FreeLdr 0.0.2 release notes.

1. An introduction about a concept of a multiboot loader.

The osFree project boot team presents the first preview of its bootloader – FreeLdr. The FreeLdr is a general purpose bootloader for different kinds of operating systems, using the multiboot protocol, the common protocol for interacting between kernel and a bootloader. This protocol defines the format of kernel image (a so-called multiboot header in first 8192 bytes of kernel image) and a format of information, passed from loader to kernel. This information is called the multiboot information structure. It holds a system info along with info about loaded files – a multiboot modules. The role of loader is to collect various information about system configuration (the amount of machine memory, available video modes, memory map, apm info, rom configuration table etc), load a number of files to memory and to present all this information to the kernel in a standardized format.

So, the concept of a multiboot loader is abstracted from the operating system. It is not OS-specific and must not know a details about OS internals. Its purpose is to provide an integration of different OSes on one machine (like a boot manager, but bootmanager just selects a partition to load and passes control to corresponding OS loader. The multiboot loader also does this thing, it is called a chainloading, when one loader starts, or chainloads another. But the multiboot loader can do more – It presents a menu to user, a user selects a menu item along with a boot script, associated with it and it can pass parameters, chosen by user, to a kernel along with multiboot information structure.). So, it not only selects an operating system to be loaded, but allows to pass additional configuration info from boot menu to every OS part, as a corresponding file command line or variables in its configuration file.

As it was mentioned already, the multiboot loader loads a number of files as is from disk to memory. These files are called a multiboot modules. Also, the loader loads a program, called a multiboot kernel. It loads and fixes up the kernel executable, then starts it, passing it a multiboot info structure. Then, multiboot kernel can start the OS using this information. The multiboot kernel is just a term for an OS initializer program – this is the program which accept info from the loader and boots up the OS. So, the multiboot kernel must not be a real OS kernel, it can be a mediator between the real kernel and multiboot loader. For example, L4 microkernel does not play the role of multiboot kernel. Instead, this role plays the bootstrap program in the case of L4/Fiasco, and kickstart program it the case of L4Ka::Pistachio. The bootstrap or kickstart initialize initial structures (a so-called Kernel Interface Page) and starts the L4 kernel. So, we can say that a multiboot kernel is often not a kernel itself, but a intermediate layer between real kernel and multiboot loader.

Note. The GRUB was developed specifically for microkernel systems – for GNU HURD to be exact. It is commonly used with L4 microkernel, with FreeDOS32 and with many amateur OS projects, like MDF, GrindarsOS and others. Also, the Xen supervisor is started via GRUB. The GRUB has support for Linux and *BSD's, but these OSes are not it's primary goals. They even don't support the multiboot protocol and are treated by GRUB as special cases. Our OS (osFree or OS/3, these terms can be used equally) is based on L4 and so, it depends on a multiboot protocol. So, we need a multiboot loader. We do not use GRUB because it lacks some features common for OS/2. osFree is an OS/2 clone, so it must inherit its ideology, including boot sequence features.

2. Our loader ideas -- why we did not took GRUB as is.

a) why not to take GRUB as is?

A concept of a multiboot loader was introduced by GNU GRUB bootloader. It is a very good and productive concept, and we decided to use it in our loader. Also, not only ideas were borrowed from GRUB, but we also decided not to reinvent the wheel, and our loader is partly based on GRUB sources.

We think that GRUB is a very good bootloader, it introduced many good ideas. But if all were good with GRUB internal structure, we just could to choose to use GRUB as is, not to develop our own loader. So, the main thing we don't like in GRUB is that it is monolithic. It supports about 10 filesystems, but to add a new filesystem, its code must to be linked with GRUB executable. In contrast, OS/2 boot process use the concept of microfsd – a filesystem support code is separated from the OS/2 loader. So, a new filesystem support can be added without the need in loader recompile. For example, JFS filesystem support was added to OS/2 with the release of WSeB (Aurora). But JFS had not an option to be a bootable FS. With the release of WSeB, IBM dropped OS/2 support and did not opened its sources. But despite that, the support of OS/2 booting off the JFS partition was added about year 2002 by Pavel Shtemenko. That were possible only because the modular approach in OS/2 FS support. The good feature of OS/2 is that many OS extensions can be added to the OS without the need of kernel or loader recompile. Thanks to IBM OS designers, we could add the ACPI support and bootable JFS support without having OS sources and without OS vendor support (thanks, IBM). (Greets to Pasha Shtemenko ;) – the author of acpi.psd and JFS boot). With GRUB, extensibility without loader recompile can't be done. So, we decided to keep a succession with OS/2 and to develop our own loader.

b) special features OS/2 depend on

Also, it is known that to boot OS/2 from logical partitions, the special support from the boot manager is needed. Except the IBM bootmanager, only Veit Kannegieser's VPart and Martin Kiewitz' AirBoot have this feature. GRUB cannot do this. It can boot OS/2 only from primary partitions. So, to boot OS/2, at least, GRUB must to be enhanced. And at most, we can use our own loader, supporting booting OS/2 from logical partitions. This support means the following. When booting OS/2, the OS/2 bootmanager loads the bootsector and not just starts it. It fixes several fields in a bootsector BPB, the main of which is the Hiddensectors field. The bootmanager adds a partition offset from the beginning of hard disk to the hiddensectors value and after that the logical partition can be treated like the primary one. Then, the physical device number and logical drive number are written in the extended BPB. Then the bootsector is passed on from the OS/2 loader to OS/2 kernel. Such a way, the OS/2 kernel knows what drive letter to assign to a boot drive etc. So, we plan to support such feature in our loader.

c) an OS-specific part of the loader: the multiboot kernel

We believe that the boot loader must be OS-independent. The OS-specific part of the loader must be implemented as a multiboot kernel. In GRUB, there was a built-in support for loading Linux and *BSD kernels, the OS kernels were specified in a boot script in place of a multiboot kernel and the loader had autodetect, which kind of kernel it loads – the Linux or *BSD or multiboot compliant kernel. In our loader, we decided to leave in the loader only support of multiboot protocol. So, we separated Linux kernel loading support and even chainloading support from the loader and implemented it as multiboot kernels. For this purpose, linux.mdl and chain.mdl kernels were created. The first one loads Linux-like kernels (not only Linux itself, but also memtest86 memory test and memdisk from syslinux

loader, which loads DOS-like OSes from diskette or hard disk image). The second one starts another OS loader by loading it and passing it the control (it is called the chainloading). Also, we plan to implement as multiboot kernels the image.mdl program, which will be like memdisk, but instead of using a Linux-like boot protocol, it will use a multiboot protocol. Then, we plan to implement the multiboot kernels bootos2.mdl and bootbsd.mdl, for loading OS/2 and *BSD respectively. The *BSD support can be ported from GRUB, like Linux support. For OS/2 support, it is planned to make the program (multiboot kernel) which takes the files os2ldr, os2ldr.msg, os2krnl, config.sys and several base device drivers and the boot drive IFS as multiboot modules from the loader (actually, only small number – about a dozen – must be loaded such way) and then bootos2.mdl emulates access to these files loaded in memory through standard microfsd and minifsd interfaces. So, the base filesystem support will be loaded, and then OS/2 will load as usual – by using IFS driver. This approach has some advantages. For example, a support of loading OS/2 from a CDROM filesystem can be added. Our loader has support for several filesystems, ported from GRUB. So, to load OS/2 from iso9660 filesystem, for example, we need to load a base disk support using GRUB routines. After that, all the needed files are in memory, so they can be retrieved from there when os2ldr or os2krnl will ask for them. The minifsd is not needed in this situation – these interfaces will be emulated, so the files will be got from memory. This approach is similar to booting eCS from memdisk or booting from a diskette image – no real isofs minifsd needed, only FAT one to read files from the disk image. In our situation, the files are not on a memdisk, but in a special filesystem-like structure in memory. This approach for loading OSes is commonly used in L4 environment, which currently lacks a filesystem support. Instead, it uses a simplified filesystem-like interface called bmodfs, which provides access to a multiboot modules in memory. A similar approach was used by OS/2 for PowerPC bootloader, (see an “OS/2 Warp Connect (PowerPC edition): first look” redbook from IBM for details) so it is not a new idea.

So, to summarize, the idea is the following: the OS loading must be separated to two stages 1) an OS-independent part (multiboot loader) and an OS-dependent part (a multiboot kernel). The multiboot loader is OS-neutral and for each OS must be a specific multiboot kernel. These are needed for integration between OS and loader. Also, the chainloading option exists, but it means no integration.

d) a pre-loader and blackboxes

The OS/2 boot process separates a filesystem support from the loader. For this, it uses a concept of a blackbox. The blackbox is a microfsd (a micro filesystem driver) – a program which provides four routines to the loader: to open, read, close the files and to terminate the microfsd. These routines provide a common interface to a filesystem for loader. The loader must not know the FS internal structure. The blackbox is loaded by the bootsector from the filesystem boot block, then it loads the OS/2 loader and a minifsd (a minifsd is similar to the microfsd, but it is for kernel, not loader and executes in a protected mode, not a real one). The blackbox passes the loader a structure called the FileTable with addresses of loaded files and microfsd entry points. After that, the loader uses the microfsd for FS access.

The idea of a blackbox is to abstract the loader from some details. In our loader, we extend this approach to different kinds of blackboxes. The blackbox idea is to separate a loader from knowing some details of handling different formats or protocols. The blackbox does this internally, only exposing a standard set of functions. In our loader, there will be the blackboxes for filesystem access, for parsing and relocating executables, for transparent decompression, for working with different kinds of terminals (like ordinary EGA/VGA console, hercules one, serial console and graphic vbe mode console with textmode emulation). For details, see ldr-design.txt file.

Each blackbox contains a format or protocol specific code. There is a module containing a generic code for loading and handling different blackboxes, or, in other words, the blackbox manager. This manager is called the pre-loader. The pre-loader loads with a boot drive microfsd and then it parses its .ini file, from which it knows, where to find the blackboxes and blackboxes settings and loader and minifsd names, addresses, where to load them and another configuration info. The blackboxes and a pre-loader are implemented mostly as a 32-bit protected mode programs, but they can contain a small 16-bit real mode part. For example, a microfsd is based on 32-bit code ported from GRUB. But to read files from disk, it uses int 13h BIOS routines, so, for disk reading it switches temporarily to a real mode and after executing int 13h, it returns back to protect mode. Also, for 32-bit microfsd routines, a 16-bit wrappers are made, which follow the standard IBM's 16-bit microfsd interface. So, the pre-loader provides two interfaces: an old 16-bit one, compatible with OS/2, and a new 32-bit one. A 32-bit functions are similar to 16-bit microfsd functions and are intended for calling from 32-bit protected mode.

e) a posibility to use OS2LDR instead of multiboot loader

A pre-loader exports not only filesystem access functions. It also provides a loader with a compact set (a small number, about a dozen) of functions for video mode support, for terminal access, for low-level disk access, for loading executable files etc. These functions provide a base for a loader implementation. The pre-loader serves as a low-level base for loader. The role of a loader is to implement a set of commands for execution in a boot scripts and a user interface with menus and a command line. So, the multiboot loader consists of a high level part (commands and user interface) snd a low level part, which provides a base for it through use of blackboxes. The pre-loader loads and passes control to a loader. It can be used for loading our 32-bit multiboot loader as well as good old 16-bit OS2LDR. In the latter case, the standard 16-bit interface between os2ldr and microfsd is used.

3. Loader and preloader configuration options.

a) loader configuration options

A loader and a pre-loader each have their configuration files. The freeldr.cfg is a multiboot loader config file. It is currently similar to GRUB's menu.lst file. It describes a set of menu items with associated boot scripts. The included freeldr.cfg file provides examples for available options. As in GRUB, the base commands are a kernel command and a module command. Also, for setting a VBE video mode, vbeset command is used. The modaddr command was ported from Adam Lackorzynski patch for GRUB, allowing to set a multiboot modules loading base. – The modules are loaded starting from the specified address. Also, varexpand, set and toggle commands were ported from the same GRUB patch. These commands provide the ability to set variables using the following syntax:

set var=value

then, these variables can be included in command lines in the form of:

${var}

then, ${var} will be expanded to its value 'value'. The possibility to expand variables is very convenient option. – The variable can be set in one place and its value automatically inserted in many places.

An option “module –type lip” adds a special module containing the LIP (Loader Interface Page) structure to the multiboot info. The LIP structure is a structure similar to the IBM's FileTable structure. It contains a pre-loader 32-bit entry points addresses. This structure is intended for use by a multiboot kernel. So, a multiboot kernel can use a pre-loader functions (a microfsd functions, terminal access functions or the ones for loading executable files). This option is for simplifying the life for multiboot kernels developers, so they could not reinvent the wheel. For example, bt_chain and bt_linux kernels can use pre-loader functions for screen output.

b) pre-loader .ini file

The pre-loader .ini file has several sections. A 'global' section is for global parameters. The 'driveletter' option specifies the drive letter passed through the BPB to the loader. For booting off the CDROM drive, it is assigned a value of 'u:'.

The 'multiboot' global option can have three values. The 'no' value chooses an option to use an old 16-bit OS2LDR interface with the loader. The 'yes' value selects a new 32-bit multiboot loader interface. The 'multiboot = ask' mode can be specified – then the pre-loader asks the user, which mode to use, multiboot = (y)es/(n)o? This mode is good for demo CD, when user cannot edit preldr.ini file and it is needed to test both options ('yes' and 'no'). Also, it can be useful as a pre-loader built-in default, which can be activated when preldr.ini is deleted and a pre-loader does not know what mode to use. Then it can ask from user.

Also, for minifsd and loader, there are corresponding sections in the .ini file. They have a 'name' and 'base' options, which are self-explaining. Their purpose is to specify the name of minifsd/loader and its load address. For loader, 'name = default' can be specified. It specifies that the loader name takes the default value (\os2ldr when 'multiboot = no' selected and \boot\freeldr\freeldr when 'multiboot = yes' selected). For minifsd, 'name = none' can be specified. It is useful when no need to load a minifsd, for example, in 'multiboot = yes' mode when minifsd is not needed.

A 'microfsd' and 'term' sections contain microfsd and terminals options, respectively. Both sections have 'list' option, which specify the list of all blackboxes of corresponding type. For microfsd, option 'ignorecase = yes/no' can be specified, which provides an ability to select an option to try opening files first as is, then if no such file, the second time a file is tried to be opened uppercase, then lowercase. It is almost case insensitive mode. If 'no' is selected, then filenames are case sensitive.

For terminals, the 'default' option specifies which terminal to initialize at pre-loader startup.

4. Missing or not working features

1. When loading os2ldr and os2krnl, there is not an isofs minifsd available, only microfsd. So, OS/2 kernel loads successfully, but when it tries to initialize a minifsd, it finds that it is given incorrect minifsd version. (Actually, we load a hpfs minifsd, but an isofs one is needed). So, the kernel panics and an IPE (internal processing error) message is displayed. It is normal, and it will be corrected when we will have a real minifsd for isofs.
2. Loading files from a partition, different from a boot one is not yet fully working. Loading from a primary FAT partition works. But loading from a JFS partition and from a FAT floppy not yet working. The code is being debugged.
3. Terminals, different from 'console' do not working. A 'serial' terminal doesn't work. Output to 'hercules' video adapter is not tested, as we don't have such monitor/adapter, and probably

don't work.

5. Legal issues and notes for those who want to test the loader with os2krnl/os2ldr

a) legal notice

This work is based on the GPL code and so, licensed under GPL (version 2 or later) as well. OS/2 kernel and loader are © IBM Corporation, all rights reserved. As it known, OS/2 binaries can't be distributed by third parties, only by IBM itself or its authorised partners, like Serenity Systems Inc. So, we were unable to include OS/2 binaries with our boot CD without IBM's permission.

But to test OS/2 loading functionality, each legal owner of OS/2 copy can add the missing files to the boot CD iso image and rebuild it. The instructions will follow in the next section.

Also, with the boot CD a binaries of several Opensource projects are included, namely FreeDOS, FreeDOS32, memdisk from SysLinux loader, memtest86, GNU GRUB binaries on the FreeDOS32 bootable diskette image, memdisk by Veit Kannegieser and probably more. These binaries are free software and can be distributed freely.

b) how to rebuild the CD

To rebuild the CD, the mkisofs program from cdrtools is included in \tools directory. And also, there is a script mkiso.cmd which calls mkisofs with proper parameters. You must copy all files from this CD to a cd\ directory on a hard disk, place mkiso.cmd and mkisofs.exe in .\cd\..\ directory (a parent dir for cd\). Also, place additional files to their intended directories and edit loader configuration files. Then, launch mkiso.cmd and wait when it builds the CD image. After that, burn the CD-R disk with the resulting image.

To test OS/2 loading, the OS/2 binaries can be added to the boot CD. For that, place the following files to the cd\ directory: os2krnl, os2boot, os2ldr, os2ldr.msg, config.sys. For the control to be passed to the os2ldr, an option multiboot = no (or better, multiboot = ask) in preeldr.ini file must be selected. Also, loader = default option in [loader] section must be selected. Also, the minifsd name must be set to 'os2boot'.

To test Veit Kannegieser's memdisk loading via chainloading, you must add pack files (memdisk.pf and ecsdisk?.pf) to the \bootimgs directory from the eCS installation CD or eCS demo CD. The eCS demo CD is freely available at http://www.ecomstation.com/democd/ for evaluation purposes.

c) loader sources

As with all opensource software, the osFree boot sequence sources are available. They are distributed with the CD in \src directory. To build the loader from sources, first make the config file with paths set. For a template, see the valerius.conf or note.conf or another .conf file in \src directory. Ensure the paths are correct.

For the boot sequence sources to build, you need the OpenWatcom Compiler. You will need a C

compiler (a 16-bit one and a 32-bit one), linker, assembler, librarian. No need in Fortran ;). Also, REXX interpreter is needed for scripts. We tested our scripts under Classic REXX and OOREXX under OS/2 and with Regina REXX under Windows and Linux. The build works under OS/2 and Windows, and mostly works under Linux. Under Linux, OpenWatcom version for Linux is needed. You can get its sources from ftp://ftp.openwatcom.org and compile it yourself. You can make it under Linux, as well as under Windows and then copy Linux binaries to Linux system. The Linux binaries can be got from OS/2 or Windows OpenWatcom distribution, but for some reason not all needed binaries are included there. For example, I didn't found wmake and wlib in OS/2 Watcom distribution.

Also, some tools are included with sources, in \src\tools directory, for example, awk interpreter etc. The root sources directory must be set (root=...) and ISO images directory must be set (imgdir=...). Then, shell (OS/2 or Windows cmd.exe, Linux /bin/sh or OS/2 4os2.exe etc.) and REXX interpreter path must be set. For OS/2 REXX the string for REXX must be empty. Also, in the example files there are also FreePascal and OS/2 toolkit paths. They are not needed for boot sequence sources.

After all, you must open a shell window and issue 'setenv.cmd <config_file.conf> ' command. This must set the needed environment variables. Then, say 'wmake all' and the build process starts.