The Operating System Initialization in ix86 Architecture

Operating Systems I

UFSC

Rafael Luiz Cancian, Dr. Eng.

LISHA

Introduction

Initialization of software is one of first issues for OS development.

Initialization of software is fully dependable on target architecture.

In ix86, initialization has additional tasks, as changing processing modes.

Tasks for initialization are closely related to memory management and processor control.

1. System Registers

The registers designed for use by systems programmers fall into these classes:

- EFLAGS
- Memory Management Registers
- Control Registers
- Debug Registers
- Test registers

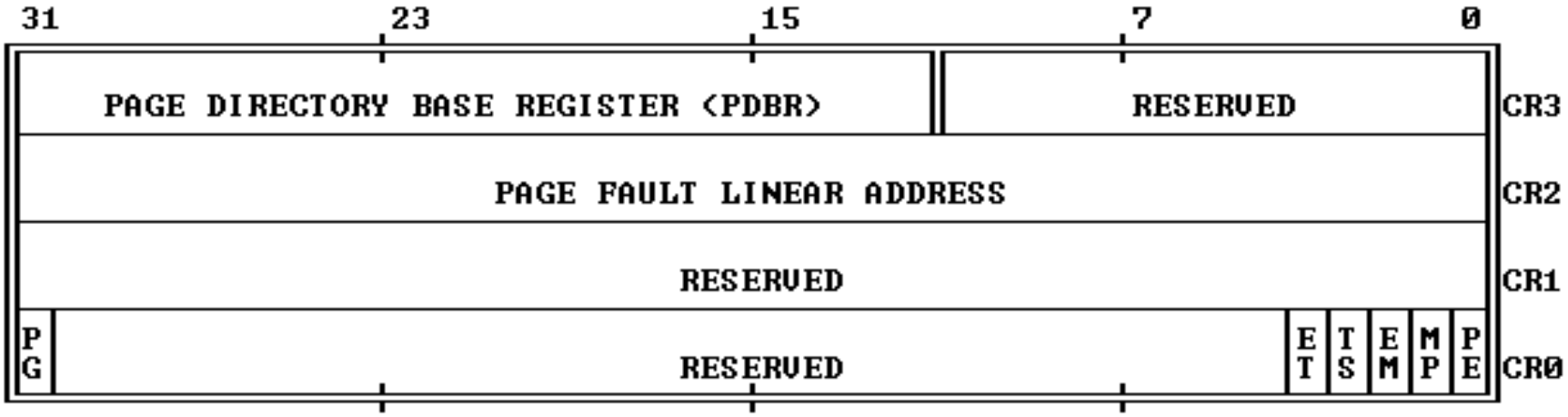
Most important for initialization software are Memory Management and Control Registers

1. System Registers

Memory Management Registers include GDTR, LDTR, IDTR and TR registers.

Control Registers include CR0 (most important), CR2, and CR3

Figure 4-2. Control Registers



2. Memory Management

Memory Management consists of segmentation and paging.

Segmentation is used to give each program several independent and protected address spaces.

Paging is used to support an environment where large address spaces are simulated using a small amount of RAM and some disk storage.

Segmentation hardware translate a segmented logical address into a linear address, while paging translates linear address into a physical address.

2. Memory Management

Figure 5-2. Segment Translation

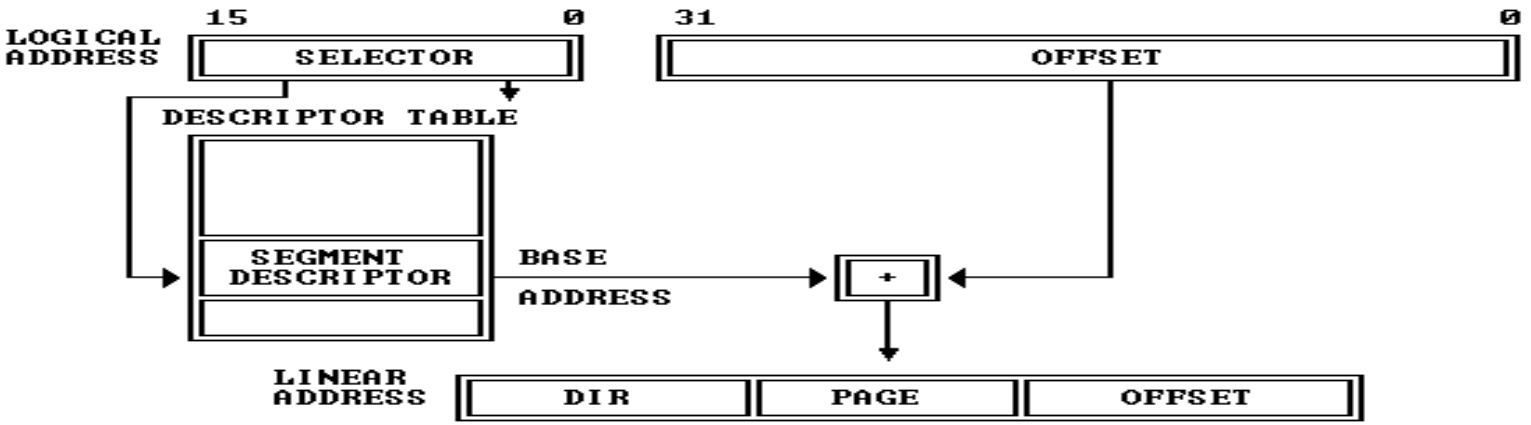
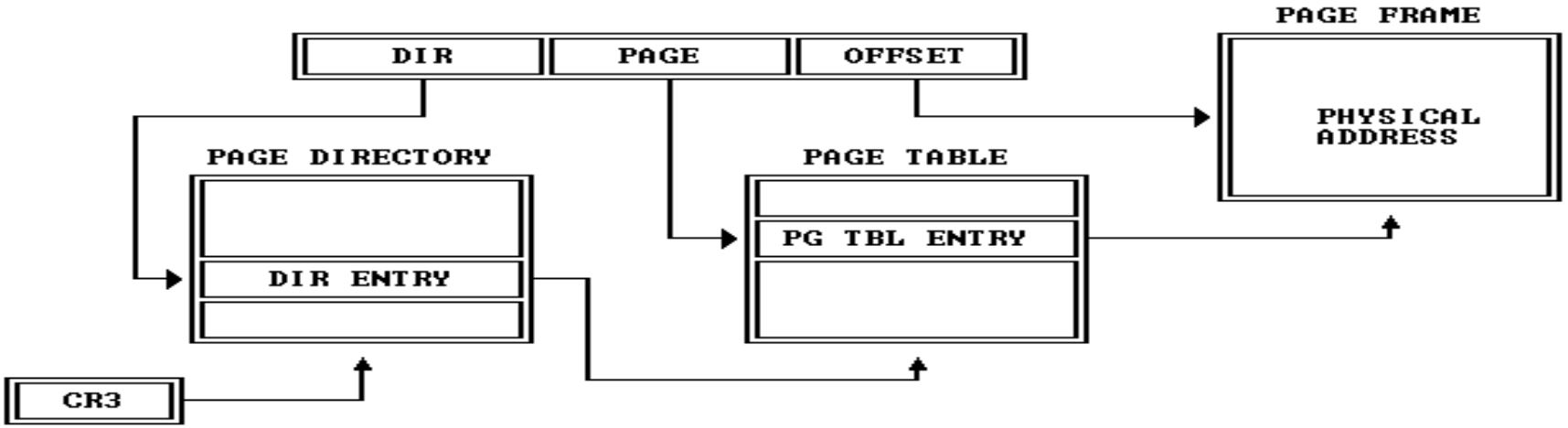


Figure 5-9. Page Translation



2.1 Segmentation Model

To perform segmentation translation, the processor uses the following data structures:

- Segment Registers
- Segment Selectors
- Segment Descriptors
- Segment Descriptor Tables

2.1.1 Segment Registers

Each memory reference is associated with a segment register.

Code, data and stack references access the segments specified by the contents of theirs segment registers.

Every segment register has a “visible” part and an “invisible part” (TLB alike).

Segment registers are CS, DS, ES, FS, GS, SS

2.1.2 Segment Registers

Some instructions change the visible part. The invisible part is loaded by the processor.

When these instructions are used, the visible part of the segment register is load with a “segment selector”.

The processor automatically fetches the base address, limit, type and other information from the descriptor and loads the invisible part of the segment register.

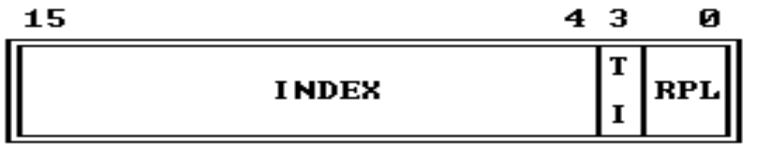
2.1.3 Segment Selectors

A segment selector points to the information which defines a segment, called a segment descriptor.

The segment selector identifies a segment descriptor by specifying a descriptor table (TI field) and a descriptor within that table (Index Field)

The segment selector also specifies a Requester Privilege Level (RPI), related to current task.

Figure 5-6. Format of a Selector



TI – TABLE INDICATOR
RPL – REQUESTOR'S PRIVILEGE LEVEL

2.1.4 Segment Descriptors

A segment descriptor is a data structure in memory which provides the processor with the size and location of a segment, as well as control and status information.

There are several types of segment descriptors, classified as application or special system descriptors.

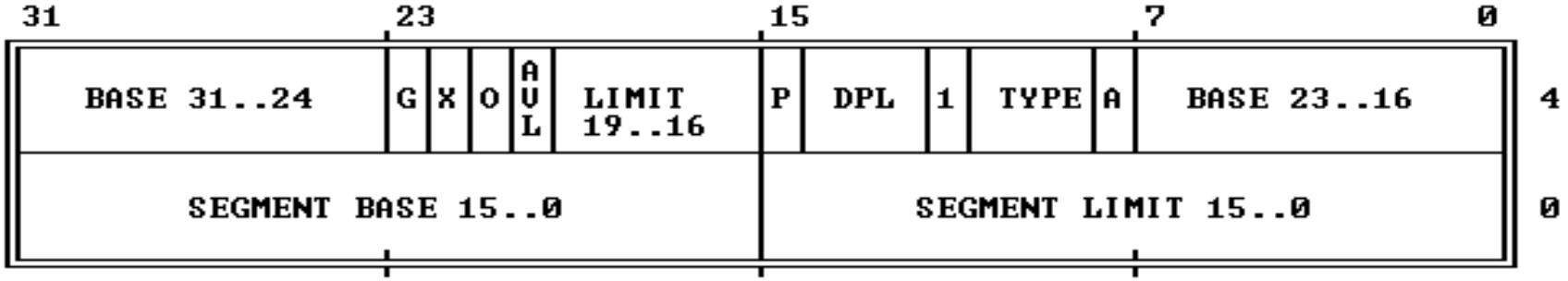
Application descriptors describe code, data and stack segments in memory.

Special system descriptors describe TSS, Interrupt, Task or Trap Gates, amount others.

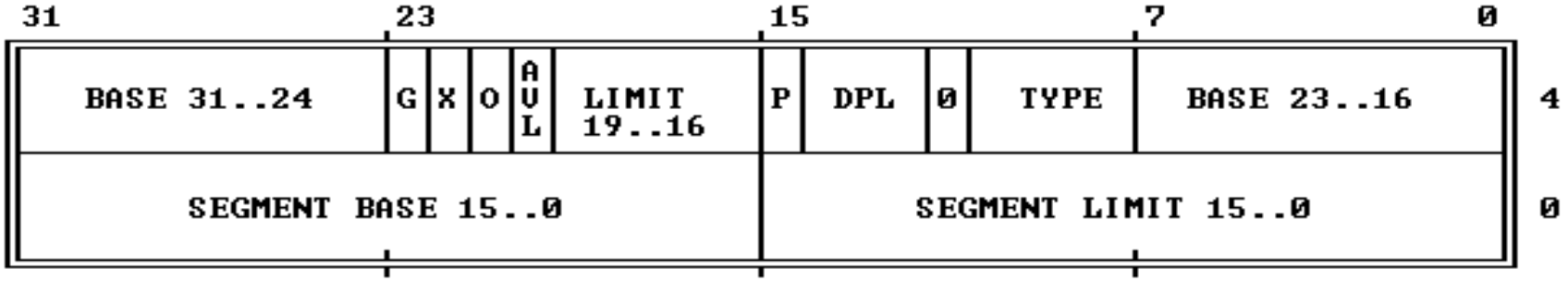
2.1.4 Segment Descriptors

Figure 5-3. General Segment-Descriptor Format

DESCRIPTORS USED FOR APPLICATIONS CODE AND DATA SEGMENTS



DESCRIPTORS USED FOR SPECIAL SYSTEM SEGMENTS



A - ACCESSED
AVL - AVAILABLE FOR USE BY SYSTEMS PROGRAMMERS
DPL - DESCRIPTOR PRIVILEGE LEVEL
G - GRANULARITY
P - SEGMENT PRESENT

2.1.5 Segment Descriptor Tables

A segment descriptor table is an array of segment descriptors. There are two kinds of descriptor tables:

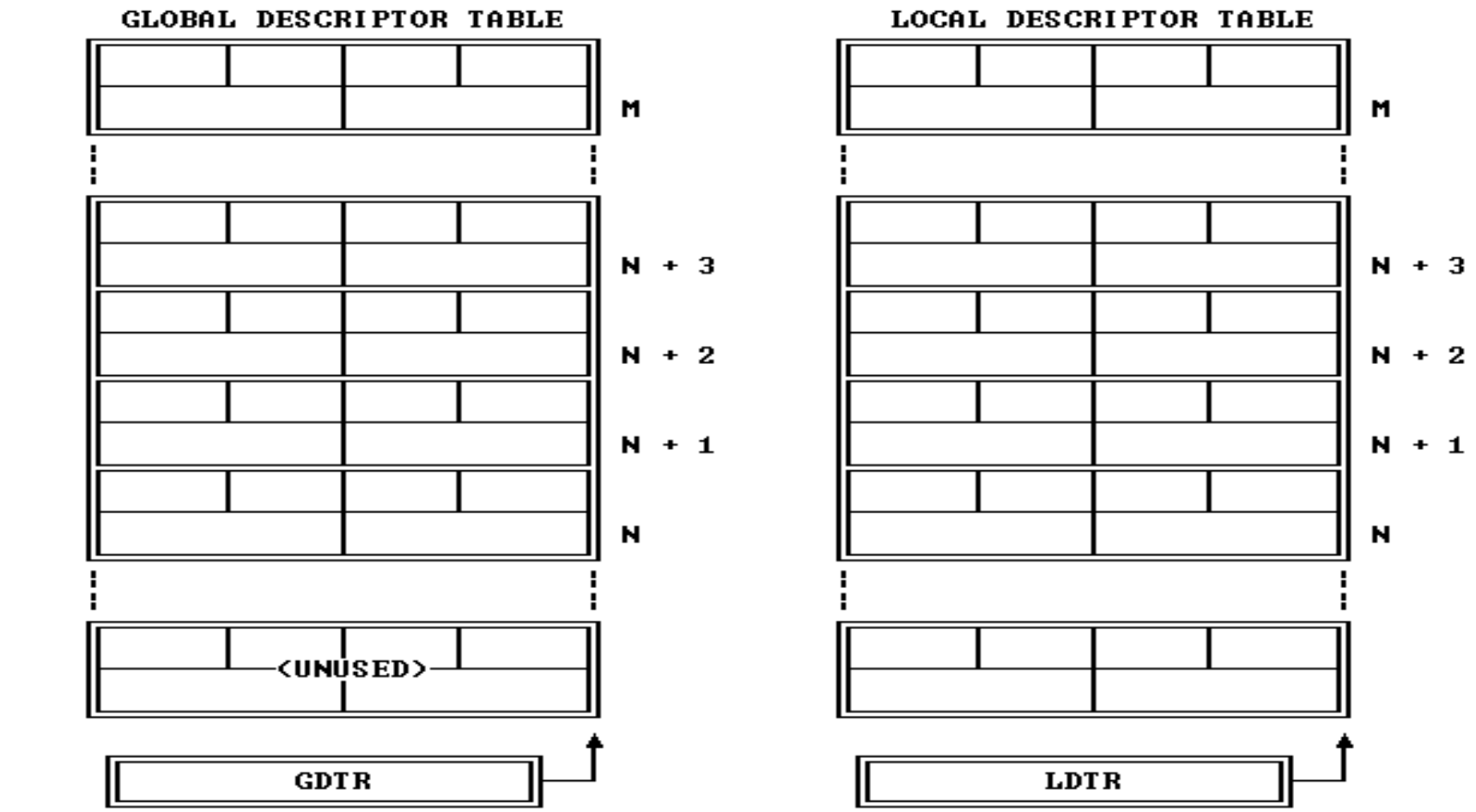
- The global descriptor table (GDT)
- The local descriptor table (LDT)

The is only one GDT for all tasks and and LDT for each task being run.

The first descriptor in the GDT is not used by the processor.

2.1.5 Segment Descriptor Tables

Figure 5-5. Descriptor Tables



2.1.6 Descriptor Table Base Registers

The processor finds the GDT and LDT using the GDTR and LDTR registers.

These registers hold 32-bit base addresses for table in linear address space. They also hold 16-bit limit for the size of these table.

The instructions LGDT, SGDT, LIDT, SIDT are used to load and store value in the GDTR and IDTR registers.

A third register, IDTR holds information about the IDT and can be handled using LIDT ans SIDT instructions.

2.2 Paging Model

Paging is different from segmentation through its use of small, fixed-size pages.

Pages are always 4KB (two level paging) or 4MB (one level paging).

The information which maps linear addresses into physical addresses and exceptions is held in data structures in memory called page tables.

The paging mechanisms treats the 32-bit linear address as having three parts, two 10-bit indexes into the page tables and a 12-bit offset into the page addressed by the page tables.

2.2 Paging Model

The CR3 register (usually called PDBR) holds the page frame address of the page table.

The upper 10 bits of linear address are scaled by four and added to the value in the PDBR register to get the physical address of an entry in page directory.

When the entry in the page directory is accessed, a number of checks are performed.

2.2 Paging Model

Figure 5-8. Format of a Linear Address

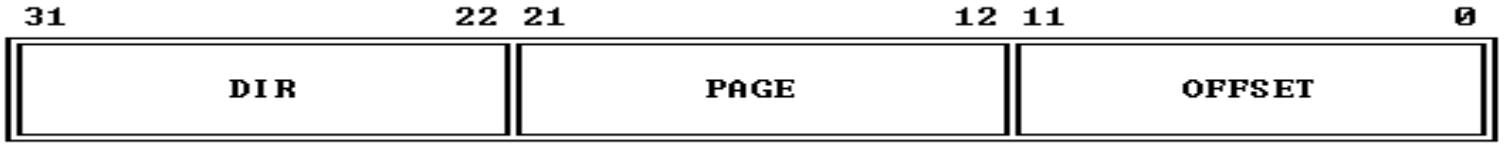
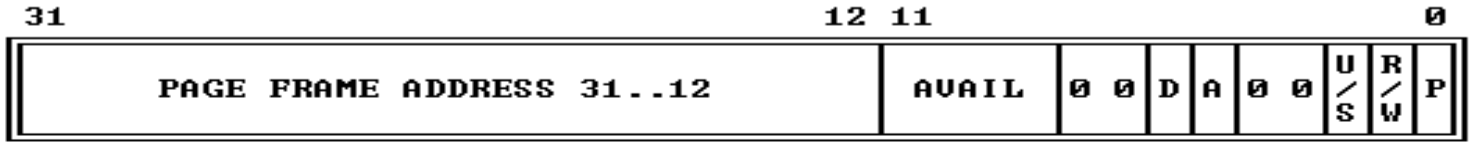


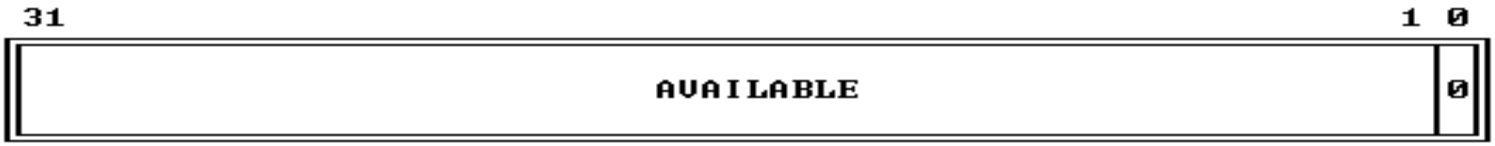
Figure 5-10. Format of a Page Table Entry



P — PRESENT
R/W — READ/WRITE
U/S — USER/SUPERVISOR
D — DIRTY
AVAIL — AVAILABLE FOR SYSTEMS PROGRAMMER USE

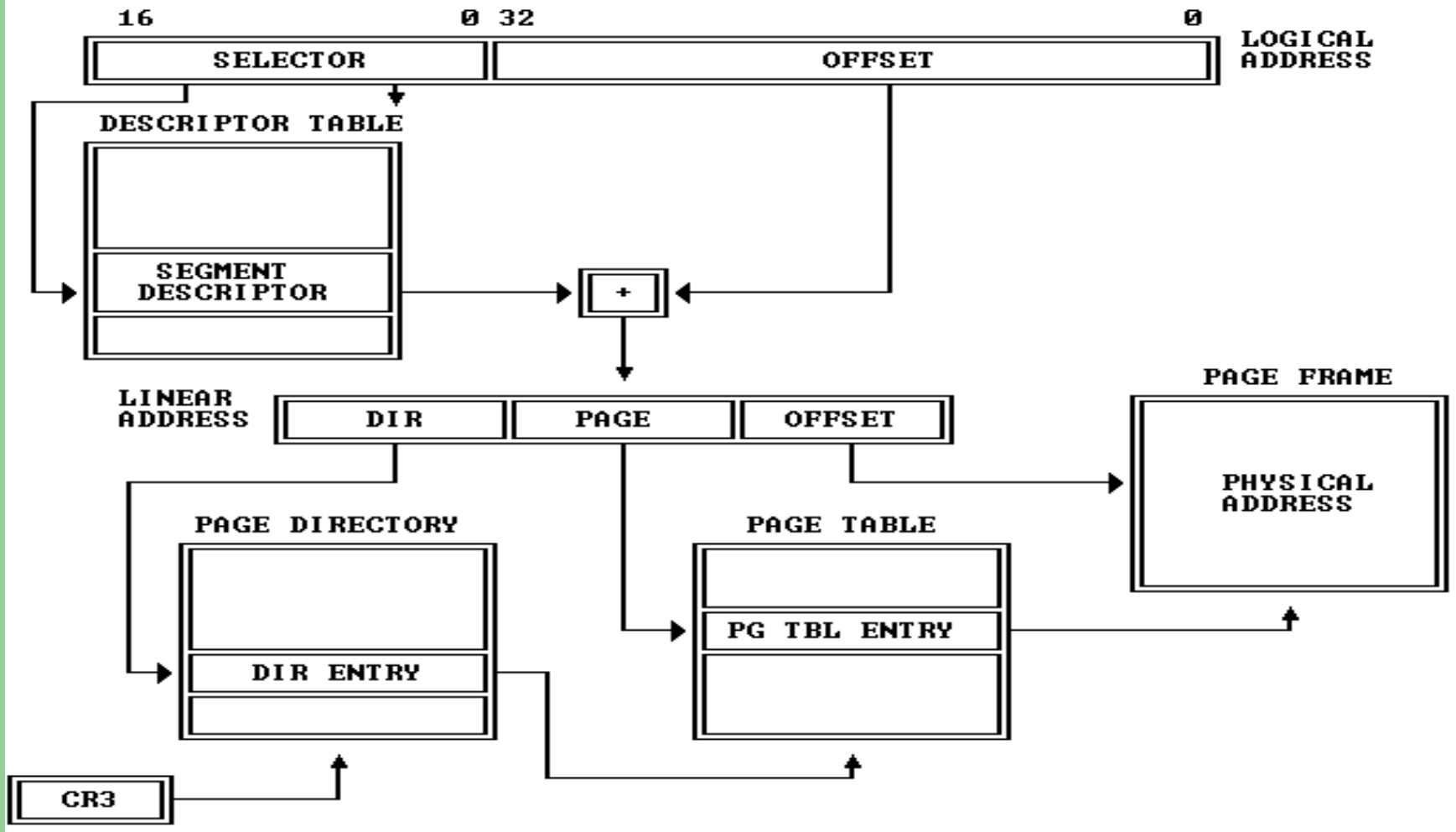
NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.

Figure 5-11. Invalid Page Table Entry



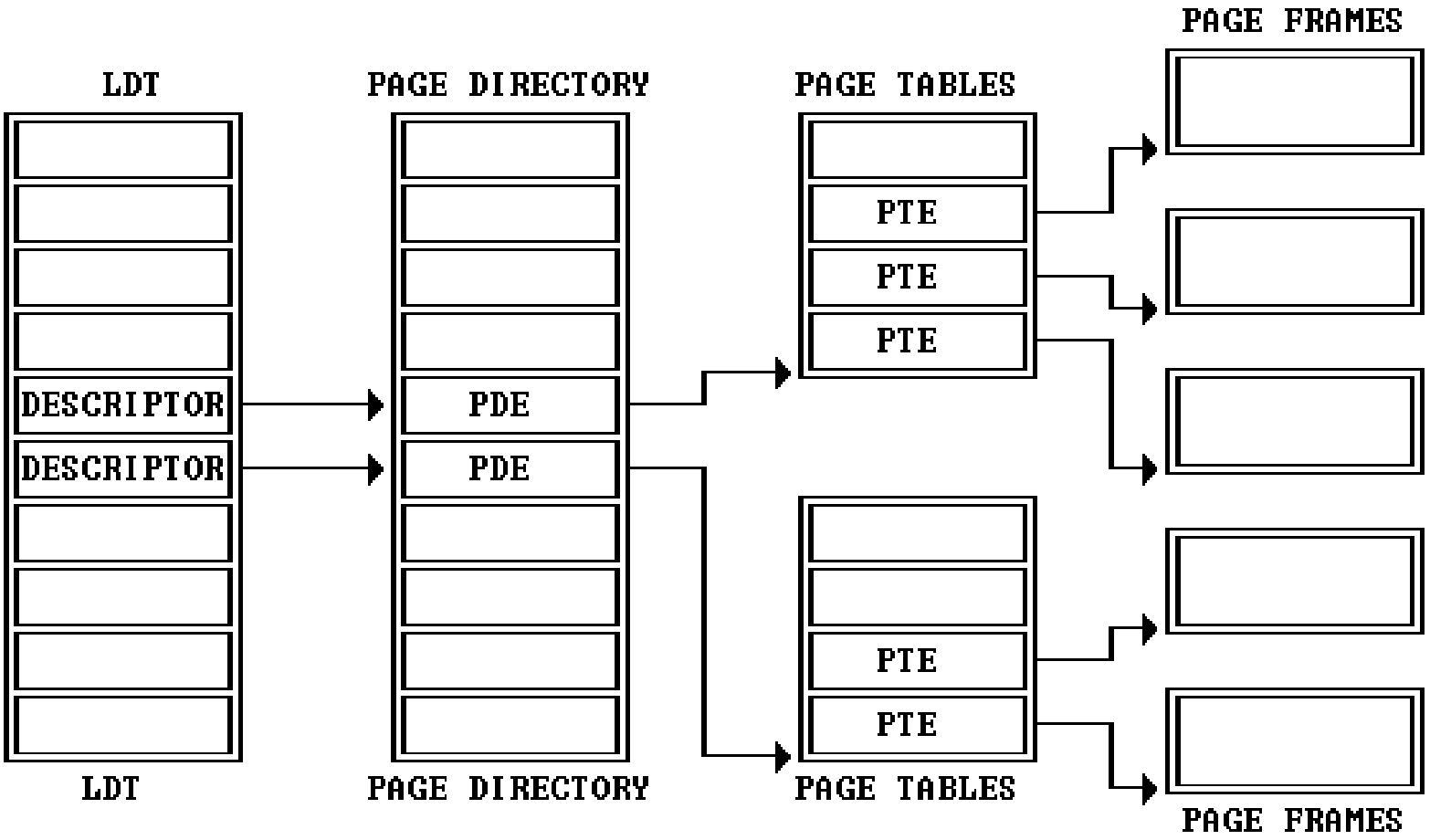
2.3 Combining Segmentation and Paging

Figure 5-12. 80306 Addressing Machanism



2.3 Combining Segmentation and Paging

Figure 5-13. Descriptor per Page Table



3. The Initialization

After RESET is asserted, some registers are set to known states.

This known states are sufficient to allow software to begin execution.

Software can build data structures in memory, such as GDT and IDT tables.

ix86 has several processing modes. It begins in 8086 mode (arggh!), called real-address mode.

If YOU want to use a 32 bits processor, YOU need to set up some structures and special bits.

3.1 Processor State After Reset

A self test may be requested at power-up by asserting the AHOLD input during the falling edge of the RESET signal.

It´s responsibility of the hardware designer to provide the request for self test.

The EAX register is clear if the ix86 processor passed the tested (if self test is not requested, its content is undefined).

The DX register holds a component identifier and revision number (DH →x86, DL →revision)

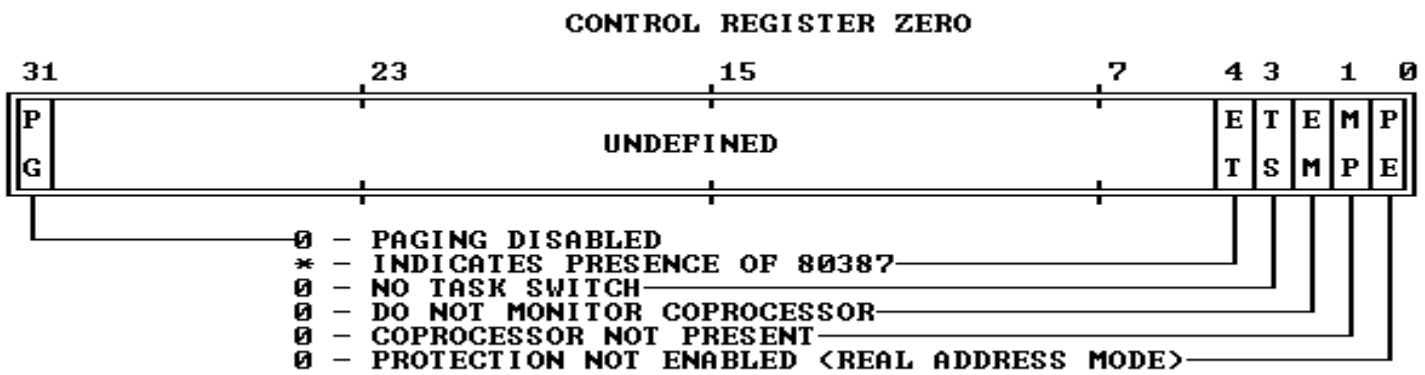
3.2 Processor State After Reset

Registers EBX, ECX, ESI, EDI, EBP, ESP, GDTR, LDTR, TR are undefined.

Invisible parts of CS and DS are initialized to base 0 and limit of top memory minus 64KB.

CRO register has the following values:

Figure 10-2. Initial Contents of CR0



3.3 Initialization in Real-Address Mode

Software sets up data structures needed for the processor to perform basic system functions, such as handling interrupts.

If processor is going to operate in protected mode, software sets ups data structures used by ix86 processors, then switches modes.

In real-mode, no descriptor tables are used, but the IDT needs to loaded with pointers to exception and interrupt handlers before interrupts can be enabled.

3.3 Initialization in Real-Address Mode

The NMI (Non Maskable Interrupts) are always enabled. Hardware must provide a mechanism to prevent an NMI interrupt to being generated while software is being initialized

Execution begins with the instruction addressed by initial contents of CS and IP registers: 0x0FFF FFF0

In real-mode, only “near” jumps are possible. After a “far” jump is executed, addresses issued for the code segment are clear in the 12 MSB.

3.3 Initialization in Real-Address Mode

To finally switch to protected mode, at least one descriptor table, the GDT, and two descriptors (code and data) must be created.

Base address and limit for GDT must be loaded into the GDTR register using LGDT instruction.

A IDT and a gate for the NMI interrupt handler need to created.

Base address and limit for IDT must be loaded into the IDTR register using LIDT instruction.

3.3 Initialization in Real-Address Mode

Protected mode is entered by setting the PE bit in the CR0 register using LMSW or MOV CR0 instructions.

A JMP instruction immediately after LMSW instruction change the flow of execution (emptying processor pre-fetched instructions)

Software need to reload all segment registers.

Execution begins with a CPL of 0 (the most privileged level).

3.4 Initialization in Protected Mode

The data structures needed are determined by the memory management features are being used (segmentation, paging).

Segmentation is always enabled, but it it can be used from:

- A single and uniform address space (flat model) with one segment (almost just like disabling segmentation)
- A hight structured model with several independent, protected across spaces for each task (multi-segmented model)

3.4 Initialization in Protected Mode

A flat model only requires a GDT with one code and one data segment descriptor.

A multi-segmented model require additional segments for the OS, as well segments and LDTs for each task.

Until 8192 segments per table are possible, but only 6 of them are available simultaneously (there are six segment registers)

3.4 Initialization in Protected Mode

Paging is controlled by a mode bit, the PG bit in the CR0 register.

Before setting PG bit (enabling paging), the following conditions must be true:

- Software has created at least two page tables: the page directory and at least one second-level page table.
- The PDBR register (CR3) is loaded with base address of the page directory
- The processor is in protected mode.

3.4 Initialization in Protected Mode

If multi task is to be used, is necessary to initialize the TR register.

A TSS and a TSS descriptor for the initialization software must be created.

TSS descriptors must be marked as busy when they are created (TSS descriptors reside in the GDT).

The LTR instruction is used to load a selector for the TSS descriptor of the initialization software into the TR register.

Code for Initialization Software

```
; CONSTANTS
;=============================================================================
; FHYSICAL MEMORY MAP
; 0x0000 0000 -+-----------------------+ BOOT_IDT
;              | IDT (4 K)             |
; 0x0000 1000 -+-----------------------+ BOOT_GDT
;              | GDT (4 K)             |
; 0x0000 2000 -+-----------------------+
;              :                       :
;              | BOOT STACK (23 K)     |
; 0x0000 7c00 -+-----------------------+ BOOTSTRAP_STACK, BOOTSTRAP_CODE
;              | BOOT CODE (512 b)     |
; 0x0000 7e00 -+-----------------------+
;              | RESERVED (512 b)      |
; 0x0000 8000 -+-----------------------+ DISK_IMAGE
;              | DISK IMAGE (608 K)    |
;              :                       :
;              |                       |
; 0x000a 0000 -+-----------------------+
;              | UNUSED (384K)         |
;              :                       :
;              |                       |
; 0x000f f000 -+-----------------------+
```

Code for Initialization Software

```
BOOT_IDT        = 0x0000
BOOT_GDT        = 0x1000
BOOTSTRAP_STACK = 0x7c00 ; Descendent
BOOTSTRAP_CODE  = 0x7c00 ; The bootstrap code (512 bytes)
DISK_IMAGE      = 0x8000 ; = BOOT_IMAGE_PHY_ADDR from memory_map

; The size of a disk sector in bytes
DISK_SECT_SIZE =  512

; The size of ELF object's header in bytes
; ELF_HDR_SIZE =  116  ;(not stripped)
ELF_HDR_SIZE =    84
```

Code for Initialization Software

```
; DISK IMAGE LAYOUT
; -+-----------------------+ DISK_IMAGE_SYS_INFO
;  | SYS_INFO (512 bytes)  |
; -+-----------------------+ DISK_IMAGE_SETUP
;  | SETUP                 |
;  :                       :
; -+-----------------------+
;  | SYSTEM                |
;  :                       :
; -+-----------------------+
;  | INIT                  |
;  :                       :
; -+-----------------------+
;  | LOADER/APP1           |
;  :                       :
; -+-----------------------+
;  | APP1                  |
;  :                       :
; -+-----------------------+
;  :                       :
; -+-----------------------+
;  | APPn                  |
;  :                       :
; -+-----------------------+
```

Code for Initialization Software

```
; BOOT IMAGE LAYOUT
; System Information
DISK_IMAGE_SYS_INFO =    DISK_IMAGE

; System Information
DISK_IMAGE_SETUP =       (DISK_IMAGE + DISK_SECT_SIZE)

; SETUP entry point
SETUP_ENTRY =            (DISK_IMAGE_SETUP + ELF_HDR_SIZE)

;=============================================================================
; DEFINITIONS                                                               =
;=============================================================================
; 8086 segment addresses
IMAGE_SEG       = DISK_IMAGE >> 4
INFO_SEG        = DISK_IMAGE_SYS_INFO >> 4
```

Code for Initialization Software

```
;=============================================================================
; MAIN                                                                       =
;=============================================================================
.text
entry main
main:
     cli                 ; disable interrupts
     xor  ax,ax          ; data segment base = 0x00000
     mov  ds,ax
     mov  es,ax
     mov  ss,ax
     mov  sp,#BOOTSTRAP_STACK ; set stack pointer

; Load the boot image from the disk into "DISK_IMAGE"
     mov     si,#msg_loading
     call    print_msg
     push    es
     mov     ax,#IMAGE_SEG
     mov     es,ax       ; don't try to load es directly;
     mov     bx,#0       ; set es:bx to DISK_IMAGE
     mov     ax,[n_sec_image]
     mov     cx,#0x0002  ; starts at track #0, sector #2,
     mov     dx,#0x0000  ;  side #0, first drive
     call    load_image
     pop     es
     mov     si,#msg_done
     call    print_msg
```

Code for Initialization Software

```
; Stop the drive motor
     call stop_drive

; Get extended memory size (in K)
;    xor  dx,dx
;    mov  ah,#0x88
;    int  0x15      ; what if memory size > 64 Mb?
;    push ds
;    push #INFO_SEG
;    pop  ds
;    mov  [0],ax
;    mov  [2],dx
;    pop  ds

; Say hello;
     mov  si,#msg_hello
     call print_msg

; Enable A20 bus line
     call enable_a20

; Zero IDT and GDT
     cld
     xor  ax,ax
     mov  cx,#0x1000     ; IDT + GDT = 8K (4K WORDS)
     mov  di,#BOOT_IDT   ; initial address (relative to ES)
     rep                 ; zero IDT and GDT with AX
     stosw
```

Code for Initialization Software

```
; Set GDT
     mov  si,#GDT_CODE    ; Set GDT[1]=GDT_CODE and
     mov  di,#BOOT_GDT    ; GDT[2]=GDT_DATA
     add  di,#8           ; offset GDT[1] = 8
     mov  cx,#8           ; sizeof GDT[1] + GDT[2] = 8 WORDS
     rep             ; move WORDS
     movsw

; Set GDTR
     lgdt GDTR

; Enable Protected Mode
     mov  eax,cr0
     or   al,#0x01  ; set PE flag and MP flag
     mov  cr0,eax

; Adjust selectors
     mov  bx,#2 * 8 ; adjust data selectors to use
     mov  ds,bx     ; GDT[2] (DATA) with RPL = 00
     mov  es,bx
     mov  fs,bx
     mov  gs,bx
     mov  ss,bx
```

Code for Initialization Software

```
; As Linux as86 can't generate 32 bit instructions, we have to code it by hand.
; The instruction below is a inter segment jump to GDT[GDT_CODE]:SETUP.
; Jump into "SETUP" (actually ix86 Protected Mode starts here)
;    jmp  0x0008:#SETUP_ENTRY
     .byte     0x66
     .byte     0xEA
     .long     SETUP_ENTRY
     .word     0x0008

;=============================================================================
; PRINT_MSG                                                                  =
;                                                                            =
; Desc: Print a \0 terminated string on the screen using the BIOS      =
;       Message must end with 00h ;                                          =
;                                                                            =
; Parm: si  -> pointer to the string                                         =
;=============================================================================
print_msg:
     pushf
     push ax
     push bx
     push bp
     cld
```

Code for Initialization Software

```
print_char:
     lodsb
     cmp  al,#0
     jz   end_print
     mov  ah,#0x0E
     mov  bx,#0x0007
     int  0x10
     jmp  print_char
end_print:
     pop  bp
     pop  bx
     pop  ax
     popf
     ret

;=========================================================================
; SAY_Z                                                                  =
; Desc: Print 'z' on the screen.                                         =
;=========================================================================
say_z:
     pushf
     push si
     mov  si,#msg_z
     call print_msg
     pop  si
     popf
     ret
```

Code for Initialization Software

```
;=========================================================================
; STOP_DRIVE                                                             =
;                                                                        =
; Desc: Stops the drive motor.                                           =
;=========================================================================
stop_drive:
                pushf
                push     ax
                push     dx
                mov dx,#0x03F2
                xor al,al
                out dx,al
                pop dx
                pop ax
                popf
                ret
```

Code for Initialization Software

```
;=========================================================================
; LOAD_ONE_SECTOR                                                        =
;                                                                        =
; Desc: Load a single sector from disk using the BIOS.                   =
;                                                                        =
; Parm: es:bx  -> buffer                                                 =
;       cx     -> track (ch) and sector number (cl)                      =
;       dx     -> side (dh) and drive number (dl)                        =
;=========================================================================
load_sector:
     pushf
     push ax

     mov     ax,#0x0201      ; function #2, load 1 sector
     int     0x13
     cli                     ; int 0x13 sets IF
     jc      ls_disk_error    ; if CY=1, error on reading
     pop  ax
     popf
     ret
ls_disk_error:
     mov     si,#msg_disk_error
     call    print_msg      ; print error msg if disk is bad
     call    stop_drive
ls_disk_halt:
     jmp     ls_disk_halt   ; halt
```

Code for Initialization Software

```
;=========================================================================
; LOAD_IMAGE                                                            =
;                                                                       =
; Desc: Load the the image from disk into a buffer.                     =
;                                                                       =
; Parm: es:bx  -> buffer                                                =
;       ax     -> number of sectors to load                             =
;       cx     -> initial track (ch) and sector number (cl)             =
;       dx     -> inital side (dh) and drive number (dl)                =
;=========================================================================
load_image:
     pushf
     push ax
     push bx
     push cx
     push dx
     push es
li_check:
     cmp     ax,#0
     jz      li_done
     call    load_sector
     push ax
     mov  ax,es
     add  ax,#0x20  ; get next buffer position
     mov  es,ax
     pop  ax
     dec  ax
```

Code for Initialization Software

```
     inc     cl              ; get next sector
     cmp     cl,#19      ; was this last sector?
     jnz     li_next
     mov     cl, #1
     inc     dh              ; get next side
     cmp     dh, #2          ; read both?
     jnz     li_next
     mov     dh, #0
     inc     ch              ; get next track
     call    say_z
li_next:
     jmp     li_check

li_done:
     pop  es
     pop  dx
     pop  cx
     pop  bx
     pop  ax
     popf
     ret
```

Code for Initialization Software

```
;=========================================================================
; ENABLE_A20                                                            =
;                                                                       =
; Desc: Enables the 20th address bus line by writing some bytes to the  =
;       keyboard port.                                                  =
;=========================================================================
enable_a20:
     pushf
     push ax

     call keyb_flush     ; empty keyb controller
     mov  al,#0xd1  ; keyb->cmd = write
     out  #0x64,al
     call keyb_flush
     mov  al,#0xdf  ; keyb->data = 0xdf
     outb #0x60,al
     call keyb_flush

     pop  ax
     popf
     ret
```

Code for Initialization Software

```
;=========================================================================
; FLUSH_KEYB                                                          =
;                                                                       =
; Desc: Flushes the keyboard controler                                =
;=========================================================================
keyb_flush:
     pushf
     push ax

kf_stat:  call    kf_delay
     in       al,#0x64         ; get keyb status
     test     al,#1            ; output buffer full?
     jz       kf_emptyt
     call     kf_delay
     in       al,#0x60         ; get data
     jmp      kf_stat
kf_emptyt:       test    al,#2           ; input buffer full?
     jnz      kf_stat

     pop  ax
     popf
     ret

kf_delay: ret
```

Code for Initialization Software

```
;*****************************************************************************
; DATA SEGMENT                                                               =
;*****************************************************************************
GDTR:
     .word     0x0FFF          ; GDT limit - 4K
     .long     0x00001000      ; GDT base address - also 4K

GDT_CODE:
     .word 0xFFFF            ; limit 15:00
     .word 0x0000            ; base 15:00
     .byte 0x00              ; base 23:16
     .byte 0x9A              ; 10011001 flags (p/dpl/s/code/c/w/a)
     .byte 0xCF              ; 11001111 (g/d/0/avl) & limit 19:16
     .byte 0x00              ; base 31:24

GDT_DATA:
     .word 0xFFFF            ; limit 15:00
     .word 0x0000            ; base 15:00
     .byte 0x00              ; base 23:16
     .byte 0x92              ; 10010011 flags (p/dpl/s/data/e/w/a)
     .byte 0xCF              ; 11001111 (g/d/0/avl) & limit 19:16
     .byte 0x00              ; base 31:24
```

Code for Initialization Software

```
msg_disk_error:
     .ascii "Disk error: system halted;"
     .word 0x0D0A
     .byte 0x00
msg_hello:
     .ascii  "This is EPOS;"
     .word 0x0D0A
     .byte 0x00
msg_z:
     .ascii "."
     .byte 0x00
msg_loading:
     .ascii "Loading EPOS "
     .byte 0x00
msg_done:
     .ascii " done;"
     .word 0x0D0A
     .byte 0x00
; The following is to enable "sys" to define the actual size of the image.
; The default value of 360 will be only used if you don't use sys.
     .align 508
n_sec_image:
     .word 360

; Tag the boot sector as "bootable"
     .word 0xAA55
```