Assignment 3 (20 points)
Issued 6/3 Due 16/3

Goal: To be familiar with Boot Sectors and the Booting process.

Background:

The disk bootstrap is a short program loaded by the BIOS upon system startup. The BIOS has no information about the environment required by the operating system and therefore can do nothing to initialize the system beyond putting the hardware into a known state. This is where the bootstrap program comes into play. The BIOS loads the bootstrap from a known location and transfers control. Operating system specific bootstraps either load the operating system itself or load a more advanced initialization program. It is the bootstrap's responsibility to load an appropriate operating environment.

The boot sector on a disk is always the first sector on a disk— cylinder (track) 0, head zero, sector one (0:0:1 in CHS disk addressing format). When the computer is powered on (or reset), the BIOS starts up and does the POST (Power On Self Test). It initializes all of its data, and then it looks for a valid boot sector. First it looks at the A: drive, then it looks to C:. If it doesn't find it then interrupt 18h is called, which, on original IBM PCs, started the ROM BASIC. A valid boot sector (to the BIOS) is one that has 0AA55h at offset 510 in the boot sector. A bootstrap must be exactly 512 bytes long because of the two-byte check and the one sector limitation.

When the BIOS finds the boot sector, it reads that sector (512 bytes) off of the disk and into memory at 0:7C00h. Then it jumps to 0:7C00h and the boot sector code gets control. At this point, all that has been initialized is the BIOS data area (40h:0) and the BIOS interrupts (10h - 1Ah). At this point, memory is mostly unused, but not necessarily cleared to 0. At the entry of the bootsector code, the CPU register DL is set to the boot drive from which the bootsector has been loaded (0h = floppy A, 1h = floppy B, 80h = primary hard disk).

The simplest bootstrap can be written as follows:

```
;************************************
ORG     0
hang:   jmp hang ; Hang out...
ORG     510
DW      0AA55h
;************************************
```

To use this code, you must compile it with either MASM or TASM. Link it together as a COM file (Need to use link and exe2bin programs).

The quickest way to put the binary file onto the disk is to use DEBUG.EXE. The following example demonstrates using debug to write at memory offset 100h one sector starting at sector zero on disk zero.

```
C:\DEBUG.EXE BOOTSTRAP.BIN
-W 100 0 0 1
-Q
C:\
```

The command "w" tells debug to write to the disk. It will begin by copying bytes from memory to the designated disk and sector. The final parameter indicates how many sectors to write. The "l" command uses the same parameters but reads from the disk. Together, these can be used to write new bootsectors or examine existing ones. Type "?" to get a listing of more commands.

Note: If you can't compile your code as a COM file, compile it as an EXE, but only write out the last 512 bytes of the file (i.e. skip the EXE file header).

Example 1 below displays a message on the screen, waits for the user to press any key and then reboots. Note the code starts with adjusting the data segment register to cope with the memory location at which the boot sector will be load by the BIOS. Note also that the code can make use of only BIOS interrupts (cannot use INT 21h for example).

```
;************************************
; EXAMPE 1
cseg SEGMENT
            assume CS:cseg, DS:cseg
            ORG 0
            jmp AfterData                ; skip over data

;Put any data here!
msg         db  'Hello Cyberspace!',0

AfterData:
            mov ax, 07C0h           ; necessary house keeping
            mov ds, ax        ; BIOS puts us at 0:07C00h, so set DS accordingly


            mov si, offset msg            ; Print msg
print:
            lodsb                   ; AL=memory contents at DS:SI
            cmp al, 0               ; If AL=0 then get a key and reboot
            je getkey
            mov ah, 0Eh             ; Print AL
            mov bx, 7
            int 10h
            jmp print               ; Print next character

 getkey:
            mov ah, 0               ; wait for key
            int 016h

reboot:
             db 0EAh          ; machine language to jump to FFFF:0000 (cold reboot)
             dw 0000h
             dw 0FFFFh

            ORG 510            ; Make the file 512 bytes long
            dw 0AA55h          ; Add the boot signature
cseg ENDS
end
;************************************
```

Bootstrap programs are put to a variety of uses. They are capable of initializing certain pieces of hardware, putting the processor into advanced operating modes, or performing a dedicated processing task. Usually, however, bootstrap programs are used to load a larger, secondary file into memory with more functionality than can be placed into a 512 byte block. There are two separate techniques for making this happen. The first assumes that the file to be loaded is located immediately after the bootsector on the disk. The bootstrap only needs to know how many sectors to load and can immediately load the appropriate sectors into memory and transfer control to the loaded file. This, however, typically destroys existing file systems on the disk. Although this makes coding the bootsector easier, it is more difficult to put the secondary file onto the disk.

The more advanced solution requires a bootstrap program of greater complexity. A common solution is to write the bootstrap to be compliant with an existing file system (e.g., the FAT12 file system is commonly used on floppy disks). Doing so allows the secondary file to be copied or edited directly on the disk. A bootstrap must therefore be able to browse the file systems to both determine the presence of and the location of the secondary file.

The source code of Example 2 (accompanying this assignment) demonstrates how to read the root directory to search for a secondary file, how to traverse the FAT to load the file into memory, and how to begin executing the loaded code. Note that code defines the BootSector Identification (Data) Block, which keeps the disk readable by the operating system.

Requirements:

1. (***2 pts***) Compile Example 1. Save it to the boot sector of a floppy disk image using a program, such as WinImage (http://www.winimage.com). Test your boot sector by booting the disk image in DOSBox environment. You may use the command "boot image.img –l a" in DOSBox. Check whether the booting disk (after writing the boot sector to) is identifiable by DOS/Windows or not.
2. (***5 pts***) Create a disk bootstrap that outputs "Booting from drive...", identifies the booting drive, awaits a user key-press and then performs **warm** rebooting.
3. (***2 pts***) Compile and test Example 2 as you did for Example 1. You may also use WinImage to create the image of the floppy disk containing your boot sector. Then mount the floppy disk image using the command "imgmount" in DOSBox. Check whether the booting disk (after writing the boot sector to) is identifiable by DOS/Windows or not. Copy the secondary file to it. Then boot your virtual DOS machine from your virtual floppy disk.
4. (***6 pts***) Create a secondary, binary file that simply prints a number of messages on the screen. Write it to the second sector on the startup disk. Then create a bootstrap that loads that secondary program off the floppy disk and jumps to it. In case of any error reading the secondary file, the bootstrap program should warn the user and hang out. (Hint: use **int 13h**, function 0 to reset drive at first, then use **int 13h**, function 2 to read the file from sector 2 to a known memory location, say 1000h:0000h).
5. (***5 pts***) Redo (4) but this time the secondary file can be written at any location on the startup disk, then loaded by the disk bootstrap, and run successfully.