Table of Contents

12. Boot Loader

1. Synopsis

A boot loader is a small program that loads the operating system into the computer's memory when the system is booted and also starts the operating system. It is more fundamental than an operating system in the sense that it is responsible for transferring control to the operating system.

The Aim of this project is to create a boot loader capable of doing the following:

- Load a binary image into memory from a FAT file system.
- Transfer Control to it
- Enable the 21st Addressing Line (A21)
- Reprogram the Programmable Interrupt Controller
- Setup the Descriptor Tables
- Switch to protected mode

2. Design

The bootstrap loader is a small program built into ROM that is run automatically after the POST (Power On Self Test). The bootstrap loader loads the first sector from the secondary storage into memory and transfers control to it.

Ideally, the boot loader (different from the bootstrap loader) resides on this first sector. That would severely restrict the functionality of the boot loader because on the limitation on size (1 sector). Hence to remove the size restrictions the design discussed in the following page is used.

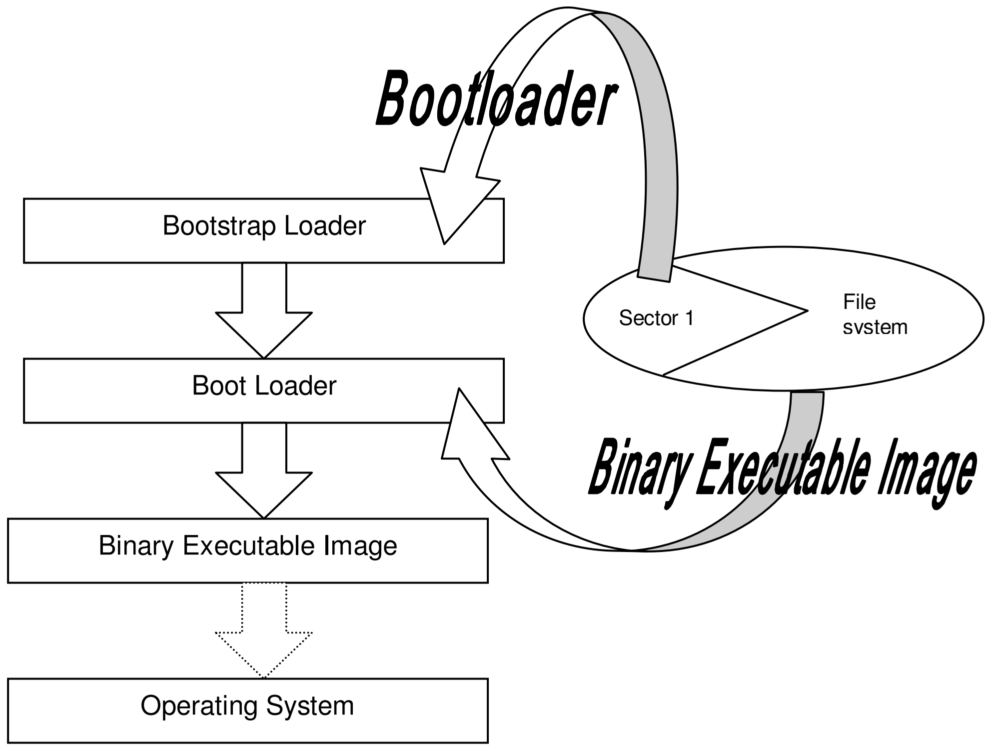


Fig1 – Design of the boot loader

The bootstrap loader loads the boot loader from the first sector of secondary storage and then transfers control to it. The boot loader then reads a predetermined binary executable image from the root directory of the file system on the root directory and then transfers control to it. The ability to read the binary executable image from the file system lifts the restrictions on the size of the binary executable image.

The binary executable image performs the remaining operations and the transfers control to the operating system, whose image can be preloaded at a predetermined address.

3. Loading a file from the file system

The boot loader attempts to load the binary executable file from the secondary storage. Here the root directory of the floppy disk contains the binary executable image. The FAT12 file system is an efficient and widely used file system for floppy disks and generally for storage media with small capacity. It reads sectors from the floppy disk using the services under BIOS Interrupt 13h. It's a convention to attempt reading at least thrice, since the floppy motor takes some time to start. The FAT12 file system is described briefly below.

4. The FAT file system

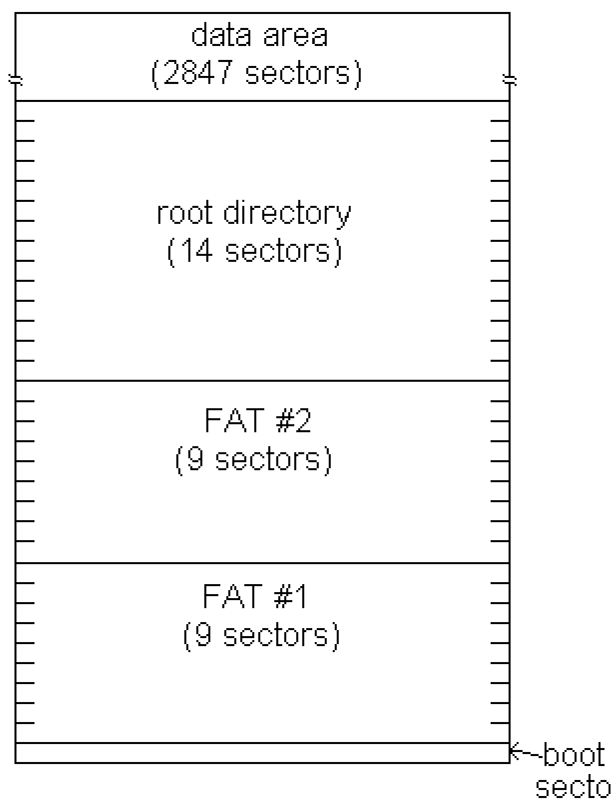
FAT stands for *File Allocation Table*, the main feature of this file system. The FAT file system is associated with DOS and some versions of Windows. *VFAT* (virtual FAT) is FAT with *long filenames.* Newly purchased floppy disks advertised as 'formatted' usually contain a FAT12 file system.

4.1 Layout of a FAT volume

A FAT Volume contains:

- Boot sector (more than one for FAT32)
- One or more copies of the FAT (almost always 2 copies)
- Root directory (not present for FAT32)

1.44 meg FAT12 floppy



- The data area, where files and subdirectories are stored. For FAT32, the root directory is also stored here.

4.2 FAT Boot sector and BPB

The FAT boot sector contains:

- Code to load and run the DOS kernel
- The (poorly named) *BIOS Parameter Block* (BPB), with disk geometry and file system info.
- Magic values: 55h at offset 510, 0AAh at offset 511.

There are additional fields, which are required by FAT32, but are optional for FAT12 and FAT16.

The BIOS Parameter Block

```
typedef unsigned char  uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long  uint32_t;
struct fat_bootsector{
     uint8_t  jump[3];
     uint8_t  oem_id[8];
     uint16_t bytes_per_sector;
     uint8_t  sectors_per_cluster;
     uint16_t num_boot_sectors;
     uint16_t num_root_dir_ents;
     uint16_t total_sectors;
     uint8_t  media_ID_byte;
     uint16_t sectors_per_fat;
     uint16_t sectors_per_track;
     uint16_t heads;
     uint32_t hidden_sectors;
     uint32_t total_sectors_large;
     uint8_t  boot_code[474];
     uint8_t  magic[2];
```

The Directory Entry

```
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
struct fat_dirent
{
    uint8_t  name[8];
    uint8_t  ext[3];
    uint8_t  attrib;
    uint8_t  reserved;
    uint8_t  ctime_ms;
    uint16_t ctime;
    uint16_t cdate;
    uint16_t adate;
    uint16_t st_clust_msw;
    uint16_t mtime;
    uint16_t mdate;
    uint16_t st_clust;
    uint32_t file_size;
```

Directory Entry Attribute

```
struct attrib
{
     int read_only : 1;      /* b0 */
     int hidden : 1;
     int system : 1;
     int volume_label : 1;
     int directory : 1;
     int archive : 1;
     int reserved : 2;       /* b6, b7 */
};/* 1 byte total */
```

DOS File Time format

```
struct dos_time
{/* ordered from lsb to msb */
    unsigned two_secs : 5;  /* 2sec increments */
    unsigned minutes : 6;   /* minutes */
    unsigned hours : 5;     /* hours(0-23) */
}; /* 2 bytes total */
```

DOS File Date format

```
{/* ordered from lsb to msb */
    unsigned date : 5;      /* date (1-31) */
    unsigned month : 4;     /*(1-12) */
    unsigned year : 7;      /*year - 1980 */
};/* 2 bytes total */
```

4.3 The File Allocation Table

Entries in the FAT can be 12 bits wide (FAT12), 16 bits wide (FAT16), or 32 bits wide (FAT32). FAT entries do not necessarily refer to disk sectors, but to *clusters*, which are groups of contiguous sectors. The number of sectors per cluster is always a power of 2. Only the number of clusters in the volume determines the FAT format used:

- FAT12: 1...4084 (0FF4h) clusters
- FAT16: 4085...65524 (0FFF4h) clusters
- FAT32: 65525... clusters

Used FAT entries form singly linked lists, indicating which clusters are used by each file or subdirectory. Some FAT entry values are special.

Meaning of FAT entry value	FAT12	FAT16	FAT32
Free cluster	0	0	0
Used cluster; pointer to next	2-0FF5h	2-0FFF5h	2-0FFFFFF5h **(28-bit)**
Reserved	0FF6h	0FFF6h	0FFFFFF6h **(28-bit)**
Bad cluster	0FF7h	0FFF7h	0FFFFFF7h **(28-bit)**
Reserved	0FF8h-0FFEh	0FFF8h-0FFFEh	0FFFFFF8h-0FFFFFFEh**(28-bit)**
Used cluster; last in chain	0FFFh	0FFFFh	0FFFFFFFh **(28-bit)**

5. The twenty first addressing line

The 8088 in the original PC had only 20 address lines, sufficient for 1 MB. The maximum address FFFF:FFFF addresses 0x10ffef, and this would silently wrap to 0x0ffef. When the 80286 (with 24 address lines) was introduced, it had a real mode that was intended to be 100% compatible with the 8088. However, it failed to do this address truncation (a bug), and people found that there existed programs that actually depended on this truncation. Trying to achieve perfect compatibility, IBM invented a switch to enable/disable the 0x100000 address bit. Since the 8042 keyboard controller happened to have a spare pin, that was used to control the AND gate that disables this address bit. The signal is called A20, and if it is zero, bit 20 of all addresses is cleared.

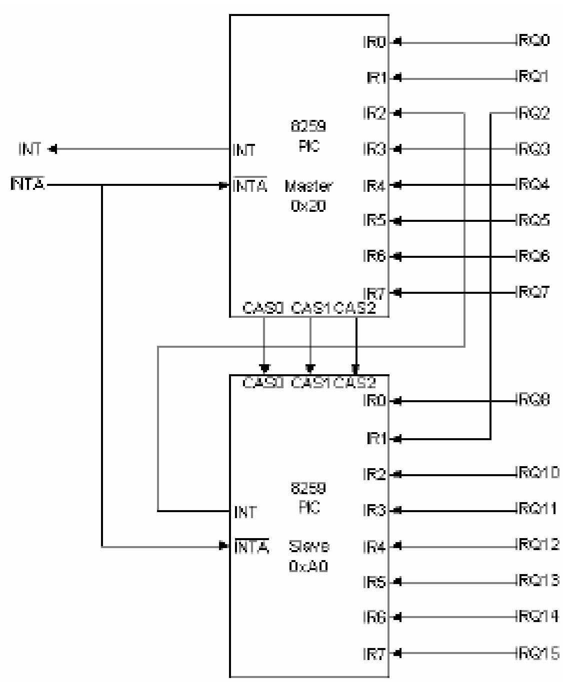
By default the A20 address line is disabled at boot time, so the software has to find out how to enable it, and that may be nontrivial since the details depend on the chipset used. Protected mode needs that the 21st addressing line be enabled. Most common chipsets use the keyboard controller. Others use i/o ports to control the A21 line. Some chipsets have emulators, so that both these methods would work. If one finds it difficult, maybe impossible, to write a routine that will enable A20 on all PCs, one might ask the BIOS to do so. Many recent BIOS versions implement INT15 AX=240h functions.

Bios Interrupt 15h

```
INT 15 AX=2400 disable A20
INT 15 AX=2401 enable A20
INT 15 AX=2402 query status A20
INT 15 AX=2403 query A20 support (kbd
               or port 92)
Return:
  If successful: CF clear, AH = 00h
  On error: CF set, AH = status
Status:
01h    keyboard controller is in
       securemode
86h   function not supported
For AX=2402 the status (0: disabled, 1:
enabled) is returned in AL.
For AX=2403 the status (bit 0: kbd, bit
1: port 92) is returned in BX.
```

6. Programming the Programmable Interrupt Controller (PIC)

Intel reserves the first 32 interrupts for its own use as exception handlers. However since the BIOS sets up the CPU in real mode, it uses vectors 8-15 for hardware interrupt service routines. So we have to reprogram the interrupt controller(s) to use vectors above 32(20h). In real



mode, The master PIC uses vectors 8h-Fh and the slave PIC uses vectors 70h-77h we program them to use vectors from 20h-27h and 28h-2Eh respectively.

7. Switching to Protected Mode

Setting the PE bit of the MSW in CR0 causes the 80386 to begin executing in protected mode. The current privilege level (CPL) starts at zero. The segment registers continue to point to the same linear addresses as in real address mode (in real address mode, linear addresses are the same physical addresses).

Immediately after setting the PE flag, the initialization code must flush the processor's instruction prefetch queue by executing a JMP instruction. The 80386 fetches and decodes instructions and addresses before they are used; however, after a change into protected mode, the prefetched instruction information (which pertains to real-address mode) is no longer valid. A JMP forces the processor to discard the invalid information.

Most of the initialization needed for protected mode can be done either before or after switching to protected mode. If done in protected mode, however, the initialization procedures must not use protected-mode features that are not yet initialized.

Interrupt Descriptor Table:

The IDTR may be loaded in either real-address or protected mode. However, the format of the interrupt table for protected mode is different than that for real-address mode. It is not possible to change to protected mode and change interrupt table formats at the same time; therefore, it is inevitable that, if IDTR selects an interrupt table, it will have the wrong format at some time. An interrupt or exception that occurs at this time will have unpredictable results. To avoid this unpredictability, interrupts should remain disabled until interrupt handlers are in place and a valid IDT has been created in protected mode.

Stack:

The SS register may be loaded in either real-address mode or protected mode. If loaded in real-address mode, SS continues to point to the same linear base-address after the switch to protected mode.

Global Descriptor Table:

Before any segment register is changed in protected mode, the GDT register must point to a valid GDT. Initialization of the GDT and GDTR may be done in real-address mode. The GDT (as well as LDTs) should reside in RAM, because the processor modifies the accessed bit of descriptors.

Page Tables:

Page tables and the PDBR in CR3 can be initialized in either real-address mode or in protected mode; however, the paging enabled (PG) bit of CR0 cannot be set until the processor is in protected mode. PG may be set simultaneously with PE, or later. When PG is set, the PDBR in CR3 should already be initialized with a physical address that points to a valid page directory. The initialization procedure should adopt one of the following strategies to ensure consistent addressing before and after paging is enabled:

- The page that is currently being executed should map to the same
- physical addresses both before and after PG is set.
- A JMP instruction should immediately follow the setting of PG.

First Task:

The initialization procedure can run awhile in protected mode without initializing the task register; however, before the first task switch, the following conditions must prevail:

- There must be a valid task state segment (TSS) for the new task. The stack pointers in the TSS for privilege levels numerically less than or equal to the initial CPL must point to valid stack segments.

- The task register must point to an area in which to save the current task state. After the first task switch, the information dumped in this area is not needed, and the area can be used for other purposes.

8. Software Used

Bochs

Bochs is a highly portable open source IA-32 (x86) PC emulator written in C++, that runs on most popular platforms. It includes emulation of the Intel x86 CPU, common I/O devices, and a custom BIOS. Currently, Bochs can be compiled to emulate a 386, 486, Pentium, Pentium Pro or AMD64 CPU, including optional MMX, SSE, SSE2 and 3DNow instructions. Bochs is capable of running most Operating Systems inside the emulation including Linux, Windows® 95, DOS, and Windows® NT 4. Bochs was written by Kevin Lawton and is currently maintained by this project.

Bochs can be compiled and used in a variety of modes, some which are still in development. The 'typical' use of bochs is to provide complete x86 PC emulation, including the x86 processor, hardware devices, and memory. This allows one to run OS's and software within the emulator on your workstation, much like you have a machine inside of a machine. For instance, if one has a Unix/X11 workstation, but wanted to run Win'95 applications. Bochs will allow you to run Win 95 and associated software on your Unix/X11 workstation, displaying a window on your workstation, simulating a monitor on a PC.

Bochs, though slow, is an important tool for OS development as it allows one to debug the software, examine physical memory and other things that can be accomplished only with a hardware debugger.

Assemblers

NASM

NASM is an 80x86 assembler designed for portability and modularity. It supports a range of object file formats including Linux a.out and ELF, COFF, Microsoft 16-bit OBJ and Win32. It will also output plain binary files. Its syntax is designed to be simple and easy to understand, similar to Intel's but less complex. It supports Pentium, P6, MMX, 3DNow! and SSE opcodes, and has macro capability. It includes a disassembler as well. The main advantage of NASM is its ability to generate different output formats.

MASM 6.0

It is one of the most widely used assemblers that supports a wide range of macros and helps the assembly language programmer structure his code. It has a rich syntax and directly supports DOS and Windows executable formats.

PARTCOPY 2.0

This program copies contiguous ranges of raw data between files and/or disks; useful for reading and writing boot sectors.

Disk Explorer

Useful for manipulating disk images, which are for use with Bochs.

9. References

1. The Intel 80386 Programmers Reference Manual (available on the Internet by searching for INTEL386.ZIP).
2. The Intel Microprocessor – Architecture, Programming and Interfacing by Barry B. Brey
3. Resources at http://www.osdever.net/
4. Resources at http://www.osdev.org/
5. The Operating System Resource Center - http://www.nondot.org/sabre/os/articles/ProtectedMode/
6. Write your own OS - http://my.execpc.com/CE/AC/geezer/osd/
7. The Bochs official site - http://bochs.sourceforge.net/
8. Ralf Brown's Interrupt list - http://www.ctyme.com/rbrown.htm
9. Resources from Norton's guide - http://www.clipx.net/norton.php
10. www.pcguide.com

10. Source Code

Loader2.s

```
; This code is derivedfrom the code in Loader1.s, The code for loading
; a binary file into memory has been moved to a separate procedure and
; the main proc calls the procedure. So you can always call the function
; again to load another binary file at yet another address.

     [BITS 16]
     jmp     START

     OEM_ID                db "QUASI-OS"
     BytesPerSector        dw 0x0200
     SectorsPerCluster     db 0x01
     ReservedSectors       dw 0x0001
     TotalFATs             db 0x02
     MaxRootEntries        dw 0x00E0
     TotalSectorsSmall     dw 0x0B40
     MediaDescriptor       db 0xF0
     SectorsPerFAT         dw 0x0009
     SectorsPerTrack       dw 0x0012
     NumHeads              dw 0x0002
     HiddenSectors         dd 0x00000000
     TotalSectorsLarge     dd 0x00000000
     DriveNumber           db 0x00
     Flags                 db 0x00
     Signature             db 0x29
     VolumeID              dd 0xFFFFFFFF
     VolumeLabel           db "QUASI  BOOT"
     SystemID              db "FAT12   "

     START:
     ; code located at 0000:7C00, adjust segment registers
          cli
          mov     ax, 0x07C0
          mov     ds, ax
          mov     es, ax
          mov     fs, ax
          mov     gs, ax
     ; create stack
          mov     ax, 0x0000
          mov     ss, ax
          mov     sp, 0xFFFF
          sti
     ; post message
          mov     si, msgLoading
          call    DisplayMessage

     ; Load Binary Image
          push    WORD 0x1018                ; arg2 - Load address
          push    WORD ImageName             ; arg1 - ImageName offset
          call    LoadBinaryImage
```

```
     add     sp, 2                       ; 0x01018 is still on the stack
     push    WORD 0x0000
     retf

;*************************************************************************
; PROCEDURE LoadBinaryImage
; Offset of ImageName at ss:sp and load segment at ss:sp+2
;*************************************************************************
LoadBinaryImage:
     mov     bp, sp
     add     bp, 2                       ; We dont want the return
                                         ; address
LOAD_ROOT:
; compute size of root directory and store in 'cx'
     xor     cx, cx
     xor     dx, dx
     mov     ax, 0x0020                        ; 32 byte directory entry
     mul     WORD [MaxRootEntries]             ; total size of directory
     div     WORD [BytesPerSector]             ; sectors used by directory
     xchg    ax, cx
; compute location of root directory and store in 'ax'
     mov     al, BYTE [TotalFATs]              ; number of FATs
     mul     WORD [SectorsPerFAT]              ; sectors used by FATs
     add     ax, WORD [ReservedSectors]        ; adjust for bootsector
     mov     WORD [datasector], ax             ; base of root directory
     add     WORD [datasector], cx
; read root directory into memory (7C00:0200)
     mov     bx, 0x0200                        ; copy root dir above bootcode
     call    ReadSectors
; browse root directory for binary image
     mov     cx, WORD [MaxRootEntries]         ; load loop counter
     mov     di, 0x0200                        ; locate first root entry
.LOOP:
     push    cx
     mov     cx, 0x000B                        ; eleven character name
     mov     si, WORD [bp]                     ; image name to find
     push    di
rep  cmpsb                                     ; test for entry match
     pop     di
     je      LOAD_FAT
     pop     cx
     add     di, 0x0020                        ; queue next directory entry
     loop    .LOOP
     jmp     FAILURE
LOAD_FAT:
; save starting cluster of boot image
     mov     dx, WORD [di + 0x001A]
     mov     WORD [cluster], dx                ; file's first cluster
; compute size of FAT and store in 'cx'
     xor     ax, ax
     mov     al, BYTE [TotalFATs]              ; number of FATs
     mul     WORD [SectorsPerFAT]              ; sectors used by FATs
     mov     cx, ax
; compute location of FAT and store in 'ax'
     mov     ax, WORD [ReservedSectors]        ; adjust for bootsector
; read FAT into memory (7C00:0200)
     mov     bx, 0x0200                        ; copy FAT above bootcode
     call    ReadSectors
; read image file into memory (1018:0000)
     mov     ax, WORD [bp+2]
     mov     es, ax                            ; destination for image
     mov     bx, 0x0000                        ; destination for image
     push    bx
LOAD_IMAGE:
     mov     ax, WORD [cluster]                ; cluster to read
     pop     bx                                ; buffer to read into
     call    ClusterLBA                        ; convert cluster to LBA
     xor     cx, cx
     mov     cl, BYTE [SectorsPerCluster]      ; sectors to read
     call    ReadSectors
     push    bx
; compute next cluster
     mov     ax, WORD [cluster]                ; identify current cluster
     mov     cx, ax                            ; copy current cluster
     mov     dx, ax                            ; copy current cluster
```

```
     shr     dx, 0x0001                         ; divide by two
     add     cx, dx                             ; sum for (3/2)
     mov     bx, 0x0200                         ; location of FAT in memory
     add     bx, cx                             ; index into FAT
     mov     dx, WORD [bx]                      ; read two bytes from FAT
     test    ax, 0x0001
     jnz     .ODD_CLUSTER
.EVEN_CLUSTER:
     and     dx, 0000111111111111b              ; take low twelve bits
    jmp     .DONE
.ODD_CLUSTER:
     shr     dx, 0x0004                         ; take high twelve bits
.DONE:
     mov     WORD [cluster], dx                 ; store new cluster
     cmp     dx, 0x0FF0                         ; test for end of file
     jb      LOAD_IMAGE
DONE:
     sub     bp, 2
     mov     sp, bp
     ret
FAILURE:
     mov     si, msgFailure
     call    DisplayMessage
     mov     ah, 0x00
     int     0x16                               ; await keypress
     int     0x19                               ; warm boot computer

;*************************************************************************
; PROCEDURE DisplayMessage
; display ASCIIZ string at ds:si via BIOS
;*************************************************************************
DisplayMessage:
     lodsb                                      ; load next character
     or      al, al                             ; test for NUL character
     jz      .DONE
     mov     ah, 0x0E                           ; BIOS teletype
     mov     bh, 0x00                           ; display page 0
     mov     bl, 0x07                           ; text attribute
     int     0x10                               ; invoke BIOS
     jmp     DisplayMessage
.DONE:
     ret

;*************************************************************************
; PROCEDURE ReadSectors
; reads 'cx' sectors from disk starting at 'ax' into memory location
; 'es:bx'
;*************************************************************************
ReadSectors:
.MAIN
     mov     di, 0x0005                         ; five retries for error
.SECTORLOOP
     push    ax
     push    bx
     push    cx
     call    LBACHS
     mov     ah, 0x02                           ; BIOS read sector
     mov     al, 0x01                           ; read one sector
     mov     ch, BYTE [absoluteTrack]           ; track
     mov     cl, BYTE [absoluteSector]          ; sector
     mov     dh, BYTE [absoluteHead]            ; head
     mov     dl, BYTE [DriveNumber]             ; drive
     int     0x13                               ; invoke BIOS
     jnc     .SUCCESS                           ; test for read error
     xor     ax, ax                             ; BIOS reset disk
     int     0x13                               ; invoke BIOS
     dec     di                                 ; decrement error counter
     pop     cx
     pop     bx
     pop     ax
     jnz     .SECTORLOOP                        ; attempt to read again
     int     0x18
.SUCCESS
     mov     si, msgProgress
     call    DisplayMessage
```

```
     pop     cx
     pop     bx
     pop     ax
     add     bx, WORD [BytesPerSector]          ; queue next buffer
     inc     ax                                 ; queue next sector
     loop    .MAIN                              ; read next sector
     ret

;*************************************************************************
; PROCEDURE ClusterLBA
; convert FAT cluster into LBA addressing scheme
; LBA = (cluster - 2) * sectors per cluster
;*************************************************************************
ClusterLBA:
     sub     ax, 0x0002                         ; zero base cluster number
     xor     cx, cx
     mov     cl, BYTE [SectorsPerCluster]       ; convert byte to word
     mul     cx
     add     ax, WORD [datasector]              ; base data sector
     ret

;*************************************************************************
; PROCEDURE LBACHS
; convert 'ax' LBA addressing scheme to CHS addressing scheme
; absolute sector = (logical sector / sectors per track) + 1
; absolute head   = (logical sector/sectors per track) MOD number of heads
; absolute track  = logical sector / (sectors per track * number of heads)
;*************************************************************************
LBACHS:
     xor     dx, dx                             ; prepare dx:ax for operation
     div     WORD [SectorsPerTrack]             ; calculate
     inc     dl                                 ; adjust for sector 0
     mov     BYTE [absoluteSector], dl
     xor     dx, dx                             ; prepare dx:ax for operation
     div     WORD [NumHeads]                    ; calculate
     mov     BYTE [absoluteHead], dl
     mov     BYTE [absoluteTrack], al
     ret

absoluteSector    db 0x00
absoluteHead      db 0x00
absoluteTrack     db 0x00

datasector        dw 0x0000
cluster           dw 0x0000

ImageName         db "SYSINIT BIN"

msgLoading        db 0x0D, 0x0A, "Loading", 0x00
msgProgress       db ".", 0x00
msgFailure        db " :(", 0x0D, 0x0A, "ERROR", 0x0D, 0x0A, 0x0D, 0x0A, 0x00

     TIMES 510-($-$$) DB 0

     DW 0xAA55
;*************************************************************************
```

a20bios.inc

```
;*********************************************************************
;        Procedure enableA20BIOS: enables the A20 line through the
;              BIOS instead of the keyboard controller or port 92h
;              CF set on error
;*********************************************************************

enableA20      proc    near

        mov ax, 2401h  ; enable A20 gate service
        int 15h

enableA20      endp
```

a20quick.inc

```
;*******************************************************************
;        Procedure enableA20: enables the A20 line through the
;              port 92h instead of the keyboard controller. CF set
;              on failure.
;*******************************************************************

enableA20      proc    near

       push cx
       mov cx, 6              ; try six times

       tryPort:

              in al, 92h      ; it is suggested that the A20 be enabled
              test al, 2      ; only if it isn't already enabled
              jnz noneed
              orb al, 2       ; set bit2 - the A20 gate
              and al, 0FEh    ; disable the reset bit (it probably is a write
              out 92h, al     ; only bit). just in case ...

       loop tryPort

       stc                    ; we failed
noneed:

       clc                    ; everything fine
       pop cx
       ret

enableA20      endp
```

a20kbd.inc

```
;====================================================================
;                      8042 kbd controller ports
;====================================================================
; PORT  ACTION  PURPOSE
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
; 0x60  READ    Output register for getting data from the keyboard
; 0x60  WRITE   Data register for sending kbd controller commands
; 0x64  READ    Status register that can be read for kbd status
; 0x64  WRITE   Commmand register used to set kbd controller options
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

; visit http://www.clipx.net/ng/hardware/ng15f80.php for more info
; on the keyboard controller

;*******************************************************************
;        Procedure kbdWait: waits for the keyboard controller
;              parameter: ah contains 2^<bit to be tested>
;*******************************************************************

kbdWait proc    near

kbdWtLoop:

       in al, 64h      ; read byte from status port into al
       test al, ah     ; test bit specified by ah ( = 1 or 2)
       jnz kbdWtLoop   ; if the bit is 1, the controller
                       ; is not ready
       ret

kbdWait endp

;*******************************************************************
;        Procedure getKbdOutPort: stores the value of the kbd output
;              port into al and copies it to dl. it then tests the
;              A20 bit and updates flags accordingly. used as helper
;              for proc enableA20. returns with ah = 1
;*******************************************************************
```

```
getKbdOutPort  proc   near

       mov ah, 2      ; test write-to not ready flag
       call kbdWait   ; wait for keybd contoller to accept our cmds

       mov al, 0D0h   ; send 'read output port' command
       out 64h, al    ; to the kbd command port

       mov ah, 1      ; test read-from not ready flag
       call kbdWait   ; wait for keybd controller to give our data

       in al, 60h     ; read the 'output port' data
       mov dl, al     ; copy value into dl
       test al, 2     ; update flags, by testing the A20 gate bit

       ret

getKbdOutPort  endp

;*******************************************************************
;        Procedure enableA20: Enables the A20 line. sets CF on
;                             failure
;*******************************************************************




;==================================================================
;              8042 Status Register (port 64h read)
;==================================================================
;  |7|6|5|4|3|2|1|0|  8042 Status Register
;              | +---- output register (60h) has data for system
;              +----- input register (60h/64h) has data for 8042
;
;  bit0 == 1 -> write-to not ready  (call with ah = 1 for kbdWait)
;  bit1 == 1 -> read-from not ready (call with ah = 2 for kbdWait)
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

enableA20      proc   near

       push cx        ; cx is used as loop counter

       ; disable interrupts while enabling the A20(the doc says so)
       ; and thats already been done at the beginning (in main)


       mov cx, 6      ; try enabling A20 6 times

       try_a20:

              call getKbdOutPort
                             ; returns al=dl=output port value
                             ; and zf!=0 when a20 is enabled

              jnz a20_done   ; if so, just stop trying (quit loop)

              or dl, 2       ; bit2 is A20 gate. enable it

              mov ah, 2      ; test write-to not ready flag
              call kbdWait   ; wait for keybd contoller to accept our cmds

              mov al, 0D1h   ; send 'write output port' command
              out 64h, al    ; to the kbd command port

                             ; ah = 2 already
              call kbdWait   ; wait for keybd contoller to accept our data

              mov al, dl     ; saved value with a20 enabled
              out 60h, al    ; write new 'output port' value to data register
                             ; at 60h
       loop try_a20          ; try again

; if execution reaches here, it means we failed to enable a20. lets try another
```

```
; method. some chips use command DFh to enable the A20.

        push si                 ; tell the user that we
        mov si, offset method2msg
        call disp               ; are trying the second
        pop si                  ; method as the first one failed

        mov cx, 6               ; try enabling A20 6 times

        try_method2:

                                ; ah = 2 already. test write-to not ready flag
                call kbdWait    ; wait for keybd contoller to accept our cmds

                mov al, 0DFh    ; send 'Enable A20' command
                out 64h, al

                call getKbdOutPort
                                ; returns al=dl=output port value
                                ; and zf!=0 when a20 is enabled

                jnz a20_done    ; if so, just stop trying (quit loop)
                mov ah, 2       ; ah should be 2 at beginning of loop

        loop try_method2

; I'm sorry the A20 line failed us and our OS is ... :-(

        stc                     ; carry set means failure

a20_done:                       ; The A20 is enabled :-)

        clc                     ; carry clear means were fine

        pop cx
        ret

enableA20       endp
```

Prog8259.inc

```
;*********************************************************************
;        Procedure iodelay: idles for a small amount of time. used
;              with i/o delay.
;
; NOTE: this proc violates the procedure conventions, because it is
;       intented to be used only with procedure prog8259
;*********************************************************************

iodelay proc    near

        mov cx, 50      ; loop fifty times
iowait: loop iowait     ; doing nothing

        ret

iodelay endp

;*********************************************************************
;        Procedure prog8259: reprograms the interrupt controller
;              to use vectors 20h - 27h and 28h - 2Eh
;*********************************************************************

prog8259        proc    near

        mov al,11h      ; send ICW1 = use IC4. bit4 should be 1 for ICW1
        out 20h, al     ; for master 8259A
        call iodelay
        out 0A0h, al    ; then to slave 8259A
        call iodelay

        mov al,20h      ; ICW2 = starting vector number = 20h
```

```
        out 21h, al     ; send it to master
        call iodelay
        mov al, 28h     ; ICW2 = starting vector number = 28h
        out 0A1h, al    ; send it to slave
        call iodelay

        mov al, 4       ; ICW3, 8259-1 is master and 8259-2
        out 21h, al     ; is connected as slave to pin IRQ2
        call iodelay
        mov al, 2       ; ICW3, 8259-2 is a slave
        out 0A1h, al    ; and is connected to IRQ2
        call iodelay
        mov al, 1       ; ICW4, 8086 mode for both master and slave
        out 21h, al     ; to master
        call iodelay
        out 0A1h, al    ; to slave
        call iodelay

        mov al, 0FFh    ; OCW1 mask off all interrupts for now
        out 21h, al     ; both master
        call iodelay
        out 0A1h, al    ; and slave

        ret

prog8259        endp
```

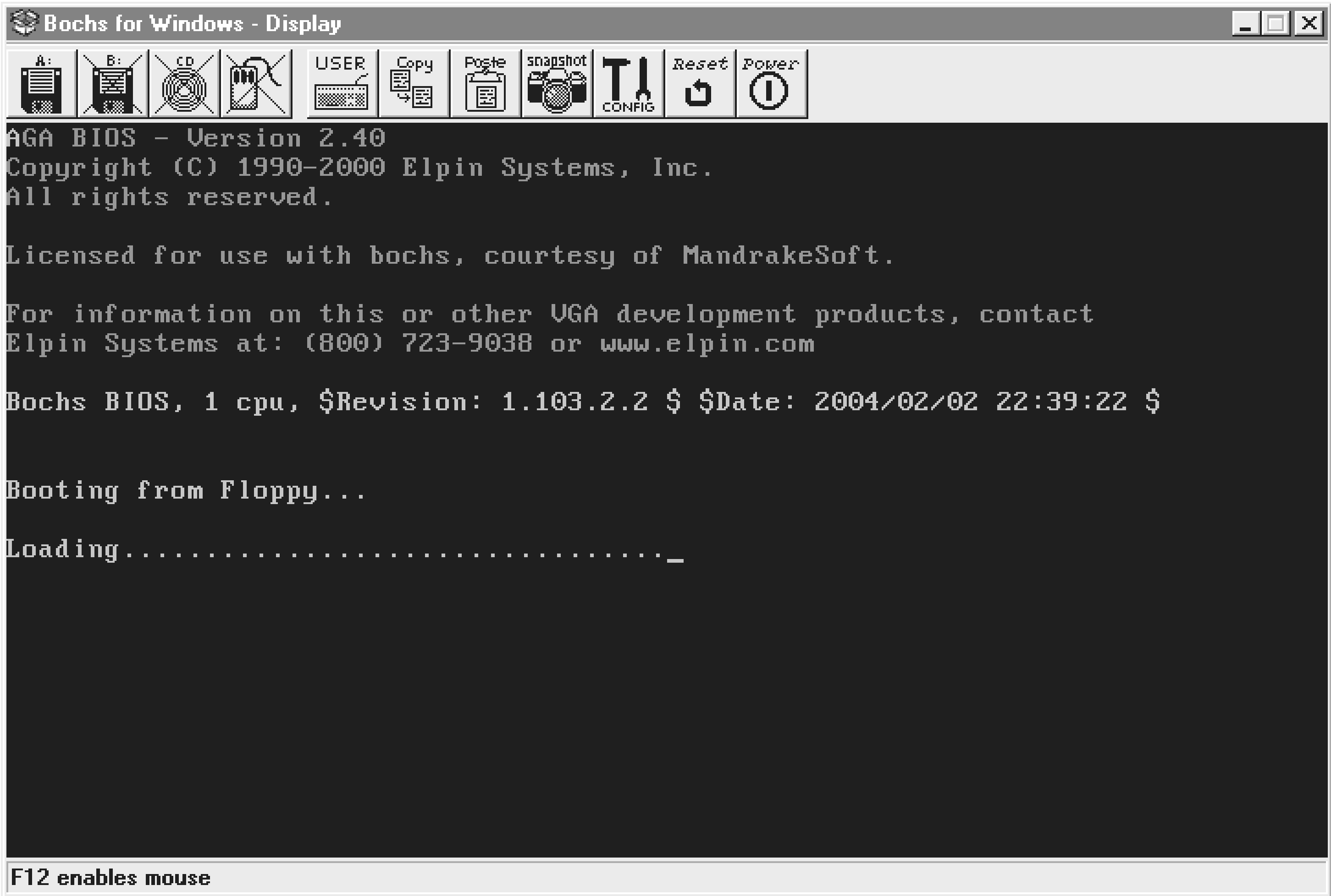
boot.asm

```
.386p

                ;-------------------------------------------------
                ;       Assemble this with MASM 6 or greater
                ;-------------------------------------------------
                ; masm 5.1 gets the lidt, lgdt operand size wrong
                ; in an USE16 segement
                ;-------------------------------------------------

code segment use16

        ASSUME cs:code, ds:code, ss:code

        main    proc    near

        ; adjust segment registers. load them absolutely.
        ; the boot strap loader loads this program at 0000:7C00.
        ; so the code segment starts at physical address 07C00h
        ; or segment address 07C0h. This program has only one segment
        ; hence code = data = stack.

        ; There's one more problem ;| Some systems use cs=0000h some use
        ; cs=07C0h. So its up to us to set it right: lets make it 07C0h
        ; using a far jump in cryptic machine language.

               db 0EAh                 ; machine code for far jump
               dd 07C00005h            ; the future cs:ip value (5 byte instr)
                                       ; points to the cli following this

                cli                      ; some(most) CPU's disable interrupts
                mov ax, 07C0h            ; automatically when loading segment
                mov ds, ax               ; registers. lets do it anyway.
                mov ss, ax
                mov sp, 0FFFFh           ; cs is already setup
                                         ; by the BIOS boot program

        ; NOTE: the interrupt flag remains disabled. Just tell those devices
        ; out there that we'll attend to them after getting into pmode.

        ; NOTE: only the space left over from the code and data parts can be
        ; used for the stack. should be changed for larger programs.

        ; Refer http://www.pcguide.com/ref/ram/logic_XMS.htm for a thorough
        ; explantion of why the A20 line was gated first of all. Also try
        ; http://www.win.tue.nl/~aeb/linux/kbd/A20.html for various methods of
        ; enabling the A20 gate - We'll have to enable it before reaching pmode.

                mov si, offset enableMsg
                call disp

                call enableA20

                jnc announce_a20_OK      ; success, goto next step

                mov si, offset notOKmsg
                call disp                ; Sorry the A20 line was not enabled
                jmp end_of_world         ; and our OS cannot ... :-(

        announce_a20_OK:

                mov si, offset OKmsg
                call disp                ; Tell the user we are fine

        ; The next thing that we do is reprogram the 8259's ( Programmble
        ; interrupt controllers ). Intel reserves the first 32 interrupts
        ; for its own use as exception handlers. However since the BIOS
        ; sets up the CPU in real mode, it uses vectors 8-15 for hardware
        ; interrupt service routines. so we just reprogram the interrupt
        ; controller(s) to use vectors above 32(20h). In real mode,
        ;       PIC1 (master) - uses vectors 8h-Fh
        ;       PIC2 (slave) - uses vectors 70h-77h
        ; we program them to use vectors from 20h-27h and 28h-2Eh resp.
```

```
        mov si, offset prog8259msg
        call disp

        call prog8259           ; program the PIC

        mov si, offset OKmsg
        call disp

; We don't worry about intializing the IDT or clearing unused areas
; of the IDT, GDT. Saves us some space. Just take care not to use
; invalid descriptors. We'll intialize later. Now just copy the GDT
; image. ;)

       mov si, offset GDTIMGBEGIN
       mov cx, 180h
       mov es, cx              ; GDT begins at segment 018h
       xor di, di              ; offset 0
       mov cx, (offset GDTIMGEND - offset GDTIMGBEGIN)/2
       rep movsw               ; transfer words

; intitialize GDTR and IDTR

       lidt fword ptr idtptr
       lgdt fword ptr gdtptr

; set the PE bit

       mov eax, cr0
       or eax, 1
       mov cr0, eax

       mov ax, 8h
       mov ds, ax
       mov ds:[0], word ptr 0741h

 ; This is the end of our OS

 end_of_world:
                                ; we've already cleared the interrupt flag
        hlt                     ; its a must for HLT to work properly.
dead:   jmp dead                ; bye! its now safe to reboot ur computer.

main    endp

;---------------------------------------------------------------------

        ;--------------------------------------------------
        ;  Procedure conventions:
        ;       * The procedures use no arguments
        ;       * values in ax, dx, are not saved
        ;       * other registers are preserved if used
        ;--------------------------------------------------

;*********************************************************************
;        Procedure disp: displays the asciiz string at [si]
;*********************************************************************

disp    proc    near

        push bx

disploop:

        lodsb
        or al, al
        jz loopend      ; break if al==0
        mov ah, 0Eh
        mov bh, 0       ; page number
        mov bl, 7       ; attribute
        int 10h
        jmp disploop    ; loop forever

loopend:
```

```
        pop bx
        ret

disp    endp

INCLUDE a20Kbd.inc      ; include file to define A20 gating method
                        ; appropriately. includes the following proc

;*********************************************************************
;       Procedure enableA20: Enables the A20 line. sets CF on
;                               failure
;*********************************************************************

INCLUDE prog8259.inc    ; includes the follwing proc

;*********************************************************************
;       Procedure prog8259: reprograms the interrupt controller
;               to use vectors 20h - 27h and 28h - 2Eh
;*********************************************************************

;---------------------------------------------------------------------

DATABEGIN:

        enableMsg       db      'Enable A20 ...',0
        method2msg      db      '. ',0
        prog8259msg     db      'Adjust PIC ...',0
        OKmsg           db      ' :)',13,10,0
        notOKmsg        db      ' :o',0

; Its time to discuss about the IDT and GDT. The IDT starts at physical
; address 0h and is 48*8=180h bytes long coz 32 reserved and 16 h/w
; interrupt lines are present. The GDT follows at 180h and is at its max
; size (8192*8 bytes=64kb)

        idtptr          dw      48*8
                        dd      0
        gdtptr          dw      0FFFFh
                        dd      180h

GDTIMGBEGIN label   byte      ;---------------------

        nulldesc      dd      0
                        dd      0
        videodesc       dw      0FFFFh
                        dw      8000h
                        db      0Bh
                        db      10010010b, 0
                        db      0

GDTIMGEND   label   byte      ;---------------------

org 510

; the boot sector is 512 bytes long. It is identified as a boot
; sector by the presence of 055AAh in the last two bytes. we
; write the id as 0AA55h because of the little endian format of
; the 80x86s.

        _bootid         dw      0AA55h  ; gets overwritten by the stack
                                        ; during execution

code ends

end
```

11. Sample Output



A Bochs window on booting from a floppy image containing the boot loader

(There is not much output since most of the code is concerned with initializing the hardware. The ‘A’ on the top-left of the screen, is the output of the binary executable image)