kaneton K0
Bootstrap

Matthieu Bucchianeri and Renaud Voltz

January 25, 2007

Delivery date:	Sunday, 28th
	11:42 pm
Assistants :	Matthieu Bucchianeri - chichelover@epita.fr
	Renaud Voltz - voltz_r@epita.fr
Dedicated googlegroup:	kaneton-students
Programming languages:	Assembly, C
Architecture:	Intel 32-bit
Students per group:	1
Tarball:	k0-login_l.tar.bz2
Permissions:	we don't care

Before starting

- You will use NASM to assemble your code. NASM is present on the PIE. To test, you will use QEMU, which can be run using the script `~bucchi_m/qemu/run.sh` and adding the correct parameters (`-fda` for a floppy image).
- A bootsector is not an ELF binary, but a flat object (without any headers). Obtaining flat binaries with NASM is done using the `-f` flag (refer to the manual).
- The bootsector is loaded at address 0x7c00, you must find a way to tell NASM that the code will be loaded there.
- Remember that the microprocessor starts in 16-bit mode, so you must find a directive to tell NASM to assemble 16-bit code. Then, when you switch to 32-bit mode, find another directive to tell NASM the new assembly mode.
- Your bootsector must end with a signature (0xAA55). This means that blank characters must be inserted until byte 510, and then these two bytes must be present.

```
times 510-($-$$) db 0   ; fill the rest of the sector with zeros
dw    0xAA55            ; add the bootloader signature to the end
```

Implementation

Exercise 1: string display

- **Source tree**
  directory: /k0-login_l/ex1/
  filename: ex1.s

- **Subject**
  Print a string at the (20, 10) coordinates. You must use the BIOS calls.

- **Steps**

  1. `print_char`
     Print a character at the current cursor position, and update the cursor position.
  2. `print_string`
     Print the string pointed by *%si* register at the cursor position and update the cursor position.
  3. `cursor_set`
     Set the cursor position.

Exercise 2: libc

- **Source tree**
  directory: /k0-login_l/ex2/
  filename: ex2.s

- **Subject**
  Write a program wich dumps the registers values.

- **Steps**
  1. `malloc`
     Very stupid malloc:
     – Declare the heap.
     – Declare a break value at the begining of the heap.
     – malloc returns the break value in *%ax* and then increments it.
  2. `itoa`
     Basic itoa (hey, why not using it to test your malloc ?!).
  3. `itoa_hex`
     Hexadecimal itoa.
     16-bit hexadecimal outputs must match the following format: `0x00a2`.

- **Output**

```
ax = 0x1234 = 4660
bx = 0x0000 = 0
cx = 0xabcd = 43981
dx = 0x00ff = 255

bp = 0x1000 = 4096
sp = 0x0ff8 = 4088
ip = 0x7c00 = 31744
```

Exercise 3: keyboard inputs

- **Source tree**
  directory: /k0-login_l/ex3/
  filename: ex3.s

- **Subject**
  Write a prompt which gets a string from the keyboard and wich displays it when you press `ENTER`.

  You must display alpha-numeric characters and punctuation.
  Do not implement key combinations (using modifiers like `SHIFT`, `ALT`, `CTRL`, ...).
  Pressing `ENTER` must result in a newline.

- **Steps**
  1. `kbd_get_scancode`
     Get the next scancode from the keyboard buffer.
  2. `scancode_to_ascii`
     Convert a scancode to an ASCII character.

- **Output**

```
Enter your name: Renaud
Hello Renaud !
```

Exercise 4: floppy drive

- **Source tree**
  directory: /k0-login_l/ex4/
  filename: ex4.s

- **Subject**
  Write a program which loads the bootsector of a floppy disk and which checks wether it does contain a bootloader.

  (The bootsector contains a bootloader if it is ended by the 0xAA55 magic.)

  Your program must print the magic value as shown in the given output.

- **Steps**
  1. `floppy_read_sector`
     Read n sectors from the floppy drive (A:).

- **Output**

```
Loading floppy bootsector ... OK
magic found: 0xaa55

Loading floppy bootsector ... OK
ERROR: bad magic: 0xt824
```

Exercise 5: operating modes switching

- **Source tree**
  directory: /k0-login_l/ex5/
  filename: ex5.s

- **Subject**
  Write a program which turns the microprocessor into protected mode.
  Once in protected mode, your program must clear the whole screen and print a message indicating that protected mode is enabled.

- **Steps**
  1. `pmode_enable`
     Switch from real mode to protected mode.
  2. `print_string_fb`
     Print a string (in protected mode). See appendix *VGA text framebuffer*.
  3. `memset`
     Basic memset function.
  4. `memcpy`
     Basic memcpy function.

Exercise 6: ELF loader

- **Source tree**

directory:	/k0-login_l/ex6/
filenames:	ex6.lds
	ex6.s

- **Subject**

  You will now write a complete bootloader. This one will load an ELF file from the disk (located at the sector just after the bootsector) and then relocate it in memory at the right place, before jumping to it.

  The ELF file **must** contain two segments, one with the code (which must be loaded at 1 Mb) and the other with the data (loaded at 2 Mb). Example:

```
42sh> readelf -l bootloader

Elf file type is EXEC (Executable file)
Entry point 0x1000cc
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00100000 0x00100000 0x000df 0x000df R E 0x1000
  LOAD           0x002000 0x00200000 0x00200000 0x00022 0x00028 RW  0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .text
   01     .data .rodata .bss
```

  Your bootloader **must** reset the BSS memory. To keep the thing simple, put the *.bss* section at the end of the second segment. Its size can be determined by computing the difference between MemSize and FileSiz.

  Your tarball **must** include the ld-script used to create such ELF binaries.

  Before starting, you should watch the ELF documentation, especially about the ELF header and the Program Header. The task is not as harder as it looks like (our code dealing with ELF files is about 30 instructions long).

- **Steps**

  1. Get the `floppy_read_sector` function and call it correctly to store the binary in a temporary location.
  2. Reuse your `pmode_enable` function.
  3. Read the loaded binary to find its segments and extract their load addresses, size and source location into the file.
  4. Use your `memcpy` to relocate the code and the initialized data, use your `memset` to reset the BSS section.
  5. Find the binary entry point (2 instructions !) and jump on it after having initialized a correct stack).

  You must write your own test ELF binary in C, which must be able to write text on the screen. See appendix *VGA text framebuffer*.

1 Appendixes

- BIOS services:
  - int 10: video services
  - int 13: disk services
  - int 16: keyboard services
- VGA Text Framebuffer
- Executable & Linkable Format (ELF) documentation
- Intel IA-32 mode switching procedures

INT 10 - Video BIOS Services

For more information, see the following topics:


```
Warning: Some BIOS implementations have a bug that causes register
BP to be destroyed.   It is advisable to save BP before a call to
Video BIOS routines on these systems.

- registers CS, DS, ES, SS, BX, CX, DX are preserved unless
  explicitly changed
- see  INT 1F  INT 1D  INT 29  INT 21,2  INT 21,6  INT 21,9
```

INT 10,2 - Set Cursor Position

```
AH = 02
BH = page number (0 for graphics modes)
DH = row
DL = column


returns nothing


- positions relative to 0,0 origin
- 80x25 uses coordinates 0,0 to 24,79;  40x25 uses 0,0 to 24,39
- the 6845 can also be used to perform this function
- setting the data in the BIOS Data Area at location 40:50 does not
  take immediate effect and is not recommended
- see   VIDEO PAGES   6845   BDA
```

INT 10,3 - Read Cursor Position and Size

```
AH = 03
BH = video page


on return:
CH = cursor starting scan line (low order 5 bits)
CL = cursor ending scan line (low order 5 bits)
DH = row
DL = column


- returns data from BIOS DATA AREA locations 40:50, 40:60 and 40:61
- the 6845 can also be used to read the cursor position
- the return data can be circumvented by direct port I/O to the 6845
  CRT Controller since this function returns the data found in the
  BIOS Data Area without actually checking the controller
```

INT 10,E - Write Text in Teletype Mode

```
AH = 0E
AL = ASCII character to write
BH = page number (text modes)
BL = foreground pixel color (graphics modes)


returns nothing


- cursor advances after write
- characters BEL (7), BS (8), LF (A), and CR (D) are
  treated as control codes
- for some older BIOS (10/19/81), the BH register must point
  to the currently displayed page
- on CGA adapters this function can disable the video signal while
  performing the output which causes flitter.
```

INT 13 - Diskette BIOS Services

For more information see the following topics:

Most disk BIOS calls use the following parameter scheme:

```
    AH = function request number
    AL = number of sectors  (1-128 dec.)
    CH = cylinder number  (0-1023 dec.)
    CL = sector number  (1-17 dec.)
    DH = head number  (0-15 dec.)
    DL = drive number (0=A:, 1=2nd floppy, 80h=drive 0, 81h=drive 1)
    DL = drive number (0=A:, 1=2nd floppy, 80h=C:, 81h=D:)
         Note that some programming references use (0-3) as the
         drive number which represents diskettes only.
    ES:BX = address of user buffer
```

and return with:

```
CF = 0 if successful
   = 1 if error
AH = status of operation  (see INT 13,STATUS)
```

```
- INT 13 diskette read functions should be retried at least 3
  times to assure the disk motor has time to spin up to speed

- registers DS, BX, CX and DX are preserved
- see  INT 13,STATUS
```

INT 13,0 - Reset Disk System

```
AH = 00
DL = drive number (0=A:, 1=2nd floppy, 80h=drive 0, 81h=drive 1)


on return:
AH = disk operation status  (see INT 13,STATUS)
CF = 0 if successful
   = 1 if error


- clears reset flag in controller and pulls heads to track 0
- setting the controller reset flag causes the disk to recalibrate
  on the next disk operation
- if bit 7 is set, the diskette drive indicated by the lower 7 bits
  will reset then the hard disk will follow; return code in AH is
  for the drive requested
```

INT 13,1 - Disk Status

```
AH = 01

on return:
AL = status:
```

Status in AL

```
00  no error
01  bad command passed to driver
02  address mark not found or bad sector
03  diskette write protect error
04  sector not found
05  fixed disk reset failed
06  diskette changed or removed
07  bad fixed disk parameter table
08  DMA overrun
09  DMA access across 64k boundary
0A  bad fixed disk sector flag
0B  bad fixed disk cylinder
0C  unsupported track/invalid media
0D  invalid number of sectors on fixed disk format
0E  fixed disk controlled data address mark detected
0F  fixed disk DMA arbitration level out of range
10  ECC/CRC error on disk read
11  recoverable fixed disk data error, data fixed by ECC
20  controller error (NEC for floppies)
40  seek failure
80  time out, drive not ready
AA  fixed disk drive not ready
BB  fixed disk undefined error
CC  fixed disk write fault on selected drive
E0  fixed disk status error/Error reg = 0
FF  sense operation failed
```

```
- codes represent controller status after last disk operation
- returns the status byte located at 40:41 in the BIOS Data Area
```

INT 13,2 - Read Disk Sectors

```
AH = 02
AL = number of sectors to read  (1-128 dec.)
CH = track/cylinder number  (0-1023 dec., see below)
CL = sector number  (1-17 dec.)
DH = head number  (0-15 dec.)
DL = drive number (0=A:, 1=2nd floppy, 80h=drive 0, 81h=drive 1)
ES:BX = pointer to buffer


on return:
AH = status  (see INT 13,STATUS)
AL = number of sectors read
CF = 0 if successful
   = 1 if error


- BIOS disk reads should be retried at least three times and the
  controller should be reset upon error detection
- be sure ES:BX does not cross a 64K segment boundary or a
  DMA boundary error will occur
- many programming references list only floppy disk register values
- only the disk number is checked for validity
- the parameters in CX change depending on the number of cylinders;
  the track/cylinder number is a 10 bit value taken from the 2 high
  order bits of CL and the 8 bits in CH (low order 8 bits of track):

  |F|E|D|C|B|A|9|8|7|6|5-0|  CX
   | | | | | | | | | |  `-----  sector number
   | | | | | | | | `---------  high order 2 bits of track/cylinder
   `------------------------  low order 8 bits of track/cyl number

- see   INT 13,A
```

INT 16 - Keyboard BIOS Services

For more information, see the following topics:

```
INT 16,0   Wait for keystroke and read
INT 16,1   Get keystroke status
INT 16,2   Get shift status


- with IBM BIOS's, INT 16 functions do not restore the flags to
  the pre-interrupt state to allow returning of information via
  the flags register
- all registers are preserved except AX and FLAGS
- see   SCAN CODES
```

INT 16,0 - Wait for Keypress and Read Character

```
AH = 00


on return:
AH = keyboard scan code
AL = ASCII character or zero if special function key


- halts program until key with a scancode is pressed
- see  SCAN CODES
```

INT 16,1 - Get Keyboard Status

```
AH = 01


on return:
ZF = 0 if a key pressed (even Ctrl-Break)
AX = 0 if no scan code is available
AH = scan code
AL = ASCII character or zero if special function key


- data code is not removed from buffer
- Ctrl-Break places a zero word in the keyboard buffer but does
  register a keypress.
```

INT 16 - Keyboard Scan Codes

Key	Normal	Shifted	w/Ctrl	w/Alt
A	1E61	1E41	1E01	1E00
B	3062	3042	3002	3000
C	2E63	2E42	2E03	2E00
D	2064	2044	2004	2000
E	1265	1245	1205	1200
F	2166	2146	2106	2100
G	2267	2247	2207	2200
H	2368	2348	2308	2300
I	1769	1749	1709	1700
J	246A	244A	240A	2400
K	256B	254B	250B	2500
L	266C	264C	260C	2600
M	326D	324D	320D	3200
N	316E	314E	310E	3100
O	186F	184F	180F	1800
P	1970	1950	1910	1900
Q	1071	1051	1011	1000
R	1372	1352	1312	1300
S	1F73	1F53	1F13	1F00
T	1474	1454	1414	1400
U	1675	1655	1615	1600
V	2F76	2F56	2F16	2F00
W	1177	1157	1117	1100
X	2D78	2D58	2D18	2D00
Y	1579	1559	1519	1500
Z	2C7A	2C5A	2C1A	2C00

Key	Normal	Shifted	w/Ctrl	w/Alt
1	0231	0221		7800
2	0332	0340	0300	7900
3	0433	0423		7A00
4	0534	0524		7B00
5	0635	0625		7C00
6	0736	075E	071E	7D00
7	0837	0826		7E00
8	0938	092A		7F00
9	0A39	0A28		8000
0	0B30	0B29		8100

Key	Normal	Shifted	w/Ctrl	w/Alt
-	0C2D	0C5F	0C1F	8200
=	0D3D	0D2B		8300
[	1A5B	1A7B	1A1B	1A00
]	1B5D	1B7D	1B1D	1B00
;	273B	273A		2700
'	2827	2822		
`	2960	297E		
\	2B5C	2B7C	2B1C	2600 (same as Alt L)
,	332C	333C		
.	342E	343E		
/	352F	353F		

Key	Normal	Shifted	w/Ctrl	w/Alt
F1	3B00	5400	5E00	6800
F2	3C00	5500	5F00	6900
F3	3D00	5600	6000	6A00
F4	3E00	5700	6100	6B00
F5	3F00	5800	6200	6C00
F6	4000	5900	6300	6D00
F7	4100	5A00	6400	6E00
F8	4200	5B00	6500	6F00
F9	4300	5C00	6600	7000
F10	4400	5D00	6700	7100
F11	8500	8700	8900	8B00
F12	8600	8800	8A00	8C00

Key	Normal	Shifted	w/Ctrl	w/Alt

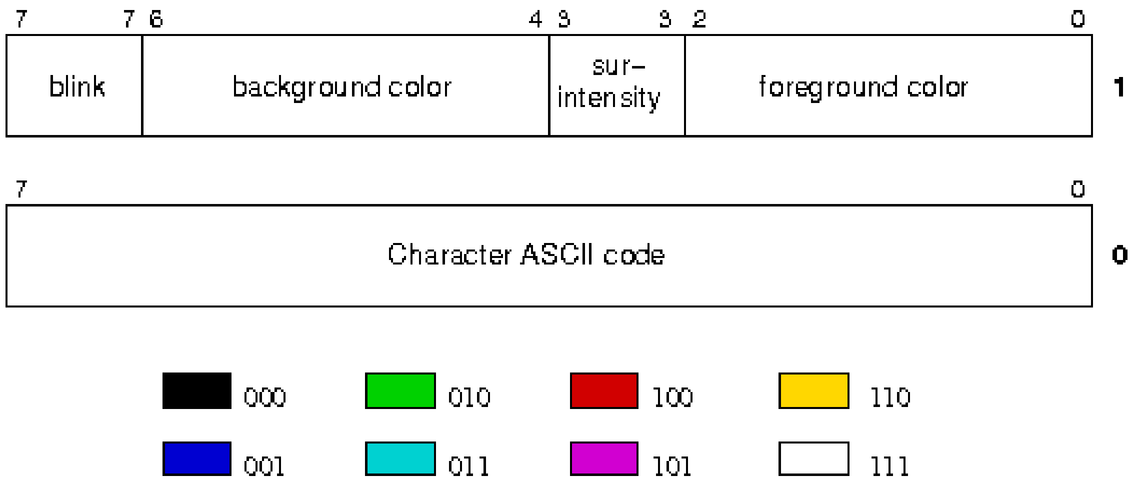
BackSpace	0E08	0E08	0E7F	0E00
Del	5300	532E	9300	A300
Down Arrow	5000	5032	9100	A000
End	4F00	4F31	7500	9F00
Enter	1C0D	1C0D	1C0A	A600
Esc	011B	011B	011B	0100
Home	4700	4737	7700	9700
Ins	5200	5230	9200	A200
Keypad 5		4C35	8F00	
Keypad *	372A		9600	3700
Keypad -	4A2D	4A2D	8E00	4A00
Keypad +	4E2B	4E2B		4E00
Keypad /	352F	352F	9500	A400
Left Arrow	4B00	4B34	7300	9B00
PgDn	5100	5133	7600	A100
PgUp	4900	4939	8400	9900
PrtSc			7200	
Right Arrow	4D00	4D36	7400	9D00
SpaceBar	3920	3920	3920	3920
Tab	0F09	0F00	9400	A500
Up Arrow	4800	4838	8D00	9800

- Some key combinations are not available on all systems. The PS/2 includes many that aren’t available on the PC, XT and AT.
- To retrieve the character from a scan code logical AND the word with 0x00FF.
- see INT 16 MAKE CODES

VGA Text Framebuffer

Displaying text on the screen can be done using the BIOS services (int 10). But once in protected mode, the BIOS is not available anymore. So we must find an alternate way to drive the console.

At startup, a VGA card is initialized in color-text mode, with the 80x25 resolution. Each cell is represented by the following couple of bytes in memory:



The console *framebuffer* is a memory area (formatted with cells as described above) which maps the screen.

Write 0x4b at the address 0xb8000 and 0x96 at 0xb8001 will print the character 'K' blinking in yellow on a blue background, at the top-left corner of the screen. The second character is located at 0xb8002.

The ELF File Format

Figure: Linking and Execution Views: This figure illustrates the format of an ELF object file.

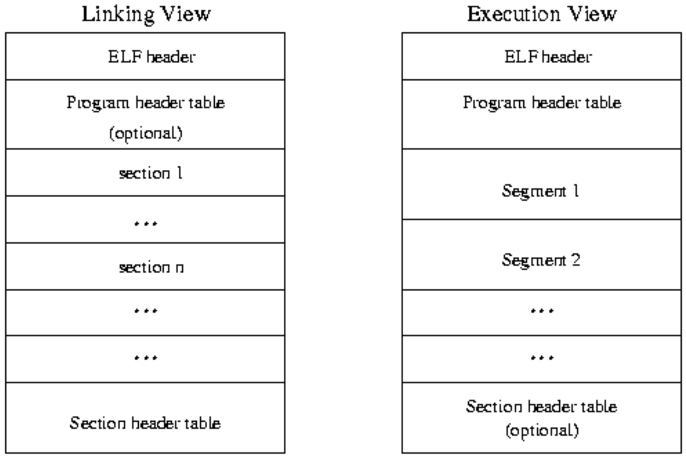


Figure: The ELF Header

```
#define EI'NIDENT    16

typedef struct –
    unsigned char    e'ident[EI'NIDENT];    //  file ID, interpretation
    Elf32'Half       e'type;                //  object file type
    Elf32'Half       e'machine;             //  target architecture
    Elf32'Word       e'version;             //  ELF version
    Elf32'Addr       e'entry;               //  starting virtual address
    Elf32'Off        e'phoff;               //  file offset to program hdr
    Elf32'Off        e'shoff;               //  file offset to section hdr
    Elf32'Word       e'flags;               //  processor-specific flags
    Elf32'Half       e'ehsize;              //  the ELF header's size
    Elf32'Half       e'phentsize;           //  program hdr entry size
    Elf32'Half       e'phnum;               //  program hdr entry number
    Elf32'Half       e'shentsize;           //  section hdr entry size
    Elf32'Half       e'shnum;               //  section hdr entry number
    Elf32'Half       e'shstrndx;            //  section hdr index for strings
˝ Elf32'Ehdr;
```

There are two views for each of the three file types described in the previous section. These views support both the linking and execution of a program. The two views are summarized in Figure 2.5 where the view on the left of the figure is the link view and the view on the right of the figure is the execution view. The link view of the ELF object file is partitioned by sections and the execution view of the ELF object file is partitioned by segments. Thus, the programmer interested in obtaining section information about the program items such as symbol tables, relocation, specific executable code or dynamic linking information will use the link view; the programmer interested in obtaining segment information such as the location of the `text` segment or `data` segment will use the execution view. The ELF access library, libelf, provides a programmer with tools to extract and manipulate ELF object file contents for either view. The ELF header describes the layout of the rest of the object file. It provides information on where and how to access the other sections. The Section Header Table gives the location and description of the sections and is mostly used in linking. The Program Header Table provides the location and description of segments and is mostly used in creating a programs' process image. Both sections and segments hold the majority of data in an object file including: instructions, data, symbol table, relocation information, and dynamic linking information.

The ELF Header

The ELF Header is the only section that has a fixed position in the object file. It is always the first section of the file. The other sections are not guaranteed to be in any order or to even be present. The ELF Header describes the type of the object file (relocatable, executable, shared, core), its target architecture, and the version of ELF it is using. The location of the Program Header table, Section Header table, and String table along with associated number and size of entries for each table are also given. Lastly, the ELF Header contains the location of the first executable instruction. The specific fields along with their size requirements that are present in the ELF header are shown in Figure 2.6.

Figure: The Program Header

```
typedef struct –
    Elf32'Word      p'type;     //  type of the segment
    Elf32'Off       p'offset;   //  file offset to segment
    Elf32'Addr      p'vaddr;    //  virtual address of first byte
    Elf32'Addr      p'paddr;    //  segments' physical address, i
    Elf32'Word      p'filesz;   //  size of file image of segment
    Elf32'Word      p'memsz;    //  size of memory image of se
    Elf32'Word      p'flags;    //  segment-specific flags
    Elf32'Word      p'align;    //  alignment requirements
˝ Elf32'Phdr;
```

The Program Header Table

Program headers are only important in executable and shared object files. The program header table is an array of entries where each entry is a structure describing a segment in the object file or other information needed to create an executable process image. The size of an entry in the table and the number of entries in the table are given in the ELF header (See Figure 2.6). Each entry in the program header table (see Figure 2.7) contains the type, file offset, physical address, virtual address, file size, memory image size, and alignment for a segment in the program. The program header is crucial to creating a process image for the object file. The operating system copies the segment (if it is loadable, i.e., if `p_type` is PT_LOAD) into memory according to the location and size information. The `p_type` field is shown in Figure 2.7 as the first item in the struct.

Figure: The Section Header

```
typedef struct –
    Elf32`Word      sh`name;        //  name of section
    Elf32`Word      sh`type;        //  type of the section
    Elf32`Word      sh`flags;       //  section-specific attributes
    Elf32`Addr      sh`addr;        //  memory location of sectio
    Elf32`Off       sh`offset;      //  file offset to section
    Elf32`Word      sh`size;        //  size of section
    Elf32`Word      sh`link;        //  section type dependent
    Elf32`Word      sh`info;        //  extra information
    Elf32`Word      sh`addralign;   //  address alignment constrai
    Elf32`Word      sh`entsize;     //  size of an entry in section
˜ Elf32`Shdr;
```

The Section Header Table

All sections in object files can be found using the Section header table. The section header, similar to the program header, is an array of structures. Each entry correlates to a section in the file. The entry provides the name, type, memory image starting address (if loadable), file offset, the section's size in bytes, alignment, and how the information in the section should be interpreted. Figure 2.8 details the specific fields of the structure. The name provided in the structure is actually an index into the string table (a section in the object file) where the actual string representation of the name of the section exists. Sections will be discussed further below.

Figure: Special Sections. A brief description of sections that can appear in an ELF object file.

Names of sections	Description of the section
.bss	Uninitialized Data present in process image
.comment	Version control information
.data and .data1	Initialized data present in process image
.debug	Information for symbolic debugging
.dynamic	Dynamic linking information
.dynstr	Strings needed for dynamic linking
.dynsym	Dynamic linking symbol table
.fini	Process termination code
.got	Global offset table
.hash	Symbol hash table
.init	Process initialization code
.interp	Path name for a program interpreter
.line	Line number information for symbolic debugging
.note	File notes
.plt	Procedure linkage table
.relname and .relaname	Relocation Information
.rodata and .rodata1	Read-only data
.shstrtab	Section names
.strtab	Usually names associated with symbol table entries
.symtab	Symbol Table
.text	Executable instructions

ELF Sections

There are a number of types of sections described by entries in the section header table. Sections can hold executable code, data, dynamic linking information, debugging data, symbol tables, relocation information, comments, string tables, and notes. Some sections are loaded into the process image and some provide information needed in the building of a process image while still others are used only in linking object files. Figure 2.9 displays a list of special sections along with a brief description.

ELF Segments

Segments are a way of grouping related sections. For example, the `text` segment groups executable code, the `data` segment groups the program data, and the `dynamic` segment groups information relevant to dynamic loading. Each segment consists of one or more sections. A process image is created by loading and interpreting segments. The operating system logically copies a file's segment to a virtual memory segment according to the information provided in the program header table. The OS can also use segments to create a shared memory resource. Figure 2.9 summarizes the sections that might be included in a segment.

Figure: Data representation. This figure illustrates the representation of ELF data. These data descriptions are machine independent so that a data type that is designated as an Elf32_Half will be the same size on all machines. An Elf32_Half might be used to represent an unsigned short or an unsigned char on some machines. The association between language data types and ELF data types is made in the file <sys/elftypes.h>.

Name	Size	Alignment	Purpose
Elf32`Addr	4	4	Unsigned program address
Elf32`Half	2	2	Unsigned medium integer
Elf32`Off	4	4	Unsigned file offset
Elf32`Sword	4	4	Signed large integer
Elf32`Word	4	4	Unsigned large integer
unsigned char	1	1	Unsigned small integer

ELF Data Representation

The ELF control data is represented in a machine-independent format so that it can be accessed and interpreted seamlessly across machines. Figure 2.10 lists the definitions for the storage classes of the ELF control data. The remaining data in the object file, the data other than the control data, can be encoded to agree with the byte order, in the way necessary for the target machine. All data structures that the object file format defines follow the size and alignment guidelines for the relevant storage class[8]. If necessary, data structures are padded to ensure alignment; for example, a data structure might contain explicit padding to ensure 4-byte alignment for 4-byte objects, to force structure sizes to be a multiple of 4[8]. Alignment information is also included in the structures for sections and segments so that these structures, when placed in memory, can be properly aligned. In order to maintain a high level of portability, data fields in structures are expressed in bytes rather than bits since bit manipulation can be machine dependent. The cost of this portability is some wasted space.

9.9 MODE SWITCHING

To use the processor in protected mode after hardware or software reset, a mode switch must be performed from real-address mode. Once in protected mode, software generally does not need to return to real-address mode. To run software written to run in real-address mode (8086 mode), it is generally more convenient to run the software in virtual-8086 mode, than to switch back to real-address mode.

9.9.1 Switching to Protected Mode

Before switching to protected mode from real mode, a minimum set of system data structures and code modules must be loaded into memory, as described in Section 9.8, “Software Initialization for Protected-Mode Operation.” Once these tables are created, software initialization code can switch into protected mode.

Protected mode is entered by executing a MOV CR0 instruction that sets the PE flag in the CR0 register. (In the same instruction, the PG flag in register CR0 can be set to enable paging.) Execution in protected mode begins with a CPL of 0.

Intel 64 and IA-32 processors have slightly different requirements for switching to protected mode. To insure upwards and downwards code compatibility with Intel 64 and IA-32 processors, we recommend that you follow these steps:

1. Disable interrupts. A CLI instruction disables maskable hardware interrupts. NMI interrupts can be disabled with external circuitry. (Software must guarantee that no exceptions or interrupts are generated during the mode switching operation.)
2. Execute the LGDT instruction to load the GDTR register with the base address of the GDT.
3. Execute a MOV CR0 instruction that sets the PE flag (and optionally the PG flag) in control register CR0.
4. Immediately following the MOV CR0 instruction, execute a far JMP or far CALL instruction. (This operation is typically a far jump or call to the next instruction in the instruction stream.)

   The JMP or CALL instruction immediately after the MOV CR0 instruction changes the flow of execution and serializes the processor.

   If paging is enabled, the code for the MOV CR0 instruction and the JMP or CALL instruction must come from a page that is identity mapped (that is, the linear address before the jump is the same as the physical address after paging and protected mode is enabled). The target instruction for the JMP or CALL instruction does not need to be identity mapped.
5. If a local descriptor table is going to be used, execute the LLDT instruction to load the segment selector for the LDT in the LDTR register.

6. Execute the LTR instruction to load the task register with a segment selector to the initial protected-mode task or to a writable area of memory that can be used to store TSS information on a task switch.
7. After entering protected mode, the segment registers continue to hold the contents they had in real-address mode. The JMP or CALL instruction in step 4 resets the CS register. Perform one of the following operations to update the contents of the remaining segment registers.
   — Reload segment registers DS, SS, ES, FS, and GS. If the ES, FS, and/or GS registers are not going to be used, load them with a null selector.
   — Perform a JMP or CALL instruction to a new task, which automatically resets the values of the segment registers and branches to a new code segment.
8. Execute the LIDT instruction to load the IDTR register with the address and limit of the protected-mode IDT.
9. Execute the STI instruction to enable maskable hardware interrupts and perform the necessary hardware operation to enable NMI interrupts.

Random failures can occur if other instructions exist between steps 3 and 4 above. Failures will be readily seen in some situations, such as when instructions that reference memory are inserted between steps 3 and 4 while in system management mode.

9.9.2 Switching Back to Real-Address Mode

The processor switches from protected mode back to real-address mode if software clears the PE bit in the CR0 register with a MOV CR0 instruction. A procedure that re-enters real-address mode should perform the following steps:

1. Disable interrupts. A CLI instruction disables maskable hardware interrupts. NMI interrupts can be disabled with external circuitry.
2. If paging is enabled, perform the following operations:
   — Transfer program control to linear addresses that are identity mapped to physical addresses (that is, linear addresses equal physical addresses).
   — Insure that the GDT and IDT are in identity mapped pages.
   — Clear the PG bit in the CR0 register.
   — Move 0H into the CR3 register to flush the TLB.
3. Transfer program control to a readable segment that has a limit of 64 KBytes (FFFFH). This operation loads the CS register with the segment limit required in real-address mode.
4. Load segment registers SS, DS, ES, FS, and GS with a selector for a descriptor containing the following values, which are appropriate for real-address mode:
   — Limit = 64 KBytes (0FFFFH)

— Byte granular (G = 0)

— Expand up (E = 0)

— Writable (W = 1)

— Present (P = 1)

— Base = any value

The segment registers must be loaded with non-null segment selectors or the segment registers will be unusable in real-address mode. Note that if the segment registers are not reloaded, execution continues using the descriptor attributes loaded during protected mode.

5. Execute an LIDT instruction to point to a real-address mode interrupt table that is within the 1-MByte real-address mode address range.
6. Clear the PE flag in the CR0 register to switch to real-address mode.
7. Execute a far JMP instruction to jump to a real-address mode program. This operation flushes the instruction queue and loads the appropriate base and access rights values in the CS register.
8. Load the SS, DS, ES, FS, and GS registers as needed by the real-address mode code. If any of the registers are not going to be used in real-address mode, write 0s to them.
9. Execute the STI instruction to enable maskable hardware interrupts and perform the necessary hardware operation to enable NMI interrupts.

NOTE

All the code that is executed in steps 1 through 9 must be in a single page and the linear addresses in that page must be identity mapped to physical addresses.

9.10 INITIALIZATION AND MODE SWITCHING EXAMPLE

This section provides an initialization and mode switching example that can be incorporated into an application. This code was originally written to initialize the Intel386 processor, but it will execute successfully on the Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors. The code in this example is intended to reside in EPROM and to run following a hardware reset of the processor. The function of the code is to do the following:

- Establish a basic real-address mode operating environment.
- Load the necessary protected-mode system data structures into RAM.
- Load the system registers with the necessary pointers to the data structures and the appropriate flag settings for protected-mode operation.
- Switch the processor to protected mode.

Figure 9-3 shows the physical memory layout for the processor following a hardware reset and the starting point of this example. The EPROM that contains the initialization code resides at the upper end of the processor's physical memory address range, starting at address FFFFFFFFH and going down from there. The address of the first instruction to be executed is at FFFFFFF0H, the default starting address for the processor following a hardware reset.

The main steps carried out in this example are summarized in Table 9-4. The source listing for the example (with the filename STARTUP.ASM) is given in Example 9-1. The line numbers given in Table 9-4 refer to the source listing.

The following are some additional notes concerning this example:

- When the processor is switched into protected mode, the original code segment base-address value of FFFF0000H (located in the hidden part of the CS register) is retained and execution continues from the current offset in the EIP register. The processor will thus continue to execute code in the EPROM until a far jump or call is made to a new code segment, at which time, the base address in the CS register will be changed.
- Maskable hardware interrupts are disabled after a hardware reset and should remain disabled until the necessary interrupt handlers have been installed. The NMI interrupt is not disabled following a reset. The NMI# pin must thus be inhibited from being asserted until an NMI handler has been loaded and made available to the processor.
- The use of a temporary GDT allows simple transfer of tables from the EPROM to anywhere in the RAM area. A GDT entry is constructed with its base pointing to address 0 and a limit of 4 GBytes. When the DS and ES registers are loaded with this descriptor, the temporary GDT is no longer needed and can be replaced by the application GDT.
- This code loads one TSS and no LDTs. If more TSSs exist in the application, they must be loaded into RAM. If there are LDTs they may be loaded as well.