

Dietmar Schütz  
Siemens AG, CT SE 2

Otto-Hahn-Ring 6  
81739 München  
Germany

eMail: dietmar.schuetz@siemens.com  
Phone: +49 (89) 636-57380

## ***BOOT LOADER***

**Also known as** Bootstrap Loader

### **Summary**

---

---

The *BOOT LOADER* pattern describes the mechanisms that are necessary to start a computer, from being switched on, up to full operability. In order to run-up into a defined state of operation, with the operating system initialized and started, a sequence of single bootstrap steps is performed, each gaining a higher level of operability. This technique also supports flexibility in different dimensions, e.g. selecting a software version, a boot device, or even updating the whole software.

---

---

### **Context**

General Purpose Computers (such as PCs, workstations, mainframes), Embedded Systems.

### **Example**

Consider a personal computer. Each time it is turned on, it should run up into a defined condition, thereby starting the operating system, and maybe a default application. In addition to this standard behaviour, it sometimes is necessary to perform a different start-up procedure. Typical scenarios are updating the system (install a new release of the operating system or the application software), or adding new hardware and software components. Things get even more complicated if a component that essentially contributes to the start-up is malfunctioning and needs to be replaced, such as a hard disk drive that contains the operating system.

### **Problem**

How can you prepare a computer to perform the same steps and return to the same initial state each time it is turned on, and also enable changing this start-up behaviour easily?

The following **forces** influence the solution:

- A processor that has just been switched on has almost no capabilities.
- The executable memory (i.e. the memory sections that hold the program code and can be directly accessed by the processor for execution) is mostly built from type RAM, which is volatile, hence

---

Copyright granted to Hillside Europe for use with/at the EuroPLoP 06 conference

its content is lost and at least undefined after the power has been switched off.

- Persistent *and* executable memory that keeps its content during power off (e.g. EEPROM, or Flash<sup>1</sup>) has a lot of drawbacks. It is more expensive than RAM, limited capacity due to packing density, often insufficient performance because of low read rates. Last but not least, most persistent memory types cannot be written like RAM, which is mandatory for some programming models, and for debugging.
- The initial software and the resulting start state must not be carved in stone, for example to update the operating system, or install another one. Changing the software must be possible with minimal hardware effort.
- The start-up process should support a wide range of media for (local resources such as Floppy Disk, Hard Disk, CD-ROM, Flash Card, USB-Stick), and also remote resources accessible by means of some communication network.
- The start-up state should be selectable from a number of different choices, e.g. in order to support a “dual boot” feature, that allows to start a PC either with Windows™ or Linux.
- It is necessary to support different hardware configurations, given by different combinations, and variants for the components.
- Support of programmable hardware (such as FPGAs), that loses its “program” and hence its functionality when power is turned of.

### **Solution**

Use the minimal initialization routine of the processor to load a more capable routine from a predefined location within persistent memory into executable memory. With this routine, called the “bootstrap loader”, load the desired executable into executable memory, using other input media. If necessary, iterate over this procedure at higher levels to further increase the capabilities to the desired extent.

### **Structure**

Although only passive, two elements are important in the context of a *BOOT LOADER*, both providing storage locations with different characteristics: *Persistent memory* is required to store the image across periods without power supply. The destination of the loading process must be part of the *executable memory*; it is directly connected to the processor on the address and data bus, which is necessary to fetch instructions for execution.

---

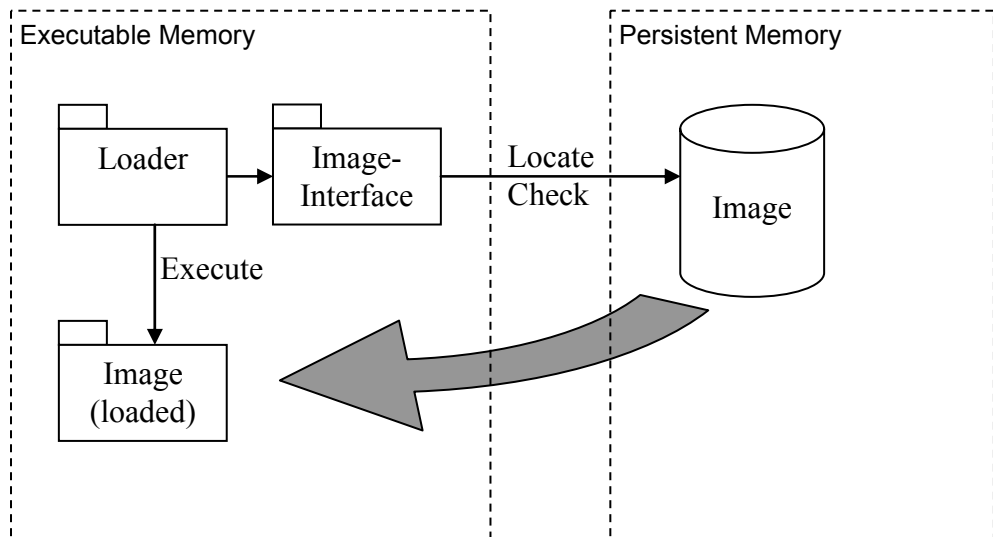
<sup>1</sup> To be precise: NOR-Flash, but not NAND-Flash

Each boot loader stage is defined by three participants that interact in the bootstrap procedure:

The *image* carries all the functionality that should be available after the bootstrap procedure, as well as all the steps that are necessary to activate it. Typically, the image can be exchanged somehow from outside the system.

The *loader* is a program that has the responsibility to verify, load, and activate an image into executable memory. To this end, it must be able to address and read from the persistent memory that hosts the image.

A *convention* defines the interface that ties loader and image together. This convention specifies where to find the image, how to validate it, and possibly additional semantics that affect the loading procedure. The loader accesses the image according to this convention, and the image must adhere to that.



## Dynamics

The scenario below depicts the activities during a single bootstrap phase. The system is in a well defined start state<sup>2</sup>: the program counter of the processor references the first instruction of the loader, starting the execution there.

The loader initializes the persistent memory for reading; this may also include activating peripheral hardware components, such as a hard-disk drive. Afterwards, the processor reads those parts of the image that are necessary to check the validity of the image. If ok, the execution memory is prepared to receive the image. With all this setup-up done, the whole image is copied chunk by chunk into executable memory, until it is complete and

---

<sup>2</sup> For the first boot loader stage, this requires some functionality hard-wired into the processor. Typically this incorporates setting critical registers into a safe state, disabling any interrupts, and setting the program counter to zero.



be arise the need to change this part (or the whole remaining parts) of the boot process. Size constraints or other external restrictions may apply too.

- ☛ The boot process of a x86-based PC is split into three stages.
  1. The Basic Input Output System (BIOS), stored in EEPROM at the processor start address. The BIOS is capable to access different disk drives. It checks the devices in a given order, until it discovers a valid image (a so called Master Boot record, MBR), or reaches the end of list without success, hence displaying an error on the screen.
  2. The MBR contains a partition table for the disk drive it is located on, and code to interpret the other data blocks on the disk as a file system. Within that file system, it can use a given name to look up the operating system file, and load it.
  3. The operating system file contains all parts of the operating system, and the code that is necessary to initialize and start it ☞

2 *Decide the location where the image resides.* The location is an abstract term, with many possible concrete refinements. Examples are:

- A memory address that indicates the start position of the image
- A hardware address to access a device such as a serial input line, or a disc drive (in raw mode: simple sequential access, limited size, a byte block/sequence without external structure)
- A path and filename to retrieve the image from a file system
- A look-up mechanism and communication protocol to retrieve the image from a remote computer across a network.

All functionality necessary to address the “location” must be available upfront. In consequence, resolving paths and names in file system structure like NTFS, or using a XYZ network address, is seldom applicable for the first boot loader stage.

- ☛ In the x86-arcitectures, the MBR is read (by the BIOS) from the first sector of a disk drive: head zero, track zero, sector one, 512 bytes. ☞

3 Define the *format* of the image. In most cases, an identical “one-by-one” copy of the structure aimed at in memory is an appropriate choice. If the image has a variable size, the length must be incorporated into the image too.

Other options are using an encoding scheme (in order to enable image transfer across text-oriented serial lines), or a compressed image that is inflated whilst loading (reduced image size, better performance when reading from slow media, e.g. flash).

- ☛ In the x86-arcitectures, the BIOS as well as the MBR are directly executable after copying.

*FAST BOOT* uses a compressed image of the whole RAM content, which is decompressed whilst loading. ☞

- 4 *Image Validation.* Decide if it is necessary to validate the loadable image prior to further processing. This is typically a good idea if the image resides on a changeable media, such as a floppy disk or a compact flash card. Even in the case of a fixed media built into the system, e.g. a serial EEPROM, security requirements can suggest a sophisticated validity check on the image.

There are two common approaches for a validity check:

- A *COOKIE*, (a special pattern of byte values within the image, expected at a specific offset) is sufficient to prevent unintended mistaking a random data for a boot image.
- A checksum, computed across the whole image, is viable to ensure the integrity of the image.

• MBRs compliant to the x86 bootloader are validated using a cookie with values 0x55 and 0xAA in the last two bytes. The most simple (hence useless) structure that is recognized as a “valid” MBR is defined by the following assembler code:

```

;*****
[BITS 16]
ORG      0
INT      0x18          ; generate a "Machine check" interrupt

TIMES    510-($-$$) DB 0 ; fill the remaining block with 0
DW      0xAA55

;*****

```

- 5 *Select target area.* The target area is part of the executable memory. It is defined by the memory address to where the image should be loaded. The destination is typically a contiguous region, but it also can consist of multiple chunks at different locations.

Inside the target area is also the start address, where the processor will continue execution after the image has been loaded successfully. In most cases, the start address is transparent to the image, but in some cases the image might also need to know where it has been loaded, e.g. in order to do some linking.

- 6 *Implement the loader.* The loader incorporates:

- Initialize access to the image, in accordance with the defined image location, e.g. by opening the communication link
- Run the validity check on image to approve its compliance and correctness.
- Prepare the destination area to receive the image. This may for example incorporate loading new parameters into the memory mapping tables.
- Start the copy routine, transferring the image from the persistent location to its destination in executable memory. During copying,

the image might be processed “on the fly”, for example to decompress or de-crypt it.

- Change processing mode, e.g. switch from 16bit real mode to 16bit protected-mode, or to 32bit-protected mode.
- Jump to the execution start address, thereby passing control to the loaded image.

If necessary, leave a backdoor in the boot loader, which may prove to be useful for update scenarios.

During the whole boot-loading process, it is essential not to damage the previous stage. If this happens, the system loses its ability to start over, and a software update gets necessary, to be performed by a lower (hopefully accessible and still consistent) stage.

### Example Resolved

After power-on, the processor is put into a reset state (all internal registers hold the initial value, program counter is zero), and starts executing the code at address zero. This part of the address space is associated with a non-volatile memory (flash in modern computers, EPROM or even ROM in the past) that holds the code of the BIOS (basic input output system). The BIOS initialises the hardware.

Then it checks whether the disk drive contains a valid image, by comparing the last word of the master boot record with the cookie value 0x55aa. If proven valid, the image is loaded and started. The image usually contains a second stage boot loader, which is capable to interact with the file system on the hard disk, hence locates and eventually loads the operating system.

### Consequences

The Boot Loader pattern provides the **benefits** depicted below:

- *Manageable Complexity.* The procedure of starting a computer is divided into manageable peaces, each providing a significant but not too complex step towards full functionality.
- *System Scalability.* The procedure of boot loading opens up optimization potential regarding the system-wide memory and execution structure. It enables the combination of cheap executable memory (e.g. RAM) with cheap non-volatile memory (such as flash), thereby lowering the overall hardware costs without limiting performance.
- *Support for future requirements.* The ability to replace the image (and thereby the functionality of the next stage) provides a strong variation point for upcoming (and even unanticipated) requirements.
- *Supports variability.* By offering a small number of choices to continue operation, every stage can introduce variation. For example, a second stage boot loader can offer different operating systems that can be started on the same computer.

On the other hand, the pattern carries the following **liabilities**:

- *Hardware-Dependent*. At least the lower stages of a boot loader chain are significantly interrelated with the hardware (the processor as well as its surrounding bus structures and other circuits connected to it).
- *Low-Level-Code*. The code of the first boot-loader stages contains a lot of hardware related stuff. Some parts must be written in Assembler language. And even those that can be written in C are operating on hardware registers, and are obfuscate in comparison to usual application code. The programmer must know about memory models, addressing modes, timing constraints, caching implementations, and whatever else lives deep down there.
- *Difficult to organize, understand*. Designing boot loader stages requires a lot of system know how. Especially in the context of evolving hardware, many upcoming extensions must be anticipated and addressed in advance.

## Variants

*Boot Manager*. The loader provides different options to choose from. Default selection after time out.

*De-compressor*. The image is not necessarily an identical copy of the content that should reside in executable memory after the bootstrap. In order to save some space in the persistent memory, the boot loader can inflate a previously compressed whilst loading. This is technique especially useful in the *FAST-BOOT* pattern, where in some constellations reading a smaller compressed image and decompressing it on the fly is faster than reading the corresponding full size image (see [Radermacher+2002] for details on appropriate compression algorithms).

*Link-Loader*. If some routines of the code loaded in previous stages should be used by the upcoming stage, it is necessary to refine abstract addresses within the image to concrete addresses. The similar problem is usually solved when statically linking different objects and libraries into one executable. Although this linking could be done upfront, resulting in an image that must fit to loader, resolving such reference at loading time reduces coupling, and allows replacing the loader by a different one without being forced to adapt the image accordingly.

*FPGA-Initialization*. In some kinds of systems, the processor itself is not “carved out of silicon”, but provides programmable functionality. Some types of these devices even hold their “program” in volatile SRAM, and therefore must load the gate configuration from some external memory, in order to create a processor. Consequently, the configuration must contain the initialization routine for the processor.

*Multi-Processor.* Use on processor to initialize the other. If this is done vice versa, it's a wonderful scenario for updates, since there is always one stable running instance.

*Hibernate support.* Before running the normal boot procedure, the boot loader determines if the last shutdown has generated a hibernate image. If so, instead of loading and starting operating system, the previous application state is restored by loading back the hibernate image into RAM.

*Anything else.* The boot loader is nothing more than a piece of software that runs in a dedicated environment. Hence, it can serve other purposes than loading the next stage image. The most useful scenario here is probably running some diagnostic software, or special functionality that is required during production or maintenance.

### Known Uses

Whenever there is a computer, there is a boot loader too. Some prominent examples are:

- Every PC
- Windows Boot Manager
- Grub LiLo for Linux
- The Mars probe “Surveyor” was launched with (from the software point of view) nothing more than a boot loader.
- AVR ISP routine, Butterfly boot loader

### See Also

A boot loader should also cover include a *POWER ON SELF TEST* to check the correct behaviour of essential system parts.

Boot Loader is essential to run a reliable *SOFTWARE UPDATE*.

Boot Loader may also support power saving modes for the computer, e.g. *HIBERNATE*, *STAND-BY*.

*FAST BOOT* and *ULTRA FAST BOOT* provide techniques for starting a computer extremely fast.

### Credits

This work is nothing more than a new perspective on knowledge that has been made public before. I like to thank many nameless contributors to Wikipedia, the free encyclopedia [Wikipedia], and especially Matthew Vea, who authored the X86 boot loader tutorial [X86].

Shaping the content into accessible structure has been supported by the pattern community. Most of all, I gratefully thank my shepherd Markus Völter, for offering his time, expertise, and help. He gave me an external view, additional insight, and even more useful comments.

### References

[X86]

Matthew Vea: BootStrap Tutorial,  
<http://www.geocities.com/vea/bootstrap.htm>

[Wikipedia]

Articles on Booting, Bootstrapping, and related links

<http://wikipedia.org>

[POSA96]

F. Buschmann, R. Meunier, H. Rohnert, M. Stal, P. Sommerlad:  
Pattern Oriented Software Architecture, A System of Patterns;  
Wiley 1997

[Radermacher+2002]

Ansgar Radermacher, Jürgen Schmitt: Fast Decompression,  
published in “EuroPLOP ’02 Proceedings of the 7th European  
Conference on Pattern Languages of Programs, 2002” pp 633ff,  
UVK Universitätsverlag Konstanz GmbH 2003