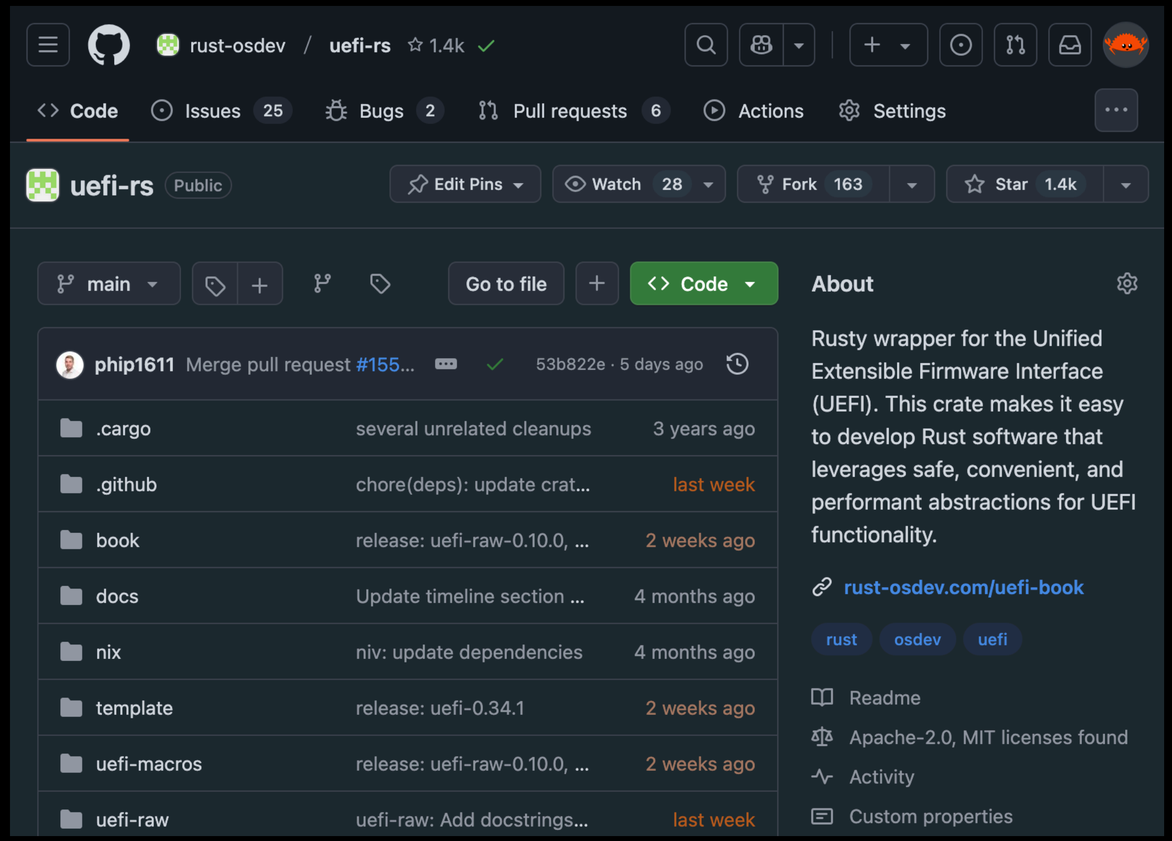
uefi-rs

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).

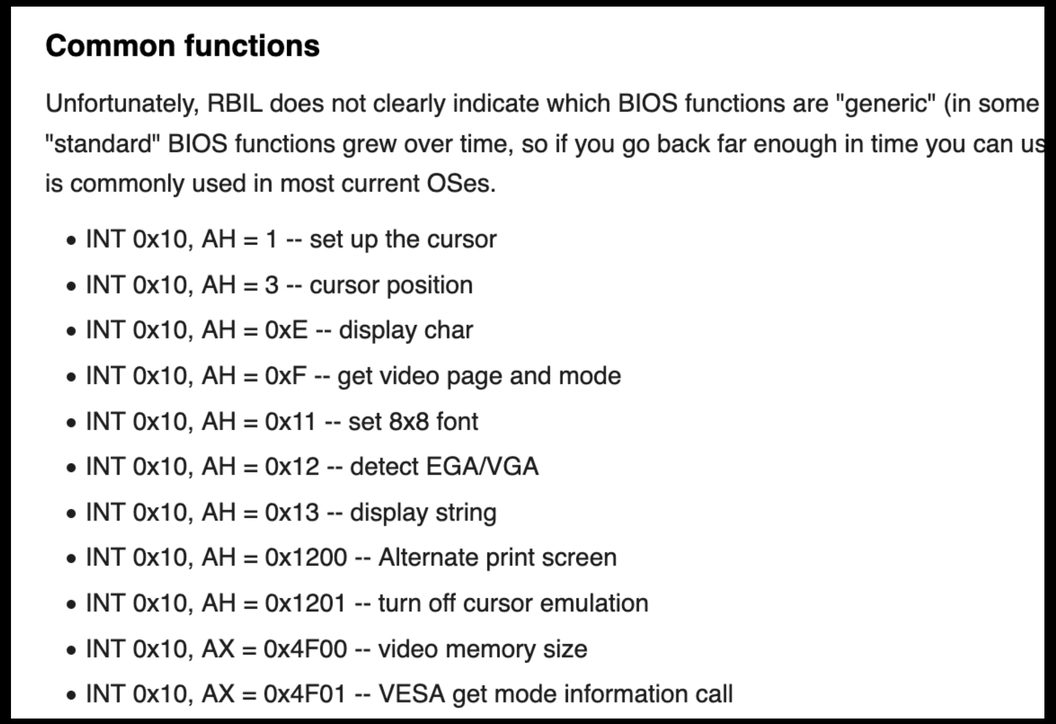
The Basic Input/Output System (BIOS)

- 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

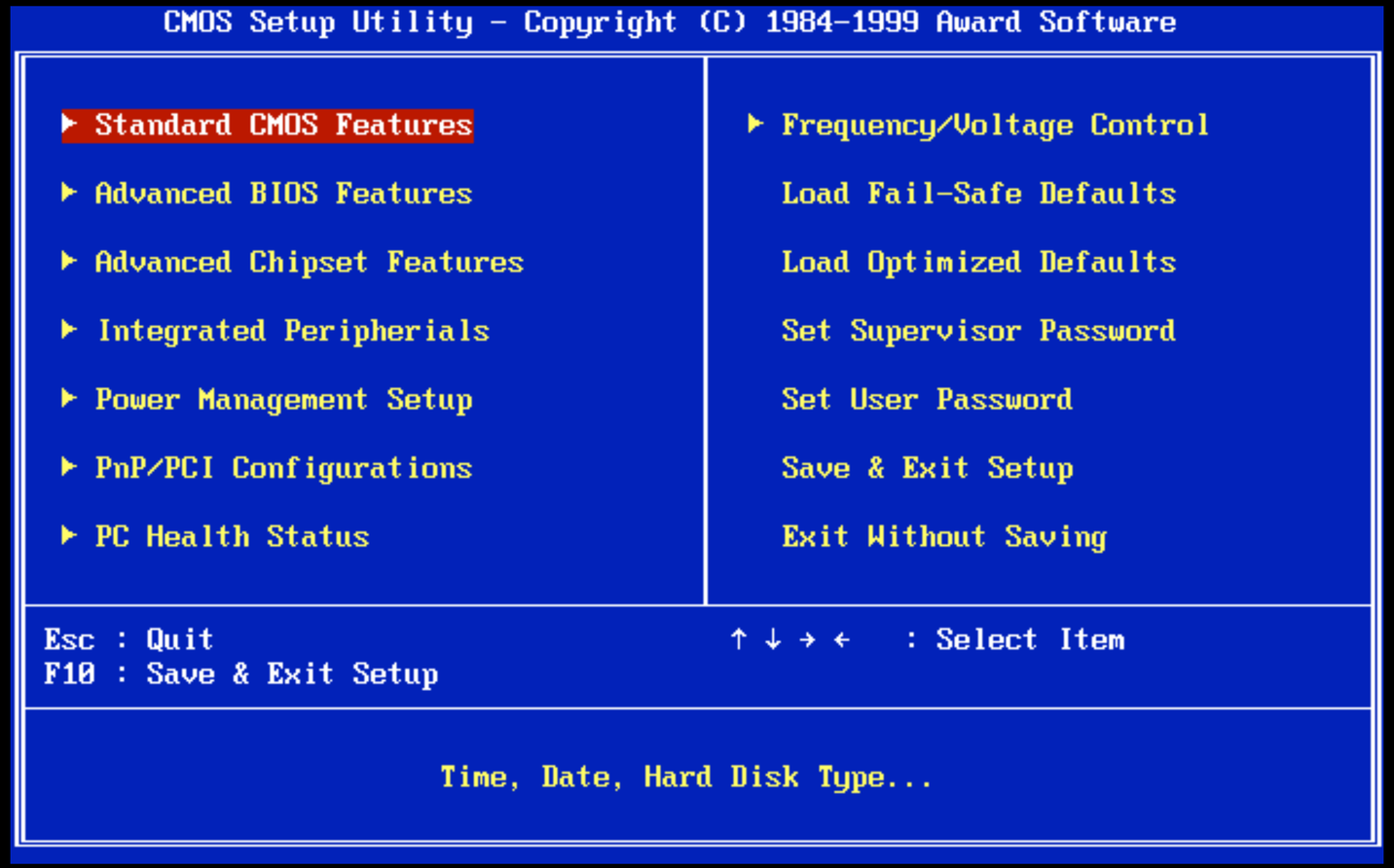
Unfortunately, RBIL does not clearly indicate which BIOS functions are "generic" (in some
"standard" BIOS functions grew over time, so if you go back far enough in time you can us
is commonly used in most current OSes.

- 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



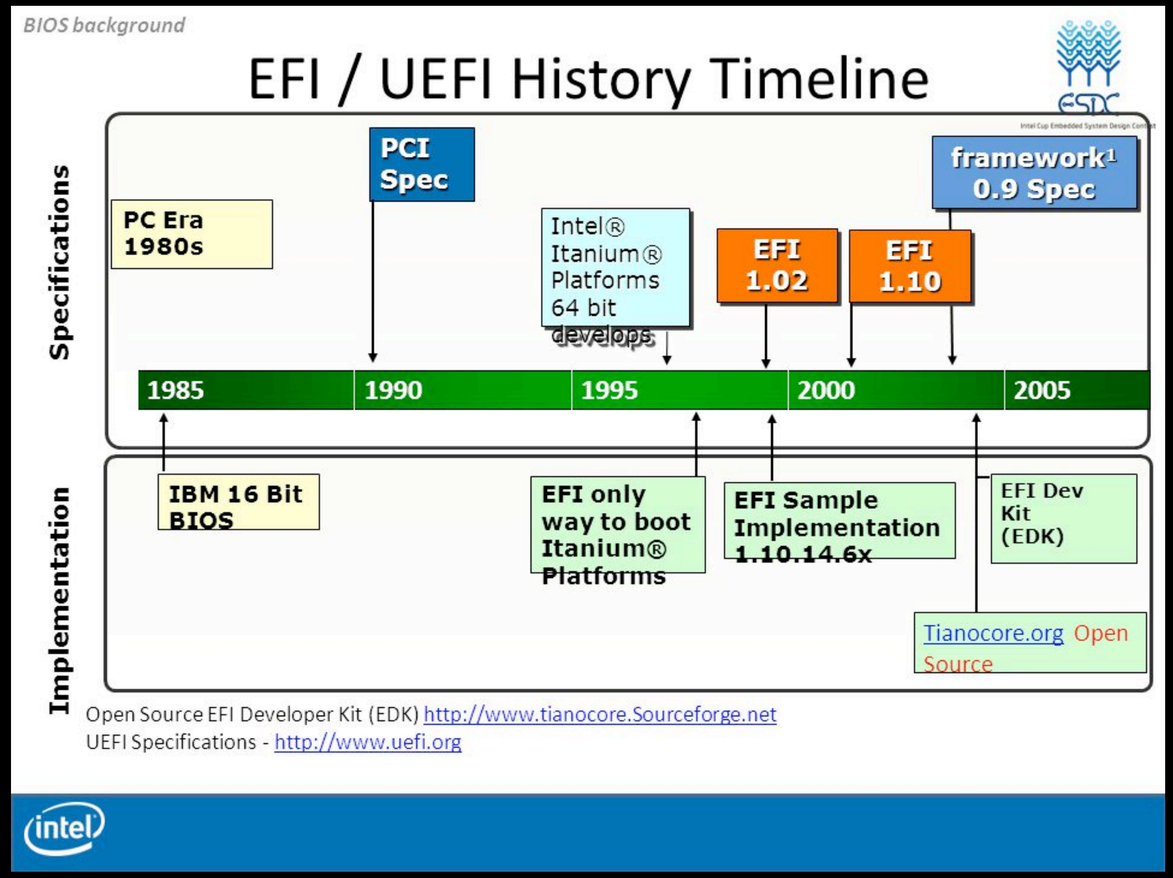
So what's the problem with the BIOS?

- 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.

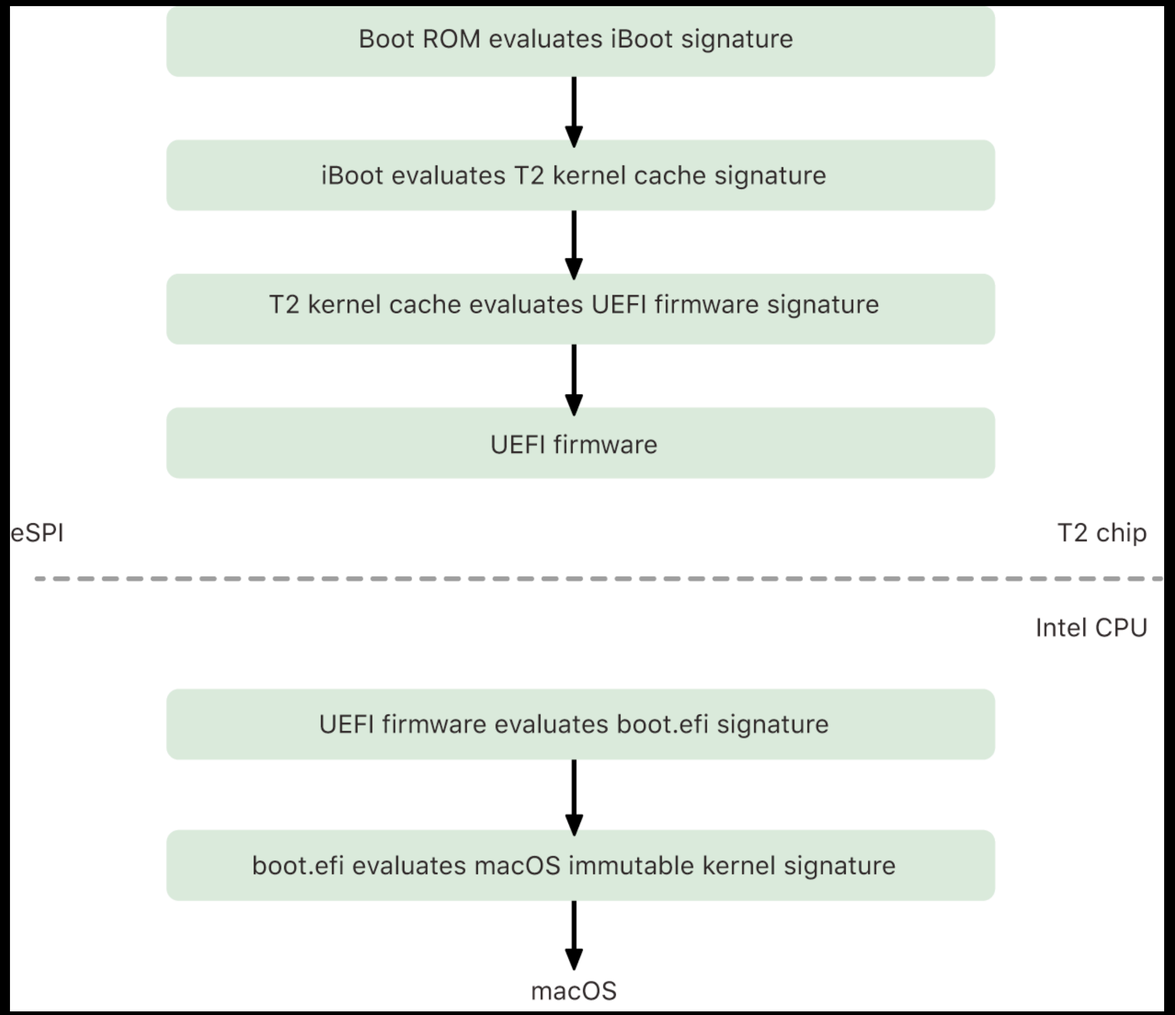


Unified EFI

- 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                 Hdr;
  CHAR16                           *FirmwareVendor;
  UINT32                           FirmwareRevision;
  EFI_HANDLE                       ConsoleInHandle;
  EFI_SIMPLE_TEXT_INPUT_PROTOCOL   *ConIn;
```

4.3. EFI System Table

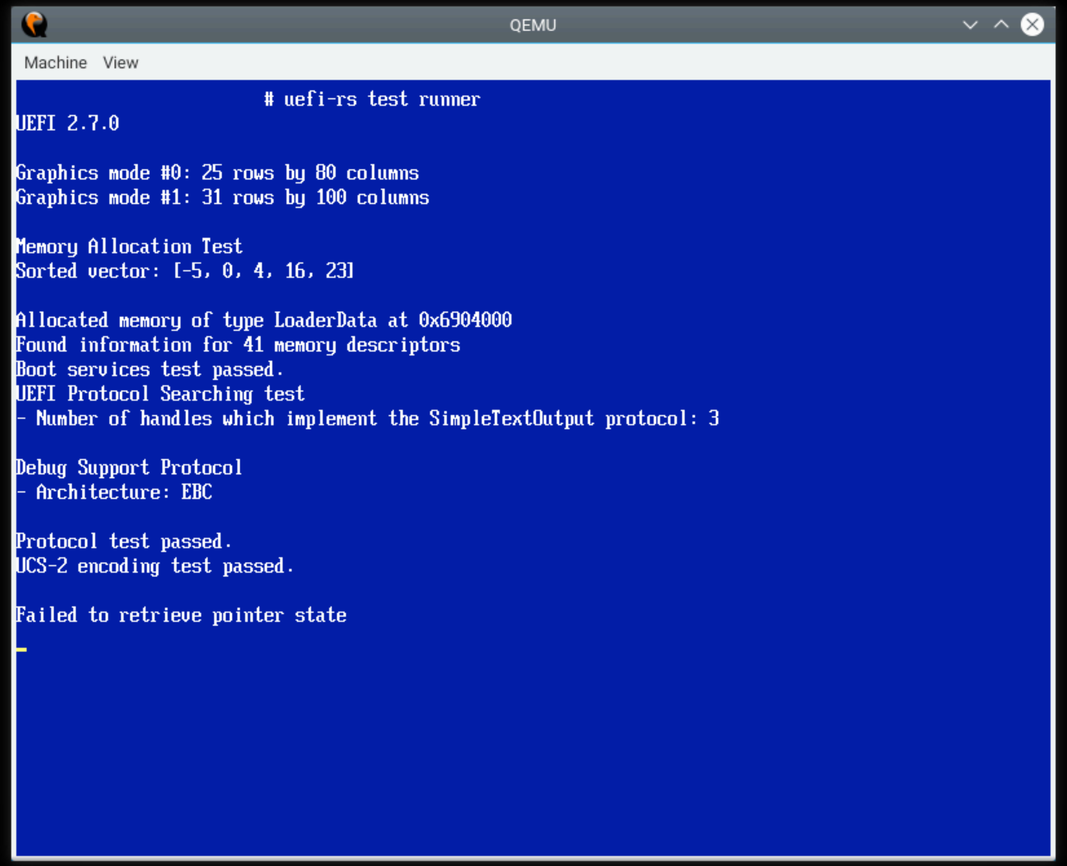
What is `uefi-rs`?

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

```rust

use std::os::uefi as uefi_std;
use uefi::runtime::ResetType;
use uefi::{Handle, Status};

/// Performs the necessary setup code for the `uefi` crate.
fn setup_uefi_crate() {
    let st = uefi_std::env::system_table();
    let ih = uefi_std::env::image_handle();

    // Mandatory setup code for `uefi` crate.
    unsafe {
        uefi::table::set_system_table(st.as_ptr().cast());

        let ih = Handle::from_ptr(ih.as_ptr().cast()).unwrap();
        uefi::boot::set_image_handle(ih);
    }
}

fn main() {
    println!("Hello World from uefi_std");
    setup_uefi_crate();
    println!("UEFI-Version is {}", uefi::system::uefi_revision());
    uefi::runtime::reset(ResetType::SHUTDOWN, Status::SUCCESS, None);
}
```



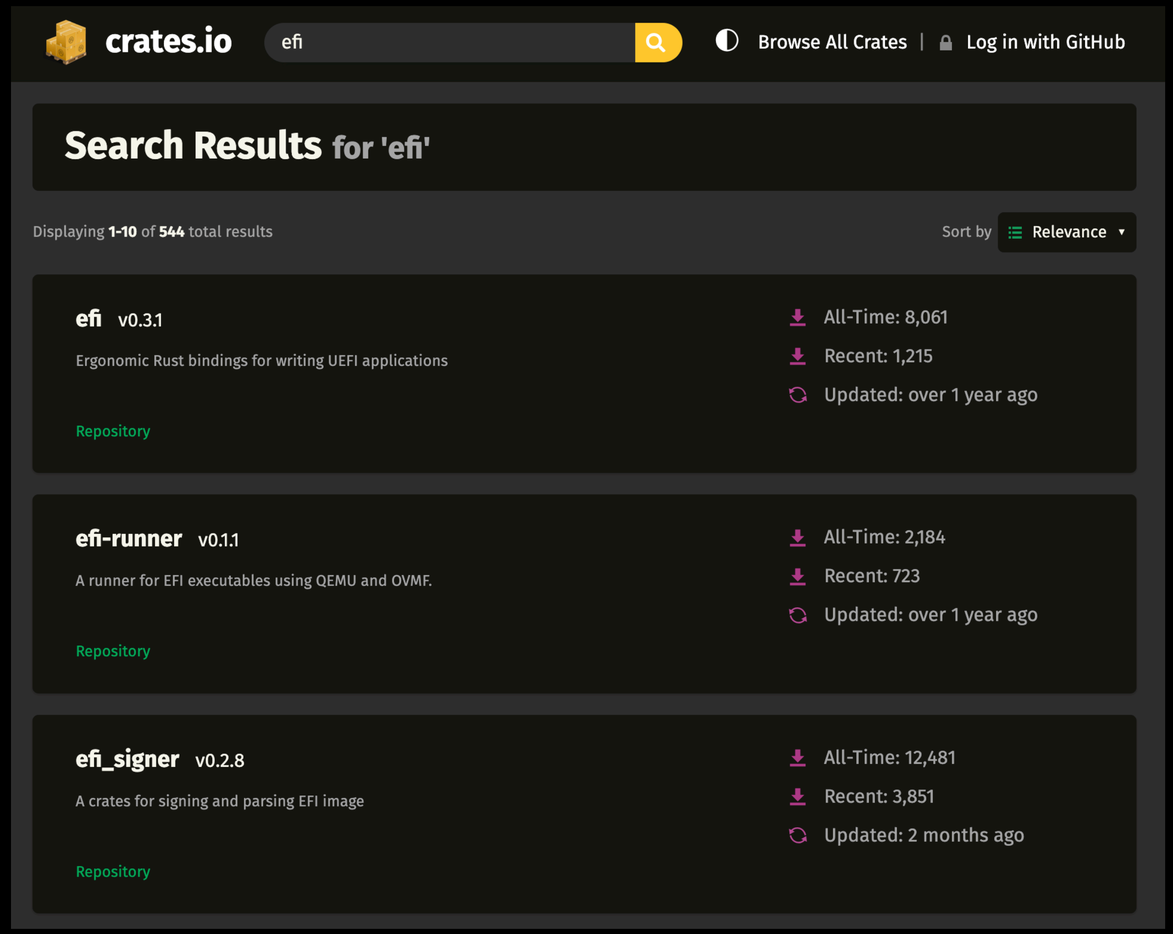
Features offered by `uefi-rs`

- 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`?

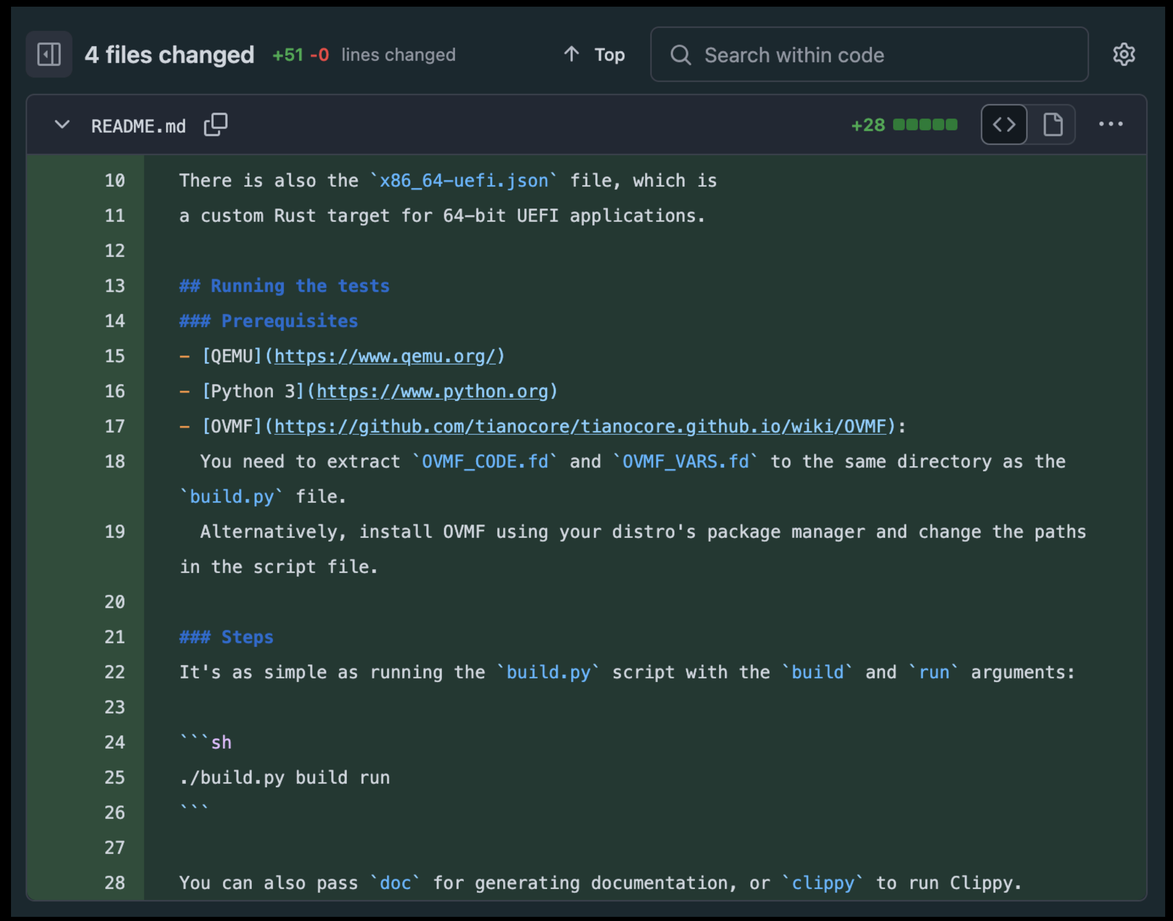
The Rust-UEFI ecosystem

- `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).



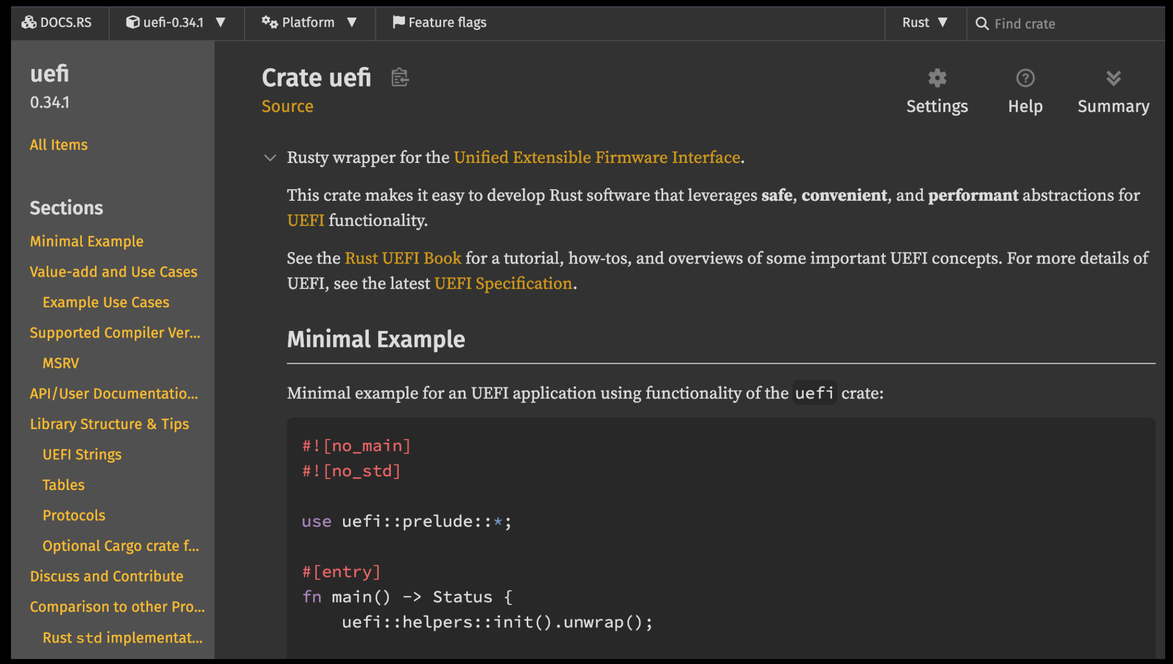
`uefi-rs` vs. the rest

- 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.



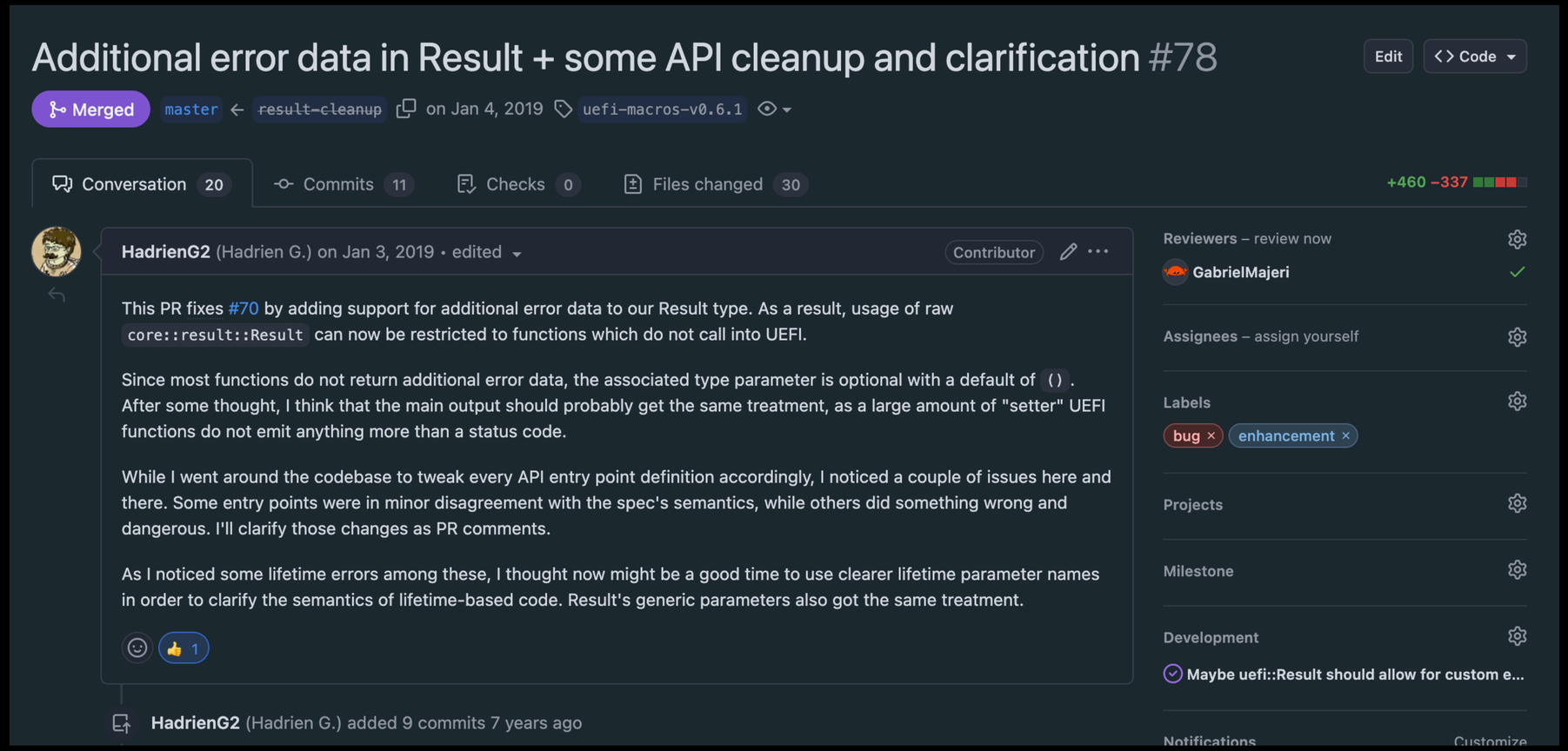
`uefi-rs` vs. the rest

- 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.



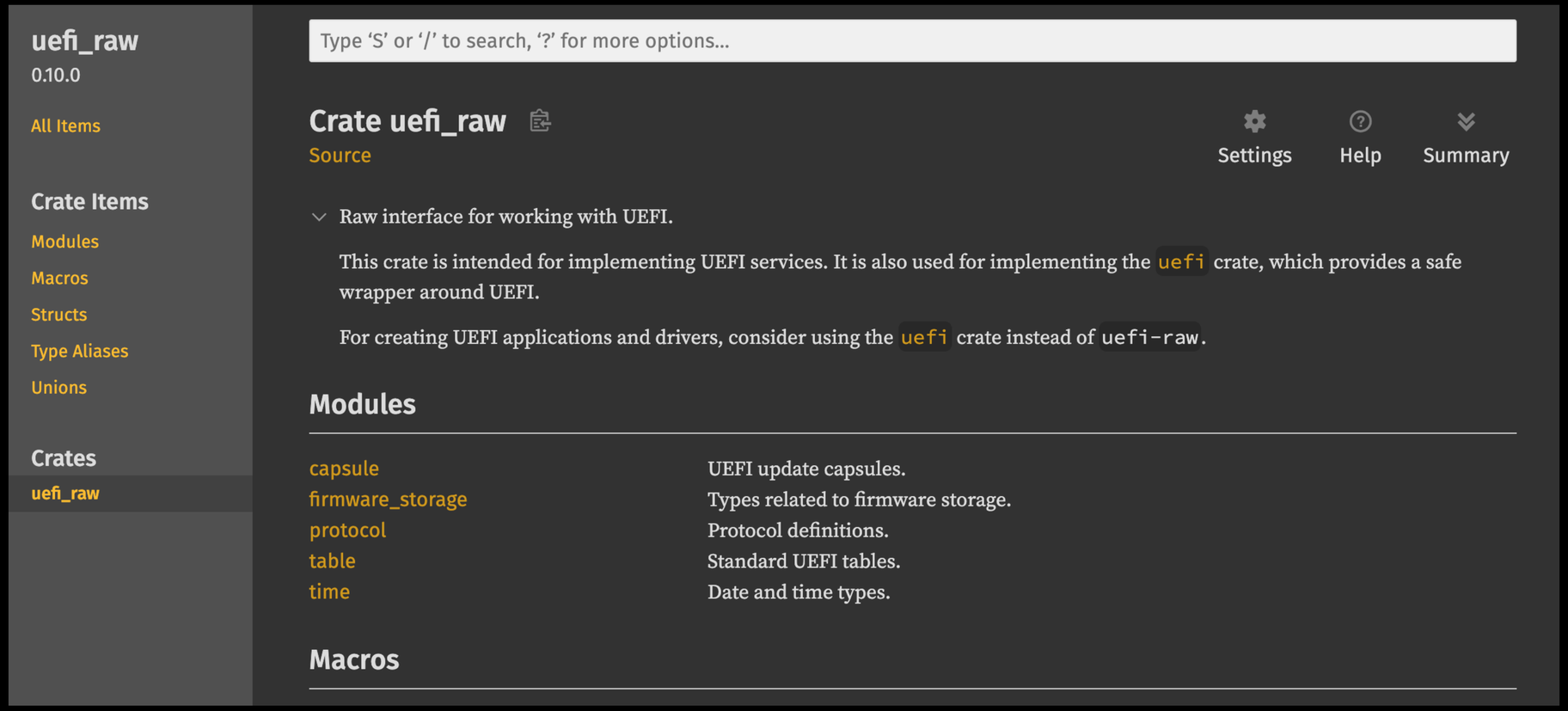
Some general software engineering advice

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



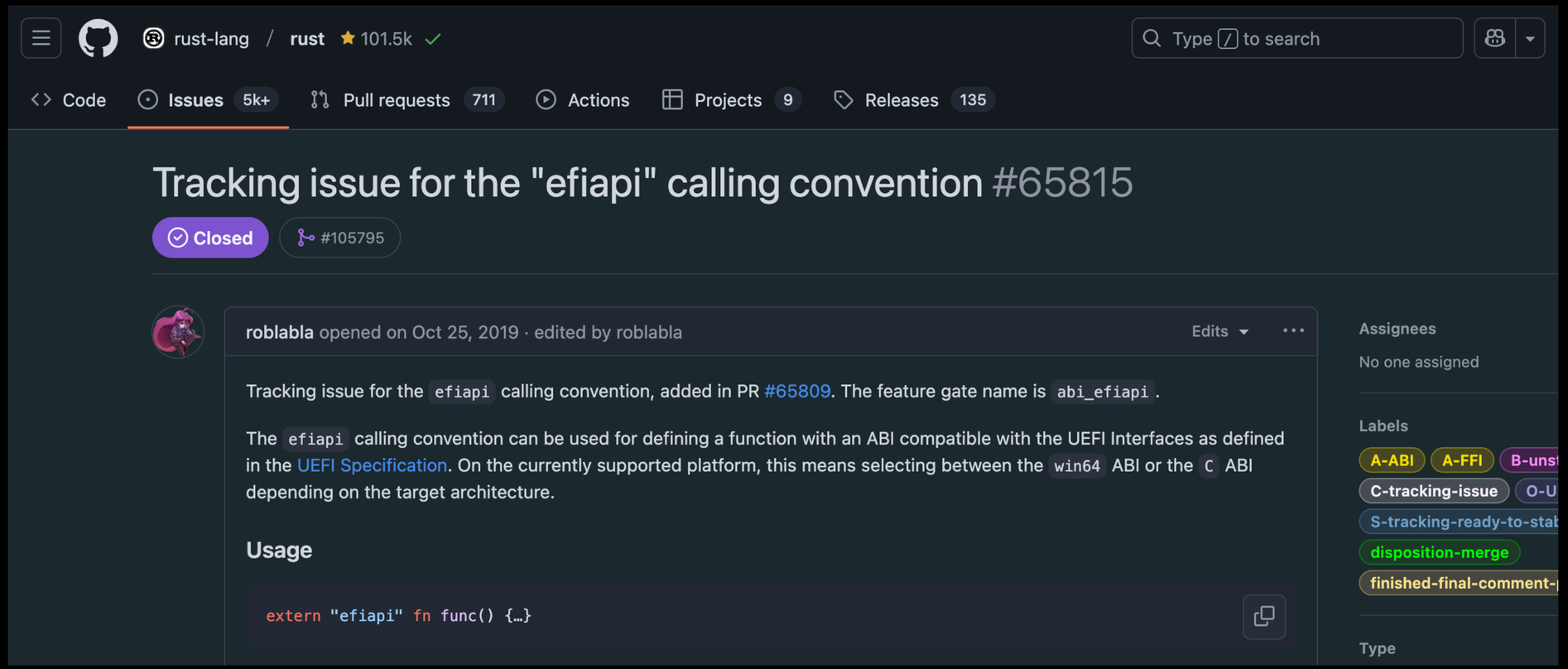
Some general software engineering advice

- Provide escape hatches for when people need custom behaviors



Some general software engineering advice

- Upstream and reuse as much as possible



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

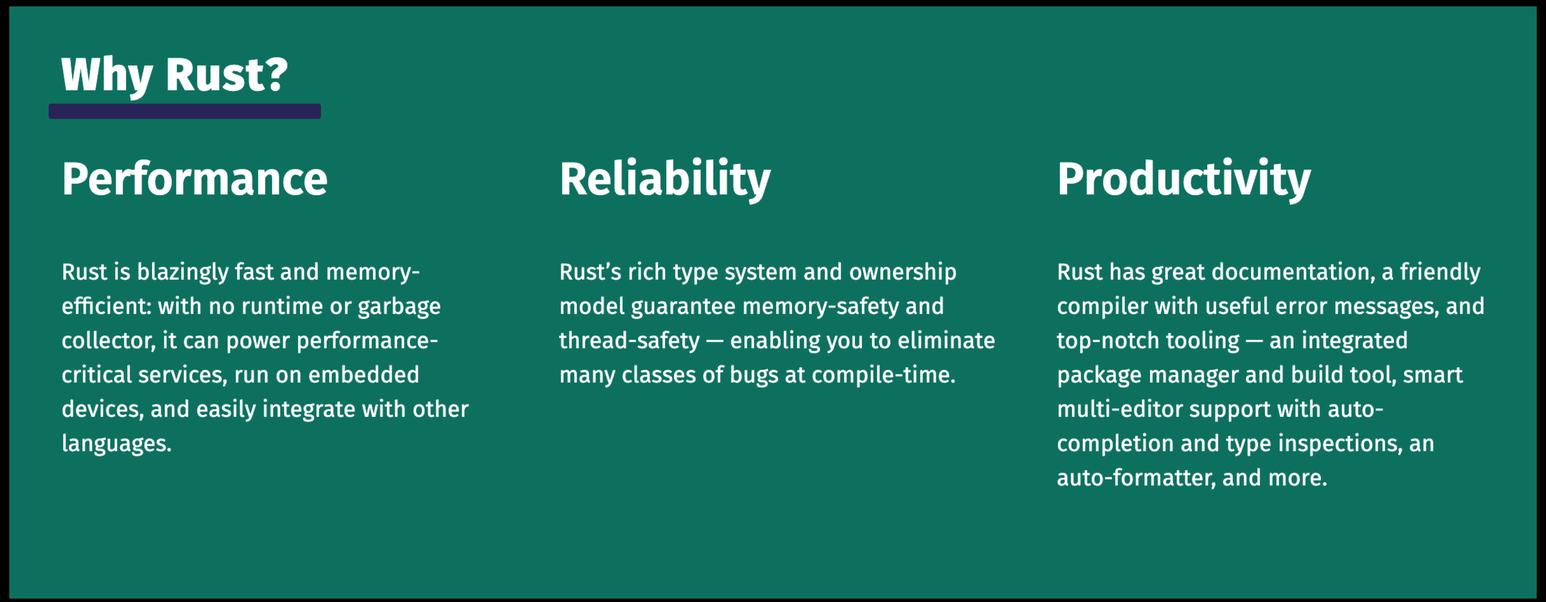
How I got here

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?

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 :)

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?