Laboratory Handout # 3: Understanding the PC Boot Proces and Writing a Bootloader

Objectives

At the end of this meeting, students should be able to:

- describe the PC boot process;
- describe the characteristics and purpose of the bootloader;
- write a simple bootloader; and
- write a simple kernel (shell) using assembly language

Discussion

PC Boot Process

1. The processor is pre-programmed to always look at the same place in the system BIOS ROM for the start of the BIOS boot program. This is normally location FFFF0h, right at the end of the system memory. Since there are only 16 bytes left from there to the end of conventional memory, this location just contains a "jump" instruction telling the processor where to go to find the real BIOS startup program.
2. After the BIOS has finished its job, the bootloader from the bootsector will be loaded in the memory location 7C00h. The bootloader program is limited to 512 bytes (which is the size of a typical disk sector).
3. Then, a “jump” instruction to 7C00h is executed. Control is now transferred to the bootloader.
4. The bootloader will perform some initializations which eventually transfers the control to the kernel.

Writing a Simple Bootloader

General Rules:

1. It MUST be 512 bytes long.
2. It MUST end with the bootloader signature, '55 AA'. Without this signature the BIOS won't recognize this as a bootable disk!

Interrupts

Interrupts are used to access BIOS services or OS services. The PC has several interrupts that can be used for displaying characters, accepting input, and others. In writing the bootloader, BIOS interrupts are used since there is still no OS running. Issuing an interrupt is similar to making a function call.

INT 0x10 – BIOS Video Interrupt

INT 0x10 is the BIOS video interrupt. All the display related calls (in theory) are made through this interrupt.
So how do you use it? Well you have to have certain values in certain registers to use it.

AH	0x0E
AL	ASCII character to display
BH	Page number (For most of our work this will remain 0x00)
BL	Text attribute (For most of our work this will remain 0x07) change the value for different colour etc.

Then you call the interrupt once the registers are in place.

INT 0x13,*Service 2 - Read Disk Sectors*

AH	0x02
AL	number of sectors to read (1-128 dec.)
CH	track/cylinder number (0-1023 dec., see below)
CL	sector number (1-17 dec.)
DH	head number (0-15 dec.)
DL	drive number (0=A:, 1=2nd floppy, 80h=drive 0, 81h=drive 1)
ES:BX	pointer to buffer

On return

AH	status (see ~INT 13,STATUS~)
AL	number of sectors read
CF	0 if successful 1 if error

- BIOS disk reads should be retried at least three times and the controller should be reset upon error detection
- be sure ES:BX does not cross a 64K segment boundary or a DMA boundary error will occur
- many programming references list only floppy disk register values
- only the disk number is checked for validity
- the parameters in CX change depending on the number of cylinders; the track/cylinder number is a 10 bit value taken from the 2 high order bits of CL and the 8 bits in CH (low order 8 bits of track):

```
¦F¦E¦D¦C¦B¦A¦9¦8¦7¦6¦5-0¦ CX
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦ +----- sector number
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦ +--------- high order 2 bits of track/cylinder
+------------------------ low order 8 bits of track/cyl number
```

Requirements

- An x86 CPU based computer. (AMD/Intel basicly, most home PC's) 286/286/486...
- And a floppy disk to install the boot sector on. In our case, we are not going to use an actual floppy disk, but a floppy disk image instead.
- And NASM to compile the source code.

Installing a Bootloader

1. Write the bootloader program in assembly language. Name the file as bootsect.asm.
2. Generate the binary of the bootloader: `nasm -o bootsect.bin bootsect.asm`
3. Check the compiled file is 512 bytes big exactly. Use the `ls -l` command
4. Install the boot loader to the first sector of the disk. (Varies per OS)
   `dd if=bootsect.bin bs=512 of=ics-os-floppy.img`

Assembly Language Examples

Nothing boot loader:

```
; Start matter

[BITS 16]          ; Tells the compiler to make this into 16bit code generation
                   ;  code
[ORG 0x7C00]       ; Origin, tells the compiler where the code is going to be
                   ;  in memory after it has been loaded. (hex number)

; End matter
times 510-($-$$) db 0   ; Fill the rest of the sector with zero's
dw 0xAA55               ; Add the boot loader signature to the end
```

Never ending loop boot sector:

```
[BITS 16]    ; 16 bit code
[ORG 0x7C00] ; Code origin set to 7C00

main:        ; Main code label (Not really needed now but will be later)
jmp $        ; Jump to the start of the instruction (never ending loop)
             ; An alternative would be 'jmp main' that would have the exact same
             ;  effect.

; End matter
times 510-($-$$) db 0
dw 0xAA55
```

Character on the screen boot loader:

```
[BITS 16]         ; 16 bit code generation
[ORG 0x7C00]      ; ORGin location is 7C00

;Main program
main:             ; Main program label

mov ah,0x0E       ; This number is the number of the function in the BIOS to run.
                  ;  This function is put character on screen function
mov bh,0x00       ; Page number (I'm not 100% sure of this myself but it is best
                  ;  to leave it as zero for most of the work we will be doing)
mov bl,0x07       ; Text attribute (Controls the background and foreground colour
                  ;  and possibly some other options)
                  ;  07 = White text, black background.
                  ; (Feel free to play with this value as it shouldn't harm
                  ;  anything)
mov al,65         ; This should (in theory) put a ASCII value into al to be
                  ;  displayed. (This is not the normal way to do this)
int 0x10          ; Call the BIOS video interrupt.

jmp $             ; Put it into a coninuous loop to stop it running off into
                  ;  the memory running any junk it may find there.

; End matter
times 510-($-$$) db 0   ; Fill the rest of the sector with zeros
dw 0xAA55               ; Boot signature
```

Storing data:

```
[BITS 16]       ; 16 bit code generation

[ORG 0x7C00]    ; Origin of the program. (Start position)

; Main program
main:           ; Put a label defining the start of the main program

 call PutChar   ; Run the procedure

jmp $           ; Put the program into a never ending loop

; Everything here is out of the main program
; Procedures

PutChar:                ; Label to call procedure
 mov ah,0x0E            ; Put char function number (Teletype)
 mov bh,0x00            ; Page number (Ignore for now)
 mov bl,0x07            ; Normal attribute
 mov al,65              ; ASCII character code
 int 0x10               ; Run interrupt
 ret                    ; Return to main program

; This data is never run, not even as a procedure
; Data

TestHugeNum dd 0x00     ; This can be a huge number (1 double word)
                        ;  Upto ffffffff hex
TestLargeNum dw 0x00    ; This can be a nice large number (1 word)
                        ;  Upto ffff hex
TestSmallNum db 0x00    ; This can be a small number (1 byte)
                        ;  Upto ff hex

TestString db 'Test String',13,10,0     ; This is a string (Can be quite long)

; End matter
times 510-($-$$) db 0   ; Zero's for the rest of the sector
dw 0xAA55               ; Bootloader signature
```

Displaying the whole string:

```
[BITS 16]      ; 16 bit code generation

[ORG 0x7C00]   ; Origin location

; Main program
main:           ; Label for the start of the main program

 mov ax,0x0000  ; Setup the Data Segment register
                ; Location of data is DS:Offset
 mov ds,ax      ; This can not be loaded directly it has to be in two steps.
                ; 'mov ds, 0x0000' will NOT work due to limitations on the CPU

 mov si, HelloWorld     ; Load the string into position for the procedure.
 call PutStr    ; Call/start the procedure

jmp $           ; Never ending loop
```

```
; Procedures
PutStr:         ; Procedure label/start
 ; Set up the registers for the interrupt call
 mov ah,0x0E    ; The function to display a chacter (teletype)
 mov bh,0x00    ; Page number
 mov bl,0x07    ; Normal text attribute

.nextchar       ; Internal label (needed to loop round for the next character)
 lodsb          ; I think of this as LOaD String Block
                ; (Not sure if thats the real meaning though)
                ; Loads [SI] into AL and increases SI by one
 ; Check for end of string '0'
 or al,al       ; Sets the zero flag if al = 0
                ; (OR outputs 0's where there is a zero bit in the register)
 jz .return     ; If the zero flag has been set go to the end of the procedure.
                ; Zero flag gets set when an instruction returns 0 as the answer.
 int 0x10       ; Run the BIOS video interrupt
 jmp .nextchar  ; Loop back round tothe top
.return         ; Label at the end to jump to when complete
 ret            ; Return to main program

; Data

HelloWorld db 'Hello World',13,10,0

; End Matter
times 510-($-$$) db 0   ; Fill the rest with zeros
dw 0xAA55               ; Boot loader signature
```