The Multiboot Specification version 1.6

Yoshinori K. Okuji, Bryan Ford, Erich Stefan Boleyn, Kunihiro Ishiguro,

1 Introduction to Multiboot Specification

This chapter describes some rough information on the Multiboot Specifcation. Note that this is not a part of the specifcation itself.

1.1 The background of Multiboot Specification

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 diferent 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 fx this problem in existing proprietary operating systems, it shouldn't be too dif cult 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 specifcation aims for. Basically, it specifes 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 specifcation does *not* specify how boot loaders should work | only how they must interface with the operating system being loaded.

1.2 The target architecture

This specifcation 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 specifcation and do not have one already, a variation of this specifcation, stripped of the x86-specifc details, could be adopted for them as well.

1.3 The target operating systems

This specifcation is targeted toward free 32-bit operating systems that can be fairly easily modifed to support the specifcation without going through lots of bureaucratic rigmarole. The particular free operating systems that this specifcation is being primarily designed for are Linux, the kernels of FreeBSD and 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 proprietary operating system vendors eventually adopted this specifcation as well, but that's probably a pipe dream.

1.4 Boot sources

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

Disk-based boot loaders may use a variety of techniques to fnd the relevant OS image and boot module data on disk, such as by interpretation of specifc fle systems (e.g. the BSD/Mach boot loader), using precalculated *blocklists* (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.

1.5 Configure an operating system at boot-time

It is often necessary for one reason or another for the user to be able to provide some confguration information to an operating system dynamically at boot time. While this specifcation should not dictate how this confguration 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.

1.6 How to make OS development easier

OS images should be easy to generate. Ideally, an OS image should simply be an ordinary 32-bit executable fle in whatever fle 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 required to create OS images in a *special* fle format. If this means shifting some work from the operating system to a 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 di f cult.

Unfortunately, there is a horrendous variety of executable fle formats even among free Unix-like PC-based operating systems | generally a diferent 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 diferent types of executable fle formats in existence in order to load the OS image | otherwise the boot loader efectively becomes operating system specifc again.

This specifcation adopts a compromise solution to this problem. Multiboot-compliant OS images always contain a magic *Multiboot header* (see Section 3.1 [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 fle, so kernel images can still conform to the local a.out format variant in addition to being Multiboot-compliant.

1.7 Boot modules

Many modern operating system kernels, such as Mach and the microkernel in VSTa, 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 fle 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 fexible, more space-e f cient,

and more convenient to the operating system and user if the boot loader can load these additional modules independently in the frst place.

Thus, this specifcation 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.

2 The definitions of terms used through the specification

must We use the term *must*, when any boot loader or OS image needs to follow a rule | otherwise, the boot loader or OS image is *not* Multiboot-compliant.

should We use the term *should*, when any boot loader or OS image is recommended to follow a rule, but it doesn't need to follow the rule.

may We use the term *may*, when any boot loader or OS image is allowed to follow a rule.

boot loader
Whatever program or set of programs loads the image of the fnal 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 specifcation. Only the *final* stage of the boot loader | the stage that eventually transfers control to an operating system | must follow the rules specifed in this document in order to be *Multiboot-compliant*; earlier boot loader stages may be designed in whatever way is most convenient.

OS image, kernel
The initial binary image that a boot loader loads into memory and transfers control to start an operating system. The OS image is typically an executable containing the operating system kernel. However it doesn't need to be a part of any OS and may be any kind of system tool.

boot module
Other auxiliary fles that a boot loader loads into memory along with an OS image, but does not interpret in any way other than passing their locations to the operating system when it is invoked.

Multiboot-compliant
A boot loader or an OS image which follows the rules defned as *must* is Multiboot-compliant. When this specifcation specifes a rule as *should* or *may*, a Multiboot-complaint boot loader/OS image doesn't need to follow the rule.

u8 The type of unsigned 8-bit data.

u16 The type of unsigned 16-bit data. Because the target architecture is little-endian, u16 is coded in little-endian.

u32 The type of unsigned 32-bit data. Because the target architecture is little-endian, u32 is coded in little-endian.

u64 The type of unsigned 64-bit data. Because the target architecture is little-endian, u64 is coded in little-endian.

u_phys The type of unsigned data of the same size as target architecture physical address size.

u_virt The type of unsigned data of the same size as target architecture virtual address size.

3 The exact definitions of Multiboot Specification

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

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

3.1 OS image format

An OS 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.

An OS image must contain an additional header called *Multiboot header*, besides the headers of the format used by the OS image. The Multiboot header must be contained completely within the first 32768 bytes of the OS image, and must be 64-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.

3.1.1 The layout of Multiboot header

The layout of the Multiboot header must be as follows:

Offset	Type	Field Name	Note
0	u32	magic	required
4	u32	architecture	required
8	u32	header_length	required
12	u32	checksum	required
16-XX		tags	required

The fields `magic', `architecture', `header_length' and `checksum' are defined in Section 3.1.2 [Header magic fields], page 5, `tags' are defined in Section 3.1.3 [Header tags], page 6. All fields are in native endianness. On bi-endian platforms native-endianness means the endiannes OS image starts in.

3.1.2 The magic fields of Multiboot header

`magic'
: The field `magic' is the magic number identifying the header, which must be the hexadecimal value `0xE85250D6'.

`architecture'
: The field `architecture' specifies the Central Processing Unit Instruction Set Architecture. Since `magic' isn't a palindrome it already specifes the endianness ISAs differing only in endianness recieve the same ID. `0' means 32-bit (protected) mode of i386. `4' means 32-bit MIPS.

`header_length'
: The field `header_length' specifes the Length of multiboot header in bytes including magic fields.

`checksum'

The feld `checksum' is a 32-bit unsigned value which, when added to the other magic felds (i.e. `magic', `architecture' and `header_length'), must have a 32-bit unsigned sum of zero.

3.1.3 General tag structure

Tags constitutes a bufer of structures following each other padded on `u_virt' size. Every structure has following format:

```
        +------------------+
u16     | type             |
u16     | flags            |
u32     | size             |
        +------------------+
```

`type' is divided into 2 parts. Lower contains an identifer of contents of the rest of the tag. `size' contains the size of tag including header felds. If bit `0' of `flags' (also known as `optional') is set if bootloader may ignore this tag if it lacks relevant support. Tags are terminated by a tag of type `0' and size `8'.

3.1.4 Multiboot information request

```
        +------------------+
u16     | type = 1         |
u16     | flags            |
u32     | size             |
u32[n]  | mbi_tag_types    |
        +------------------+
```

`mbi_tag_types' is an array of u32 each one representing an information request If this tag is present and `optional' is set to `0' information conveyed by requested tag types must be present. If bootloader is unable to supply this information it must fail with an error

Note: it doesn't garantee that any tags of type `mbi_tag_types' will actually be present. E.g. on a videoless system even if you requested tag `8' no tags of type `8' will be present in mbi.

3.1.5 The address tag of Multiboot header

```
        +------------------+
u16     | type = 2         |
u16     | flags            |
u32     | size             |
u_virt  | header_addr      |
u_virt  | load_addr        |
u_virt  | load_end_addr    |
u_virt  | bss_end_addr     |
        +------------------+
```

All of the address felds in this tag 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 feld serves to *synchronize* the mapping between OS image ofsets and physical memory addresses.

`load_addr`
: Contains the physical address of the beginning of the text segment. The ofset in the OS image fle at which to start loading is defned by the ofset 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) specifes 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. If this feld is zero, the boot loader assumes that the text and data segments occupy the whole OS image fle.

`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. If this feld is zero, the boot loader assumes that no bss segment is present.

3.1.6 The entry address tag of Multiboot header

```
          +-------------------+
u16       | type = 3          |
u16       | flags             |
u32       | size              |
u_virt    | entry_addr        |
          +-------------------+
```

All of the address felds in this tag are physical addresses. The meaning of each is as follows:

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

3.1.7 Flags tag

```
          +-------------------+
u16       | type = 4          |
u16       | flags             |
u32       | size = 12         |
u32       | console_flags     |
          +-------------------+
```

If this tag is present and bit 0 of `console_flags' is set at least one of supported consoles must be present and information about it must be available in mbi. If bit `1' of `console_flags' is set it indicates that the OS image has EGA text support.

3.1.8 The framebuffer tag of Multiboot header

```
            +-------------------+
    u16     | type = 5          |
    u16     | flags             |
    u32     | size = 20         |
    u32     | width             |
    u32     | height            |
    u32     | depth             |
            +-------------------+
```

This tag specifes the preferred graphics mode. If this tag is present bootloader assumes that the payload has framebufer support. Note that that is only a *recommended* mode by the OS image. Boot loader may choose a diferent mode if it sees ft.

The meaning of each is as follows:

`width` Contains the number of the columns. This is specifed in pixels in a graphics mode, and in characters in a text mode. The value zero indicates that the OS image has no preference.

`height` Contains the number of the lines. This is specifed in pixels in a graphics mode, and in characters in a text mode. The value zero indicates that the OS image has no preference.

`depth` Contains the number of bits per pixel in a graphics mode, and zero in a text mode. The value zero indicates that the OS image has no preference.

3.1.9 Module alignment tag

```
            +-------------------+
    u16     | type = 6          |
    u16     | flags             |
    u32     | size = 12         |
            +-------------------+
```

If this tag is present modules must be page aligned.

3.2 MIPS machine state

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

`` `R4 (also known as A0)' ``
Must contain the magic value `` `0x36d76289' ``; 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).

`` `R5 (also known as A1)' ``
Must contain the 32-bit physical address of the Multiboot information structure provided by the boot loader (see Section 3.4 [Boot information format], page 9).

All other processor registers and fag bits are undefned. This includes, in particular:

`` `R29/SP' `` The OS image must create its own stack as soon as it needs one.

3.3 I386 machine state

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

`EAX' Must contain the magic value `0x36d76289'; 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 3.4 [Boot information format], page 9).

`CS' Must be a 32-bit read/execute code segment with an offset of `0' and a limit of `0xFFFFFFFF'. The exact value is undefned.

`DS'
`ES'
`FS'
`GS'
`SS' Must be a 32-bit read/write data segment with an offset of `0' and a limit of `0xFFFFFFFF'. The exact values are all undefned.

`A20 gate' Must be enabled.

`CR0' Bit 31 (PG) must be cleared. Bit 0 (PE) must be set. Other bits are all undefned.

`EFLAGS' Bit 17 (VM) must be cleared. Bit 9 (IF) must be cleared. Other bits are all undefned.

All other processor registers and fag bits are undefned. This includes, in particular:

`ESP' The OS 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 OS image must not load any segment registers (even just reloading the same values!) until it sets up its own `GDT'.

`IDTR' The OS image must leave interrupts disabled until it sets up its own IDT.

3.4 Boot information

3.4.1 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.

3.4.2 Basic tags structure

Boot information consists of fxed part and a series of tags. Its start is 8-bytes aligned. Fixed part is as following:

```
        +-------------------+
u32     | total_size        |
u32     | reserved          |
        +-------------------+
```

`total_size' contains the total size of boot information including this feld and terminating tag in bytes

`reserved' is always set to zero and must be ignored by OS image

Every tag begins with following felds:

```
        +-------------------+
u32     | type              |
u32     | size              |
        +-------------------+
```

`type' contains an identifer of contents of the rest of the tag. `size' contains the size of tag including header felds but not including padding. Tags follow one another padded when necessary in order for each tag to start at 8-bytes aligned address. Tags are terminated by a tag of type `0' and size `8'.

3.4.3 Basic memory information

```
        +-------------------+
u32     | type = 4          |
u32     | size = 16         |
u32     | mem_lower         |
u32     | mem_upper         |
        +-------------------+
```

`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 frst upper memory hole minus 1 megabyte. It is not guaranteed to be this value.

3.4.4 BIOS Boot device

```
        +-------------------+
u32     | type = 5          |
u32     | size = 20         |
u32     | biosdev           |
u32     | partition         |
u32     | sub_parition      |
        +-------------------+
```

This tag 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 tag must not be present. The operating system may use this feld as a hint for determining its own *root* device, but is not required to.

The `biosdev' contains the BIOS drive number as understood by the BIOS INT 0x13 low-level disk interface: e.g. 0x00 for the frst foppy disk or 0x80 for the frst hard disk.

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

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 `partition' will be 5, and `sub_partiton' will be 0xFFFFFFFF.

3.4.5 Boot command line

```
        +-------------------+
u32     | type = 1          |
u32     | size              |
u8[n]   | string            |
        +-------------------+
```

`string' contains command line. The command line is a normal C-style zero-terminated UTF-8 string.

3.4.6 Modules

```
        +-------------------+
u32     | type = 3          |
u32     | size              |
u_phys  | mod_start         |
u_phys  | mod_end           |
u8[n]   | string            |
        +-------------------+
```

This tag indicates to the kernel what boot module was loaded along with the kernel image, and where it can be found.

The `mod_start' and `mod_end' contain the start and end addresses of the boot module itself. The `string' feld provides an arbitrary string to be associated with that particular boot module; it is a zero-terminated UTF-8 string, just like the kernel command line. 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 fles in a fle system), but its exact use is specifc to the operating system.

One tag appears per module. This tag type may appear multiple times.

3.4.7 ELF-Symbols

```
        +-------------------+
u32     | type = 9          |
u32     | size              |
u16     | num               |
u16     | entsize           |
u16     | shndx             |
u16     | reserved          |
varies  | section headers   |
        +-------------------+
```

This tag contains section header table from an ELF kernel, 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) specifcation in the program header. All sections are loaded, and the physical address felds 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)).

3.4.8 Memory map

This tag provides memory map.

```
        +-------------------+
u32     | type = 6          |
u32     | size              |
u32     | entry_size        |
u32     | entry_version     |
varies  | entries           |
        +-------------------+
```

`entry_size' contains the size of one entry so that in future new felds may be added to it. It's guaranteed to be a multiple of 8. `entry_version' is currently set at `0'. Future versions will increment this feld. Future version are guranteed to be backward compatible with older format. Each entry has the following structure:

```
        +-------------------+
u64     | base_addr         |
u64     | length            |
u32     | type              |
u32     | reserved          |
        +-------------------+
```

`size' contains the size of current entry including this feld itself. It may be bigger than 24 bytes in future versions but is guaranteed to be `base_addr' is the starting physical address. `length' is the size of the memory region in bytes. `type' is the variety of address range represented, where a value of 1 indicates available RAM, value of 3 indicates usable memory holding ACPI information, value of 4 indicates reserved memory which needs to be preserved on hibernation and all other values currently indicated a reserved area. `reserved' is set to `0' by bootloader and must be ignored by the OS image.

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

3.4.9 Boot loader name

```
        +------------------+
u32     | type = 2         |
u32     | size             |
u8[n]   | string           |
        +------------------+
```

`string' contains the name of a boot loader booting the kernel. The name is a normal C-style UTF-8 zero-terminated string.

3.4.10 APM table

The tag type 10 contains APM table

```
        +----------------------+
u32     | type = 10            |
u32     | size = 28            |
u16     | version              |
u16     | cseg                 |
u32     | offset               |
u16     | cseg_16              |
u16     | dseg                 |
u16     | flags                |
u16     | cseg_len             |
u16     | cseg_16_len          |
u16     | dseg_len             |
        +----------------------+
```

The felds `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 ofset of the entry point, the protected mode 16-bit code segment, the protected mode 16-bit data segment, the fags, 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 feld `offset' is 4 bytes, and the others are 2 bytes. See Advanced Power Management (APM) BIOS Interface Specifcation, for more information.

3.4.11 VBE info

```
        +------------------+
u32     | type = 7         |
u32     | size = 784       |
u16     | vbe_mode         |
u16     | vbe_interface_seg |
u16     | vbe_interface_off |
u16     | vbe_interface_len |
u8[512] | vbe_control_info  |
u8[256] | vbe_mode_info     |
        +------------------+
```

The felds `vbe_control_info' and `vbe_mode_info' contain VBE control information returned by the VBE Function 00h and VBE mode information returned by the VBE Function 01h, respectively.

The feld `vbe_mode' indicates current video mode in the format specifed in VBE 3.0.

The rest felds `vbe_interface_seg', `vbe_interface_off', and `vbe_interface_len' contain the table of a protected mode interface defned in VBE 2.0+. If this information is not available, those felds contain zero. Note that VBE 3.0 defnes 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 fnd the table yourself.

3.4.12 Framebuffer info

```
        +--------------------+
u32     | type = 8           |
u32     | size               |
u64     | framebuffer_addr   |
u32     | framebuffer_pitch  |
u32     | framebuffer_width  |
u32     | framebuffer_height |
u8      | framebuffer_bpp    |
u8      | framebuffer_type   |
u8      | reserved           |
varies  | color_info         |
        +--------------------+
```

The feld `framebuffer_addr' contains framebufer physical address. This feld is 64-bit wide but bootloader *should* set it under 4GiB if possible for compatibility with payloads which aren't aware of PAE or amd64. The feld `framebuffer_pitch' contains pitch in bytes. The felds `framebuffer_width', `framebuffer_height' contain framebufer dimensions in pixels. The feld `framebuffer_bpp' contains number of bits per pixel. `reserved' always contains 0 in current version of specifcation and must be ignored by OS image. If `framebuffer_type' is set to 0 it means indexed color. In this case color_info is defned as follows:

```
        +----------------------------------+
u32     | framebuffer_palette_num_colors   |
varies  | framebuffer_palette              |
        +----------------------------------+
```

`framebuffer_palette' is an array of colour descriptors. Each colour descriptor has following structure:

```
        +-------------+
u8      | red_value   |
u8      | green_value |
u8      | blue_value  |
        +-------------+
```

If `framebuffer_type' is set to `1' it means direct RGB color. Then color_type is defned as follows:

```
        +----------------------------------+
u8      | framebuffer_red_field_position   |
u8      | framebuffer_red_mask_size        |
u8      | framebuffer_green_field_position |
u8      | framebuffer_green_mask_size      |
u8      | framebuffer_blue_field_position  |
u8      | framebuffer_blue_mask_size       |
        +----------------------------------+
```

If `framebuffer_type' is set to `2' it means EGA text. In this case `framebuffer_width' and `framebuffer_height' are expressed in characters and not in pixels. `framebuffer_bpp' is equal 16 (16 bits per character) and `framebuffer_pitch' is expressed in bytes per text line. All further values of `framebuffer_type' are reserved for future expansion

4 Examples

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

4.1 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 4.2 [BIOS device mapping techniques], page 16).

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 unmodifed with any reasonable extensions of the BIOS interface, passing along any extra data to be interpreted by the operating system as desired.

4.2 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 fushed 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.

4.2.1 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 identifed.

After activating the device drivers, compare data from the drives using the operating system drivers. This should hopefully be su f cient 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).

4.2.2 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.

4.3 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 `doc' in the Multiboot source distribution.

The kernel `kernel' consists of only three files: `boot.S', `kernel.c' and `multiboot2.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 `multiboot2.h' defines some macros, such as the magic number for the Multiboot header, the Multiboot header structure and the Multiboot information structure.

4.3.1 multiboot2.h

This is the source code in the file `multiboot2.h':

```
/*  multiboot2.h - Multiboot 2 header file. */
/*  Copyright (C) 1999,2003,2007,2008,2009,2010  Free Software Foundation, Inc.
 *
 *  Permission is hereby granted, free of charge, to any person obtaining a copy
 *  of this software and associated documentation files (the "Software"), to
 *  deal in the Software without restriction, including without limitation the
```

```
 *  rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
 *  sell copies of the Software, and to permit persons to whom the Software is
 *  furnished to do so, subject to the following conditions:
 *
 *  The above copyright notice and this permission notice shall be included in
 *  all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EX-
PRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL A
 *  DEVELOPER OR DISTRIBUTOR BE LIABLE FOR ANY CLAIM, DAM-
AGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARIS-
ING FROM, OUT OF OR
 * IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEAL-
INGS IN THE SOFTWARE.
 */

#ifndef MULTIBOOT_HEADER
#define MULTIBOOT_HEADER 1

/*  How many bytes from the start of the fle we search for the header.  */
#define MULTIBOOT_SEARCH                        32768
#define MULTIBOOT_HEADER_ALIGN                  8

/*  The magic feld should contain this.  */
#define MULTIBOOT2_HEADER_MAGIC                 0xe85250d6

/*  This should be in %eax.  */
#define MULTIBOOT2_BOOTLOADER_MAGIC             0x36d76289

/*  Alignment of multiboot modules.  */
#define MULTIBOOT_MOD_ALIGN                     0x00001000

/*  Alignment of the multiboot info structure.  */
#define MULTIBOOT_INFO_ALIGN                    0x00000008

/*  Flags set in the 'fags' member of the multiboot header.  */

#define MULTIBOOT_TAG_ALIGN                  8
#define MULTIBOOT_TAG_TYPE_END               0
#define MULTIBOOT_TAG_TYPE_CMDLINE           1
#define MULTIBOOT_TAG_TYPE_BOOT_LOADER_NAME  2
#define MULTIBOOT_TAG_TYPE_MODULE            3
#define MULTIBOOT_TAG_TYPE_BASIC_MEMINFO     4
#define MULTIBOOT_TAG_TYPE_BOOTDEV           5
```

```
#define MULTIBOOT_TAG_TYPE_MMAP              6
#define MULTIBOOT_TAG_TYPE_VBE               7
#define MULTIBOOT_TAG_TYPE_FRAMEBUFFER       8
#define MULTIBOOT_TAG_TYPE_ELF_SECTIONS      9
#define MULTIBOOT_TAG_TYPE_APM               10

#define MULTIBOOT_HEADER_TAG_END  0
#define MULTIBOOT_HEADER_TAG_INFORMATION_REQUEST  1
#define MULTIBOOT_HEADER_TAG_ADDRESS  2
#define MULTIBOOT_HEADER_TAG_ENTRY_ADDRESS  3
#define MULTIBOOT_HEADER_TAG_CONSOLE_FLAGS  4
#define MULTIBOOT_HEADER_TAG_FRAMEBUFFER  5
#define MULTIBOOT_HEADER_TAG_MODULE_ALIGN  6

#define MULTIBOOT_ARCHITECTURE_I386  0
#define MULTIBOOT_ARCHITECTURE_MIPS32  4
#define MULTIBOOT_HEADER_TAG_OPTIONAL 1

#define MULTIBOOT_CONSOLE_FLAGS_CONSOLE_REQUIRED 1
#define MULTIBOOT_CONSOLE_FLAGS_EGA_TEXT_SUPPORTED 2

#ifndef ASM_FILE

typedef unsigned char           multiboot_uint8_t;
typedef unsigned short          multiboot_uint16_t;
typedef unsigned int            multiboot_uint32_t;
typedef unsigned long long      multiboot_uint64_t;

struct multiboot_header
{
  /* Must be MULTIBOOT_MAGIC - see above. */
  multiboot_uint32_t magic;

  /* ISA */
  multiboot_uint32_t architecture;

  /* Total header length. */
  multiboot_uint32_t header_length;

  /* The above fields plus this one must equal 0 mod 2^32. */
  multiboot_uint32_t checksum;
};

struct multiboot_header_tag
{
  multiboot_uint16_t type;
  multiboot_uint16_t flags;
```

```
  multiboot_uint32_t size;
};

struct multiboot_header_tag_information_request
{
  multiboot_uint16_t type;
  multiboot_uint16_t flags;
  multiboot_uint32_t size;
  multiboot_uint32_t requests[0];
};

struct multiboot_header_tag_address
{
  multiboot_uint16_t type;
  multiboot_uint16_t flags;
  multiboot_uint32_t size;
  multiboot_uint32_t header_addr;
  multiboot_uint32_t load_addr;
  multiboot_uint32_t load_end_addr;
  multiboot_uint32_t bss_end_addr;
};

struct multiboot_header_tag_entry_address
{
  multiboot_uint16_t type;
  multiboot_uint16_t flags;
  multiboot_uint32_t size;
  multiboot_uint32_t entry_addr;
};

struct multiboot_header_tag_console_flags
{
  multiboot_uint16_t type;
  multiboot_uint16_t flags;
  multiboot_uint32_t size;
  multiboot_uint32_t console_flags;
};

struct multiboot_header_tag_framebuffer
{
  multiboot_uint16_t type;
  multiboot_uint16_t flags;
  multiboot_uint32_t size;
  multiboot_uint32_t width;
  multiboot_uint32_t height;
  multiboot_uint32_t depth;
};
```

```
struct multiboot_header_tag_module_align
{
  multiboot_uint16_t type;
  multiboot_uint16_t flags;
  multiboot_uint32_t size;
  multiboot_uint32_t width;
  multiboot_uint32_t height;
  multiboot_uint32_t depth;
};

struct multiboot_color
{
  multiboot_uint8_t red;
  multiboot_uint8_t green;
  multiboot_uint8_t blue;
};

struct multiboot_mmap_entry
{
  multiboot_uint64_t addr;
  multiboot_uint64_t len;
#define MULTIBOOT_MEMORY_AVAILABLE              1
#define MULTIBOOT_MEMORY_RESERVED               2
#define MULTIBOOT_MEMORY_ACPI_RECLAIMABLE       3
#define MULTIBOOT_MEMORY_NVS                    4
  multiboot_uint32_t type;
  multiboot_uint32_t zero;
} __attribute__((packed));
typedef struct multiboot_mmap_entry multiboot_memory_map_t;

struct multiboot_tag
{
  multiboot_uint32_t type;
  multiboot_uint32_t size;
};

struct multiboot_tag_string
{
  multiboot_uint32_t type;
  multiboot_uint32_t size;
  char string[0];
};

struct multiboot_tag_module
{
  multiboot_uint32_t type;
```

```
  multiboot_uint32_t size;
  multiboot_uint32_t mod_start;
  multiboot_uint32_t mod_end;
  char cmdline[0];
};

struct multiboot_tag_basic_meminfo
{
  multiboot_uint32_t type;
  multiboot_uint32_t size;
  multiboot_uint32_t mem_lower;
  multiboot_uint32_t mem_upper;
};

struct multiboot_tag_bootdev
{
  multiboot_uint32_t type;
  multiboot_uint32_t size;
  multiboot_uint32_t biosdev;
  multiboot_uint32_t slice;
  multiboot_uint32_t part;
};

struct multiboot_tag_mmap
{
  multiboot_uint32_t type;
  multiboot_uint32_t size;
  multiboot_uint32_t entry_size;
  multiboot_uint32_t entry_version;
  struct multiboot_mmap_entry entries[0];
};

struct multiboot_vbe_info_block
{
  multiboot_uint8_t external_specification[512];
};

struct multiboot_vbe_mode_info_block
{
  multiboot_uint8_t external_specification[256];
};

struct multiboot_tag_vbe
{
  multiboot_uint32_t type;
  multiboot_uint32_t size;
```

```
  multiboot_uint16_t vbe_mode;
  multiboot_uint16_t vbe_interface_seg;
  multiboot_uint16_t vbe_interface_off;
  multiboot_uint16_t vbe_interface_len;

  struct multiboot_vbe_info_block vbe_control_info;
  struct multiboot_vbe_mode_info_block vbe_mode_info;
};

struct multiboot_tag_framebuffer_common
{
  multiboot_uint32_t type;
  multiboot_uint32_t size;

  multiboot_uint64_t framebuffer_addr;
  multiboot_uint32_t framebuffer_pitch;
  multiboot_uint32_t framebuffer_width;
  multiboot_uint32_t framebuffer_height;
  multiboot_uint8_t framebuffer_bpp;
#define MULTIBOOT_FRAMEBUFFER_TYPE_INDEXED 0
#define MULTIBOOT_FRAMEBUFFER_TYPE_RGB     1
#define MULTIBOOT_FRAMEBUFFER_TYPE_EGA_TEXT     2
  multiboot_uint8_t framebuffer_type;
  multiboot_uint16_t reserved;
};

struct multiboot_tag_framebuffer
{
  struct multiboot_tag_framebuffer_common common;

  union
  {
    struct
    {
      multiboot_uint16_t framebuffer_palette_num_colors;
      struct multiboot_color framebuffer_palette[0];
    };
    struct
    {
      multiboot_uint8_t framebuffer_red_field_position;
      multiboot_uint8_t framebuffer_red_mask_size;
      multiboot_uint8_t framebuffer_green_field_position;
      multiboot_uint8_t framebuffer_green_mask_size;
      multiboot_uint8_t framebuffer_blue_field_position;
      multiboot_uint8_t framebuffer_blue_mask_size;
    };
  };
```

```
};

struct multiboot_tag_elf_sections
{
  multiboot_uint32_t type;
  multiboot_uint32_t size;
  multiboot_uint32_t num;
  multiboot_uint32_t entsize;
  multiboot_uint32_t shndx;
  char sections[0];
};

struct multiboot_tag_apm
{
  multiboot_uint32_t type;
  multiboot_uint32_t size;
  multiboot_uint16_t version;
  multiboot_uint16_t cseg;
  multiboot_uint32_t offset;
  multiboot_uint16_t cseg_16;
  multiboot_uint16_t dseg;
  multiboot_uint16_t flags;
  multiboot_uint16_t cseg_len;
  multiboot_uint16_t cseg_16_len;
  multiboot_uint16_t dseg_len;
};

#endif /* ! ASM_FILE */

#endif /* ! MULTIBOOT_HEADER */
```

4.3.2 boot.S

In the file `boot.S':

```
/* boot.S - bootstrap the kernel */
/* Copyright (C) 1999, 2001, 2010 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 3 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, see <http://www.gnu.org/licenses/>.
 */

#define ASM_FILE        1
#include <multiboot2.h>

/*  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

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

/*  The flags for the Multiboot header. */
#ifdef __ELF__
# define AOUT_KLUDGE 0
#else
# define AOUT_KLUDGE MULTIBOOT_AOUT_KLUDGE
#endif

        .text

        .globl  start, _start
start:
_start:
        jmp     multiboot_entry

        /*  Align 64 bits boundary. */
        .align  8

        /*  Multiboot header. */
multiboot_header:
        /*  magic */
        .long   MULTIBOOT2_HEADER_MAGIC
        /*  ISA: i386 */
        .long   GRUB_MULTIBOOT_ARCHITECTURE_I386
        /*  Header length. */
        .long   multiboot_header_end - multiboot_header
        /*  checksum */
        .long   -(MULTIBOOT2_HEADER_MAGIC + GRUB_MULTIBOOT_ARCHITECTURE_I386 + (multib
#ifndef __ELF__
address_tag_start:
        .short MULTIBOOT_HEADER_TAG_ADDRESS
```

```
        .short MULTIBOOT_HEADER_TAG_OPTIONAL
        .long address_tag_end - address_tag_start
        /* header_addr */
        .long   multiboot_header
        /* load_addr */
        .long   _start
        /* load_end_addr */
        .long   _edata
        /* bss_end_addr */
        .long   _end
address_tag_end:
entry_address_tag_start:
        .short MULTIBOOT_HEADER_TAG_ENTRY_ADDRESS
        .short MULTIBOOT_HEADER_TAG_OPTIONAL
        .long entry_address_tag_end - entry_address_tag_start
        /* entry_addr */
        .long multiboot_entry
entry_address_tag_end:
#endif /* __ELF__ */
framebuffer_tag_start:
        .short MULTIBOOT_HEADER_TAG_FRAMEBUFFER
        .short MULTIBOOT_HEADER_TAG_OPTIONAL
        .long framebuffer_tag_end - framebuffer_tag_start
        .long 1024
        .long 768
        .long 32
framebuffer_tag_end:
        .short MULTIBOOT_HEADER_TAG_END
        .short 0
        .long 8
multiboot_header_end:
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
```

4.3.3 kernel.c

And, in the file `kernel.c':

```
/* kernel.c - the C part of the kernel */
/* Copyright (C) 1999, 2010  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 3 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, see <http://www.gnu.org/licenses/>.
 */

#include "multiboot2.h"

/* Macros. */

/* Some screen stuf. */
/* 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)
{
  struct multiboot_tag *tag;
  unsigned size;

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

  /* Am I booted by a Multiboot-compliant boot loader? */
  if (magic != MULTIBOOT2_BOOTLOADER_MAGIC)
    {
      printf ("Invalid magic number: 0x%x\n", (unsigned) magic);
      return;
    }

  if (addr & 7)
    {
      printf ("Unaligned mbi: 0x%x\n", addr);
      return;
    }

  size = *(unsigned *) addr;
  printf ("Announced mbi size 0x%x\n", size);
  for (tag = (struct multiboot_tag *) (addr + 8);
       tag->type != MULTIBOOT_TAG_TYPE_END;
       tag = (struct multiboot_tag *) ((multiboot_uint8_t *) tag
                                       + ((tag->size + 7) & ~7)))
    {
      printf ("Tag 0x%x, Size 0x%x\n", tag->type, tag->size);
```

```
switch (tag->type)
  {
  case MULTIBOOT_TAG_TYPE_CMDLINE:
    printf ("Command line = %s\n",
            ((struct multiboot_tag_string *) tag)->string);
    break;
  case MULTIBOOT_TAG_TYPE_BOOT_LOADER_NAME:
    printf ("Boot loader name = %s\n",
            ((struct multiboot_tag_string *) tag)->string);
    break;
  case MULTIBOOT_TAG_TYPE_MODULE:
    printf ("Module at 0x%x-0x%x. Command line %s\n",
            ((struct multiboot_tag_module *) tag)->mod_start,
            ((struct multiboot_tag_module *) tag)->mod_end,
            ((struct multiboot_tag_module *) tag)->cmdline);
    break;
  case MULTIBOOT_TAG_TYPE_BASIC_MEMINFO:
    printf ("mem_lower = %uKB, mem_upper = %uKB\n",
            ((struct multiboot_tag_basic_meminfo *) tag)->mem_lower,
            ((struct multiboot_tag_basic_meminfo *) tag)->mem_upper);
    break;
  case MULTIBOOT_TAG_TYPE_BOOTDEV:
    printf ("Boot device 0x%x,%u,%u\n",
            ((struct multiboot_tag_bootdev *) tag)->biosdev,
            ((struct multiboot_tag_bootdev *) tag)->slice,
            ((struct multiboot_tag_bootdev *) tag)->part);
    break;
  case MULTIBOOT_TAG_TYPE_MMAP:
    {
      multiboot_memory_map_t *mmap;

      printf ("mmap\n");

      for (mmap = ((struct multiboot_tag_mmap *) tag)->entries;
           (multiboot_uint8_t *) mmap
             < (multiboot_uint8_t *) tag + tag->size;
           mmap = (multiboot_memory_map_t *)
             ((unsigned long) mmap
              + ((struct multiboot_tag_mmap *) tag)->entry_size))
        printf (" base_addr = 0x%x%x,"
                " length = 0x%x%x, type = 0x%x\n",
                (unsigned) (mmap->addr >> 32),
                (unsigned) (mmap->addr & 0xffffffff),
                (unsigned) (mmap->len >> 32),
                (unsigned) (mmap->len & 0xffffffff),
                (unsigned) mmap->type);
    }
```

```
  break;
case MULTIBOOT_TAG_TYPE_FRAMEBUFFER:
  {
    multiboot_uint32_t color;
    unsigned i;
    struct multiboot_tag_framebuffer *tagfb
      = (struct multiboot_tag_framebuffer *) tag;
    void *fb = (void *) (unsigned long) tagfb->common.framebuffer_addr;

    switch (tagfb->common.framebuffer_type)
      {
      case MULTIBOOT_FRAMEBUFFER_TYPE_INDEXED:
        {
          unsigned best_distance, distance;
          struct multiboot_color *palette;

          palette = tagfb->framebuffer_palette;

          color = 0;
          best_distance = 4*256*256;

          for (i = 0; i < tagfb->framebuffer_palette_num_colors; i++)
            {
              distance = (0xff - palette[i].blue)
                * (0xff - palette[i].blue)
                + palette[i].red * palette[i].red
                + palette[i].green * palette[i].green;
              if (distance < best_distance)
                {
                  color = i;
                  best_distance = distance;
                }
            }
        }
        break;

      case MULTIBOOT_FRAMEBUFFER_TYPE_RGB:
        color = ((1 << tagfb->framebuffer_blue_mask_size) - 1)
          << tagfb->framebuffer_blue_field_position;
        break;

      case MULTIBOOT_FRAMEBUFFER_TYPE_EGA_TEXT:
        color = '\\' | 0x0100;
        break;

      default:
        color = 0xffffffff;
```

```
                break;
              }

            for (i = 0; i < tagfb->common.framebuffer_width
                   && i < tagfb->common.framebuffer_height; i++)
              {
                switch (tagfb->common.framebuffer_bpp)
                  {
                  case 8:
                    {
                      multiboot_uint8_t *pixel = fb
                        + tagfb->common.framebuffer_pitch * i + i;
                      *pixel = color;
                    }
                    break;
                  case 15:
                  case 16:
                    {
                      multiboot_uint16_t *pixel
                        = fb + tagfb->common.framebuffer_pitch * i + 2 * i;
                      *pixel = color;
                    }
                    break;
                  case 24:
                    {
                      multiboot_uint32_t *pixel
                        = fb + tagfb->common.framebuffer_pitch * i + 3 * i;
                      *pixel = (color & 0xffffff) | (*pixel & 0xff000000);
                    }
                    break;

                  case 32:
                    {
                      multiboot_uint32_t *pixel
                        = fb + tagfb->common.framebuffer_pitch * i + 4 * i;
                      *pixel = color;
                    }
                    break;
                  }
              }
            break;
          }

      }
  }
tag = (struct multiboot_tag *) ((multiboot_uint8_t *) tag
                                + ((tag->size + 7) & ~7));
```

```
  printf ("Total mbi size 0x%x\n", (unsigned) tag - addr);
}

/* Clear the screen and initialize VIDEO, XPOS and YPOS. */
static void
cls (void)
{
  putchar ('\n');
  putchar ('\r');

#if 0
  int i;

  video = (unsigned char *) VIDEO;

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

  xpos = 0;
  ypos = 0;
#endif
}

/* 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 specifed 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--;
    }
}

#define UART_LSR        5
#define UART_EMPTY_TRANSMITTER  0x20

/* Put the character C on the screen. */
static void
putchar (int c)
{
  while ((*(volatile char *) (0xbff003f8 + UART_LSR) & UART_EMPTY_TRANSMITTER)
         == 0);
  *(volatile char *) 0xbff003f8 = c;
  if (c == '\n')
    putchar ('\r');
#if 0
  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;
#endif
}

/* 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, *p2;
          int pad0 = 0, pad = 0;

          c = *format++;
          if (c == '0')
            {
              pad0 = 1;
              c = *format++;
            }

          if (c >= '0' && c <= '9')
            {
              pad = c - '0';
              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:
                for (p2 = p; *p2; p2++);
                for (; p2 < p + pad; p2++)
                  putchar (pad0 ? '0' : ' ');
                while (*p)
                  putchar (*p++);
                break;

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

4.3.4 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 specifcation.

4.4 Example boot loader code

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

ftp://alpha.gnu.org/gnu/grub

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

5 The change log of this specification

0.7

- *Multiboot Standard* is renamed to *Multiboot Specification*.
- Graphics felds are added to Multiboot header.
- BIOS drive information, BIOS confguration table, the name of a boot loader, APM information, and graphics information are added to Multiboot information.
- Rewritten in Texinfo format.
- Rewritten, using more strict words.
- The maintainer changes to the GNU GRUB maintainer team bug-grub@gnu.org, from Bryan Ford and Erich Stefan Boleyn.
- The byte order of the `boot_device' in Multiboot information is reversed. This was a mistake.
- The ofset of the address felds were wrong.
- The format is adapted to a newer Texinfo, and the version number is specifed more explicitly in the title.

0.6

- A few wording changes.
- Header checksum.
- Classifcation of machine state passed to an operating system.

0.5

- Name change.

0.4

- Major changes plus HTMLifcation.

Index

(Index is nonexistent)

Table of Contents

1 Introduction to Multiboot Specification 1
1.1 The background of Multiboot Specifcation 1
1.2 The target architecture 1
1.3 The target operating systems 1
1.4 Boot sources 1
1.5 Confgure an operating system at boot-time 2
1.6 How to make OS development easier 2
1.7 Boot modules 2

2 The definitions of terms used through the specification 4

3 The exact definitions of Multiboot Specification 5
3.1 OS image format 5
3.1.1 The layout of Multiboot header 5
3.1.2 The magic felds of Multiboot header 5
3.1.3 General tag structure 6
3.1.4 Multiboot information request 6
3.1.5 The address tag of Multiboot header 6
3.1.6 The entry address tag of Multiboot header 7
3.1.7 Flags tag 7
3.1.8 The framebufer tag of Multiboot header 8
3.1.9 Module alignment tag 8
3.2 MIPS machine state 8
3.3 I386 machine state 9
3.4 Boot information 9
3.4.1 Boot information format 9
3.4.2 Basic tags structure 10
3.4.3 Basic memory information 10
3.4.4 BIOS Boot device 10
3.4.5 Boot command line 11
3.4.6 Modules 11
3.4.7 ELF-Symbols 12
3.4.8 Memory map 12
3.4.9 Boot loader name 13
3.4.10 APM table 13
3.4.11 VBE info 13
3.4.12 Framebufer info 14

4 Examples 16
4.1 Notes on PC 16
4.2 BIOS device mapping techniques 16
4.2.1 Data comparison technique 16
4.2.2 I/O restriction technique 17
4.3 Example OS code 17
4.3.1 multiboot2.h 17
4.3.2 boot.S 24
4.3.3 kernel.c 27
4.3.4 Other Multiboot kernels 35
4.4 Example boot loader code 35

5 The change log of this specification 36

Index 37