It is the part that starts the system up and loads the operating system kernel.

the role of the boot-loader and, in particular, how it passes control from itself to

the kernel using a data structure called a **device tree**, also known as a **flattened**

**device tree** or **FDT**.

**What does a boot-loader do?**

In an embedded Linux system, the boot-loader has two main jobs: to initialize the

system to a basic level and to load the kernel. In fact, the first job is somewhat

subsidiary to the second, in that it is only necessary to get as much of the system

working as is needed to load the kernel.

When the first lines of the boot-loader code are executed, following a power-on

or a reset, the system is in a very minimal state.

* The DRAM controller would not have been set up, and so the main memory would not be accessible. Likewise, other interfaces would not have been configured, so storage accessed via NAND flash controllers, MMC controllers, and so on, would also not be usable.
* Typically, the only resources operational at the beginning are a single CPU core and some on-chip static memory. As a result, system bootstrap consists of several phases of code, each bringing more of the system into operation. The final act of the boot-loader is to load the kernel into RAM and create an execution environment for it.
* The details of the interface between the boot-loader and the kernel are architecture-specific, but in each case it has to do two things. First, boot-loader has to pass a pointer to a structure containing information about the hardware configuration, and second it has to pass a pointer to the kernel command line. The kernel command line is a text string that controls the behavior of Linux. Once the kernel has begun executing, the boot-loader is no longer needed and all the memory it was using can be reclaimed.
* the bootloader code running in NOR flash memory can initialize the DRAM controller, so that the main memory, the DRAM, becomes available and then it copies itself into the **DRAM**. Once fully operational, the bootloader can load the kernel from flash memory into **DRAM** and transfer control to it.
* However, once you move away from a simple linearly addressable storage

medium like **NOR flash**, the boot sequence becomes a complex, multi-stage

procedure. The details are very specific to each SoC, but they generally follow

each of the following phases.

**Phase 1 – ROM code**

In the absence of reliable external memory, the code that runs immediately after

a reset or power-on has to be stored on-chip in the SoC; this is known as **ROM**

**code**. It is loaded into the chip when it is manufactured, and hence the ROM

code is proprietary and cannot be replaced by an open source equivalent.

Usually, it does not include code to initialize the memory controller, since

DRAM configurations are highly device-specific, and so it can only use **Static**

**Random Access Memory (SRAM)**, which does not require a memory

controller.

Most embedded SoC designs have a small amount of SRAM on-chip, varying in

size from as little as 4 KB to several hundred KB:

The ROM code is capable of loading a small chunk of code from one of several

pre-programmed locations into the SRAM