

# Writing A Bootsector

Jeff Weeks

Writing your own boot sector is probably actually easier then you think. All you really need to know is how the Intel processor boots up. A valid boot sector has the code 0xAA55 at an offset of 510, and is located in the very first sector of the disk. Therefore, the BIOS simply checks drive 0 (A:) for this code. If not found, it then checks drive 128 (C:). If a valid boot sector is found, it is loaded into memory at location 0:07C00h.

So, all you have to do is write a boot sector, assemble it into a plain binary file (there is no format or header to a boot sector), and write it to the first sector of your disk. The best way to do that would be to either use nasm (The netwide assembler can produce plain binary files) or assemble into a DOS .EXE

and remove the first 512 bytes. You can also write your own program to write the bootsector to sector 1 of the disk using BIOS INT 13h AH=02h.

Pretty simple eh? Well, in case you're still a little confused, here's a little bootsector from PolyOS that simply switches to protected mode, after checking that you have a 386+ computer. Actually, it even loads in the PolyFS superblock and checks if it's valid, but that's about it. Soon it'll load in the kernel and jump to it. The bootsector was written with Nasm.

```
; -----
; PolyOS boot loader code          (c)1997 Jeff Weeks of Code X Software
; -----
; This little bit of assembly is the boot loader for my operating system.
; -----

[BITS 16]      ; the bios starts out in 16-bit real mode
[ORG 0]

; -----
; SECTOR ONE: THE BOOT LOADER
; -----
; This sector detects your processor.  If a 386 is found, it loads the
; kernel from the disk and executes it (atleast it will in the future :).
; -----

jmp start      ; skip over our data and functions

; -----
; Data used in the boot-loading process
; -----
bootdrv        db 0
bootmsg        db 'Booting PolyOS (c)1997 Cipher of Code X',13,10,0
loadmsg        db 'Loading kernel',13,10,0
jumpmsg        db 'Jumping to kernel',13,10,0
rebootmsg      db 'Press any key to reboot',13,10,0

; these are used in the processor identification
processormsg   db 'Checking for 386+ processor: ',0
need386        db 'Sorry... 386+ required!',13,10,0
found386       db 'Excellent!',13,10,0

; these are used when entering protected mode
a20msg         db 'Setting A20 address line',13,10,0
pmodemsg       db 'Setting CR0 -> Entering PMode',13,10,0

; Here's the locations of my IDT and GDT.  Remember, Intel's are
```

# Writing A Bootsector

Jeff Weeks

```
; little endian processors, therefore, these are in reversed order.
; Also note that lidt and lgdt accept a 32-bit address and 16-bit
; limit, therefore, these are 48-bit variables.
```

```
pIDT          dw 7FFh          ; limit of 256 IDT slots
              dd 0000h        ; starting at 0000
```

```
pGDT          dw 17FFh        ; limit of 768 GDT slots
              dd 0800h        ; starting at 0800h (after IDT)
```

```
; -----
; Functions used in the boot-loading process
```

```
; -----
detect_cpu:
    mov si, processormsg      ; tell the user what we're doing
    call message

    ; test if 8088/8086 is present (flag bits 12-15 will be set)
    pushf                    ; save the flags original value

    xor ah,ah                 ; ah = 0
    push ax                   ; copy ax into the flags
    popf                      ; with bits 12-15 clear

    pushf                     ; Read flags back into ax
    pop ax
    and ah,0f0h               ; check if bits 12-15 are set
    cmp ah,0f0h
    je no386                  ; no 386 detected (8088/8086 present)

    ; check for a 286 (bits 12-15 are clear)
    mov ah,0f0h               ; set bits 12-15
    push ax                   ; copy ax onto the flags
    popf

    pushf                     ; copy the flags into ax
    pop ax
    and ah,0f0h               ; check if bits 12-15 are clear
    jz no386                  ; no 386 detected (80286 present)
    popf                      ; pop the original flags back

    mov si, found386
    call message

    ret                       ; no 8088/8086 or 286, so atleast 386
no386:
    mov si,need386            ; tell the user the problem
    call message
    jmp reboot                ; and reboot when key pressed
```

```
; -----
message:
    ; Dump ds:si to screen.
    lodsb                    ; load byte at ds:si into al
    or al,al                 ; test if character is 0 (end)
    jz done
    mov ah,0eh                ; put character
    mov bx,0007               ; attribute
    int 0x10                  ; call BIOS
```

# Writing A Bootsector

Jeff Weeks

```
        jmp message
done:
        ret
; -----
getkey:
        mov ah, 0                ; wait for key
        int 016h
        ret
; -----
reboot:
        mov si, rebootmsg       ; be polite, and say we're rebooting
        call message
        call getkey             ; and even wait for a key :)

        db 0EAh                 ; machine language to jump to FFFF:0000
(reboot)
        dw 0000h
        dw 0FFFFh
        ; no ret required; we're rebooting! (Hey, I just saved a byte :)
; -----
; The actual code of our boot loading process
; -----
start:
        mov ax,0x7c0            ; BIOS puts us at 0:07C00h, so set DS accordingly
        mov ds,ax              ; Therefore, we don't have to add 07C00h to all our data

        mov [bootdrv], dl      ; quickly save what drive we booted from

        cli                     ; clear interrupts while we setup a stack
        mov ax,0x9000          ; this seems to be the typical place for a stack
        mov ss,ax
        mov sp,0xffff          ; let's use the whole segment. Why not? We can :)
        sti                     ; put our interrupts back on

        ; Interestingly enough, apparently the processor will disable
        ; interrupts itself when you directly access the stack segment!
        ; Atleast it does in protected mode, I'm not sure about real mode.

        mov si,bootmsg         ; display our startup message
        call message

        call detect_cpu        ; check if we've got a 386

.386        ; use 386 instructions from now on (I don't want to manually include
        ; operand-size(66h) or address-size(67h) prefixes... it's annoying :)

        mov si,loadmsg         ; tell the user we're loading the kernel
        call message
        call getkey

read_me:
        ; first, reset the disk controller
        xor ax, ax
        int 0x13
        jc reboot              ; reboot on error
```

# Writing A Bootsector

Jeff Weeks

```
; then load in the PolyFS superblock
mov ax,0x09000          ; superblock goes to 9000:0000 (above stack)
mov es,ax
xor bx,bx

; I could condense a few of these high/low 8-bit movs into one 16-bit
; mov, but, for simplicity, I'll leave it as is, unless necessary.
mov ax,0x0202          ; load one block (two sectors)
mov ch,0               ; cylinder = 0
mov cl,3               ; sector = 2 (starts at sector 1 not 0)
mov dh,0               ; head = 0 = side one
mov dl,[bootdrv]      ; disk = what we booted from
int 0x13               ; read it
jc read_me             ; if there's an error then we'll try again.
                       ; Often there is not error but requires a few
                       ; tries. Ofcourse, this may end up as an
                       ; infinite loop... but only on a bad disk...

; Check if we have a valid super block (BTW: ES still equals 0x9000)
mov di, 0              ; offset of PolyFS magic signature
mov si, polymagic     ; offset of PolyFS magic to check for (in ds)
cmpsw                  ; compare ES:[DI] with DS:[SI]
jnz reboot            ; reboot on error (otherwise, we've got a PolyFS)

; Ideally, we'd load the kernel right here

mov si, a20msg         ; tell the user we're setting the A20 line
call message

; set A20 line
cli                    ; no more interuptions! :)
xor cx, cx

clear_buf:
in al, 64h             ; get input from keyboard status port
test al, 02h           ; test the buffer full flag
loopnz clear_buf      ; loop until buffer is empty
mov al, 0D1h           ; keyboard: write to output port
out 64h, al            ; output command to keyboard

clear_buf2:
in al, 64h             ; wait 'till buffer is empty again
test al, 02h
loopnz clear_buf2
mov al, 0dfh           ; keyboard: set A20
out 60h, al            ; send it to the keyboard controller
mov cx, 14h

wait_kbc:               ; this is approx. a 25uS delay to wait
out 0edh, ax           ; for the kb controler to execute our
loop wait_kbc          ; command.

; the A20 line is on now. Let's load in our ITD and GDT tables...
; Ideally, there will actually be data in their locations (by loading
; the kernel)
lidt [pIDT]
lgdt [pGDT]

; now let's enter pmode...
```

# Writing A Bootsector

Jeff Weeks

```
mov si, pmodemsg
call message
call getkey

mov eax, cr0          ; load the control register in
or  al, 1            ; set bit 1: pmode bit
mov cr0, eax         ; copy it back to the control register
jmp $+2              ; and clear the prefetch queue
nop
nop

; jump to the kernel that we've loaded in...
; For now, we'll actually just reboot (this really doesn't
; work in protected mode, but it does reboot :)
db 0xEA
dw 0x0000
dw 0xFFFF

; The boot sector is supposed to have to have 0xAA55 at the end of
; the sector (the word at 510 bytes) to be loaded by the BIOS...
times 510-($-$$) db 0
dw 0xAA55
```