A Bootstrap Loader for x86 Based Workstations

Jacques Eloff
eloff@cs.sun.ac.za

CS314 – Operating Systems
Department of Computer Science
University of Stellenbosch

1 Introduction

The primary responsibility of a bootstrap loader is to load an operating system or runtime environment into memory. When the code of the bootstrap loader is executed, the *central processing unit* (CPU) operates in real mode (16-bit mode). For older systems such as MS-DOS this is not a problem. However, modern operating systems like Windows, OS/2, Linux and Native Oberon operate in protected mode (32-bit mode) to utilize all of the system's resources and switching to protected mode becomes an additional responsibility of the bootstrap loader.

The report is divided into a number of sections. An overview of a typical start-up sequence is presented in Section 2. The primary and secondary stages of a simple bootstrap loader are described in Sections 3 and 4 respectively.

2 The Bootstrap Process

The term *cold boot* refers to the boot process when a PC is switched on and boots for the first time. A *warm boot* will only occur if the PC was reset, either by executing BIOS interrupt `19h` using the `INT nn` instruction or pressing the *Alt-Ctrl-Del* key sequence while the BIOS reset flag at physical address `00472h` is set to `1234h` [2]. The processor automatically starts executing at a fixed address in memory (`FFFF0h`). A jump instruction is executed and control is transferred to the *power-on self test* (POST) routines if a cold boot was executed, otherwise the POST is skipped.

The POST routines conduct a number of tests, such as determining the amount of memory and video display adapter type. Once these tests are completed, control is transferred to the *read-only memory* (ROM) bootstrap loader. The ROM bootstrap loader proceeds by searching for a suitable device containing a bootstrap loader. Such a device may include a removable diskette, fixed disk, CD-ROM or other bootable media. Once located, the ROM bootstrap routine will read the first sector (boot sector) from this device into memory at address `07C0h:0000h` (physical address `07C00h`) before transferring control to the newly loaded program.

The physical size of the boot sector is typically only 512 bytes. Depending on the requirements of the system, this either implies that the bootstrap loader must fit into 512 bytes (executable code and data) or that the loader must be separated into different stages, spread over multiple sectors and making the primary stage responsible for loading the remainder of the bootstrap loader into memory. The loader described in this document follows the latter approach.

3 The Primary Stage

The ROM bootstrap routine read the boot sector and copied the data to physical address `07C00h` before executing a jump instruction to transfer control to the bootstrap loader. The primary stage

of the bootstrap loader will now perform the following tasks:

Lines 36..38 The BIOS screen services, provided by software interrupt `10h`, are used to initialize the video display (Mode 3, 80x25 alphanumeric text).

Lines 40..44 The `JMP` instruction that transferred control to the loader only initialized the code segment selector (`CS`) and instruction pointer `IP`. The location and size of the stack is unknown and the remaining segment selectors (`DS`, `ES`, `SS`) also contain unknown values. The segment selectors are now initialized and a temporary stack is created at address `8000:9000` (physical address `89000`) as illustrated in Figure 1(a).

Lines 46..52 The BIOS equipment flag is read using software interrupt `11h` and its value is stored in the `Equipment_Status` variable. The contents of this variable will be used at a later stage to obtain additional information related to the system's configuration. At this point the loader isolates bits 4 and 5 to determine the type of the display adapter (colour or monochrome). The loader initially assumes that a colour adapter is present, but will modify the `Display_Base` address variable if a monochrome adapter is present, ensuring proper behaviour of the loader's output primitives [6].

Although the BIOS provides a number of display primitives, the loader requires more specialized functions, for example, displaying a value in hexadecimal notation. Furthermore, the BIOS routines are fairly slow and will be obsolete once the processor is switched to protected mode, making it impossible for the loader to display important information.

Lines 61..77 After selecting an appriopriate display base address, the loader determines whether or not the installed CPU is at least an 80386 processor. The detection routine is based on certain characteristics of the 80286 processor and the loader will display an error message if an 80286 processor was discovered [2, 5].

Lines 79..106 The loader now uses the BIOS disk services (software interrupt `13h`) to load its remaining portions into memory. The loader will assume that it is booting from a 1.44Mb removable diskette and will attempt the load operation three times when encountering problems before displaying an error message and aborting the load operation. The secondary stage is placed directly after the primary stage to form a contigious block as illustrated in Figure 1(b).

Lines 113..119 The loader is now relocated to physical address `80000h` to create gap in memory large enough to hold the kernel or runtime environment and is illustrated in Figure 1(c). Finally, control is transferred to the secondary stage by executing a `JMP` instruction to address `8000h:0000h`.

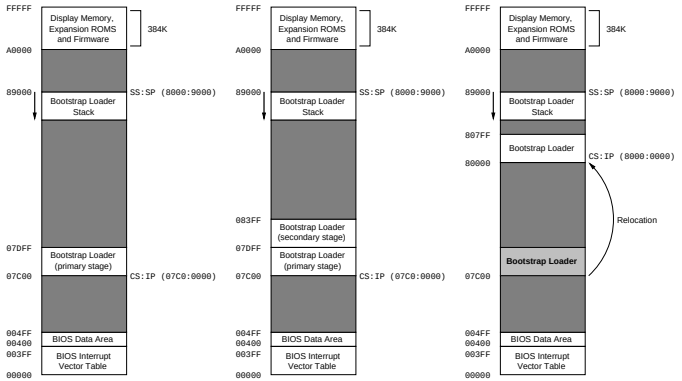
Lines 127..314 This portion of the loader contains the various display primitives and variables used throughout the loader.

4 Stage 2

The secondary stage of the loader performs a number of tasks, including switching the processor to protected mode.

Lines 326, 327 The data segment selector `DS` is reloaded allowing memory references to be correctly resolved after the loader was relocated.

Lines 332..449 The loader now gathers additional information regarding the system's configuration and stores the information in the boot table as illutstrated in Figure 4. The boot table provides a mechanism to transfer information to the kernel to assist with configuring various devices. The first entry in the boot table is located at physical address `009FCh`. Every entry



(a) Primary stage loaded (b) Secondary stage loaded (c) Bootstrap loader is relocated

Figure 1: Memory layout during the primary stage of the bootstrap loader.

is exactly 4 bytes long and the boot table grows down toward address 0 as illustrated in Figure 2(a). Determining the amount of memory installed is somewhat problematic and a more detailed explanation can be found in Section 5.

Lines 454..461 The kernel or runtime environment is loaded into memory at physical address `01000h` as illustrated in Figure 2(b). The `LOAD_IMAGE` procedure (lines 709 to 829) is responsible for loading the kernel image into memory.

For simplicity, it is assumed that the system is contained on a 1.44Mb diskette. The geometry of a 1.44Mb diskette is fairly simple: 80 tracks in total, 18 sectors per track, 2 heads and a sector size of 512 bytes. A temporary disk buffer located at physical address `90000h` is used to facilitate the transfer. The buffer's location also ensures that DMA boundary crossings will not occur.

Although it is possible to read the runtime system directly into memory, the disk buffer makes it easier to deal with segment boundaries. The runtime system's entry point is located at `0000:1000` (physical address `1000`). Given that only 9K (18×512) is transferred, loading the runtime system directly into memory would present a problem because the last 9K read into a segment would overflow into the next. By using the disk buffer, the loading process is separated into two parts. The first deals with segment overruns when the amount of space left in the current segment is less than 9K (lines 782 to 805) and is illustrated in Figure 3(a). The second part simply copies the whole disk buffer to the area allocated for the runtime system (lines 807 to 814) and is illustrated in Figure 3(b).

Lines 463..508 The kernel stack is now allocated directly above the kernel. The loader assumes that the kernel image was created using the *Oberon* linker. Images created with the *Oberon* linker contain a special header that records important information such as the address at which the image was linked and where the image stops. This information is used to determine where the kernel stack should start. The loader will automatically choose an address that is aligned on a 4K page boundary and will ensure that the end of the kernel image and stack bottom are separated by a full 4K page. The linear address of the stack top is calculated

and stored in the boot table along with the stack size. The loader now switches to the kernel stack by converting the linear address of the stack top into a segment and offset. The `SS` segment selector and stack pointer `SP` are reloaded to complete the stack switch.

Lines 525..527 The loader begins preparing the system for the switch to protected mode. The diskette motor is turned off because the BIOS may not do so in time before interrupts are disabled.

Lines 533..536 Maskable interrupts are disabled by executing a `CLI` instruction. Only *non-maskable interrupts* (NMIs) can occur at this point such as memory parity errors.

Lines 548..577 Address line 20 (`A20`) is activated. When A20 is inactive, memory addresses above `FFFFFh` will be truncated to fall within the first 1Mb of addressable memory, effectively wrapping around. This feature was kept to ensure backwards compatibility because it was found that certain legacy software systems actually relied on this feature of the 8088 processor.

Matters are futher complicated by the fact that a pin on the keyboard controller is used to control the AND gate that toggles A20. When IBM introduced the ability to activate A20, they found that the keyboard controller contained an unused pin and decided to use it to control A20. However, this method does not always work. Certain systems contain a setting in the CMOS that allow users to select a fast gating option. When this option is enabled, the system will not always respond to the preceding method. An alternative is to use the system control I/O port (port `92h`) to enable A20. The loader will always attempt to use the keyboard controller first. A simple test is performed to see if A20 is enabled. If not, the system control port is used and the test is repeated. The loader will display and error message if A20 is still not active.

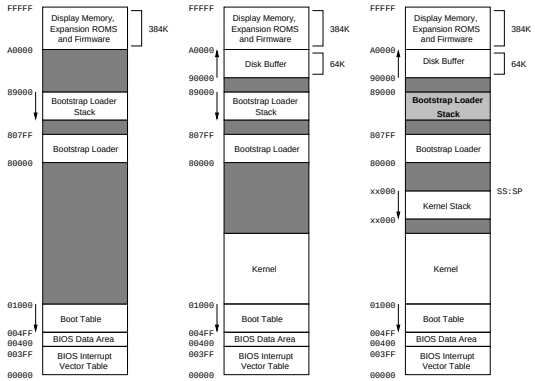
Lines 585..611 The processor requires a number of data structures to operate in protected mode [1, 3, 4]. The first structure that is initialized is called the *interrupt descriptor table* (IDT) and will replace the old *interrupt vector table* (IVT) used in real mode. The base address and size of the IDT is loaded into the *interrupt descriptor table register* (`IDTR`) using the `LIDT` machine instruction. Note that an empty IDT is specified because the kernel will create the actual IDT.

The second structure is called the *Global Descriptor Table* (GDT) and contains entries that describe various memory segments (refer to lines 918..923). The loader creates three segment descriptors: a NULL descriptor, a code segment descriptor and a data segment descriptor. Note that these descriptors specify single segments spanning the complete 4GB of addressable memory, effectively creating a linear address space. The `LGDT` instruction is used to load the *global descriptor table register* (GDTR) with the GDT's base address and size.

Lines 618..630 The entry point of the kernel is pushed on the stack. The first item is the code segment selector (`CS`). Note that its actual value is 8 (the offset of the code segment descriptor in the GDT). The second value contains the offset inside the code segment (`EIP`). The linear offset of the stack top is computed and stored in `EDX` and will later be loaded into `ESP` when the processor operates in protected mode.

Lines 646..661 The processor is switched to protected mode by setting the *Protection Enable* bit in control register 0 (`CR0`). The 80286 did not contain any control registers. Instead, the *machine status word* (`MSW`) register was used to switch the processor to protected mode and special instructions (`LMSW` and `SMSW`) were provided to load and store the `MSW` register. The `MSW` register became the lower 16 bits of `CR0` when control registers were introduced in the 80386. A `JMP` instruction is executed after switching to protected mode to serialize the CPU by flushing the instruction pipeline and prefetch queues.

All the segment selectors, except for `CS`, are reloaded to reference the data segment descriptors in the GDT and the stack pointer `ESP` is loaded with the offset of the kernel stack top

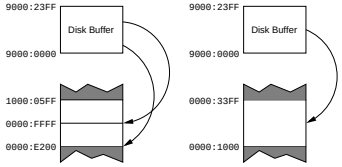


(a) Boot table is updated

(b) Kernel image is loaded

(c) Kernel stack is allocated

Figure 2: Memory layout during the secondary stage of the bootstrap loader.



(a) Two memory copies are executed when crossing a segment boundary

(b) A single copy is executed if the disk buffer's contents fits within the current segment

Figure 3: Copying the disk buffer when loading the runtime system.

that was computed earlier. A far return (`RET`) instruction is executed. The `RET` instruction will pop the instruction pointer (`EIP`) and code segment selector (`CS`) from the stack and transfer control to the kernel.

5 Memory Detection

Determining the amount of memory installed in a system is a fairly complex and error prone task. The bootstrap loader uses three techniques based on various functions provided by BIOS interrupt `15h`.

00FFC	Display base address
00FF8	Diskette installed
00FF4	Diskette total
00FF0	Serial total
00FEC	COM4 base address
00FE8	COM3 base address
00FE4	COM2 base address
00FE0	COM1 base address
00FDC	Display adapter address register
00FD8	Kernel stack top
00FD4	Kernel stack size
00FD0	Extended memory in Kb (Int 15H, 88H)
00FCC	Extended memory in Kb (Int 15H, E801H)
00FC8	Number of SMAP Entries (Int 15H, E820H)

Figure 4: The boot table

5.1 Interrupt `15h`, function `88h`

This function is present in almost every BIOS and reports the amount of extended memory up to 63Mb, but on some systems it only reports the first 15Mb. The operating system should attempt to discover if additional memory is installed if the reported amount is either 63Mb (64512K) or 15Mb (15360K).

5.2 Interrupt `15h`, function `e801h`

This function reports the amount of contigious 1K blocks up to the 16Mb limit as well as the number of 64K blocks present above the 16Mb limit. The loader converts the amount of 64K blocks into 1K blocks and stores the total amount of 1K memory blocks in the boot table (lines 409 to 414).

5.3 Interrupt `15h`, function `e820h`

This function forms part of the *advanced configuration and power management interface* (ACPI) and is only available in newer BIOS releases. The function returns a memory map and must be called multiple times to obtain the complete map. The loader records the number of entries returned and also stores the map. Each entry in the memory map requires 20 bytes and conforms to the structure shown in Table 1

Offset	Size	Description
0	4	Base address (bits 0 . . . 31)
4	4	Base address (bits 32 . . . 63)
8	4	Length (bits 0 . . . 31)
12	4	Length (bits 32 . . . 63)
16	4	Block type (1=ARM, 2=ARR)

Table 1: Layout of System Memory Map entries

Address range memory (ARM) blocks can be used by the operating system's memory manager for allocation. *Address range reserved* (ARR) blocks are reserved and may not be used for allocation by the operating system.

6 Disk Errors

The bootstrap loaders will display the BIOS disk error codes whenever a disk error occurs, The codes and their meanings are listed in Table 2.

Status	Meaning	Status	Meaning
00h	No error	09h	DMA boundary error
01h	Bad command	10h	CRC error on disk read
02h	Address mark not found	20h	Controller failed
03h	Write protected	40h	Seek failed
04h	Invalid track/sector	80h	Device not responding
05h	Reset operation failed	AAh	Drive not ready
06h	Diskette change line active	BBh	Undefined
07h	Illegal drive parameters	CCh	Write fault
08h	DMA overrun		

Table 2: BIOS disk status.

References

[1] John H. Crawford and Patrick P. Gelsinger. *Programming the 80386*. Sybex, 1997.

[2] Terry Dettman and Allen L. Wyatt. *DOS Programmer's Reference Manual*. QUE, 4th edition, 1994.

[3] Intel Corporation. *Intel Architecture Developer's Manual, Volume 3: System Programming*, 1997.

[4] Tom Shanley. *Protected Mode Software Architecture*. Addison Wesley, 1996.

[5] Frank van Gilluwe. *The Undocumented PC*. Addison Wesley, 1997.

[6] Richard Wilson. *Programmer's Guide to PC and PS/2 Video Systems*. Microsoft Press, 1987.

A Source Code

```
;*****************************************************************************************
;* Bootstrap loader for CS314, 28.12.2003                                                *
;* University of Stellenbosch                                                            *
;* Jacques Eloff (eloff@cs.sun.ac.za)                                                    *
;*                                                                                       *
;* References:                                                                           *
;*  - Allen L. Wyatt, DOS Programmer's Reference, 4th edition, 1994, QUE Publishing      *
;*  - John H. Crawford and Patrick P. Gelsinger, Programming the 80386, 1986, Sybex      *
;*  - Intel Architecture Developer's Manual, Volume 3: System Programming, order no      *
;*    243192, 1997, Intel Corporation (Chapter 8)                                        *
;*  - IBM Personal Computer Technical Reference, 1st edition, 1986, International        *
;*    Business Machines Corporation                                                      *
;*  - Pieter Muller, A Bootstrap Mechanism for the Gneiss Kernel, 1995, Technical Report *
;*    TR-950100, Department of Computer Science, University of Stellenbosch              *
;*  - Native Oberon bootstrap loader (OBL.ASM) by Pieter Muller, ETH Zurich              *
;*  - van Gilluwe, Undocumented PC, 2nd Edition, Addison Wesley                          *
;*  - BIOS documentation by Ralph Brown                                                  *
;*****************************************************************************************

%include "boot.inc"                     ; Constants
%include "boot.mac"                     ; Macro's
ORG 0
CODESEG

;*****************************************************************************************
;* The object code is loaded at address 07c0:0000 (physical address 07c00)               *
;* by the ROM bootstrap routines. At this point, only CS points to segment               *
;* 07c0 so the remaining segment registers must be initialized to satisfy                *
;* memory references. The following tasks are performed:                                 *
;* - Switch to 80x25 text mode using the BIOS screen services and determine              *
;*   the display type (colour or monochrome)                                             *
;* - Initialize the segment registers                                                    *
;* - Create a small stack at 8000:9000 (physical address 89000)                         *
;* - Check to ensure that the processor is at least an 80386                             *
;*****************************************************************************************
   mov    ax, 03h                   ; AH = 0: Initialize display function
                                    ; AL = 3: Select 80x25 text mode
   int    10h                       ; Call BIOS screen services

   mov    ax, 07c0h                 ; Setup registers to point to current segment
   mov    ds, ax
   mov    ax, STACK_SEGMENT
   mov    ss, ax
   mov    sp, STACK_OFFSET          ; Stack starts at 8000:9000

   int    11h                       ; Call BIOS 'Get Equipment Status' service
   mov    [Equipment_Status], ax    ; Store equipment information for later
   and    ax, 30h                   ; Isolate bits 4 and 5 (display adapter
                                    ;  information)
   cmp    ax, 30h
   jne    detect_386
   mov    dword [Display_Base], 0b0000000h   ; Adjust display address for monochrome

;*****************************************************************************************
;* The bootstrap loader is designed for a 80386 or later model processor. The flag       *
;* register of the 80286 contains the IOPL bits, but won't allow us to set them. We can  *
;* use this to determine if we're working with a 80386 or later model CPU. The bootstrap *
;* loader will stop executing and display an error message if an 80286 processor is      *
;* detected.                                                                             *
;*****************************************************************************************
detect_386:
   ClearDisplay BG_BLACK | FG_WHITE  ; Clear the display, set display attribute
                                     ;  and display location to (0, 0)
   pushf
   pop    ax
```

```
    or     ax, 3000h                        ; Set IOPL bits
    push   ax
    popf                                    ; If it is an 80286 processor, then the
                                            ;  IOPL bits will be reset by the CPU
    pushf
    pop    ax
    test   ax, 3000h
    jnz    load_2nd_stage
    WriteString Error_Msg1                  ; Display error message

.no_386:
    jmp    .no_386

load_2nd_stage:                             ; Load the remaining portion of the
                                            ;  bootstrap loader into memory

.repeat:
    call   RESET_DISK_DRIVE
    les    bx, [Address_Stage_2]            ; ES:BX points to disk buffer
    mov    ah, BIOS_READ_SECTOR             ; AH = 2: Read Disk Sector function
    mov    al, BOOTSTRAP_SECTORS            ; AL = ?: Number of sectors to read
    mov    cx, 0002h                        ; CH = 0: Track number
                                            ; CL = 2: Starting sector (BIOS numbers disk
                                            ;  sectors from 1)
    xor    dx, dx                           ; DH = 0: Head, DL = 0: Drive
    int    13h                              ; Call BIOS disk services
    jnc    relocate_loader                  ; Carry flag (CF) set if error occurred
                                            ;  (AH contains the controller status)
    dec    byte [Retry]                     ; Retry operation at least three times
    jnz    .repeat                          ;  before reporting the disk error

    push   eax
    WriteString Error_Msg2                  ; Display error message

    pop    eax
    and    eax, 0ff00h
    shr    eax, 8
    call   WRITE_HEX                        ; Display result code of disk operation

.disk_error:
    jmp   .disk_error                       ; Go into endless loop. User must reset
                                            ;  system to try again or use a new disk

;******************************************************************************
;* Relocate the complete bootstrap loader to the top of memory (128K below the display    *
;* memory).                                                                   *
;******************************************************************************
relocate_loader:
    les    di, [Relocate_Destination]
    lds    si, [Relocate_Source]
    mov    ecx, BOOTSTRAP_SIZE
    cld
    rep    movsd
    jmp    SEGMENT_2ND:OFFSET_2ND           ; Transfer control to 2nd stage

;******************************************************************************
;* PROCEDURE: RESET_DISK_DRIVE                                                *
;* INPUT: none                                                                *
;* Reset the disk drive. This procedure should be called to prepare for disk I/O or when  *
;* encountering disk access errors.                                           *
;******************************************************************************
RESET_DISK_DRIVE:
    mov    ah, BIOS_RESET_DISKETTE          ; AH = 0: Reset Disk System function
    xor    dx, dx                           ; DL = 0: Drive
    int    13h                              ; Call BIOS disk services
    ret
```



```
;******************************************************************************
;* PROCEDURE: WRITE_CHAR                                                      *
;* INPUT: AL                                                                  *
;* Writes the character specified in AL to the display at the current screen position     *
;* using the current display attribute.                                       *
;******************************************************************************
WRITE_CHAR:
    cld
    les    di, [Display_Base]
    add    di, [Display_Pos]
    mov    ah, [Display_Attribute]
    stosw
    add    word [Display_Pos], 2
    mov    ax, [Display_Pos]
    cwd
    mov    cx, DISPLAY_SIZE
    div    cx
    mov    word [Display_Pos], dx
    ret

;******************************************************************************
;* PROCEDURE: WRITE_STRING                                                    *
;* INPUT: SI                                                                  *
;* Writes the character string specified in SI to the display at the current screen       *
;* position using the current display attribute. The string must be NULL terminated.     *
;******************************************************************************
WRITE_STRING:
.while:
    lodsb
    cmp    al, 0
    jz     .done
    push   si
    call   WRITE_CHAR
    pop    si
    jmp    .while
.done:
    ret

;******************************************************************************
;* PROCEDURE: WRITE_LN                                                        *
;* INPUT: none                                                                *
;* Updates the current screen position to the next line. The position will automatically  *
;* wrap to the start of the display if WRITE_LN is called when the current screen         *
;* position references the last line.                                         *
;******************************************************************************
WRITE_LN:
.while:
    mov    ax, [Display_Pos]
    cwd
    mov    cx, DISPLAY_WIDTH
    shl    cx, 1
    div    cx
    cmp    dx, 0
    jz     .done
    mov    al, 20h
    call   WRITE_CHAR
    jmp    .while
.done:
    ret

;******************************************************************************
;* PROCEDURE: WRITE_HEX                                                       *
;* INPUT: EAX                                                                 *
;* Displays the contents specified in EAX in hexadecimal at the current screen position   *
;* using the current display attribute.                                       *
;******************************************************************************
WRITE_HEX:
```

```
    cld
    mov    bx, Hex_Chars
    mov    ecx, eax
    mov    si, 8
.while:
    rol    ecx, 4
    mov    al, cl
    and    al, 0fh
    xlatb
    mov    ah, [Display_Attribute]
    les    di, [Display_Base]
    add    di, [Display_Pos]
    stosw
    add    word [Display_Pos], 2
    mov    ax, [Display_Pos]
    cwd
    mov    di, DISPLAY_SIZE
    div    di
    mov    word [Display_Pos], dx
    dec    si
    jnz    .while
    ret

;*****************************************************************************************
;* PROCEDURE: GOTOXY                                                                     *
;* INPUT: AL, BL                                                                         *
;* Changes the current position to the specified row (AL) and column (BL)                *
;*****************************************************************************************
GOTOXY:
    and    ax, 0ffh
    mov    cl, DISPLAY_WIDTH
    imul   cl
    and    bx, 0ffh
    add    ax, bx
    shl    ax, 1                            ; AX: ((y*DISPLAY_WIDTH)+x)*2
    cwd
    mov    bx, DISPLAY_SIZE
    div    bx
    mov    [Display_Pos], dx
    ret

;*****************************************************************************************
;* PROCEDURE: CLEAR_DISPLAY                                                              *
;* INPUT: AH                                                                             *
;* Clears the display to the attribute specified in AH. Resets the current position      *
;* to 0,0 and sets the current attribute to AH.                                          *
;*****************************************************************************************
CLEAR_DISPLAY:
    cld
    les    di, [Display_Base]
    mov    cx, DISPLAY_SIZE
    shr    cx, 1
    mov    al, 20h
    rep    stosw
    mov    word [Display_Pos], 0h
    mov    [Display_Attribute], ah
    ret

;*****************************************************************************************
;*                             INITIALIZED  VARIABLES                                    *
;*****************************************************************************************
DATASEG1
    Error_Msg1          db 'ERROR: CPU is not an 80386', 0
    Error_Msg2          db 'ERROR: Disk error - ', 0
    Hex_Chars           db "0123456789ABCDEF"

    Display_Base        dd 0b8000000h
```



```
    Display_Attribute   db 07h
    Display_Pos         dw 0000h

; Segment:Offset where second stage is loaded
    Address_Stage_2     dd 07c00200h

; Number of times to retry disk operations
    Retry               db 03h

; Segment:Offset of bootstrap loader at startup
    Relocate_Source     dd 07c00000h

; Segment:Offset where bootstrap loader will be relocated
    Relocate_Destination dd 80000000h

; BIOS equipment status word result (int 11h)
    Equipment_Status    dw 0000h

;*****************************************************************************************
;* Some variables are needed to track the progress of loading the kernel from the        *
;* diskette into memory. (Also refer to the LOAD_IMAGE procedure)                        *
;*****************************************************************************************
; Destination segment where images are loaded (LOAD_IMAGE)
    D_Segment           dw 0000h

; Destination offset where images are loaded (LOAD_IMAGE)
    D_Offset            dw 0000h

; Segment:Offset in memory where disk buffer resides (LOAD_IMAGE)
    dsk_Buffer          dd 90000000h

; Initially, LOAD_IMAGE starts with head 1
    dsk_Head            db 01h

; Initially, LOAD_IMAGE starts with track 0
    dsk_Track           db 00h

; Total number of tracks read
    trk_Read            dw 0000h

; Total number of tracks to read
    trk_Total           dw 0000h

; Current sector position. The BIOS and FDC number disk sectors from 1, not 0
    sec_Current         db 01h

TIMES  510-($-$$) db 0                      ; Zero-out up to 510-byte boundary
    dw 0xAA55                               ;  and mark end of sector

;*****************************************************************************************
;*                          SECOND STAGE OF BOOTSTRAP LOADER                             *
;*                                                                                       *
;* The second stage of the bootstrap loader spans a number of sectors and contains code  *
;* to perform the following functions:                                                   *
;* - Setting up the boot table                                                           *
;* - Loading the runtime environment into memory                                         *
;* - Switching the processor to protected mode and transferring control to               *
;*   the runtime environment                                                             *
;*****************************************************************************************
    mov    ax, SEGMENT_2ND
    mov    ds, ax

;*****************************************************************************************
;* Gather some additional information and update the boot table                          *
;*****************************************************************************************
    xor    ax, ax
    mov    es, ax
```

```
    mov    di, BT_DISPLAY_BASE
    mov    eax, [Display_Base]
    shr    eax, 12
    mov    [es:di], eax                    ; Update boot table

    mov    ax, [Equipment_Status]
    and    eax, 00000001h                  ; Bit 0 of equipment status indicates
                                           ;  if a diskette drive is installed
    mov    di, BT_DISKETTE_INSTALLED
    mov    [es:di], eax                    ; Update boot table

    mov    ax, [Equipment_Status]
    shr    eax, 6
    and    eax, 00000003h                  ; Bits 6 and 7 of equipment status
                                           ;  contain the number of diskette drives
    inc    eax                             ; 00 - 1 drive installed when bit 0 was set
    mov    di, BT_DISKETTE_TOTAL
    mov    [es:di], eax                    ; Update boot table

    mov    ax, [Equipment_Status]
    shr    eax, 9
    and    eax, 00000007h                  ; Bits 9, 10 and 11 of equipment status
                                           ;  contain the number of serial adapters
    mov    di, BT_SERIAL_TOTAL
    mov    [es:di], eax                    ; Update boot table

;*****************************************************************************************
;* Retrieve base addresses for serial ports from BIOS data area and store the            *
;* information in the boot table                                                         *
;*****************************************************************************************
    mov    si, BIOS_COM4
    mov    di, BT_COM4
    mov    cx, 4

.read_serial:
    mov    bx, [es:si]
    and    ebx, 0000ffffh
    mov    [es:di], ebx
    sub    si, 2
    sub    di, 4
    dec    cx
    jnz    .read_serial

    mov    si, BIOS_DISPLAY_ADR_REG
    mov    ax, [es:si]
    mov    di, BT_DISPLAY_ADR_REG
    mov    [es:di], ax                     ; Update boot table

;*****************************************************************************************
;* Memory detection using three different methods:                                       *
;* - INT 15h, function 88h should be supported by all PC's. Unfortunately, this function *
;*   only reports the amount of extended memory between 1Mb and 64Mb and on some systems *
;*   it will only report the amount between 1Mb and 15Mb                                 *
;* - INT 15h, function e801h reports the number of 1K blocks between 1 and 16Mb in AX    *
;*   and the number of 64K blocks above 16Mb in BX. This function is only supported by   *
;*   newer BIOS implementations (Phoenix v4.0 and AMI 8/23/94 or later for example)      *
;* - INT 15h, function e820h forms part of Advanced Configuration and Power Management  *
;*   Interface (ACPI) and is only supported by newer BIOS implementations. This function *
;*   returns a memory map (20 bytes at a time) that can be used to isolate contigious    *
;*   blocks for use by operating systems                                                 *
;*****************************************************************************************
    mov    ah, BIOS_GET_EXT_MEMORY
    int    15h
    mov    di, BT_MEM_88H
    xor    bx, bx
    mov    es, bx
    and    eax, 0000ffffh
```



```
    mov    [es:di], eax                    ; Update boot table

    mov    di, BT_MEM_E801H
    mov    dword [es:di], 0                ; Entry contains 0 if not supported
    mov    ax, BIOS_GET_EXT_MEMORY_64
    int    15h
    jc     .smap                           ; Try function e820h if not supported

    and    eax, 0ffffh
    and    ebx, 0ffffh
    shl    ebx, 6                          ; Convert 64K blocks to 1K blocks
    add    eax, ebx
    mov    di, BT_MEM_E801H
    mov    [es:di], eax                    ; Update boot table

.smap:
    mov    eax, BIOS_GET_SMAP
    mov    edx, SMAP_ID                    ; EDX = "SMAP"
    mov    ebx, 0                          ; EBX = Continuation value, 0 indicates
                                           ;  that operation starts at the beginning of
                                           ;  of the memory map
    mov    ecx, 20                         ; ECX = buffer size, minimum value is 20 bytes
    les    di, [smap_ptr]                  ; ES:DI = pointer to buffer used for result
    int    15h

.smap_loop:
    jc     .smap_done
    cmp    eax, SMAP_ID
    jne    .smap_done
    cmp    ecx, 20
    jl     .smap_done

    add    di, 20
    inc    dword [smap_count]
    cmp    dword [smap_count], MAX_SMAP_ENTRIES
    jge    .smap_done

    cmp    ebx, 0
    je     .smap_done

    mov    eax, BIOS_GET_SMAP
    mov    edx, SMAP_ID
    int    15h
    jmp    .smap_loop

.smap_done:
    mov    di, BT_MEM_SMAP_COUNT
    mov    eax, [smap_count]
    mov    [es:di], eax                    ; Update the boot table

;*****************************************************************************************
;* Load the runtime environment into memory                                              *
;*****************************************************************************************
    ClearDisplay BG_BLACK | FG_WHITE
    WriteString Load_Msg1
    call   WRITE_LN
    mov    word [D_Segment], KERNEL_SEGMENT
    mov    word [D_Offset], KERNEL_OFFSET
    mov    word [trk_Total], TRKS_KERNEL
    call   LOAD_IMAGE                      ; Load the runtime system into memory
    call   WRITE_LN

    mov    ax, KERNEL_SEGMENT
    mov    es, ax
    mov    eax, [es:KERNEL_OFFSET+LINK_BASE]
    cmp    eax, ENTRY_POINT
    je     setup_kernel_stack
```

```
468
469     WriteString Error_Msg3
470 .link_base_error:
471     jmp     .link_base_error
472
473 ;*****************************************************************************
474 ;* Create the stack for the runtime environment. The stack is aligned to the nearest 4K   *
475 ;* boundary above the runtime system. For example, consider a runtime system (5724 bytes  *
476 ;* in size) linked at 11000h. The runtime system will stop at address 1265Ch. The nearest *
477 ;* 4K boundary is located at address 13000h. Given a 16K stack for the runtime system     *
478 ;* and the NULL page that separates the kernel and its stack, the new stack top will      *
479 ;* become 18000h (13000h+4000h+1000h).                                                    *
480 ;*****************************************************************************
481 setup_kernel_stack:
482     mov     ax, KERNEL_SEGMENT
483     mov     es, ax
484     mov     eax, [es:KERNEL_OFFSET+HEAP_START] ; Determine linear address where runtime
485     add     eax, 0fffh                         ;  system stops and align it on a 4K boundary
486     mov     ebx, 0fffh
487     not     ebx
488     and     eax, ebx
489     add     eax, KERNEL_STACK_SIZE             ; Add the kernel stack size to the aligned
490     add     eax, 4096                          ;  address and allocate room for the bottom
491                                                ;  NULL page. The runtime system will be
492                                                ;  responsible for allocating the top
493                                                ;  NULL page
494
495     xor     bx, bx
496     mov     es, bx
497     mov     bx, BT_KERNEL_STACK_TOP
498     mov     [es:bx], eax                       ; Update the boot table
499     mov     bx, BT_KERNEL_STACK_SIZE
500     mov     edx, KERNEL_STACK_SIZE
501     mov     [es:bx], edx                       ; Update the boot table
502
503     mov     ebx, eax
504     and     eax, 0ffffh
505     mov     sp, ax                             ; Convert the linear address back into
506     and     ebx, 0f0000h                       ;  a segment:offset pair since we are still
507     shr     ebx, 4                             ;  running in real mode
508     mov     ss, bx
509
510 ;*****************************************************************************
511 ;* The motor of the disk drive may still be on and the BIOS may not switch it off in time *
512 ;* when the switch to protected mode occurs. The motor may be stopped by sending a        *
513 ;* command to the digital output register (DOR) of the floppy disk controller (FDC). The  *
514 ;* DOR is an 8 bit register and has the following format:                                 *
515 ;*                                                                                       *
516 ;*     7      6      5      4      3      2      1      0                                 *
517 ;*  +------+------+------+------+------+------+------+------+                            *
518 ;*  | MOTD | MOTC | MOTB | MOTA | DMA  | REST | DR1  | DR0  |                            *
519 ;*  +------+------+------+------+------+------+------+------+                            *
520 ;*                                                                                       *
521 ;* MOTD, MOTC, MOTB and MOTA control the status of the drive motor (1 = start, 0 = stop), *
522 ;* DR1 and DR0 control drive selection (00 = Drive A, 01 = Drive B, etc). The REST bit    *
523 ;* must be set to allow the controller to accept commands                                 *
524 ;*****************************************************************************
525     mov   dx, FDC_DOR1
526     mov   al, FDC_DOR_REST+FDC_DOR_DMA         ; Reset controller, DMA/IRQ enabled
527     out   dx, al
528
529 ;*****************************************************************************
530 ;* STEP 1                                                                    *
531 ;* Disable interrupts by clearing the interrupt flag (IF = 0). NMI's are not disabled     *
532 ;*****************************************************************************
533     mov     si, PM_Switch_Msg1
534     call    WRITE_STRING
```



```
535     call    WRITE_LN
536     cli
537
538 ;*****************************************************************************
539 ;* STEP 2                                                                    *
540 ;* Activate address line 20 (A20). The output of this line is gated through the 8042      *
541 ;* keyboard controller. However, the keyboard input buffer must be empty before A20 may   *
542 ;* be gated on. A simple test is also performed to determine if A20 is really enabled     *
543 ;* because some systems can be problematic, especially those using an MCA bus or          *
544 ;* if the A20 gating option in the CMOS is set to fast. If the test fails, an alternative *
545 ;* method using the System Control port (92h) is applied before testing A20 again. An     *
546 ;* error message is displayed if this method also fails.                                  *
547 ;*****************************************************************************
548     mov     si, PM_Switch_Msg2
549     call    WRITE_STRING
550     call    WRITE_LN
551
552     call    FLUSH_KBD_BUFFER
553     mov     al, KBD_CMD_WRITE_OUTPUT           ; The next data byte written to port 60h
554                                                ;  will be written to the micro-controller
555                                                ;  output port
556     out     KBD_STATUS_PORT, al
557     call    FLUSH_KBD_BUFFER
558     mov     al, 0dfh                           ; Enable A20
559     out     KBD_INPUT_PORT_0, al
560     call    FLUSH_KBD_BUFFER
561
562     call    TEST_A20
563     jnc     .init_idt
564
565     in      al, SYSTEM_CONTROL_PORT
566     or      al, A20_STATUS                     ; Enable A20
567     and     al, 0feh                           ; Mask out the reset bit
568     out     SYSTEM_CONTROL_PORT, al
569
570     call    TEST_A20
571     jnc     .init_idt
572
573     mov     si, PM_A20_Failed_Str
574     call    WRITE_STRING
575     call    WRITE_LN
576 .a20_error:
577     jmp     .a20_error
578
579 ;*****************************************************************************
580 ;* STEP 3                                                                    *
581 ;* Create an empty interrupt descriptor table (IDT) by loading the address of the table   *
582 ;* into the interrupt descriptor table register (IDTR). Interrupts generated when an      *
583 ;* empty IDT is present will cause a shutdown.                                            *
584 ;*****************************************************************************
585 .init_idt:
586     mov     si, PM_Switch_Msg3
587     call    WRITE_STRING
588     call    WRITE_LN
589
590     db 66h                                     ; Ensure that a 32-bit instruction is executed
591     lidt    [ptrIDT]                           ;  and load the IDTR
592
593 ;*****************************************************************************
594 ;* STEP 4                                                                    *
595 ;* Initialise the global descriptor table (GDT) by loading the global descriptor table    *
596 ;* register (GDTR). The GDTR is a 48-bit register consisting out of a 16-bit limit and    *
597 ;* 32-bit base address. The limit has been pre-computed and only the base address must be *
598 ;* set.                                                                                   *
599 ;*****************************************************************************
600     mov     si, PM_Switch_Msg4
601     call    WRITE_STRING
```

```
602     call   WRITE_LN
603
604     xor    eax, eax
605     mov    ax, cs
606     shl    eax, 4
607     mov    ebx, GDT
608     add    eax, ebx                        ; EAX contains physical address of GDT
609     mov    dword [ptrGDT+2], eax           ; Store base address
610     db 66h                                 ; Ensure that a 32-bit instruction is executed
611     lgdt   [ptrGDT]                        ;  and load the GDTR
612
613 ;*****************************************************************************************
614 ;* STEP 5                                                                                *
615 ;* Store the address of the entry point to the system on the stack and compute value of   *
616 ;* ESP in EDX.                                                                           *
617 ;*****************************************************************************************
618     mov    si, PM_Switch_Msg5
619     call   WRITE_STRING
620     call   WRITE_LN
621
622     push   word GDT_CODE_SELECTOR
623     push   dword ENTRY_POINT               ; Store system entry point on the stack
624
625     xor    edx, edx
626     mov    dx, ss
627     shl    edx, 4
628     xor    eax, eax
629     mov    ax, sp
630     add    edx, eax                        ; Store linear address of stack in EDX
631
632 ;*****************************************************************************************
633 ;* STEP 6                                                                                *
634 ;* Swith to protected mode by settig the Protection Enable bit (PE) and executing a JMP  *
635 ;* instruction to serialize the CPU. There are two ways of doing the switch:             *
636 ;* 1) The traditional 80286 method using SMSW and LMSW:                                  *
637 ;*       smsw ax                                                                         *
638 ;*       or   al, 1                                                                      *
639 ;*       lmsw ax                                                                         *
640 ;*       jmp  continue                                                                   *
641 ;* 2) Setting the bit directly in control register 0 (CR0). The lower 16 bits of CR0     *
642 ;*    contains the old Machine Status Word. Control registers can only be accessed       *
643 ;*    starting with the 80386 processor. At this point it has already been confirmed     *
644 ;*    that at least an 80386 CPU is present, and consequently this method is used.       *
645 ;*****************************************************************************************
646     mov    eax, cr0
647     or     eax, CR0_PROTECTION_ENABLE      ; Set PE bit
648     mov    cr0, eax
649     jmp    .serialize                      ; Serialize CPU
650
651 .serialize:
652     mov    ax, GDT_DATA_SELECTOR           ; Reload all the segment selectors
653     mov    ds, ax
654     mov    ss, ax
655     mov    es, ax                          ; ES, FS and GS can also be loaded with the
656     mov    fs, ax                          ;  null descriptor if they are not going to
657     mov    gs, ax                          ;  be used
658     mov    esp, edx
659
660     db 66h                                 ; Ensure that a 32-bit instruction is
661     retf                                   ;  generated and return to entry point
662
663 ;*****************************************************************************************
664 ;* PROCEDURE TEST_A20                                                                    *
665 ;* INPUT: none                                                                           *
666 ;* Determine if A20 line is active. CF is set if A20 is not enabled                      *
667 ;*****************************************************************************************
668 TEST_A20:
```



```
669     mov    di, 10h
670     mov    ax, 0ffffh
671     mov    es, ax
672     xor    ax, ax
673     mov    si, ax
674     mov    fs, ax
675
676     mov    ax, [es:di]
677     cmp    ax, [fs:si]
678     jne    .a20_done
679
680 .a20_unknown:
681     not    word [fs:si]
682     mov    ax, [es:di]
683     cmp    ax, [fs:si]
684     je     .a20_done
685     stc                                    ; Set CF to indicate error
686     not    word [fs:si]
687     ret
688
689 .a20_done:
690     clc
691     not    word [fs:si]
692     ret
693
694 ;*****************************************************************************************
695 ;* PROCEDURE: FLUSH_KBD_BUFFER                                                           *
696 ;* INPUT: none                                                                           *
697 ;* Waits until the keyboard buffer is empty                                              *
698 ;*****************************************************************************************
699 FLUSH_KBD_BUFFER:
700     in     al, KBD_STATUS_PORT             ; Read Keyboard status port
701     and    al, KBD_BUFFER_FULL             ; Test if input buffer full flag (bit 2)
702     jnz    FLUSH_KBD_BUFFER                ;  is set
703     ret
704
705 ;*****************************************************************************************
706 ;* PROCEDURE: LOAD_IMAGE                                                                 *
707 ;* INPUT: D_Segment, D_Offset, dsk_Track, dsk_Head, trk_Total                             *
708 ;* Loads an image from disk into memory starting at the address specified in D_Segment   *
709 ;* and D_Offset. The operation starts at the first sector of the track specified in      *
710 ;* dsk_Track using the head number supplied in dsk_Head. The number of tracks to be read *
711 ;* are specified in trk_Total.                                                           *
712 ;*****************************************************************************************
713 LOAD_IMAGE:
714     cld
715     mov    byte [trk_Read], 0              ; Initialize variables for loading
716
717 .while0:
718     mov    ax, [trk_Read]
719     cmp    ax, [trk_Total]                 ; WHILE trkRead < trkTotal DO
720     jl     .continue1
721     jmp    .end0
722
723 .continue1:
724     mov    byte [sec_Current], 1           ;   secCurrent := 1;
725     mov    byte [Retry], 3
726
727 .while1:
728     cmp    byte [sec_Current], SECTOR_PER_TRACK    ;   WHILE secCurrent < secPerTrack DO
729     jl     .read
730     jmp    .end1
731
732 .read:                                     ;     REPEAT
733     call   RESET_DISK_DRIVE
734     mov    ah, BIOS_READ_SECTOR
735     mov    al, SECTOR_PER_READ
```

```
736     les    bx, [dsk_Buffer]
737     mov    ch, [dsk_Track]
738     mov    cl, [sec_Current]
739     mov    dh, [dsk_Head]
740     mov    dl, 0
741     int    13h                                ;       Read data into disk buffer
742     jnc    .copy
743     dec    byte [Retry]
744     jnz    .read                              ;     UNTIL (Result = 0) OR (Retry = 0)
745                                               ;     IF Result # 0 THEN
746     push   eax
747     mov    esi, Error_Msg2
748     call   WRITE_STRING                       ;       Display error message
749
750     pop    eax
751     and    eax, 0ff00h
752     shr    eax, 8
753     call   WRITE_HEX
754
755 .disk_error:
756     jmp   .disk_error                         ;       LOOP END
757
758 ; The data can either fit within the current segment (.copy2) or else the data must be
759 ; transferred using two operations (.copy1 and .copy2). This first operation will fill the
760 ; current segment before the remainder of the disk buffer is transferred to the following
761 ; segment. It is also possible that the complete buffer must be copied to a new
762 ; segment, so D_Offset and D_Segment are first adjusted to compensate for this.
763 .copy:                                        ;     ELSE
764     mov    al, 0b1h
765     call   WRITE_CHAR
766     mov    bx, 0fffh
767     sub    bx, [D_Offset]
768     inc    bx                                 ;       n := 0FFFFH-D_Offset+1;
769     jnz    .copy1                             ;       IF n = 0 THEN
770     mov    [D_Offset], bx                     ;         D_Offset := n;
771     add    word [D_Segment], 1000h            ;         INC(D_Segment, 1000H)
772                                               ;       END;
773 .copy1:
774     mov    di, [D_Offset]
775     mov    ax, [D_Segment]
776     mov    es, ax
777     cmp    bx, 0
778     je     .copy2
779     cmp    bx, BUFFER_SIZE                    ;       IF (n # 0) & (n < bufSize) THEN
780     jae    .copy2
781
782     push   ds
783     mov    cx, bx
784     lds    si, [dsk_Buffer]
785     shr    cx, 2
786     rep    movsd                              ;         Copy data from disk buffer to
787                                               ;         the destination buffer
788     pop    ds
789     add    word [D_Segment], 1000h            ;         INC(D_Segment, 1000H);
790     mov    cx, BUFFER_SIZE
791     sub    cx, bx
792     push   ds
793     xor    di, di                             ;         D_Offset := 0;
794     mov    ax, [D_Segment]
795     mov    es, ax
796     lds    si, [dsk_Buffer]
797     add    si, bx
798     shr    cx, 2
799     rep    movsd                              ;         Copy data from disk buffer to
800                                               ;         to destination buffer
801     pop    ds
802
```



```
803     mov    word [D_Offset], BUFFER_SIZE
804     sub    word [D_Offset], bx                ;         D_Offset := BufSize-n;
805     jmp    .inc2
806
807 .copy2:                                       ;       ELSE
808     push   ds
809     mov    cx, BUFFER_SIZE_4
810     lds    si, [dsk_Buffer]
811     rep    movsd                              ;         Copy whole disk buffer to
812                                               ;         to destination buffer
813     pop    ds
814     add    word [D_Offset], BUFFER_SIZE       ;         INC(D_Offset, bufSize)
815                                               ;       END
816                                               ;     END;
817 .inc2:
818     add    byte [sec_Current], SECTOR_PER_READ ;     INC(secCurrent, secPerRead)
819     jmp    .while1                            ;   END;
820
821 .end1:
822     mov    al, [dsk_Head]
823     add    [dsk_Track], al                    ;   INC(dskTrack, dskHead);
824     xor    byte [dsk_Head], 1
825     add    byte [trk_Read], al                ;   INC(trkRead)
826     jmp    .while0                            ; END
827
828 .end0:
829     ret
830
831 DATASEG2
832     Load_Msg1         db 'Loading runtime system...', 0
833     Error_Msg3        db 'ERROR: Invalid link base in image', 0
834     PM_Switch_Msg1    db 'Disabling interrupts', 0
835     PM_Switch_Msg2    db 'Activating A20', 0
836     PM_Switch_Msg3    db 'Initializing IDT', 0
837     PM_Switch_Msg4    db 'Initializing GDT', 0
838     PM_Switch_Msg5    db 'Setting entry point and new stack', 0
839
840     PM_A20_Failed_Str db 'A20 not enabled',0
841
842     smap_ptr          dd 00000600h
843     smap_count        dd 0
844
845 ;*******************************************************************************************
846 ;* Segment descriptors are 8 bytes (64 bits) long and every descriptor consists out of    *
847 ;* three parts: the segment base address, segment limit and segment attributes. The       *
848 ;* segment base is 32-bits in size, the segment limit is also 32 bits in size, but only   *
849 ;* 20 bits are used. If the byte granularity is used, only small limits can be specified, *
850 ;* while 4K granularity allows for the addition of large segment limits (See Crawford,    *
851 ;* chapter 5).                                                                           *
852 ;*                                                                                       *
853 ;*  Example (byte granularity)                                                           *
854 ;*    Base  = 12340000                                                                   *
855 ;*    Limit = 0                                                                          *
856 ;*    This example describes a segment consisting of 1 byte stored at                    *
857 ;*    linear address 12340000                                                            *
858 ;*                                                                                       *
859 ;*  Example (4K granularity)                                                             *
860 ;*    Base = 12340000                                                                    *
861 ;*    Limit = 0                                                                          *
862 ;*    This example describes a segment consisting of 4K of memory                        *
863 ;*    starting at linear address 12340000 and ending at address 12340FFF                 *
864 ;*                                                                                       *
865 ;*  The layout of a segment descriptor is shown below                                    *
866 ;*                                                                                       *
867 ;* 63            55                  40 39               16 15              0          *
868 ;* +-----------+ +-------------------+ +-----------------+ +---------------+          *
869 ;* |           | |                   | |                 | |               |          *
```

```
;* | 31..24 of | | Segment Attributes | |      0..23      | |   0..15 of    |            *
;* |  segment  | |                    | | of segment base | | segment limit |            *
;* |   base    | |                    | |                 | |               |            *
;* +-----------+ +--------------------+ +-----------------+ +---------------+            *
;*                                                                                       *
;* The segment attribute is 16 bits long and divided into the following fields:          *
;*                                                                                       *
;*  15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0                      *
;* +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+                    *
;* |   |   |   | A |     19..16    |   |       | D |               |                    *
;* | G | D | 0 | V |   of segment  | P |  DPL  | T |      TYPE     |                    *
;* |   |   |   | L |     limit     |   |       | 1 |               |                    *
;* +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+                    *
;*                                                                                       *
;* G    - Granularity attribute. Set to 1 implies 4K granular limit                     *
;* D    - 1 for 80386, 0 for 80286. The interpretation of this bit depends              *
;*        on the type of the segment:                                                    *
;*         1 - Executable segments use the D bit to determine the                      *
;*             default operand size. D=1 defaults to 32-bit addresses and              *
;*             32-bit and 8-bit operands. D=0 defaults to 16-bit addresses             *
;*             and 16-bit and 8-bit operands                                           *
;*         2 - Expand down segments use this bit to determine the upper limit          *
;*             D=1 indicates a 4G upperlimit. D=0 indicates a 64K upper limit          *
;*         3 - Segments addressed by SS use this bit to determine whether              *
;*             to use ESP (D=1) or SP (D=0) for explicit stack reference               *
;*             by way of PUSH and POP instructions                                     *
;* AVL  - Availabe-for-software. Value of this bit is not interpreted by               *
;*        80386 and Intel promises that future compatible processors will not          *
;*        define a use for this bit                                                    *
;* P    - Present bit                                                                  *
;* DPL  - Descriptor Privilege Level. Value defines the privilege level                *
;*        associated with the segment                                                  *
;* DT   - Distinguishes between system segments (DT=1) and gates (DT=0)                *
;* TYPE - Defines the type of the memory descriptor.                                   *
;***************************************************************************************

;***************************************************************************************
;* Global Descriptor Table (GDT) - This GDT is only a temporary table and will be      *
;* replaced by the runtime system's own GDT. However, a GDT is necessary to operate in *
;* protected mode until the runtime system's own GDT is created. The following         *
;* descriptors are created:                                                            *
;*                                                                                     *
;*  DESCRIPTOR NAME    BASE  LIMIT  ATTRIBUTES                                         *
;*  null descriptor    0     0      n/a                                                *
;*  code segment       0     FFFFF  TYPE = Execute/Read, 4K granular limit, DPL = 0    *
;*  data segment       0     FFFFF  TYPE = Read/Write, 4K granular limit, DPL = 0      *
;***************************************************************************************

ALIGN 4
  GDT TIMES 8 db 0                                    ; Null descriptor
              db 0ffh, 0ffh, 0, 0, 0, 9ah, 0cfh, 0    ; Code segment descriptor
              db 0ffh, 0ffh, 0, 0, 0, 92h, 0cfh, 0    ; Data segment descriptor

  GDT_SIZE              EQU $-GDT

;***************************************************************************************
;* Both the GDTR and LDTR are composed of two parts. A 16-bit limit and a 32-bit base  *
;* address.                                                                            *
;***************************************************************************************
  ptrGDT dw GDT_SIZE-1, 0, 0                ; 48-bit pointer to GDT
  ptrIDT dw 0, 0, 0                         ; 48-bit null pointer to IDT

TIMES  ((BOOTSTRAP_SECTORS+1)*SECTOR_SIZE)-($-$$) db 0
```