The Multiboot Specification

Bryan Ford
Erich Stefan Boleyn
Kunihiro Ishiguro
OKUJI Yoshinori

1 Motivation

Every operating system ever created tends to have its own boot loader. Installing a new operating system on a machine generally involves installing a whole new set of boot mechanisms, each with completely different install-time and boot-time user interfaces. Getting multiple operating systems to coexist reliably on one machine through typical *chaining* mechanisms can be a nightmare. There is little or no choice of boot loaders for a particular operating system — if the one that comes with the operating system doesn't do exactly what you want, or doesn't work on your machine, you're screwed.

While we may not be able to fix this problem in existing commercial operating systems, it shouldn't be too difficult for a few people in the free operating system communities to put their heads together and solve this problem for the popular free operating systems. That's what this specification aims for. Basically, it specifies an interface between a boot loader and a operating system, such that any complying boot loader should be able to load any complying operating system. This specification does *not* specify how boot loaders should work — only how they must interface with the operating system being loaded.

2 Terminology

Throughout this document, the term *boot loader* means whatever program or set of programs loads the image of the final operating system to be run on the machine. The boot loader may itself consist of several stages, but that is an implementation detail not relevant to this specification. Only the *final* stage of the boot loader — the stage that eventually transfers control to the operating system — needs to follow the rules specified in this document in order to be *MultiBoot compliant*; earlier boot loader stages can be designed in whatever way is most convenient.

The term *OS image* is used to refer to the initial binary image that the boot loader loads into memory and transfers control to start the operating system. The OS image is typically an executable containing the operating system kernel.

The term *boot module* refers to other auxiliary files that the boot loader loads into memory along with the OS image, but does not interpret in any way other than passing their locations to the operating system when it is invoked.

3 Scope and requirements

3.1 Architecture

This specification is primarily targeted at PC, since they are the most common and have the largest variety of operating systems and boot loaders. However, to the extent that certain other architectures may need a boot specification and do not have one already, a variation of this specification, stripped of the x86-specific details, could be adopted for them as well.

3.2 Operating systems

This specification is targeted toward free 32-bit operating systems that can be fairly easily modified to support the specification without going through lots of bureaucratic rigmarole. The particular free operating systems that this specification is being primarily designed for are Linux, FreeBSD, NetBSD, Mach, and VSTa. It is hoped that other emerging free operating systems will adopt it from the start, and thus immediately be able to take advantage of existing boot loaders. It would be nice if commercial operating system vendors eventually adopted this specification as well, but that's probably a pipe dream.

3.3 Boot sources

It should be possible to write compliant boot loaders that load the OS image from a variety of sources, including floppy disk, hard disk, and across a network.

Disk-based boot loaders may use a variety of techniques to find the relevant OS image and boot module data on disk, such as by interpretation of specific file systems (e.g. the BSD/Mach boot loader), using precalculated *block lists* (e.g. LILO), loading from a special *boot partition* (e.g. OS/2), or even loading from within another operating system (e.g. the VSTa boot code, which loads from DOS). Similarly, network-based boot loaders could use a variety of network hardware and protocols.

It is hoped that boot loaders will be created that support multiple loading mechanisms, increasing their portability, robustness, and user-friendliness.

3.4 Boot-time configuration

It is often necessary for one reason or another for the user to be able to provide some configuration information to the operating system dynamically at boot time. While this specification should not dictate how this configuration information is obtained by the boot loader, it should provide a standard means for the boot loader to pass such information to the operating system.

3.5 Convenience to the operating system

OS images should be easy to generate. Ideally, an OS image should simply be an ordinary 32-bit executable file in whatever file format the operating system normally uses. It should be possible to `nm` or disassemble OS images just like normal executables. Specialized tools should not be needed to create OS images in a *special* file format. If this means shifting some work from the operating system to the boot loader, that is probably appropriate, because all the memory consumed by the boot loader will typically be made available again after the boot process is created, whereas every bit of code in the OS image typically has to remain in memory forever. The operating system should not have to worry about getting into 32-bit mode initially, because mode switching code generally needs to be in the boot loader anyway in order to load operating system data above the 1MB boundary, and forcing the operating system to do this makes creation of OS images much more difficult.

Unfortunately, there is a horrendous variety of executable file formats even among free Unix-like PC-based operating systems — generally a different format for each operating system. Most of the relevant free operating systems use some variant of a.out format, but some are moving to ELF. It is highly desirable for boot loaders not to have to be able to interpret all the different types of executable file formats in existence in order to load the OS image — otherwise the boot loader effectively becomes operating system specific again.

This specification adopts a compromise solution to this problem. Multiboot compliant boot images always contain a magic *Multiboot header* (see Section 4.1 [OS image format], page 5), which allows the boot loader to load the image without having to understand numerous a.out variants or other executable formats. This magic header does not need to be at the very beginning of the executable file, so kernel images can still conform to the local a.out format variant in addition to being Multiboot compliant.

3.6 Boot modules

Many modern operating system kernels, such as those of VSTa and Mach, do not by themselves contain enough mechanism to get the system fully operational: they require the presence of additional software modules at boot time in order to access devices, mount file systems, etc. While these additional modules could be embedded in the main OS image along with the kernel itself, and the resulting image be split apart manually by the operating system when it receives control, it is often more flexible, more space-efficient, and more convenient to the operating system and user if the boot loader can load these additional modules independently in the first place.

Thus, this specification should provide a standard method for a boot loader to indicate to the operating system what auxiliary boot modules were loaded, and where they can be found. Boot loaders don't have to support multiple boot modules, but they are strongly encouraged to, because some operating systems will be unable to boot without them.

4 Details

There are three main aspects of the boot loader/OS image interface:

1. The format of the OS image as seen by the boot loader.
2. The state of the machine when the boot loader starts the operating system.
3. The format of the information passed by the boot loader to the operating system.

4.1 OS image format

An OS image is generally just 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 can't use shared libraries or other fancy features.

Unfortunately, the exact meaning of the text, data, bss, and entry fields of a.out headers tends to vary widely between different executable flavors, and it is sometimes very difficult to distinguish one flavor from another (e.g. Linux ZMAGIC executables and Mach ZMAGIC executables). Furthermore, there is no simple, reliable way of determining at what address in memory the text segment is supposed to start. Therefore, this specification requires that an additional header, known as a *Multiboot header*, appear somewhere near the beginning of the executable file. In general it should come *as early as possible*, and is typically embedded in the beginning of the text segment after the *real* executable header. It *must* be contained completely within the first 8192 bytes of the executable file, and must be longword (32-bit) aligned. These rules allow the boot loader to find and synchronize with the text segment in the a.out file without knowing beforehand the details of the a.out variant. The layout of the header is as follows:

```
             +-------------------+
0            | magic: 0x1BADB002 |   (required)
4            | flags             |   (required)
8            | checksum          |   (required)
             +-------------------+
12           | header_addr       |   (present if flags[16] is set)
16           | load_addr         |   (present if flags[16] is set)
20           | load_end_addr     |   (present if flags[16] is set)
24           | bss_end_addr      |   (present if flags[16] is set)
28           | entry_addr        |   (present if flags[16] is set)
             +-------------------+
32           | mode_type         |   (present if flags[2] is set)
36           | width             |   (present if flags[2] is set)
40           | height            |   (present if flags[2] is set)
44           | depth             |   (present if flags[2] is set)
             +-------------------+
```

All fields are in little-endian byte order, of course. The first field is the magic number identifying the header, which must be the hex value 0x1BADB002.

The '`flags`' field specifies features that the OS image requests or requires of the boot loader. Bits 0-15 indicate requirements; if the boot 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 OS image. Bits 16-31 indicate optional features; if any bits in this range are set but the boot loader doesn't understand them, it can simply ignore them and proceed as usual. Naturally, all as-yet-undefined bits in the '`flags`' word must be set to zero in OS 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 Section 4.3 [Boot information format], page 8) must be included. If the bootloader is capable of passing a memory map (the '`mmap_*`' fields) and one exists, then it must be included as well.

If bit 2 in the '`flags`' word is set, information about the video mode table, defined later, must be available to the kernel.

Also, information about a preferred default mode can be specified in the '`mode_type`', '`width`', '`height`' and '`depth`' fields. This is only a *recommended* mode by the kernel. If the mode exists, the boot loader should set it if the user hasn't specified a mode. If not, it should fall back to a similar mode, if available.

Valid numbers for '`mode_type`' is 0 for linear graphics mode and 1 for EGA-standard text mode. Everything else is reserved for future expansion. Please note that even if you set this field to indicate that you want a graphics mode, you might get a text mode.

'`width`' and '`height`' is specified in pixels, if graphics mode, or characters in EGA text mode. '`depth`' is given in bits per pixel for graphics, or zero for EGA text modes.

Any, some or all of those three fields may be set to zero, indicating to the boot loader that no preference is given. This makes it possible for a kernel to just say it wants a 32-bit resolution, for example.

If bit 16 in the '`flags`' word is set, then the fields at offsets 8-24 in the Multiboot header are valid, and the boot loader should use them instead of the fields in the actual executable header to calculate where to load the OS image. This information does not need to be provided if the kernel image is in ELF format, but it *must* be provided if the images is in a.out format or in some other format. Compliant boot 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.

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 OS image offsets and physical memory addresses.

`load_addr`
: Contains the physical address of the beginning of the text segment. The offset in the OS 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 OS image; this is true for existing a.out executable formats.

`bss_end_addr`
: Contains the physical address of the end of the bss segment. The boot 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.

`entry_addr`
: The physical address to which the boot loader should jump in order to start running the operating system.

The checksum is a 32-bit unsigned value which, when added to the other required fields, must have a 32-bit unsigned sum of zero.

4.2 Machine state

When the boot loader invokes the 32-bit operating system, the machine must have the following state:

- `CS` must be a 32-bit read/execute code segment with an offset of 0 and a limit of 0xFFFFFFFF.
- `DS`, `ES`, `FS`, `GS`, and `SS` must be a 32-bit read/write data segment with an offset of 0 and a limit of 0xFFFFFFFF.
- The address 20 line must be usable for standard linear 32-bit addressing of memory (in standard PC hardware, it is wired to 0 at bootup, forcing addresses in the 1-2 MB range to be mapped to the 0-1 MB range, 3-4 is mapped to 2-3, etc.).
- Paging must be turned off.
- The processor interrupt flag must be turned off.
- `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 boot loader (e.g. as opposed to another type of boot 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 boot loader (see Section 4.3 [Boot information format], page 8).

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

- `ESP`: the 32-bit operating system 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 operating system must not load any segment registers (even just reloading the same values!) until it sets up its own `GDT`.
- `IDTR`: The operating system must leave interrupts disabled until it sets up its own `IDT`.

However, other machine state should be left by the boot loader in *normal working order*, i.e. as initialized by the BIOS (or DOS, if that's what the boot 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 boot loader must leave the PIC programmed with the normal BIOS/DOS values, even if it changed them during the switch to 32-bit mode.

4.3 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 boot 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 boot loader is advisory only.

The Multiboot information structure and its related substructures may be placed anywhere in memory by the boot 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:

```
             +-------------------+
     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  |    (present if flags[11] is set)
     76      | vbe_mode_info     |
     80      | vbe_mode          |
     82      | vbe_interface_seg |
     84      | vbe_interface_off |
     86      | vbe_interface_len |
             +-------------------+
```

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 boot loader. Any set bits that 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 boot loader loaded the OS image from. If the OS 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:

```
+-------+-------+-------+-------+
| drive | part1 | part2 | part3 |
+-------+-------+-------+-------+
```

The first byte contains the BIOS drive number as understood by the BIOS INT 0x13 low-level disk interface: e.g. 0x00 for the first floppy disk or 0x80 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 boot 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:

```
        +-------------------+
0       | mod_start         |
4       | mod_end           |
        +-------------------+
8       | string            |
        +-------------------+
12      | reserved (0)      |
        +-------------------+
```

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 ASCII 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 boot 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:

```
        +-------------------+
28      | tabsize           |
32      | strsize           |
36      | addr              |
40      | reserved (0)      |
        +-------------------+
```

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 ASCII 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:

```
        +-------------------+
28      | num               |
32      | size              |
36      | addr              |
40      | shndx             |
        +-------------------+
```

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):

```
        +-------------------+
-4      | size              |
        +-------------------+
0       | base_addr_low     |
4       | base_addr_high    |
8       | length_low        |
12      | length_high       |
16      | type              |
        +-------------------+
```

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 formatted as follows:

```
             +-------------------+
     0       | size              |
             +-------------------+
     4       | drive_number      |
             +-------------------+
     5       | drive_mode        |
             +-------------------+
     6       | drive_cylinders   |
     8       | drive_heads       |
     9       | drive_sectors     |
             +-------------------+
     10 - xx | drive_ports       |
             +-------------------+
```

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 boot loader. Currently, the following modes are defined:

'0' CHS mode (traditional cylinder/head/sector addressing mode).

'1' LBA mode (Logical Block Addressing mode).

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 boot 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:

```
            +----------------------+
0           | version              |
2           | cseg                 |
4           | offset               |
8           | cseg_16              |
10          | dseg                 |
12          | flags                |
14          | cseg_len             |
16          | cseg_16_len          |
18          | dseg_len             |
            +----------------------+
```

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 (http://www.microsoft.com/hwdev/busbios/amp_12.htm), for more information.

If bit 11 in the 'flags' is set, the graphics table is available. This must only be done if the kernel has indicated in the 'Multiboot Header' that it accepts a graphics mode.

The fields 'vbe_control_info' and 'vbe_mode_info' contain the physical addresses of VBE control information returned by the VBE Function 00h and VBE mode information returned by the VBE Function 01h, respectively.

The field 'vbe_mode' indicates current video mode in the format specified in VBE 3.0.

The rest fields 'vbe_interface_seg', 'vbe_interface_off', and 'vbe_interface_len' contain the table of a protected mode interface defined in VBE 2.0+. If this information is not available, those fields contain zero. Note that VBE 3.0 defines another protected mode interface which is incompatible with the old one. If you want to use the new protected mode interface, you will have to find the table yourself.

The fields for the graphics table are designed for VBE, but Multiboot boot loaders may simulate VBE on non-VBE modes, as if they were VBE modes.

5 Examples

Caution: The following items are not part of the specification document, but are included for prospective operating system and boot loader writers.

5.1 Notes on PC

In reference to bit 0 of the '`flags`' parameter in the Multiboot information structure, if the bootloader in question uses older BIOS interfaces, or the newest ones are not available (see description about bit 6), then a maximum of either 15 or 63 megabytes of memory may be reported. It is *highly* recommended that boot loaders perform a thorough memory probe.

In reference to bit 1 of the '`flags`' parameter in the Multiboot information structure, it is recognized that determination of which BIOS drive maps to which device driver in an operating system is non-trivial, at best. Many kludges have been made to various operating systems instead of solving this problem, most of them breaking under many conditions. To encourage the use of general-purpose solutions to this problem, there are 2 BIOS device mapping techniques (see Section 5.2 [BIOS device mapping techniques], page 14).

In reference to bit 6 of the '`flags`' parameter in the Multiboot information structure, it is important to note that the data structure used there (starting with '`BaseAddrLow`') is the data returned by the INT 15h, AX=E820h — Query System Address Map call. See See section "Query System Address Map" in *The GRUB Manual*, for more information. The interface here is meant to allow a boot loader to work unmodified with any reasonable extensions of the BIOS interface, passing along any extra data to be interpreted by the operating system as desired.

5.2 BIOS device mapping techniques

Both of these techniques should be usable from any PC operating system, and neither require any special support in the drivers themselves. This section will be flushed out into detailed explanations, particularly for the I/O restriction technique.

The general rule is that the data comparison technique is the quick and dirty solution. It works most of the time, but doesn't cover all the bases, and is relatively simple.

The I/O restriction technique is much more complex, but it has potential to solve the problem under all conditions, plus allow access of the remaining BIOS devices when not all of them have operating system drivers.

5.2.1 Data comparison technique

Before activating *any* of the device drivers, gather enough data from similar sectors on each of the disks such that each one can be uniquely identified.

After activating the device drivers, compare data from the drives using the operating system drivers. This should hopefully be sufficient to provide such a mapping.

Problems:

1. The data on some BIOS devices might be identical (so the part reading the drives from the BIOS should have some mechanism to give up).

2. There might be extra drives not accessible from the BIOS which are identical to some drive used by the BIOS (so it should be capable of giving up there as well).

5.2.2 I/O restriction technique

This first step may be unnecessary, but first create copy-on-write mappings for the device drivers writing into PC RAM. Keep the original copies for the *clean* BIOS *virtual machine* to be created later.

For each device driver brought online, determine which BIOS devices become inaccessible by:

1. Create a *clean* BIOS *virtual machine*.
2. Set the I/O permission map for the I/O area claimed by the device driver to no permissions (neither read nor write).
3. Access each device.
4. Record which devices succeed, and those which try to access the *restricted* I/O areas (hopefully, this will be an *xor* situation).

For each device driver, given how many of the BIOS devices were subsumed by it (there should be no gaps in this list), it should be easy to determine which devices on the controller these are.

In general, you have at most 2 disks from each controller given BIOS numbers, but they pretty much always count from the lowest logically numbered devices on the controller.

5.3 Example OS code

In this distribution, the example Multiboot kernel '`kernel`' is included. The kernel just prints out the Multiboot information structure on the screen, so you can make use of the kernel to test a Multiboot-compliant boot loader and for reference to how to implement a Multiboot kernel. The source files can be found under the directory '`docs`' in the GRUB distribution.

The kernel '`kernel`' consists of only three files: '`boot.S`', '`kernel.c`' and '`multiboot.h`'. The assembly source '`boot.S`' is written in GAS (see section "GNU assembler" in *The GNU assembler*), and contains the Multiboot information structure to comply with the specification. When a Multiboot-compliant boot loader loads and execute it, it initialize the stack pointer and `EFLAGS`, and then call the function `cmain` defined in '`kernel.c`'. If `cmain` returns to the callee, then it shows a message to inform the user of the halt state and stops forever until you push the reset key. The file '`kernel.c`' contains the function `cmain`, which checks if the magic number passed by the boot loader is valid and so on, and some functions to print messages on the screen. The file '`multiboot.h`' defines some macros, such as the magic number for the Multiboot header, the Multiboot header structure and the Multiboot information structure.

5.3.1 multiboot.h

This is the source code in the file '`multiboot.h`':

```
/* multiboot.h - the header for Multiboot */
/* Copyright (C) 1999  Free Software Foundation, Inc.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA. */

/* Macros. */

/* The magic number for the Multiboot header. */
#define MULTIBOOT_HEADER_MAGIC          0x1BADB002

/* The flags for the Multiboot header. */
#define MULTIBOOT_HEADER_FLAGS          0x00010003

/* The magic number passed by a Multiboot-compliant boot loader. */
#define MULTIBOOT_BOOTLOADER_MAGIC      0x2BADB002

/* The size of our stack (16KB). */
#define STACK_SIZE                      0x4000

/* C symbol format. HAVE_ASM_USCORE is defined by configure. */
#ifdef HAVE_ASM_USCORE
# define EXT_C(sym)                     _ ## sym
#else
# define EXT_C(sym)                     sym
#endif

#ifndef ASM
/* Do not include here in boot.S. */

/* Types. */

/* The Multiboot header. */
typedef struct multiboot_header
{
  unsigned long magic;
  unsigned long flags;
  unsigned long checksum;
```

```
  unsigned long header_addr;
  unsigned long load_addr;
  unsigned long load_end_addr;
  unsigned long bss_end_addr;
  unsigned long entry_addr;
} multiboot_header_t;

/* The symbol table for a.out. */
typedef struct aout_symbol_table
{
  unsigned long tabsize;
  unsigned long strsize;
  unsigned long addr;
  unsigned long reserved;
} aout_symbol_table_t;

/* The section header table for ELF. */
typedef struct elf_section_header_table
{
  unsigned long num;
  unsigned long size;
  unsigned long addr;
  unsigned long shndx;
} elf_section_header_table_t;

/* The Multiboot information. */
typedef struct multiboot_info
{
  unsigned long flags;
  unsigned long mem_lower;
  unsigned long mem_upper;
  unsigned long boot_device;
  unsigned long cmdline;
  unsigned long mods_count;
  unsigned long mods_addr;
  union
  {
    aout_symbol_table_t aout_sym;
    elf_section_header_table_t elf_sec;
  } u;
  unsigned long mmap_length;
  unsigned long mmap_addr;
} multiboot_info_t;

/* The module structure. */
typedef struct module
{
  unsigned long mod_start;
  unsigned long mod_end;
```

```
  unsigned long string;
  unsigned long reserved;
} module_t;

/* The memory map. Be careful that the offset 0 is base_addr_low
  but no size. */
typedef struct memory_map
{
  unsigned long size;
  unsigned long base_addr_low;
  unsigned long base_addr_high;
  unsigned long length_low;
  unsigned long length_high;
  unsigned long type;
} memory_map_t;

#endif /* ! ASM */
```

5.3.2 boot.S

In the file 'boot.S':

```
/* boot.S - bootstrap the kernel */
/* Copyright (C) 1999  Free Software Foundation, Inc.

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  You should have received a copy of the GNU General Public License
  along with this program; if not, write to the Free Software
  Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA. */

#define ASM     1
#include <multiboot.h>

        .text

        .globl  start, _start

        /* This entry is not used actually. */
start:
_start:
        jmp     multiboot_entry
```

```
        /* Align 32 bits boundary. */
        .align  4

        /* Multiboot header. */
multiboot_header:
        /* magic */
        .long   MULTIBOOT_HEADER_MAGIC
        /* flags */
        .long   MULTIBOOT_HEADER_FLAGS
        /* checksum */
        .long   -(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS)
        /* header_addr */
        .long   multiboot_header
        /* load_addr */
        .long   _start
        /* load_end_addr */
        .long   _edata
        /* bss_end_addr */
        .long   _end
        /* entry_addr */
        .long   multiboot_entry

multiboot_entry:
        /* Initialize the stack pointer. */
        movl    $(stack + STACK_SIZE), %esp

        /* Reset EFLAGS. */
        pushl   $0
        popf

        /* Push the pointer to the Multiboot information structure. */
        pushl   %ebx
        /* Push the magic value. */
        pushl   %eax

        /* Now enter the C main function... */
        call    EXT_C(cmain)

        /* Halt. */
        pushl   $halt_message
        call    EXT_C(printf)

loop:   hlt
        jmp     loop

halt_message:
        .asciz  "Halted."
```

```
        /* Our stack area. */
        .comm   stack, STACK_SIZE
```

5.3.3 kernel.c

And, in the file 'kernel.c':

```
/* kernel.c - the C part of the kernel */
/* Copyright (C) 1999  Free Software Foundation, Inc.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA. */

#include <multiboot.h>

/* Macros. */

/* Check if the bit BIT in FLAGS is set. */
#define CHECK_FLAG(flags,bit)   ((flags) & (1 << (bit)))

/* Some screen stuff. */
/* The number of columns. */
#define COLUMNS                 80
/* The number of lines. */
#define LINES                   24
/* The attribute of an character. */
#define ATTRIBUTE               7
/* The video memory address. */
#define VIDEO                   0xB8000

/* Variables. */
/* Save the X position. */
static int xpos;
/* Save the Y position. */
static int ypos;
/* Point to the video memory. */
static volatile unsigned char *video;
```

```
/* Forward declarations. */
void cmain (unsigned long magic, unsigned long addr);
static void cls (void);
static void itoa (char *buf, int base, int d);
static void putchar (int c);
void printf (const char *format, ...);

/* Check if MAGIC is valid and print the Multiboot information structure
  pointed by ADDR. */
void
cmain (unsigned long magic, unsigned long addr)
{
  multiboot_info_t *mbi;

  /* Clear the screen. */
  cls ();

  /* Am I booted by a Multiboot-compliant boot loader? */
  if (magic != MULTIBOOT_BOOTLOADER_MAGIC)
    {
      printf ("Invalid magic number: 0x%x\n", magic);
      return;
    }

  /* Set MBI to the address of the Multiboot information structure. */
  mbi = (multiboot_info_t *) addr;

  /* Print out the flags. */
  printf ("flags = 0x%x\n", mbi->flags);

  /* Are mem_* valid? */
  if (CHECK_FLAG (mbi->flags, 0))
    printf ("mem_lower = %dKB, mem_upper = %dKB\n",
            mbi->mem_lower, mbi->mem_upper);

  /* Is boot_device valid? */
  if (CHECK_FLAG (mbi->flags, 1))
    printf ("boot_device = 0x%x\n", mbi->boot_device);

  /* Is the command line passed? */
  if (CHECK_FLAG (mbi->flags, 2))
    printf ("cmdline = %s\n", (char *) mbi->cmdline);

  /* Are mods_* valid? */
  if (CHECK_FLAG (mbi->flags, 3))
    {
      module_t *mod;
      int i;
```

```
    printf ("mods_count = %d, mods_addr = 0x%x\n",
            mbi->mods_count, mbi->mods_addr);
    for (i = 0, mod = (module_t *) mbi->mods_addr;
         i < mbi->mods_count;
         i++, mod += sizeof (module_t))
      printf (" mod_start = 0x%x, mod_end = 0x%x, string = %s\n",
              mod->mod_start, mod->mod_end, (char *) mod->string);
  }

/* Bits 4 and 5 are mutually exclusive! */
if (CHECK_FLAG (mbi->flags, 4) && CHECK_FLAG (mbi->flags, 5))
  {
    printf ("Both bits 4 and 5 are set.\n");
    return;
  }

/* Is the symbol table of a.out valid? */
if (CHECK_FLAG (mbi->flags, 4))
  {
    aout_symbol_table_t *aout_sym = &(mbi->u.aout_sym);

    printf ("aout_symbol_table: tabsize = 0x%0x, "
            "strsize = 0x%x, addr = 0x%x\n",
            aout_sym->tabsize, aout_sym->strsize, aout_sym->addr);
  }

/* Is the section header table of ELF valid? */
if (CHECK_FLAG (mbi->flags, 5))
  {
    elf_section_header_table_t *elf_sec = &(mbi->u.elf_sec);

    printf ("elf_sec: num = %d, size = 0x%x,"
            " addr = 0x%x, shndx = 0x%x\n",
            elf_sec->num, elf_sec->size,
            elf_sec->addr, elf_sec->shndx);
  }

/* Are mmap_* valid? */
if (CHECK_FLAG (mbi->flags, 6))
  {
    memory_map_t *mmap;

    printf ("mmap_addr = 0x%x, mmap_length = 0x%x\n",
            mbi->mmap_addr, mbi->mmap_length);
    for (mmap = (memory_map_t *) mbi->mmap_addr;
         (unsigned long) mmap < mbi->mmap_addr + mbi->mmap_length;
         mmap = (memory_map_t *) ((unsigned long) mmap
                                  + mmap->size + sizeof (mmap->size)))
      printf (" size = 0x%x, base_addr = 0x%x%x,"
```

```
                " length = 0x%x%x, type = 0x%x\n",
                mmap->size, mmap->base_addr_high, mmap->base_addr_low,
                mmap->length_high, mmap->length_low, mmap->type);
    }
}

/* Clear the screen and initialize VIDEO, XPOS and YPOS. */
static void
cls (void)
{
  int i;

  video = (unsigned char *) VIDEO;

  for (i = 0; i < COLUMNS * LINES * 2; i++)
    *(video + i) = 0;

  xpos = 0;
  ypos = 0;
}

/* Convert the integer D to a string and save the string in BUF. If
   BASE is equal to 'd', interpret that D is decimal, and if BASE is
   equal to 'x', interpret that D is hexadecimal. */
static void
itoa (char *buf, int base, int d)
{
  char *p = buf;
  char *p1, *p2;
  unsigned long ud = d;
  int divisor = 10;

  /* If %d is specified and D is minus, put `-' in the head. */
  if (base == 'd' && d < 0)
    {
      *p++ = '-';
      buf++;
      ud = -d;
    }
  else if (base == 'x')
    divisor = 16;

  /* Divide UD by DIVISOR until UD == 0. */
  do
    {
      int remainder = ud % divisor;

      *p++ = (remainder < 10) ? remainder + '0' : remainder + 'a' - 10;
    }
```

```
    while (ud /= divisor);

    /* Terminate BUF. */
    *p = 0;

    /* Reverse BUF. */
    p1 = buf;
    p2 = p - 1;
    while (p1 < p2)
      {
        char tmp = *p1;
        *p1 = *p2;
        *p2 = tmp;
        p1++;
        p2--;
      }
  }

  /* Put the character C on the screen. */
  static void
  putchar (int c)
  {
    if (c == '\n' || c == '\r')
      {
      newline:
        xpos = 0;
        ypos++;
        if (ypos >= LINES)
          ypos = 0;
        return;
      }

    *(video + (xpos + ypos * COLUMNS) * 2) = c & 0xFF;
    *(video + (xpos + ypos * COLUMNS) * 2 + 1) = ATTRIBUTE;

    xpos++;
    if (xpos >= COLUMNS)
      goto newline;
  }

  /* Format a string and print it on the screen, just like the libc
     function printf. */
  void
  printf (const char *format, ...)
  {
    char **arg = (char **) &format;
    int c;
    char buf[20];
```

```
    arg++;

    while ((c = *format++) != 0)
      {
        if (c != '%')
          putchar (c);
        else
          {
            char *p;

            c = *format++;
            switch (c)
              {
              case 'd':
              case 'u':
              case 'x':
                itoa (buf, c, *((int *) arg++));
                p = buf;
                goto string;
                break;

              case 's':
                p = *arg++;
                if (! p)
                  p = "(null)";

              string:
                while (*p)
                  putchar (*p++);
                break;

              default:
                putchar (*((int *) arg++));
                break;
              }
          }
      }
  }
```

5.3.4 Other Multiboot kernels

Other useful information should be available in Multiboot kernels, such as GNU Mach and Fiasco `http://os.inf.tu-dresden.de/fiasco/`. And, it is worth mentioning the OSKit `http://www.cs.utah.edu/projects/flux/oskit/`, which provides a library supporting the specification.

5.4 Example boot loader code

The GNU GRUB (see section "GRUB" in *The GRUB manual*) project is a full Multiboot-compliant boot loader, supporting all required and optional features present in this specification. A public release has not been made, but the test release is available from:

`ftp://alpha.gnu.org/gnu/hurd/src/`

See the webpage `http://www.gnu.org/software/grub/grub.html`, for more information.

Index

(Index is nonexistent)

Table of Contents