OREBOOT

Ron Minnich
Ryan O'Leary
Gan Shun Lim
Prachi Laud
Chris Koch
Ian Goegebuer

With thanks to:
Andres Richter, Rust Embedded WG

In this talk...

1. What is Oreboot?
2. Firmware Challenges
3. Oreboot Design
4. Rust Challenges
5. Targets
6. Getting involved

fig 1. Oreboot developers in their natural habitat

1. What is Oreboot?

Open-Source Firmware Projects (An incomplete history)

- U-boot (1999-)
- LinuxBIOS (1999-2008)
- Coreboot (2008-)
- NERF (2016-)
- Linuxboot (2017-)
- u-bmc (2018-)
- SlimBoot (2018-) [sort of, it’s a UEFI DXE]

Oreboot = Coreboot - "C"
And much more!

- Downstream fork of coreboot
  - Open-source and GPLv2
- Rust
  - Absolutely no C code.
  - Small pieces of assembly where necessary (ex: initializing stack pointer)
  - Coreboot assembly code is very useful for these tricky bits
- Jump to kernel as quickly as possible
  - Firmware contains no network stack, disk drivers, debug shells, ...
  - Those features are provided by payloads such as LinuxBoot
- Strict policy for accepting closed-source blobs
  - Only an issue for the x86 port
- Current RISC-V ports are fully open-source

2. Firmware Challenges

Simple View of Firmware

Three jobs:

1. Initialize hardware (CPU, Buses, Memory, ...)
2. Select and run boot media
3. Provide runtime services (optional)

Oreboot Bootflow

1. Boot Blob	2. Rom Stage	3. Payloader Stage	4. LinuxBoot	5. YOUR Software
• Executes directly from flash • First instruction • Initialize CPU • Debug UART print “Welcome to Oreboot” • Setup SRAM/CAR • Find and jump to Rom Stage • A fair bit of assembly code	• Executes directly from flash • Has very little ~30KiB-8MiB of SRAM/CAR. Not enough for Linux yet! • Initialize RAM	• Has one job • Find, load and run a payload • The payloader has no drivers, storage drivers, USB stack, etc… This is a big complexity reduction compared to your classic coreboot.	• Linux + Initramfs (or another kernel of your choosing) • Kernel (not oreboot) can optionally load another kernel from the disk or network and *kexec*	

One Second Boot: The Holy Grail

- Firmware bloat epidemic:
  - Consumer laptops/desktops take minutes to boot
  - Servers taking 10+ minutes to boot
  - BMCs taking almost 1 minute to boot, in serial with the host
- Counterpoint:
  - Chromebooks can boot in seconds
  - 2.4Ghz x86 server nodes could boot in seconds *in 2004*
  - Linux-based automobile computers have held to 800ms since 2006
- Fix pain points:
  - Memory training. This can be cached and is easy to do if reference code is open-source.
  - Run drivers and probe devices concurrently. See coroutines slide.
  - If boot is 1s, need not waste time loading a splash screen, progress bar, video driver, fonts, …
  - Defer all network and disk access to Linux à la LinuxBoot. Decades have gone into optimization its disk and network drivers.
- Our goal: Boot AST2500 (a BMC) in <1s.

3. Oreboot Design

Driver Model

string error messages

end-of-file returned when reached the end of a “block device”

```
pub type Result<T> = core::result::Result<T, &'static str>;
pub const EOF: Result<usize> = Err("EOF");
pub const NOT_IMPLEMENTED: Result<usize> = Err("not implemented");

pub trait Driver {
	/// Initialize the device.
	fn init(&mut self) -> Result<()>;
	/// Positional read. Returns number of bytes read.
	fn pread(&self, data: &mut [u8], pos: usize) -> Result<usize>;
	/// Positional write. Returns number of bytes written.
	fn pwrite(&mut self, data: &[u8], pos: usize) -> Result<usize>;
	/// Shutdown the device.
	fn shutdown(&mut self);
}
```

no cursor

for “char devices”, pos doesn’t matter

driver model works without memory allocation

Example Block Device

100 byte block device

32 byte block
32 byte block
32 byte block
4 byte block

pread(&mut buffer, 0) -> 32

pread(&mut buffer, 32) -> 32

pread(&mut buffer, 64) -> 32

pread(&mut buffer, 96) -> 4

pread(&mut buffer, 100) -> EOF

32 byte buffer

Driver Model

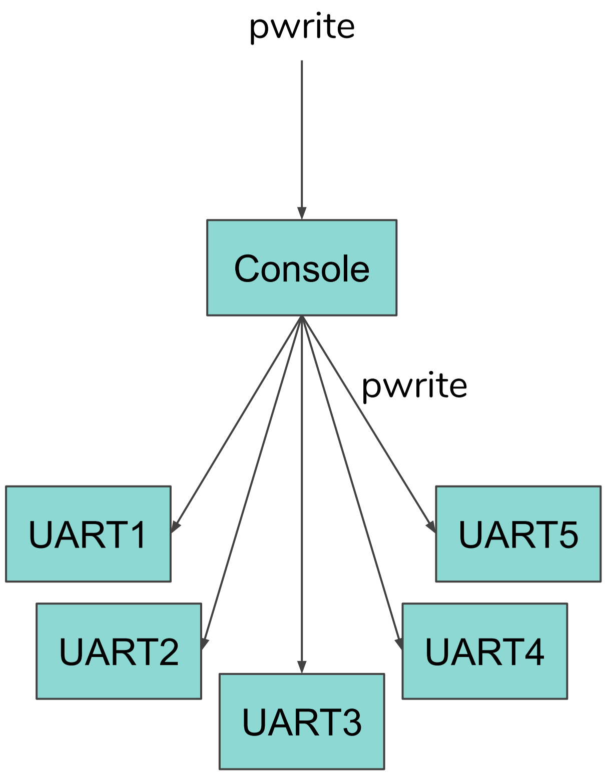
Physical Drivers	
Name	**Description**
Memory	Reads/writes to physical memory addresses
PL011	Reads/writes to serial
NS16550	Reads/writes to serial
MMU	Control MMU. Making it a driver is unique to oreboot.
sifive/spi	Read from SiFive SPI master

Virtual Drivers	
Name	**Description**
Union	Writes are duplicated to each driver in a slice of drivers, &[&mut dyn Driver]
SliceReader	Reads from a slice, &[u8]
SectionReader	Reads from a window (offset&size) of a another Driver. Returns EOF when the end of the window is reached.

Example Serial Device

```
let mut uarts = [
    &mut NS16550::new(0x1E78_3000, 115200) as &mut dyn Driver, // UART1
    &mut NS16550::new(0x1E78_D000, 115200) as &mut dyn Driver, // UART2
    &mut NS16550::new(0x1E78_E000, 115200) as &mut dyn Driver, // UART3
    &mut NS16550::new(0x1E78_F000, 115200) as &mut dyn Driver, // UART4
    &mut NS16550::new(0x1E78_4000, 115200) as &mut dyn Driver, // UART5
];
let console = &mut Union::new(&mut uarts[..]);
console.init();
console.pwrite(b"Welcome to oreboot\r\n", 0).unwrap();

let w = &mut print::WriteTo::new(console);
fmt::write(w, format_args!("{} {}\r\n", "Formatted output:", 7)).unwrap();
```



- Get printf for free!
- Easy add/remove and configure new drivers.
- Unsafe problem: If two separate modules initialize the same NS16550 driver with the same mmio address, they will conflict. The driver does not “own” the underlying mmio address.

HELP WANTED

Flash Layout

Example layout for a 16MiB flash part

Boot Blob 500KiB
Fixed DTFS 500KiB
NVRAM A + B 500KiB + 500KiB
RomPayload DTFS A + B 1MiB + 1MiB
RamPayload DTFS A + B 6MiB + 6MiB

- CBFS (coreboot file system) replaced with DTFS
- DTFS = Device Tree File System
  - Can be parsed by existing OSes without any modification. See /sys/firmware/dt/...
  - Firmware can expose layout of flash chip without any extra OS code.
  - Easy to parse
  - Self describing
  - Can also be used for:
    - Metadata
    - Splash screens

Fixed DTFS

```
/dts-v1/;

/ {
    #address-cells = <1>;
    #size-cells = <1>;

    flash-info {
        compatible = "ore-flashinfo";
        board-name = "HiFive Unleashed";
        category = "SiFive";
        board-url =
"https://www.sifive.com/boards/hifive-unleashed";
        areas {
            area@0 {
                description = "Boot Blob and
Ramstage";
                offset = <0x0>;
                size = <0x80000>; // 512KiB
                file =
"target/riscv64imac-unknown-none-elf/debug/boot
blob.bin";
            };
```

```
            area@1 {
                description = "Fixed DTFS";
                offset = <0x80000>;
                size = <0x80000>; // 512KiB
                file =
"target/riscv64imac-unknown-none-elf/debug/fix
ed-dtfs.dtb";
            };
            area@2 {
                description = "Payload A";
                offset = <0x100000>;
                size = <0x600000>; // 6MiB
                file = "payloadA";
            };
            area@3 {
                description = "Payload B";
                offset = <0x700000>;
                size = <0x600000>; // 6MiB
                file = "payloadB";
            };
```

```
            area@4 {
                description = "Payload C";
                offset = <0xd00000>;
                size = <0x300000>; // 3MiB
                file = "payloadC";
            };
            area@5 {
                description = "Empty Space";
                offset = <0x1000000>;
                size = <0x1000000>; // 16MiB
            };
        };
    };
};
```

4. Rust Challenges

Source Organization

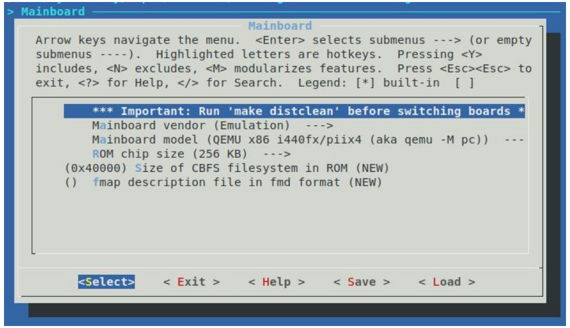
- README.md
- src/mainboard/opentitan/crb/{Makefile.toml, Cargo.toml}
- src/mainboard/opentitan/crb/src/*.{rs,S}
- src/cpu/lowrisc/ibex/Carbo.toml
- src/cpu/lowrisc/ibex/*.rs
- src/soc/opentitan/src/Cargo.toml
- src/soc/opentitan/src/*.rs
- src/drivers/Cargo.toml
- src/drivers/src/*.rs
- payloads/Cargo.toml
- payloads/src/*.rs
- ...

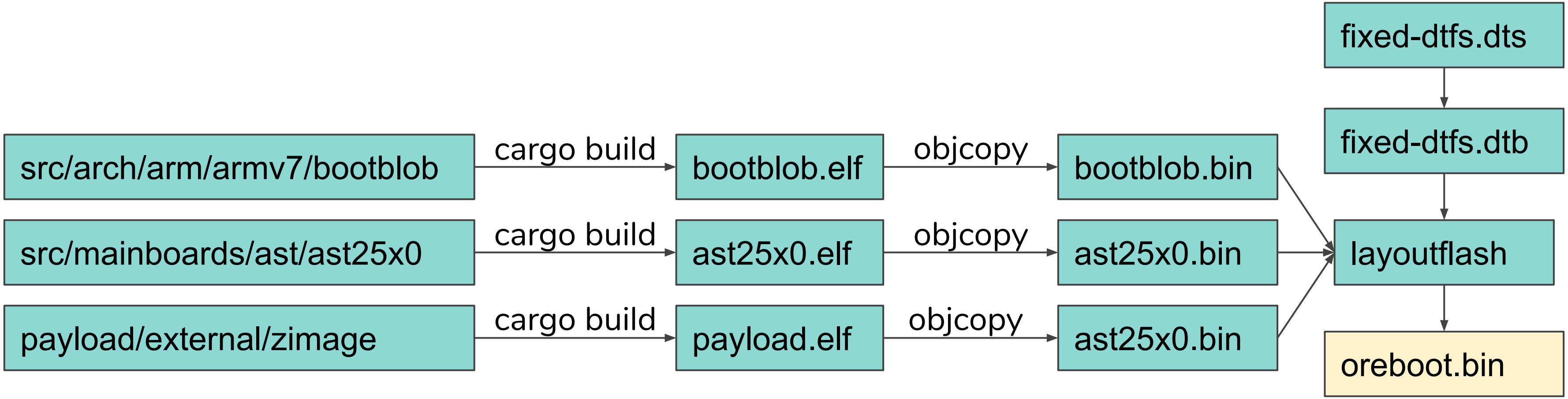
Contains multiple, conditionally compiled modules

```
#[cfg(feature = "ns16550")]
pub mod ns16550;
#[cfg(feature = "pl011")]
pub mod pl011;
```

cargo-make and user-configuration

- Oreboot has a few post-build steps
  - Build with “cargo make” / Makefile.toml
- Configuration
  - Coreboot uses “make menuconfig” / KConfig
  - No such system for Cargo
  - Currently, oreboot is using conditional compilation / cfg





No dynamic allocation

- No dynamic allocation until memory is initialized
- All memory is stack-allocated
- Want to guarantee that stack size is less than SRAM/CAR (ex: 36KiB) at build time.
    - Use tools such as cargo-call-stack to determine stack size.

HELP WANTED

HELP WANTED

HELP WANTED

Coroutines

- Polling I/O is very slow
  - A few UART prints = 0.01s
  - Read from SPI and verify loop
- Interrupt-based I/O is difficult to do well
  - Puts us on slippery slope to becoming a kernel
- Non-preemptive threading has been shown to be “good enough” in firmware
- Implementation details
  - Save state
  - Long jump
  - (simple) Scheduler -- round robin has been shown to be good enough
- Coreboot had threading and ||ism support off and on over the last 20 years
  - It was always so tricky to use it was usually unused/removed
  - Seems like rust could make this easier and safer

HELP WANTED

HELP WANTED

HELP WANTED

HELP WANTED

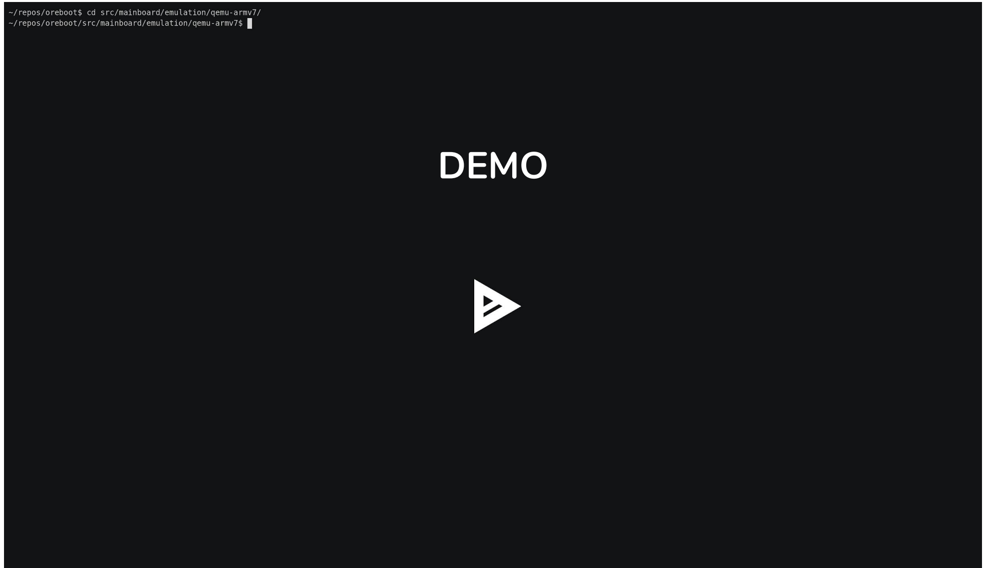
HELP WANTED

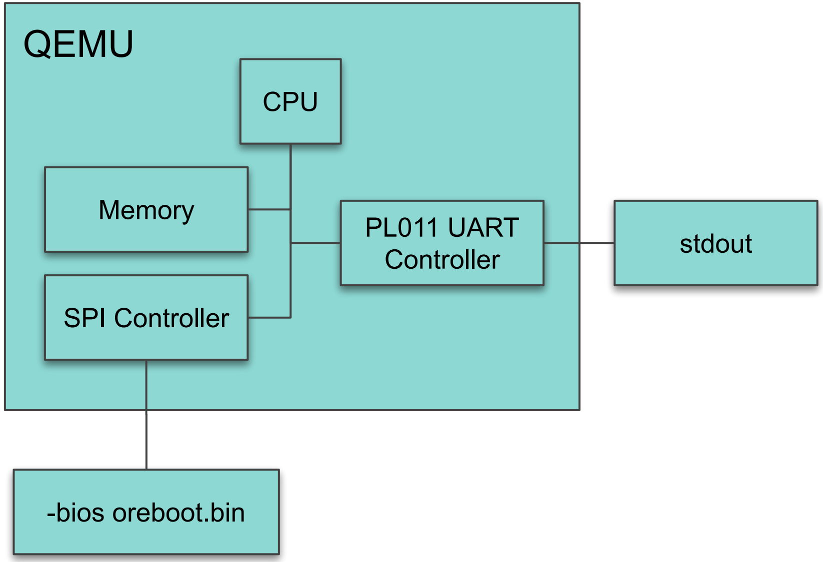
5. Targets

First Target: qemu-system-arm

- -machine virt
- Two day effort (thanks Rust!)
- Memory already initialized
- Device tree is currently hard-coded

HELP WANTED

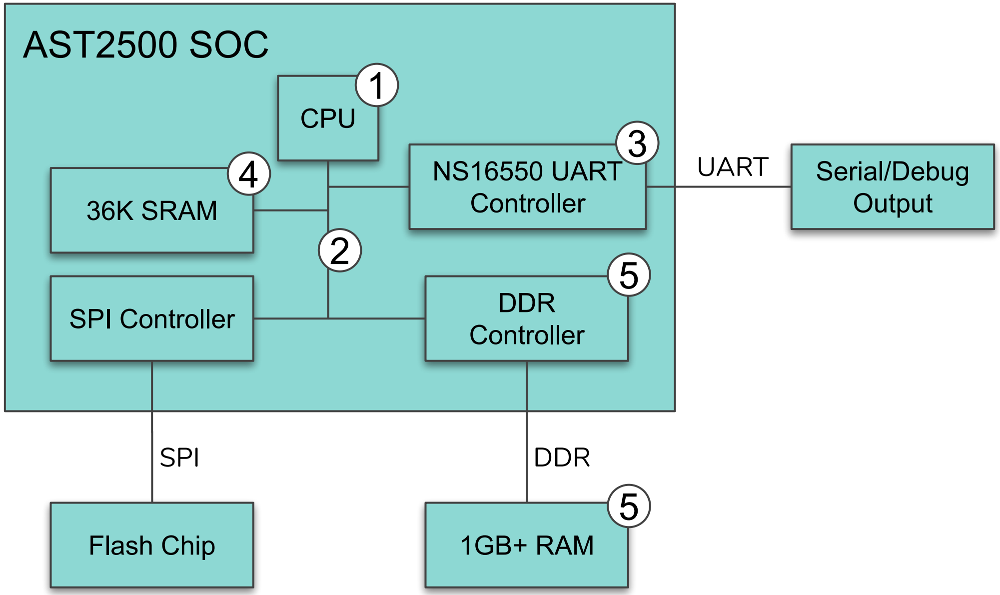




RUST_TARGET_PATH=$(pwd) cargo make run -p release

HELP WANTED

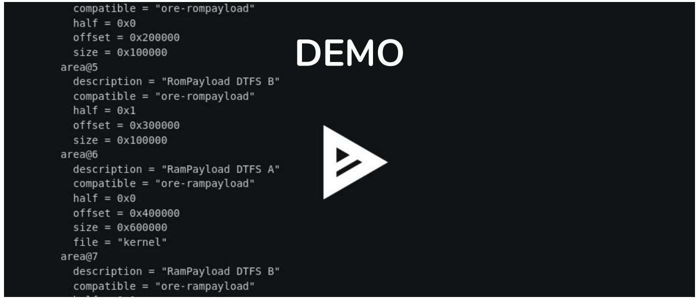
First Hardware Target: AST2500

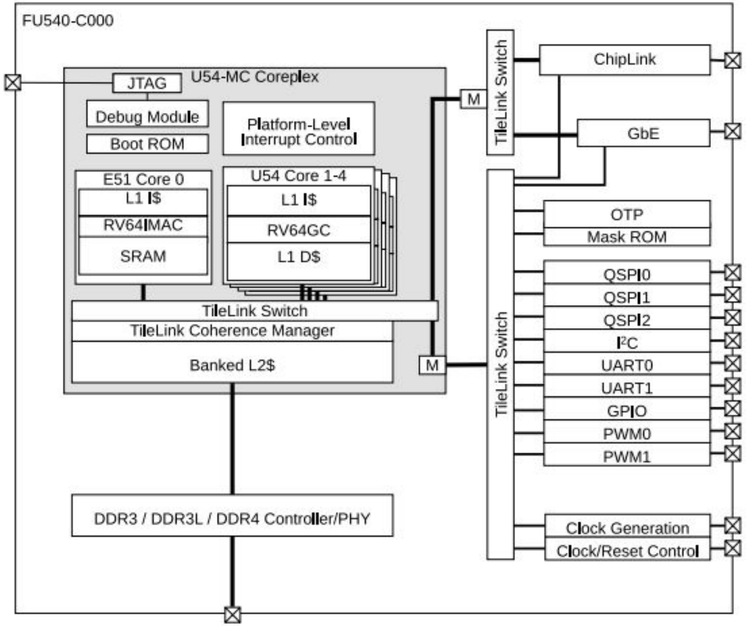


- ARM11
- BMC Platform
- Open-source memory initialization code https://github.com/u-root/u-bmc/blob/master/platform/ast2500/boot/platform_g5.S
  - Converted to .rs with a Go program
- Bootblob
  - 1. Initialize CPU
  - 2. Initialize Bus
  - 3. Initial UART Print
  - 4. Initialize SRAM
- Romstage
  - 5. Memory init

Second Target: HiFive Unleashed (FU540)

- RV64GC, 4 harts
- Linux capable
- Prefer to run Linux in M-Mode with no MMU
  - Patches from Christoph Hellwig
  - CONFIG_RISCV_M_MODE=y
- Memory init is open-source and implemented in native Rust code!





Third Target: OpenTitan earlgrey

- RISC-V rv32imc
- Embedded, not Linux capable
- Open-source design for a root-of-trust
- https://github.com/lowRISC/opentitan
- No ASIC available yet
- Oreboot current runs on:
  - Verilator
  - FPGA (Artix-7)
- Currently trying to boot Tock kernel (https://www.tockos.org/)

x86

Ground Rules for x86

1. We prefer all pieces of the firmware to be open-source; but can accept an ME and FSP binary blob for x86 architectures.
2. Blobs must be essential to boot the system and not provide any extraneous functionality which could not be implemented in Oreboot.
3. Blobs must be redistributable and are ideally available on GitHub.
4. Blobs must not be submitted to github.com/oreboot/oreboot. We prefer blobs to be submitted to github.com/oreboot/blobs, github.com/coreboot/blobs or some other GitHub repository.
5. The blobs must be in a binary format. No additional C code, assembly files or header files are acceptable.
6. Any compromises to the language safety features of Rust must be explicitly stated.

As a "measure" for how open-source firmware is, use the percentage of the final binary size. For example, if 70% of the firmware bytes are closed-source blob and 30% built from Oreboot source code, we would say the firmware is 30% open-source.

Getting Involved

Join the Discussion

http://slack.u-root.com/

Join the oreboot channel

Github

https://github.com/oreboot/oreboot

Help Wanted

- Improve CI system
- OpenTitan Port
- HiFive Port
- SPI and other drivers
- Security and vboot
- …

HELP
WANTED