# IA-32 Boot Sector Code

- **Boot Sector Code**
- **Boot Loader (Real & Protected Mode)**
- **Real-Mode Test Kernel**
- **Protected-Mode Test Kernel**

WEQAAR A. JANJUA
<janjua@asu.edu>

Department of Computing Studies
Arizona State University - Polytechnic

# 0. ABSTRACT

The project is to write FAT-12 filesystem on a floppy disk and a bootloader that boots two different test kernels i.e. Real-mode and Protected-mode by first switching to the appropriate mode and handing the control over to the kernel. The kernels just print a string on bootup, they are not real kernels i.e. written just for the sole purpose of demonstrating boot loading process.

# 1. INTRODUCTION

A computer system is a complex machinery, and the operating system is an elaborate tool that unrolls hardware complexities to end up showing a simple and standardized environment to the end user. When the power is turned on, however, the system software must work in a limited environment, and it must load the kernel using this scarce operating environment. This paper describes the booting process of IA-32 Platform.

# 2. THE COMPUTER AT POWER-ON

In order to be able to do something with the computer when power is applied, things are arranged so that the processor begins execution from the system's firmware. The firmware is "unmovable software" found in ROM memory; some companies call it BIOS (Basic Input-Output System) to underline its software role, some call it PROM or "flash" to stress on its hardware implementation, while someone else calls it "console" to focus on user interaction.

The firmware usually checks that the hardware is correctly working, and retrieves part (or all) of the kernel from a storage medium and executes it. This first part of the kernel must load the rest of itself and initialize the whole system.

# 3. THE PC

When the x86 processor is turned on in a personal computer, it is a 16-bit processor that only sees one Meg of RAM. This environment is known as "real mode", and is dictated by compatibility with older processors of the same family. It performs a POST(Power On Self Test) that initializes the chip set and checks that the computer is able to function correctly. Since the BIOS provide some basic hardware access it also initializes that and performs whatever internal house keeping that is necessary. One of the thing that is does is set up the BIOS Data Area.

When the BIOS is done starting up it loads the first sector of the floppy disk into memory at *0x0000:0x7c00*. That first sector is the *BOOT SECTOR*. The BIOS checks the format of the boot sector and will usually complain with some BIOS dependent message like "No system on disk" if it encounters an error.

**FORMAT OF THE BOOT SECTOR:**

The boot sector is 512 bytes long and is the very first sector on a floppy disk. The first 3 bytes of the sector must be a jump or a short jump followed by a *NOP*.

```
jmp start_code
        ; some code
        start_code:
OR
        jmp short start_code
       nop                        ;required nop
       ; some code
start_code:
```

Some BIOS's reportedly check for the *NOP*. For FAT compatibility the next 59 bytes must contain the BIOS Parameter Block.

**THE BIOS PARAMETER BLOCK**

| **Offset** | **Size** | **Description (default FAT12 1.44 Mb value)** |
|---|---|---|
| 0 | 8 | Name of operating system |
| 8 | 2 | Bytes per sector (0x200) |
| 10 | 1 | Sectors per cluster (1) |
| 11 | 2 | Reserved sectors (1) |
| 13 | 1 | Number of FATS (2) |
| 14 | 2 | Root directory entries (0x00E0) |
| 16 | 2 | Total sectors (0x0B40) |
| 18 | 1 | Media Descriptor (0xF0) |
| 19 | 2 | Sectors per FAT (9) |
| 21 | 2 | Sector per track (0x12) |
| 23 | 2 | Number of heads (2) |
| 25 | 4 | Hidden sectors (0) |
| 29 | 4 | Total sectors huge (0) |
| 33 | 1 | Drive number (0) |
| 34 | 1 | Reserved |
| 35 | 1 | Signature (0x29) |
| 36 | 4 | Volume ID |
| 40 | 11 | Volume name[1] |
| 51 | 8 | File system type ('FAT12') |

(All the strings must be padded with spaces.)

## FLOPPY DISK MEDIA DESCRIPTORS

| **Descriptor** | **Format** | **Size** | **Cylinders** | **Heads** | **Sectors[1]** | **FAT size[2]** | **Root size[2]** |
|---|---|---|---|---|---|---|---|
| 0xfe | 160 Kb | 5 1/4 | 40 | 1 | 8 | ? | ? |
| 0xfc | 180 Kb | 5 1/4 | 40 | 1 | 9 | ? | 4 |
| 0xff | 320 Kb | 5 1/4 | 40 | 2 | 8 | ? | ? |
| 0xfd | 360 Kb | 5 1/4 | 40 | 2 | 9 | 4 | 7 |
| 0xf9 | 720 Kb | 3 1/2 | 80 | 2 | 9 | 6 | 7 |
| 0xf9 | 1.2 Mb | 5 1/4 | 80 | 2 | 15 | 14 | 14 |
| 0xf0 | 1.44 Mb | 3 1/2 | 80 | 2 | 18 | 18 | 14 |
| ? | 2.88 Mb | 3 1/2 | 80 | 2 | 36 | ? | ? |

[1] Sectors per. cylinder
[2] Size in sectors

Then comes the code that will load an operating system from disk.

At the very end of the boot sector at *OFFSET 510* (just two bytes before the end) you must store the boot disk signature.

*dw 0xAA55 ; boot disk signature*

## FORMAT OF BOOT SECTOR

| Offset (byte) | Size (bytes) | Description |
|---|---|---|
| 0 | 3 | Jump |
| 3 | 59 | BIOS Parameter Block |
| 62 | 448 | Your code |
| 510 | 2 | Signature |

The BIOS leaves the number of the boot drive in the '*dl*' register before transferring control to your boot sector by jumping to *0x0000:0x7C00*. All other registers are undefined.

**MEMORY MAP:**

This is a map of the first megabyte of memory right after the BIOS has transferred control to the boot sector code

| Address | Size | Name |
|---|---|---|
| 0x0000:0x0000 | 1024 bytes | Interrupt Vector Table |
| 0x0040:0x0000 | 256 bytes | BIOS Data Area |
| 0x0050:0x0000 | ? | Free memory |
| 0x07C0:0x0000 | 512 bytes | Boot sector code |
| 0x07E0:0x0000 | ? | Free memory |
| 0xA000:0x0000 | 64 Kb | Graphics Video Memory |
| 0xB000:0x0000 | 32 Kb | Monochrome Text Video Memory |
| 0xB800:0x0000 | 32 Kb | Color Text Video Memory |
| 0xC000:0x0000 | 256 Kb[1] | ROM Code Memory |
| 0xFFFF:0x0000 | 16 bytes | More BIOS data |

## POSSIBLE VALUES OF DL WHEN THE BOOT SECTOR CODE STARTS EXECUTING

| Value of dl register | Corresponding drive |
|---|---|
| 0x00 | First floppy drive |
| 0x01 | Second floppy drive |
| 0x80 | First hard drive |
| 0x81 | Second hard drive |

What is actually in the boot sector is the code to execute and possible some data too. Since the computer will attempt to execute the data in the boot sector it has to contain valid code.

The task of the boot sector is to prepare for and load the next step of the operating system. The simplest is to load an image from disk and transferring control to it immediately. But there is plenty of room left for doing more things in the boot sector. It could be entering protected mode (which is demonstrated practically in the project).

The first thing the boot sector should do after the jump is to initialize the data segment and set up a stack.

If you have some data in your boot sector, i.e. text to display on the screen, you have to initialize *ds* to a known value before using it to index the data with. Using the segment where the BIOS loaded the boot sector *(0x07C0)* is very convenient.

## INITIALIZING DS REGISTER:

```
mov ax, 0x07C0
mov ds, ax              ; setup ds register
```

## SETTING UP A STACK:

The stack can be put on any place, as long as it does not interfere with the location of the boot sector code or some other areas of reserved memory areas. You should also pay attention later when you load the kernel image or maybe relocate the code. I have chosen to place it at 0x9000:0x0000. Without a stack it can be dangerous to call the BIOS, since you don't know whether it has its own stack or is using yours. If it is not set up probably it could possibly corrupt data or code.

```
mov ax, 0x9000
mov ss, ax              ;setup a stack
mov sp, 0x2000          ;8 kb
```

**INITIALIZING DS REGISTER:**

```
mov ax, 0x2000
mov ds, ax                ; setup ds to match new location
jmp 0x2000:0x0000       ; transfer control new location
```

The bootloader provides the user with two options:

- Load REAL-MODE KERNEL
- Load PROTECTED-MODE KERNEL

**TRANSFERRING CONTROL TO THE REAL MODE TEST KERNEL:**

The bootloader loads the kernel at the very bottom of the memory. Therefore the boot sector code needs to be relocated. For the same reason I also relocate the BIOS Data Area by moving it to *0x7000:0x0000*. Then after the kernel has loaded it can extract the data that it needs from there. The next step is to reset the disk drive and read the kernel image into memory. The kernel is stored directly after the boot sector on the floppy disk, it fits into one sector.

The last step is to transfer the control to the test kernel. This is simply done using a *jmp*. But before that I set up the *ds* register so that the kernel doesn't have to do that. I prefer to enter the kernel in a known and stable state.

```
mov ax, 0x2000
mov ds, ax             ;setup ds to match new location
jmp 0x2000:0x0000    ; transfer control new location
```

**TRANSFERRING CONTROL TO THE PROTECTED MODE TEST KERNEL:**

In order to switch from real-mode to protected mode:

- *CLI*: Disable interrupts, because the installed interrupts are all written for real mode and if an interrupt would occur after the mode switch, your system would probably reboot.
- Load the GDTR using *lgdt*, to set up the GDT.
- Execute a *mov CR0* instruction to set the PE bit of control register 0.
- Immediately after the *mov, cr0* instruction perform a far jump to clear the instruction prefetch queue, because it's still filled with real mode instructions and addresses.
- Reload all the segment registers except CS. (which is reloaded by the far jump)
- Load the Interrupt descriptor tables to make interrupts possible
- *STI*: Re-enable interrupts.
- Enable the A20 line to prevent memory wrap.
- Disable NMI (non-maskable interrupts)

**ENABLE THE A20 ADDRESS LINE:**

In order to use the full amount of RAM plugged in your computer you have to enable the *a20* address line. As mentioned earlier enabling a line of the floppy controller can do this. Setting the appropriate bit can change the state of this line. This bit is the second bit of the AT keyboard controller output port *(port 064h)*. So in theory we can enable the *a20* address line by simply setting this second bit.

**DISABLING THE NMI:**

The NMI belongs to the interrupts issued by the hardware. But a NMI (Non Maskable Interrupt) is supplied to the processor directly and not via the 8259A PIC. The NMI usually reports a parity error when reading a byte from memory.

The problem is that you can't disable the NMI with the *CLI* instruction. However, there are times you have to disable it (e.g. when switching into protected mode). A register is provided for this purpose.

The NMI mask register allows to disable (or enable the NMI). This register is controlled by bit 7 of *port 0A0h* for the PC/XT and by bit 7 of *port 70h* for the AT and his successors. Note that in the AT the address register for the CMOS RAM and the real-time clock are also located at port address *70h*. You should take care of modifying bit 7 only.

**4. WHAT IS PROTECTED MODE**

The 8088 CPU used in the original IBM PC was not very scalable. In particular, there was no easy way to access more than 1 megabyte of physical memory. To get around this while allowing backward compatibility, Intel designed the 80286 CPU with two modes of operation: real mode, in which the '286' acts like a fast 8088, and protected mode (now called 16-bit protected mode).

Protected mode allows programs to access more than 1 megabyte of physical memory, and protects against misuse of memory (i.e. programs can't execute a data segment, or write into a code segment). An improved version, 32-bit protected mode, first appeared on the '386 CPU'.

**DIFFERENCES BETWEEN REAL- AND PROTECTED MODES**

| | **REAL MODE** | **16-BIT PROTECTED MODE** | **32-BIT PROTECTED MODE** |
|---|---|---|---|
| Segment base address | 20-bit (1M byte range) = 16 * segment register | 24-bit (16M byte range), from descriptor | 32-bit (4G byte range), from descriptor |
| Segment size (limit) | 16-bit, 64K bytes (fixed) | 16-bit, 1-64K bytes | 20-bit, 1-1M bytes or 4K-4G bytes |
| Segment protection | no | yes | yes |
| Segment register | segment base adr / 16 | selector | selector |

**PROTECTED MODE AND SEGMENTED MEMORY:**

The segments are still there, but in 32-bit protected mode, you can set the segment limit to 4G bytes. This is the maximum amount of physical memory addressable by a CPU with a 32-bit address bus. Limit-wise, the segment then "disappears" (though other protection mechanisms remain in effect). This reason alone makes 32-bit protected mode popular.

**DESCRIPTOR:**

In real mode, there is little to know about the segments. Each is 64K bytes in size, and you can do with the segment what you wish: store data in it, put your stack there, or execute code stored in the segment. The base address of the segment is simply 16 times the value in one of the segment registers.

In protected mode, besides the segment base address, we also need the segment size (limit) and some flags indicating what the segment is used for. This information goes into an 8-byte data structure called a descriptor:

## CODE/DATA SEGMENT DESCRIPTOR

| **Lowest byte** | **Byte 1** | **Byte 2** | **Byte 3** | **Byte 4** | **Byte 5** | **Byte 6** | **Highest byte** |
|---|---|---|---|---|---|---|---|
| Limit 7:0 | Limit 15:8 | Base 7:0 | Base 15:8 | Base 23:16 | Access | Flags, Limit 19:16 | Base 31:24 |

This is a 32-bit ('386) descriptor. 16-bit ('286) descriptors have to top two bytes (Limit 19:16, Flags, and Base 31:24) set to zero. The Access byte indicates segment usage (data segment, stack segment, code segment, etc.):

## ACCESS BYTE OF CODE/DATA SEGMENT DESCRIPTOR

| **Highest bit** | **Bits 6, 5** | **Bit 4** | **Bits 3** | **Bit 2** | **Bit 1** | **Lowest bit** |
|---|---|---|---|---|---|---|
| Present | Privilege | **1** | Executable | Expansion direction/ conforming | Writable/ readable | Accessed |

**Present bit:** Must be set to one to permit segment access.

**Privilege:** Zero is the highest level of privilege (Ring 0), three is the lowest (Ring 3).

**Executable bit:** If one, this is a code segment, otherwise it's a stack/data segment.

**Expansion direction (stack/data segment):** If one, segment grows downward, and offsets within the segment must be greater than the limit.

**Conforming (code segment):** Privilege-related.

**Writable (stack/data segment):** If one, segment can be written to.

**Readable (code segment):** If one, segment can be read from. (Code segments are not writable.)

**Accessed:** This bit is set whenever the segment is read from or written to.

The 4-bit Flags value is non-zero only for 32-bit segments:

**FLAGS NYBBLE**

| **Highest bit** | **Bit 6** | **Bit 5** | **Bit 4** |
|---|---|---|---|
| Granularity | Default Size | 0 | 0 |

The granularity bit indicates if the segment limit is in units of 4K byte pages (G=1) or if the limit is in units of bytes (G=0). For stack segments, the default Size bit is also known as the B (Big) bit, and controls whether 16- or 32-bit values are pushed and popped. For code segments, the D bit indicates whether instructions will operate on 16-bit (D=0) or 32-bit (D=1) quantities by default. To expand upon this: when the D bit is set, the segment is *USE32*, named after the assembler directive of the same name. The following sequence of hex bytes '*B8 90 90 90 90*' will be treated by the CPU as a 32-bit instruction, and will disassemble as *mov eax, 90909090h*. In a 16-bit (*USE16*) code segment, the same sequence of bytes would be equivalent to

*mov ax,9090h*
*nop*
*nop*

Two special opcode bytes called the Operand Size Prefix and the Address Length Prefix reverse the sense of the D bit for the instruction destination and source, respectively. These prefixes affect only the instruction that immediately follows them.

Bit 4 of the Access byte is set to one for code or data/stack segments. If this bit is zero, you have a system segment. These come in several varieties:

**Task State Segment (TSS):** These are used to simplify multitasking. The '386 or higher CPU has four sub-types of TSS.

**Local Descriptor Table (LDT):** Tasks can store their own private descriptors here, instead of the GDT.

**Gates:** These control CPU transitions from one level of privilege to another. Gate descriptors have a different format than other descriptors:

**GATE DESCRIPTOR**

| Lowest byte | Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Highest byte |
|---|---|---|---|---|---|---|---|
| Offset 7:0 | Offset 15:8 | Selector 7:0 | Selector 15:8 | Word Count 4:0 | Access | Offset 23:16 | Offset 31:24 |

Note the Selector field. Gates work through indirection, and require a separate code or TSS descriptor to function.

**ACCESS BYTE OF SYSTEM SEGMENT DESCRIPTOR**

| Highest bit | Bits 6, 5 | Bit 4 | Bits 3, 2, 1, 0 |
|---|---|---|---|
| Present | Privilege | **0** | Type |

**SYSTEM SEGMENT TYPES**

| Type | Segment function | | Type | Segment function |
|---|---|---|---|---|
| 0 | (Invalid) | | 8 | (Invalid) |
| 1 | Available '286 TSS | | 9 | Available '386 TSS |
| 2 | LDT | | 10 | (Undefined, reserved) |
| 3 | Busy '286 TSS | | 11 | Busy '386 TSS |
| 4 | '286 Call Gate | | 12 | '386 Call Gate |
| 5 | Task Gate | | 13 | (Undefined, reserved) |
| 6 | '286 Interrupt Gate | | 14 | '386 Interrupt Gate |
| 7 | '286 Trap Gate | | 15 | '386 Trap Gate |

For now, TSSes, LDTs, and gates are the three main types of system segment.

**DESCRIPTORS**

They are stored in a table in memory: the Global Descriptor Table (GDT), Interrupt Descriptor Table (IDT), or one of the Local Descriptor Tables.

The CPU contains three registers: GDTR, which must point to the GDT, IDTR, which must point to the IDT (if interrupts are used), and LDTR, which must point to the LDT (if the LDT is used). Each of these tables can hold up to 8192 descriptors.

**SELECTOR:**

In protected mode, the segment registers contain selectors, which index into one of the descriptor tables. Only the top 13 bits of the selector are used for this index. The next lower bit chooses between the GDT and LDT. The lowest two bits of the selector set a privilege value.

**HOW TO ENTER PROTECTED MODE**

Entering protected mode is actually rather simple.

**You must:**

- Create a valid Global Descriptor Table (GDT)
- (Optional) create a valid Interrupt Descriptor Table (IDT)
- Disable interrupts
- Point GDTR to your GDT
- (Optional) point IDTR to your IDT
- Set the PE bit in the MSW register
- Do a far jump (load both CS and IP/EIP) to enter protected mode (load CS with the code segment selector)
- Load the DS and SS registers with the data/stack segment selector
- Set up a pmode stack
- (Optional) enable interrupts

## SOURCE CODE SECTION

## BOOTSECTOR AND BOOTLOADER:

```
;-----------------------------------------------------------------------------
; BOOT SECTOR CODE by Weqaar A. Janjua
;-----------------------------------------------------------------------------

; to assemble: nasm bootsec.asm -f bin -o bootsec.bin
; to transfer to disk: partcopy bootsec.bin 0 200 -f0
[bits 16]
[org 0]
        jmp short start
        nop                             ; required nop as some BIOS'es need it
;-----------------------------------------------------------------------------
; BIOS PARAMETER BLOCK (definitions for protected mode)
;-----------------------------------------------------------------------------
; FIELD SIZE (bytes)
        osname                  db 'RAPTOR  '                   ; 8
        bytespersector          dw 0x200                        ; 2
        sectorspercluster       db 1                            ; 1
        reservedsectors         dw 1                            ; 2
        numberoffats            db 2                            ; 1
        rootdirectoryentries    dw 0x00E0  ;224                 ; 2
        totalsectors            dw 0x0B40  ;2880                ; 2
        mediadescriptor         db 0xF0    ;1.44 MB             ; 1
        sectorsperfat           dw 2                            ; 2
        sectorspertrack         dw 0x12                         ; 2
        numberofheads           dw 2                            ; 2
        hiddensectors           dd 0                            ; 4
        totalsectorshuge        dd 0                            ; 4
        drivenumber             db 0                            ; 1
        reserved                db 0                            ; 1
        signature               db 0x29                         ; 4
        volumeid                dd 0                            ; 1
        volumename              db 'NONAME     '                ; 1
        filesystemtype          db 'FAT12   '                   ; 8

;-----------------------------------------------------------------------------
; CODE
;-----------------------------------------------------------------------------

; ------------------------------------------
; Functions used in the boot-loading process
; ------------------------------------------

start:
        cli                             ; diable interrupts
        mov ax, 0x07C0
        mov ds, ax                      ; setup ds register
        mov ax, 0x9000
        mov es, ax                      ; setup a stack
        mov sp, 0x2000                  ; 8 kb
        sti                             ; enable interrupts
        mov [bootdrive], dl             ; save boot drive
```

```
; relocate code
        mov ax, 0x8000
        mov es, ax
        mov di, 0                       ; destination address
        mov si, 0                       ; source address.
        mov cx, 512                     ; length is 512 bytes
        cld                             ; direction forward
        rep movsb                       ; move the boot sector
        jmp 0x8000:relocation_ok        ; transfer control to new location

relocation_ok:
; relocate bios data area
        mov ax, 0x7000
        mov es, ax
        mov di, 0                       ; destination
        mov ax, 0x0040
        mov ds, ax
        mov si, 0                       ; source
        mov cx, 256                     ; length is 256 bytes
        cld                             ; set direction forward
        rep movsb                       ; move bios data area
        mov ax, 0x8000
        mov ds, ax                      ; setup ds to match new location

call_user:
        mov si, ask_user
        call bios_print_string
        mov si, option1
        call bios_print_string
        mov si, option2
        call bios_print_string
        jmp o_a

o_a:
        mov ah, 0               ; wait for key
        int 016h
        cmp al, 'p'
        je s_p
        jne s_r
        ret

s_p:                            ; protected mode section
        call bios_clear_screen
    ;   load kernel image
        call bios_reset_drive
        jnc drive_ok
        mov si, driveerr
        call bios_print_string
        call reboot

drive_ok:
        mov ax, 0x200
        mov es, ax
        mov bx, 0                       ; kernel image destination
        mov al, 1                       ; read 1 sector
        mov cl, 2                       ; starting at sector 2
```

```
        call bios_read_sectors

; ENABLE THE A20 LINE:
;In order to use the full amount of RAM plugged in your computer you have to enable the a20
addressline. This can be done by enabling a line of the floppy controller. The state of this line
can be changed by setting the appropriate bit. This bit is the second bit of the AT  keyboard
controller output port (port 064h). So in theory we can enable the a20 address line by simply
setting this second bit.

        cli                             ; disable interrupts
        mov bl, 0xd0                    ; read current status command
        call kbd_send_ctrl_cmd
        call kbd_read_data
        or al, 2                        ; set the a20 enable bit
        push ax
        mov bl, 0xd1                    ; write current status command
        call kbd_send_ctrl_cmd
        pop bx
        call kbd_write_data             ; write the new status
        mov bl, 0xd0                    ; read current status command
        call kbd_send_ctrl_cmd
        call kbd_read_data              ; read the current status
        and al, 2
        sti                             ; enable interrupts
        jnz a20_ok
        mov si, a20err
        call bios_print_string
        call reboot

a20_ok:
; setup global descriptor table
        mov ax, 0
        mov es, ax
        mov di, 0x800                   ; destination
        mov si, gdt                     ; source
        mov cx, 24                      ; length
        cld                             ; forward direction
        rep movsb                       ; move gtd to its new location
        lgdt [gdtptr]                   ; load gdt register

; Disable ALL interrupts
        cli                             ; disable interrupts
        mov al, 11111111b               ; select to mask of all IRQs
        out 0x21, al                    ; write it to the PIC controller

;Disable NMI
        in al, 0x70                     ; read a value
        or al, 10000000b                ; set the nmi disable bit
        out 0x70, al                    ; write it back again

; Enter protected mode
        mov eax, cr0
        or al, 1                        ; set protected mode bit
        mov cr0, eax
```

```
; Transfer control to kernel
        mov ax, 0x10
        mov ds, ax                  ; load global data selector into ds
        jmp 0x08:0x2000             ; transfer control to test kernel

ret
;---------------------------------------------------------------------------
; start of realmode kernel proc
;---------------------------------------------------------------------------
s_r:                                    ; real-mode section
    call bios_clear_screen
    call bios_reset_drive
    jnc reset_ok
    mov si,driveerr
    call bios_print_string
    call reboot

reset_ok:
    mov al, 1       ; sector count
    mov cl, 3       ; start sector
    mov ax, 0x2000
    mov es, ax
    mov bx, 0       ; kernel image destination
    call bios_read_sectors
    jmp 0x2000:0x0000 ; transfer control to kernel
    call call_user
ret

;---------------------------------------------------------------------------
; END of realmode kernel proc
;---------------------------------------------------------------------------

;---------------------------------------------------------------------------
; Functions
;---------------------------------------------------------------------------

bios_print_string:
; input : ds:si points to zero terminated string
        cld                         ; direction forward
        lodsb                       ; get next character
        cmp al, 0
        jz bios_print_string_done
        mov ah, 0x0E                ; write character as tty function
        int 0x10                    ; call bios video services
        jmp bios_print_string

bios_print_string_done:
        ret

bios_clear_screen:
        mov al, 3                   ; select video mode 3 - color text
        mov ah, 0                   ; set video mode function
        int 0x10                    ; call bios video services
        ret
```

```
reboot:
        mov si, pm
        call bios_print_string
        mov ah, 0                       ; read keypress function
        int 0x16                        ; call bios keyboard services
        jmp 0xFFFF:0x0000

bios_reset_drive:
        mov ah, 0                       ; reset drive function
        int 0x13                        ; call bios disk i/o
        ret

bios_read_sectors
; input : es:bx = address of destination
;         al = sector count
;         cl = sector start number
        mov ah, 0x02                    ; read sectors function
        mov ch, 0                       ; cylinder 0
        mov dl, [bootdrive]             ; drive number
        mov dh, 0                       ; head number
        int 0x13                        ; call bios disk i/o
        jc bios_read_sectors
        ret

kbd_wait_cmd:
        in al, 0x64                     ; read the controller status port
        and al, 2                       ; check if the controller is ready
        jnz kbd_wait_cmd                ; to accept the next command (or
        ret                             ; piece of data)

kbd_wait_data:
        in al, 0x64                     ; read the controller status port
        and al, 1                       ; check if the data is ready
        jz kbd_wait_data
        ret

kbd_send_ctrl_cmd:
; input : bl = command
        call kbd_wait_cmd
        mov al, bl
        out 0x64, al                    ; send the command to the control
        ret                             ; register

kbd_read_data:
; output : al = data
        call kbd_wait_data
        in al, 0x60                     ; read data from input/output port
        ret

kbd_write_data:
; input bl = data
        call kbd_wait_cmd
        mov al, bl
        out 0x60, al                    ; write data to input/output port
        ret
```

```
;---------------------------------------------------------------------------
; data
;---------------------------------------------------------------------------

; messages (with carriage return and line feed and zero terminated)
        bm_p        db    'PROTECTED', 0
        bm_r        db    'REAL', 0
        pm          db    'rb', 0 ; reboot message
        driveerr    db    'E', 0  ; DRIVE ERROR
        a20err      db    'E', 0   ; A20 ERROR
        ask_user    db    'SELECT:',13,10,0
        option1     db    'p',13,10,0
        option2     db    'r',13,10,0
        bootdrive   db    0

; global descriptor table
    ; null selector (required)
        gdt dw 0, 0, 0, 0

    ; kernel code selector
        dw 0xffff           ; segment limit (4 gb total)
        dw 0                ; base address (bits 0-15)
        db 0                ; base address (bits (16-24)
        db 10011000b        ; dpl 0, code (execute only)
        db 11001111b        ; granlurarity (4k), 32-bit, limit high nibble = f
        db 0                ; base address (bits 24-32)

    ; kernel data selector
        dw 0xffff           ; segment limit (4 gb total)
        dw 0                ; base address (bits 0-15)
        db 0                ; base address (bits (16-24)
        db 10010010b        ; dpl 0, data (read/write)
        db 11001111b        ; granlurarity (4k), 32-bit, limit high nibble = f
        db 0                ; base address (bits 24-32)

        gdtptr dw 0x7ff ; limit (256 slots)
        dd 0x800            ; base (physical address)

;---------------------------------------------------------------------------
; signature
;---------------------------------------------------------------------------

    times 510-($-$$) db 0        ; padding - fill the empty space with 512 bytes !!
    dw 0xAA55                    ; boot signature
```

;←-------END OF BOOTLOADER & BOOTSECTOR CODE-–-→

## REAL-MODE TEST KERNEL:

```
[bits 16]

start:
        call ClrScr
        mov ah,13h
        mov al,3 ; write mode (advance cursor, ASCII+attribute string)
```

```
            mov bh,0 ; video page
            mov cx,7 ; string length
            mov dh,1 ; starting row
            mov dl,1 ; starting col
            push cs
            pop es
            mov bp,kernelmsg
            int 10h
            call reboot

;---------------------------------------------------------------------------
; functions
;---------------------------------------------------------------------------

gotoxy:
            mov ah,02h ; select video service 2 (position cursor)
            mov bh,0   ; stay with video page 0
            int 10h
ret

ClrScr:
            pusha
            mov cx,0
            mov dx,LRXY

ClrWin:
            mov al,0

ScrlWin:
            mov bh,07h

video6:
            mov ah,06h
            int 10h
            popa
ret

bios_print_string:
; input : ds:si points to zero terminated string
        cld                     ; direction forward
        lodsb                   ; get next character
        cmp al,0
        jz bios_print_string_done
        mov bh,0                ; setting video page (0)
        mov bl,14h                          ; blue background and red font
        mov ah,0x0E             ; write character as tty function
        int 0x10                ; call bios video services
        jmp bios_print_string

bios_print_string_done:
        ret

reboot:
            mov ah,13h
            mov al,3                        ; write mode (advance cursor, ASCII+attribute string)
            mov bh,0                        ; video page
```

```
            mov bl,02                    ; attribute (black on green)
            mov cx,23                    ; string length
            mov dh,2                     ; starting row
            mov dl,2                     ; starting col
            push cs
            pop es
            mov bp,presskeymsg
            int 10h
            mov ah, 0                    ; read keypress function
            int 0x16                     ; call bios keyboard services
            call do_reset
ret

do_reset:
            jmp 0xFFFF:0x0000

ret
;---------------------------------------------------------------------------
; data
;---------------------------------------------------------------------------

kernelmsg     db 'S', 01, 'U', 02, 'C', 03, 'C', 04, 'E', 05, 'S', 06, 'S', 07, 13, 10
presskeymsg db 'P',01,'R',02,'E',03,'S',04,'S',05,' ',06,'A',07,'N',08,'Y',09,'
                    ',10,'K',11,'E',12,'Y',13,' ',14,'T',15,'O',16,'
                    ',17,'R',18,'E',19,'B',20,'O',21,'O',22,'T',13,10
LRXY            dw          184Fh
times 512-($-$$) db 0          ; padding

;←---------END OF REAL-MODE TEST KERNEL CODE---------→
```

**PROTECTED-MODE TEST KERNEL:**

```
[bits 32]
[org 0x2000]
        mov ax, 0x10
        mov ds, ax
        mov es, ax
        mov esi, kernelmsg
        call pmode_print_string
        mov esi, presskeymsg
        call pmode_print_string

dummy:
        jmp dummy
;---------------------------------------------------------------------------
; function
;---------------------------------------------------------------------------
        xposition db 0
        yposition db 1

pmode_print_character:
; input al : character
;       ah : attribute
        pushad                          ; save registers
        cmp al, 10                      ; line feed
```

```
        jnz not_line_feed
        add byte [yposition], 1
        jmp pmode_print_character_done

not_line_feed:
        cmp al, 13                      ; carriage return
        jnz not_carriage_return
        mov byte [xposition], 0
        jmp pmode_print_character_done

not_carriage_return:
        mov ecx, eax                    ; save character and attribute
        mov ebx, 0
        mov bl, [xposition]
        shl bl, 1                       ; calculate x offset
        mov eax, 0
        mov al, [yposition]
        mov edx, 160
        mul edx                         ; calculate y offset
        mov edi, 0xb8000                ; start of video memory
        add edi, eax                    ; add y offset
        add edi, ebx                    ; add x offset
        mov ax, cx                      ; restore character and attribute
        cld                             ; forward direction
        stosw                           ; write character and attribute
        add byte [xposition], 1

pmode_print_character_done:
        call hardware_move_cursor
        popad                           ; restore registers
        ret

pmode_print_string:
; input ds:esi = points to zero terminated string
        lodsb
        cmp al, 0
        jz pmode_print_string_done
        mov ah, 0x0F                    ; white text, black background
        call pmode_print_character
        jmp pmode_print_string

pmode_print_string_done:
        ret

hardware_move_cursor:
        pushad                          ; save registers
        mov ebx, 0
        mov bl, [xposition]             ; get x offset
        mov eax, 0
        mov al, [yposition]
        mov edx, 80
        mul edx                         ; calculate y offset
        add ebx, eax                    ; calculate index
        ; select to write low byte of index
        mov al, 0xf
        mov dx, 0x03d4
```

```
        out dx, al
        ; write it
        mov al, bl
        mov dx, 0x03d5
        out dx, al
        ; select to write high byte of index
        mov al, 0xe
        mov dx, 0x03d4
        out dx, al
        ; write it
        mov al, bh
        mov dx, 0x03d5
        out dx, al
        popad                           ; restore registers
        ret
;--------------------------------------------------------------------------
; data
;--------------------------------------------------------------------------

        kernelmsg       db 'Protected mode test kernel loaded successfully', 13, 10, 0
        presskeymsg  db 'Please remove disk and reboot', 0
        times 512-($-$$) db 0           ; padding

;←--------END OF PMODE TEST KERNEL CODE------→

;--------------------------------------------------------------------------
; END of code
;--------------------------------------------------------------------------
```