Bochs Boot Disk "Hello World!" Tutorial in FASM

Knowledge Required

It's assumed that the reader has knowledge of the registers and opcodes of the Intel processor. These can be found at the Intel website for free in documents that explain each opcode thoroughly as well as everything related to the IA-32 architecture. It's also assumed that the reader is familiar with the documentation for FASM and how its syntax works. Furthermore, it is assumed that the reader can set up the Bochs emulator to boot up using a floppy disk image file. If not then be sure to read the Bochs documentation to find out how.

Tools Used

Bochs (Emulator) [http://bochs.sourceforge.net/]
FASM (Assembler) [http://www.flatassembler.net/]
WinImage (Floppy Disk Image Creator) [http://www.winimage.com/]

Bochs and FASM are both freeware. Bochs is needed in order to emulate a PC on your PC and thus allow you to test the procedure (you could of course use your own PC and actually create a boot disk, but the restarting of your computer would likely get annoying). FASM is an assembly compiler and is what the code of this tutorial is written in. WinImage allows you to create a floppy disk image on your computer and it also lets you write to the boot sector of the disk. WinImage however, is not freeware. There are other freeware available on the net which will allow you to do this so feel free to search around.

The Boot Process

For the purposes of this tutorial we will only analyze the boot process as it relates to a 1.44 MB floppy disk. When Bochs powers on it emulates the functions of a computer upon booting. It performs a series of tests to ensure that all the hardware is functioning properly, initializes memory to certain parameters along with the system and hardware interrupts necessary for basic operation. Once this is done the BIOS then searches the hardware for a bootable sector (typically located in a floppy/hard/CD drive). In the case of a floppy, the bootable sector is the first 512 bytes of data on the disk at Cylinder 0, Head 0. The boot sector must meet certain criteria in order to be successfully loaded into RAM and assume control of the boot process. If all the criterion are met then the sector will be loaded into memory at the address 0000:7C00h.

Boot Sector Structure

The boot sector can be divided into 4 sections:

Offset (hex)	Data
0000 - 0002	Jump to Boot Code
0003 - 003D	Disk Parameters
003E - 01FD	Boot Code
01FE - 01FF	Signature

Section 1
The first assembly instruction in the boot sector must be a near JMP to the address 7C3Eh. Immediately followed by a NOP instruction.

Section 2
This next section contains a series of bytes of information that specify the type of drive that the boot sector was loaded from (for an in-depth explanation [http://www.exegesis.uklinux.net/gandalf/encrypt/disk.htm]). For the sake of simplicity in our tutorial we will simply fill this section with 59 bytes worth of 1's.

Section 3
This section contains all the instructions that will display the message "Hello World!" on the screen. This is where all the real work takes place, from formatting disks to launching and installing operating systems.

Section 4
It is necessary to pad the code so that it reaches 510 bytes in length. Then, the boot sector's final two bytes (the 511th and 512th byte) must contain the value 55AAh which is located at address 7DFEh. This is often referred to as the "magic number."

The Boot Code

`jmp boot_code` `nop`	Section 1
`times 59 db 0ffh`	Section 2

`boot_code:`
`mov si, message + 07c00h`
`@@:`
`lodsb`
`or al, al`
`jz @f`
`mov ah, 0eh`
`mov bx, 07h`
`int 10h`
`jmp @b`
`@@:`
`nop`
`jmp @b`
 	
`message db 'Hello World!', 00h`
	Section 3

`times 510-$ db 00h` `db 055h, 0aah`	Section 4

Section 1

jmp boot_code	Jumps to the section that contains the boot code.
nop	Does nothing.

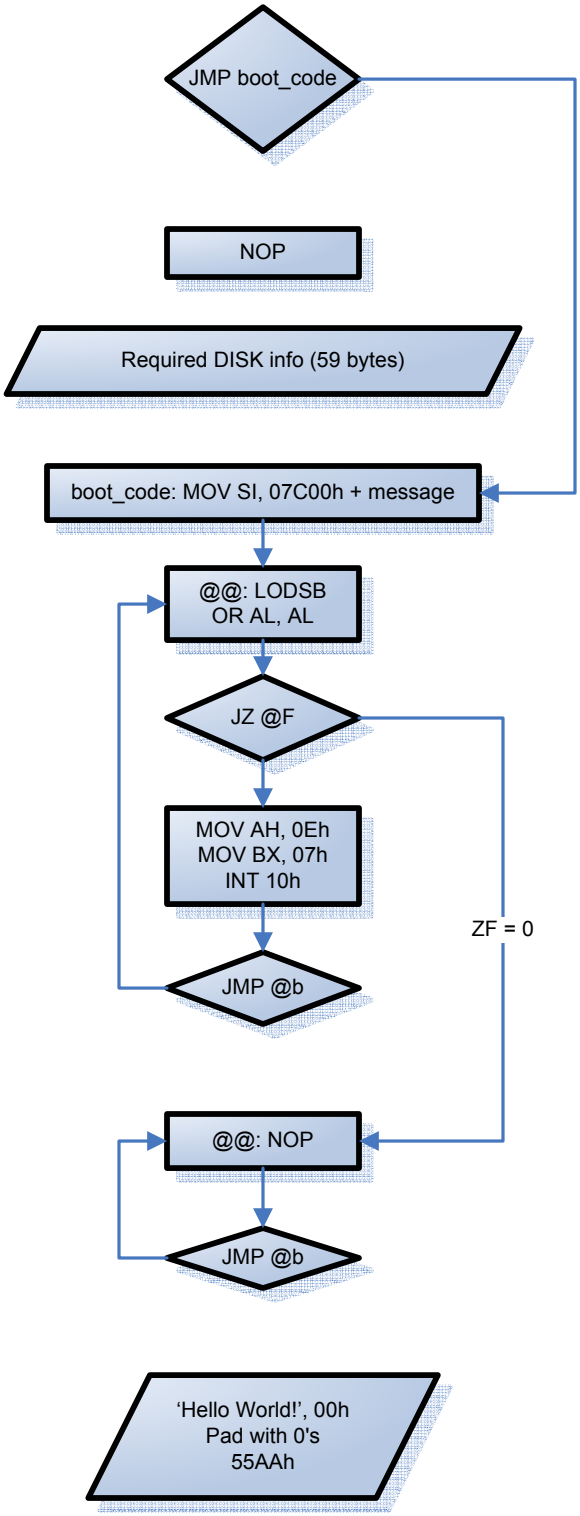
Section 2

times 59 db 0ffh	None of these statements are recognized opcodes, they are compiler directives particular to FASM. *times 59* means repeat the following code 59 times. d*b 0ffh* means allocate a byte filled with 1's. Therefore, there will be 59 bytes filled with 1's.

Section 3	
boot_code:	A label that highlights where the boot code begins. The *jmp* instruction from section 1 brings execution here.
mov si, message + 07c00h	Moves the address of the location of the string message into the SI register. The SI register is the source pointer for string operations.
@@:	An anonymous label (FASM specific).
lodsb	Loads the byte pointed to by the SI register into the AL register, and then changes the SI register to point to the next byte of the string.
or al, al	ORs the AL register with itself.
jz @f	Jumps to the next label. If the zero flag (ZF) is set. @f means the next anonymous label forward of this position (@f is FASM specific).
mov ah, 0eh	Moves the value 0Eh into the AH register. The AH register holds the value which indicates to the BIOS interrupt 10h to set up the hardware for teletype output.
mov bx, 07h	Moves the value 07h into the BX register. The BX register holds the page number and text color for the teletype output of the 10h BIOS interrupt.
int 10h	Calls the 10h interrupt routine which displays a character on the screen based on the values of the AX and BX register.
jmp @b	Jumps to the previous label. @b means the previous anonymous label from this position (@b is FASM specific).
@@:	Another anonymous label.
nop	Does nothing.
jmp @b	See previous *jmp @b* for explanation.
message db 'Hello World!', 00h	This creates a series of contiguous bytes using the *db* directive that contain the string *Hello World!* terminated with a *null* (00h) character. *message* is the label that denotes the position in memory where this string starts.
Section 4	
times 510 - $ db 00h	This is another series of FASM directives that creates a contiguous sequence of bytes filled with 0's.
db 055h, 0aah	This appends the "magic number" to the end of the boot sector.

Code Logic

Sections 1 and 2 satisfy requirements instilled by the BIOS of Bochs. Section 3 however is all user logic and there are many ways to achieve the same effect. Following is a diagram that illustrates what is occurring.


JMP boot_code
NOP
Required DISK info (59 bytes)
boot_code: MOV SI, 07C00h + message
@@: LODSB
OR AL, AL
JZ @F
MOV AH, 0Eh
MOV BX, 07h
INT 10h
JMP @b
ZF = 0
@@: NOP
JMP @b
'Hello World!', 00h
Pad with 0's
55AAh

The logic behind section 3 is this, the SI register points to the place where *'Hello World!'* is located in memory. Therefore, we retrieve each character one at a time, test it to see if its the null character and if it's not display it on the screen by making use of a specific BIOS interrupt. When we encounter the null character we simply transfer program execution to a non-terminating loop. In section 4 we must meet the requirement that the boot sector be exactly 512 bytes long and that it end in 55AAh. By ignoring the signature we know that there are 510 bytes of other data. Therefore, we want to fill in 510 bytes minus whatever our code takes up in bytes with 0's. The FASM direction *times 510 - $ db 00h* does just that. It allots a space in bytes equal to 510 - $ filled with 0's. The $ character in FASM (in this particular case) is interpreted as the number of bytes up to that point. This now makes our compiled code 510 bytes long so all we have left to do is declare two bytes, 55h and AAh and compile. You now have a functional boot strapper for Bochs (and any other similar PC system).

Other Stuff

The 10h interrupt knows to advance the cursor position automatically when a character is displayed on the screen.
When FASM compiles the output file will be 512 bytes long. This binary file must then be written to the boot sector of a floppy disk image.
The floppy disk image must be placed in the Bochs directory and designated as the boot disk for the simulation to work successfully.

Author

http://grimpirate.t35.com/