG) must be cleared. Bit 0 (PE) must be set. Other bits are all undefined. EFLAGS Bit 17 (VM) must be cleared. Bit 9 (IF) must be cleared. Other bits are all undefined.

All other processor registers and flag bits are undefined. This includes, in particular:

 ESP The Kernel image must create its own stack as soon as it needs one. GDTR Even though the segment registers are set up as described above, the GDTR may be invalid, so the Kernel image must not load any segment registers (even just reloading the same values!) until it sets up its own GDT. IDTR The Kernel image must leave interrupts disabled until it sets up its own IDT.

However, besides this the machine state should be left by the kernel loader in normal working order, i.e. as initialized by the BIOS (or DOS, if that's what the kernel loader runs from). In other words, the operating system should be able to make BIOS calls and such after being loaded, as long as it does not overwrite the BIOS data structures before doing so. Also, the kernel loader must leave the PIC programmed with the normal BIOS/DOS values, even if it changed them during the switch to 32-bit mode.

 Boot information format Upon entry to the operating system, the EBX register contains the physical address of a Multiboot information data structure, through which the kernel loader communicates vital information to the operating system. The operating system can use or ignore any parts of the structure as it chooses; all information passed by the kernel loader is advisory only.

 The Multiboot information structure and its related substructures may be placed anywhere in memory by the kernel loader (with the exception of the memory reserved for the kernel and boot modules, of course). It is the operating system's responsibility to avoid overwriting this memory until it is done using it.

 The format of the Multiboot information structure (as defined so far) follows:

 <pre class="code">0 Flags (required) 4 mem_lower (present if flags[0] is set) 8 mem_upper (present if flags[0] is set) 12 boot_device (present if flags[1] is set) 16 Cmdline (present if flags[2] is set) 20 mods_count (present if flags[3] is set) 24 mods_addr (present if flags[3] is set) 28 - 40 syms (present if flags[4] or flags[5] is set) 44 mmap_length (present if flags[6] is set) 48 mmap_addr (present if flags[6] is set) 52 drives_length (present if flags[7] is set) 56 drives_addr (present if flags[7] is set) 60 config_table (present if flags[8] is set) 64 boot_loader_name (present if flags[9] is set) 68 apm_table (present if flags[10] is set) 72 vbe_control_info (must be filled by) 76 vbe_mode_info 80 vbe_mode 82 vbe_interface_seg 84 vbe_interface_off 86 vbe_interface_len </pre>

 The first longword indicates the presence and validity of other fields in the Multiboot information structure. All as-yet-undefined bits must be set to zero by the kernel loader. Any set bits which the operating system does not understand should be ignored. Thus, the flags field also functions as a version indicator, allowing the Multiboot information structure to be expanded in the future without breaking anything.

 If bit 0 in the flags word is set, then the mem_* fields are valid. mem_lower and mem_upper indicate the amount of lower and upper memory, respectively, in kilobytes. Lower memory starts at address 0, and upper memory starts at address 1 megabyte. The maximum possible value for lower memory is 640 kilobytes. The value returned for upper memory is maximally the address of the first upper memory hole minus 1 megabyte. It is not guaranteed to be this value.

 If bit 1 in the flags word is set, then the boot_device field is valid, and indicates which BIOS disk device the kernel loader loaded the Kernel image from. If the Kernel image was not loaded from a BIOS disk, then this field must not be present (bit 3 must be clear). The operating system may use this field as a hint for determining its own root device, but is not required to. The boot_device field is laid out in four one-byte subfields as follows:

 <pre class="code">Drive part1 part2 part3

```
</pre>
```

 The first byte contains the BIOS drive number as understood by the BIOS INT 0×13 low-level disk interface: e.g. 0×00 for the first floppy disk or 0×80 for the first hard disk.

 The three remaining bytes specify the boot partition. part1 specifies the top-level partition number, part2 specifies a sub-partition in the top-level partition, etc. Partition numbers always start from zero. Unused partition bytes must be set to 0xFF. For example, if the disk is partitioned using a simple one-level DOS partitioning scheme, then part1 contains the DOS partition number, and part2 and part3 are both 0xFF. As another example, if a disk is partitioned first into DOS partitions, and then one of those DOS partitions is subdivided into several BSD partitions using BSD's disklabel strategy, then part1 contains the DOS partition number, part2 contains the BSD sub-partition within that DOS partition, and part3 is 0xFF.

 DOS extended partitions are indicated as partition numbers starting from 4 and increasing, rather than as nested sub-partitions, even though the underlying disk layout of extended partitions is hierarchical in nature. For example, if the kernel loader boots from the second extended partition on a disk partitioned in conventional DOS style, then part1 will be 5, and part2 and part3 will both be 0xFF.

 If bit 2 of the flags longword is set, the cmdline field is valid, and contains the physical address of the command line to be passed to the kernel. The command line is a normal C-style zero-terminated string.

 If bit 3 of the flags is set, then the mods fields indicate to the kernel what boot modules were loaded along with the kernel image, and where they can be found. mods_count contains the number of modules loaded; mods_addr contains the physical address of the first module structure. mods_count may be zero, indicating no boot modules were loaded, even if bit 1 of flags is set. Each module structure is formatted as follows:

 <pre class=“code”>0 mod_start 4 mod_end 8 String 12 reserved(0)</pre>

 The first two fields contain the start and end addresses of the boot module itself. The string field provides an arbitrary string to be associated with that particular boot module; it is a zero-terminated <acronym title=“American Standard Code for Information Interchange”>ASCII</acronym> string, just like the kernel command line. The string field may be 0 if there is no string associated with the module. Typically the string might be a command line (e.g. if the operating system treats boot modules as executable programs), or a pathname (e.g. if the operating system treats boot modules as files in a file system), but its exact use is specific to the operating system. The reserved field must be set to 0 by the kernel loader and ignored by the operating system.

 Caution: Bits 4 & 5 are mutually exclusive.

 If bit 4 in the flags word is set, then the following fields in the Multiboot information structure starting at byte 28 are valid:

 <pre class=“code”>28 Tabsize 32 Strsize 36 Addr 40 reserved (0)</pre>

These indicate where the symbol table from an a.out kernel image can be found. addr is the physical address of the size (4-byte unsigned long) of an array of a.out format nlist structures, followed immediately by the array itself, then the size (4-byte unsigned long) of a set of zero-terminated <acronym title=“American Standard Code for Information Interchange”>ASCII</acronym> strings (plus sizeof(unsigned long) in this case), and finally the set of strings itself. tabsize is equal to its size

parameter (found at the beginning of the symbol section), and strsize is equal to its size parameter (found at the beginning of the string section) of the following string table to which the symbol table refers. Note that tabsize may be 0, indicating no symbols, even if bit 4 in the flags word is set.

 If bit 5 in the flags word is set, then the following fields in the Multiboot information structure starting at byte 28 are valid:

 <pre class="code">28 Num 32 Size 36 Addr 40 Shndx</pre>

These indicate where the section header table from an ELF kernel is, the size of each entry, number of entries, and the string table used as the index of names. They correspond to the shdr_* entries (shdr_num, etc.) in the Executable and Linkable Format (ELF) specification in the program header. All sections are loaded, and the physical address fields of the ELF section header then refer to where the sections are in memory (refer to the i386 ELF documentation for details as to how to read the section header(s)). Note that shdr_num may be 0, indicating no symbols, even if bit 5 in the flags word is set.

If bit 6 in the flags word is set, then the mmap_* fields are valid, and indicate the address and length of a buffer containing a memory map of the machine provided by the BIOS. mmap_addr is the address, and mmap_length is the total size of the buffer. The buffer consists of one or more of the following size/structure pairs (size is really used for skipping to the next pair):

  <li class="level1"> 4 Size

```
0   base_addr_low
```

 <pre class="code">4 base_addr_high 8 length_low 12 length_high 16 Type</pre>

where size is the size of the associated structure in bytes, which can be greater than the minimum of 20 bytes. base_addr_low is the lower 32 bits of the starting address, and base_addr_high is the upper 32 bits, for a total of a 64-bit starting address. length_low is the lower 32 bits of the size of the memory region in bytes, and length_high is the upper 32 bits, for a total of a 64-bit length. type is the variety of address range represented, where a value of 1 indicates available RAM, and all other values currently indicated a reserved area.

 The map provided is guaranteed to list all standard RAM that should be available for normal use.

 If bit 7 in the flags is set, then the drives_* fields are valid, and indicate the address of the physical address of the first drive structure and the size of drive structures. drives_addr is the address, and drives_length is the total size of drive structures. Note that drives_length may be zero. Each drive structure is formated as follows:

 <pre class="code">0 Size 4 drive_number 5 drive_mode 6 drive_cylinders 8 drive_heads 9 drive_sectors 10-xx drive_ports</pre>

The size field specifies the size of this structure. The size varies, depending on the number of ports. Note that the size may not be equal to (10 + 2 * the number of ports), because of an alignment.

 The drive_number field contains the BIOS drive number. The drive_mode field represents the access mode used by the kernel loader. Currently, the following modes are defined:

 <pre class="code">0 CHS mode (traditional cylinder/head/sector addressing mode) 1 LBA mode (Logical Block Addressing mode)</pre>

The three fields, drive_cylinders, drive_heads and drive_sectors, indicate the geometry of the drive detected by the BIOS. drive_cylinders contains the number of the cylinders. drive_heads contains the number of the heads. drive_sectors contains the number of the sectors per track.

 The drive_ports field contains the array of the I/O ports used for the drive in the BIOS code. The array consists of zero or more unsigned two-bytes integers, and is terminated with zero. Note that the array may contain any number of I/O ports that are not related to the drive actually (such as DMA controller's ports).

 If bit 8 in the flags is set, then the config_table field is valid and indicates the address of the ROM configuration table returned by the GET CONFIGURATION BIOS call. If the BIOS call fails, then the size of the table must be zero.

 If bit 9 in the flags is set, the boot_loader_name field is valid, and contains the physical address of the name of a kernel loader booting the kernel. The name is a normal C-style zero-terminated string.

 If bit 10 in the flags is set, the apm_table field is valid, and contains the physical address of an APM table defined as below:

 <pre class="code">0 version 2 cseg 4 Offset 8 cseg_16 10 Dseg 12 Flags 14 cseg_len 16 cseg_16_len 18 dseg_len </pre>

 The fields version, cseg, offset, cseg_16, dseg, flags, cseg_len, cseg_16_len, dseg_len indicate the version number, the protected mode 32-bit code segment, the offset of the entry point, the protected mode 16-bit code segment, the protected mode 16-bit data segment, the flags, the length of the protected mode 32-bit code segment, the length of the protected mode 16-bit code segment, and the length of the protected mode 16-bit data segment, respectively. Only the field offset is 4 bytes, and the others are 2 bytes. See Advanced Power Management (APM) BIOS Interface Specification, for more information.

 The bit 11 in the flags must be zero.

<h2>Für Entwickler von installierbaren Dateisystemen</h2>

Installierbare Dateisystem-Treiber (engl.: installable file systems, IFS) werden in einem IFS-Dokument

beschrieben (Noch nicht veröffentlicht).

<h2>Kernel-Loader-internes</h2> <div class="“level2”>

 Der Kernel-Loader ist purer 16-bit/32-bit Binärcode (wie <acronym title="“Microsoft”>MS</acronym>/PC-DOS COM-Dateien, aber nicht von 100h sondern von 0h gestartet).

  <li class="“level1”><div class="“li”> Als aller erstes speichert der Kernel-Loader alle Informationen aus den Prozessor-Registern in interne Datenstrukturen <li class="“level1”><div class="“li”> Abhängig von diesen Informationen speichert er Informationen über die Speicherbelegung <li class="“level1”><div class="“li”> Anschließend wird Information auf den Bildschirm ausgegeben <li class="“level1”><div class="“li”> Der Kernel wird geladen <li class="“level1”><div class="“li”> Der Computer wird in den “protected-mode” umgeschaltet <li class="“level1”><div class="“li”> Der Multiboot-Kompatible Kernel wird ausgeführt

Mehr gibts hier nicht zu sagen! War es nicht einfach?