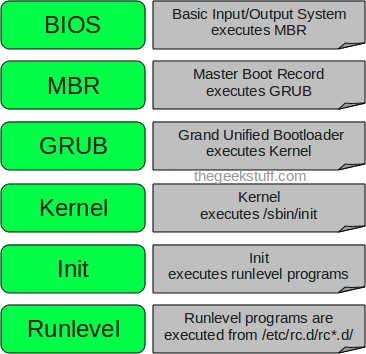
# Linux Boot Process

**Booting** a computer refers to the **process** of powering on the computer and starting the operating system. ... The **boot process** loads the operating system into main memory or the random access memory (RAM) installed on your computer.

Press the power button on your system, and after few moments you see the Linux login prompt.

Have you ever wondered what happens behind the scenes from the time you press the power button until the Linux login prompt appears?

The following are the 6 high level stages of a typical Linux boot process.



### 1. BIOS

* BIOS stands for Basic Input/Output System
* Performs some system integrity checks
* Searches, loads, and executes the boot loader program.
* It looks for boot loader in floppy, cd-rom, or hard drive. You can press a key (typically F12 of F2, but it depends on your system) during the BIOS startup to change the boot sequence.
* Once the boot loader program is detected and loaded into the memory, BIOS gives the control to it.
* So, in simple terms BIOS loads and executes the MBR boot loader.

### 2. MBR

* MBR stands for Master Boot Record.
* It is located in the 1st sector of the bootable disk. Typically /dev/hda, or /dev/sda
* MBR is less than 512 bytes in size. This has three components 1) primary boot loader info in 1st 446 bytes 2) partition table info in next 64 bytes 3) mbr validation check in last 2 bytes.
* It contains information about GRUB (or LILO in old systems).
* So, in simple terms MBR loads and executes the GRUB boot loader.

### 3. GRUB

* GRUB stands for **Grand Unified Bootloader.**
* If you have multiple kernel images installed on your system, you can choose which one to be executed.
* GRUB displays a splash screen, waits for few seconds, if you don’t enter anything, it loads the default kernel image as specified in the grub configuration file.
* GRUB has the knowledge of the filesystem (the older Linux loader LILO didn’t understand filesystem).
* Grub configuration file is /boot/grub/grub.conf (/etc/grub.conf is a link to this). The following is sample grub.conf of CentOS.

#boot=/dev/sda

default=0

timeout=5

splashimage=(hd0,0)/boot/grub/splash.xpm.gz

hiddenmenu

title CentOS (2.6.18-194.el5PAE)

root (hd0,0)

kernel /boot/vmlinuz-2.6.18-194.el5PAE ro root=LABEL=/

initrd /boot/initrd-2.6.18-194.el5PAE.img

* As you notice from the above info, it contains kernel and initrd image.
* So, in simple terms GRUB just loads and executes Kernel and initrd images.

### 4. Kernel

* Mounts the root file system as specified in the “root=” in grub.conf
* Kernel executes the /sbin/init program
* Since init was the 1st program to be executed by Linux Kernel, it has the process id (PID) of 1. Do a ‘ps -ef | grep init’ and check the pid.
* initrd stands for Initial RAM Disk.
* initrd is used by kernel as temporary root file system until kernel is booted and the real root file system is mounted. It also contains necessary drivers compiled inside, which helps it to access the hard drive partitions, and other hardware.

### 5. Init

* Looks at the /etc/inittab file to decide the Linux run level.
* Following are the available run levels
  + 0 – halt
  + 1 – Single user mode
  + 2 – Multiuser, without NFS
  + 3 – Full multiuser mode
  + 4 – unused
  + 5 – X11
  + 6 – reboot
* Init identifies the default initlevel from /etc/inittab and uses that to load all appropriate program.
* Execute ‘grep initdefault /etc/inittab’ on your system to identify the default run level
* If you want to get into trouble, you can set the default run level to 0 or 6. Since you know what 0 and 6 means, probably you might not do that.
* Typically you would set the default run level to either 3 or 5.

### 6. Runlevel programs

* When the Linux system is booting up, you might see various services getting started. For example, it might say “starting sendmail …. OK”. Those are the runlevel programs, executed from the run level directory as defined by your run level.
* Depending on your default init level setting, the system will execute the programs from one of the following directories.
  + Run level 0 – /etc/rc.d/rc0.d/
  + Run level 1 – /etc/rc.d/rc1.d/
  + Run level 2 – /etc/rc.d/rc2.d/
  + Run level 3 – /etc/rc.d/rc3.d/
  + Run level 4 – /etc/rc.d/rc4.d/
  + Run level 5 – /etc/rc.d/rc5.d/
  + Run level 6 – /etc/rc.d/rc6.d/
* Please note that there are also symbolic links available for these directory under /etc directly. So, /etc/rc0.d is linked to /etc/rc.d/rc0.d.
* Under the /etc/rc.d/rc\*.d/ directories, you would see programs that start with S and K.
* Programs starts with S are used during startup. S for startup.
* Programs starts with K are used during shutdown. K for kill.
* There are numbers right next to S and K in the program names. Those are the sequence number in which the programs should be started or killed.
* For example, S12syslog is to start the syslog deamon, which has the sequence number of 12. S80sendmail is to start the sendmail daemon, which has the sequence number of 80. So, syslog program will be started before sendmail.

There you have it. That is what happens during the Linux boot process.

(So this first-stage bootloader is **U**-**Boot SPL**; and second-stage bootloader is regular **U**-**Boot** (or **U**-**Boot** proper). To be clear: **SPL** stands for Secondary Program Loader. ... So usually **boot** sequence is next: ROM code -> **SPL** -> **u**-**boot** -> kernel. And actually it's very similar to PC **boot**, which is: BIOS -> MBR -> GRUB -> kernel)

**ARM BOOT PROCESS:**

Most of the embedded Linux devices have customized boot sequence as per application specs and device security. But the generic sequence is mentioned below

1. On powering up ARM SOC, the ROM boot code which is already flashed by vendor will start executing from internal RAM of ARM SOC. The main purpose of this ROM code is to load second stage boot loader aka SPL(**SPL** stands for Secondary Program Loader) (**MLO**) from a recognizable boot media
2. The SPL simply sets up the boot process to load next stage boot loader called **u-boot**
3. Now U-boot gives many boot options to load kernel into DDR (RAM). It can load **kernel** from network, sd card, eMMC, SPI flash etc. Also it contains a console for user interaction
4. Once kernel is loaded by the U-boot. It will load other system peripheral drivers and mount the Root File System specified by kernel boot argument
5. At this point, the kernel has mounted RFS and it will execute a specific binary called **/sbin/init** which is the ancestor of all other processes
6. The /sbin/init program will read a file called **inittab**, which contains which script to be executed on some actions such as system start or system shutdown
7. On system startup, the **rcS** script will be called which is mentioned in inittab. The rcS script will start all startup scripts available in /etc/init.d/
8. After executing the startup scripts, rcS will call a script called **rc.local** where user defined programs will start executing. Now the boot sequence is completed.

Apart from the above init process called **SysVinit**, there are other init processes called systemd, upstart etc., available. Please go through below link for more info

[The Boot Process](http://processors.wiki.ti.com/index.php/The_Boot_Process)

## What is Bootloader?

In embedded systems the bootloader is a special piece of software whose main purpose is to load the kernel and hand over the control to it. To achieve this, it needs to initialize the required peripherals which helps the device to carry out its intended functionality. In other words, it initializes the absolutely needed peripherals alone and hands over the control to the O.S aka kernel.

## Das U-Boot — the Universal Boot Loader

U-Boot is the most popular boot loader in linux based embedded devices. It is released as open source under the GNU GPLv2 license. It supports a wide range of microprocessors like MIPS, ARM, PPC, Blackfin, AVR32 and x86. It even supports FPGA based nios platforms. If your hardware design is based out of any of these processors and if you are looking for a bootloader the best bet is to try U-Boot first. It also supports different methods of booting which is pretty much needed on fallback situations.

For example, it has support to boot from USB, SD Card, NOR and NAND flash (non volatile memory). It also has the support to boot linux kernel from the network using TFTP. The list of filesystems supported by U-Boot is huge. So you are covered in all aspects that is needed from a bootloader and more so.

## Stages in boot loading

For starters, U-Boot is both a first stage and second stage bootloader. When U-Boot is compiled we get two images, first stage (MLO) and second stage (u-boot.img) images. It is loaded by the system’s ROM code (this code resides inside the SoC’s and it is already preprogrammed) from a supported boot device. The ROM code checks for the various bootable devices that is available. And starts execution from the device which is capable of booting. This can be controlled through jumpers, though some resistor based methods also exists. Since each platform is different and it is advised to look into the platforms datasheet for more details.

Stage 1 bootloader is sometimes called a small SPL (Secondary Program Loader). SPL would do initial hardware configuration and load the rest of U-Boot i.e. second stage loader. Regardless of whether the SPL is used, U-Boot performs both first-stage and second-stage booting.

In first stage, U-Boot initializes the memory controller and SDRAM. This is needed as rest of the execution of the code depends on this. Depending upon the list of devices supported by the platform it initializes the rest. For example, if your platform has the capability to boot through USB and there is no support for network connectivity, then U-Boot can be programmed to do exactly the same.

If you are planning to use linux kernel, then setting up of the memory controller is the only mandatory thing expected by linux kernel. If memory controller is not initialized properly then linux kernel won’t be able to boot.