towboot – a Multiboot-compatible bootloader for UEFI-based x86 systems

Niklas Sombert
niklas.sombert@uni-duesseldorf.de
Heinrich Heine University
Düsseldorf, North Rhine-Westphalia, Germany

Abstract

Current desktop and server computers run on firmware that is compatible to the UEFI standard[37] and responsible for initializing the hardware and loading a bootloader, e.g. GRUB[8]. The bootloader then prepares the early boot environment and hands control over to the operating system. Modern bootloaders provide many features and functionalities such as interactive scripting languages which results in large code bases – GRUB consists of 300 thousand lines of C/C++ and 64 thousand lines of assembly code. It is not surprising that numerous memory safety violations have been detected in various existing bootloaders[39]. We think many of these issues can be addressed by writing a bootloader in Rust. In this paper we present the design and implementation of *towboot* – a bootloader written in Rust for Multiboot 1 and 2 kernels on UEFI-based x86 / x86_64 systems.

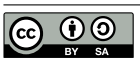
1 Introduction

Rust[27] is a systems programming language that allows writing low-level code using both high-level abstractions and assembly code. The compiler checks type- and memory-safety which is especially useful for programs that have to manually manage memory, such as bootloaders, as Uzlu and Şaykol[38] have suggested. Multiboot[6][7] is a file format used by many kernels.

towboot is a bootloader for Multiboot kernels on UEFI-based x86 systems, written in Rust, supporting most features of Multiboot 1 and 2, such as ACPI and SMBIOS. It comes with *towbootctl*, a small tool for installing the bootloader, a configuration file, kernels and modules to a disk or a newly created image.

2 Background

A bootloader is the glue between firmware and kernel, so it can make good use of existing technologies and software. The UEFI API is high-level and platform-agnostic, but the Multiboot standard requires dealing with some x86 specifics.



 DOI: https://dx.doi.org/10.1145.nnnnn. GI/ITG FG Operating Systems, February 2026, Düsseldorf, DE.

2.1 UEFI

The Unified Extensible Firmware Interface[37] is a standard specifying the API of a computer's firmware. It is designed to be both extensible and backwards compatible by using *protocols* with a UUID and an optional revision. In order for an application to use a protocol it has to be supported by the firmware or by an installed third-party driver or application. Each entity (e.g. a device or file) can support multiple protocols and is identified by an opaque handle. The `efi_main` function of an application receives a pointer to the *System Table*, alongside a handle identifying the application. This table contains a revision, handles for standard input, output and error and pointers to the *Boot* and *Runtime Services* and to additional configuration tables. It is available both during the boot process and the normal operation of a system, but the Boot Services are only available during boot. They allow an application to manage memory, to run other applications and to access all available protocols. Runtime Services are also available after booting and allow access to, for instance, the clock, power management, UEFI variables and the firmware's update mechanism.

The following protocols are important for building a bootloader, though there are more protocols which could be useful, e.g. mouse or block-based disk access:

- Loaded Image Protocol
- Simple File System Protocol / File Protocol
- Simple Text Input / Output Protocol
- Graphics Output Protocol

2.1.1 Execution environment. UEFI applications are relocatable Portable Executables that are loaded by the firmware or by another application. They are executed in the same CPU mode as the firmware, so (in most cases) 64-Bit Mode, with a 1:1 memory mapping. There are two booting modes:

Boot variables. Operating systems installed on the device write their bootloader and its command line to a new variable named `BootXXXX` (`XXXX` being a four-digit hexadecimal number) and add the number to the `BootOrder` variable.

Removable media. Operating systems installed on a disk place their bootloader at `\EFI\BOOT\BOOTarch.EFI` (where `arch` is the name of the architecture). That way, a single boot

medium can support multiple architectures, but supplying command line arguments is not possible in this case.

2.2 Multiboot

Multiboot[6][7] is a standard specifying the interaction between kernels and bootloaders. It emerged from the GNU GRUB[8] project which acts as a sort of reference implementation on the bootloader side. gnumach[11], GNU HURD[9]'s kernel, is another GNU project implementing the kernel side.

There are currently two major versions of Multiboot in use: 0.6.96 ("1") and 2.0 ("2"). While Multiboot 2 is modular and more flexible, version 1 remains popular, with even gnumach still on version 1. The information passed in version 2 is very similar, it is mostly the structure of the headers which differs. The standard allows a bootloader to load a kernel and modules such as an initial ramdisk image, passing information about the system.

The Rust bindings for Multiboot are provided by the *multiboot*[42] and *multiboot2*[25] crates, to which we added support for setting values and parsing the headers[31][30].

2.2.1 Passing information to the bootloader. The kernel image contains static information in the form of the Multiboot header, starting with a magic value and a checksum. It may contain more information that may require the bootloader to align the loaded modules at 4KB, to pass memory or video mode information, or to load the kernel as a flat binary. It may also signify that the kernel is UEFI-aware.

2.2.2 Passing information to the kernel. The bootloader places a Multiboot information struct in memory and a pointer to it in the `EBX` register. A magic value in `EAX` lets the kernel know that it was booted via Multiboot. This struct may contain the following information:

- Memory information
- Kernel command line
- Module information
- Symbol information
- Bootloader name
- Framebuffer information
- EFI pointers
- SMBIOS
- ACPI
- Networking information
- Relocation information
- Legacy: boot device, drives, config table, APM, VBE

2.2.3 Machine state. The standard requires the CPU to be in 32-bit Protected Mode, without paging or interrupts. Multiboot 2 also allows for UEFI-aware kernels to be started in the firmware's native mode. This is easier for both bootloader and kernel and makes it possible to stay in 64-Bit Mode on most systems.

2.2.4 Alternatives. Limine[16] has its own, operating system agnostic boot protocol. Both Linux[33] and NetBSD[21] roll their own boot protocol, although the Linux kernel also supports being loaded as a UEFI application and NetBSD also supports Multiboot. The *bootloader* crate[24] has its own boot protocol.

3 Related work

There are other bootloaders supporting UEFI and Multiboot:

Table 1: comparison of bootloaders

bootloader	version	LOC[a]	features
GRUB	2.14 (2026)	452249 C: 359613 ASM: 63912	scripting language, interactive shell, support for many disk / partition / file formats
Syslinux	6.03 (2014)	383509 C: 347028 ASM: 9287	lightweight, support for multiple file formats
Limine	10.6.3 (2026)	23347 C: 20939 ASM: 396	support for multiple file formats, own boot protocol
Easyboot	562783be (2025)	17653 C: 16446 ASM: 307	support for multiple file formats, own boot protocol
towboot	0.11.0 (2026)	5514[b] Rust: 3431 ASM: 20	small, written in mostly safe Rust[c]

[a] measured with *tokei*[41]
[b] excluding dependencies
[c] 95 percent of expressions, measured with *count-unsafe*[19]

- GRUB[8] is a widely used bootloader supporting various platforms and operating system interfaces. It features a menu where entries can be interactively edited and is configured via a complex scripting language. It acts as a sort of reference implementation of the Multiboot standard on the bootloader side.
- Syslinux[35] is a very lightweight bootloader supporting multiple platforms and some operating system interfaces. Its current release is from 2014.
- Limine[16] supports various platforms and operating systems. It also has an own boot protocol.
- Easyboot / Simpleboot[3][28] support many kernel file formats, operating systems and architectures.

These bootloaders are written in C and some of them have had multiple security-related bugs in the past.[39] Uzlu and Şaykol[38] suggested to use Rust for developing bootloaders for its memory safety and high-level semantics back in 2016. There are other bootloaders written in Rust:

- Redox[26] and Hermit[13] have bootloaders, but they are specific to the operating system.
- Sprout[4] only supports chain-loading.
- The bootloader crate[24] has its own boot protocol.
- Lukas Markeffsky[18] built a bootloader in his Bachelor's thesis, but it has no support for Multiboot.

4 Boot process

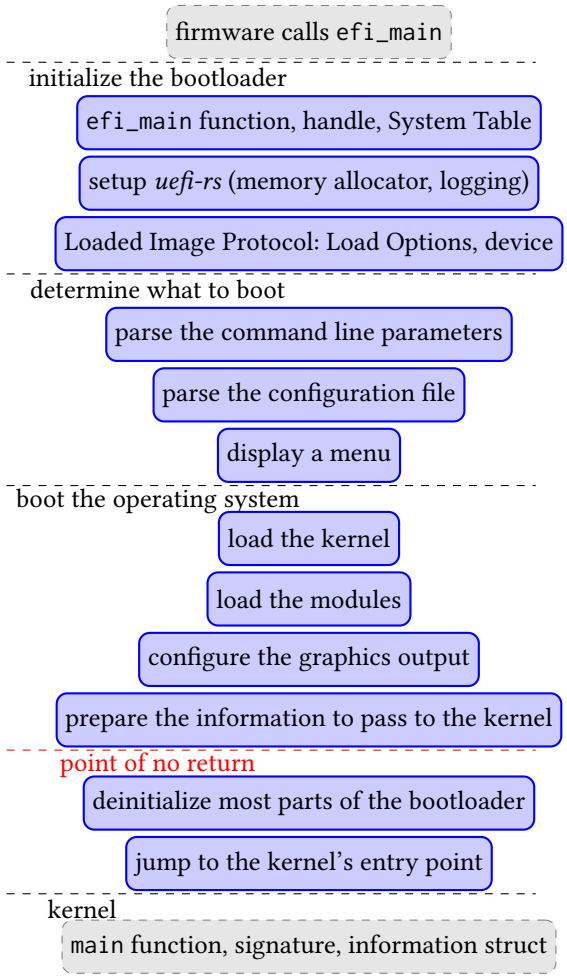


Figure 1: boot process (bootloader, other components)

The process from entering the `efi_main` function to jumping into the kernel can be split into the following steps:

4.1 Initialize the bootloader

The firmware calls the `efi_main` function with two parameters: a handle representing the application and a reference to the System Table. Together, they are used to setup `uefi-rs`, which will initialize the memory allocator and the logging framework, and, by using UEFI's Loaded Image Protocol, to acquire more information about the current state of the system, namely the options passed on the command line ("load options") and the device the application was loaded from (which is most likely going to be the ESP[1]).

4.2 Determine what to boot

The configuration containing what kernel and modules to load and what parameters to pass to them can come from the command line or from a TOML file (that itself may be specified on the command line). So, *towboot* first parses the command line parameters to determine whether to load a configuration file (and which one). It then loads the configuration file and displays a menu.

Command line parameters can be used for either manually booting from the EFI shell or in combination with another bootloader, such as rEFInd[29] or systemd-boot[32]. This method can also be used to configure `Boot####` entries in the firmware and use the firmware's boot manager (see [37] section 3.1).

4.3 Boot the operating system

4.3.1 Load the kernel. The kernel image can be a flat binary (AOut) or an ELF file. In both cases the file is loaded completely into memory. The Multiboot header then specifies the type and contains more information. Then parts of the file are copied to newly allocated memory (specified either by the Multiboot header or by the ELF Program Header). In case of an ELF file, the Section Header and symbols are also copied to memory.

4.3.2 Load the modules. The specified module images are loaded to newly allocated memory pages.

4.3.3 Configure the graphics output. The kernel's Multiboot header may specify a preferred text or graphics mode but UEFI's Graphics Output Protocol only allows a graphical mode to be used by writing to a framebuffer in memory (and this is not even guaranteed to work on all setups) – text mode, resolution change and more graphics features are only available by calling methods which the kernel may not have support for.

So, *towboot* lists the possible modes of the first GPU and compares them to the kernel's preferred resolution. If there is a match it is used, else it keeps using the current mode; in many cases, this will be the native resolution. This can also be forced with the `KeepResolution` quirk.

4.3.4 Prepare the information to pass to the kernel. The bootloader allocates the Multiboot information struct and fills it with information about the kernel command line, the loaded

[1] EFI System Partition, a FAT partition where installed UEFI applications reside

modules (and their command line), the ELF Section Header and symbols (if possible), the bootloader name, the framebuffer, the System Table, the image handle, ACPI, SMBIOS, whether the Boot Services have been exited, and where the kernel was loaded. Information about the memory configuration is not passed here because it could still change.

4.3.5 Deinitialize most parts of the bootloader. Exiting the Boot Services frees some parts of the memory and produces a memory map, but also causes the bootloader to lose access to the file system, console and memory allocator. This is the point of no return: It is not possible to go back to a menu or exit with an error code. The only error handling still possible is panicking: printing a message to the console (which may itself fail at this point), waiting and resetting the machine.

This memory map is then converted into the respective Multiboot structs: everything except broken memory, the Runtime Services' memory, memory-mapped IO or memory containing ACPI tables is marked as available, then adjacent entries are joined. The two fields describing the legacy "lower" and "upper" memory are also computed from the generated memory map. Then, the kernel is moved to the correct position in memory if this failed initially.

4.3.6 Jump to the kernel's entry point. If the kernel is not aware of UEFI, *towboot* sets up the machine state the Multiboot standard requires, before jumping to the kernel's entry point, passing the Multiboot signature and a pointer to the Multiboot information struct via the `EAX` and `EBX` registers.

5 Rust and memory management

5.1 UEFI and Rust

Rust[27] is a system programming language developed originally at Mozilla. It allows writing low-level code using both high-level abstractions and assembly code, if necessary. The compiler checks type- and memory-safety of the program, excluding code explicitly marked as `unsafe`. It comes with *cargo*, a tool that manages (among others) (cross-)compilation, tests, documentation and dependencies[2].

Compiling UEFI applications is supported by Rust itself as the `i686-unknown-uefi` and `x86_64-unknown-uefi` targets. They are rated as "Tier 2"[34] which means that prebuilt binaries exist, but they are not guaranteed to work. The standard library is partially usable: `core` and `alloc` work, and parts of `std` exist, but they are not useful enough yet to build a bootloader.

The *uefi-rs* crate[17] provides access to the API and maps it to mostly safe data structures and methods.

UEFI applications are usually written in C using either gnu-efi[10] or the EFI Development Kit[36]. C makes structuring the software much harder as there is no built-in support for either namespacing, dependency management or build automation. Using gnu-efi to build a UEFI application is further complicated by the different calling conventions and executable file formats between the GNU toolchain and UEFI. C also does not provide type- or memory-safety which makes bugs in the code much easier to miss.

[2]Both libraries and executables are called crates.

5.2 Bootloaders and memory management

Bootloaders are special when it comes to memory management, as noted by Uzlu and Şaykol[38]: In addition to the stack and the heap that are being used as usual, modules and kernel code need to be loaded to specific locations. Also, even though memory is managed by the firmware, bootloaders have full access to the whole memory; out-of-bounds array accesses or dereferencing invalid pointers usually do not cause exceptions or page faults.

That is a major reason why Rust is useful for writing bootloaders: invalid memory accesses are checked in large parts at compile-time and (in debug builds, at least) also at runtime.

5.2.1 Stack. Placing variables on the Rust stack by using local variables does always work, but is limited to structures that have their size known at compile time. This is used for most runtime data. This memory is tracked by *rustc* at compile time.

5.2.2 Heap. uefi-rs binds Rust's global memory allocator to the UEFI Boot Services' `allocate_pool`, so this memory is tracked both by *rustc* at compile time to some extend and by the firmware at runtime. This is used for everything with a dynamic size and no further special requirements, for instance the configuration file.

5.2.3 Whole pages. There are allocations with such special requirements, however: The kernel code has to be placed at the exact same spot which it was built to be placed at. Modules may need to be loaded page-aligned, so that the kernel can simply map them into its paging. Some kernels expect the information structs to be placed low in memory.

In these cases, the UEFI Boot Services' `allocate_pages` function is wrapped by the `mem::UefiAllocation` struct. It contains a custom `Drop` implementation which calls the Boot Services' `free_pages` function to propagate freed memory back from Rust to UEFI. This struct is, in turn, wrapped by the `mem::Allocation` struct, because `allocate_pages` can just allocate whole pages and kernel code sections are often not page-aligned.

There is another caveat: The address at which the kernel has to be loaded to may not be available when loading the kernel. It might be used by UEFI's Boot Services, for example. This is solved by loading the kernel to a different address first and copying it to the correct address after exiting Boot

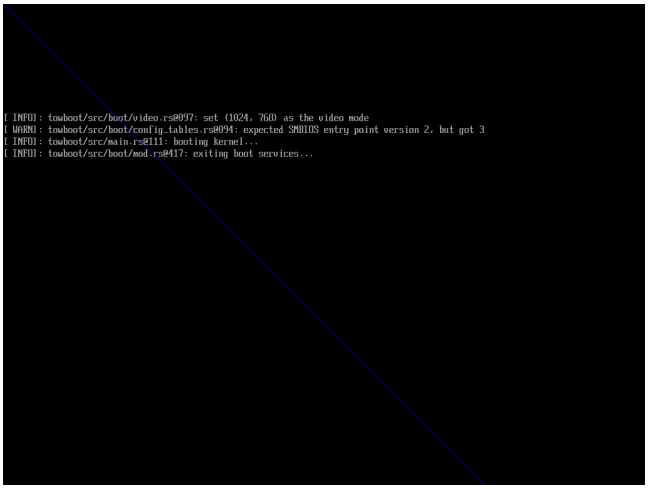


Figure 2: booting the Multiboot 2 example kernel

Services, if the destination address is not marked as reserved in the memory map passed to the kernel.

5.2.4 References passed to the kernel. Information is passed to the kernel by writing the address of the Multiboot information struct to a register before jumping to the kernel's entry point. Some parts of this information are scalar values contained in the passed struct but other parts are pointers to additional structs. `rustc` may not able to determine that these structs are being used by the kernel, so they are intentionally leaked by calling `core::mem::forget` on them to make sure that they are not preemptively freed.

6 Evaluation

6.1 Automated testing

The *multiboot* and *multiboot2* crates have unit tests.

This is not feasible for the whole bootloader: large parts of it depends on the UEFI API. This API has a quite large surface and would be difficult to mock. So instead, there are integration tests that compile the bootloader, build images containing the bootloader, a configuration file and a kernel, boot them in QEMU and ensure the output matches the expected one. We used the example kernels from the Multiboot specifications[6][7] for this, slightly altered to output to the serial port instead of CGA which is easier to capture for testing and the used firmware does not support CGA. Framebuffer usage is not checked automatically.

6.2 Manual testing

Booting the example kernels manually displays a diagonal blue line via the passed framebuffer (see figure 2) and outputs various information about the system on the serial port:

```
cargo xtask build -- -kernel "tests/multiboot2/kernel foo"
Finished dev profile [unoptimized + debuginfo] target(s) in 0.26s
Running `target/debug/xtask build -- -kernel 'tests/multiboot2/kernel foo'`
[INFO  xtask] building for i686, pass --no-i686 to skip this
Finished dev profile [unoptimized + debuginfo] target(s) in 0.18s
[INFO  xtask] building for x86_64, pass --no-x86-64 to skip this
Finished dev profile [unoptimized + debuginfo] target(s) in 0.17s
[INFO  towbootctl] calculating image size
[INFO  towbootctl] adding "tests/multiboot2/kernel" as "kernel"
[INFO  towbootctl] adding "/tmp/.tmpyk6aoe" as "towboot.toml"
[INFO  towbootctl] adding "target/i686-unknown-uefi/debug/towboot.efi" as "EFI/Boot/bootia32.efi"
[INFO  towbootctl] adding "target/x86_64-unknown-uefi/debug/towboot.efi" as "EFI/Boot/bootx64.efi"
[INFO  towbootctl] creating image at image.img (size: 4 MiB)
> cargo xtask boot-image
Finished dev profile [unoptimized + debuginfo] target(s) in 0.24s
Running `target/debug/xtask boot-image`
[INFO  towbootctl] getting firmware
[INFO  cached_path::cache] Cached version of https://retrage.github.io/edk2-nightly/bin/RELEASEIa32_OVMF.fd is up-to-date
[INFO  towbootctl] spawning QEMU
WARNING: Image format was not specified for 'image.img' and probing guessed raw. Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted. Specify the 'raw' format explicitly to remove the restrictions.
0
0
BdsDxe: loading Boot0002 "UEFI QEMU HARDDISK QM00001 " from PciRoot(0x0)/Pci(0x1,0x1)/Ata(Primary,Master,0x0)
BdsDxe: starting Boot0002 "UEFI QEMU HARDDISK QM00001 " from PciRoot(0x0)/Pci(0x1,0x1)/Ata(Primary,Master,0x0)
[ INFO]: towboot/src/file.rs@053: loading file '\towboot.toml'...
[ INFO]: towboot/src/main.rs@107: loading kernel...
[ INFO]: towboot/src/file.rs@053: loading file 'kernel'...
[ INFO]: towboot/src/boot/mod.rs@349: kernel is loaded and bootable
[ INFO]: towboot/src/boot/mod.rs@357: loaded 0 modules
[ INFO]: towboot/src/boot/video.rs@028: setting up the video...
[ WARN]: towboot/src/boot/video.rs@039: color depth will be 24-bit, but the kernel wants 32
[ INFO]: towboot/src/boot/video.rs@097: set (1024, 768) as the video mode
[ WARN]: towboot/src/boot/config_tables.rs@094: expected SMBIOS entry point version 2, but got 3
[ INFO]: towboot/src/main.rs@111: booting kernel...
[ INFO]: towboot/src/boot/mod.rs@417: exiting boot services...
Announced mbi size 0x2220
Tag 0x1, Size 0xc
Command line = foo
Tag 0x2, Size 0x17
Boot loader name = towboot 0.11.0
Tag 0x4, Size 0x10
mem_lower = 640KB, mem_upper = 7192KB
Tag 0x6, Size 0xa78
mmap
 base_addr = 0x00, length = 0x0a0000, type = 0x1
 base_addr = 0x0100000, length = 0x0706000, type = 0x1
 base_addr = 0x0806000, length = 0x02000, type = 0x4
 base_addr = 0x0808000, length = 0x08000, type = 0x1
 base_addr = 0x0810000, length = 0x0f0000, type = 0x4
 base_addr = 0x0900000, length = 0x0e7bc000, type = 0x1
 base_addr = 0x0f0bc000, length = 0x0114000, type = 0x2
 base_addr = 0x0f1d0000, length = 0x09ab000, type = 0x1
 base_addr = 0x0fb7b000, length = 0x0280000, type = 0x2
 base_addr = 0x0fdfb000, length = 0x012000, type = 0x3
 base_addr = 0x0fe0d000, length = 0x080000, type = 0x4
 base_addr = 0x0fe8d000, length = 0x03d000, type = 0x1
 base_addr = 0x0feca000, length = 0x02000, type = 0x4
 base_addr = 0x0fecc000, length = 0x028000, type = 0x1
 base_addr = 0x0fef4000, length = 0x084000, type = 0x2
 base_addr = 0x0ff78000, length = 0x088000, type = 0x4
 base_addr = 0x00, length = 0x00, type = 0x2
 [...]
Tag 0x8, Size 0x26
Tag 0x9, Size 0x1f4
```

```
65 Tag 0xb, Size 0xc
66 Tag 0xd, Size 0x1b5
67 Tag 0xd, Size 0x1ae
68 Tag 0xe, Size 0x1c
69 Tag 0xf, Size 0x2c
70 Tag 0x11, Size 0x1168
71 Tag 0x13, Size 0xc
72 Total mbi size 0x2220
73 Halted.
```

Listing 1: booting the Multiboot 2 example kernel

This manual test was performed both in QEMU 8.2.2 and on real hardware (a Dell Optiplex 9020 with an Intel Core i7-4770 and 8GiB RAM).

Various existing operating systems have kernels that are Multiboot-compatible. We tried booting some of them:

Table 2: tested operating systems

OS	version	status
NetBSD[20]	10.1 (i386)	works, with 64-bit OVMF and the ForceOverwrite quirk
HelenOS[12]	0.14.1 (ia32, amd64)	works
Lemon OS[15]	nightly-2024-07-12	works
GNU HURD[9]	2025 (i386)	does not work due to ACPI table placement on QEMU+OVMF
FlingOS[22]	2015-09-14	works, with 64-bit OVMF and the ForceOverwrite quirk
FiwixOS[5]	3.5	works, with the LowerAllocations quirk
9front[1]	2025-10-11 RELEASE (amd64)	works, with the LowerAllocations quirk
OpenIndiana[23]	2025.10	does not work

7 Conclusion

7.1 Summary

The abstractions provided by UEFI allow for writing straightforward and high-level bootloaders in comparison to the legacy BIOS API. *uefi-rs* maps them nicely to Rust data structures and methods. The Rust support for this environment is rather good, but there are still parts of the standard library missing. The machine state specified by Multiboot requires a bit of x86-specifics and assembly code.

towboot shows that is possible to boot operating systems on present-day hardware by using Rust, gaining type- and memory-safety[3], higher-level programming paradigms such as *map*, *match* and *filter*, better and earlier error messages and dependency management in comparison to using C. In comparison to other bootloaders, *towboot* has fewer features (no drivers for file systems, no scripting language, no support for displaying images and a pretty primitive menu), but fewer features also lead to a smaller attack surface[39]. The code is available at https://github.com/hhuOS/towboot.

7.2 Further Improvements

Some features are missing and could be added in the future:

7.2.1 Secure Boot. Most modern systems come with Secure Boot enabled which should ensure that only signed code is running in Ring 0. For this to work, the firmware checks the signature before loading applications which provides some protection against malware.[40]

Just signing the main bootloader executable with a key that is trusted by the firmware would be enough to be able to boot, but this would entirely circumvent this security measure: Any Multiboot-compatible kernel could be booted, no matter whether correctly signed or not.

Properly implementing this would mean to add signatures for at least the kernel[4] in a backwards-compatible way and requiring the kernels to verify code loaded into kernel space.

7.2.2 64-bit, UEFI-unaware kernels. *towboot* only supports i686 kernels as specified by Multiboot 1 and 2 and UEFI-aware i686 and x86_64 kernels as specified by Multiboot 2, the latter only on systems with 64-bit firmware. Some bootloaders (e.g. Syslinux) support elf64 Multiboot 1 kernels[2]. Adding support for this requires detecting CPU support and switching to Long Mode on 32-bit firmware.

7.2.3 other CPU architectures. UEFI also supports Itanium, ARM and RISC-V (see [37] section 3.5.1.1). Multiboot 2 supports just x86 and MIPS (see [7] section 3.2), so there is no other official overlap. There have been unofficial proposals[14] for Multiboot on ARM, though.

7.2.4 compatibility with more OS kernels. As seen in section 6.2, there are existing Multiboot kernels that *towboot* fails to boot. This can probably be fixed.

References

[1] [SW], 9front. URL: https://9front.org/Retrieved July 7, 2025 from.
[2] [SW], Doc/mboot. Syslinux Wiki. URL: https://wiki.syslinux.org/wiki/index.php?title=Doc/mbootRetrieved Jan. 1, 2021 from.
[3] [SW], Easyboot. URL: https://gitlab.com/bztsrc/easyboot/Retrieved Oct. 31, 2025 from.

[3] Interfacing with some of the UEFI APIs, memory management and jumping to the kernel require a few pieces of unsafe code.

[4] The modules could be signed, too, but just signing the kernel would bring about the level of security modern Linux distributions provide.

[4] [SW] edera-dev, sprout. URL: https://github.com/edera-dev/sprout Retrieved Dec. 1, 2025 from.

[5] [SW], Fiwix. URL: https://www.fiwix.org/Retrieved July 7, 2025 from.

[6] Bryan Ford and Erich Stefan Boleyn. [n. d.] *Multiboot Specification version 0.6.96*. Free Software Foundation, Inc. Retrieved Oct. 13, 2020 from https://www.gnu.org/software/grub/manual/multiboot/multiboot.html.

[7] Bryan Ford and Erich Stefan Boleyn. [n. d.] *Multiboot2 Specification version 2.0*. GNU. Free Software Foundation, Inc. Retrieved Oct. 13, 2020 from https://www.gnu.org/software/grub/manual/multiboot2/multiboot.html.

[8] [SW], GNU GRUB. GNU. URL: https://www.gnu.org/software/grub/Retrieved Oct. 13, 2020 from.

[9] [SW], GNU Hurd. GNU. URL: https://www.gnu.org/software/hurd/Retrieved Jan. 1, 2021 from.

[10] [SW], gnu-efi. URL: https://sourceforge.net/projects/gnu-efi/Retrieved Jan. 1, 2021 from.

[11] [SW], gnumach. GNU. URL: https://www.gnu.org/software/hurd/microkernel/mach/gnumach.htmlRetrieved Dec. 5, 2020 from.

[12] [SW], HelenOS. URL: http://www.helenos.org/Retrieved June 30, 2023 from.

[13] [SW] hermit, loader. URL: https://github.com/hermit-os/loader Retrieved Dec. 1, 2025 from.

[14] jncronin. [n. d.] *Multiboot ARM extensions v0.1*. Retrieved Oct. 9, 2025 from https://github.com/jncronin/rpi-boot/blob/master/MULTIBOOT-ARM.

[15] [SW], Lemon OS. URL: https://lemonos.org/Retrieved June 30, 2023 from.

[16] [SW], Limine. URL: https://limine-bootloader.org/Retrieved June 8, 2023 from.

[17] [SW] Gabriel Majeri, uefi. crates.io. URL: https://crates.io/crates/uefiRetrieved Dec. 4, 2020 from.

[18] Lukas Markeffsky. 2023. Ein minimaler bootloader in rust. (2023). https://markeffl.pages.cms.hu-berlin.de/yaros/doc/thesis.pdf.

[19] [SW], mkroening/count-unsafe. URL: https://github.com/mkroening/count-unsafeRetrieved Jan. 19, 2026 from.

[20] [SW], NetBSD. URL: https://netbsd.org/Retrieved June 30, 2023 from.

[21] [n. d.] *NetBSD Manual Pages*. Chap. boot(8). Retrieved Jan. 4, 2021 from https://man.netbsd.org/NetBSD-9.1-STABLE/x86/boot.8.

[22] [SW] Edward Nutting, FlingOS. URL: http://www.flingos.co.uk/Retrieved Jan. 1, 2021 from.

[23] [SW], OpenIndiana. URL: https://www.openindiana.org/Retrieved June 30, 2023 from.

[24] [SW] Philipp Oppermann, bootloader. crates.io. URL: https://crates.io/crates/bootloaderRetrieved Nov. 5, 2025 from.

[25] [SW] Philipp Oppermann, multiboot2. crates.io. URL: https://crates.io/crates/multiboot2Retrieved June 23, 2023 from.

[26] [SW], redox-os/bootloader. URL: https://gitlab.redox-os.org/redox-os/bootloaderRetrieved Dec. 1, 2025 from.

[27] [SW], Rust Programming Language. URL: https://www.rust-lang.org/Retrieved Jan. 1, 2021 from.

[28] [SW], Simpleboot. URL: https://gitlab.com/bztsrc/simpleboot Retrieved Oct. 31, 2025 from.

[29] [SW] Roderick W. Smith, The rEFInd Boot Manager. URL: https://www.rodsbooks.com/refind/Retrieved Oct. 14, 2020 from.

[30] [SW] Niklas Sombert, Add a builder to multiboot2 (Pull Request 133). rust-osdev/multiboot2. URL: https://github.com/rust-osdev/multiboot2/pull/133Retrieved June 23, 2023 from.

[31] [SW] Niklas Sombert, Add some functionality for using this in bootloaders (Pull Request 8). gz/rust-multiboot. URL: https://github.com/gz/rust-multiboot/pull/8Retrieved Jan. 4, 2021 from.

[32] [SW], systemd-boot UEFI Boot Manager. URL: https://www.freedesktop.org/wiki/Software/systemd/systemd-boot/Retrieved Oct. 14, 2020 from.

[33] [n. d.] *The Linux Kernel Documentation*. Chap. x86-specific Documentation: The Linux/x86 Boot Protocol. Retrieved Jan. 4, 2021 from https://www.kernel.org/doc/html/v5.10/x86/boot.html.

[34] [n. d.] *The rustc book*. Chap. Platform Support. Retrieved June 23, 2023 from https://doc.rust-lang.org/nightly/rustc/platform-support.html.

[35] [SW], The Syslinux Project. URL: https://www.syslinux.org/Retrieved Jan. 2, 2021 from.

[36] [SW], tianocore/edk2. URL: https://github.com/tianocore/edk2 Retrieved Jan. 1, 2021 from.

[37] 2020. *Unified Extensible Firmware Interface (UEFI) Specification, Version 2.8 Errata B, May 2020*. UEFI Forum. Retrieved Oct. 13, 2020 from https://uefi.org/sites/default/files/resources/UEFI%20Spec%202.8B%20May%202020.pdf.

[38] Tunç Uzlu and Ediz Şaykol. 2016. Utilizing rust programming language for efi-based bootloader design. In *RTA-CSIT*, 100–106. Retrieved Oct. 22, 2025 from https://ceur-ws.org/Vol-1746/.

[39] Jianqiang Wang, Meng Wang, Qinying Wang, Nils Langius, Li Shi, Ali Abbasi, and Thorsten Holz. 2025. A comprehensive memory safety analysis of bootloaders. In Network and Distributed System Security Symposium, (Feb. 2025). Retrieved Oct. 22, 2025 from https://www.ndss-symposium.org/wp-content/uploads/2025-330-paper.pdf.

[40] Richard Wilkins and Brian Richardson. 2013. Uefi secure boot in modern computer security solutions. (2013). Retrieved Oct. 25, 2020 from https://uefi.org/sites/default/files/resources/UEFI_Secure_Boot_in_Modern_Computer_Security_Solutions_2019.pdf.

[41] [SW], XAMPPRocky/tokei. URL: https://github.com/XAMPPRocky/tokeiRetrieved Jan. 19, 2026 from.

[42] [SW] Gerd Zellweger, multiboot. crates.io. URL: https://crates.io/crates/multibootRetrieved Jan. 4, 2021 from.