# Booting an Operating System

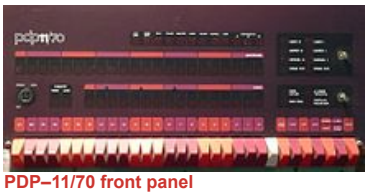
### How do you run that first program?

By Paul Krzyzanowski
*January 26, 2015*

## Introduction

An operating sytem is sometimes described as "the first program," one that allows you to run other programs. However, it is usually stored as a file (or, more commonly, a collection of files) on a disk. How does this "first" program get to run?

The operating system is loaded through a **bootstrapping** process, more succinctly known as **booting**. A **boot loader** is a program whose task is to load a bigger program, such as the operating system.


PDP–11/70 front panel

When you turn on a computer, its memory is usually uninitialized. Hence, there is nothing to run. Early computers would have hardware that would enable the operator to press a button to load a sequence of bytes from punched cards, punched paper tape, or a tape drive. Switches on the computer's front panel would define the source of the data and the target memory address. In some cases, the boot loader software would be hard wired as non-volatile memory (in early computers, this would be a grid of wires with cuts in the appropirate places where a 0-bit was needed).

In early minicomputer and microcomputer systems systems, a computer operator would use the switches on the computer's front panel to toggle in the code to load in a bigger program, programming each memory location and then starting the program. This program might do something basic such as read successive bytes into memory from a paper tape attached to a teletype.

In later systems, read-only memory would contain a small bootloader that would have basic intelligence to read, say, the first sector (512 bytes) of a disk.

Since this initial program had to be as small as possible, it would have minimal capabilities. What often happened is that the boot loader would load another boot loader, called a **second stage loader**, which was more sophisticated. This second stage loader would have error checking, among possibly other features, such as giving the user a choice of operating systems to boot, the ability to load diagnostic software, or enabling diagnostic modes in the operating system. This **multi-stage boot loader**, having a boot loader load a bigger boot loader, is called **chain loading**.

The boot loader will often perform some core initialization of the system hardware and will then load the operating system. Once the operating system is loaded, the boot loader transfers control to it and is no longer needed. The operating system will initialize itself, configure the system hardware (e.g., set up memory management, set timers, set interrupts), and load device drivers, if needed.

## Intel-based (IA–32) startup

To make the example of the boot process concrete, let us take a look at 32-bit Intel-compatible PCs (we'll get to 64-bit systems in a bit). This architecture is known as IA–32 (Intel Architecture, 32-bit) and defines the instruction set of most Intel microprocessors since the Intel 80386 that was introduced in 1986. It is still supported on Intel's latest processors.

An IA–32-based PC is expected to have a BIOS (**Basic Input/Output System**, which comprises the bootloader firmware) in non-volatile memory (ROM in the past and NOR flash memory these days). The BIOS is a descendent of the BIOS found on early CP/M systems in that it contains low-level functions for accessing some basic system devices, such as performing disk I/O, reading from the keyboard, and accessing the video display. It also contains code to load a **stage 1 boot loader**.

When the CPU is reset at startup, the computer starts execution at memory location `0xffff0` (the IA–32 architecture uses a segment:offset form of addressing; the code segment is set to `0xf000` and the instruction pointer is set to `fff0`).

The processor starts up in **real mode**, which gives it access to only a 20-bit memory address space and provides it with direct access to I/O, interrupts, and memory (32-bit addressing and virtual memory comes into play when the processor is switched to *protected mode*). The location at `0xffff0` is actually at the end of the BIOS ROM and contains a jump instruction to a region of the BIOS that contains start-up code.

Upon start-up, the BIOS goes through the following sequence:

1. Power-on self-test (POST)
2. Detect the video card's (chip's) BIOS and execute its code to initialize the video hardware
3. Detect any other device BIOSes and invoke their initialize functions
4. Display the BIOS start-up screen
5. Perform a brief memory test (identify how much memory is in the system)
6. Set memory and drive parameters
7. Configure Plug & Play devices (traditionally PCI bus devices)
8. Assign resources (DMA channels & IRQs)
9. Identify the boot device

When the BIOS identifies the boot device (typically one of several disks that has been tagged as the bootable disk), it reads block 0 from that device into memory location `0x7c00` and jumps there.

### Stage 1: the Master Boot Record

This first disk block, block 0, is called the **Master Boot Record** (**MBR**) and contains the first stage boot loader. Since the standard block size is 512 bytes, the entire boot loader has to fit into this space. The contents of the MBR are:

- First stage boot loader (≤ 440 bytes)
- Disk signature (4 bytes)
- Disk partition table, which identifies distinct regions of the disk (16 bytes per partition × 4 partitions)

### Stage 2: the Volume Boot Record

Once the BIOS transfers control to the start of the MBR that was loaded into memory, the MBR code scans through its partition table and loads the **Volume Boot Record** (**VBR**) for that partition. The VBR is a sequence of consecutive blocks starting at the first disk block of the designated partition. The first block of the VBR identifies the partition type and size and contains an **Initial Program Loader** (**IPL**), which is code that will load the additional blocks that comprise the **second stage boot loader**. On Windows NT-derived systems (e.g., Windows Server 2012, Windows 8), the IPL loads a program called **NTLDR**, which then loads the operating system.

One reason that low level boot loaders have a difficult time with loading a full OS, especially one that may be composed of multiple files, is that doing so requires the ability to parse a file system structure. This means understanding how directories and file names are laid out and how to find the data blocks that correspond to a specific file. Without much code, it is far easier to just read consecutive blocks. A higher-level loader, such as Microsoft's NTLDR, can read NTFS, FAT, and ISO 9660 (CD) file formats.

## Beyond Windows

There are many variations on booting other operating systems on an Intel PC. One popular boot loader on Linux systems is GRUB, or GRand Unified Bootloader. GRUB is also a multistage boot loader. The BIOS, of course, does what it always does: identifies a bootable device, loads the Master Boot Record, and transfers control to this newly-loaded code. Under GRUB, the MBR typically contains a first-stage boot loader called **GRUB Stage 1**. This Stage 1 boot loader loads **GRUB Stage 2**. The Stage 2 loader presents the user with a choice of operating systems to boot and allows the user to specify any additional boot parameters for those systems (e.g., force maximum memory, enable debugging). It then reads in the selected operating system kernel and transfers control to it.

A specific problem with using GRUB to boot Windows is that Windows is not Multiboot compliant. Multiboot is a Free Software Foundation specification on loading multiple operating systems using a single boot loader. What GRUB does in this case is fake a conventional Windows boot process. It boots a bootloader that would normally reside in the MBR (or run the Windows boot menu program). From that point onward, GRUB is out of the picture, Windows has no idea what happened, and the native Windows boot process takes over.

## Good-bye, BIOS. Hello UEFI

As 64-bit architectures emerged to replace 32-bit architectures, the BIOS was starting to look quite dated. Intel set out to create a specification of a BIOS successor that had no restrictions on having to run the startup code in 16-bit mode with 20-bit addressing. This specification is called the **Unified Extensible Firmware Interface**, or **UEFI**. Although developed by Intel, it was managed since 2005 by the Unified EFI Forum. It is used by many newer 64-bit systems, including Macs, which also have legacy BIOS support for running Windows.

Some of the features that EFI supports are:

*BIOS components*

preserved some components from the BIOS, including power management (Advanced Configuration & Power Interface, ACPI) and system management components (e.g., reading and setting date).

*Support for larger disks*

The BIOS only supported four partitions per disk, with a capacity of up to 2.2 TB per partition. UEFI supports a maximum partition size of 9.4 ZB ($9.4 \times 10^{21}$ bytes).

*No need to start up in 16-bit (real) mode*

The pre-boot execution environment gives you direct access to all of system memory.

*Device drivers*

UEFI includes device drivers, including the ability to interpret architecture-independent EFI Byte Code (EBC). Operating systems use their own drivers, however, so — as with the BIOS — the drivers are generally used only for the boot process.

*Boot manager*

This is a significant one. The old BIOS only had the smarts to load a single block, which necessitates multi-stage booting process. UEFI has its own command interpreter and complete boot manager. You no longer need a dedicated boot loader. As long as you place the bootable files into the UEFI boot partition, which is formatted as a FAT file system (the standard file system format in older Windows systems; one that just about every operating system knows how to handle).

*Extensibility*

The firmware is extensible. Extensions to UEFI can be loaded into non-volatile memory.

### Booting with UEFI

With UEFI, there is no longer a need for the Master Boot Record to store a stage 1 boot loader; UEFI has the smarts to parse a file system and load a file on its own, even if that file does not occupy contiguous disk blocks. Instead, UEFI reads the **GUID** (Globally Unique IDentifier) **Partition Table** (**GPT**), which is located in blocks immediately after block 0 (which is where the MBR still sits for legacy reasons). The GPT describes the layout of the partition table on a disk. From this, the EFI boot loader identifies the EFI System Partition. This system partition contains boot loaders for all operating systems that are installed on other partitions on the device. For EFI-aware Windows systems, UEFI loads the *Windows Boot Manager* (bootmgfw.efi). For older 64-bit NT systems, EFI would load *IA64ldr*. For Linux, there are many options. Two common ones are to use an EFI-aware version of GRUB (the Grand Unified Bootloader) and load a file such as *grub.efi* or to have EFI load load *elilo.efi*, the EFI loader.

In general, even with UEFI, the dominant approach is to load an boot loader dedicated to a specific operating system rather than load that operating system directly. However, the need for a multi-stage boot process that requires loading multiple bootloaders is no longer necessary.

## Non-Intel Systems

Our entire discussion thus far has focused on booting with the Intel PC-based architecture (which includes IA–32/IA–64 compatible architectures, such as those by AMD). This is the dominant architecture in today's PCs (notebooks through servers) but there are many, many non-Intel devices out there, particularly in embedded devices, such as cell phones. What about them?

There are numerous implementations of the boot process. Many embedded devices will not load an operating system but have one already stored in non-volatile memory (such as flash or ROM). Those that load an OS, such as ARM-based Android phones, for instance, will execute code in read-only memory (typically in NOR flash memory) when the device is powered on. This boot code is embedded within the CPU ASIC on some devices so you do not need a separate flash memory chip on the board.

When the system is reset (which includes a power-up), the processor is in supervisor (SVC) mode and interrupts are disabled. On ARM-based systems the processor starts

execution at address `0x00000000`. The flash memory containing start-up code is mapped to address `0x00000000` on reset. This code performs various initializations, including setting up an exception vector table in DRAM and copying application code from ROM to DRAM (code runs faster in DRAM). The code remaps the DRAM to address 0, thus hiding the flash memory (the processor has a REMAP bit to change the mapping of flash memory). The memory system is then initialized. This involves setting up memory protection and setting up the system stacks. I/O devices are then initialized and the processor is changed to user mode. The boot firmware detects bootable media and loads and runs the second stage boot loader (if necessary). The second stage boot loader is often GRUB for larger systems or uBoot for embedded systems. This second stage loader loads the operating system and transfers control to it.

## Mac OS X

Older PowerPC-based versions of Apple Macintosh systems, as of at least OS 8 as well as OS X, were based on Open Firmware. Open Firmware originated at Sun and was used in non-Intel Sun computers. Once Apple switched to Intel systems, it adopted UEFI as its boot-level firmware.

### Older Macs

Open Firmware is stored in ROM and, like the PC BIOS, is executed on power-on. Since Open Firmware was designed to be platform independent, it is implemented in Forth (a simple stack-based language) and compiled to bytecodes rather than native machine instructions. The firmware contains a byte code interpreter.

Unlike the BIOS, Open Firmware provides the user with a command-line processor from which one can edit system configuration parameters, such as reduce the amount of physical memory, debug the system, or even start a telnet server so that you can interact with the firmware and boot process from a remote machine via an ethernet connection.

Before booting the operating system, the Open Firmware generates a device tree by probing components on the main board and expansion devices.

Like the PC BIOS, Open Firmware contains device drivers that the boot process in the firmware can use to access the disk, keyboard, monitor, and network. However, these drivers are all implemented in FCode, the Forth bytecode system. Also like the BIOS, these drivers are used only during the boot process. The operating system itself has its own native runtime drivers.

Unlike the BIOS, Open Firmware could parse the HFS/HFS+ file systems (the native file system on Macs), so you could use the Open Firmware command interpreter to load in a boot file from the hard disk and run it. By default, Open Firmware loads a file from the system partition. On OS 9 systems, this was the file called "Mac OS ROM" in the *System* folder. On OS X systems, it loads `/System/Library/CoreServices/BootX`. `BootX` is bootloader that then loads in the kernel.

### The Mac today

The Mac uses UEFI for its system firmware.

When the Mac starts up, the first code that gets executed is the BootROM. This sets up EFI drivers for relevant hardware devices, initializes some of the hardware interfaces, validates that sufficient memory is available, and performs a brief power-on self-test. Unlike the PC BIOS, which knew nothing about file systems and could only read raw disk blocks, UEFI on the Mac has been extended to parse both FAT (legacy DOS/Windows) and HFS+ (native Mac) filesystems on a disk. It reads the GPT (GUID Partition Table) to identify disk partitions. The default boot volume is stored in NVRAM.

Instead of specifying a path to a boot loader, the HFS+ volume header (data at the start of an HFS+ file system) points to a *blessed file* or *blessed directory* (see the bless command. If a directory is blessed, that tells the EFI firmware to look in that directory for the boot loader. If a file is blessed, that tells the EFI firmware to load that file as the boot loader (there are extra variations, such as booting from an unmounted volume).

By default, the boot loader is located in `/System/Library/CoreServices/boot.efi` on the root (often only) partition of the disk.

Alternatively, the firmware supports downloading a second-stage bootloader or a kernel from a network server (netboot server).

When the `boot.efi` file is loaded, the computer displays a metallic Apple logo on the screen. The boot loader loads in the kernel as well as essential driver extensions, which then runs *launchd*, which executes the various startup scripts and programs. Once the kernel is loaded, the spinning gear gear appears below the Apple logo. When the kernel runs the first process, *launchd*, the screen turns blue.

A description of how OS X starts up can be found in What is Mac OS X.

To support booting BIOS-based operating systems, such as older Windows systems and Linux systems that use GRUB or other BIOS-aware boot loaders, the EFI installs a "compatibility support module" (CSM) component from the system firmware. This then starts a BIOS-based boot process. This compatibility support module is loaded only when the user selects Windows as the default OS to boot. The boot process now is a standard BIOS-based boot. The Master Boot Record (MBR) is loaded and executed, which then locates and loads the Volume Boot Record of the Windows (or Linux) partition.

## References


- Booting, Wikipedia
- System Boot Sequence, The PC Guide
- GNU Multiboot spec
- UEFI consortium and specifications
- EFI, Wikipedia article
- What UEFI Can Do For You, Tom's Hardware
- GUID Partition Table, Wikipedia article
- EFI System Partition, Wikipedia article
- Boot Configuration Data in Windows Vista, Microsoft Hardware Developer Central
- Windows Vista startup process, Wikipedia article
- Windows NT startup process, Wikipedia article
- ELILO: EFI Linux Boot Loader
- ARM architecture tutorial

Mac booting references

- Open Firmware Home Page
- Open Firmware, Wikipedia article
- What is Mac OS X, Mac OS X Internals
- Mac-OF-I, Mac OS X Reference Library, Apple Inc.

- Mac-OF-II, Mac OS X Reference Library, Apple Inc.

Mac OS X references:

- Mac-GPT, Mac OS X Reference Library, Apple Inc.
- Mac-Boot, Mac OS X Reference Library, Apple Inc.
- Mac-Intel, HT2674, Apple Inc.
- Mac-Admin, © 2007 Apple Inc.
- Mac-bless
- Mac-Intel-boot, rEFIt Sourceforge Project


This is an updated version of the original document, which was written on September 14, 2010.

---