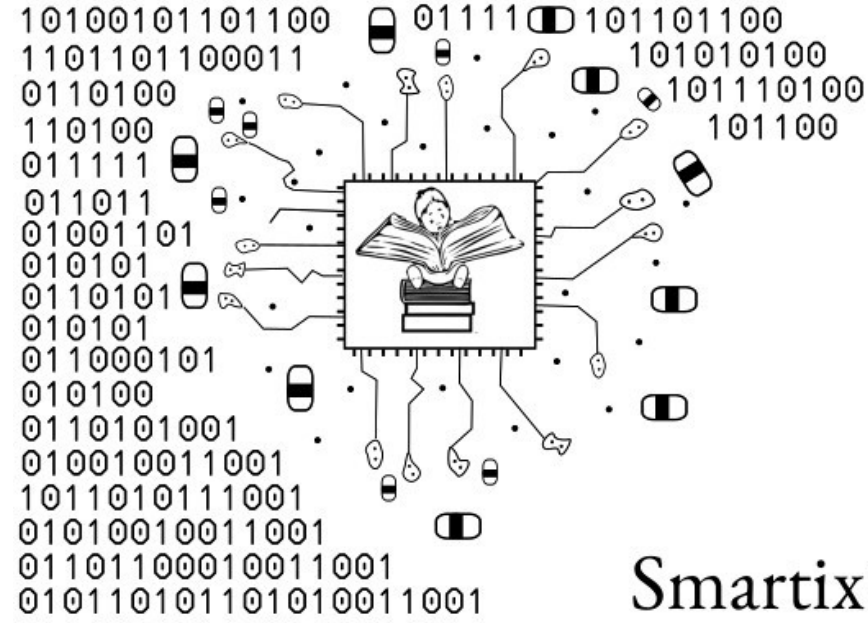


SmartixOS and FreeBSD

Extending the bootloader with a theming system

Document Produced by: The SmartixOS Project
Revision Date: Fri Mar 17 08:12:53 UTC 2023
Art Work by: Ghislain Pissingadaga
BSD User Groups Represented: West Africa Stallions, East Africa Lions

Contributors

Ghisain Pissingadaga, Felix Ndayikengurukiye

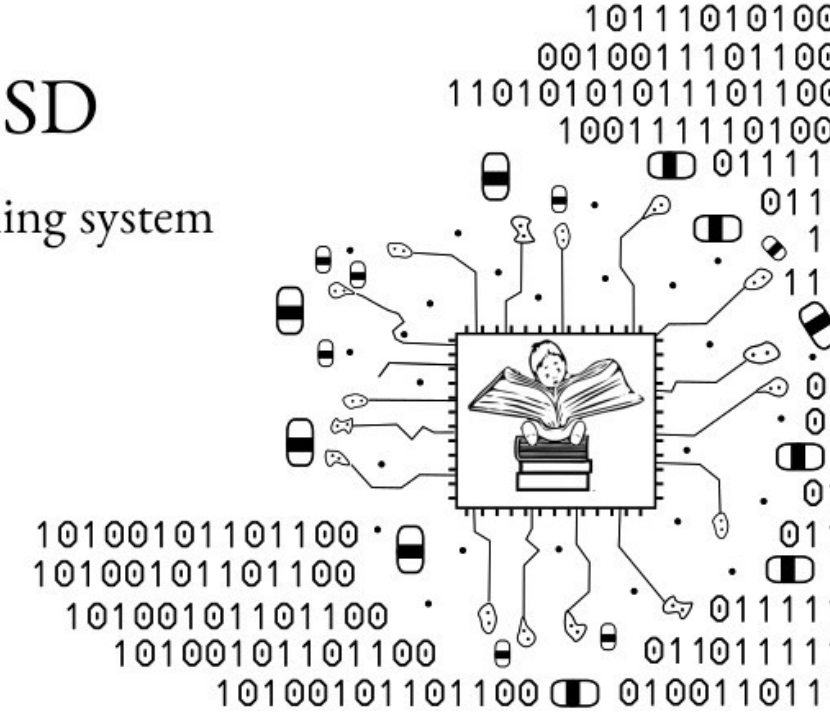


Table of content

Why should you care about the boot process and the boot display

One of the main reasons why we got interested in the boot procedure and more specifically the bootloader menu display is that we needed an easy to display brands/logos in such a way to represent diverse environments. The current bootloader although powerful and flexible, some aspects of its display require manual intervention which make manipulating the display a little difficult in some contexts.

This mechanical work over time, can become very tedious depending on the complexity of the menu to display. So to make things a bit easier the bootloader library was extended to add new features aimed at facilitating displaying colorful menus.

Quick Detailed Overview of the boot procedure

It is very important to have a good understanding of how any system starts and the different procedures involved so a quick review of the boot procedure with technical details is given as the necessary foundation.We may certainly already be aware that the FreeBSD Operating system boot procedure is broken down into three important stages. From the moment you power on or reboot your machine, the BIOS goes into action and checks your hardware for hardware faults and problems before anything can happen. When that completes, the BIOS loads the first sector from your hard drive assuming that you have a simple configuration with one hard-drive. This first sector is known as the master boot record (MBR) and under FreeBSD it is a program called boot0. This assembly language code is executed in the first stage and its instructions know just enough to load the next stage. This first sector has a fixed location and a fixed size of 512 bytes. The second program called boot1, is also fixed to 512 bytes and contains few rudimentary instructions to read disk information and load boot2. boot1 and boot2 can logically be thought of as one giant stage that was broken down into two steps due to limitations incurred in the use of the MBR specification. Once boot2 is loaded, it in turn has the ability to boot the system directly or load the loader program; the program that this presentation is mostly concerned about. This interpreter is quite sophisticated and designed to allow more control over how exactly the system boots.

The MBR or Stage 0

Stage 0 or boot0: is the first piece of code loaded from the BIOS and that small program resides on the first sector of your disk. That partition is formally known as the Master Boot Record (MBR) and it is the place where the partition table is also kept. Please, it is good to note that several details were left out so we can concentrate on standard installations and not the most exotic ones. That being said, one important detail that needs mentioning is the fact that other partitioning schemes like GPT may be used and any modern computer can be set to boot from different medias including CDROM and floppies but for the sake of simplicity, this paper focuses on the most basic use case.

MBR continued

After the MBR gets successfully loaded into memory, boot0 goes searching for the partition table normally contained in that sector and reads it. After the successful location of the next instruction to execute which is boot1, some bookkeeping work at the assembly language level is done and the address of the next instruction is loaded into memory and that is where boot1 takes over. These steps are concretely reflected on the command line through visual clues and messages that help the administrator situate himself/herself.

At this point, if there are more than one distinct OS installed, the user is briefly given a set choices to select from before the system can move to the next step. If no selection is made within a small time period, boot0 will load the next boot block into memory and transfer control to it. Different Operating Systems boot code can be installed and loaded from this stage. If and when that it is the case, hitting say the "**escape**" key on the keyboard will interrupt the booting procedure and drop you into an environment that looks like the one below:

Boot0 Prompt

`boot0`

```
F1    FreeBSD
F2    BSD
F5    Disk 2
```

Stage 1: From boot0 to boot1

Stage 1 : Similarly to boot1, boot1 is a very simple program. Furthermore because both boot0 and boot1 are constrained to 512 bytes, the same limitations appear in both contexts. Boot1 is also located at a specific location and the whole procedure relies on it being there for things to go smoothly; the good news is that all of this is hard coded so the administrator seldom needs to worry about this part of the system. However, it is always helpful to have a good understanding of the plumbing under the hood.

As we recall, the restrictions imposed by the MBR specification constrain boot1's ability to executing few rudimentary instructions. If things go well, control is passed to the next block in the chain. Once boot1 successfully finishes, it goes searching for boot2.

Stage 2: From boot1 to boot2

Once boot2 is located and loaded, it needs its environment to be setup in a certain manner so extra instructions are included to allow the program to set up boot2's environment. Boot2 is a BTX client (Boot Extender) and is a little more sophisticated than the previous boot0 and boot1 programs. As a consequence, the block containing boot2 consumes more disk space but allows more sophisticated code to be stored. One advantage is that the problems incurred previously by the size limitation are lifted at this stage. Also, due to the substantial size difference, boot2 had to be stored in a different boot block, big enough to accommodate the code that makes it. However, even though boot1 loads and transfers control to boot2, we usually think of these programs has one giant stage 2.

Stage 2 : So far we've loaded two small blocks and one large program into memory, transferred control twice, both times resetting the environment (stack, segment registers etc..), performing some limited I/O and using the BIOS in the process. We haven't loaded the operating system itself yet but at this point the low level circuitry marks the hardware as not having faults and ready to boot. In this state what we have so far is a machine with low level buses, interrupt requests and other peripherals ready to be controlled by the Operating System and its drivers.

Boot2 Prompt

Stopping the boot procedure at the right time and paying close attention to your screen, reveals a screen like the one below

`boot2` [1]

```
>>FreeBSD/x86 BOOT
Default: 0:ad(0p4)/boot/loader
boot:
```

Boot2 continued

Pressing enter in this context will simply instruct boot2 to load the default loader. You can of course pick the option that best fits your needs and proceed with the procedure.

As mentioned earlier, boot2 is a BTX (Boot Extender) client and one of the advantages that boot extenders provide is a basic virtual memory addressing environment. A quick history and further description of what that stands for is indispensable to put things in perspective.

BTX clients: A quick History and Description

Until now, we have not discussed memory addressing schemes because it is a bit complex and because it may not necessarily be relevant or relate to your daily work. However this paper strives to be thorough by giving enough background to avoid certain nuances so let’s switch our attention to the historical context around BTX clients.

BTX clients: A quick History and Description

Approximately around the release of the **8088** computer through the release of the **80186** computer, Intel processors had only one way to address memory called real mode. These early CPUs had 16 bits registers and 20 bits memory addresses. One problem that appeared due to this ABI [1] architecture was the problem of storing a 20 bits address in a 16 bits register. The answer to this concern was to use two 16 bits registers, with one register serving as a base and the other as an offset (position) to this base. The base register is then shifted left 4 bits after which the two registers are combined to form a 20 bits address that can be calculated to produce a meaning output.

1 Architecture Binary Interface. A term different from API (Application Programming Interface). The difference is where the application is applied sort of say. ABIs apply to low level specifications so think machine and assembly language while APIs apply to applications at nearly all levels.

BTX clients: A quick History and Description

With all these nifty tricks, the early Intel processors could address a total of 1 megabyte. Today this would not even be large enough to store a simple document like this one. Faced with growing consumer demands due to users and programs demanding and consuming more memory, the 1 megabyte address space limit quickly reached its limit.

So to solved the problem, a new addressing scheme called protected mode was devised. The new protected mode allowed for addressing of up to 4 G of memory.

Advantages of BTX clients

Another big advantage of this new scheme was that it was much easier to implement for assembly language programmers. The main difference is in what the new extended registers could contain. Indeed, these registers were allowed to contain full 32 bits address in a protected fashion. Programs were not allowed to write or read them. These segment registers are now used to locate real addresses in memory and this process includes checking bits for permission (read, write etc..) and involves the MMU (memory management unit).

From the BTX client to Stage 3 – The Bootloader

With this history covered we can revert back to the main topic of the demonstration which pertains to BTX clients and bootloaders. Yet another advantage in leveraging the BTX program is that it gives flexibility to system developers. the BTX software provides enough services so that small programs with nice interface can be written. It also allows for greater flexibility in loading the kernel. On FreeBSD systems, the bootloader is loaded from the BTX client or boot2 or stage 2. Counting stage 1 and 2 as a giant stage 2, the bootloader program can be thought of as stage 3. A snapshot that gives a better illustration is provided below.

From the BTX client to the Bootloader

loader

```
BTX loader 1.00 BTX version is 1.02
Consoles: internal video/keyboard
BIOS drive C: is disk0
BIOS 639kB/2096064kB available memory

FreeBSD/x86 bootstrap loader, Revision 1.1
Console internal video/keyboard
(root@releng1.nyi.freebsd.org, Fri Apr  9 04:04:45 UTC 2021)
Loading /boot/defaults/loader.conf
/boot/kernel/kernel text=0xed9008 data=0x117d28+0x176650 syms=[0x8+0x137988+0x8+0x1515f8]
```

The Bootloader or Stage 3

The final boot step before the operating system is loaded consists of the bootloader or the loader for short. The loader program is a combination of standard commands (referred to as "built in commands") and a Forth interpreter (which is based on ficl). The loader allows the user to interact with the system while it boots (one can choose to boot into multi-user or single-user mode etc…). It also allows for diverse system maintenance and troubleshooting tasks. From the loader, the end user can choose to load or unload different kernels and/or kernel modules. The user can also set and/or unset specific boot and sysctl variables; the root device can be redefined and more. These settings can also be controlled in the configuration file "**/boot/loader.conf**" and several others stored on the root partition. However, the default file that the loader reads is located in "**/boot/defaults/loader.conf** ". This default file contains many of the system variables already mentioned and more.

Capabilities of the interpreter

Today the FreeBSD system has implemented the loader using two programming languages sort of say. The legacy loader called 4th is written in FICL and has a specific dictionary of commands that it understands. This loader is quite sophisticated but the language used to write its command; in order words, the language used to write the dictionary of commands it understands is quite difficult to grasp let alone utilize.

The next generation bootloader interpreter is built on top of the legacy loader that uses FICL but adds more features and flexibility.This flexibility comes from the language use to write the loader; Lua. Lua is a programming language because it can be complied but it is also a scripting language because it can be interpreted directly like any other interpreted language like Bash. Lua is very powerful and in fact very easy to learn.

For programmers that have a C/C++ background Lua is easy to learn because of the high degree of similarity in the syntax and the functionalities offered by the standard functions library.The power of Lua also comes from the fact that the interpreter is written in C. In addition, the Lua library allows you to interact with the C library directly if and when necessary.

The loader, Termcap and Ncurses

The next generation bootloader is controlled by a specific directory that contains few important scripts that make this magic happen. These scripts achieve all this work by leveraging Ncurses along with your terminal capabilities.

Terminal Capabilities: A quick Overview

Termcap or Terminal capabilities is a technical term used to describe what your terminal is capable of doing. The origins of this term can be traced back to the early days of modern computing; yes the era of teletypes. Terminal capabilities back then meant exactly that; what control your terminal gave you through your keyboard and its control keys sometimes referred to as Terminal Escape Sequences.

The terminal capabilities specification also covers other aspects like the screen/monitor and the features that its supports. Features like the ability to print colors and manipulate the cursor are also important areas covered by this specification.

Terminal Capabilities and Escape sequences

Terminal escape sequences are sequences of characters that when put in a specific order or when combined in a specific way acquire a meaning in the eyes of the terminal which induces them to act in the requested way. These sequences are the basis for the Ncurses library.

What exactly is Ncurses

The Ncurses library is a library written in C and also implemented in other programming languages that provides an API to manipulate terminal capabilities. The scripts responsible for the bootloader display make heavy use of these capabilities.

Extension Entry point and the main directories involved

Going from a developer point of view to a power user or system administrator point of view to a regular user point of view, the main entry point is the /src/stand directory. This is where all the development happens before it is even build and installed on a machine. This directory contains source code for both versions of the bootloader and for different architectures (e.g x86, i386 etc...). Few important directories within the "stand" directory in the source code tree are the “**common”**, “**ficl”**, “**forth”**, “**liblua”**, “**lua”** and “**man”** directories.

- the common dir: Contains code relevant to all interpreters
- the ficl dir : Contains code pertaining to the FICL language and its dictionary
- the forth dir: Contains code relevant to the 4th interpreter
- the liblua dir : Contains libraries and APIs used by the scripts in the lua directory
- the lua dir : Contains code relevant to the Lua interpreter
- the man dir: Contains relevant documentation

Important scripts controlling the display and their main function

- cli.lua(8) - FreeBSD Lua CLI module
- color.lua(8) - FreeBSD color module
- config.lua(8) - FreeBSD config module
- core.lua(8) - FreeBSD core module
- drawer.lua(8) - FreeBSD menu/screen drawer module
- hook.lua(8) - FreeBSD hook module
- menu.lua(8) - FreeBSD dynamic menu boot module
- password.lua(8) - FreeBSD password module
- screen.lua(8) - FreeBSD screen manipulation module

Environment and Bootloader variables controlling the procedure

- loader_logo : controls the actual logo being displayed
- loader_logo_x : controls the horizontal positioning or x-axis of the logo being displayed
- loader_logo_y : controls the vertical positioning or y-axis of the logo being displayed
- loader_brand : controls the actual brand being displayed
- loader_brand_x : controls the horizontal positioning or x-axis of the brand being displayed
- loader_brand_y : controls the horizontal positioning or y-axis of the brand being displayed
- loader_menu_frame : controls the style of the frame or the rectangular box around the boot options
- loader_menu_title : controls the title of the menu
- loader_color : turns color on or off

The theming system described

The new theming system introduces mainly three bootloader variables to control the colorization of the different sections of the bootloader display. Each section can be manipulated independently of the other. The Brand, the menu and the logo can all be stylized to match or to follow a specific color code that has a meaning. And there seems to be a relationship between the meaning assigned to the color pattern and the four color theorem in mathematics that asserts that no more than four colors are required to color the regions of any map so that no two adjacent regions have the same color.

The new theming system bootloader variables:

- loader_menu_theme : controls the colorization ability of the menu

- loader_brand_theme : controls the colorization ability of the brand

- loader_logo_theme : controls the colorization ability of the logo

The new theming system: Taking it to the next level

On top of these important variables, the scripts responsible for the display have been extended and certain parts rewritten to accommodate this theming system. The library now makes it very easy to add and remove logos, brands etc... This is how we are able to produce colorful displays easily.

Work is in progress to theme the terminal as well after the operating system has booted. The same concept applies but within the operating system there is more flexibility so interesting ideas are being tested. This last part of the system introduces few variables like system_terminal_theme.

Benefits and Applications

- ✓ Good utilization of your terminal capabilities. This presentation started with the boot procedure and different capabilities offered under BSD. One benefit of this system is that it tries to make good utilization of your terminal capabilities and as such, you can think of this as a small upgrade.
- ✓ Flexibility in the color code system. These features were added with the purpose to help the administrator visually separate, represent and identify environments, localities, regions, system functions (QA, test, Production environment) etc... The real test is done by the user so any constructive feedback is welcome.
- ✓ Added benefits for vendors: Vendors like us appreciate foundational work that can be extended to fit a specific purpose easily.
- ✓ The FreeBSD source code tree is a laboratory that has all the tools necessary for various experiments. This is as of today an experiment and the concept so far has helped increase productivity by cutting down on the time normally used to figured out different environments etc... The visual clues provided by the theming system tell us from start to end what environment we are in, what machine we are using and the like. With all this foundational work in place everything related to manipulating the bootloader display seems to be much easier to do.
- ✓ Environments separation, representation and identification through a flexible color code system (theming system)
- ✓ Easy production of distributions that are based on FreeBSD through an easy way to quickly customize the display.

Demonstration

About US

We are a user group of FreeBSD and its derivative mainly located in West and East Africa where we are currently using FreeBSD as a work horse for everything IT.

In addition we are a strong community of Linux users and as such we do cool things with the penguin as well.

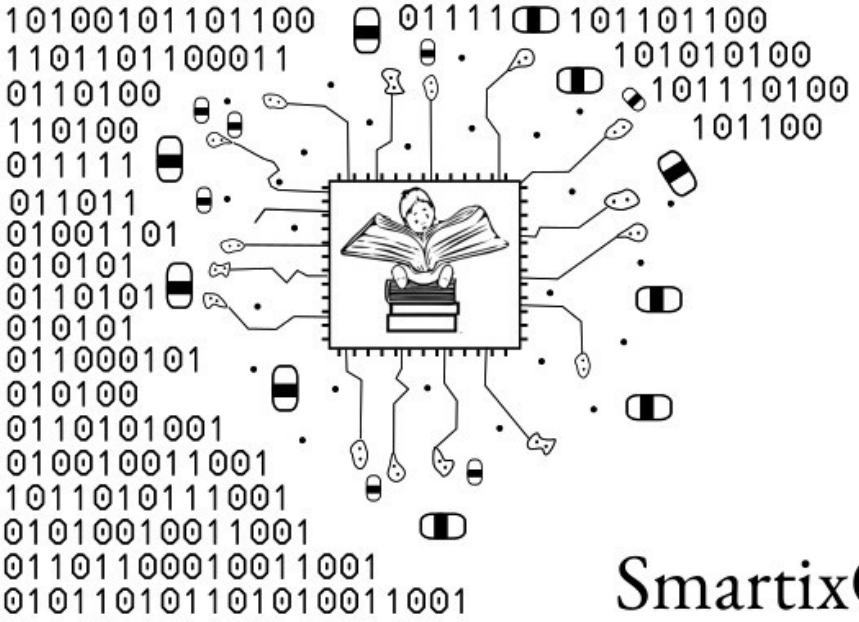
Contact | Contribution | Feedback information

Github: https://github.com/SMARTIX-LLC

Thank you for your interest !

References

- https://docs.freebsd.org/en/books/handbook/boot/
- https://www.lua.org/docs.html
- https://invisible-island.net/xterm/ctlseqs/ctlseqs.html
- https://en.wikipedia.org/wiki/ANSI_escape_code
- https://invisible-island.net/ncurses/ncurses-intro.html
- https://invisible-island.net/ncurses/howto/NCURSES-Programming-HOWTO.html
- https://www.khmere.com/freebsd_book/html/ch02.html
- https://en.wikipedia.org/wiki/Four_color_theorem



SmartixOS and FreeBSD

Extending the bootloader with a theming system

Document Produced by: The SmartixOS Project
Revision Date: Fri Mar 17 08:12:53 UTC 2023
Art Work by: Ghislain Pissingadaga
BSD User Groups Represented: West Africa Stallions, East Africa Lions

Contributors

Ghisain Pissingadaga, Felix Ndayikengurukiye