X86 Assembly/Bootloaders

When a computer is turned on, there is some beeping, and some flashing lights, and then a loading screen appears. And then magically, the operating system loads into memory. The question is then raised, how does the operating system load up? What gets the ball rolling? The answer is **bootloaders**.

Contents

What is a Bootloader?

Bootloaders are small pieces of software that play a role in getting an operating system loaded and ready for execution when a computer is turned on. The way this happens varies between different computer designs (early computers required a person to manually set the computer up whenever it was turned on), and often there are several stages in the process of boot loading.

It's crucial to understand that the term "bootloader" is simply a classification of software (and sometimes a blurry one). To the processor, a bootloader is just another piece of code that it blindly executes. There are many different kinds of boot loaders. Some are small, others are large; some follow very simple rules while others show fancy screens and give the user a selection to choose from.

On IBM PC compatibles, the first program to load is the Basic Input/Output System (BIOS). The BIOS performs many tests and initialisations, and if everything is OK, the BIOS's boot loader begins. Its purpose is to load another boot loader! It selects a disk (or some other storage media) from which it loads a secondary boot loader.

In some cases, *this* boot loader loads enough of an operating system to start running it. In other cases, it loads yet another boot loader from somewhere else. This often happens when multiple operating systems are installed on a single computer; each OS may have its own specific bootloader, with a "central" bootloader that loads one of the specific ones according to the user's selection.

Most bootloaders are written exclusively in assembly language (or even machine code), because they need to be compact, they don't have access to OS routines (such as memory allocation) that other languages might require, they need to follow some unusual requirements, and they make frequent use of low-level features. However some bootloaders, particularly those that have many features and allow user input, are quite heavyweight. These are often written in a combination of assembly and C. The GRand Unified Bootloader (http://www.gnu.org/software/grub/) (GRUB) is an example of such.

Some boot loaders are highly OS-specific, while others are less so - certainly the BIOS boot loader is not OS-specific. The MS-DOS boot loader (which was placed on *all* MS-DOS formatted floppy disks) simply checks if the files **IO.SYS** and **MSDOS.SYS** exist; if they are not present it displays the error "Non-System disk or disk error" otherwise it loads and begins execution of **IO.SYS**.

The final stage boot loader may be expected (by the OS) to prepare the computer in some way, for instance by placing the processor in protected mode and programming the interrupt controller. While it would be possible to do these things inside the OS's initialisation procedure, moving them into the bootloader can simplify the OS design. Some operating systems require their bootloader to set up a small, basic GDT (Global Descriptor Table) and enter protected mode, in order to remove the need for the OS to have any 16-bit code. However, the OS might replace this with its own sophisticated GDT soon after.

The Bootsector

The first 512 bytes of a disk are known as the **bootsector** or **Master Boot Record**. The boot sector is an area of the disk reserved for booting purposes. If the bootsector of a disk contains a valid boot sector (the last word of the sector must contain the signature 0xAA55), then the disk is treated by the BIOS as bootable.

The Boot Process

When switched on or reset, an x86 processor begins executing the instructions it finds at address FFFF:0000 (at this stage it is operating in **Real Mode**). In IBM PC compatible processors, this address is mapped to a ROM chip that contains the computer's Basic Input/Output System (BIOS) code. The BIOS is responsible for many tests and initialisations; for instance the BIOS may perform a memory test, initialise the interrupt controller and system timer, and test that these devices are working.

Eventually the actual boot loading begins. First the BIOS searches for and initialises available storage media (such as floppy drives, hard disks, CD drives), then it decides which of these it will attempt to boot from. It checks each device for availability (e.g. ensuring a floppy drive contains a disk), then the 0xAA55 signature, in some predefined order (often the order is configurable using the BIOS setup tool). It loads the first sector of the first bootable device it comes across into RAM, and initiates execution.

Ideally, this will be another boot loader, and it will continue the job, making a few preparations, then passing control to something else.

While BIOSes remain compatible with 20-year-old software, they have also become more sophisticated over time. Early BIOSes could not boot from CD drives, but now CD and even DVD booting are standard BIOS features. Booting from USB storage devices is also possible, and some systems can boot from over the network. To achieve such advanced functioning, BIOSes sometimes enter protected mode and the like, but then return to real mode in order to be compatible with legacy boot loaders. This creates a chicken-and-egg problem: bootloaders are written to work with the ubiquitous BIOS, and BIOSes are written to support all those bootloaders, preventing much in the way of new boot loading features.

However, a new bootstrap technology, the EFI, is beginning to gain momentum. It is much more sophisticated and will not be discussed in this article.

Note also that other computer systems - even some that use x86 processors - may boot in different ways. Indeed, some embedded systems whose software is compact enough to be stored on ROM chips may not need bootloaders at all.

Technical Details

A bootloader runs under certain conditions that the programmer must appreciate in order to make a successful bootloader. The following pertains to bootloaders initiated by the PC BIOS:

1. The first sector of a drive contains its boot loader.
2. One sector is 512 bytes — the last two bytes of which *must* be 0xAA55 (i.e. 0x55 followed by 0xAA), or else the BIOS will treat the drive as unbootable.
3. If everything is in order, said first sector will be placed at RAM address 0000:7C00, and the BIOS's role is over as it transfers control to 0000:7C00 (that is, it JMPs to that address).
4. The DL register will contain the drive number that is being booted from, useful if you want to read more data from elsewhere on the drive.
5. The BIOS leaves behind a lot of code, both to handle hardware interrupts

(such as a keypress) and to provide services to the bootloader and OS (such as keyboard input, disk read, and writing to the screen). You must understand the purpose of the Interrupt Vector Table (IVT), and be careful not to interfere with the parts of the BIOS that you depend on. Most operating systems replace the BIOS code with their own code, but the boot loader can't use anything but its own code and what the BIOS provides. Useful BIOS serve include `int 10h` (for displaying text/graphics), `int 13h` (disk functions) and `int 16h` (keyboard input).

6. This means that any code or data that the boot loader needs must either be included in the first sector (be careful not to accidentally execute data) or manually loaded from another sector of the disk to somewhere in RAM. Because the OS is not running yet, most of the RAM will be unused. However, you must take care not to interfere with the RAM that is required by the BIOS interrupt handlers and services mentioned above.
7. The OS code itself (or the next bootloader) will need to be loaded into RAM as well.
8. The BIOS places the stack pointer 512 bytes beyond the end of the boot sector, meaning that the stack cannot exceed 512 bytes. It may be necessary to move the stack to a larger area.
9. There are some conventions that need to be respected if the disk is to be readable under mainstream operating systems. For instance you may wish to include a BIOS Parameter Block on a floppy disk to render the disk readable under most PC operating systems.

Most assemblers will have a command or directive similar to `ORG 7C00h` that informs the assembler that the code will be loaded starting at offset 7C00h. The assembler will take this into account when calculating instruction and data addresses. If you leave this out, the assembler assumes the code is loaded at address 0 and this must be compensated for manually in the code.

Usually, the bootloader will load the kernel into memory, and then jump to the kernel. The kernel will then be able to reclaim the memory used by the bootloader (because it has already performed its job). However it is possible to include OS code within the boot sector and keep it resident after the OS begins.

Here is a simple bootloader demo designed for NASM:

```
        org 7C00h

        jmp short Start ;Jump over the data (the 'short' keyword makes the jmp instruction smaller)

Msg:    db "Hello World! "
EndMsg:

Start:  mov bx, 000Fh   ;Page 0, colour attribute 15 (white) for the int 10 calls below
        mov cx, 1       ;We will want to write 1 character
        xor dx, dx      ;Start at top left corner
        mov ds, dx      ;Ensure ds = 0 (to let us load the message)
        cld             ;Ensure direction flag is cleared (for LODSB)

Print:  mov si, Msg     ;Loads the address of the first byte of the message, 7C02h in this case

                        ;PC BIOS Interrupt 10 Subfunction 2 - Set cursor position
                        ;AH = 2
Char:   mov ah, 2       ;BH = page, DH = row, DL = column
        int 10h
        lodsb           ;Load a byte of the message into AL.
                        ;Remember that DS is 0 and SI holds the
                        ;offset of one of the bytes of the message.

                        ;PC BIOS Interrupt 10 Subfunction 9 - Write character and colour
                        ;AH = 9
        mov ah, 9       ;BH = page, AL = character, BL = attribute, CX = character count
        int 10h

        inc dl          ;Advance cursor

        cmp dl, 80      ;Wrap around edge of screen if necessary
        jne Skip
        xor dl, dl
        inc dh

        cmp dh, 25      ;Wrap around bottom of screen if necessary
        jne Skip
        xor dh, dh

Skip:   cmp si, EndMsg  ;If we're not at end of message,
        jne Char        ;continue loading characters
        jmp Print       ;otherwise restart from the beginning of the message


times 0200h - 2 - ($ - $$)  db 0    ;Zerofill up to 510 bytes

        dw 0AA55h       ;Boot Sector signature

;OPTIONAL:
;To zerofill up to the size of a standard 1.44MB, 3.5" floppy disk
;times 1474560 - ($ - $$) db 0
```

To compile the above file, suppose it is called 'floppy.asm', you can use following command:

```
nasm -f bin -o floppy.img floppy.asm
```

While strictly speaking this is not a bootloader, it is bootable, and demonstrates several things:

- How to include and access data in the boot sector
- How to skip over included data (this is required for a BIOS Parameter Block)
- How to place the 0xAA55 signature at the end of the sector (NASM will issue an error if there is too much code to fit in a sector)
- The use of BIOS interrupts

On Linux, you can issue a command like

```
cat floppy.img > /dev/fd0
```

to write the image to the floppy disk (the image may be smaller than the size of the disk in which case only as much information as is in the image will be written to the disk). Under Windows you can use software such as RAWRITE.

Hard disks

Hard disks usually add an extra layer to this process, since they may be partitioned. The first sector of a hard disk is known as the Master Boot Record (MBR). Conventionally, the partition information for a hard disk is included at the end of the MBR, just before the 0xAA55 signature.

The role of the BIOS is no different to before: to read the first sector of the disk (that is, the MBR) into RAM, and transfer execution to the first byte of this sector. The BIOS is oblivious to partitioning schemes - all it checks for is the presence of the 0xAA55 signature.

While this means that one can use the MBR in any way one would like (for instance, omit or extend the partition table) this is seldom done. Despite the fact that the partition table design is very old and limited - it is limited to four partitions - virtually all operating systems for IBM PC compatibles assume that the MBR will be formatted like this. Therefore to break with convention is to render your disk inoperable except to operating systems specifically designed to use it.

In practice, the MBR usually contains a boot loader whose purpose is to load another boot loader - to be found at the start of one of the partitions. This is often a very simple program which finds the first partition marked *Active*, loads its first sector into RAM, and commences its execution. Since by convention the new boot loader is also loaded to address 7C00h, the old loader may need to relocate all or part of itself to a different location before doing this. Also, ES:SI is expected to contain the address in RAM of the partition table, and DL the boot drive number. Breaking such conventions may render a bootloader incompatible with other bootloaders.

However, many boot managers (software that enables the user to select a partition, and sometimes even kernel, to boot from) use custom MBR code which loads the remainder of the boot manager code from somewhere on disk, then provides the user with options on how to continue the bootstrap process. It is also possible for the boot manager to reside within a partition, in which case it must first be loaded by another boot loader.

Most boot managers support chain loading (that is, starting another boot loader via the usual first-sector-of-partition-to-address-7C00 process) and this is often used for systems such as DOS and Windows. However, some boot managers (notably GRUB) support the loading of a user-selected kernel image. This can be used with systems such as GNU/Linux and Solaris, allowing more flexibility in starting the system. The mechanism may differ somewhat from that of chain loading.

Clearly, the partition table presents a chicken-and-egg problem that is placing unreasonable limitations on partitioning schemes. One solution gaining momentum is the GUID Partition Table; it uses a dummy MBR partition table so that legacy operating systems will not interfere with the GPT, while newer operating systems can take advantage of the many improvements offered by the system.

GNU GRUB

The GRand Unified Bootloader (http://www.gnu.org/software/grub/) supports the flexible *multiboot* (http://www.gnu.org/software/grub/manual/multiboot/html_node/index.html) boot protocol. This protocol aims to simplify the boot process by providing a single, flexible protocol for booting a variety of operating systems. Many free operating systems can be booted using multiboot.

GRUB is extremely powerful and is practically a small operating system. It can read various file systems and thus lets you specify a kernel image *by filename* as well as separate module files that the kernel may make use of. Command-line arguments can be passed to the kernel as well - this is a nice way of starting an OS in maintenance mode, or "safe mode", or with VGA graphics, and so on. GRUB can provide a menu for the user to select from as well as allowing custom loading parameters to be entered.

Obviously this functionality cannot possibly be provided in 512 bytes of code. This is why GRUB is split into two or three "stages":

- Stage 1 - this is a 512-byte block that has the location of stage 1.5 or stage 2 hardcoded into it. It loads the next stage.
- Stage 1.5 - an optional stage which understands the filesystem (e.g. FAT32 or ext3) where stage 2 resides. It will find out where stage 2 is located and load it. This stage is quite small and is located in a fixed area, often just after Stage 1.
- Stage 2 - this is a much larger image that has all the GRUB functionality.

Note that Stage 1 may be installed to the Master Boot Record of a hard disk, or may be installed in one of the partitions and chainloaded by another boot loader.

Windows can not be loaded using multiboot, but the Windows bootloader (like those of other non-multiboot operating systems) can be chainloaded from GRUB, which isn't quite as good, but does let you boot such systems.

Example of a Boot Loader – Linux Kernel v0.01

```
SYSSIZE=0x8000
|
|       boot.s
|
| boot.s is loaded at 0x7c00 by the bios-startup routines, and moves itself
| out of the way to address 0x90000, and jumps there.
|
| It then loads the system at 0x10000, using BIOS interrupts. Thereafter
| it disables all interrupts, moves the system down to 0x0000, changes
| to protected mode, and calls the start of system. System then must
| RE-initialize the protected mode in it's own tables, and enable
| interrupts as needed.
|
| NOTE! currently system is at most 8*65536 bytes long. This should be no
| problem, even in the future. I want to keep it simple. This 512 kB
| kernel size should be enough - in fact more would mean we'd have to move
| not just these start-up routines, but also do something about the cache-
| memory (block IO devices). The area left over in the lower 640 kB is meant
| for these. No other memory is assumed to be "physical", i.e. all memory
| over 1Mb is demand-paging. All addresses under 1Mb are guaranteed to match
| their physical addresses.
|
| NOTE1 above is no longer valid in it's entirety. cache-memory is allocated
| above the 1Mb mark as well as below. Otherwise it is mainly correct.
|
| NOTE 2! The boot disk type must be set at compile-time, by setting
| the following equ. Having the boot-up procedure hunt for the right
| disk type is severe brain-damage.
| The loader has been made as simple as possible (had to, to get it
| in 512 bytes with the code to move to protected mode), and continuous
| read errors will result in a unbreakable loop. Reboot by hand. It
| loads pretty fast by getting whole sectors at a time whenever possible.

| 1.44Mb disks:
sectors = 18
| 1.2Mb disks:
| sectors = 15
| 720kB disks:
| sectors = 9

.globl begtext, begdata, begbss, endtext, enddata, endbss
.text
begtext:
.data
begdata:
.bss
begbss:
.text

BOOTSEG = 0x07c0
INITSEG = 0x9000
SYSSEG  = 0x1000                        | system loaded at 0x10000 (65536).
ENDSEG  = SYSSEG + SYSSIZE

entry start
start:
        mov     ax,#BOOTSEG
        mov     ds,ax
        mov     ax,#INITSEG
        mov     es,ax
        mov     cx,#256
        sub     si,si
        sub     di,di
        rep
        movw
        jmpi    go,INITSEG
go:     mov     ax,cs
        mov     ds,ax
        mov     es,ax
        mov     ss,ax
        mov     sp,#0x400               | arbitrary value >>512

        mov     ah,#0x03        | read cursor pos
        xor     bh,bh
        int     0x10

        mov     cx,#24
        mov     bx,#0x0007      | page 0, attribute 7 (normal)
        mov     bp,#msg1
        mov     ax,#0x1301      | write string, move cursor
        int     0x10

| ok, we've written the message, now
| we want to load the system (at 0x10000)

        mov     ax,#SYSSEG
        mov     es,ax           | segment of 0x010000
        call    read_it
        call    kill_motor

| if the read went well we get current cursor position ans save it for
| posterity.

        mov     ah,#0x03        | read cursor pos
        xor     bh,bh
        int     0x10            | save it in known place, con_init fetches
        mov     [510],dx        | it from 0x90510.

| now we want to move to protected mode ...

        cli                     | no interrupts allowed !

| first we move the system to it's rightful place

        mov     ax,#0x0000
        cld                     | 'direction'=0, movs moves forward
do_move:
        mov     es,ax           | destination segment
        add     ax,#0x1000
        cmp     ax,#0x9000
        jz      end_move
        mov     ds,ax           | source segment
        sub     di,di
        sub     si,si
        mov     cx,#0x8000
        rep
        movsw
        j       do_move

| then we load the segment descriptors

end_move:

        mov     ax,cs           | right, forgot this at first. didn't work :-)
        mov     ds,ax
        lidt    idt_48          | load idt with 0,0
        lgdt    gdt_48          | load gdt with whatever appropriate

| that was painless, now we enable A20

        call    empty_8042
        mov     al,#0xD1                | command write
        out     #0x64,al
        call    empty_8042
        mov     al,#0xDF                | A20 on
        out     #0x60,al
        call    empty_8042

| well, that went ok, I hope. Now we have to reprogram the interrupts :-(
| we put them right after the intel-reserved hardware interrupts, at
| int 0x20-0x2F. There they won't mess up anything. Sadly IBM really
| messed this up with the original PC, and they haven't been able to
| rectify it afterwards. Thus the BIOS puts interrupts at 0x08-0x0f,
| which is used for the internal hardware interrupts as well. We just
| have to reprogram the 8259's, and it isn't fun.

        mov     al,#0x11                | initialization sequence
        out     #0x20,al                | send it to 8259A-1
        .word   0x00eb,0x00eb           | jmp $+2, jmp $+2
        out     #0xA0,al                | and to 8259A-2
        .word   0x00eb,0x00eb
        mov     al,#0x20                | start of hardware int's (0x20)
        out     #0x21,al
        .word   0x00eb,0x00eb
        mov     al,#0x28                | start of hardware int's 2 (0x28)
        out     #0xA1,al
        .word   0x00eb,0x00eb
        mov     al,#0x04                | 8259-1 is master
        out     #0x21,al
        .word   0x00eb,0x00eb
        mov     al,#0x02                | 8259-2 is slave
        out     #0xA1,al
        .word   0x00eb,0x00eb
        mov     al,#0x01                | 8086 mode for both
        out     #0x21,al
        .word   0x00eb,0x00eb
        out     #0xA1,al
        .word   0x00eb,0x00eb
        mov     al,#0xFF                | mask off all interrupts for now
        out     #0x21,al
        .word   0x00eb,0x00eb
        out     #0xA1,al

| well, that certainly wasn't fun :-(. Hopefully it works, and we don't
| need no steenking BIOS anyway (except for the initial loading :-).
| The BIOS-routine wants lots of unnecessary data, and it's less
| "interesting" anyway. This is how REAL programmers do it.
|
| Well, now's the time to actually move into protected mode. To make
| things as simple as possible, we do no register set-up or anything,
| we let the gnu-compiled 32-bit programs do that. We just jump to
| absolute address 0x00000, in 32-bit protected mode.

        mov     ax,#0x0001      | protected mode (PE) bit
        lmsw    ax              | This is it!
        jmpi    0,8             | jmp offset 0 of segment 8 (cs)

| This routine checks that the keyboard command queue is empty
| No timeout is used - if this hangs there is something wrong with
| the machine, and we probably couldn't proceed anyway.
empty_8042:
        .word   0x00eb,0x00eb
        in      al,#0x64        | 8042 status port
        test    al,#2           | is input buffer full?
        jnz     empty_8042      | yes - loop
        ret

| This routine loads the system at address 0x10000, making sure
| no 64kB boundaries are crossed. We try to load it as fast as
| possible, loading whole tracks whenever we can.
```

```
|
| in:   es - starting address segment (normally 0x1000)
|
| This routine has to be recompiled to fit another drive type,
| just change the "sectors" variable at the start of the file
| (originally 18, for a 1.44Mb drive)
|
sread:  .word 1                 | sectors read of current track
head:   .word 0                 | current head
track:  .word 0                 | current track
read_it:
        mov ax,es
        test ax,#0x0fff
die:    jne die                 | es must be at 64kB boundary
        xor bx,bx               | bx is starting address within segment
rp_read:
        mov ax,es
        cmp ax,#ENDSEG          | have we loaded all yet?
        jb ok1_read
        ret
ok1_read:
        mov ax,#sectors
        sub ax,sread
        mov cx,ax
        shl cx,#9
        add cx,bx
        jnc ok2_read
        je ok2_read
        xor ax,ax
        sub ax,bx
        shr ax,#9
ok2_read:
        call read_track
        mov cx,ax
        add ax,sread
        cmp ax,#sectors
        jne ok3_read
        mov ax,#1
        sub ax,head
        jne ok4_read
        inc track
ok4_read:
        mov head,ax
        xor ax,ax
ok3_read:
        mov sread,ax
        shl cx,#9
        add bx,cx
        jnc rp_read
        mov ax,es
        add ax,#0x1000
        mov es,ax
        xor bx,bx
        jmp rp_read

read_track:
        push ax
        push bx
        push cx
        push dx
        mov dx,track
        mov cx,sread
        inc cx
        mov ch,dl
        mov dx,head
        mov dh,dl
        mov dl,#0
        and dx,#0x0100
        mov ah,#2
        int 0x13
        jc bad_rt
        pop dx
        pop cx
        pop bx
        pop ax
        ret
bad_rt: mov ax,#0
        mov dx,#0
        int 0x13
        pop dx
        pop cx
        pop bx
        pop ax
        jmp read_track

/*
 * This procedure turns off the floppy drive motor, so
 * that we enter the kernel in a known state, and
 * don't have to worry about it later.
 */
kill_motor:
        push dx
        mov dx,#0x3f2
        mov al,#0
        outb
        pop dx
        ret

gdt:
        .word   0,0,0,0         | dummy

        .word   0x07FF          | 8Mb - limit=2047 (2048*4096=8Mb)
        .word   0x0000          | base address=0
        .word   0x9A00          | code read/exec
        .word   0x00C0          | granularity=4096, 386

        .word   0x07FF          | 8Mb - limit=2047 (2048*4096=8Mb)
        .word   0x0000          | base address=0
        .word   0x9200          | data read/write
        .word   0x00C0          | granularity=4096, 386

idt_48:
        .word   0                       | idt limit=0
        .word   0,0                     | idt base=0L

gdt_48:
        .word   0x800           | gdt limit=2048, 256 GDT entries
        .word   gdt,0x9         | gdt base = 0X9xxxx

msg1:
        .byte 13,10
        .ascii "Loading system ..."
        .byte 13,10,13,10

.text
endtext:
.data
enddata:
.bss
endbss:
```

Testing the Bootloader

Perhaps the easiest way to test a bootloader is inside a virtual machine, like VirtualBox or VMware.[1]

Sometimes it is useful if the bootloader supports the GDB remote debug protocol.[2]

Further Reading

1. ↑ "How to develop your own Boot Loader" (http://www.codeproject.com/KB/tips/boot-loader.aspx?fid=1541607&df=90&mpp=25&noise=3&sort=Position&view=Quick&fr=1#_Toc231383187) by Alex Kolesnyk 2009
2. ↑ "RedBoot Debug and Bootstrap Firmware" (http://www.ecoscentric.com/ecos/redboot.shtml)

- Embedded Systems/Bootloaders and Bootsectors describes bootloaders for a variety of embedded systems. (Most embedded systems do not have a x86 processor).