Writing an OS-Loader in Rust with uefi-rs

Philipp Schuster, Cyberus Technology

1. Introduction

1.0 Outlook

1.0 Outlook

- UEFI is a firmware interface making things easier

1.0 Outlook

- UEFI is a firmware interface making things easier
  - Hardware manufacturers

1.0 Outlook

- UEFI is a firmware interface making things easier
  - Hardware manufacturers
  - Firmware developers

1.0 Outlook

- UEFI is a firmware interface making things easier
  - Hardware manufacturers
  - Firmware developers
  - Software developers

1.0 Outlook

- UEFI is a firmware interface making things easier
  - Hardware manufacturers
  - Firmware developers
  - Software developers
- `uefi-rs` is a convenient library for UEFI in Rust

1.0 Outlook

- UEFI is a firmware interface making things easier
  - Hardware manufacturers
  - Firmware developers
  - Software developers
- `uefi-rs` is a convenient library for UEFI in Rust
- Convenient high-level* abstractions for OS-loaders / bootloaders

*High-level from a low-level perspective. Not Python- or Java-like high-level.

1.1 About Me

1.1 About Me

- Philipp Schuster, Dresden 🇩🇪🇪🇺

1.1 About Me

- Philipp Schuster, Dresden 🇩🇪🇪🇺
- Working at Cyberus Technology as Software Engineer

1.1 About Me

- Philipp Schuster, Dresden 🇩🇪🇪🇺
- Working at Cyberus Technology as Software Engineer
- Nix and NixOS enthusiast

1.1 About Me

- Philipp Schuster, Dresden 🇩🇪🇪🇺
- Working at Cyberus Technology as Software Engineer
- Nix and NixOS enthusiast
- Enjoy conferences and meetups

1.1 About Me

- Philipp Schuster, Dresden 🇩🇪🇪🇺
- Working at Cyberus Technology as Software Engineer
- Nix and NixOS enthusiast
- Enjoy conferences and meetups
- Organizing Systems Meetup

1.1 About Me

- Philipp Schuster, Dresden 🇩🇪🇪🇺
- Working at Cyberus Technology as Software Engineer
- Nix and NixOS enthusiast
- Enjoy conferences and meetups
- Organizing Systems Meetup
- GitHub @phip1611

1.1 About Me

- Philipp Schuster, Dresden 🇩🇪🇪🇺
- Working at Cyberus Technology as Software Engineer
- Nix and NixOS enthusiast
- Enjoy conferences and meetups
- Organizing Systems Meetup

- GitHub @phip1611
- Reddit @phip1611

1.1 About Me

- Philipp Schuster, Dresden 🇩🇪🇪🇺
- Working at Cyberus Technology as Software Engineer
- Nix and NixOS enthusiast
- Enjoy conferences and meetups
- Organizing Systems Meetup

- GitHub @phip1611
- Reddit @phip1611
- Blog phip1611.de

1.2 My Rust Experience

Working with Rust since 2019.

1.2 My Rust Experience

Working with Rust since 2019.

Hobby Projects

1.2 My Rust Experience

Working with Rust since 2019.

Hobby Projects

- github.com/rust-osdev

1.2 My Rust Experience

Working with Rust since 2019.

Hobby Projects

- github.com/rust-osdev
  - `multiboot2`

1.2 My Rust Experience

Working with Rust since 2019.

Hobby Projects

- github.com/rust-osdev
  - `multiboot2`
  - `uefi`

1.2 My Rust Experience

Working with Rust since 2019.

Hobby Projects

- github.com/rust-osdev
  - `multiboot2`
  - `uefi`
- Author of various smaller crates

1.2 My Rust Experience

Working with Rust since 2019.

Hobby Projects

- github.com/rust-osdev
  - `multiboot2`
  - `uefi`
- Author of various smaller crates

Work Projects

1.2 My Rust Experience

Working with Rust since 2019.

Hobby Projects

- github.com/rust-osdev
  - `multiboot2`
  - `uefi`
- Author of various smaller crates

Work Projects

- Cloud Hypervisor

1.2 My Rust Experience

Working with Rust since 2019.

Hobby Projects

- github.com/rust-osdev
  - `multiboot2`
  - `uefi`
- Author of various smaller crates

Work Projects

- Cloud Hypervisor
- `rust-vmm` ecosystem

1.3 My Relation to JUG Saxony e.V.

1.3 My Relation to JUG Saxony e.V.

- Started studying computer science in October 2015

1.3 My Relation to JUG Saxony e.V.

- Started studying computer science in October 2015
- Started student software engineer ("Werkstudent") position at Telekom MMS (T-Systems Multimedia Solutions GmbH)

1.3 My Relation to JUG Saxony e.V.

- Started studying computer science in October 2015
- Started student software engineer ("Werkstudent") position at Telekom MMS (T-Systems Multimedia Solutions GmbH)
- JUG Saxony Day

1.3 My Relation to JUG Saxony e.V.

- Started studying computer science in October 2015
- Started student software engineer ("Werkstudent") position at Telekom MMS (T-Systems Multimedia Solutions GmbH)
- JUG Saxony Day
  - 2018

1.3 My Relation to JUG Saxony e.V.

- Started studying computer science in October 2015
- Started student software engineer ("Werkstudent") position at Telekom MMS (T-Systems Multimedia Solutions GmbH)
- JUG Saxony Day
  - 2018
  - 2022 (Speaker)

1.3 My Relation to JUG Saxony e.V.

- Started studying computer science in October 2015
- Started student software engineer ("Werkstudent") position at Telekom MMS (T-Systems Multimedia Solutions GmbH)
- JUG Saxony Day
  - 2018
  - 2022 (Speaker)
  - 2025

1.3 My Relation to JUG Saxony e.V.

- Started studying computer science in October 2015
- Started student software engineer ("Werkstudent") position at Telekom MMS (T-Systems Multimedia Solutions GmbH)
- JUG Saxony Day
  - 2018
  - 2022 (Speaker)
  - 2025
- Personally met Falk Hartmann at EuroRust 2024 in Vienna

1.4 About Cyberus Technology

1.4 About Cyberus Technology

- Founded 2017 by 6 founders in Dresden (today ≈30)

1.4 About Cyberus Technology

- Founded 2017 by 6 founders in Dresden (today ≈30)
- Independent, profitable

1.4 About Cyberus Technology

- Founded 2017 by 6 founders in Dresden (today ≈30)
- Independent, profitable
- World-class expertise in x86 and virtualization

1.4 About Cyberus Technology

- Founded 2017 by 6 founders in Dresden (today ≈30)
- Independent, profitable
- World-class expertise in x86 and virtualization
- Main products

1.4 About Cyberus Technology

- Founded 2017 by 6 founders in Dresden (today ≈30)
- Independent, profitable
- World-class expertise in x86 and virtualization
- Main products
  - Cyberus Hypervisor (Cloud Hypervisor + Linux/KVM + Service & Expertise)

1.4 About Cyberus Technology

- Founded 2017 by 6 founders in Dresden (today ≈30)
- Independent, profitable
- World-class expertise in x86 and virtualization
- Main products
  - Cyberus Hypervisor (Cloud Hypervisor + Linux/KVM + Service & Expertise)
  - CTRL-OS (NixOS LTS + Embedded System Building Blocks)

1.4 About Cyberus Technology

1.4 About Cyberus Technology

- Cloud department

1.4 About Cyberus Technology

- Cloud department
  - Public European cloud developed with SAP ("Apeiro")

    Cyberus is responsible for virtualization layer ← My work

1.4 About Cyberus Technology

- Cloud department
  - Public European cloud developed with SAP ("Apeiro")

    Cyberus is responsible for virtualization layer ← My work
  - Soon BSI*-accredited virtualization stack with open-source software
    - Cloud Hypervisor/KVM will be accredited

* BSI: Bundesamt für Sicherheit in der Informationstechnik

1.4 My Role at Cyberus Technology

1.4 My Role at Cyberus Technology

- 2021-05 – 2022-05: Student software engineer

  Diplomarbeit (Master's Thesis)

1.4 My Role at Cyberus Technology

- 2021-05 – 2022-05: Student software engineer

  Diplomarbeit (Master's Thesis)
- 2022-06 – present: Full time software engineer

1.4 My Role at Cyberus Technology

- 2021-05 – 2022-05: Student software engineer

  Diplomarbeit (Master's Thesis)
- 2022-06 – present: Full time software engineer
- Everything "low in the stack"

1.4 My Role at Cyberus Technology

- 2021-05 – 2022-05: Student software engineer

  Diplomarbeit (Master's Thesis)
- 2022-06 – present: Full time software engineer
- Everything "low in the stack"
  - Rust, C/C++, Assembly, …

1.4 My Role at Cyberus Technology

- 2021-05 – 2022-05: Student software engineer

  Diplomarbeit (Master's Thesis)
- 2022-06 – present: Full time software engineer
- Everything "low in the stack"
  - Rust, C/C++, Assembly, …
  - libvirt, Linux kernel, GRUB, UEFI/edk2…

1.4 My Role at Cyberus Technology

- 2021-05 – 2022-05: Student software engineer

  Diplomarbeit (Master's Thesis)
- 2022-06 – present: Full time software engineer
- Everything "low in the stack"
  - Rust, C/C++, Assembly, …
  - libvirt, Linux kernel, GRUB, UEFI/edk2…
- Nix, NixOS, and nixpkgs

1.4 My Role at Cyberus Technology

- 2021-05 – 2022-05: Student software engineer

  Diplomarbeit (Master's Thesis)
- 2022-06 – present: Full time software engineer
- Everything "low in the stack"
  - Rust, C/C++, Assembly, …
  - libvirt, Linux kernel, GRUB, UEFI/edk2…
- Nix, NixOS, and nixpkgs
- Conferences, Networking, Meetups

  Organizing *Dresden Systems Meetup*

1.4 My Role at Cyberus Technology

1.4 My Role at Cyberus Technology

- Main contributor to Cloud Hypervisor

1.4 My Role at Cyberus Technology

- Main contributor to Cloud Hypervisor
  - Virtual Machine Monitor (VMM) utilizing Linux/KVM

1.4 My Role at Cyberus Technology

- Main contributor to Cloud Hypervisor
  - Virtual Machine Monitor (VMM) utilizing Linux/KVM
  - Akin to VirtualBox or QEMU

1.4 My Role at Cyberus Technology

- Main contributor to Cloud Hypervisor
  - Virtual Machine Monitor (VMM) utilizing Linux/KVM
  - Akin to VirtualBox or QEMU
  - Tailored to cloud usecase

1.4 My Role at Cyberus Technology

- Main contributor to Cloud Hypervisor
  - Virtual Machine Monitor (VMM) utilizing Linux/KVM
  - Akin to VirtualBox or QEMU
  - Tailored to cloud usecase
- **Virtualization requires understanding every concept of the platform and typical software stack.**

1.5 What (Not) To Expect

1.5 What (Not) To Expect

- We have time (60 min) → no rush

1.5 What (Not) To Expect

- We have time (60 min) → no rush
- Overview of how an x86 computer works

  → Give you a good understanding of the x86 platform.

1.5 What (Not) To Expect

- We have time (60 min) → no rush
- Overview of how an x86 computer works

  → Give you a good understanding of the x86 platform.
- What is Firmware?

1.5 What (Not) To Expect

- We have time (60 min) → no rush
- Overview of how an x86 computer works

  → Give you a good understanding of the x86 platform.
- What is Firmware?
- UEFI: Context + Concepts

1.5 What (Not) To Expect

- We have time (60 min) → no rush
- Overview of how an x86 computer works

  → Give you a good understanding of the x86 platform.
- What is Firmware?
- UEFI: Context + Concepts
- Rust library ("crate") `uefi-rs`

1.5 What (Not) To Expect

- We have time (60 min) → no rush
- Overview of how an x86 computer works

  → Give you a good understanding of the x86 platform.
- What is Firmware?
- UEFI: Context + Concepts
- Rust library ("crate") `uefi-rs`
- Code & Demo: Example UEFI OS-loader

* Same thing, different name: OS-loader, OS-specific loader, bootloader

1.6 Goal of an OS Project

Why does one need to understand the firmware?

1.6 Goal of an OS Project

Why does one need to understand the firmware?

- Fully bootstrapped system (Desktop environment, sound, ...)

1.6 Goal of an OS Project

Why does one need to understand the firmware?

- Fully bootstrapped system (Desktop environment, sound, ...)
- Kernel running in 64-bit mode

1.6 Goal of an OS Project

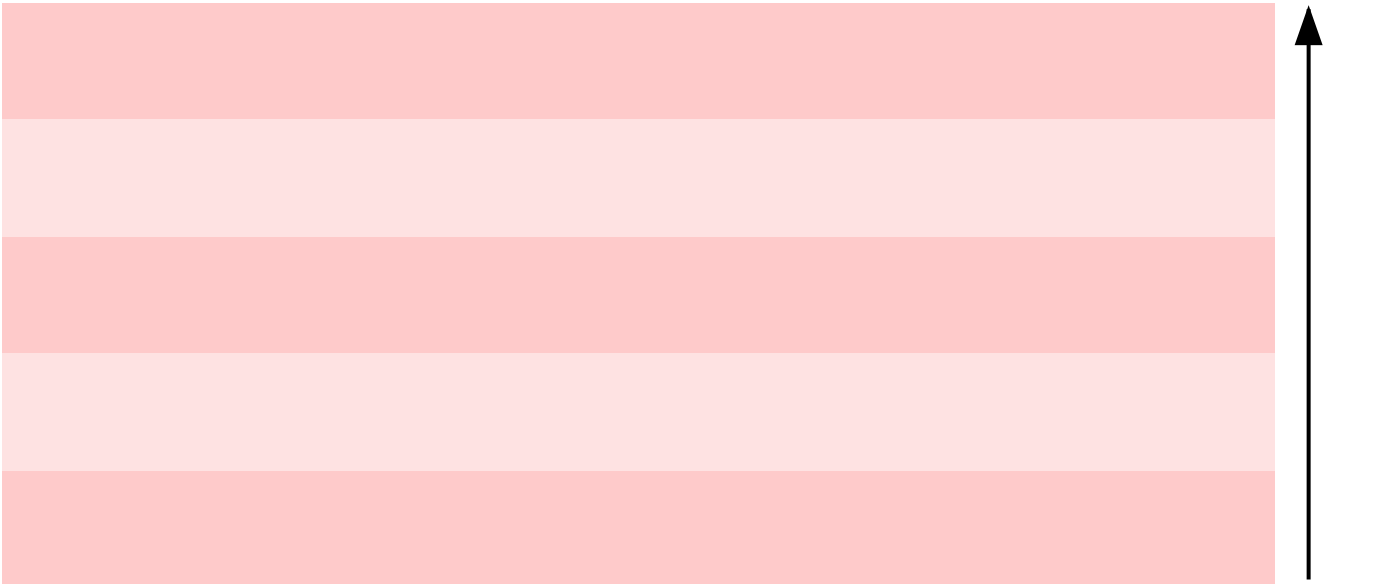
Why does one need to understand the firmware?

- Fully bootstrapped system (Desktop environment, sound, ...)
- Kernel running in 64-bit mode
- Firmware (UEFI) eventually leads to our kernel being loaded

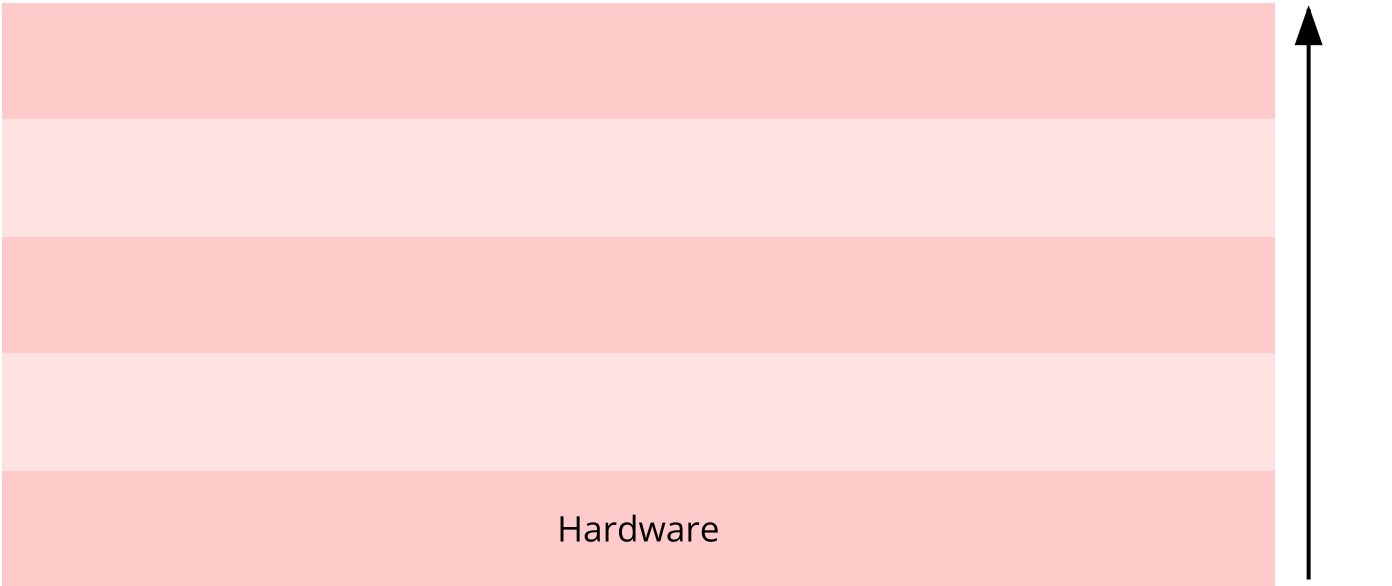
  → We need to understand UEFI

2. Background

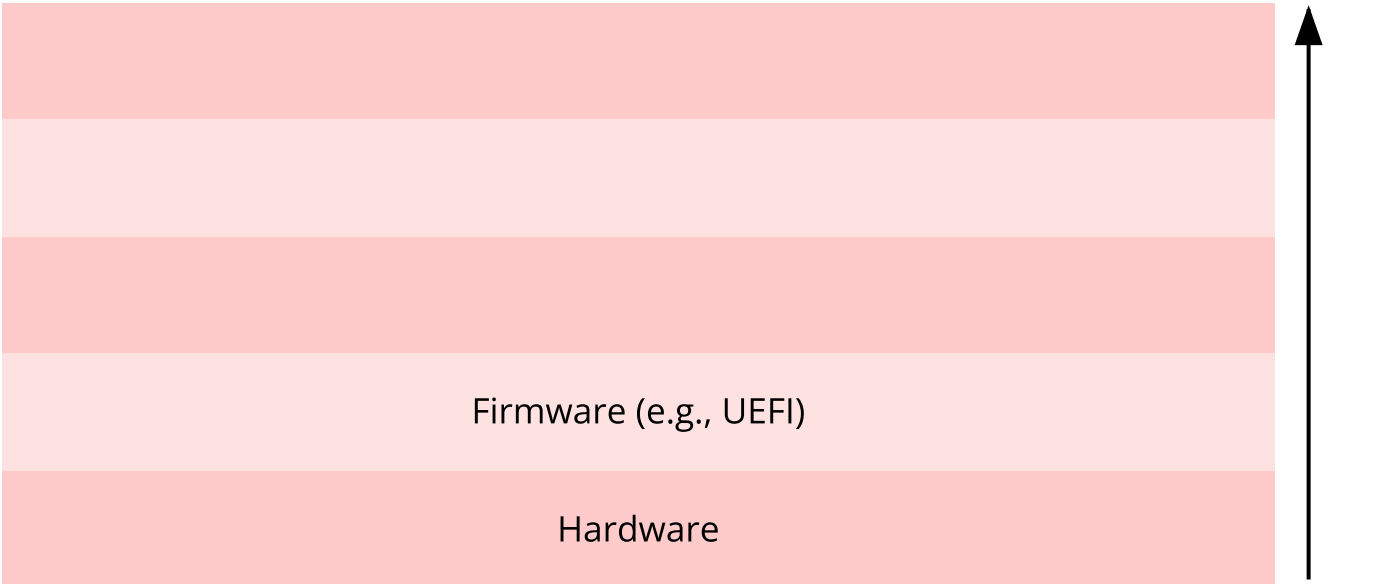
2.1 How does a Computer Boot?



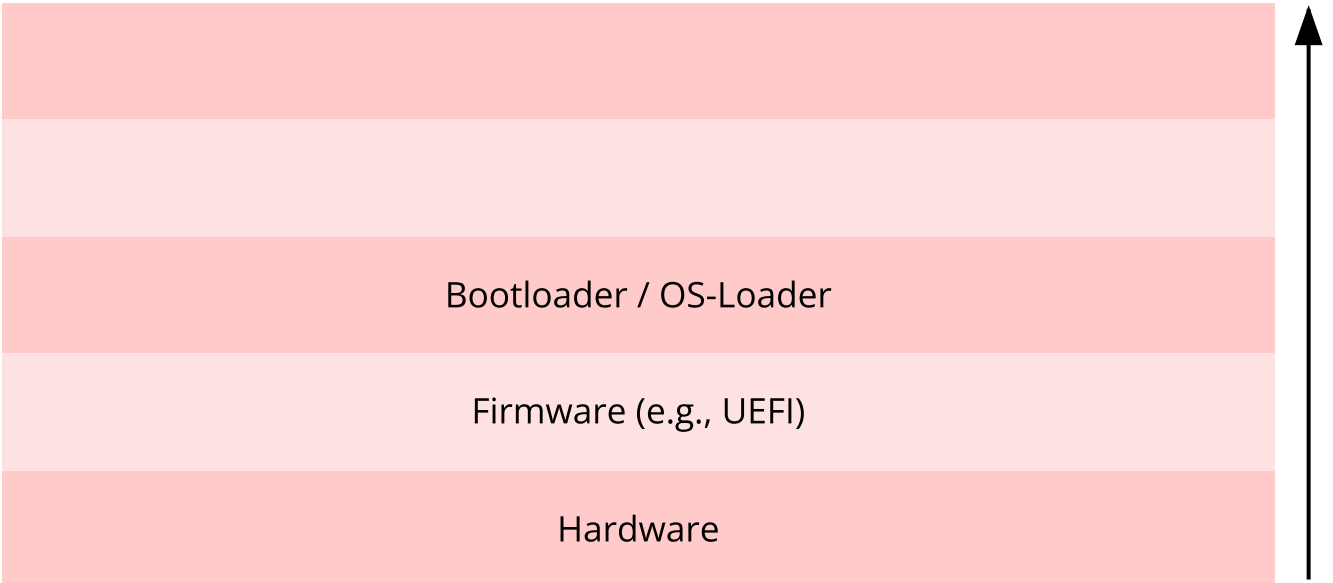
2.1 How does a Computer Boot?



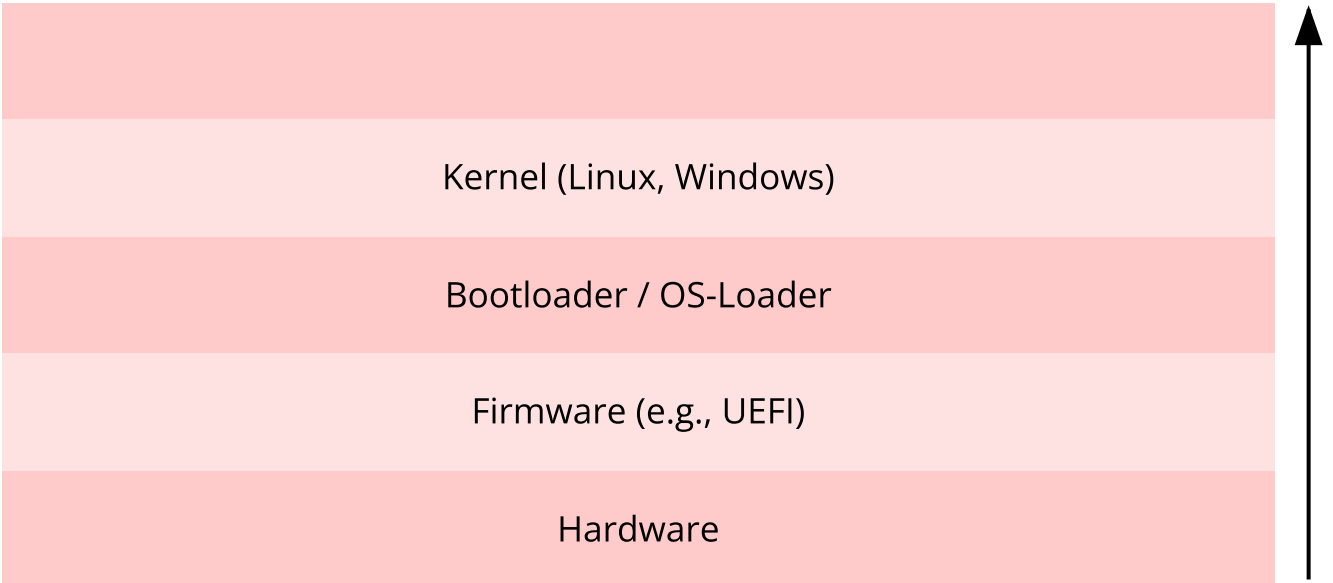
2.1 How does a Computer Boot?



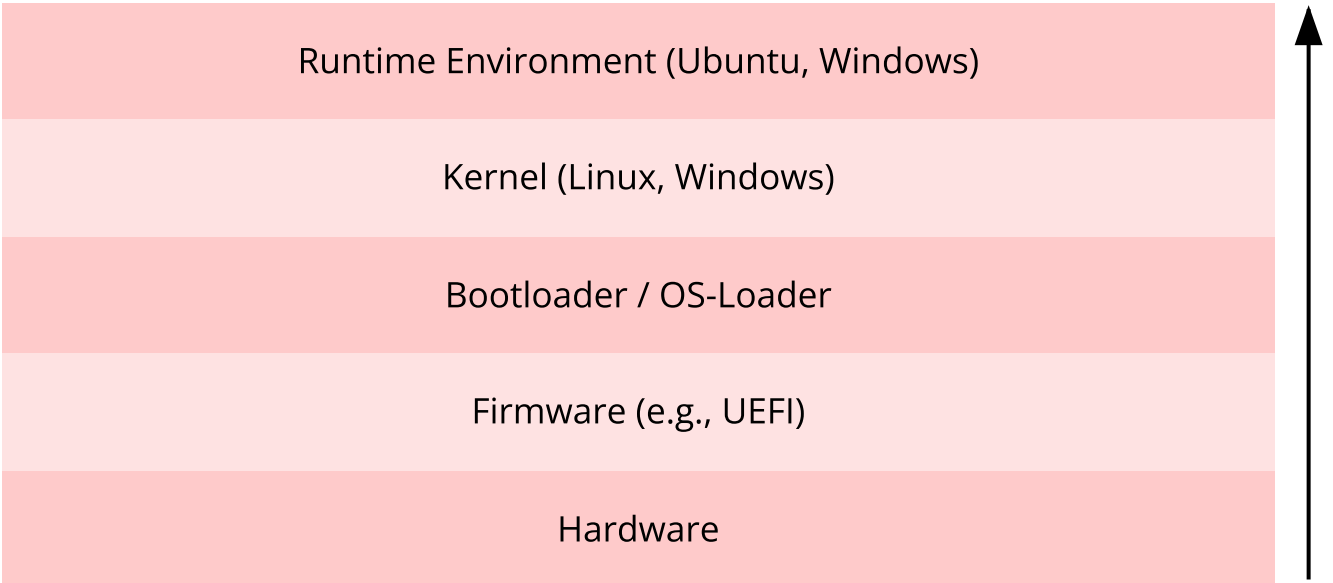
2.1 How does a Computer Boot?



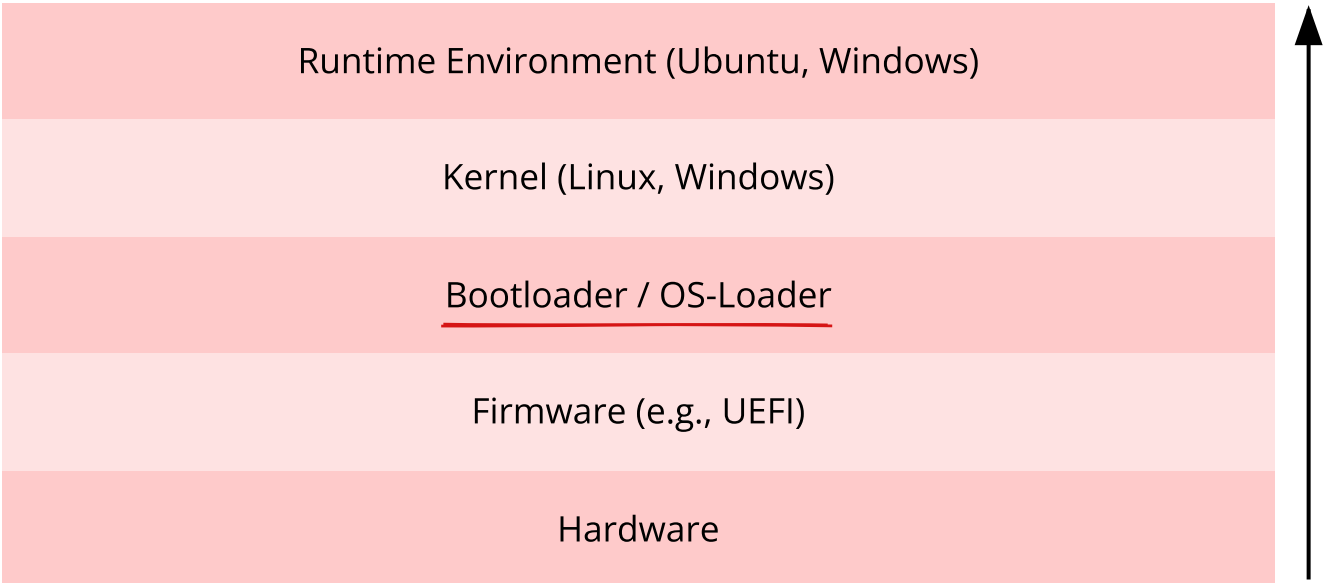
2.1 How does a Computer Boot?



2.1 How does a Computer Boot?



2.1 How does a Computer Boot?



2.2 CPU Terminology (x86)

2.2 CPU Terminology (x86)

- **CPU**: Central Processing Unit, computing resource:

  Everyday language: refers to whole package or computing resource

2.2 CPU Terminology (x86)

- **CPU**: Central Processing Unit, computing resource:

  Everyday language: refers to whole package or computing resource
- **Package/Socket/Processor***: The thing mounted onto the mainboard

* Inconsistencies even in Intel Manual (grown historically)

2.2 CPU Terminology (x86)

- **CPU**: Central Processing Unit, computing resource:

  Everyday language: refers to whole package or computing resource
- **Package/Socket/Processor***: The thing mounted onto the mainboard
- **Die**: Holds cores, caches, and additional logic (I/O, L3 cache)

* Inconsistencies even in Intel Manual (grown historically)

2.2 CPU Terminology (x86)

- **CPU**: Central Processing Unit, computing resource:

  Everyday language: refers to whole package or computing resource
- **Package/Socket/Processor***: The thing mounted onto the mainboard
- **Die**: Holds cores, caches, and additional logic (I/O, L3 cache)
- **Core**: Independent execution engine (L1, L2 caches)

* Inconsistencies even in Intel Manual (grown historically)

2.2 CPU Terminology (x86)

- **CPU**: Central Processing Unit, computing resource:

  Everyday language: refers to whole package or computing resource
- **Package/Socket/Processor***: The thing mounted onto the mainboard
- **Die**: Holds cores, caches, and additional logic (I/O, L3 cache)
- **Core**: Independent execution engine (L1, L2 caches)
- **(Logical) CPU**:

* Inconsistencies even in Intel Manual (grown historically)

2.2 CPU Terminology (x86)

- **CPU**: Central Processing Unit, computing resource:

  Everyday language: refers to whole package or computing resource
- **Package/Socket/Processor***: The thing mounted onto the mainboard
- **Die**: Holds cores, caches, and additional logic (I/O, L3 cache)
- **Core**: Independent execution engine (L1, L2 caches)
- **(Logical) CPU**:
  - Software-visible computing resource within a core

* Inconsistencies even in Intel Manual (grown historically)

2.2 CPU Terminology (x86)

- **CPU**: Central Processing Unit, computing resource:

  Everyday language: refers to whole package or computing resource
- **Package/Socket/Processor***: The thing mounted onto the mainboard
- **Die**: Holds cores, caches, and additional logic (I/O, L3 cache)
- **Core**: Independent execution engine (L1, L2 caches)
- **(Logical) CPU**:
  - Software-visible computing resource within a core
  - Implements the instruction set ("API of the CPU")

* Inconsistencies even in Intel Manual (grown historically)

2.2 CPU Terminology (x86)

- **CPU**: Central Processing Unit, computing resource:

  Everyday language: refers to whole package or computing resource
- **Package/Socket/Processor***: The thing mounted onto the mainboard
- **Die**: Holds cores, caches, and additional logic (I/O, L3 cache)
- **Core**: Independent execution engine (L1, L2 caches)
- **(Logical) CPU**:
  - Software-visible computing resource within a core
  - Implements the instruction set ("API of the CPU")
- Often fluid transitions and overlaps (architecture, manufacturer, platform)

* Inconsistencies even in Intel Manual (grown historically)

2.3 The many modes of an x86 CPU

2.3 The many modes of an x86 CPU

- 16-bit ("real mode")

2.3 The many modes of an x86 CPU

- 16-bit ("real mode")
- 32-bit protected mode, without paging

2.3 The many modes of an x86 CPU

- 16-bit ("real mode")
- 32-bit protected mode, without paging
- 32-bit protected mode, with paging

2.3 The many modes of an x86 CPU

- 16-bit ("real mode")
- 32-bit protected mode, without paging
- 32-bit protected mode, with paging
- 64-bit with 32-bit opcodes ("compatibility IA-32e mode")

  → Allows 32-bit software in an 64-bit operating system

2.3 The many modes of an x86 CPU

- 16-bit ("real mode")
- 32-bit protected mode, without paging
- 32-bit protected mode, with paging
- 64-bit with 32-bit opcodes ("compatibility IA-32e mode")

  → Allows 32-bit software in an 64-bit operating system
- **64-bit mode** ("64-bit IA-32e mode"$_{\text{Intel}}$, "long mode"$_{\text{AMD}}$)

2.4 Overview of an x86 Computer

2.4 Overview of an x86 Computer

- Platform/SoC: Processor + Chipset

2.4 Overview of an x86 Computer

- Platform/SoC: Processor + Chipset
- Mainboard: Processor + Chipset + additional stuff (ports, power units)

2.4 Overview of an x86 Computer

- Platform/SoC: Processor + Chipset
- Mainboard: Processor + Chipset + additional stuff (ports, power units)
- Chipset

2.4 Overview of an x86 Computer

- Platform/SoC: Processor + Chipset
- Mainboard: Processor + Chipset + additional stuff (ports, power units)
- Chipset
  - Necessary logical functionality for CPU to work

2.4 Overview of an x86 Computer

- Platform/SoC: Processor + Chipset
- Mainboard: Processor + Chipset + additional stuff (ports, power units)
- Chipset
  - Necessary logical functionality for CPU to work
  - Built into your mainboard

2.4 Overview of an x86 Computer

- Platform/SoC: Processor + Chipset
- Mainboard: Processor + Chipset + additional stuff (ports, power units)
- Chipset
  - Necessary logical functionality for CPU to work
  - Built into your mainboard
  - Managing data flow between processor and memory & peripherals

2.4 Overview of an x86 Computer

- Platform/SoC: Processor + Chipset
- Mainboard: Processor + Chipset + additional stuff (ports, power units)
- Chipset
  - Necessary logical functionality for CPU to work
  - Built into your mainboard
  - Managing data flow between processor and memory & peripherals
- PCIe

2.4 Overview of an x86 Computer

- Platform/SoC: Processor + Chipset
- Mainboard: Processor + Chipset + additional stuff (ports, power units)
- Chipset
  - Necessary logical functionality for CPU to work
  - Built into your mainboard
  - Managing data flow between processor and memory & peripherals
- PCIe
  - Main interface/bus to orchestrate hardware and connect with chipset

2.4 Overview of an x86 Computer

- Platform/SoC: Processor + Chipset
- Mainboard: Processor + Chipset + additional stuff (ports, power units)
- Chipset
  - Necessary logical functionality for CPU to work
  - Built into your mainboard
  - Managing data flow between processor and memory & peripherals
- PCIe
  - Main interface/bus to orchestrate hardware and connect with chipset
  - Controller typically integrated into processor

2.4 Overview of an x86 Computer

- Platform/SoC: Processor + Chipset
- Mainboard: Processor + Chipset + additional stuff (ports, power units)
- Chipset
  - Necessary logical functionality for CPU to work
  - Built into your mainboard
  - Managing data flow between processor and memory & peripherals
- PCIe
  - Main interface/bus to orchestrate hardware and connect with chipset
  - Controller typically integrated into processor
  - Chipset has PCIe lanes

2.4 Processor, Chipset, Hardware

2.4 Processor, Chipset, Hardware

- Platform Controller Hub (PCH)

2.4 Processor, Chipset, Hardware

- Platform Controller Hub (PCH)
  - Intel’s name for a chipset family

2.4 Processor, Chipset, Hardware

- Platform Controller Hub (PCH)
  - Intel’s name for a chipset family
  - Before 2009: "Northbridge" + "Southbridge"

2.4 Processor, Chipset, Hardware

- Platform Controller Hub (PCH)
  - Intel’s name for a chipset family
  - Before 2009: "Northbridge" + "Southbridge"
  - Connects socket (processor) with memory, PCI lanes, power, ...

2.4 Processor, Chipset, Hardware

- Platform Controller Hub (PCH)
  - Intel’s name for a chipset family
  - Before 2009: "Northbridge" + "Southbridge"
  - Connects socket (processor) with memory, PCI lanes, power, ...
  - Example: USB controller and NVME controller appear as PCIe device

2.4 Processor, Chipset, Hardware

- Platform Controller Hub (PCH)
  - Intel’s name for a chipset family
  - Before 2009: "Northbridge" + "Southbridge"
  - Connects socket (processor) with memory, PCI lanes, power, ...
  - Example: USB controller and NVME controller appear as PCIe device
- Trivia: Mainboard manufacturer buys chipset IC(s) from Intel (e.g. Z390) and wires PCI lanes, memory bus, device slots. etc. as needed by the corresponding PCH spec + additional custom things

IC: Integrated Circuit

2.5 Accessing Hardware

2.5 Accessing Hardware

Memory-Mapped I/O (MMIO)

2.5 Accessing Hardware

Memory-Mapped I/O (MMIO)

- Physical memory addresses map to

2.5 Accessing Hardware

Memory-Mapped I/O (MMIO)

- Physical memory addresses map to
  - RAM cells

2.5 Accessing Hardware

Memory-Mapped I/O (MMIO)

- Physical memory addresses map to
  - RAM cells
  - Device registers

2.5 Accessing Hardware

Memory-Mapped I/O (MMIO)

- Physical memory addresses map to
  - RAM cells
  - Device registers
  - GPIO pin, ...

2.5 Accessing Hardware

Memory-Mapped I/O (MMIO)

- Physical memory addresses map to
  - RAM cells
  - Device registers
  - GPIO pin, ...
- `mov src, dst` instructions

2.5 Accessing Hardware

Memory-Mapped I/O (MMIO) Port I/O (PIO)

- Physical memory addresses map to
  - RAM cells
  - Device registers
  - GPIO pin, ...
- `mov src, dst` instructions

2.5 Accessing Hardware

Memory-Mapped I/O (MMIO)

- Physical memory addresses map to
  - RAM cells
  - Device registers
  - GPIO pin, ...
- `mov src, dst` instructions

Port I/O (PIO)

- X86 has a Port I/O address space

2.5 Accessing Hardware

Memory-Mapped I/O (MMIO)

- Physical memory addresses map to
  - RAM cells
  - Device registers
  - GPIO pin, ...
- `mov src, dst` instructions

Port I/O (PIO)

- X86 has a Port I/O address space
- “Write byte A to Port B”

2.5 Accessing Hardware

Memory-Mapped I/O (MMIO)

- Physical memory addresses map to
  - RAM cells
  - Device registers
  - GPIO pin, …
- `mov src, dst` instructions

Port I/O (PIO)

- X86 has a Port I/O address space
- “Write byte A to Port B”
- Port may map to a device register

2.5 Accessing Hardware

Memory-Mapped I/O (MMIO)

- Physical memory addresses map to
  - RAM cells
  - Device registers
  - GPIO pin, …
- `mov src, dst` instructions

Port I/O (PIO)

- X86 has a Port I/O address space
- “Write byte A to Port B”
- Port may map to a device register
- `in/out` instructions

That was a lot 😲 hardware is complex

That was a lot 😲 hardware is complex

Understanding the interfaces is key

That was a lot 😲 hardware is complex

Understanding the interfaces is key

2.6 Firmware

2.6 Firmware

- You need software to load software

2.6 Firmware

- You need software to load software
- Software that is not installable in the classic way

2.6 Firmware

- You need software to load software
- Software that is not installable in the classic way
- On-board in a simple chip with simple interface (just raw bytes)

2.6 Firmware

- You need software to load software
- Software that is not installable in the classic way
- On-board in a simple chip with simple interface (just raw bytes)
- Technically "just normal" software

2.6 Firmware

- You need software to load software
- Software that is not installable in the classic way
- On-board in a simple chip with simple interface (just raw bytes)
- Technically "just normal" software
- Examples:

2.6 Firmware

- You need software to load software
- Software that is not installable in the classic way
- On-board in a simple chip with simple interface (just raw bytes)
- Technically "just normal" software
- Examples:
  - Interfaces: Legacy BIOS ("IBM PC"), UEFI

2.6 Firmware

- You need software to load software
- Software that is not installable in the classic way
- On-board in a simple chip with simple interface (just raw bytes)
- Technically "just normal" software
- Examples:
  - Interfaces: Legacy BIOS ("IBM PC"), UEFI
  - Implementation: SeaBIOS, Coreboot, EDK2

2.6 Firmware

- You need software to load software
- Software that is not installable in the classic way
- On-board in a simple chip with simple interface (just raw bytes)
- Technically "just normal" software
- Examples:
  - Interfaces: Legacy BIOS ("IBM PC"), UEFI
  - Implementation: SeaBIOS, Coreboot, EDK2
- From CPU perspective: doesn’t know firmware variant

2.6 Firmware

2.6 Firmware

- Bootstraps the platform ("Platform initialization")

2.6 Firmware

- Bootstraps the platform ("Platform initialization")
- Brings platform and CPU into **defined state**

2.6 Firmware

- Bootstraps the platform ("Platform initialization")
- Brings platform and CPU into **defined state**
- Determines **interface for bootloader**

2.6 Firmware

- Bootstraps the platform ("Platform initialization")
- Brings platform and CPU into **defined state**
- Determines **interface for bootloader**
  - Executable format

2.6 Firmware

- Bootstraps the platform ("Platform initialization")
- Brings platform and CPU into **defined state**
- Determines **interface for bootloader**
  - Executable format
  - Environment

2.6 Trivia: Intel SDM: Initialization

2.6 Trivia: Intel SDM: Initialization

- Keyword: "Hardware Reset"

2.6 Trivia: Intel SDM: Initialization

- Keyword: "Hardware Reset"
- 10. Processor Management and Initialization

2.6 Trivia: Intel SDM: Initialization

- Keyword: "Hardware Reset"
- 10. Processor Management and Initialization
  - 10.1 INITIALIZATION OVERVIEW
    - 10.1.4 First Instruction Executed

      → Hardware software co-design

2.7 UEFI

Towards a unified firmware.

2.7 UEFI

Towards a unified firmware.

This Unified Extensible Firmware Interface (UEFI) Specification describes an interface between the operating system (OS) and the platform firmware.

2.7 UEFI

Towards a unified firmware.

This Unified Extensible Firmware Interface (UEFI) Specification describes an interface between the operating system (OS) and the platform firmware.

[...]

The interface is in the form of data tables that contain platform-related information, and boot and runtime service calls that are available to the OS loader and the OS. Together, these provide a standard environment for booting an OS.

2.7 UEFI

2.7 UEFI

- **U**nified **E**xtensible **F**irmware **I**nterface

2.7 UEFI

- **U**nified **E**xtensible **F**irmware **I**nterface
- Developed by Tianocore community

2.7 UEFI

- **U**nified **E**xtensible **F**irmware **I**nterface
- Developed by Tianocore community
- "EDK2"

2.7 UEFI

- **U**nified **E**xtensible **F**irmware **I**nterface
- Developed by Tianocore community
- "EDK2"
  - Build system

2.7 UEFI

- **U**nified **E**xtensible **F**irmware **I**nterface
- Developed by Tianocore community
- "EDK2"
  - Build system
  - Reference implementation written in C, C++, and Assembly

2.7 UEFI

- **U**nified **E**xtensible **F**irmware **I**nterface
- Developed by Tianocore community
- "EDK2"
  - Build system
  - Reference implementation written in C, C++, and Assembly
  - Open source on GitHub

2.7 UEFI

- **U**nified **E**xtensible **F**irmware **I**nterface
- Developed by Tianocore community
- "EDK2"
  - Build system
  - Reference implementation written in C, C++, and Assembly
  - Open source on GitHub
- Other implementations exists

2.7 UEFI

2.7 UEFI

- Gives us a defined machine state

2.7 UEFI

- Gives us a defined machine state
- 64-bit mode, yay!

2.7 UEFI

- Gives us a defined machine state
- 64-bit mode, yay!
- Only one CPU ("Bootstrap Processor" (BSP))

2.7 UEFI

- Gives us a defined machine state
- 64-bit mode, yay!
- Only one CPU ("Bootstrap Processor" (BSP))
- Others are ready to be woken up ("Application Processors" (APs))

2.7 UEFI

- Gives us a defined machine state
- 64-bit mode, yay!
- Only one CPU ("Bootstrap Processor" (BSP))
- Others are ready to be woken up ("Application Processors" (APs))
- Can load EFI images (binaries, executables)

2.7 UEFI

- Gives us a defined machine state
- 64-bit mode, yay!
- Only one CPU ("Bootstrap Processor" (BSP))
- Others are ready to be woken up ("Application Processors" (APs))
- Can load EFI images (binaries, executables)
  - Similar to starting an `.exe` on Windows

2.7 UEFI

- Gives us a defined machine state
- 64-bit mode, yay!
- Only one CPU ("Bootstrap Processor" (BSP))
- Others are ready to be woken up ("Application Processors" (APs))
- Can load EFI images (binaries, executables)
  - Similar to starting an `.exe` on Windows
  - Stack is provided

2.7 UEFI

- Gives us a defined machine state
- 64-bit mode, yay!
- Only one CPU ("Bootstrap Processor" (BSP))
- Others are ready to be woken up ("Application Processors" (APs))
- Can load EFI images (binaries, executables)
  - Similar to starting an `.exe` on Windows
  - Stack is provided
  - UEFI functionality is callable from EFI image

2.7 UEFI

2.7 UEFI

- Fixed set of functionality ("services") + variable part ("protocols")

2.7 UEFI

- Fixed set of functionality ("services") + variable part ("protocols")
  - Services are callable functions

2.7 UEFI

- Fixed set of functionality ("services") + variable part ("protocols")
  - Services are callable functions
  - Protocols are somewhat like interfaces in Java or traits in Rust

2.7 UEFI

- Fixed set of functionality ("services") + variable part ("protocols")
  - Services are callable functions
  - Protocols are somewhat like interfaces in Java or traits in Rust
- Two phases

2.7 UEFI

- Fixed set of functionality ("services") + variable part ("protocols")
  - Services are callable functions
  - Protocols are somewhat like interfaces in Java or traits in Rust
- Two phases
  - **Boot-Services**

2.7 UEFI

- Fixed set of functionality ("services") + variable part ("protocols")
  - Services are callable functions
  - Protocols are somewhat like interfaces in Java or traits in Rust
- Two phases
  - **Boot-Services**
    - UEFI has full control over hardware (like an OS)

2.7 UEFI

- Fixed set of functionality ("services") + variable part ("protocols")
  - Services are callable functions
  - Protocols are somewhat like interfaces in Java or traits in Rust
- Two phases
  - **Boot-Services**
    - UEFI has full control over hardware (like an OS)
    - Provide feature-rich and high(er)-level interface to hardware

2.7 UEFI

- Fixed set of functionality ("services") + variable part ("protocols")
  - Services are callable functions
  - Protocols are somewhat like interfaces in Java or traits in Rust
- Two phases
  - **Boot-Services**
    - UEFI has full control over hardware (like an OS)
    - Provide feature-rich and high(er)-level interface to hardware
    - Must be exited before OS can take over control

2.7 UEFI

- Fixed set of functionality ("services") + variable part ("protocols")
  - Services are callable functions
  - Protocols are somewhat like interfaces in Java or traits in Rust
- Two phases
  - **Boot-Services**
    - UEFI has full control over hardware (like an OS)
    - Provide feature-rich and high(er)-level interface to hardware
    - Must be exited before OS can take over control
  - Runtime-Services

2.7 UEFI

- Fixed set of functionality ("services") + variable part ("protocols")
  - Services are callable functions
  - Protocols are somewhat like interfaces in Java or traits in Rust
- Two phases
  - **Boot-Services**
    - UEFI has full control over hardware (like an OS)
    - Provide feature-rich and high(er)-level interface to hardware
    - Must be exited before OS can take over control
  - Runtime-Services
    - Tiny fraction of remaining functionality: System time, UEFI variables

2.7 UEFI

2.7 UEFI

- Identifies resources and abstracts device access with `EFI_HANDLE`s

2.7 UEFI

- Identifies resources and abstracts device access with `EFI_HANDLE`s
- Handles know their associated protocols

2.7 UEFI

- Identifies resources and abstracts device access with `EFI_HANDLE`s
- Handles know their associated protocols
- Technically, a protocol is a `C` struct holding functions and/or data, with an associated GUID

```
typedef struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
 UINT64                                          Revision;
 // This is a function pointer
 EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME     OpenVolume;
} EFI_SIMPLE_FILE_SYSTEM_PROTOCOL;
```

2.7 UEFI

- Identifies resources and abstracts device access with `EFI_HANDLE`s
- Handles know their associated protocols
- Technically, a protocol is a `C` struct holding functions and/or data, with an associated GUID

```
1  typedef struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
2   UINT64                                          Revision;
3   // This is a function pointer
4   EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME     OpenVolume;
5  } EFI_SIMPLE_FILE_SYSTEM_PROTOCOL;
```

2.7 UEFI

- Identifies resources and abstracts device access with `EFI_HANDLE`s
- Handles know their associated protocols
- Technically, a protocol is a `C` struct holding functions and/or data, with an associated GUID

```
typedef struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
 UINT64                                          Revision;
 // This is a function pointer
 EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME    OpenVolume;
} EFI_SIMPLE_FILE_SYSTEM_PROTOCOL;
```

2.7 UEFI

- Identifies resources and abstracts device access with `EFI_HANDLE`s
- Handles know their associated protocols
- Technically, a protocol is a `C` struct holding functions and/or data, with an associated GUID

```
typedef struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
 UINT64                                          Revision;
 // This is a function pointer
 EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME     OpenVolume;
} EFI_SIMPLE_FILE_SYSTEM_PROTOCOL;
```

2.7 UEFI

- Identifies resources and abstracts device access with `EFI_HANDLE`s
- Handles know their associated protocols
- Technically, a protocol is a `C` struct holding functions and/or data, with an associated GUID

```
1  typedef struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
2   UINT64                                          Revision;
3   // This is a function pointer
4   EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME     OpenVolume;
5  } EFI_SIMPLE_FILE_SYSTEM_PROTOCOL;
```

2.7 UEFI

2.7 UEFI

- Boot service examples

2.7 UEFI

- Boot service examples
  - `OpenProtocol()` : Tries opening a protocol on a given handle

2.7 UEFI

- Boot service examples
  - `OpenProtocol()` : Tries opening a protocol on a given handle
  - `LocateHandle()` : Finds handles supporting a given protocol

2.7 UEFI

- Boot service examples
  - `OpenProtocol()` : Tries opening a protocol on a given handle
  - `LocateHandle()` : Finds handles supporting a given protocol
- Protocol examples

2.7 UEFI

- Boot service examples
  - `OpenProtocol()` : Tries opening a protocol on a given handle
  - `LocateHandle()` : Finds handles supporting a given protocol
- Protocol examples
  - `EFI_GRAPHICS_OUTPUT_PROTOCOL` :

    Draw to framebuffer

2.7 UEFI

- Boot service examples
  - `OpenProtocol()` : Tries opening a protocol on a given handle
  - `LocateHandle()` : Finds handles supporting a given protocol
- Protocol examples
  - `EFI_GRAPHICS_OUTPUT_PROTOCOL` :

    Draw to framebuffer
  - `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` :

    Access files

2.7 UEFI: Extensible?

By 3rd Party Hardware

2.7 UEFI: Extensible?

By 3rd Party Hardware

- PCIe devices can advertise additional UEFI drivers ("Option ROM")

2.7 UEFI: Extensible?

By 3rd Party Hardware

- PCIe devices can advertise additional UEFI drivers ("Option ROM")
- Examples

2.7 UEFI: Extensible?

By 3rd Party Hardware

- PCIe devices can advertise additional UEFI drivers ("Option ROM")
- Examples
  - An NVIDIA GPU may install the `EFI_GRAPHICS_OUTPUT_PROTOCOL` on its corresponding handle

2.7 UEFI: Extensible?

By 3rd Party Hardware

- PCIe devices can advertise additional UEFI drivers ("Option ROM")
- Examples
  - An NVIDIA GPU may install the `EFI_GRAPHICS_OUTPUT_PROTOCOL` on its corresponding handle
  - A network card may install the `EFI_PXE_BASE_CODE_PROTOCOL` on its corresponding handle

2.7 UEFI: Extensible?

By 3rd Party Hardware

- PCIe devices can advertise additional UEFI drivers ("Option ROM")
- Examples
  - An NVIDIA GPU may install the `EFI_GRAPHICS_OUTPUT_PROTOCOL` on its corresponding handle
  - A network card may install the `EFI_PXE_BASE_CODE_PROTOCOL` on its corresponding handle
- UEFI firmware may also have built-in drivers for common hardware

2.7 UEFI: Extensible?

By 3rd Party Hardware

- PCIe devices can advertise additional UEFI drivers ("Option ROM")
- Examples
  - An NVIDIA GPU may install the `EFI_GRAPHICS_OUTPUT_PROTOCOL` on its corresponding handle
  - A network card may install the `EFI_PXE_BASE_CODE_PROTOCOL` on its corresponding handle
- UEFI firmware may also have built-in drivers for common hardware
- We as software developers can use them

2.7 UEFI: Extensible?

By software developers

2.7 UEFI: Extensible?

By software developers

- In our OS loader, we can install protocols or use protocols on any handle

2.7 UEFI: Extensible?

By software developers

- In our OS loader, we can install protocols or use protocols on any handle
- We may chainload another bootloader (systemd boot, Windows bootloader)

2.7 UEFI: Extensible?

By software developers

- In our OS loader, we can install protocols or use protocols on any handle
- We may chainload another bootloader (systemd boot, Windows bootloader)
- A lot of options!

2.8 Utilizing UEFI

2.8 Utilizing UEFI

- Writing your own (portable) OS-loader is easy

2.8 Utilizing UEFI

- Writing your own (portable) OS-loader is easy
- Defined executable file format 🎉 (somewhat similar to `.exe`)

2.8 Utilizing UEFI

- Writing your own (portable) OS-loader is easy
- Defined executable file format 🎉 (somewhat similar to `.exe`)
  - Subset of PE32+ file format (Windows' `.exe` format)

2.8 Utilizing UEFI

- Writing your own (portable) OS-loader is easy
- Defined executable file format 🎉 (somewhat similar to `.exe`)
  - Subset of PE32+ file format (Windows' `.exe` format)
  - By default, loaded from `<drive>\EFI\BOOT\BOOTX64.EFI` (FAT partition)

2.8 Utilizing UEFI

- Writing your own (portable) OS-loader is easy
- Defined executable file format 🎉 (somewhat similar to `.exe`)
  - Subset of PE32+ file format (Windows' `.exe` format)
  - By default, loaded from `<drive>\EFI\BOOT\BOOTX64.EFI` (FAT partition)
- We can use extended functionality with UEFI protocols

2.8 Utilizing UEFI

- Writing your own (portable) OS-loader is easy
- Defined executable file format 🎉 (somewhat similar to `.exe`)
  - Subset of PE32+ file format (Windows' `.exe` format)
  - By default, loaded from `<drive>\EFI\BOOT\BOOTX64.EFI` (FAT partition)
- We can use extended functionality with UEFI protocols
  - `EFI_GRAPHICS_OUTPUT_PROTOCOL`: No extra GPU driver needed

2.8 Utilizing UEFI

- Writing your own (portable) OS-loader is easy
- Defined executable file format 🎉 (somewhat similar to `.exe`)
  - Subset of PE32+ file format (Windows' `.exe` format)
  - By default, loaded from `<drive>\EFI\BOOT\BOOTX64.EFI` (FAT partition)
- We can use extended functionality with UEFI protocols
  - `EFI_GRAPHICS_OUTPUT_PROTOCOL`: No extra GPU driver needed
  - `EFI_PXE_BASE_CODE_PROTOCOL`: No extra TCP + PXE driver needed

2.8 Utilizing UEFI

- Writing your own (portable) OS-loader is easy
- Defined executable file format 🎉 (somewhat similar to `.exe`)
  - Subset of PE32+ file format (Windows' `.exe` format)
  - By default, loaded from `<drive>\EFI\BOOT\BOOTX64.EFI` (FAT partition)
- We can use extended functionality with UEFI protocols
  - `EFI_GRAPHICS_OUTPUT_PROTOCOL`: No extra GPU driver needed
  - `EFI_PXE_BASE_CODE_PROTOCOL`: No extra TCP + PXE driver needed
  - `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL`: No extra NVMe or FAT driver needed

2.8 Utilizing UEFI

- Writing your own (portable) OS-loader is easy
- Defined executable file format 🎉 (somewhat similar to `.exe`)
  - Subset of PE32+ file format (Windows' `.exe` format)
  - By default, loaded from `<drive>\EFI\BOOT\BOOTX64.EFI` (FAT partition)
- We can use extended functionality with UEFI protocols
  - `EFI_GRAPHICS_OUTPUT_PROTOCOL`: No extra GPU driver needed
  - `EFI_PXE_BASE_CODE_PROTOCOL`: No extra TCP + PXE driver needed
  - `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL`: No extra NVMe or FAT driver needed
- OS-loader typically exits boot services

2.8 Utilizing UEFI

- Writing your own (portable) OS-loader is easy
- Defined executable file format 🎉 (somewhat similar to `.exe`)
  - Subset of PE32+ file format (Windows' `.exe` format)
  - By default, loaded from `<drive>\EFI\BOOT\BOOTX64.EFI` (FAT partition)
- We can use extended functionality with UEFI protocols
  - `EFI_GRAPHICS_OUTPUT_PROTOCOL`: No extra GPU driver needed
  - `EFI_PXE_BASE_CODE_PROTOCOL`: No extra TCP + PXE driver needed
  - `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL`: No extra NVMe or FAT driver needed
- OS-loader typically exits boot services
- Kernel has its own drivers (PCIe, NVMe)

2.8 Utilizing UEFI: In a Nutshell

From Developer Perspective

2.8 Utilizing UEFI: In a Nutshell

From Developer Perspective

- We have an OS-like environment

2.8 Utilizing UEFI: In a Nutshell

From Developer Perspective

- We have an OS-like environment
- Higher-level abstractions to

2.8 Utilizing UEFI: In a Nutshell

From Developer Perspective

- We have an OS-like environment
- Higher-level abstractions to
  - Load files

2.8 Utilizing UEFI: In a Nutshell

From Developer Perspective

- We have an OS-like environment
- Higher-level abstractions to
  - Load files
  - Access network

2.8 Utilizing UEFI: In a Nutshell

From Developer Perspective

- We have an OS-like environment
- Higher-level abstractions to
  - Load files
  - Access network
  - Draw to the screen

2.8 Utilizing UEFI: In a Nutshell

From Developer Perspective

- We have an OS-like environment
- Higher-level abstractions to
  - Load files
  - Access network
  - Draw to the screen
  - Get user input

2.8 Utilizing UEFI: In a Nutshell

From Developer Perspective

- We have an OS-like environment
- Higher-level abstractions to
  - Load files
  - Access network
  - Draw to the screen
  - Get user input
- No need to fiddle with own PCIe, network drivers, or GPU drivers

2.8 Utilizing UEFI: In a Nutshell

From Developer Perspective

- We have an OS-like environment
- Higher-level abstractions to
  - Load files
  - Access network
  - Draw to the screen
  - Get user input
- No need to fiddle with own PCIe, network drivers, or GPU drivers
- Makes loading your kernel just easy

2.9 Summary

2.9 Summary

- Hardware is complex

2.9 Summary

- Hardware is complex
- UEFI is de-facto standard firmware interface making things easier

2.9 Summary

- Hardware is complex
- UEFI is de-facto standard firmware interface making things easier
  - Hardware manufacturers

2.9 Summary

- Hardware is complex
- UEFI is de-facto standard firmware interface making things easier
  - Hardware manufacturers
  - Firmware developers

2.9 Summary

- Hardware is complex
- UEFI is de-facto standard firmware interface making things easier
  - Hardware manufacturers
  - Firmware developers
  - Software developers

2.9 Summary

- Hardware is complex
- UEFI is de-facto standard firmware interface making things easier
  - Hardware manufacturers
  - Firmware developers
  - Software developers
- EDK2 is default UEFI implementation

2.9 Summary

- Hardware is complex
- UEFI is de-facto standard firmware interface making things easier
  - Hardware manufacturers
  - Firmware developers
  - Software developers
- EDK2 is default UEFI implementation
- UEFI provides higher level abstractions to access e.g. files

2.9 Summary

- Hardware is complex
- UEFI is de-facto standard firmware interface making things easier
  - Hardware manufacturers
  - Firmware developers
  - Software developers
- EDK2 is default UEFI implementation
- UEFI provides higher level abstractions to access e.g. files
- OS-loaders / bootloaders are EFI applications

3. uefi-rs

Mastering UEFI with Rust

3. uefi-rs

Mastering UEFI with Rust

- github.com/rust-osdev/uefi-rs (`uefi` library on crates.io)

3. uefi-rs

Mastering UEFI with Rust

- github.com/rust-osdev/uefi-rs (`uefi` library on crates.io)
- *Makes it easy to develop Rust software that leverages safe, convenient, and performant abstractions for UEFI functionality.*

3. uefi-rs

Mastering UEFI with Rust

- github.com/rust-osdev/uefi-rs (`uefi` library on crates.io)
- *Makes it easy to develop Rust software that leverages safe, convenient, and performant abstractions for UEFI functionality.*
- High-level wrappers for interfacing UEFI (not an UEFI implementation!)

3. uefi-rs

Mastering UEFI with Rust

- github.com/rust-osdev/uefi-rs (`uefi` library on crates.io)
- *Makes it easy to develop Rust software that leverages safe, convenient, and performant abstractions for UEFI functionality.*
- High-level wrappers for interfacing UEFI (not an UEFI implementation!)
- Maintaining since August 2022 together with Nicholas Bishop (Google)

3. uefi-rs

Mastering UEFI with Rust

- github.com/rust-osdev/uefi-rs (`uefi` library on crates.io)
- *Makes it easy to develop Rust software that leverages safe, convenient, and performant abstractions for UEFI functionality.*
- High-level wrappers for interfacing UEFI (not an UEFI implementation!)
- Maintaining since August 2022 together with Nicholas Bishop (Google)
- So far, I've touched every part of the code

3. uefi-rs

Mastering UEFI with Rust

- github.com/rust-osdev/uefi-rs (`uefi` library on crates.io)
- *Makes it easy to develop Rust software that leverages safe, convenient, and performant abstractions for UEFI functionality.*
- High-level wrappers for interfacing UEFI (not an UEFI implementation!)
- Maintaining since August 2022 together with Nicholas Bishop (Google)
- So far, I've touched every part of the code
- Code powers ChromeOS Flex notebooks and also runs in Amazon AWS

3. uefi-rs

3. uefi-rs

- `rustc` can compile EFI applications → compiler target `x86_64-unknown-uefi`

3. uefi-rs

- `rustc` can compile EFI applications → compiler target `x86_64-unknown-uefi`
- Library helps writing EFI applications

3. uefi-rs

- `rustc` can compile EFI applications → compiler target `x86_64-unknown-uefi`
- Library helps writing EFI applications
- Helps loading a kernel

3. uefi-rs

- `rustc` can compile EFI applications → compiler target `x86_64-unknown-uefi`
- Library helps writing EFI applications
- Helps loading a kernel
- Selected highlights

3. uefi-rs

- `rustc` can compile EFI applications → compiler target `x86_64-unknown-uefi`
- Library helps writing EFI applications
- Helps loading a kernel
- Selected highlights
  - File system abstraction (proudly crafted by me 😃)

3. uefi-rs

- `rustc` can compile EFI applications → compiler target `x86_64-unknown-uefi`
- Library helps writing EFI applications
- Helps loading a kernel
- Selected highlights
  - File system abstraction (proudly crafted by me 😃)
  - Handling device paths with ease

The device path protocol, also called a *device path*, is a flexible and structured sequence of binary nodes that describes a route from the UEFI root to a particular device, controller, or file.

3. uefi-rs

- `rustc` can compile EFI applications → compiler target `x86_64-unknown-uefi`
- Library helps writing EFI applications
- Helps loading a kernel
- Selected highlights
  - File system abstraction (proudly crafted by me 😃)
  - Handling device paths with ease
  - Integration of UEFI's allocator into Rust's global allocator

The device path protocol, also called a *device path*, is a flexible and structured sequence of binary nodes that describes a route from the UEFI root to a particular device, controller, or file.

3. uefi-rs: Code Example: Hello World

Creating a `no_std` binary (executable)

```
1  #![no_main]
2  #![no_std]
```

3. uefi-rs: Code Example: Hello World

Creating a `no_std` binary (executable)

```
1  #![no_main]
2  #![no_std]
```

3. uefi-rs: Code Example: Hello World

Creating a `no_std` binary (executable)

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

3. uefi-rs: Code Example: Hello World

Creating a `no_std` binary (executable)

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

use uefi::prelude::*;

#[entry]
fn main() -> Status {
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Hello World

Creating a `no_std` binary (executable)

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

use uefi::prelude::*;

#[entry]
fn main() -> Status {
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Hello World

Creating a `no_std` binary (executable)

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

use uefi::prelude::*;

#[entry]
fn main() -> Status {
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Hello World

Creating a `no_std` binary (executable)

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

use uefi::prelude::*;

#[entry]
fn main() -> Status {
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Hello World

Creating a `no_std` binary (executable)

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

use uefi::prelude::*;

#[entry]
fn main() -> Status {
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Hello World

Creating a `no_std` binary (executable)

```
#[unsafe(export_name = "efi_main")]
extern "efiapi" fn main(
    internal_image_handle: ::uefi::Handle,
    internal_system_table: *const ::core::ffi::c_void,
) -> uefi::Status {
    unsafe {
        ::uefi::boot::set_image_handle(internal_image_handle);
        ::uefi::table::set_system_table(internal_system_table.cast());
    }
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Hello World

Creating a `no_std` binary (executable)

```
#[unsafe(export_name = "efi_main")]
extern "efiapi" fn main(
    internal_image_handle: ::uefi::Handle,
    internal_system_table: *const ::core::ffi::c_void,
) -> uefi::Status {
    unsafe {
        ::uefi::boot::set_image_handle(internal_image_handle);
        ::uefi::table::set_system_table(internal_system_table.cast());
    }
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Hello World

Creating a `no_std` binary (executable)

```
#[unsafe(export_name = "efi_main")]
extern "efiapi" fn main(
    internal_image_handle: ::uefi::Handle,
    internal_system_table: *const ::core::ffi::c_void,
) -> uefi::Status {
    unsafe {
        ::uefi::boot::set_image_handle(internal_image_handle);
        ::uefi::table::set_system_table(internal_system_table.cast());
    }
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Hello World

Creating a `no_std` binary (executable)

```
#[unsafe(export_name = "efi_main")]
extern "efiapi" fn main(
    internal_image_handle: ::uefi::Handle,
    internal_system_table: *const ::core::ffi::c_void,
) -> uefi::Status {
    unsafe {
        ::uefi::boot::set_image_handle(internal_image_handle);
        ::uefi::table::set_system_table(internal_system_table.cast());
    }
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Hello World

Creating a `no_std` binary (executable)

```
#[unsafe(export_name = "efi_main")]
extern "efiapi" fn main(
    internal_image_handle: ::uefi::Handle,
    internal_system_table: *const ::core::ffi::c_void,
) -> uefi::Status {
    unsafe {
        ::uefi::boot::set_image_handle(internal_image_handle);
        ::uefi::table::set_system_table(internal_system_table.cast());
    }
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Hello World

Creating a `no_std` binary (executable)

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

use uefi::prelude::*;

#[entry]
fn main() -> Status {
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Hello World

Creating a `no_std` binary (executable)

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

use log::info;
use uefi::prelude::*;

#[entry]
fn main() -> Status {
    uefi::helpers::init().unwrap();
    info!("Hello world!");
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Hello World

Creating a `no_std` binary (executable)

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

use core::time::Duration;
use log::info;
use uefi::prelude::*;

#[entry]
fn main() -> Status {
    uefi::helpers::init().unwrap();
    info!("Hello world!");
    boot::stall(Duration::from_secs(10));
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Reading File

```
#[entry]
fn main() -> Status {
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Reading File

```
#[entry]
fn main() -> Status {
    helpers::init().unwrap(); // enable `log`-crate
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Reading File

```
#[entry]
fn main() -> Status {
    helpers::init().unwrap(); // enable `log`-crate
    let sfs_proto = boot::get_image_file_system(boot::image_handle()).unwrap();
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Reading File

```
#[entry]
fn main() -> Status {
    helpers::init().unwrap(); // enable `log`-crate
    let sfs_proto = boot::get_image_file_system(boot::image_handle()).unwrap();
    let mut fs = FileSystem::new(sfs_proto); // Abstraction similar to `std::fs`
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Reading File

```
#[entry]
fn main() -> Status {
    helpers::init().unwrap(); // enable `log`-crate
    let sfs_proto = boot::get_image_file_system(boot::image_handle()).unwrap();
    let mut fs = FileSystem::new(sfs_proto); // Abstraction similar to `std::fs`
    for entry in fs.read_dir(cstr16!("EFI\\BOOT")).unwrap() {
    }
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Reading File

```
#[entry]
fn main() -> Status {
    helpers::init().unwrap(); // enable `log`-crate
    let sfs_proto = boot::get_image_file_system(boot::image_handle()).unwrap();
    let mut fs = FileSystem::new(sfs_proto); // Abstraction similar to `std::fs`
    for entry in fs.read_dir(cstr16!("EFI\\BOOT")).unwrap() {
        let entry = entry.unwrap();
        let kind = if entry.is_directory() { "dir" } else { "file" };
        info!("Found: {kind} {}", entry.file_name());
    }
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Reading File

```
#[entry]
fn main() -> Status {
    helpers::init().unwrap(); // enable `log`-crate
    let sfs_proto = boot::get_image_file_system(boot::image_handle()).unwrap();
    let mut fs = FileSystem::new(sfs_proto); // Abstraction similar to `std::fs`
    for entry in fs.read_dir(cstr16!("EFI\\BOOT")).unwrap() {
        let entry = entry.unwrap();
        let kind = if entry.is_directory() { "dir" } else { "file" };
        info!("Found: {kind} {}", entry.file_name());
    }
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Reading File

```
#[entry]
fn main() -> Status {
    helpers::init().unwrap(); // enable `log`-crate
    let sfs_proto = boot::get_image_file_system(boot::image_handle()).unwrap();
    let mut fs = FileSystem::new(sfs_proto); // Abstraction similar to `std::fs`
    for entry in fs.read_dir(cstr16!("EFI\\BOOT")).unwrap() {
        let entry = entry.unwrap();
        let kind = if entry.is_directory() { "dir" } else { "file" };
        info!("Found: {kind} {}", entry.file_name());
    }
    Status::SUCCESS
}
```

3. uefi-rs: Code Example: Reading File

```rust
#[entry]
fn main() -> Status {
    helpers::init().unwrap(); // enable `log`-crate
    let sfs_proto = boot::get_image_file_system(boot::image_handle()).unwrap();
    let mut fs = FileSystem::new(sfs_proto); // Abstraction similar to `std::fs`
    for entry in fs.read_dir(cstr16!("EFI\\BOOT")).unwrap() {
        let entry = entry.unwrap();
        let kind = if entry.is_directory() { "dir" } else { "file" };
        info!("Found: {kind} {}", entry.file_name());
    }
    Status::SUCCESS
}
```

```
[ INFO]:  src/main.rs@165: Found: dir .
[ INFO]:  src/main.rs@165: Found: dir ..
[ INFO]:  src/main.rs@165: Found: file BOOTX64.EFI
```

3. uefi-rs: Code Example: Device Paths

```
let handles = boot::find_handles::<DevicePath>().unwrap();
for handle in handles.iter() {
}
```

3. uefi-rs: Code Example: Device Paths

```
let handles = boot::find_handles::<DevicePath>().unwrap();
for handle in handles.iter() {
}
```

3. uefi-rs: Code Example: Device Paths

```
let handles = boot::find_handles::<DevicePath>().unwrap();
for handle in handles.iter() {
}
```

3. uefi-rs: Code Example: Device Paths

```
let handles = boot::find_handles::<DevicePath>().unwrap();
for handle in handles.iter() {
    let maybe_dvp = unsafe {
        boot::open_protocol::<DevicePath>(
            OpenProtocolParams { handle: *handle,
              agent: boot::image_handle(),
              controller: None, },
            OpenProtocolAttributes::GetProtocol,
        )
    };
    // Pattern matching: Unwrap happy path or continue
    let Ok(dvp) = maybe_dvp else { continue };
    let string = dvp.to_string(DisplayOnly(true), AllowShortcuts(false)).unwrap();
    info!("Device path: {}", string);
}
```

3. uefi-rs: Code Example: Device Paths

```
let handles = boot::find_handles::<DevicePath>().unwrap();
for handle in handles.iter() {
    let maybe_dvp = unsafe {
        boot::open_protocol::<DevicePath>(
            OpenProtocolParams { handle: *handle,
              agent: boot::image_handle(),
              controller: None, },
            OpenProtocolAttributes::GetProtocol,
        )
    };
    // Pattern matching: Unwrap happy path or continue
    let Ok(dvp) = maybe_dvp else { continue };
    let string = dvp.to_string(DisplayOnly(true), AllowShortcuts(false)).unwrap();
    info!("Device path: {}", string);
}
```

3. uefi-rs: Code Example: Device Paths

```
let handles = boot::find_handles::<DevicePath>().unwrap();
for handle in handles.iter() {
    let maybe_dvp = unsafe {
        boot::open_protocol::<DevicePath>(
            OpenProtocolParams { handle: *handle,
              agent: boot::image_handle(),
              controller: None, },
            OpenProtocolAttributes::GetProtocol,
        )
    };
    // Pattern matching: Unwrap happy path or continue
    let Ok(dvp) = maybe_dvp else { continue };
    let string = dvp.to_string(DisplayOnly(true), AllowShortcuts(false)).unwrap();
    info!("Device path: {}", string);
}
```

3. uefi-rs: Code Example: Device Paths

```
let handles = boot::find_handles::<DevicePath>().unwrap();
for handle in handles.iter() {
    let dvp = /* .. */;
    let string = dvp.to_string(DisplayOnly(true), AllowShortcuts(false)).unwrap();
    info!("Device path: {}", string);
}
```

3. uefi-rs: Code Example: Device Paths

```
let handles = boot::find_handles::<DevicePath>().unwrap();
for handle in handles.iter() {
    let dvp = /* .. */;
    let string = dvp.to_string(DisplayOnly(true), AllowShortcuts(false)).unwrap();
    info!("Device path: {}", string);
}
```

```
[ INFO]:  src/main.rs@178: Device path: Fv(7CB8BDC9-F8EB-4F34-AAEA-3EE4AF6516A1)
[ INFO]:  src/main.rs@178: Device path: MemoryMapped(0xB,0x1FEDC000,0x1FF5FFFF)
[ INFO]:  src/main.rs@178: Device path: PciRoot(0x0)
[ INFO]:  src/main.rs@178: Device path: VenHw(EBF8ED7C-0DD1-4787-84F1-F48D537DCACF)
[ INFO]:  src/main.rs@178: Device path: VenHw(28A03FF4-12B3-4305-A417-BB1A4F94081E)
[ INFO]:  src/main.rs@178: Device path: VenHw(2A46715F-3581-4A55-8E73-2B769AAA30C5)
[ INFO]:  src/main.rs@178: Device path: VenHw(D9DCC5DF-4007-435E-9098-8970935504B2)
[ INFO]:  src/main.rs@178: Device path: PciRoot(0x0)/Pci(0x0,0x0)
```

3. uefi-rs: How to Test? How to Run?

3. uefi-rs: How to Test? How to Run?

- On real hardware, i.e., developer laptop

3. uefi-rs: How to Test? How to Run?

- On real hardware, i.e., developer laptop
- In a VM

3. uefi-rs: How to Test? How to Run?

- On real hardware, i.e., developer laptop
- In a VM
  - e.g., QEMU or Cloud Hypervisor with OVMF firmware

3. uefi-rs: How to Test? How to Run?

- On real hardware, i.e., developer laptop
- In a VM
  - e.g., QEMU or Cloud Hypervisor with OVMF firmware
  - OVMF is an EDK2 build for Virtual Machines

4. Code

GitHub: phip1611/uefi-systemd-chainloader

5. Summary & Conclusion

5. Summary

5. Summary

- UEFI simplifies and unifies some things

5. Summary

- UEFI simplifies and unifies some things
- The domain is complex, and so is UEFI

5. Summary

- UEFI simplifies and unifies some things
- The domain is complex, and so is UEFI
- systemd boot, GRUB, the Windows bootloader → EFI applications

5. Summary

- UEFI simplifies and unifies some things
- The domain is complex, and so is UEFI
- systemd boot, GRUB, the Windows bootloader → EFI applications
- To get started: `uefi` crate; example project; run in VM

5. Summary

- UEFI simplifies and unifies some things
- The domain is complex, and so is UEFI
- systemd boot, GRUB, the Windows bootloader → EFI applications
- To get started: `uefi` crate; example project; run in VM
  - github.com/rust-osdev/uefi-rs

5. Summary

- UEFI simplifies and unifies some things
- The domain is complex, and so is UEFI
- systemd boot, GRUB, the Windows bootloader → EFI applications
- To get started: `uefi` crate; example project; run in VM
  - github.com/rust-osdev/uefi-rs
  - crates.io/crates/uefi

5. Summary

- UEFI simplifies and unifies some things
- The domain is complex, and so is UEFI
- systemd boot, GRUB, the Windows bootloader → EFI applications
- To get started: `uefi` crate; example project; run in VM
  - github.com/rust-osdev/uefi-rs
  - crates.io/crates/uefi
  - docs.rs/uefi

5. UEFI Criticism

5. UEFI Criticism

- Spec sometimes not specific enough

5. UEFI Criticism

- Spec sometimes not specific enough
- Implementations are often buggy (derived from EDK2)

5. UEFI Criticism

- Spec sometimes not specific enough
- Implementations are often buggy (derived from EDK2)
- Overly complex and inconsistent

5. UEFI Criticism

- Spec sometimes not specific enough
- Implementations are often buggy (derived from EDK2)
- Overly complex and inconsistent
- Most vendors add closed-source additions

5. UEFI Criticism

- Spec sometimes not specific enough
- Implementations are often buggy (derived from EDK2)
- Overly complex and inconsistent
- Most vendors add closed-source additions
- Tries to be a modern OS-like environment but sticks to decade old concepts

5. UEFI Criticism

- Spec sometimes not specific enough
- Implementations are often buggy (derived from EDK2)
- Overly complex and inconsistent
- Most vendors add closed-source additions
- Tries to be a modern OS-like environment but sticks to decade old concepts
  - A single global address space for everything

5. UEFI Criticism

- Spec sometimes not specific enough
- Implementations are often buggy (derived from EDK2)
- Overly complex and inconsistent
- Most vendors add closed-source additions
- Tries to be a modern OS-like environment but sticks to decade old concepts
  - A single global address space for everything
  - No real multitasking

5. UEFI Criticism

- Spec sometimes not specific enough
- Implementations are often buggy (derived from EDK2)
- Overly complex and inconsistent
- Most vendors add closed-source additions
- Tries to be a modern OS-like environment but sticks to decade old concepts
  - A single global address space for everything
  - No real multitasking
  - Limited error handling and debugging

Thank you for your attention!