

# uefi-rs

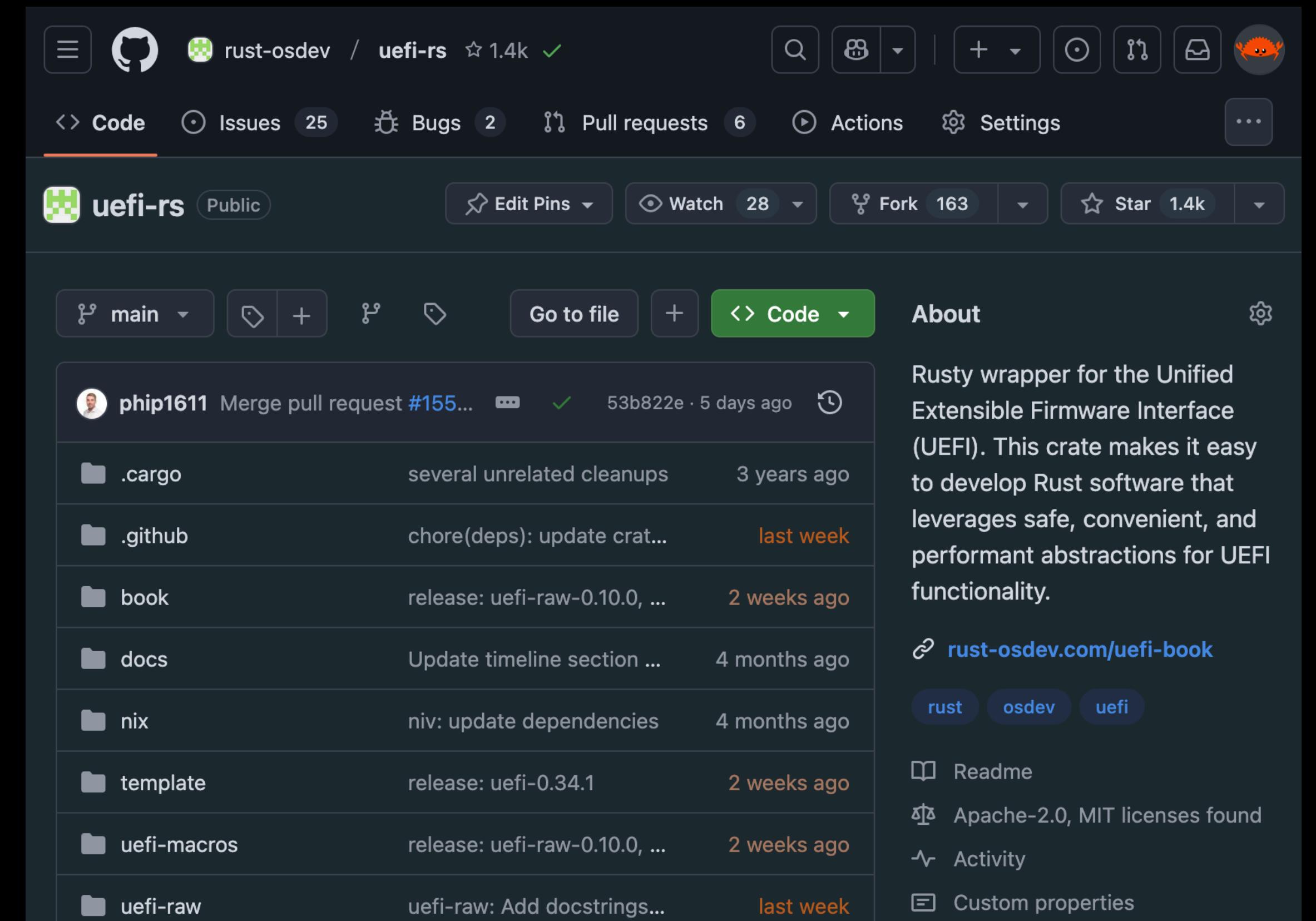
The story of a low-level Rust library



Gabriel Majeri - February 2025

# What is uefi-rs?

- uefi-rs is a low-level open source **Rust** library for interacting with **UEFI**-compatible **firmware**.
- I've started this project back in 2017 and now it has grown to **850.000+** **downloads** on [crates.io](#) and **1400+ GH stars**.
- This talk will be about what uefi-rs is, what it does, how it started and what you can do if you want to become a successful [open source maintainer](#).



# What is UEFI?



UNIFIED EXTENSIBLE  
FIRMWARE INTERFACE

# The startup process

- When you press the power switch and **turn on** your computer, the various hardware devices need to be initialized in order for the user-installed operating system to be loaded into memory and executed.
- Furthermore, the OS kernel needs a generic way to **interface** with the hardware components, at least until it loads its own drivers and starts reading the hardware configuration (e.g. through PCI).

# The Basic Input/Output System (BIOS)

- The solution is to provide such functionality through **firmware**, usually embedded into the motherboard as a **Read-Only Memory** (ROM) module.
- One of the earliest examples of such an approach is the **Basic Input/Output System** (BIOS), found in the original **IBM Personal Computer**.
- The interface of the original system was **reverse engineered** by other companies and then started to be used in other **PC clones**.



# The Basic Input/Output System (BIOS)

## Example of BIOS functions

- OS developers could rely on the BIOS to **initialize the HW**, **load their kernels into memory** and **hand over execution** while leaving the system in a **known state**.
- The BIOS offered quite a few abstract/generic **runtime services** (utility procedures) for programmers.

### Common functions

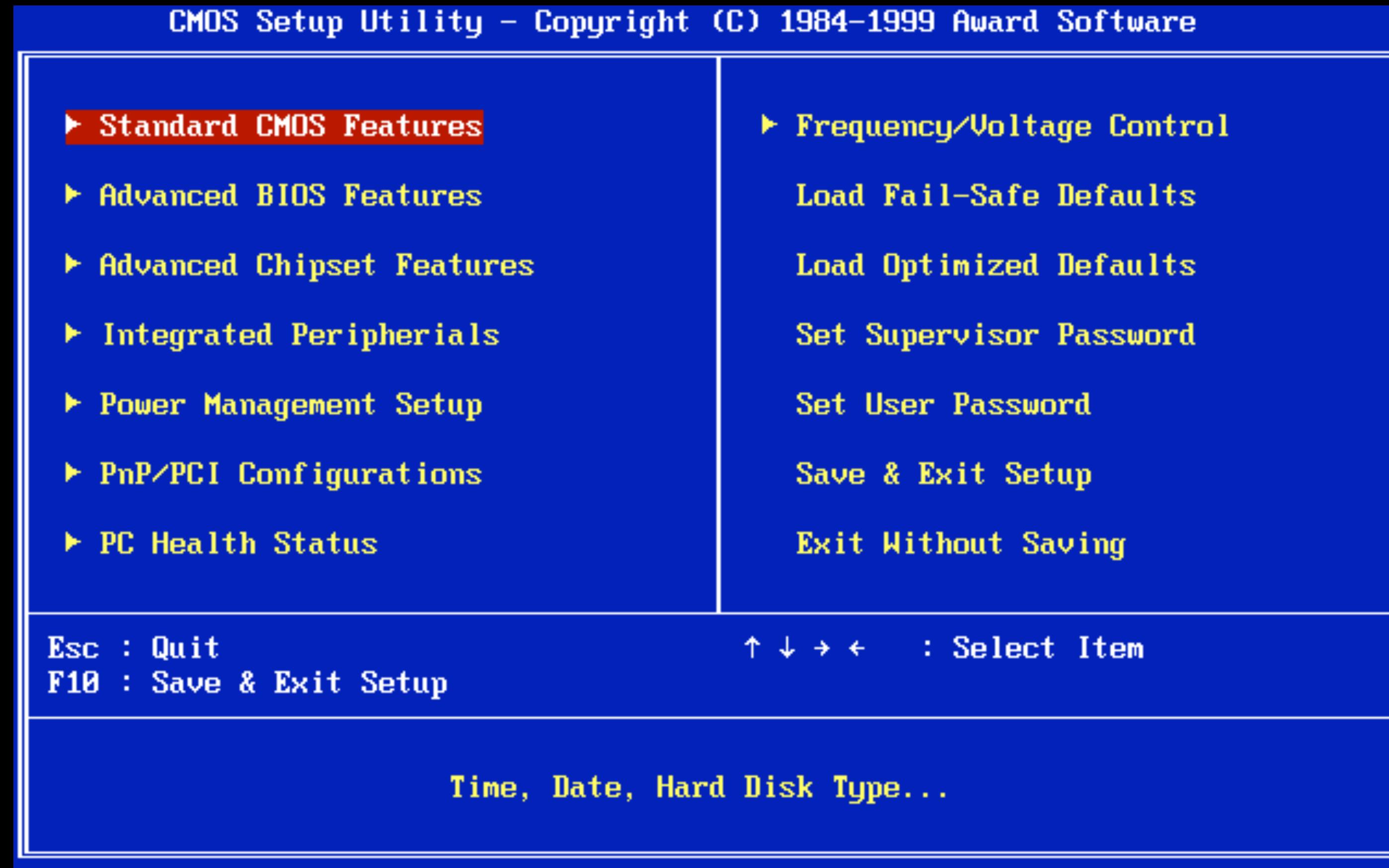
Unfortunately, RBIL does not clearly indicate which BIOS functions are "generic" (in some "standard" BIOS functions grew over time, so if you go back far enough in time you can use them). This list is commonly used in most current OSes.

- INT 0x10, AH = 1 -- set up the cursor
- INT 0x10, AH = 3 -- cursor position
- INT 0x10, AH = 0xE -- display char
- INT 0x10, AH = 0xF -- get video page and mode
- INT 0x10, AH = 0x11 -- set 8x8 font
- INT 0x10, AH = 0x12 -- detect EGA/VGA
- INT 0x10, AH = 0x13 -- display string
- INT 0x10, AH = 0x1200 -- Alternate print screen
- INT 0x10, AH = 0x1201 -- turn off cursor emulation
- INT 0x10, AX = 0x4F00 -- video memory size
- INT 0x10, AX = 0x4F01 -- VESA get mode information call

Source: [OSDev wiki](#)

# The Basic Input/Output System (BIOS)

## Example of BIOS configuration interface



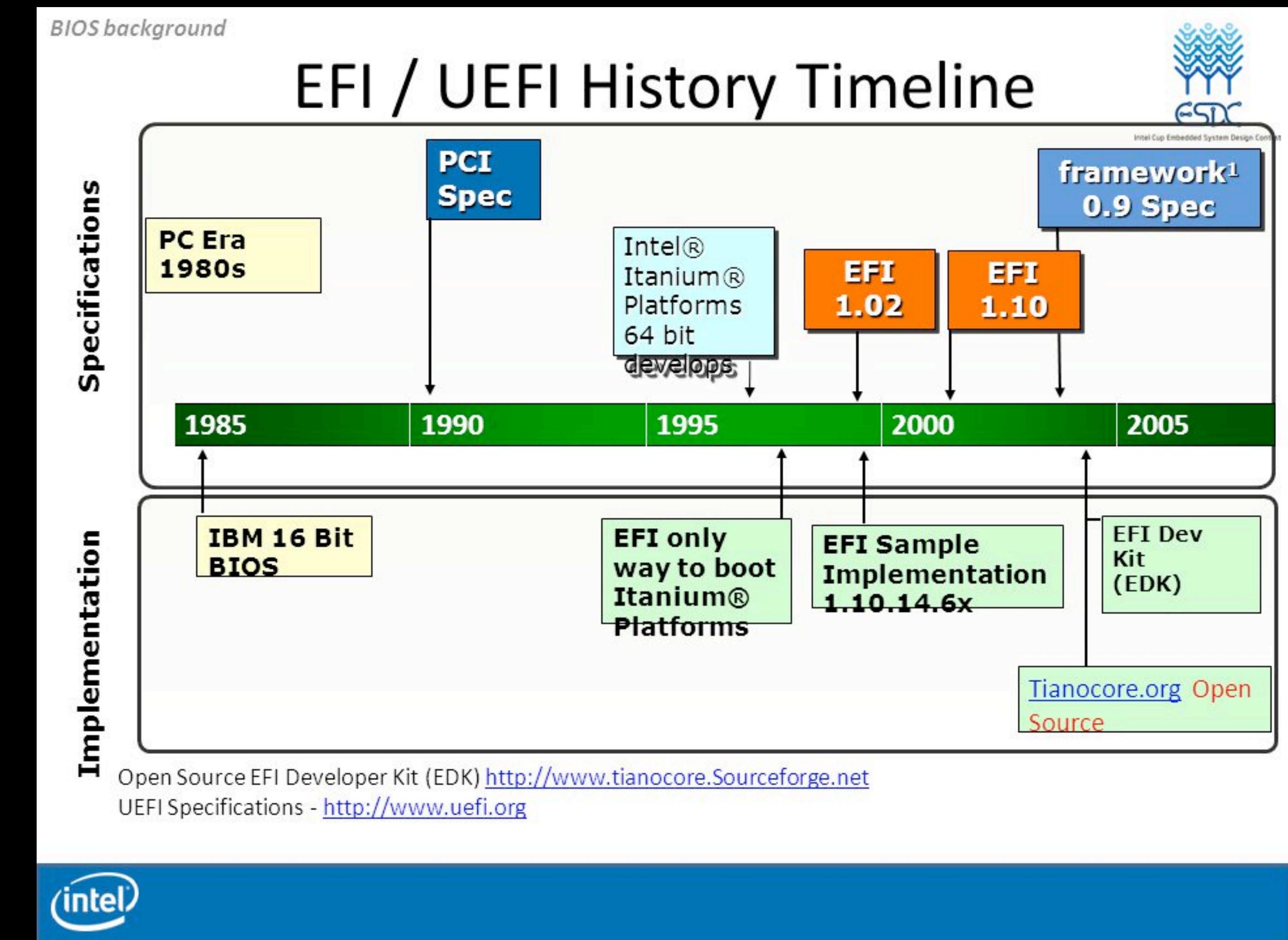
# So what's the problem with the BIOS?

- The BIOS code is written to run in **16 bit (real) mode**. Most modern OSes run on **32-bit protected mode** or more commonly nowadays **64-bit long mode**.
- The BIOS interface was, for most of its history, **undocumented** and **unstandardized**; there's no motherboard implementing “all” of the possible BIOS functions. It's also tricky to detect which features are available and which are missing (without having a hardware database prepared in advance).
- The BIOS lacks support for advanced **security** features (e.g. digital signatures for firmware code).

# Coming up with a better BIOS

## AKA how UEFI came to be

- In the 1990's, **HP** and **Intel** started developing the **Itanium** 64-bit architecture (which was backwards-incompatible with x86-32).
- They needed a **BIOS-like** interface adapted for the large, 64-bit **IA64** server systems.
- The **Extensible Firmware Interface** (EFI) standard was thus born.



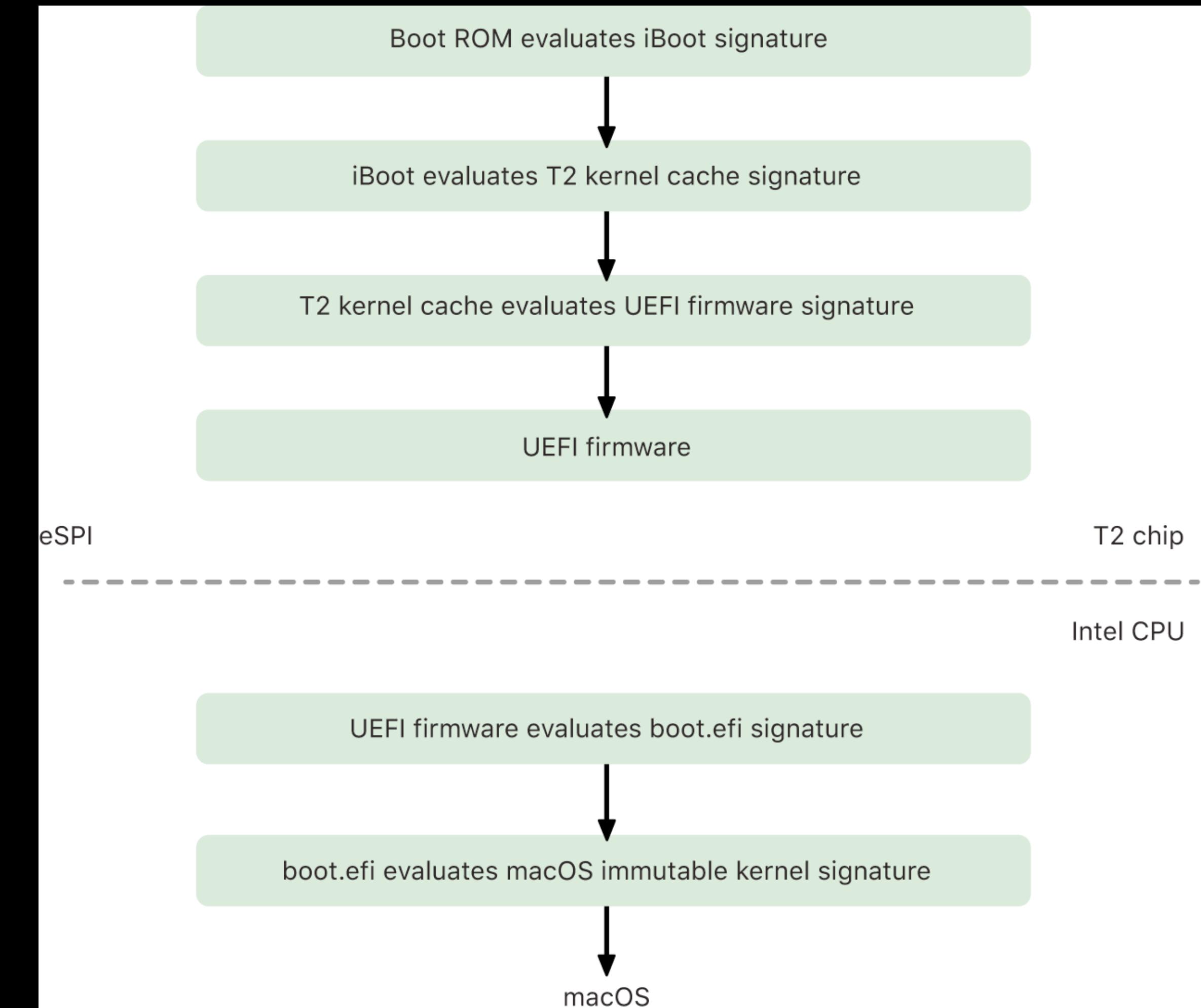
# Unified EFI

- In 2005, Intel donated the EFI specification to the **Unified EFI Forum**, an alliance of several technology companies and BIOS manufacturers.
- In 2006, version 2.0 of the UEFI specification was released, adding support for cryptographic primitives and security.



# UEFI goes viral

- When **Apple** switched to using **Intel** processors in 2006, they also adopted **UEFI** for firmware.
- Nowadays **UEFI** is used **everywhere** (especially on the desktop/laptop/server platforms): x86 PCs, Mac devices, ARM servers etc.
- Could be used for **embedded** as well (in theory), but it's pretty heavyweight and usually overkill.



# How is UEFI structured?

## The internals

- UEFI binaries (bootloaders/kernels/drivers) are actually PE/**COFF** executables with some weird peculiarities
- The main entry point of the executable is given a **pointer** to the UEFI “**system table**”, which provides a set of function pointers to built-in functionality

### 4.1.1 EFI\_IMAGE\_ENTRY\_POINT

#### Summary

This is the main entry point for a UEFI Image. This entry point is the same for UEFI applications and UEFI drivers.

#### Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IMAGE_ENTRY_POINT) (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE     *SystemTable
);
```

#### Parameters

##### ImageHandle

The firmware allocated handle for the UEFI image.

##### SystemTable

A pointer to the EFI System Table.

#### Description

This function is the entry point to an EFI image. An EFI image is loaded and relocated in system memory by the EFI Boot Service [EFI\\_BOOT\\_SERVICES.LoadImage\(\)](#). An EFI image is invoked through the EFI Boot Service [EFI\\_BOOT\\_SERVICES.StartImage\(\)](#).

```
typedef struct {
    EFI_TABLE_HEADER
    CHAR16
    UINT32
    EFI_HANDLE
    EFI_SIMPLE_TEXT_INPUT_PROTOCOL
```

```
Hdr;
*FirmwareVendor;
FirmwareRevision;
ConsoleInHandle;
*ConIn;
```

### 4.3. EFI System Table

# What is uefi-`rs`?

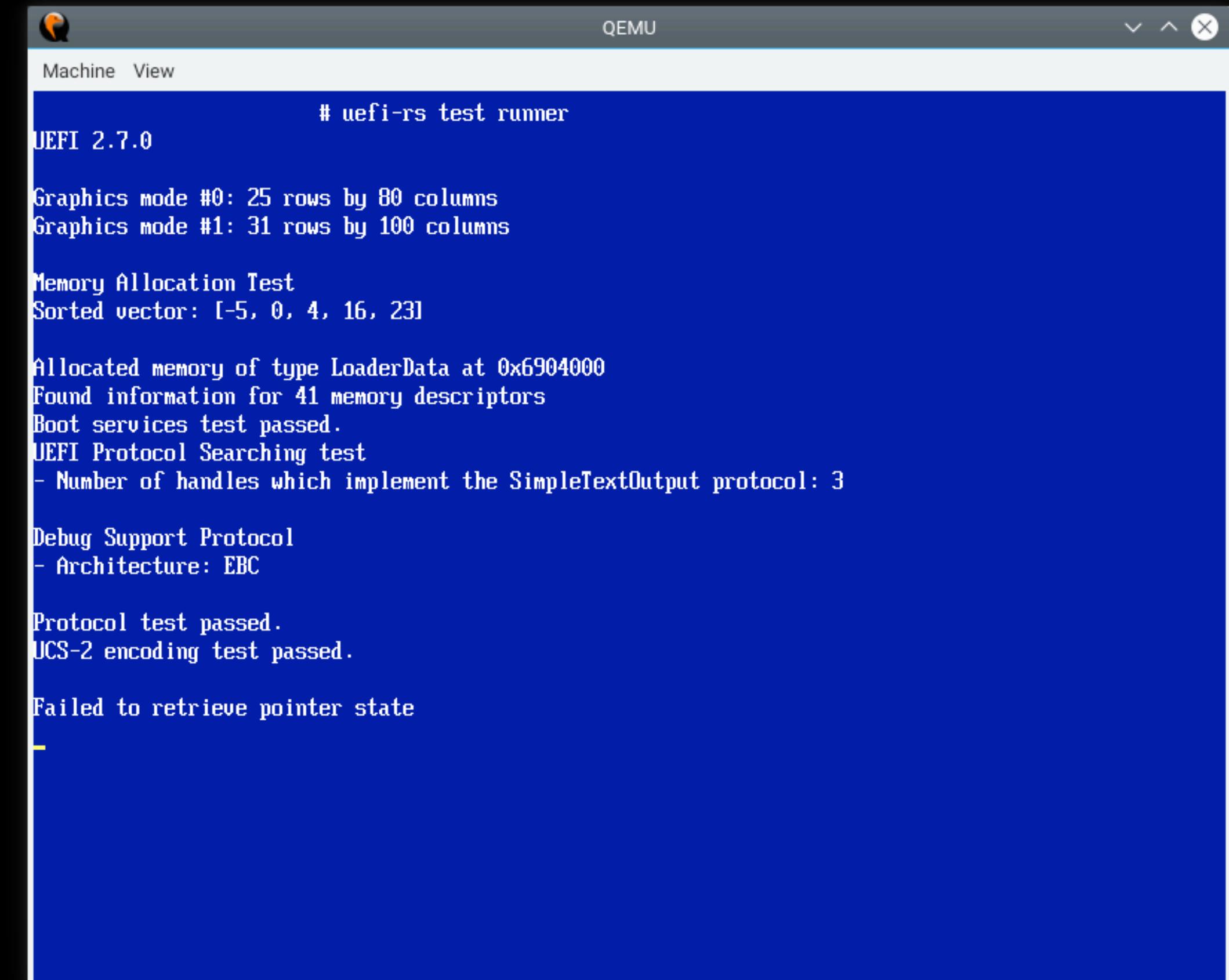


# What is uefi-rs?

- uefi-rs is an **open source** **Rust** crate (library) providing safe and performant wrappers for the most commonly-used protocols (interfaces) and data structures from the UEFI specification.
- **Not** meant to be **exhaustive** nor **un-opinionated**; we want to define what we believe to be **the best way** to write UEFI-compatible apps and drivers using Rust, and this means we still have a long way to go to covering the whole UEFI API (so far).

# uefi-rs example code

```
7
8     use std::os::uefi as uefi_std;
9     use uefi::runtime::ResetType;
10    use uefi::{Handle, Status};
11
12    /// Performs the necessary setup code for the `uefi` crate.
13    fn setup_uefi_crate() {
14        let st = uefi_std::env::system_table();
15        let ih = uefi_std::env::image_handle();
16
17        // Mandatory setup code for `uefi` crate.
18        unsafe {
19            uefi::table::set_system_table(st.as_ptr().cast());
20
21            let ih = Handle::from_ptr(ih.as_ptr().cast().unwrap());
22            uefi::boot::set_image_handle(ih);
23        }
24    }
25
26    fn main() {
27        println!("Hello World from uefi_std");
28        setup_uefi_crate();
29        println!("UEFI-Version is {}", uefi::system::uefi_revision());
30        uefi::runtime::reset(ResetType::SHUTDOWN, Status::SUCCESS, None);
31    }
}
```



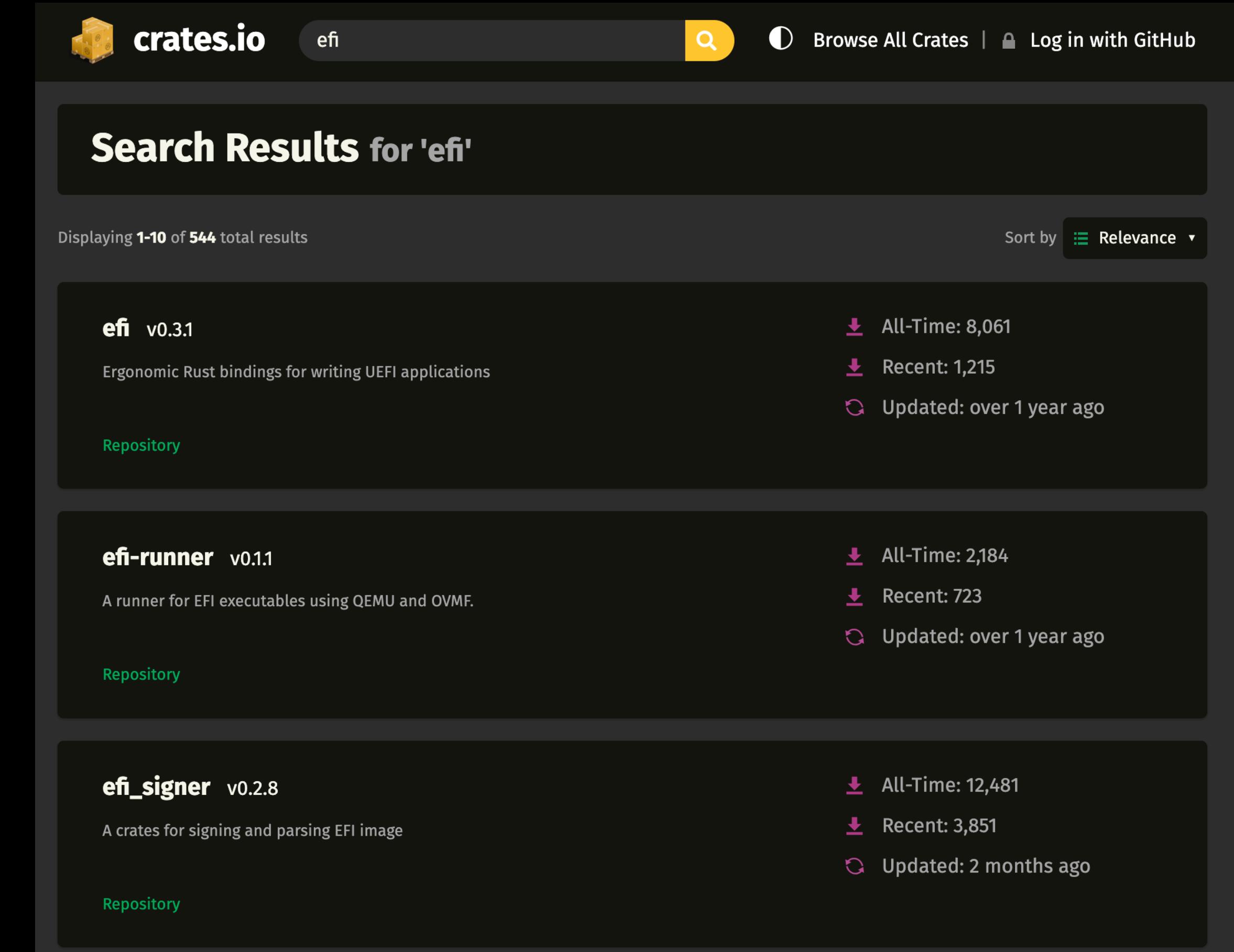
# Features offered by uefi-rs

- Scripts/tools for creating .efi binaries
- Macros for defining UEFI-compatible entry points
- Wrappers for system (boot/runtime services) tables
- Protocol (interface) localization and initialization
- Protocol definitions for various hardware devices (console, graphics output, block devices, filesystems, network controllers etc.)
- Lots of examples/sample code and unit tests for many standard UEFI features (is kinda like a conformance spec; we discovered plenty of bugs in the QEMU/OVMF implementation along the way)

# Why choose uefi-rs?

# The Rust-UEFI ecosystem

- `uefi-rs` is *not* the first *nor* the oldest Rust library for interacting with UEFI-compatible hardware.
- However, at the time I created the repo, other crates were either **raw bindings** (providing little to no abstraction or convenience wrappers) or were **badly documented** (making it hard for new users to start using them).

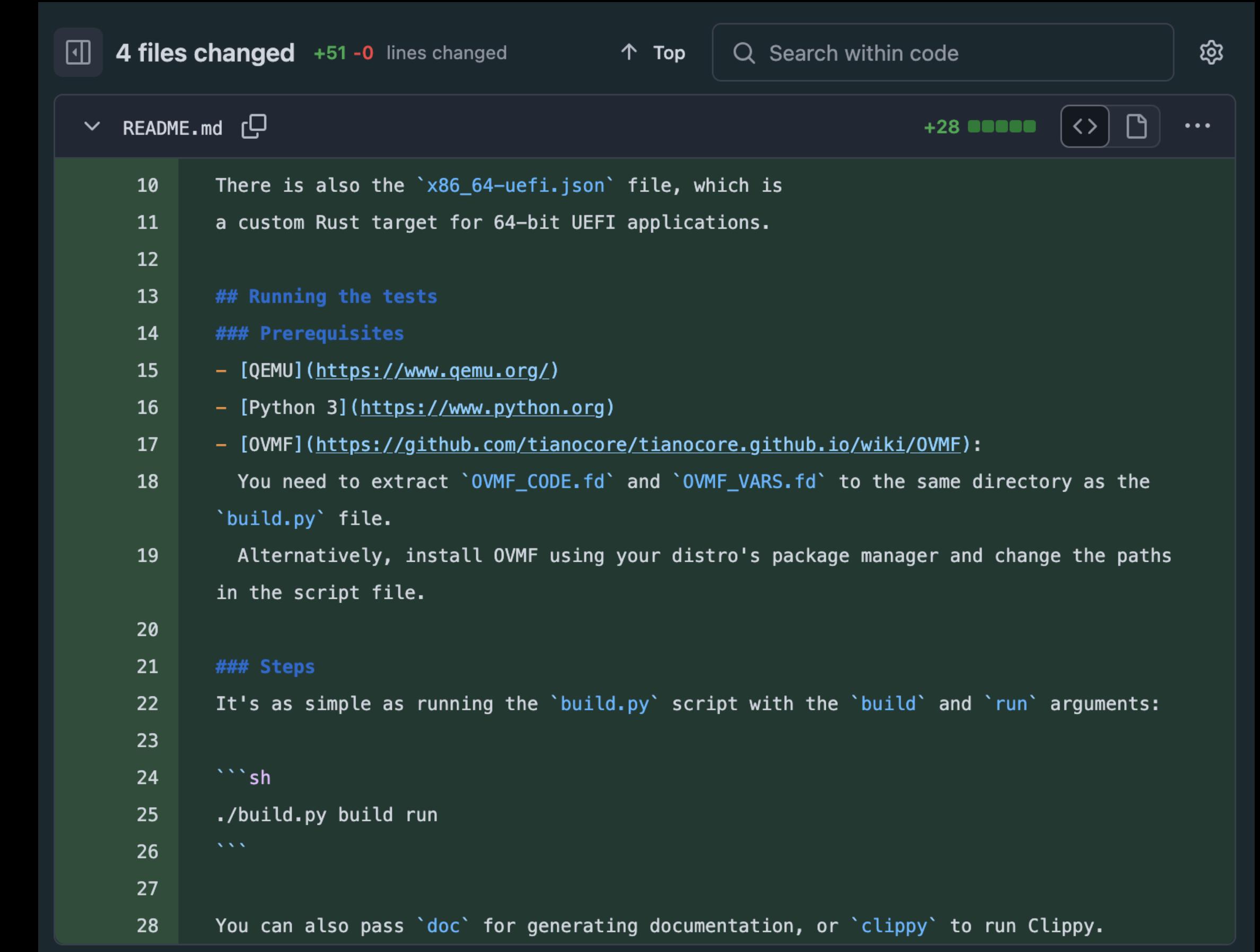


A screenshot of the crates.io search results page for 'efi'. The page shows three search results:

- efi v0.3.1**: Ergonomic Rust bindings for writing UEFI applications. Repository link. Metrics: All-Time: 8,061, Recent: 1,215, Updated: over 1 year ago.
- efi-runner v0.1.1**: A runner for EFI executables using QEMU and OVMF. Repository link. Metrics: All-Time: 2,184, Recent: 723, Updated: over 1 year ago.
- efi\_signer v0.2.8**: A crate for signing and parsing EFI image. Repository link. Metrics: All-Time: 12,481, Recent: 3,851, Updated: 2 months ago.

# uefi-rs vs. the rest

- From the very first commit, uefi-rs focused on providing **clear documentation** and **easy-to-understand example code**.
- The whole point of the library was to make it **easy** for anyone to get started with developing apps which can run on UEFI-based environments.

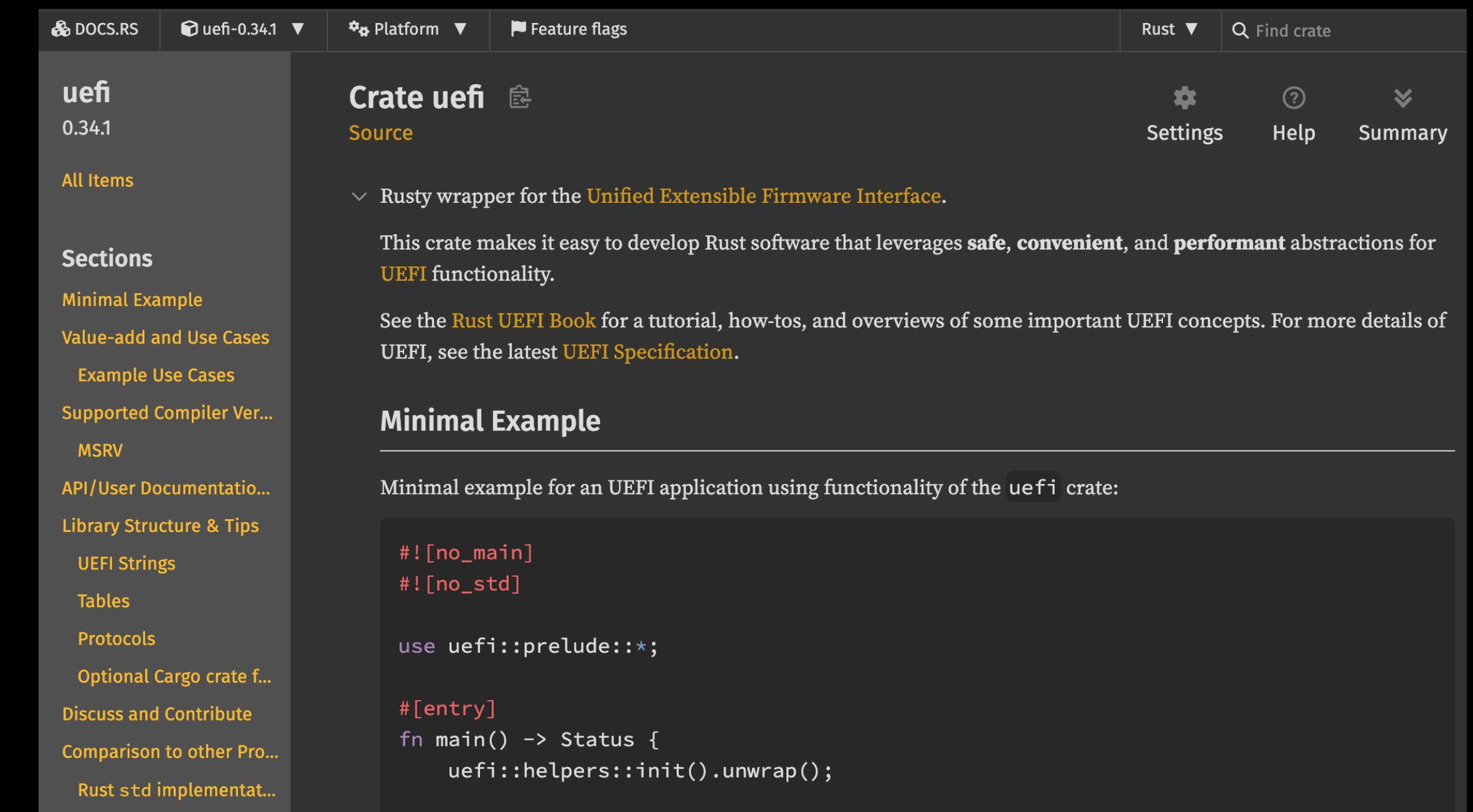


A screenshot of a GitHub pull request interface. The top bar shows "4 files changed +51 -0 lines changed". The main area displays the README.md file. The code content is as follows:

```
10 There is also the `x86_64-uefi.json` file, which is
11 a custom Rust target for 64-bit UEFI applications.
12
13 ## Running the tests
14 ### Prerequisites
15 - [QEMU] (https://www.qemu.org/)
16 - [Python 3] (https://www.python.org)
17 - [OVMF] (https://github.com/tianocore/tianocore.github.io/wiki/OVMF):
18     You need to extract `OVMF_CODE.fd` and `OVMF_VARS.fd` to the same directory as the
     `build.py` file.
19     Alternatively, install OVMF using your distro's package manager and change the paths
     in the script file.
20
21 ### Steps
22 It's as simple as running the `build.py` script with the `build` and `run` arguments:
23
24 ``sh
25 ./build.py build run
26 ```
27
28 You can also pass `doc` for generating documentation, or `clippy` to run Clippy.
```

# uefi-rs vs. the rest

- uefi-rs focuses on **simple** and **efficient abstractions**, making it easy to get started with writing boot loaders, drivers, kernels etc. without having to read through the UEFI specification beforehand.



The screenshot shows the crates.io website for the `uefi` crate. The top navigation bar includes links for DOCS.RS, uefi-0.34.1, Platform, Feature flags, Rust, and a search bar. The main content area is titled "Crate uefi" and contains sections for "Source" and "Minimal Example". The "Minimal Example" section provides a code snippet for a UEFI application:

```
#![no_main]
#![no_std]

use uefi::prelude::*;

#[entry]
fn main() -> Status {
    uefi::helpers::init().unwrap();
}
```

# Some general software engineering advice

- Designing the “perfect” interface takes time and many iterations

The screenshot shows a GitHub pull request page. The title is "Additional error data in Result + some API cleanup and clarification #78". The status is "Merged" into the "master" branch via a merge commit from "result-cleanup". The PR was opened on Jan 4, 2019, and merged on Jan 3, 2019. It has 20 conversations, 11 commits, 0 checks, and 30 files changed. The code review summary shows +460 additions and -337 deletions. The PR details section contains the following text:

This PR fixes #70 by adding support for additional error data to our Result type. As a result, usage of raw `core::result::Result` can now be restricted to functions which do not call into UEFI.

Since most functions do not return additional error data, the associated type parameter is optional with a default of `()`. After some thought, I think that the main output should probably get the same treatment, as a large amount of "setter" UEFI functions do not emit anything more than a status code.

While I went around the codebase to tweak every API entry point definition accordingly, I noticed a couple of issues here and there. Some entry points were in minor disagreement with the spec's semantics, while others did something wrong and dangerous. I'll clarify those changes as PR comments.

As I noticed some lifetime errors among these, I thought now might be a good time to use clearer lifetime parameter names in order to clarify the semantics of lifetime-based code. Result's generic parameters also got the same treatment.

HadrienG2 (Hadrien G.) added 9 commits 7 years ago

Reviewers – review now: GabrielMajeri (checked)

Assignees – assign yourself: (empty)

Labels: bug (unchecked), enhancement (checked)

Projects: (empty)

Milestone: (empty)

Development: (empty)

Customize: Notifications

# Some general software engineering advice

- Provide escape hatches for when people need custom behaviors

The screenshot shows the details page for the `uefi_raw` crate on crates.io. The page has a dark theme.

**uefi\_raw**  
0.10.0

Type 'S' or '/' to search, '?' for more options...

**Create uefi\_raw** [Edit](#)

**Source**

**Settings** **Help** **Summary**

Raw interface for working with UEFI.

This crate is intended for implementing UEFI services. It is also used for implementing the `uefi` crate, which provides a safe wrapper around UEFI.

For creating UEFI applications and drivers, consider using the `uefi` crate instead of `uefi-raw`.

**Modules**

<code>capsule</code>	UEFI update capsules.
<code>firmware_storage</code>	Types related to firmware storage.
<code>protocol</code>	Protocol definitions.
<code>table</code>	Standard UEFI tables.
<code>time</code>	Date and time types.

**Macros**

# Some general software engineering advice

- Upstream and reuse as much as possible

The screenshot shows a GitHub issue page for the "rust-lang/rust" repository. The issue is titled "Tracking issue for the "efiapi" calling convention #65815". The status is "Closed" (#105795). The issue was opened by roblabla on Oct 25, 2019, and edited by roblabla. The body of the issue contains text about the efiapi calling convention, its addition in PR #65809, and its feature gate name, abi\_efiapi. It also describes how it can be used for defining functions compatible with UEFI Interfaces. The "Usage" section shows a code snippet: `extern "efiapi" fn func() {...}`. The right sidebar shows the issue's labels: A-ABI, A-FFI, B-unstable, C-tracking-issue, O-U, S-tracking-ready-to-merge, disposition-merge, and finished-final-comment. There are also sections for Assignees (No one assigned) and Type.

Tracking issue for the "efiapi" calling convention #65815

Closed #105795

roblabla opened on Oct 25, 2019 · edited by roblabla

Tracking issue for the `efiapi` calling convention, added in PR #65809. The feature gate name is `abi_efiapi`.

The `efiapi` calling convention can be used for defining a function with an ABI compatible with the UEFI Interfaces as defined in the [UEFI Specification](#). On the currently supported platform, this means selecting between the `win64` ABI or the `C` ABI depending on the target architecture.

**Usage**

```
extern "efiapi" fn func() {...}
```

**Assignees**  
No one assigned

**Labels**

- A-ABI
- A-FFI
- B-unstable
- C-tracking-issue
- O-U
- S-tracking-ready-to-merge
- disposition-merge
- finished-final-comment

**Type**

# How did uefi-rs start?

# How I got here

It took a while

- Got interested in OS dev around 2014
- Started learning about and playing with writing my own operating system kernel from scratch in 2015, following the tutorials on OS Dev.org
- Figured out pretty quickly that I wasn't going to get anywhere building a whole OS on my own

# How I got here

## Step by step

- I thought I could at least put some code on GitHub, maybe others will find it useful for *something*
- Created uefi-cpp, the precursor to uefi-rs, in October 2016
- Started uefi-rs in November 2017, after getting too annoyed with C++ and learning some Rust

# Why Rust?

- If you've never worked with Rust before, you might be wondering why would anyone start a project in such a (young) programming language
- If you've worked (professionally) with C++ for enough time, you'll know it has plenty of issues (memory safety, concurrency, incomprehensible compiler errors, syntactic ambiguity, lack of a standard build/packaging system etc.)

## **Why Rust?**

### **Performance**

Rust is blazingly fast and memory-efficient: with no runtime or garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.

### **Reliability**

Rust's rich type system and ownership model guarantee memory-safety and thread-safety — enabling you to eliminate many classes of bugs at compile-time.

### **Productivity**

Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling — an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more.

# Why Rust?

## Even more reasons to switch to Rust

- You might also have heard that the Rust community is very active and very loyal to the language
- I'd argue there are good reasons for that :)

Technology → Admired and Desired

## **Programming, scripting, and markup languages**

JavaScript, Python and SQL are all highly-desired and admired programming languages, but Rust continues to be the most-admired programming language with an 83% score this year.

# Conclusions

- Building this library was **hard work**, but it was **fun** and I've **learned a lot of things** from it
- Open source is a lot about the **community**, and it's a great way to make friends with random people from around the world. On uefi-rs we've had contributors from: France, Germany, USA (New York specifically), Japan etc.

# Conclusions

- The fact that it became popular is also a lot thanks to **luck**. I never expected it to go anywhere when I first open sourced the repo!
- Expect most things that you finish and/or publish to be **inconsequential** and **nobody will care about them** (except for you, maybe)

**Thanks for listening!**  
Any questions?