The

BOOTBOOT

Protocol

Specification and Manual

First Edition

2017

Copyright

Table of Contents

Preface

“A beginning is a very delicate time.”

/ Frank Herbert /

In the last decade of personal computers era big changes happened in the way how computers boot. With the appearance of 64 bit, for the first time in computer’s history, the memory address space became bigger than the storage capacity alltogether. This yielded fundamental changes in firmware.

Also storage capacity kept growing if not according to Moore’s Law, but in a very fast curve. Old ways of storing partitioned data became obsolete, and new partitioning tables were inveted, one of which became the new de facto standard.

Unfortunatelly the firmware that introduced the new partiting format is way to complex and bloated, and therefore many manufacturers refuse to implement it (specially on small hardware with limited resources). As a result, there is no de facto standard for a booting interface, different hardware use different, incompatible ways of booting. Not all firmwares implemented that new partiting table either. To make things worse, many of them also kept backward compatibility with ancient machines.

There are attempts to make booting unified, but unfortunately in a so complex and bloated way again, that one could easily call that loader an OS of it’s own right.

Therefore I’ve created a specification for a common way of starting an operating system, and I’ve provided several different reference implementations one for each platform. The goal is, by the time those small platform dependent code’s execution finished, there’s an universal 64 bit environment on all platforms, capable of running an unmodified C code compiled with the same linker script. The source and the pre-compiled binaries (along with an example C kernel) can be downloaded at:

`https://github.com/bztsrc/bootboot`

Those reference implementations are Open Source and Free Software, and come without any warranty in the hope that they will be useful.

Baldaszti Zoltán Tamás

Page left blank intentionally

Introduction

When you turn on a computer, an operating system has to be loaded. There are sophisticated programs to allow you to choose from multiple systems on a single machine such as GRUB. Those are called boot managers. BOOTBOOT is not one of them. It is a boot loader, with the goal of provinding the same 64 bit environment on several different platforms (to store the bytes "BOOTBOOT" in memory requires 64 bits. The second version of the protocol for 128 bits will be called BOOTBOOT²). If you want to have multiple boot options on one computer, you must install a boot manager that has an option for a BOOTBOOT compliant loader in order to boot a BOOTBOOT compatible operating system. If you are fine with having one operating system per machine, there's no need for a boot manager, the boot loader alone enough.

The operating system can be loaded in many different ways. From ROM, from flash, from a disk, from SD card, over the network etc. The BOOTBOOT Protocol does not specify these. Neither does it specify the archive image's format of the ramdisk used. These are subject to change from time to time and from system to system.

Instead the protocol specifies an Application Programming Interface for locating a file inside an image, and a fallback option when such API is not implemented in a configuration.

The protocol mandates though that if the operating system is stored on disk, that disk must follow the GUID Partitioning Table format (or any later de facto standard partitioning format). As not all firmware support partitioning equally, it is the loader's responsibility to hide this and locate the operating system on a partitioned disk. Therefore end users do not have to care about firmware differences when they want to boot from a disk partitioned and formatted on another machine.

A final word on operating system's kernel format itself. As of writing, there is no de facto standard, but two most widely used formats: the Executable and Linkable Format, and the Portable Executable format. It would be unfair to say one is better than the other, since they both represent the same information just in a different way. Therefore both are supported by the protocol. If one of them (or a new format) became the standard, the protocol has to be revised, and should focus on that format alone, so that the end users don't have to care about executable format either.

Page left blank intentionally

Specification

The first part of this documentation contains the BOOTBOOT Protocol specification.

Booting an Operating System

The term booting a computer refers to many things, but at the end of the day it means only one: pass the control to the operating system's kernel along with environmental information.

On Operating System Kernel Designs

There are two common kinds of kernels. First one contains everything in a single, mostly statically linked image (monolithic kernel). The second kind separated into several files. That keeps the privileged duties to a small kernel (micro-kernel, exokernel, hypervisor etc.) and everything else is pushed into separated user space tasks which usually are stored in separate files.

Both kinds have initial ramdisks, to store files in memory during boot prior to any on disk file system available. For monolithic design that image is usually loaded along with the kernel and (as drivers are included in the kernel), optional. On the other hand for micro-kernels such an archive image for ramdisk is essential.

The creator of BOOTBOOT Protocol and the vast majority of OS developers consider the micro-kernel design more secure and flexible (and also most monolithic design already has it's own way of booting for each and every platform), so the BOOTBOOT Protocol is for micro-kernels.

The Initial Ramdisk Image

As the protcol focuses on micro-kernel design which needs several other files (drivers and such), it requires that the operating system has an initial ramdisk image. And as the image has to be loaded anyway, it's benefitial to store the kernel itself inside. This is not common as of writing, but simplifies booting procedure by reducing the number of required files to one just as with the monolithic design.

Compression on ramdisk image is optional. Reference implementations support gzip deflate compressed images, but other implementation may use different algorithms as long as the compression can be detected with magic bytes. A BOOTBOOT compliant loader will uncompress the image once loaded into memory. As the whole image is loaded entirely, it should be kept small (few megabytes).

The uncompressed format of the ramdisk image is not part of the protcol. Each and every operating system are free choose what's best for it's purpose. Therefore BOOTBOOT Protocol only specifies an Application Programming Interface to parse the image for a file, and a fallback option.

The Boot Partition

The protocol does not describe the whereabouts of the (optionally compressed) initial ramdisk image. It only excepts that a BOOTBOOT Protocol compliant loader can locate and load it into RAM. The reference implementations use ROM and disks with boot partition.

BOOTBOOT Protocol assumes that the disk partitioning format is the de facto standard GUID Partitioning Table. The reason for this is inter-operability among different operating systems.

A boot partition is a small (few megabytes) partition at the beginning of the disk. It may store files relevant to the firmware, but most importantly for the protocol, the initial ramdisk image.

If the boot partition has a file system, for compability reasons it has to be FAT16 or FAT32 formatted. Many firmware (such as UEFI and the Raspberry Pi) mandates that too for their firmware partition. If the boot partition holds firmware files for booting, it should have the type of "EFI System Partition" or ESP in short. This is so because GPT was introduced with the EFI firmware (superseded by UEFI). In this set-up the initial ramdisk image is a file on the boot partition, located in:

BOOTBOOT\INITRD

or with multiple architecture support on the same partition (only for live OS images):

BOOTBOOT\(arch)

like

BOOTBOOT\X86_64
BOOTBOOT\AARCH64

If the firmware's partition does not use a file system, or does not understand FAT16 or FAT32 or has a specific type, so that ESP type cannot be used, then firmware partition and boot partition became two separate partitions.

In that case an operating system designer has two option: either creating another FAT partition with a BOOTBOOT directory and the initial ramdisk file in it; or putting the ramdisk image on the whole partition, leaving the FAT file system entirely out. In either case, the boot partition has to be marked as *EFI_PART_USED_BY_OS* (bit 2 in GPT Partition Entry's attribute flag set).

Keep in mind that ramdisk image will be loaded entirely in memory, so it should be small.

File System Drivers

As mentioned before, the BOOTBOOT Protocol does not specify the initial ramdisk format, instead it uses so called file system drivers with one API function:

```
typedef struct {
    uint8_t *ptr;
    uint64_t size;
} file_t;
file_t myfs_initrd(uint8_t *initrd, char *filename);
```

In the reference implementations' source those file system drivers are separated in a file called **fs.h** (or **fs.inc**). Each supported ramdisk image format has exactly one function in that file.

Each function receives the address of the initial ramdisk image, and a pointer to a zero terminated ASCII filename. If the file referenced by filename found, the function should return a struct with a pointer to the first byte of the file content and the content's size. If needed, the file system driver allowed to allocate memory. On error (when the format not recognized or the file not found) the function must return {NULL, 0}. The protocol expects that a BOOTBOOT compliant loader iterates on the list of drivers until one returns a valid result.

The reference implementations support the following archive and file system image formats:

- *ustar*
- *cpio* (hpodc, newc and crc variants)
- *FS/Z* (OS/Z's native file system)
- *SFS*

Other archive and file system format support can be added any time according the needs of the operating system. The BOOTBOOT Protocol has one exception though. The FAT file system is not allowed as initial ramdisk format, but it's very unlikely someone want to use it that way as FAT is not efficient for an in memory file system.

If all the file system drivers failed and returned {NULL,0}, a fallback driver will be initiated. That fallback driver will scan the ramdisk for the first file which has an executable format for the architecture. So file permissions and attributes does not matter, only the file header does.

If the ramdisk format is supported by one of the file system drivers, the name of the kernel can be passed in the environment with the key *kernel*.

Kernel Format

The kernel executable should be either an Executable and Linkable Format (ELF), or a Portable Executable (PE). In both cases the format itself must be 64 bit (*ELFCLASS64* in ELF and *PE_OPT_MAGIC_PE32PLUS* in PE).

The code segment must be compiled for a native 64 bit architecture. The reference implementations support (*EM_X86_64 (62)* or *EM_AARCH64 (183)* in ELF, and *IMAGE_FILE_MACHINE_AMD64 (0x8664)* or *IMAGE_FILE_MACHINE_ARM64 (0xAA64)* in PE). The x86_64 architecture is used by the BIOS / Multiboot and UEFI loaders, while AArch64 is supported on the Raspberry Pi 3.

Protocol Levels

Now how and where the kernel is mapped depends on the loader's protocol level. The reference implementations only implement level 1, *PROTOCOL_STATIC*. The level 2, *PROTOCOL_DYNAMIC* is for future implementations.

Static

A loader that implements protocol level 1, maps the kernel and the other parts of BOOTBOOT Protocol at static locations. In the specification hereafter, the static protocol will be explained. For compatibility, all BOOTBOOT compatible kernels must provide symbols required by level 2, only right now with those fixed addresses.

Dynamic

A dynamic loader, implementing level 2 on the other hand generates memory mapping according what's specified in the kernel. It only differs from level 1 that the addresses are flexible (but still limited to the negative address range or with different terminology higher half address space).

- Kernel will be mapped to the address pointed by the executable header's *Elf64_Ehdr.p_vaddr* or *pe_hdr.code_base* field.
- The bootboot structure will be mapped at the address of *bootboot* symbol.
- The environment string will be mapped at the address of *environment* symbol.
- The linear frame buffer will be mapped at the address of *fb* symbol.

Entry Point

When BOOTBOOT compliant loader finished with booting, it will hand over the control to the kernel at the address specified in *Elf64_Ehdr.e_entry* or *pe_hdr.entry_point*.

Environment

If the boot partition has a FAT file system, the environment configuration is loaded from

BOOTBOOT\CONFIG

If the initial ramdisk occupies the whole boot partition, then file system drivers are used to locate

sys/config

If the latter is not appropriate for the operating system, the name of the file can be altered in bootboot source. The size of the environment is limited to the size of one page frame (4096 bytes).

Configuration is passed to your kernel as newline ('\n' or 0xA) separated, zero terminated UTF-8 string with "key=value" pairs. C style single line and multi line comments are allowed. BOOTBOOT Protocol only specifies two of the keys, *screen* and *kernel*, all the others and their values are up to the operating system's kernel (or device drivers) to parse. Example:

```
// BOOTBOOT Options

/* --- Loader specific --- */
// requested screen dimension. If not given, autodetected
screen=800x600
// elf or pe binary to load inside initrd
kernel=sys/core

/* --- Kernel specific, you're choosing --- */
anythingyouwant=somevalue
otherstuff=enabled
somestuff=100
someaddress=0xA0000
```

The *screen* parameter defaults to the display's natural size or 1024x768 if that cannot be detected. The minimum value is 800x600.

The *kernel* parameter defaults to sys/core as the kernel executable's filename inside the initial ramdisk image. If that does not fit for an operating system, it can be specified in the environment or can be modified in bootboot source.

Temporary variables will be appended at the end (from UEFI command line). If multiple instance exists of a key, the later takes preference over the former.

To modify the environment, one will need to insert the disk into another machine (or boot a simple OS like DOS) and edit **BOOTBOOT\CONFIG** with a text editor on the boot partition. With UEFI, you

can use the *edit* command provided by the EFI Shell or append "key=value" pairs on the command line (values specified on command line takes precedence over the one in the file).

The environment is mapped before the kernel image in memory, at address specified by the linker. In kernel, it can accessed with

```
extern unsigned char *environment;
```

The `bootboot` Structure

The bootboot struct is specified in bootboot.h, available at

`https://github.com/bztsrc/bootboot/blob/master/bootboot.h`

It is the main information structure passed to the kernel by the loader. It is written with define guard and extern “C”, so it can be safely used from a C++ kernel too.

The structure consist of a fixed 128 bytes header, and a variable sized memory map, each entry 16 bytes long. The first 64 bytes of the header common across platforms, the second 64 bytes hold platform specific pointers.

Header Fields

Platform Independent

```
uint8_t    magic[4];     // 0x00-0x03
```

The magic bytes BOOTBOOT_MAGIC, “BOOT”.

```
uint32_t   size;         // 0x04-0x07
```

The size of the bootboot struct. That is 128 bytes at least, plus the memory descriptors.

```
uint8_t    protocol;     // 0x08
```

The bits 0 – 1 encodes BOOTBOOT Protocol level, implemented by the loader which constructed the bootboot struct. Either *PROTOCOL_STATIC (1)* or *PROTOCOL_DYNAMIC (2)*. Value 3 reserved for future use. *PROTOCOL_MINIMAL (0)* can be used for embedded systems where environment is not implemented and all values and addresses are hardcoded or the frame buffer does not exists at all.

If bit 7 (the sign bit) is set, then the structure has big-endian values, *PROTOCOL_BIGENDIAN (0x80)*.

```
uint8_t    loader_type; // 0x09
```

This is nothing just an information field, either *LOADER_BIOS (0)*, *LOADER_UEFI (1)* or *LOADER_RPI (2)* for now.

```
uint8_t    pagesize;    // 0x0A
```

The size of the page frame used to map the kernel in power of two. Page size in bytes $2^{\text{bootboot.pagesize}}$.

```
uint8_t    fb_type;     // 0x0B
```

The frame buffer format, *FB_ARGB (0)* to *FB_BGRA (3)*. The most common is FB_ARGB, where the least significant byte is blue, and the most significant one is alpha (not used on fb) in little-endian order.

```
int16_t    timezone;    // 0x0C-0x0D
```

The machine's detected timezone if such a thing supported on the platform. This is in minutes from *-1440* to *1440*, and does not affect the value in datetime (which is always in UTC).

```
uint16_t   bspid;       // 0x0E-0x0F
```

The ID of the BootStrap Processor on platforms that support multiple cores (Local APIC ID).

```
uint8_t    datetime[8]; // 0x10-0x17
```

The UTC date of boot in binary coded decimal on platforms that support RTC chip. The first two bytes in hexadecimal gives the year, for example 0x2017, then one byte the month 0x12, one byte day 0x01. Followed by hours 0x23, minutes 0x59 and second 0x00 bytes. The last byte can store 1/100th second precision, but in lack of support on most platforms 0x00. Not influenced by the *timezone* field.

```
uint64_t   initrd_ptr;  // 0x18-0x1F
uint64_t   initrd_size; // 0x20-0x27
```

The address and size of the initial ramdisk in memory.

```
uint8_t   *fb_ptr;      // 0x28-0x2F
uint32_t   fb_size;     // 0x30-0x33
```

Frame buffer physical address and size in bytes. Do not confuse with linker specified *fb* virtual address.

```
uint32_t   fb_width;    // 0x33-0x37
uint32_t   fb_height;   // 0x38-0x3B
uint32_t   fb_scanline; // 0x3C-0x3F
```

The frame buffer resolution and bytes per line as stored in memory.

Platform Dependent Pointers

Only used on x86_64 architecture:

```
uint64_t x86_64.acpi_ptr;     // 0x40-0x7F
uint64_t x86_64.smbi_ptr;
uint64_t x86_64.efi_ptr;
uint64_t x86_64.mp_ptr;
uint64_t x86_64.unused0;
uint64_t x86_64.unused1;
uint64_t x86_64.unused2;
uint64_t x86_64.unused3;
```

Only on AArch64. The *mmio_ptr* field maps the BCM2837 MMIO area at kernel space:

```
uint64_t aarch64.acpi_ptr;    // 0x40-0x7F
uint64_t aarch64.mmio_ptr;
uint64_t aarch64.unused0;
uint64_t aarch64.unused1;
uint64_t aarch64.unused2;
uint64_t aarch64.unused3;
uint64_t aarch64.unused4;
uint64_t aarch64.unused5;
```

Memory Map Entries

```
MMapEnt    mmap;              // 0x80-0xFFF
```

The number of memory map entries can be calculated with

num_mmap_entries = (bootboot.size – 128) / 16;

The memory entry information can be extracted with the following C macros:

MMapEnt_Ptr(a) = the pointer of the memory area
MMapEnt_Size(a) = the size of the memory area in bytes
MMapEnt_Type(a) = the type of the memory area in range of 0 - 15

The type returns one of *MMAP_USED (0)*, *MMAP_FREE (1)*, *MMAP_ACPIFREE (2)* usable after OS parsed ACPI tables, *MMAP_ACPINVS (3)* non-volatile system RAM, *MMAP_MMIO(4)*. Any other value is considered to be *MMAP_USED*. For convinience, another macro is defined

MMapEnt_IsFree(a) = returns true if the memory area can be used by the OS.

The bootboot struct is mapped before the kernel image in memory, at address specified by the linker. In kernel, it can accessed with

```
extern BOOTBOOT bootboot;
```

Linear Frame Buffer

The frame buffer is initialized in 32 bit ARGB mode. It's resolution will be the display's native resolution or 1024x768 if that can't be detected. The requested screen resolution can be passed in environment with the *screen=WIDTHxHEIGHT* paramter.

The frame buffer is mapped along with other MMIO areas before the kernel in memory, at address specified by the linker. In kernel, it can be accessed with

```
extern uint8_t fb;
uint32_t *pixel = (uint32_t*)(&fb + offset);
```

Screen coordinates (X, Y) should be converted to offset as:

*offset = (bootboot.fb_height – Y) * bootboot.fb_scanline + 4 * X*.

Level 1 loaders limit the frame buffer's size somewhere around 4096 x 4096 pixels (depends on scanline and aspect ratio too). That's more than enough for Ultra HD 4K (3840 x 2160) resolution. Level 2 loaders will map the frame buffer where the kernel's *fb* symbol tell to, therefore they don't have such limitation.

Machine State

When the kernel gains control, hardware interrups are masked and code is running in supervisor mode. Depending on platform, failsafe exception handlers may be installed, but the operating system kernel must install it's own habdlers as soon as possible. The MMU is turned on, and memory layout goes as follows:

The RAM (up to 16G) is identity mapped in the positive address range (user space memory). Negative addresses belong to the kernel, and should not be accessible from unprivileged mode.

The uncompressed **initial ramdisk** is enitrely in the identity mapped area, and can be located using bootboot struct's *initrd_ptr* and *initrd_size* members.

The screen is properly set up with a 32 bit **linear frame buffer**, mapped at the negative address defined by the *fb* symbol at **-64M** or **0xFFFFFFFF_FC000000**, along with the other **MMIO** areas pointed by the *mmio_ptr* field. The physical address of the frame buffer can be found in the *fb_ptr* field.

The main information **bootboot structure** is mapped at *bootboot* symbol, at **-2M** or **0xFFFFFFFF_FFE00000**. Consist of a fixed 128 bytes long header followed by variable length data.

The **environment** configuration string (or command line if you like) is mapped at *environment* symbol, at **-2M + 1 page** or **0xFFFFFFFF_FFE01000**.

Kernel's combined **code and data segment** is mapped at **-2M + 2 pages** or **0xFFFFFFFF_FFE02000**. After that segment, at a linker defined address, comes the **bss data segment**, zerod out by the loader. Level 1 protocol limits the kernel's size in 2M, including code, data, bss and stack. That should be more than enough for any micro-kernel.

The **kernel stack** is at the top of the memory, starting at **zero** and growing downwards. The **first page** is mapped by the loader, other pages have to be mapped by the kernel itself when stack grows bigger.

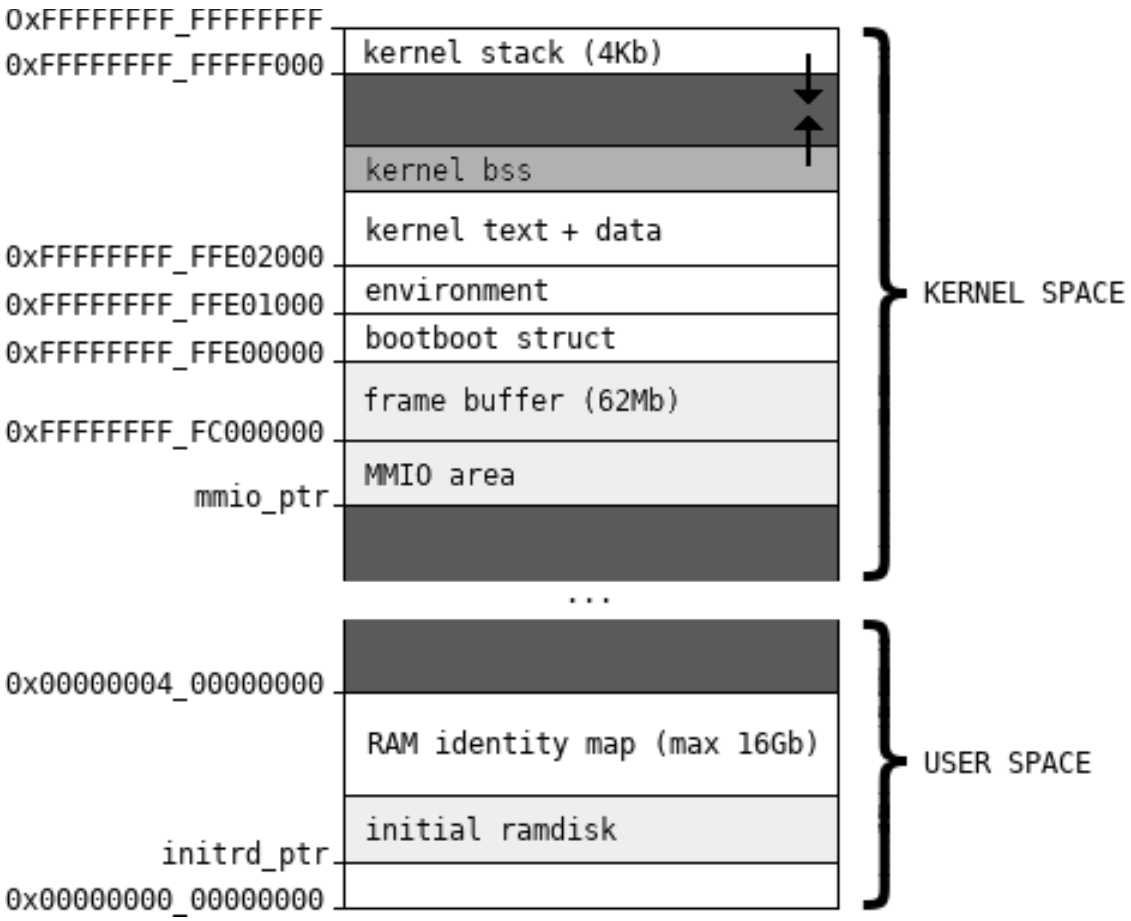

***Figure**: memory layout on kernel hand over (not to scale). Dark gray areas are not mapped.*

Reference Implementations

The second part of this documentation describes the reference implementations and serves as a user manual for them.

All implementations are freely available for download at

`https://github.com/bztsrc/bootboot`

- **x86_64-bios**: IBM PC BIOS / Multiboot implementation
- **x86_64-uefi**: IBM PC UEFI implementation
- **aarch64-rpi**: Raspberry Pi 3 implementation
- **mykernel**: a sample BOOTBOOT compatible kernel

***Figure**: The sample kernel's screen for reference*

IBM PC BIOS / Multiboot

On BIOS (http://www.scs.stanford.edu/05au-cs240c/lab/specsbbs101.pdf) based systems, the same image can be chainloaded from MBR or VBR (GPT hybrid booting), run from a BIOS Expansion ROM or loaded via Multiboot (https://www.gnu.org/software/grub/manual/multiboot/multiboot.html).

Initial Ramdisk

Supported as BIOS Expansion ROM (up to ~96k). Not much, but can be compressed. From disk the initial ramdisk is loaded with the BIOS INT 13h / AH=42h function.

Memory Map

The memory map is queried with BIOS INT 15h / AX=0E820h function.

Linear Frame Buffer

Frame buffer set up are done with VESA 2.0 VBE, INT 10h / AH=4Fh functions.

Machine State

The A20 gate is enabled, serial debug console COM1 is initialized with INT 14h / AX=0401h function to 115200,8N1. Boot date and time are queried with INT 1Ah. IRQs masked. GDT unspecified, but valid, IDT unset. Code is running in supervisor mode in ring 0.

Limitations

- As it boots in protected mode, it only maps the first 4G of RAM.
- The CMOS nvram does not store timezone, so always GMT+0 returned in bootboot.timezone.

Booting

- **BIOS disk**: copy *bootboot.bin* to ***FS0:\BOOTBOOT\LOADER***. You can also place it totally outside of any partition (with `dd conv=notrunc seek=x`). Also install *boot.bin* in the Master Boot Record (or in Volume Boot Record if you have a boot manager), saving bootboot.bin's first sector in a dword at 0x1B0. The *mkboot* (https://github.com/bztsrc/bootboot/blob/master/x86_64-bios/mkboot.c) utility will do that.
- **BIOS ROM**: install *bootboot.bin* in a ***BIOS Expansion ROM***.
- **GRUB**: specify *bootboot.bin* as a Multiboot "kernel" in grub.cfg, or you can chainload *boot.bin*.

IBM PC UEFI

On UEFI machines (http://www.uefi.org/), the PCI Option ROM is created from the standard EFI OS loader application.

Initial Ramdisk

Supported in ROM (up to 16M) as a PCI Option ROM. It is located with EFI_PCI_OPTION_ROM_TABLE protocol and direct probing for magic bytes.

From disk the initial ramdisk is loaded with the EFI_SIMPLE_FILE_SYSTEM_PROTOCOL and BLOCK_IO_PROTOCOL.

Memory Map

The memory map is queried with EFI_GET_MEMORY_MAP boot time service.

Linear Frame Buffer

Frame buffer is set up with the EFI_GRAPHICS_OUTPUT_PROTOCOL.

Machine State

Debug console is implemented with SIMPLE_TEXT_OUTPUT_INTERFACE which can be redirected to serial. Boot date and time are queried with EFI_GET_TIME. IRQs masked. GDT unspecified, but valid, IDT unset. Code is running in supervisor mode in **ring 0**.

Limitations

- The Option ROM should be signed in order to work.

Booting

- **UEFI disk**: copy *bootboot.efi* to ***FS0:\EFI\BOOT\BOOTX64.EFI***.
- **UEFI ROM**: use *bootboot.rom* which is a ***PCI Option ROM*** image of bootboot.efi.
- **GRUB, UEFI Boot Manager**: add *bootboot.efi* to boot options.

Raspberry Pi 3

On Raspberry Pi 3 (https://www.raspberrypi.org/) board the bootboot.img is loaded from the boot partition on SD card as kernel8.img by start.elf.

Initial Ramdisk

No ROM support. Ramdisk loaded by an EMMC SDHC driver implemented in bootboot.c.

Gzip compression is not recommended as reading from SD card is faster than uncompressing.

Memory Map

The memory map is handcrafted with information obtained from VideoCore MailBox's properties channel.

In addition to standard mappings, the **BCM2837 MMIO** is also mapped in kernel space before the frame buffer at **-96M** or **0xFFFFFFFF_FA000000** (other platforms may have map it elsewhere). The correct address can be acquired from *bootboot.aarch64.mmio_ptr* field of the information structure.

Linear Frame Buffer

Frame buffer is set up with VideoCore MailBox messages.

Machine State

Serial debug console is implemented on AUX mini-UART (UART1), with 115200,8N1 and USB debug cable connected to a PC. As the Raspberry Pi does not have an on-board RTC chip, always 0000-00-00 00:00:00 returned in *bootboot.datetime*. Code is running in supervisor mode, at **EL1**. Dummy exception handlers are installed, but your kernel should use it's own handlers as soon as possible.

Limitations

- Maps 1G of RAM
- SD cards other than SDHC Class 10 are not supported

Booting

- **SD card**: copy *bootboot.img* to ***FS0:\KERNEL8.IMG***. You'll need other firmware files (bootcode.bin, start.elf) as well. The GPT is not supported directly, therefore ESP partition has to be mapped in MBR so that Raspberry Pi firmware could find those files. The *mkboot* (https://github.com/bztsrc/bootboot/blob/master/aarch64-rpi/mkboot.c) utility will do that.

APPENDIX

A sample BOOTBOOT compatible kernel

```
/*
 * mykernel/kernel.c
 *
 * Copyright 2017 Public Domain BOOTBOOT bztsrc@github
 *
 * This file is part of the BOOTBOOT Protocol package.
 * @brief A sample BOOTBOOT compatible kernel
 *
 */

/* function to display a string, see below */
void puts(char *s);

/* we don't assume stdint.h exists */
typedef short int           int16_t;
typedef unsigned char       uint8_t;
typedef unsigned short int  uint16_t;
typedef unsigned int        uint32_t;
typedef unsigned long int   uint64_t;

#include <bootboot.h>

/* imported virtual addresses, see linker script */
extern BOOTBOOT bootboot;           // see bootboot.h
extern unsigned char *environment;  // configuration, UTF-8 text key=value pairs
extern uint8_t fb;                  // linear framebuffer mapped

/*****************************************
 * Entry point, called by BOOTBOOT Loader *
 *****************************************/
void _start()
{
    int x, y, s=bootboot.fb_scanline, w=bootboot.fb_width, h=bootboot.fb_height;

    // cross-hair to see screen dimension detected correctly
    for(y=0;y<h;y++) { *((uint32_t*)(&fb + s*y + (w*2)))=0x00FFFFFF; }
    for(x=0;x<w;x++) { *((uint32_t*)(&fb + s*(h/2)+x*4))=0x00FFFFFF; }

    // red, green, blue boxes in order
    for(y=0;y<20;y++) { for(x=0;x<20;x++) { *((uint32_t*)(&fb + s*(y+20) + (x+20)*4))=0x00FF0000; } }
    for(y=0;y<20;y++) { for(x=0;x<20;x++) { *((uint32_t*)(&fb + s*(y+20) + (x+50)*4))=0x0000FF00; } }
    for(y=0;y<20;y++) { for(x=0;x<20;x++) { *((uint32_t*)(&fb + s*(y+20) + (x+80)*4))=0x000000FF; } }

    // say hello
    puts("Hello from a simple BOOTBOOT kernel");

    // hang for now
    while(1);
}
```

```
/*************************
 * Display text on screen *
 *************************/
typedef struct {
    uint32_t magic;
    uint32_t version;
    uint32_t headersize;
    uint32_t flags;
    uint32_t numglyph;
    uint32_t bytesperglyph;
    uint32_t height;
    uint32_t width;
    uint8_t glyphs;
} __attribute__((packed)) psf2_t;
extern volatile unsigned char _binary_font_psf_start;

void puts(char *s)
{
    psf2_t *font = (psf2_t*)&_binary_font_psf_start;
    int x,y,kx=0,line,mask,offs;
    int bpl=(font->width+7)/8;
    while(*s) {
        unsigned char *glyph = (unsigned char*)&_binary_font_psf_start + font->headersize +
            (*s>0&&*s<font->numglyph?*s:0)*font->bytesperglyph;
        offs = (kx * (font->width+1) * 4);
        for(y=0;y<font->height;y++) {
            line=offs; mask=1<<(font->width-1);
            for(x=0;x<font->width;x++) {
                *((uint32_t*)((uint64_t)&fb+line))=((int)*glyph) & (mask)?0xFFFFFF:0;
                mask>>=1; line+=4;
            }
            *((uint32_t*)((uint64_t)&fb+line))=0; glyph+=bpl; offs+=bootboot.fb_scanline;
        }
        s++; kx++;
    }
}
```

A sample Makefile

```
#
# mykernel/Makefile
#
# Copyright 2017 Public Domain BOOTBOOT bztsrc@github
#
# This file is part of the BOOTBOOT Protocol package.
# @brief An example Makefile for sample kernel
#
#

CFLAGS = -Wall -fpic -ffreestanding -fno-stack-protector -nostdinc -nostdlib -I../

all: mykernel.x86_64.elf mykernel.aarch64.elf

mykernel.x86_64.elf: kernel.c
	x86_64-elf-gcc $(CFLAGS) -mno-red-zone -c kernel.c -o kernel.o
	x86_64-elf-ld -r -b binary -o font.o font.psf
	x86_64-elf-ld -nostdlib -nostartfiles -T link.ld kernel.o font.o -o mykernel.x86_64.elf
	x86_64-elf-strip -s -K fb -K bootboot -K environment mykernel.x86_64.elf
	x86_64-elf-readelf -hls mykernel.x86_64.elf >mykernel.x86_64.txt

mykernel.aarch64.elf: kernel.c
	aarch64-elf-gcc $(CFLAGS) -c kernel.c -o kernel.o
	aarch64-elf-ld -r -b binary -o font.o font.psf
	aarch64-elf-ld -nostdlib -nostartfiles -T link.ld kernel.o font.o -o mykernel.aarch64.elf
	aarch64-elf-strip -s -K fb -K bootboot -K environment mykernel.aarch64.elf
	aarch64-elf-readelf -hls mykernel.aarch64.elf >mykernel.aarch64.txt

clean:
	rm *.o *.elf *.txt
```

A sample linker script

```
/*
 * mykernel/link.ld
 *
 * Copyright 2017 Public Domain BOOTBOOT bztsrc@github
 *
 * This file is part of the BOOTBOOT Protocol package.
 * @brief An example linker script for sample kernel
 *
 */

mmio  = 0xfffffffffa000000;
fb    = 0xfffffffffc000000;
SECTIONS
{
    . = 0xfffffffffe00000;
    bootboot    = .; . += 4096;
    environment = .; . += 4096;
    .text : {
        KEEP(*(.text.boot)) *(.text .text.*)   /* code */
        *(.rodata .rodata.*)                   /* data */
        *(.data .data.*)
    }
    .bss (NOLOAD) : {                          /* bss */
        . = ALIGN(16);
        *(.bss .bss.*)
        *(COMMON)
    }
}
```

INDEX