The Linux/x86 Boot Protocol

On the x86 platform, the Linux kernel uses a rather complicated boot convention. This has evolved partially due to historical aspects, as well as the desire in the early days to have the kernel itself be a bootable image, the complicated PC memory model and due to changed expectations in the PC industry caused by the effective demise of real-mode DOS as a mainstream operating system.

Currently, the following versions of the Linux/x86 boot protocol exist.

Old kernels	zImage/Image support only. Some very early kernels may not even support a command line.
Protocol 2.00	(Kernel 1.3.73) Added bzImage and initrd support, as well as a formalized way to communicate between the boot loader and the kernel. setup.S made relocatable, although the traditional setup area still assumed writable.
Protocol 2.01	(Kernel 1.3.76) Added a heap overrun warning.
Protocol 2.02	(Kernel 2.4.0-test3-pre3) New command line protocol. Lower the conventional memory ceiling. No overwrite of the traditional setup area, thus making booting safe for systems which use the EBDA from SMM or 32-bit BIOS entry points. zImage deprecated but still supported.
Protocol 2.03	(Kernel 2.4.18-pre1) Explicitly makes the highest possible initrd address available to the bootloader.
Protocol 2.04	(Kernel 2.6.14) Extend the syssize field to four bytes.
Protocol 2.05	(Kernel 2.6.20) Make protected mode kernel relocatable. Introduce relocatable_kernel and kernel_alignment fields.
Protocol 2.06	(Kernel 2.6.22) Added a field that contains the size of the boot command line.
Protocol 2.07	(Kernel 2.6.24) Added paravirtualised boot protocol. Introduced hardware_subarch and hardware_subarch_data and KEEP_SEGMENTS flag in load_flags.
Protocol 2.08	(Kernel 2.6.26) Added crc32 checksum and ELF format payload. Introduced payload_offset and payload_length fields to aid in locating the payload.
Protocol 2.09	(Kernel 2.6.26) Added a field of 64-bit physical pointer to single linked list of struct setup_data.
Protocol 2.10	(Kernel 2.6.31) Added a protocol for relaxed alignment beyond the kernel_alignment added, new init_size and pref_address fields. Added extended boot loader IDs.
Protocol 2.11	(Kernel 3.6) Added a field for offset of EFI handover protocol entry point.
Protocol 2.12	(Kernel 3.8) Added the xloadflags field and extension fields to struct boot_params for loading bzImage and ramdisk above 4G in 64bit.
Protocol 2.13	(Kernel 3.14) Support 32- and 64-bit flags being set in xloadflags to support booting a 64-bit kernel from 32-bit EFI
Protocol 2.14	BURNT BY INCORRECT COMMIT ae7e1238e68f2a472a125673ab506d49158c1889 (x86/boot: Add ACPI RSDP address to setup_header) DO NOT USE!!! ASSUME SAME AS 2.13.
Protocol 2.15	(Kernel 5.5) Added the kernel_info and kernel_info.setup_type_max.

Note

The protocol version number should be changed only if the setup header is changed. There is no need to update the version number if boot_params or kernel_info are changed. Additionally, it is recommended to use xloadflags (in this case the protocol version number should not be updated either) or kernel_info to communicate supported Linux kernel features to the boot loader. Due to very limited space available in the original setup header every update to it should be considered with great care. Starting from the protocol 2.15 the primary way to communicate things to the boot loader is the kernel_info.

Memory Layout

The traditional memory map for the kernel loader, used for Image or zImage kernels, typically looks like:

```
        |                        |
0A0000  +------------------------+
        |  Reserved for BIOS     |      Do not use.  Reserved for BIOS EBDA.
09A000  +------------------------+
        |  Command line          |
        |  Stack/heap            |      For use by the kernel real-mode code.
098000  +------------------------+
        |  Kernel setup          |      The kernel real-mode code.
090200  +------------------------+
        |  Kernel boot sector    |      The kernel legacy boot sector.
090000  +------------------------+
        |  Protected-mode kernel |      The bulk of the kernel image.
010000  +------------------------+
        |  Boot loader           |      <- Boot sector entry point 0000:7C00
001000  +------------------------+
        |  Reserved for MBR/BIOS |
000800  +------------------------+
```

```
        |  Typically used by MBR |
000600  +------------------------+
        |  BIOS use only         |
000000  +------------------------+
```

When using bzImage, the protected-mode kernel was relocated to 0x100000 ("high memory"), and the kernel real-mode block (boot sector, setup, and stack/heap) was made relocatable to any address between 0x10000 and end of low memory. Unfortunately, in protocols 2.00 and 2.01 the 0x90000+ memory range is still used internally by the kernel; the 2.02 protocol resolves that problem.

It is desirable to keep the "memory ceiling" -- the highest point in low memory touched by the boot loader -- as low as possible, since some newer BIOSes have begun to allocate some rather large amounts of memory, called the Extended BIOS Data Area, near the top of low memory. The boot loader should use the "INT 12h" BIOS call to verify how much low memory is available.

Unfortunately, if INT 12h reports that the amount of memory is too low, there is usually nothing the boot loader can do but to report an error to the user. The boot loader should therefore be designed to take up as little space in low memory as it reasonably can. For zImage or old bzImage kernels, which need data written into the 0x90000 segment, the boot loader should make sure not to use memory above the 0x9A000 point; too many BIOSes will break above that point.

For a modern bzImage kernel with boot protocol version >= 2.02, a memory layout like the following is suggested:

```
        ~                        ~
        |  Protected-mode kernel |
100000  +------------------------+
        |  I/O memory hole       |
0A0000  +------------------------+
        |  Reserved for BIOS     |      Leave as much as possible unused
        ~                        ~
        |  Command line          |      (Can also be below the X+10000 mark)
X+10000 +------------------------+
        |  Stack/heap            |      For use by the kernel real-mode code.
X+08000 +------------------------+
        |  Kernel setup          |      The kernel real-mode code.
        |  Kernel boot sector    |      The kernel legacy boot sector.
X       +------------------------+
        |  Boot loader           |      <- Boot sector entry point 0000:7C00
001000  +------------------------+
        |  Reserved for MBR/BIOS |
000800  +------------------------+
        |  Typically used by MBR |
000600  +------------------------+
        |  BIOS use only         |
000000  +------------------------+

... where the address X is as low as the design of the boot loader permits.
```

The Real-Mode Kernel Header

In the following text, and anywhere in the kernel boot sequence, "a sector" refers to 512 bytes. It is independent of the actual sector size of the underlying medium.

The first step in loading a Linux kernel should be to load the real-mode code (boot sector and setup code) and then examine the following header at offset 0x01f1. The real-mode code can total up to 32K, although the boot loader may choose to load only the first two sectors (1K) and then examine the bootup sector size.

The header looks like:

Offset/Size	Proto	Name	Meaning
01F1/1	ALL(1)	setup_sects	The size of the setup in sectors
01F2/2	ALL	root_flags	If set, the root is mounted readonly
01F4/4	2.04+(2)	syssize	The size of the 32-bit code in 16-byte paras
01F8/2	ALL	ram_size	DO NOT USE - for bootsect.S use only
01FA/2	ALL	vid_mode	Video mode control
01FC/2	ALL	root_dev	Default root device number
01FE/2	ALL	boot_flag	0xAA55 magic number
0200/2	2.00+	jump	Jump instruction
0202/4	2.00+	header	Magic signature "HdrS"
0206/2	2.00+	version	Boot protocol version supported
0208/4	2.00+	realmode_swtch	Boot loader hook (see below)
020C/2	2.00+	start_sys_seg	The load-low segment (0x1000) (obsolete)
020E/2	2.00+	kernel_version	Pointer to kernel version string
0210/1	2.00+	type_of_loader	Boot loader identifier
0211/1	2.00+	loadflags	Boot protocol option flags
0212/2	2.00+	setup_move_size	Move to high memory size (used with hooks)
0214/4	2.00+	code32_start	Boot loader hook (see below)

Offset/Size	Proto	Name	Meaning
0218/4	2.00+	ramdisk_image	initrd load address (set by boot loader)
021C/4	2.00+	ramdisk_size	initrd size (set by boot loader)
0220/4	2.00+	bootsect_kludge	DO NOT USE - for bootsect.S use only
0224/2	2.01+	heap_end_ptr	Free memory after setup end
0226/1	2.02+(3)	ext_loader_ver	Extended boot loader version
0227/1	2.02+(3)	ext_loader_type	Extended boot loader ID
0228/4	2.02+	cmd_line_ptr	32-bit pointer to the kernel command line
022C/4	2.03+	initrd_addr_max	Highest legal initrd address
0230/4	2.05+	kernel_alignment	Physical addr alignment required for kernel
0234/1	2.05+	relocatable_kernel	Whether kernel is relocatable or not
0235/1	2.10+	min_alignment	Minimum alignment, as a power of two
0236/2	2.12+	xloadflags	Boot protocol option flags
0238/4	2.06+	cmdline_size	Maximum size of the kernel command line
023C/4	2.07+	hardware_subarch	Hardware subarchitecture
0240/8	2.07+	hardware_subarch_data	Subarchitecture-specific data
0248/4	2.08+	payload_offset	Offset of kernel payload
024C/4	2.08+	payload_length	Length of kernel payload
0250/8	2.09+	setup_data	64-bit physical pointer to linked list of struct setup_data
0258/8	2.10+	pref_address	Preferred loading address
0260/4	2.10+	init_size	Linear memory required during initialization
0264/4	2.11+	handover_offset	Offset of handover entry point
0268/4	2.15+	kernel_info_offset	Offset of the kernel_info

Note

1. For backwards compatibility, if the setup_sects field contains 0, the real value is 4.
2. For boot protocol prior to 2.04, the upper two bytes of the syssize field are unusable, which means the size of a bzImage kernel cannot be determined.
3. Ignored, but safe to set, for boot protocols 2.02-2.09.

If the "HdrS" (0x53726448) magic number is not found at offset 0x202, the boot protocol version is "old". Loading an old kernel, the following parameters should be assumed:

```
Image type = zImage
initrd not supported
Real-mode kernel must be located at 0x90000.
```

Otherwise, the "version" field contains the protocol version, e.g. protocol version 2.01 will contain 0x0201 in this field. When setting fields in the header, you must make sure only to set fields supported by the protocol version in use.

Details of Header Fields

For each field, some are information from the kernel to the bootloader ("read"), some are expected to be filled out by the bootloader ("write"), and some are expected to be read and modified by the bootloader ("modify").

All general purpose boot loaders should write the fields marked (obligatory). Boot loaders who want to load the kernel at a nonstandard address should fill in the fields marked (reloc); other boot loaders can ignore those fields.

The byte order of all fields is littleendian (this is x86, after all.)

Field name:	setup_sects
Type:	read
Offset/size:	0x1f1/1
Protocol:	ALL

The size of the setup code in 512-byte sectors. If this field is 0, the real value is 4. The real-mode code consists of the boot sector (always one 512-byte sector) plus the setup code.

Field name:	root_flags
Type:	modify (optional)
Offset/size:	0x1f2/2
Protocol:	ALL

If this field is nonzero, the root defaults to readonly. The use of this field is deprecated; use the "ro" or "rw" options on the command line instead.

Field name:	syssize
Type:	read
Offset/size:	0x1f4/4 (protocol 2.04+) 0x1f4/2 (protocol ALL)
Protocol:	2.04+

The size of the protected-mode code in units of 16-byte paragraphs. For protocol versions older than 2.04 this field is only two bytes wide, and therefore cannot be trusted for the size of a kernel if the LOAD_HIGH flag is set.

Field name:	ram_size
Type:	kernel internal
Offset/size:	0x1f8/2
Protocol:	ALL

This field is obsolete.

Field name:	vid_mode
Type:	modify (obligatory)
Offset/size:	0x1fa/2

Please see the section on SPECIAL COMMAND LINE OPTIONS.

Field name:	root_dev
Type:	modify (optional)
Offset/size:	0x1fc/2
Protocol:	ALL

The default root device device number. The use of this field is deprecated, use the "root=" option on the command line instead.

Field name:	boot_flag
Type:	read
Offset/size:	0x1fe/2
Protocol:	ALL

Contains 0xAA55. This is the closest thing old Linux kernels have to a magic number.

Field name:	jump
Type:	read
Offset/size:	0x200/2
Protocol:	2.00+

Contains an x86 jump instruction, 0xEB followed by a signed offset relative to byte 0x202. This can be used to determine the size of the header.

Field name:	header
Type:	read
Offset/size:	0x202/4
Protocol:	2.00+

Contains the magic number "HdrS" (0x53726448).

Field name:	version
Type:	read
Offset/size:	0x206/2
Protocol:	2.00+

Contains the boot protocol version, in (major << 8)+minor format, e.g. 0x0204 for version 2.04, and 0x0a11 for a hypothetical version 10.17.

Field name:	realmode_swtch
Type:	modify (optional)
Offset/size:	0x208/4
Protocol:	2.00+

Boot loader hook (see ADVANCED BOOT LOADER HOOKS below.)

Field name:	start_sys_seg
Type:	read
Offset/size:	0x20c/2
Protocol:	2.00+

The load low segment (0x1000). Obsolete.

Field name:	kernel_version
Type:	read
Offset/size:	0x20e/2
Protocol:	2.00+

If set to a nonzero value, contains a pointer to a NUL-terminated human-readable kernel version number string, less 0x200. This can be used to display the kernel version to the user. This value should be less than (0x200*setup_sects).

For example, if this value is set to 0x1c00, the kernel version number string can be found at offset 0x1e00 in the kernel file. This is a valid value if and only if the "setup_sects" field contains the value 15 or higher, as:

```
0x1c00  < 15*0x200 (= 0x1e00) but
0x1c00 >= 14*0x200 (= 0x1c00)

0x1c00 >> 9 = 14, So the minimum value for setup_secs is 15.
```

Field name:	type_of_loader
Type:	write (obligatory)
Offset/size:	0x210/1
Protocol:	2.00+

If your boot loader has an assigned id (see table below), enter 0xTV here, where T is an identifier for the boot loader and V is a version number. Otherwise, enter 0xFF here.

For boot loader IDs above T = 0xD, write T = 0xE to this field and write the extended ID minus 0x10 to the ext_loader_type field. Similarly, the ext_loader_ver field can be used to provide more than four bits for the bootloader version.

For example, for T = 0x15, V = 0x234, write:

```
type_of_loader  <- 0xE4
ext_loader_type <- 0x05
ext_loader_ver  <- 0x23
```

Assigned boot loader ids (hexadecimal):

0	LILO (0x00 reserved for pre-2.00 bootloader)
1	Loadlin
2	bootsect-loader (0x20, all other values reserved)
3	Syslinux
4	Etherboot/gPXE/iPXE
5	ELILO
7	GRUB
8	U-Boot
9	Xen
A	Gujin
B	Qemu
C	Arcturus Networks uCbootloader
D	kexec-tools
E	Extended (see ext_loader_type)
F	Special (0xFF = undefined)
10	Reserved
11	Minimal Linux Bootloader <http://sebastian-plotz.blogspot.de>
12	OVMF UEFI virtualization stack

Please contact <hpa@zytor.com> if you need a bootloader ID value assigned.

Field name:	loadflags
Type:	modify (obligatory)

Offset/size:	0x211/1
Protocol:	2.00+

This field is a bitmask.

Bit 0 (read): LOADED_HIGH

- If 0, the protected-mode code is loaded at 0x10000.
- If 1, the protected-mode code is loaded at 0x100000.

Bit 1 (kernel internal): KASLR_FLAG

- Used internally by the compressed kernel to communicate KASLR status to kernel proper.
  - If 1, KASLR enabled.
  - If 0, KASLR disabled.

Bit 5 (write): QUIET_FLAG

- If 0, print early messages.
- If 1, suppress early messages.

  This requests to the kernel (decompressor and early kernel) to not write early messages that require accessing the display hardware directly.

Bit 6 (obsolete): KEEP_SEGMENTS

Protocol: 2.07+

- This flag is obsolete.

Bit 7 (write): CAN_USE_HEAP

Set this bit to 1 to indicate that the value entered in the heap_end_ptr is valid. If this field is clear, some setup code functionality will be disabled.

Field name:	setup_move_size
Type:	modify (obligatory)
Offset/size:	0x212/2
Protocol:	2.00-2.01

When using protocol 2.00 or 2.01, if the real mode kernel is not loaded at 0x90000, it gets moved there later in the loading sequence. Fill in this field if you want additional data (such as the kernel command line) moved in addition to the real-mode kernel itself.

The unit is bytes starting with the beginning of the boot sector.

This field is can be ignored when the protocol is 2.02 or higher, or if the real-mode code is loaded at 0x90000.

Field name:	code32_start
Type:	modify (optional, reloc)
Offset/size:	0x214/4
Protocol:	2.00+

The address to jump to in protected mode. This defaults to the load address of the kernel, and can be used by the boot loader to determine the proper load address.

This field can be modified for two purposes:

1. as a boot loader hook (see Advanced Boot Loader Hooks below.)
2. if a bootloader which does not install a hook loads a relocatable kernel at a nonstandard address it will have to modify this field to point to the load address.

Field name:	ramdisk_image
Type:	write (obligatory)
Offset/size:	0x218/4
Protocol:	2.00+

The 32-bit linear address of the initial ramdisk or ramfs. Leave at zero if there is no initial ramdisk/ramfs.

Field name:	ramdisk_size
Type:	write (obligatory)
Offset/size:	0x21c/4
Protocol:	2.00+

Size of the initial ramdisk or ramfs. Leave at zero if there is no initial ramdisk/ramfs.

Field name:	bootsect_kludge
Type:	kernel internal
Offset/size:	0x220/4
Protocol:	2.00+

This field is obsolete.

Field name:	heap_end_ptr
Type:	write (obligatory)
Offset/size:	0x224/2
Protocol:	2.01+

Set this field to the offset (from the beginning of the real-mode code) of the end of the setup stack/heap, minus 0x0200.

Field name:	ext_loader_ver
Type:	write (optional)
Offset/size:	0x226/1
Protocol:	2.02+

This field is used as an extension of the version number in the type_of_loader field. The total version number is considered to be (type_of_loader & 0x0f) + (ext_loader_ver << 4).

The use of this field is boot loader specific. If not written, it is zero.

Kernels prior to 2.6.31 did not recognize this field, but it is safe to write for protocol version 2.02 or higher.

Field name:	ext_loader_type
Type:	write (obligatory if (type_of_loader & 0xf0) == 0xe0)
Offset/size:	0x227/1
Protocol:	2.02+

This field is used as an extension of the type number in type_of_loader field. If the type in type_of_loader is 0xE, then the actual type is (ext_loader_type + 0x10).

This field is ignored if the type in type_of_loader is not 0xE.

Kernels prior to 2.6.31 did not recognize this field, but it is safe to write for protocol version 2.02 or higher.

Field name:	cmd_line_ptr
Type:	write (obligatory)
Offset/size:	0x228/4
Protocol:	2.02+

Set this field to the linear address of the kernel command line. The kernel command line can be located anywhere between the end of the setup heap and 0xA0000; it does not have to be located in the same 64K segment as the real-mode code itself.

Fill in this field even if your boot loader does not support a command line, in which case you can point this to an empty string (or better yet, to the string "auto".) If this field is left at zero, the kernel will assume that your boot loader does not support the 2.02+ protocol.

Field name:	initrd_addr_max
Type:	read
Offset/size:	0x22c/4
Protocol:	2.03+

The maximum address that may be occupied by the initial ramdisk/ramfs contents. For boot protocols 2.02 or earlier, this field is not present, and the maximum address is 0x37FFFFFF. (This address is defined as the address of the highest safe byte, so if your ramdisk is exactly 131072 bytes long and this field is 0x37FFFFFF, you can start your ramdisk at 0x37FE0000.)

Field name:	kernel_alignment
Type:	read/modify (reloc)
Offset/size:	0x230/4
Protocol:	2.05+ (read), 2.10+ (modify)

Alignment unit required by the kernel (if relocatable_kernel is true.) A relocatable kernel that is loaded at an alignment incompatible with the value in this field will be realigned during kernel initialization.

Starting with protocol version 2.10, this reflects the kernel alignment preferred for optimal performance; it is possible for the loader to modify this field to permit a lesser alignment. See the min_alignment and pref_address field below.

Field name:	relocatable_kernel
Type:	read (reloc)
Offset/size:	0x234/1
Protocol:	2.05+

If this field is nonzero, the protected-mode part of the kernel can be loaded at any address that satisfies the kernel_alignment field. After loading, the boot loader must set the code32_start field to point to the loaded code, or to a boot loader hook.

Field name:	min_alignment
Type:	read (reloc)
Offset/size:	0x235/1
Protocol:	2.10+

This field, if nonzero, indicates as a power of two the minimum alignment required, as opposed to preferred, by the kernel to boot. If a boot loader makes use of this field, it should update the kernel_alignment field with the alignment unit desired; typically:

```
kernel_alignment = 1 << min_alignment
```

There may be a considerable performance cost with an excessively misaligned kernel. Therefore, a loader should typically try each power-of-two alignment from kernel_alignment down to this alignment.

Field name:	xloadflags
Type:	read
Offset/size:	0x236/2
Protocol:	2.12+

This field is a bitmask.

Bit 0 (read): XLF_KERNEL_64

- If 1, this kernel has the legacy 64-bit entry point at 0x200.

Bit 1 (read): XLF_CAN_BE_LOADED_ABOVE_4G

- If 1, kernel/boot_params/cmdline/ramdisk can be above 4G.

Bit 2 (read): XLF_EFI_HANDOVER_32

- If 1, the kernel supports the 32-bit EFI handoff entry point given at handover_offset.

Bit 3 (read): XLF_EFI_HANDOVER_64

- If 1, the kernel supports the 64-bit EFI handoff entry point given at handover_offset + 0x200.

Bit 4 (read): XLF_EFI_KEXEC

- If 1, the kernel supports kexec EFI boot with EFI runtime support.

Field name:	cmdline_size
Type:	read
Offset/size:	0x238/4
Protocol:	2.06+

The maximum size of the command line without the terminating zero. This means that the command line can contain at most cmdline_size characters. With protocol version 2.05 and earlier, the maximum size was 255.

Field name:	hardware_subarch
Type:	write (optional, defaults to x86/PC)
Offset/size:	0x23c/4
Protocol:	2.07+

In a paravirtualized environment the hardware low level architectural pieces such as interrupt handling, page table handling, and accessing process control registers needs to be done differently.

This field allows the bootloader to inform the kernel we are in one one of those environments.

0x00000000	The default x86/PC environment
0x00000001	lguest
0x00000002	Xen
0x00000003	Moorestown MID
0x00000004	CE4100 TV Platform

Field name:	hardware_subarch_data
Type:	write (subarch-dependent)
Offset/size:	0x240/8
Protocol:	2.07+

A pointer to data that is specific to hardware subarch This field is currently unused for the default x86/PC environment, do not modify.

Field name:	payload_offset
Type:	read
Offset/size:	0x248/4
Protocol:	2.08+

If non-zero then this field contains the offset from the beginning of the protected-mode code to the payload.

The payload may be compressed. The format of both the compressed and uncompressed data should be determined using the standard magic numbers. The currently supported compression formats are gzip (magic numbers 1F 8B or 1F 9E), bzip2 (magic number 42 5A), LZMA (magic number 5D 00), XZ (magic number FD 37), LZ4 (magic number 02 21) and ZSTD (magic number 28 B5). The uncompressed payload is currently always ELF (magic number 7F 45 4C 46).

Field name:	payload_length
Type:	read
Offset/size:	0x24c/4
Protocol:	2.08+

The length of the payload.

Field name:	setup_data
Type:	write (special)
Offset/size:	0x250/8
Protocol:	2.09+

The 64-bit physical pointer to NULL terminated single linked list of struct setup_data. This is used to define a more extensible boot parameters passing mechanism. The definition of struct setup_data is as follow:

```
struct setup_data {
        u64 next;
        u32 type;
        u32 len;
        u8  data[0];
};
```

Where, the next is a 64-bit physical pointer to the next node of linked list, the next field of the last node is 0; the type is used to identify the contents of data; the len is the length of data field; the data holds the real payload.

This list may be modified at a number of points during the bootup process. Therefore, when modifying this list one should always make sure to consider the case where the linked list already contains entries.

The setup_data is a bit awkward to use for extremely large data objects, both because the setup_data header has to be adjacent to the data object and because it has a 32-bit length field. However, it is important that intermediate stages of the boot process have a way to identify which chunks of memory are occupied by kernel data.

Thus setup_indirect struct and SETUP_INDIRECT type were introduced in protocol 2.15:

```
struct setup_indirect {
  __u32 type;
  __u32 reserved;  /* Reserved, must be set to zero. */
  __u64 len;
  __u64 addr;
};
```

The type member is a SETUP_INDIRECT | SETUP_* type. However, it cannot be SETUP_INDIRECT itself since making the setup_indirect a tree structure could require a lot of stack space in something that needs to parse it and stack space can be limited in boot contexts.

Let's give an example how to point to SETUP_E820_EXT data using setup_indirect. In this case setup_data and setup_indirect will look like this:

```
struct setup_data {
  __u64 next = 0 or <addr_of_next_setup_data_struct>;
  __u32 type = SETUP_INDIRECT;
  __u32 len = sizeof(setup_indirect);
  __u8 data[sizeof(setup_indirect)] = struct setup_indirect {
    __u32 type = SETUP_INDIRECT | SETUP_E820_EXT;
    __u32 reserved = 0;
    __u64 len = <len_of_SETUP_E820_EXT_data>;
    __u64 addr = <addr_of_SETUP_E820_EXT_data>;
  }
}
```

> **Note**
>
> SETUP_INDIRECT | SETUP_NONE objects cannot be properly distinguished from SETUP_INDIRECT itself. So, this kind of objects cannot be provided by the bootloaders.

Field name:	pref_address
Type:	read (reloc)
Offset/size:	0x258/8
Protocol:	2.10+

This field, if nonzero, represents a preferred load address for the kernel. A relocating bootloader should attempt to load at this address if possible.

A non-relocatable kernel will unconditionally move itself and to run at this address.

Field name:	init_size
Type:	read
Offset/size:	0x260/4

This field indicates the amount of linear contiguous memory starting at the kernel runtime start address that the kernel needs before it is capable of examining its memory map. This is not the same thing as the total amount of memory the kernel needs to boot, but it can be used by a relocating boot loader to help select a safe load address for the kernel.

The kernel runtime start address is determined by the following algorithm:

```
if (relocatable_kernel)
runtime_start = align_up(load_address, kernel_alignment)
else
runtime_start = pref_address
```

Field name:	handover_offset
Type:	read
Offset/size:	0x264/4

This field is the offset from the beginning of the kernel image to the EFI handover protocol entry point. Boot loaders using the EFI handover protocol to boot the kernel should jump to this offset.

See EFI HANDOVER PROTOCOL below for more details.

Field name:	kernel_info_offset
Type:	read
Offset/size:	0x268/4
Protocol:	2.15+

This field is the offset from the beginning of the kernel image to the kernel_info. The kernel_info structure is embedded in the Linux image in the uncompressed protected mode region.

The kernel_info

The relationships between the headers are analogous to the various data sections:

> setup_header = .data boot_params/setup_data = .bss

What is missing from the above list? That's right:

> kernel_info = .rodata

We have been (ab)using .data for things that could go into .rodata or .bss for a long time, for lack of alternatives and -- especially early on -- inertia. Also, the BIOS stub is responsible for creating boot_params, so it isn't available to a BIOS-based loader (setup_data is, though).

setup_header is permanently limited to 144 bytes due to the reach of the 2-byte jump field, which doubles as a length field for the structure, combined with the size of the "hole" in struct boot_params that a protected-mode loader or the BIOS stub has to copy it into. It is currently 119 bytes long, which leaves us with 25 very precious bytes. This isn't something that can be fixed without revising the boot protocol entirely, breaking backwards compatibility.

boot_params proper is limited to 4096 bytes, but can be arbitrarily extended by adding setup_data entries. It cannot be used to communicate properties of the kernel image, because it is .bss and has no image-provided content.

kernel_info solves this by providing an extensible place for information about the kernel image. It is readonly, because the kernel cannot rely on a bootloader copying its contents anywhere, but that is OK; if it becomes necessary it can still contain data items that an enabled bootloader would be expected to copy into a setup_data chunk.

All kernel_info data should be part of this structure. Fixed size data have to be put before kernel_info_var_len_data label. Variable size data have to be put after kernel_info_var_len_data label. Each chunk of variable size data has to be prefixed with header/magic and its size, e.g.:

```
kernel_info:
        .ascii  "LToP"          /* Header, Linux top (structure). */
        .long   kernel_info_var_len_data - kernel_info
        .long   kernel_info_end - kernel_info
        .long   0x01234567      /* Some fixed size data for the bootloaders. */
kernel_info_var_len_data:
example_struct:                 /* Some variable size data for the bootloaders. */
        .ascii  "0123"          /* Header/Magic. */
        .long   example_struct_end - example_struct
        .ascii  "Struct"
        .long   0x89012345
example_struct_end:
example_strings:                /* Some variable size data for the bootloaders. */
        .ascii  "ABCD"          /* Header/Magic. */
        .long   example_strings_end - example_strings
        .asciz  "String_0"
        .asciz  "String_1"
example_strings_end:
kernel_info_end:
```

This way the kernel_info is self-contained blob.

> **Note**
>
> Each variable size data header/magic can be any 4-character string, without 0 at the end of the string, which does not collide with existing variable length data headers/magics.

Details of the kernel_info Fields

Field name:	header
Offset/size:	0x0000/4

> Contains the magic number "LToP" (0x506f544c).

Field name:	size
Offset/size:	0x0004/4

> This field contains the size of the kernel_info including kernel_info.header. It does not count kernel_info.kernel_info_var_len_data size. This field should be used by the bootloaders to detect supported fixed size fields in the kernel_info and beginning of kernel_info.kernel_info_var_len_data.

Field name:	size_total

Offset/size:	0x0008/4

This field contains the size of the kernel_info including kernel_info.header and kernel_info.kernel_info_var_len_data.

Field name:	setup_type_max
Offset/size:	0x000c/4

This field contains maximal allowed type for setup_data and setup_indirect structs.

The Image Checksum

From boot protocol version 2.08 onwards the CRC-32 is calculated over the entire file using the characteristic polynomial 0x04C11DB7 and an initial remainder of 0xffffffff. The checksum is appended to the file; therefore the CRC of the file up to the limit specified in the syssize field of the header is always 0.

The Kernel Command Line

The kernel command line has become an important way for the boot loader to communicate with the kernel. Some of its options are also relevant to the boot loader itself, see "special command line options" below.

The kernel command line is a null-terminated string. The maximum length can be retrieved from the field cmdline_size. Before protocol version 2.06, the maximum was 255 characters. A string that is too long will be automatically truncated by the kernel.

If the boot protocol version is 2.02 or later, the address of the kernel command line is given by the header field cmd_line_ptr (see above.) This address can be anywhere between the end of the setup heap and 0xA0000.

If the protocol version is *not* 2.02 or higher, the kernel command line is entered using the following protocol:

- At offset 0x0020 (word), "cmd_line_magic", enter the magic number 0xA33F.
- At offset 0x0022 (word), "cmd_line_offset", enter the offset of the kernel command line (relative to the start of the real-mode kernel).
- The kernel command line *must* be within the memory region covered by setup_move_size, so you may need to adjust this field.

Memory Layout of The Real-Mode Code

The real-mode code requires a stack/heap to be set up, as well as memory allocated for the kernel command line. This needs to be done in the real-mode accessible memory in bottom megabyte.

It should be noted that modern machines often have a sizable Extended BIOS Data Area (EBDA). As a result, it is advisable to use as little of the low megabyte as possible.

Unfortunately, under the following circumstances the 0x90000 memory segment has to be used:

- When loading a zImage kernel ((loadflags & 0x01) == 0).
- When loading a 2.01 or earlier boot protocol kernel.

> **Note**
>
> For the 2.00 and 2.01 boot protocols, the real-mode code can be loaded at another address, but it is internally relocated to 0x90000. For the "old" protocol, the real-mode code must be loaded at 0x90000.

When loading at 0x90000, avoid using memory above 0x9a000.

For boot protocol 2.02 or higher, the command line does not have to be located in the same 64K segment as the real-mode setup code; it is thus permitted to give the stack/heap the full 64K segment and locate the command line above it.

The kernel command line should not be located below the real-mode code, nor should it be located in high memory.

Sample Boot Configuartion

As a sample configuration, assume the following layout of the real mode segment.

When loading below 0x90000, use the entire segment:

0x0000-0x7fff	Real mode kernel
0x8000-0xdfff	Stack and heap
0xe000-0xffff	Kernel command line

When loading at 0x90000 OR the protocol version is 2.01 or earlier:

0x0000-0x7fff	Real mode kernel
0x8000-0x97ff	Stack and heap
0x9800-0x9fff	Kernel command line

Such a boot loader should enter the following fields in the header:

```
unsigned long base_ptr; /* base address for real-mode segment */

if ( setup_sects == 0 ) {
        setup_sects = 4;
}

if ( protocol >= 0x0200 ) {
        type_of_loader = <type code>;
        if ( loading_initrd ) {
                ramdisk_image = <initrd_address>;
                ramdisk_size = <initrd_size>;
        }

        if ( protocol >= 0x0202 && loadflags & 0x01 )
                heap_end = 0xe000;
        else
                heap_end = 0x9800;

        if ( protocol >= 0x0201 ) {
                heap_end_ptr = heap_end - 0x200;
                loadflags |= 0x80; /* CAN_USE_HEAP */
        }

        if ( protocol >= 0x0202 ) {
                cmd_line_ptr = base_ptr + heap_end;
                strcpy(cmd_line_ptr, cmdline);
        } else {
                cmd_line_magic  = 0xA33F;
                cmd_line_offset = heap_end;
                setup_move_size = heap_end + strlen(cmdline)+1;
                strcpy(base_ptr+cmd_line_offset, cmdline);
        }
} else {
        /* Very old kernel */

        heap_end = 0x9800;

        cmd_line_magic  = 0xA33F;
        cmd_line_offset = heap_end;

        /* A very old kernel MUST have its real-mode code
           loaded at 0x90000 */

        if ( base_ptr != 0x90000 ) {
                /* Copy the real-mode kernel */
                memcpy(0x90000, base_ptr, (setup_sects+1)*512);
                base_ptr = 0x90000;              /* Relocated */
        }

        strcpy(0x90000+cmd_line_offset, cmdline);

        /* It is recommended to clear memory up to the 32K mark */
        memset(0x90000 + (setup_sects+1)*512, 0,
               (64-(setup_sects+1))*512);
}
```

Loading The Rest of The Kernel

The 32-bit (non-real-mode) kernel starts at offset (setup_sects+1)*512 in the kernel file (again, if setup_sects == 0 the real value is 4.) It should be loaded at address 0x10000 for Image/zImage kernels and 0x100000 for bzImage kernels.

The kernel is a bzImage kernel if the protocol >= 2.00 and the 0x01 bit (LOAD_HIGH) in the loadflags field is set:

```
is_bzImage = (protocol >= 0x0200) && (loadflags & 0x01);
load_address = is_bzImage ? 0x100000 : 0x10000;
```

Note that Image/zImage kernels can be up to 512K in size, and thus use the entire 0x10000-0x90000 range of memory. This means it is pretty much a requirement for these kernels to load the real-mode part at 0x90000. bzImage kernels allow much more flexibility.

Special Command Line Options

If the command line provided by the boot loader is entered by the user, the user may expect the following command line options to work. They should normally not be deleted from the kernel command line even though not all of them are actually meaningful to the kernel. Boot loader authors who need additional command line options for the boot loader itself should get them registered in Documentation/admin-guide/kernel-parameters.rst to make sure they will not conflict with actual kernel options now or in the future.

vga=<mode>
: <mode> here is either an integer (in C notation, either decimal, octal, or hexadecimal) or one of the strings "normal" (meaning 0xFFFF), "ext" (meaning 0xFFFE) or "ask" (meaning 0xFFFD). This value should be entered into the vid_mode field, as it is used by the kernel before the command line is parsed.

mem=<size>
: <size> is an integer in C notation optionally followed by (case insensitive) K, M, G, T, P or E (meaning << 10, << 20, << 30, << 40, << 50 or << 60). This specifies the end of memory to the kernel. This affects the possible placement of an initrd, since an initrd should be placed near end of memory. Note that this is an option to *both* the kernel and the bootloader!

initrd=<file>
: An initrd should be loaded. The meaning of <file> is obviously bootloader-dependent, and some boot loaders (e.g. LILO) do not have such a command.

In addition, some boot loaders add the following options to the user-specified command line:

BOOT_IMAGE=<file>
: The boot image which was loaded. Again, the meaning of <file> is obviously bootloader-dependent.

auto
: The kernel was booted without explicit user intervention.

If these options are added by the boot loader, it is highly recommended that they are located *first*, before the user-specified or configuration-specified command line. Otherwise, "init=/bin/sh" gets confused by the "auto" option.

Running the Kernel

The kernel is started by jumping to the kernel entry point, which is located at *segment* offset 0x20 from the start of the real mode kernel. This means that if you loaded your real-mode kernel code at 0x90000, the kernel entry point is 9020:0000.

At entry, ds = es = ss should point to the start of the real-mode kernel code (0x9000 if the code is loaded at 0x90000), sp should be set up properly, normally pointing to the top of the heap, and interrupts should be disabled. Furthermore, to guard against bugs in the kernel, it is recommended that the boot loader sets fs = gs = ds = es = ss.

In our example from above, we would do:

```
/* Note: in the case of the "old" kernel protocol, base_ptr must
   be == 0x90000 at this point; see the previous sample code */

seg = base_ptr >> 4;

cli();  /* Enter with interrupts disabled! */

/* Set up the real-mode kernel stack */
_SS = seg;
_SP = heap_end;

_DS = _ES = _FS = _GS = seg;
jmp_far(seg+0x20, 0);   /* Run the kernel */
```

If your boot sector accesses a floppy drive, it is recommended to switch off the floppy motor before running the kernel, since the kernel boot leaves interrupts off and thus the motor will not be switched off, especially if the loaded kernel has the floppy driver as a demand-loaded module!

Advanced Boot Loader Hooks

If the boot loader runs in a particularly hostile environment (such as LOADLIN, which runs under DOS) it may be impossible to follow the standard memory location requirements. Such a boot loader may use the following hooks that, if set, are invoked by the kernel at the appropriate time. The use of these hooks should probably be considered an absolutely last resort!

IMPORTANT: All the hooks are required to preserve %esp, %ebp, %esi and %edi across invocation.

realmode_swtch:
: A 16-bit real mode far subroutine invoked immediately before entering protected mode. The default routine disables NMI, so your routine should probably do so, too.

code32_start:
: A 32-bit flat-mode routine *jumped* to immediately after the transition to protected mode, but before the kernel is

uncompressed. No segments, except CS, are guaranteed to be set up (current kernels do, but older ones do not); you should set them up to BOOT_DS (0x18) yourself.

After completing your hook, you should jump to the address that was in this field before your boot loader overwrote it (relocated, if appropriate.)

32-bit Boot Protocol

For machine with some new BIOS other than legacy BIOS, such as EFI, LinuxBIOS, etc, and kexec, the 16-bit real mode setup code in kernel based on legacy BIOS can not be used, so a 32-bit boot protocol needs to be defined.

In 32-bit boot protocol, the first step in loading a Linux kernel should be to setup the boot parameters (struct boot_params, traditionally known as "zero page"). The memory for struct boot_params should be allocated and initialized to all zero. Then the setup header from offset 0x01f1 of kernel image on should be loaded into struct boot_params and examined. The end of setup header can be calculated as follow:

```
0x0202 + byte value at offset 0x0201
```

In addition to read/modify/write the setup header of the struct boot_params as that of 16-bit boot protocol, the boot loader should also fill the additional fields of the struct boot_params as described in chapter Documentation/x86/zero-page.rst.

After setting up the struct boot_params, the boot loader can load the 32/64-bit kernel in the same way as that of 16-bit boot protocol.

In 32-bit boot protocol, the kernel is started by jumping to the 32-bit kernel entry point, which is the start address of loaded 32/64-bit kernel.

At entry, the CPU must be in 32-bit protected mode with paging disabled; a GDT must be loaded with the descriptors for selectors __BOOT_CS(0x10) and __BOOT_DS(0x18); both descriptors must be 4G flat segment; __BOOT_CS must have execute/read permission, and __BOOT_DS must have read/write permission; CS must be __BOOT_CS and DS, ES, SS must be __BOOT_DS; interrupt must be disabled; %esi must hold the base address of the struct boot_params; %ebp, %edi and %ebx must be zero.

64-bit Boot Protocol

For machine with 64bit cpus and 64bit kernel, we could use 64bit bootloader and we need a 64-bit boot protocol.

In 64-bit boot protocol, the first step in loading a Linux kernel should be to setup the boot parameters (struct boot_params, traditionally known as "zero page"). The memory for struct boot_params could be allocated anywhere (even above 4G) and initialized to all zero. Then, the setup header at offset 0x01f1 of kernel image on should be loaded into struct boot_params and examined. The end of setup header can be calculated as follows:

```
0x0202 + byte value at offset 0x0201
```

In addition to read/modify/write the setup header of the struct boot_params as that of 16-bit boot protocol, the boot loader should also fill the additional fields of the struct boot_params as described in chapter Documentation/x86/zero-page.rst.

After setting up the struct boot_params, the boot loader can load 64-bit kernel in the same way as that of 16-bit boot protocol, but kernel could be loaded above 4G.

In 64-bit boot protocol, the kernel is started by jumping to the 64-bit kernel entry point, which is the start address of loaded 64-bit kernel plus 0x200.

At entry, the CPU must be in 64-bit mode with paging enabled. The range with setup_header.init_size from start address of loaded kernel and zero page and command line buffer get ident mapping; a GDT must be loaded with the descriptors for selectors __BOOT_CS(0x10) and __BOOT_DS(0x18); both descriptors must be 4G flat segment; __BOOT_CS must have execute/read permission, and __BOOT_DS must have read/write permission; CS must be __BOOT_CS and DS, ES, SS must be __BOOT_DS; interrupt must be disabled; %rsi must hold the base address of the struct boot_params.

EFI Handover Protocol (deprecated)

This protocol allows boot loaders to defer initialisation to the EFI boot stub. The boot loader is required to load the kernel/initrd(s) from the boot media and jump to the EFI handover protocol entry point which is hdr->handover_offset bytes from the beginning of startup_{32,64}.

The boot loader MUST respect the kernel's PE/COFF metadata when it comes to section alignment, the memory footprint of the executable image beyond the size of the file itself, and any other aspect of the PE/COFF header that may affect correct operation of the image as a PE/COFF binary in the execution context provided by the EFI firmware.

The function prototype for the handover entry point looks like this:

```
efi_main(void *handle, efi_system_table_t *table, struct boot_params *bp)
```

'handle' is the EFI image handle passed to the boot loader by the EFI firmware, 'table' is the EFI system table - these are the first two arguments of the "handoff state" as described in section 2.3 of the UEFI specification. 'bp' is the boot loader-allocated boot params.

The boot loader *must* fill out the following fields in bp:

```
- hdr.cmd_line_ptr
- hdr.ramdisk_image (if applicable)
- hdr.ramdisk_size  (if applicable)
```

All other fields should be zero.

NOTE: The EFI Handover Protocol is deprecated in favour of the ordinary PE/COFF
: entry point, combined with the LINUX_EFI_INITRD_MEDIA_GUID based initrd loading protocol (refer to [0] for an example of the bootloader side of this), which removes the need for any knowledge on the part of the EFI bootloader regarding the internal representation of boot_params or any requirements/limitations regarding the placement of the command line and ramdisk in memory, or the placement of the kernel image itself.

[0] https://github.com/u-boot/u-boot/commit/ec80b4735a593961fe701cc3a5d717d4739b0fd0