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 boot-loader 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 boot-loader 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.

In SoCs where the SRAM is not large enough to load a full boot-loader like U-Boot,

there has to be an intermediate loader called the **secondary program**

**loader**, or **SPL**.

At the end of the ROM code phase, the SPL is present in the SRAM and the

ROM code jumps to the beginning of that code.

**Phase 2 – secondary program loader**

The SPL must set up the memory controller and other essential parts of the

system preparatory to loading the **Tertiary Program Loader** (**TPL**) into

DRAM. The functionality of the SPL is limited by the size of the SRAM.

It can read a program from a list of storage devices, as can the ROM code, once again using pre-programmed offsets from the start of a flash device. If the SPL has file system drivers built in, it can read well known file names, such as **u-boot.img,**

from a disk partition.

The SPL usually doesn't allow for any user interaction, but

it may print version information and progress messages, which you can see on

the console.

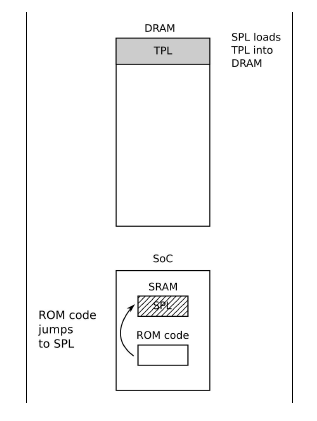
The SPL may be open source, as is the case with the TI x-loader and Atmel

AT91Bootstrap, but it is quite common for it to contain proprietary code that is

supplied by the manufacturer as a binary blob.

At the end of the second phase, the TPL is present in DRAM, and the SPL can

make a jump to that area.

****

**Phase 3 – TPL**

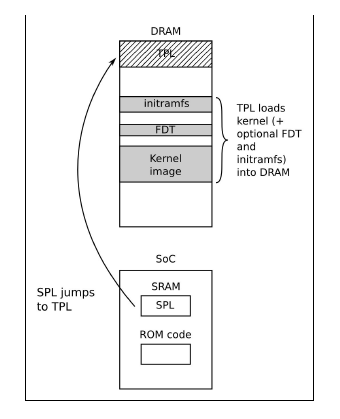
Now, at last, we are running a full boot-loader, such as U-Boot or Bare-Box.

Usually, there is a simple command-line user interface that lets you perform

maintenance tasks, such as loading new boot and kernel images into flash

storage, and loading and booting a kernel, and there is a way to load the kernel

automatically without user intervention.



At the end of the third phase, there is a kernel in memory, waiting to be started.

Embedded bootloaders usually disappear from memory once the kernel is

running, and perform no further part in the operation of the system

**Booting with UEFI firmware**

Most embedded x86/x86\_64 designs, and some ARM designs, have firmware

based on the **Universal Extensible Firmware Interface** (**UEFI**) standard. You

can take a look at the UEFI website at http://www.uefi.org/ for more information. The

boot sequence is fundamentally the same as that described in the preceding

section:

* **Phase 1**: The processor loads the platform initialization firmware from

flash memory. In some designs, it is loaded directly from NOR flash

memory, while in others, there is ROM code on-chip which loads the

firmware from SPI flash memory into some on-chip static RAM.

* **Phase 2**: The platform initialization firmware performs the role of SPL. It

initializes the DRAM controller and other system interfaces, so as to be

able to load an EFI boot manager from the **EFI System Partition** (**ESP**) on

a local disk, or from a network server via PXE boot. The ESP must be

formatted using FAT16 or FAT32 format and it should have the well-known

GUID value **of C12A7328-F81F-11D2-BA4B-00A0C93EC93B.** The path name of the boot

manager code must follow the naming convention

<efi\_system\_partition>/boot/boot<machine\_type\_short\_name>.efi. For example, the

file path to the loader on an x86\_64 system would be /efi/boot/bootx64.efi.

* **Phase 3**: The UEFI boot manager is the tertiary program loader. The TPL in

this case has to be a bootloader that is capable of loading a Linux kernel

and an optional RAM disk into memory. Common choices are:

systemd-boot: This used to be called gummiboot. It is a simple UEFIcompatible

bootloader, licensed under LGPL v2.1. The website is https:/

/www.freedesktop.org/wiki/Software/systemd/systemd-boot/.

Tummiboot: This is the gummiboot with trusted boot support (Intel's **Trusted**

**Execution Technology** (**TEX**)).

**Moving from bootloader to kernel**

When the bootloader passes control to the kernel it has to pass some basic

information, which may include some of the following:

The *machine number*, which is used on PowerPC, and ARM platforms

without support for a device tree, to identify the type of the SoC

Basic details of the hardware detected so far, including at least the size and

location of the physical RAM, and the CPU clock speed

The kernel command line

Optionally, the location and size of a device tree binary

Optionally, the location and size of an initial RAM disk, called the **initial**

**RAM file system** (**initramfs**)

The kernel command line is a plain ASCII string which controls the behavior of

Linux by giving, for example, the name of the device that contains the root

filesystem. I will look at the details of this in the next chapter. It is common to

provide the root filesystem as a RAM disk, in which case it is the responsibility

of the bootloader to load the RAM disk image into memory.

The way this information is passed is dependent on the architecture and has

changed in recent years. For instance, with PowerPC, the bootloader simply used

to pass a pointer to a board information structure, whereas, with ARM, it passed

a pointer to a list of *A tags*.

In both cases, the amount of information passed was very limited, leaving the

bulk of it to be discovered at runtime or hard-coded into the kernel as **platform**

**data**. The widespread use of platform data meant that each board had to have a

kernel configured and modified for that platform. A better way was needed, and

that way is the device tree.

In the ARM world, the move away from A tags began

in earnest in February 2013 with the release of Linux 3.8. Today, almost all

ARM systems use device tree to gather information about the specifics of the

hardware platform, allowing a single kernel binary to run on a wide range of those platforms.