System Initialization (x86)

From OSDev Wiki

When an x86-based computer is turned on (assuming that it turns on successfully), it begins a complex path to get to the stage where control is transferred to your kernel's "main()" routine. The exact sequence of steps depends on what kind of device the computer decides to boot from, and whether it uses the legacy BIOS boot method, or the new UEFI method. The new UEFI method completely changes the entire boot process, and is covered in another article. The intention is that the UEFI boot process will someday completely replace the BIOS method in all new computers, but it may not succeed.

Described below is the traditional BIOS-based system initialization process.

Contents

BIOS initialization

The very first thing that happens when the machine is turned on, is that the CPU starts trying to run a program at the very end of the 4Gb memory area. At that location must be some ROM which contains a BIOS initialization program. The initialization code can be large, and the ROM may be as large as 256Kb in size. An OS programmer cannot modify or control this stage of the process in any way.

- RAM detection -- the BIOS must use some RAM in order to perform its functions. To use the RAM, the BIOS must first detect the type and quantity of RAM chips installed. This can only be done (using chipset-specific methods) while the CPU is running code that is stored in **ROM**, specifically. Once the RAM has been detected, the BIOS may perform a simple memory test on it, and then the BIOS loads data and code into several memory areas in RAM: the BDA, the EBDA, and the 64K "BIOS area" at physical address 0xF0000 to 0xFFFFF. The BIOS also sets up a tiny stack somewhere in memory, and sets up the Real Mode IVT from physical address 0 to 0x3FF. Some of the physical memory between address 0xA0000 and 0xFFFFF is then set to "read only" mode using chipset-specific methods.

- Hardware detection/Initialization -- the BIOS detects, enumerates, configures, and initializes every bus, and almost every piece of hardware on the system, using values that the BIOS chooses. It stores a great deal of information about all of this hardware, for the OS to later parse. If the BIOS finds any ROM chips on any hardware, they are mapped (not loaded) into physical memory at addresses that the BIOS chooses. It is important to note that many BIOSes do a rather bad job of configuring, sometimes. An OS may well need to reconfigure the MTRRs, or the PCI bus, or the mapping of some PCI devices. The BIOS is supposed to always bring all the hardware into a

functional state, but that state may not be optimal, or even technically "legal" according to the specs.

- "Boot sequence" -- at this point, the BIOS is done with its initialization. Now it tries to transfer control to the next stage of the bootloader process; so the BIOS must choose the "boot device". There is a list stored in CMOS, called the "boot sequence", that tells the BIOS which devices to test, and in what order, to see if they exist and are bootable. The BIOS may try to boot from a floppy disk, hard disk "C:", a USB flash memory device, a CD, a network, or something else. All of these devices can have some type of "bootsector", and there is a flag that the BIOS can check to see if the bootsector is valid. The BIOS will transfer control to the first valid bootsector that it finds, as it searches through the boot sequence. If the BIOS never finds a valid bootsector, it will lock up with an error message.

The BIOS transfers 512 bytes of data from each device that exists, into physical memory starting at address 0x7c00. If the last two bytes transferred are 0x55, and then 0xAA, then the BIOS considers this to be a valid bootsector, and starts running the code that now begins at 0x7c00.

Transfer of Control to Bootsector

In most circumstances (except for CDs/DVDs) the bootsector will be LBA 0 of the device (or equivalently, CHS 0,0,1) -- see the specs of the particular device to find out what addressing mode it uses. For CDs/DVDs, the bootsector is LBA 17. Traditionally, there is one additional complication. Hard disks have been standardized as having "partitioning". That is, all operating systems allow you to subdivide a hard disk into smaller "partitions" or "volumes". The way this is done is by adding an additional "layer" of standardized bootsectors, called MBRs (Master Boot Records).

What this means in reality is that the BIOS boots some devices directly into your OS-specific bootloader, and the BIOS boots other devices (any hard disk, or anything emulating a hard disk) into an MBR -- which in turn boots your OS-specific bootloader. So the system always ends up running your OS-specific bootloader -- but it may happen directly, or there may be an intervening step.

This has implications if you want to write your own bootloader, and/or if you want to write your own MBR.

System "Environment"

There are very few things that are standardized about the state of the system, when the BIOS transfers control to the bootsector. The only things that are (nearly) certain are that the bootsector code is loaded and running at physical address 0x7c00, the CPU is in 16-bit Real Mode, the CPU register called DL contains the "drive number", and that only 512 bytes of the bootsector have been loaded.

Note: there are apparently a tiny number of ancient BIOSes that actually put the system in Protected Mode, instead of Real Mode. The recommendation is: do not support those BIOSes.

OS-specific Bootloaders

Every OS is expected to have its own bootloader, stored on the media with the kernel. The bootloader is expected to be stored at a particular location, so that the BIOS or MBR can find it, load it, and run it. If the boot device is a hard disk (or something else emulating a hard disk) then the bootloader is expected to be stored as the very first "block" of the partition. As said above, for all other types of devices, the bootloader is stored at LBA 0, CHS 0,0,1, or LBA 17 (for CDs and DVDs).

The main function of these bootloaders is to find the kernel, wherever it is on the media, load it, and run it. Additionally, the bootloaders need to set up a known environment for the kernel (which often includes switching to Protected Mode). The bootloaders also might collect some system data for the kernel to use (some data is much easier to get while the sysetm is still in Real Mode).

For more bootloader "theory" see the bootloader article.

There are generic bootloaders available, or you can create your own.

Either way, however, the OS-specific bootloader completes the process of system initialization.

MBR Bootsectors

Bootsectors for devices that can have multiple partitions have a standard format. Such devices always include hard disks, and can include USB Flash drives, or remote Network drives. The MBR bootsector is created on the drive when the drive is partitioned by partitioning software (such as FDISK, under DOS/Windows).

The BIOS loads the entire MBR (both the Partition Table and Bootstrap code), and executes the bootstrap code. (See System "Environment" above for additional details.)

The function of the MBR Bootstrap code is (usually) to search the Partition Table for the partition marked with an "active" flag (flag byte equals 0x80), then load and run the bootsector (that is, the OS-specific Bootloader) of that partition (the very first sector). This is what "generic" MBR Bootstraps do, such as the one that FDISK writes to a newly partitioned disk. The bootstrap code is expected to pass a pointer (in DS:SI) to a memory copy of the booted partition table entry, to the OS-specific bootloader -- as well as reproduce the above System "Environment" that the OS-specific bootloader expects to see.

Fancier MBR bootstraps can perform "Dual Booting". That is, they present a choice of disk drives and disk partitions to the user, and allow the user to select a particular partition to boot -- rather than just automatically choosing the "active partition". These fancy MBR bootstraps often use tricks to overcome that fact that they are limited to a little over 400 bytes of code.

See the MBR (x86) article for information on writing your own MBR.

Notes

See Also