**ARM Linux Boot up process**

Software components Involved in Embedded Linux Boot Process

* Bootloader(sometimes needed Bootstrap code before Bootloader)
* kernel Image
* root file system - either an initrd image or a NFS location

**Bootloader Phase**(Bootstrap sometimes)

Typically this stage does the hardware initialization and testing, loads the kernel image, and transfers control to the Linux kernel. It does the following initialization

1. Configuring the CPU speed
2. Memory initialization, such as setting up the registers, clearing the memory, and determining the size of the onboard memory
3. Turning on the caches
4. Setting up the serial port for the boot console
5. Doing the hardware diagnostics or the POST (Power On Self-Test diagnostics)
6. Once the above steps are completed successfully, the next step is loading the Linux kernel.

**Downloading Kernel Image and Initial Ram Disk**

The boot loader needs to locate the kernel image, which may be on the system flash or may be on the network. In either case, the image needs to be loaded into memory. In case the image is compressed (which often is the

case), the image needs to be decompressed. Also if an initial ram disk is present, the boot loader needs to load the image of the initial ram disk to the memory. Note that the memory address to where the kernel image is

downloaded is decided by the boot loader by reading the ELF header of the kernel image.

**Setting up Arguments**

Argument passing is a very powerful option supported by the Linux kernel. Linux provides a generic way to pass arguments to the kernel across all platforms. Typically the boot loader has to set up a memory area for argument passing, initialize it with the required data structures (that can be identified by the Linux kernel), and then fill them up with the required values.

**Jumping to Kernel Entry Point**

The kernel entry point is decided by the linker script when building the kernel (which is typically present in linker script in the architecture-specific directory). Once the boot loader jumps to the kernel entry point, its job is done and it is of no use.

**Kernel initialization phase**

This stage does the platform-specific initialization, brings up the kernel subsystems, turns on multitasking, mounts the root file system, and jumps to user space.

**CPU/Platform-Specific Initialization**

Following are the steps to initialize the desired CPU

1. *Setting up the environment for the first C routine:* The kernel entry point is an assembly language routine; the name of this entry point varies (stext on ARM, kernel\_entry on MIPS, etc.). Look at the linker script to know the entry point for your platform. This function normally resides in the arch/<name>/kernel/head.S file. This function does the following:

* On machines that do not have the MMU turned on, this turns on the MMU. Most of the boot loaders do not work with the MMU so the virtual address equals the physical address. However, the kernel is compiled with the virtual address. This stub needs to turn on the MMU so that the kernel can start using the virtual address normally.
* Set up the BSS by zeroing it out (normally you cannot rely on the boot loader to do this).
* Set up the stack so that the first C routine can be invoked. The first C routine is the start\_kernel() function in init/main.c. This function is a jumbo function that does a lot of things until it terminates in an idle task (the first task in the system having a process id of 0). This function invokes the rest of the platform initialization functions that are discussed below.

**User- space initialization phase**

Typically this phase brings up the services,does network initialization, and then issues a log-in prompt.