# How To Program a Bootstrap Loader

## CAUTION

A word of caution, tampering with the bootsector of a computer can render the machine inoperable. It is advisab using floppy disks or non-critical hard drives prior to transferring your bootsector onto a live system. The follov boot any x86 class PC with a floppy disk drive. USB floppies in legacy mode and El Torrito CDROM images w successfully as well.

## Introduction

There are many reasons for writing a custom bootstrap program. Most prominent is to achieve an increased und how a computer operates in its rawest form. Programmers that desire to write their own operating systems will g custom bootstrap code to load and initialize their system. Utilities that allow users to select different operating s boot time require a fundamental knowledge of the computer boot sequence. Data recovery services performing must understand different bootsector formats in order to restore and read lost data. All of these reasons boil bacl understanding the first step a computer makes when powering up - the bootstrap.

The bootstrap is a short program loaded by the BIOS (Basic Input Output System) upon system startup. The BI( information about the environment required by the operating system and therefore can do nothing to initialize th putting the hardware into a known state.[1] This is where the bootstrap program comes into play. The BIOS loads from a known location and transfers control. Operating system specific bootstraps either load the operating syste perform a multi-stage boot by loading a more advanced initialization program. It is the bootstrap's responsibility and build an appropriate operating environment.[2]

## Bootstrap Basics

A bootstrap is loaded from the first sector on a disk, track zero, head zero, sector one. Which disk the bootstrap dependent upon the BIOS configuration saved in NVRAM (NonVolatile RAM). This single 512 byte sector is l memory at physical address 0000:7C00. The BIOS will then examine the final two bytes of the bootstrap (offset value AA55h. This flags the bootsector as a valid, bootable disk instead of just storing disk information. A boot exactly 512 bytes long because of the two byte check and the one sector limitation. After this verification, the B 0000:7C00 and turn control over to the bootstrap.

It is important for the programmer to note the processor is operating in 16bit Real Mode when control is transfe Programming considerations must be made to ensure that segment registers are initialized and that indexed addr do not violate 64KB boundaries. Bootstraps generally do not perform processor mode changes, but that does no to 32bit Protected Mode is impossible.[4] Typically, a second stage loader is employed for more exhaustive confi system because the code will no longer be constrained to the 512 byte limitation.

The simplest bootstrap can be written as follows:

```
;*************************************
[BITS 16]
ORG     0
INT     0x18

TIMES   510-($-$$) DB 0
DW      0xAA55
;*************************************
```

Henceforth, all code examples will be referred to as bootstrap.asm. This example can be compiled using NASM Netwide Assembler available on-line.[5] The code must be compiled into plain binary:

```
NASM -O BOOTSTRAP.BIN BOOTSTRAP.ASM
```

The quickest way to put the binary file onto the disk is to use DEBUG.EXE. The following example demonstrat 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 designate sector. The final parameter indicates how many sectors to write. The "L" command uses the same parameters bu disk. Together, these can be used to write new bootsectors or examine existing ones. Type "?" to get a listing of while using the debugger.[6]

Under the various versions of UNIX, the DD command can be used to copy the image onto the bootsector. Type detailed usage information.

```
dd if=BOOTSTRAP.BIN of=/dev/fd0
```

## Enhancing the Bootstrap

Bootstrap programs are put to a variety of uses. They are capable of initializing certain pieces of hardware, putti into advanced operating modes, or performing a dedicated processing task. Usually, however, bootstrap progran load a larger file into memory with more functionality than can be placed into a 512 byte block. There are two s techniques for making this happen. The first assumes that the file to be loaded is located immediately after the b disk. The bootstrap only needs to know how many sectors to load and can immediately load the appropriate sect and transfer control to the loaded file. This, however, typically destroys existing file systems on the disk. Despit between the myriads of file systems available, their physical implementations on hard disks share common trend sector, sharing space with the bootstrap code, contains information on where to find additional file system struct Oftentimes, the additional structures have static locations on the disk. Thus, while placing the files to be loaded after the first sector will make coding the bootsectoor easier, it becomes increasingly difficult to implement a wo system on the disk afterward.

The more advanced solution requires a bootstrap program of greater complexity. A common solution is to write be compliant with an existing file system.[7] Doing so allows the bootstrap's target files to be copied or edited dir A bootstrap must therefore be able to browse the file systems to both determine the presence of and the location file.

The FAT12 file system is commonly used on floppy disks. There are two data blocks inherent to FAT12 format OEM_ID string and the BIOS Parameter Block. The OEM_ID string serves no purpose other than to identify wl performed the disk format. The BIOS Parameter Block, referred to in Microsoft documentation as the BPB, is a containing fields that describe the physical attributes of the disk.[8] These attributes can be used to determine char as the disk's total capacity. The BIOS immediately begins executing the code loaded at 0000:7C00, which inclu of data. Therefore, the first step for the bootstrap loader is to perform a JMP operation code located below these blocks.

Locating the target file requires understanding how the FAT file system works.[9] FAT12 organizes the disk into array and calls each data cell a cluster. Thus, depending on how the disk was formatted, the number of sectors p vary. A structure known as the FAT (File Allocation Table) keeps track of each cluster on the disk. The FAT its integer value indicating the next cluster in the file. By following the chain of indexes until the EOF (End Of File located, it is possible to locate the contents of any file regardless of fragmentation. FAT12 has one additional str Root Directory, responsible for indexing filenames against their first cluster. Putting everything together, a file i searching for its name in the root directory and then following a chain of indexes through the FAT to identify th disk sectors it is saved on.[10]

## Source Code

The source code example below demonstrates how to read the root directory to search for the file, how to traver load the file into memory, and how to begin executing the loaded code.

```
;**************************************************************************
[BITS 16]
ORG 0
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_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, ImageName                       ; 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     si, msgCRLF
     call    DisplayMessage
     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 (0050:0000)
     mov     si, msgCRLF
     call    DisplayMessage
     mov     ax, 0x0050
     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:
     mov     si, msgCRLF
     call    DisplayMessage
     push    WORD 0x0050
     push    WORD 0x0000
     retf
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□2; 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 "LOADER  BIN"
msgLoading  db 0x0D, 0x0A, "Loading Boot Image ", 0x0D,
0x0A, 0x00
msgCRLF     db 0x0D, 0x0A, 0x00
msgProgress db ".", 0x00
```

```
msgFailure  db 0x0D, 0x0A, "ERROR : Press Any Key to Reboot", 0x00

     TIMES 510-($-$$) DB 0
     DW 0xAA55
;*************************************************************************
```

## Breaking Down The Code

The source code above is a simple example, following the top-down programming approach for event sequencir are four basic functions to minimize code used for repeated services. *DisplayMessage* utilizes BIOS routines to the screen for keeping the user informed of progress. *ReadSectors* utilizes BIOS routines to read raw data from memory. *ClusterLBA* converts Microsoft's cluster addressing scheme into a Logical Block Address for mapping disk. *LBACHS* converts the Logical Block Address into the Cylinder Head Sector format understood by the BIC accessing the file.[11]

The main program body, identified as *START:* begins by initializing the processor's registers. This step is impor known operating environment. Errant values in the CPU registers may induce unintended side effects when mak BIOS functions. With the CPU in a known state, the code calculates the Root Directory's location from the valu BPB and loads it into memory. Looping through the Root Directory will identify the first cluster of LOADER.B file to load. The next step uses the BPB to locate the FAT and load it into memory for browsing. Using the first LOADER.BIN, the bootstrap browses the FAT and makes calls to *ReadSectors* to load the file into memory. At conclusion, control is passed to LOADER.BIN via a RETF operation.

## Conclusion

The bootsector is a simple, yet critical programming component in a computer system. Experimenting with the code will reveal techniques for loading different file systems, creating boot-time loaders or debugging a comput operating system. Security professionals are wise to understand low level coding, a domain frequently exploited hackers. Overall, the understanding of a computer's most primitive operations helps programmers develop bette increases understanding users have of their systems.[12]

---

### *Notes*


1. Owen, Gareth, "OS Development," *SourceForge.net*, accessed November 2005 at http://gaztek.sourceforge
2. Fine, John, *John Fine's Homepage* accessed November 2005 at http://my.execpc.com/~geezer/johnfine/ ↑
3. mjvines, "Quick Hack Bootsector," *OSDev Community*, accessed November 2005 at http://www.osdcom.info/download.php?view.18 ↑
4. Weeks, Jeff, "PolyOS Boot Loader Code," *the kid operating system*, accessed November 2005 at http://kos.enix.org/pub/bootwrit.html ↑
5. "Netwide Assembler," *SourceForge.net*, accessed November 2005 at http://nasm.sourceforge.net/doc/html/ ↑
6. "PC Computer Debug Routines," *Computer Hope*, accessed November 2005 at http://www.computerhope. ↑
7. Tash, Sean, "NYAOS Boot Sector," *SourceForge.net*, accessed November 2005 at http://gaztek.sourceforg

8. "Detailed Explanation of FAT Boot Sector," *Microsoft*, accessed November 2005 at http://support.microsoft.com/kb/q140418/ ↑
9. Norton, Peter, *Inside the IBM PC Access To Advanced Features and Programming*, Prentice-Hall Publishi 1983 ↑
10. Jourdain, Robert, *Programmer's Problem Solver For The IBM PC, XT & AT*, Simon & Schuster Publishing NY. 1986 ↑
11. Bass Demon, *Operating System Resources*, accessed June 1999 at http://home.c2i.net/tkjoerne/os/ ↑
12. Vea, Matthew, *Bootstrap Tutorial*, Home Page, June 1999, accessed November 2005 at http://www.geocities.com/mvea/bootstrap.htm This article is first appeared on the author's personal homep