oreboot on RISC-V

Comparing Implementations on Two Platforms

Daniel Maslowski

Agenda

- Introduction
- Allwinner D1
- StarFive JH7100
- Approaches and Future Work

What is oreboot again?

What is oreboot again?

oreboot is a fork of coreboot…

What is oreboot again?

oreboot is a fork of coreboot...

coreboot

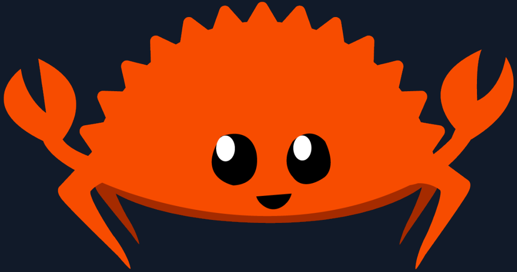
OREBOOT

```
From aa246f71de7f9900a30d938ab618c46839436616 [...]
From: "Ronald G. Minnich" <rminnich@gmail.com>
Date: Mon, 1 Apr 2019 23:48:56 +0000
Subject: [PATCH] Initial removal of C code
```

https://github.com/oreboot/oreboot

Firmware in Rust

oreboot is a fork of coreboot, with C removed, written in Rust.



Rust logo under CC BY 4.0, https://github.com/rust-lang/rust-artwork

Ferris the crab from https://rustacean.net/

Firmware Development

Firmware Development

feels like an RPG

You need to figure out how things work.

Firmware Development

feels like an RPG

You need to figure out how things work.

point on map = program counter (PC)

Sometimes you have no clue where you are.

Firmware Development

feels like an RPG

You need to figure out how things work.

point on map = program counter (PC)

Sometimes you have no clue where you are.

world map = memory map

The islands or worlds are the peripherals.

Firmware Development

feels like an RPG

You need to figure out how things work.

point on map = program counter (PC)

Sometimes you have no clue where you are.

world map = memory map

The islands or worlds are the peripherals.

player’s guide = processor/SoC manual

It may be incomplete or not at all available (at least to you).

Firmware Development

feels like an RPG

You need to figure out how things work.

point on map = program counter (PC)

Sometimes you have no clue where you are.

world map = memory map

The islands or worlds are the peripherals.

player’s guide = processor/SoC manual

It may be incomplete or not at all available (at least to you).

Internet of Things = MMORPG

Yes, it can get very dangerous.

Single Board Computers

Single Board Computers

Many are *marketed* as open source. Are they though?

Single Board Computers

Many are *marketed* as open source. Are they though?

Documentation

- *schematics* and board design
- manuals and instructions
- open *license*

open source hardware

Single Board Computers

Many are *marketed* as open source. Are they though?

Documentation

- *schematics* and board design
- manuals and instructions
- open *license*

Source Code

- *open* tools for flashing, debugging and image composition
- firmware, *from the start*, documented (U-Boot, oreboot, …)
- Linux or other OS, *mainline friendly* (git fork, *not* source dump)
- all code usable with upstream toolchains, or provide toolchains in a *reproducible* form (not only binaries for a specific architecture/OS)

Single Board Computers

Many are *marketed* as open source. Are they though?

Documentation

- *schematics* and board design
- manuals and instructions
- open *license*

Source Code

- *open* tools for flashing, debugging and image composition
- firmware, *from the start*, documented (U-Boot, oreboot, ...)
- Linux or other OS, *mainline friendly* (git fork, *not* source dump)
- all code usable with upstream toolchains, or provide toolchains in a *reproducible* form (not only binaries for a specific architecture/OS)

OSHWA Certification: https://certification.oshwa.org/

Let’s look at a manual and a memory map!

Allwinner D1

D1 SoC

D1 SoC

Production

- widely produced and easily available
- many different boards from various vendors

D1 SoC

Production

- widely produced and easily available
- many different boards from various vendors

Cores

- 1x C906 core, 1GHz
  - https://github.com/T-head-Semi/openc906
- 1x low-power core, Xtensa HiFi4

D1 SoC

Production

- widely produced and easily available
- many different boards from various vendors

Cores

- 1x C906 core, 1GHz
  - https://github.com/T-head-Semi/openc906
- 1x low-power core, Xtensa HiFi4

Documentation

- larger manual provided
  - about 1400 pages
- DRAM controller and HDMI missing

D1 Boards and SoMs

Allwinner Nezha
first board; Raspberry Pi form factor

DongshanPi Nezha STU
- SoM in custom form factor
- carrier board with many pins

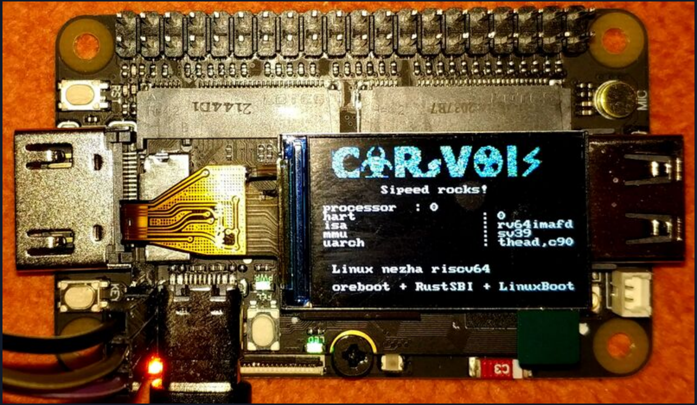
Sipeed Lichee RV
SoM and multiple carrier boards

ClockworkPi R01
- SoM in RPi CM 3 form factor
- DevTerm carrier board + case

MangoPi MQ-Pro
Raspberry Pi Zero form factor, drop-in replacement

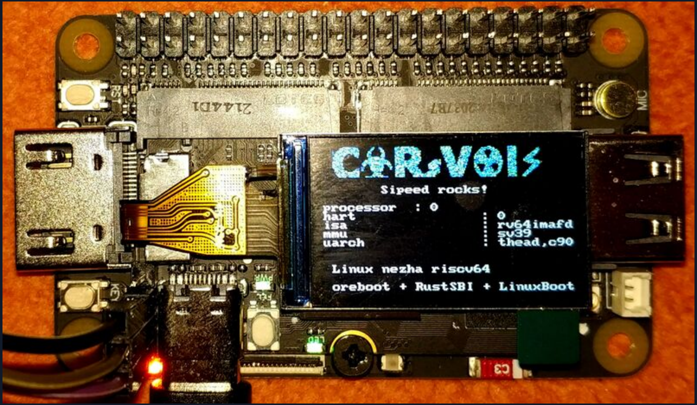
https://linux-sunxi.org/Category:D1_Boards

D1 Boards: Lichee RV + Dock



```
processor   : 0
hart        : 0
isa         : rv64imafd
mmu         : sv39
uarch       : thead,c90

Linux nezha riscv64
oreboot + RustSBI + LinuxBoot
```

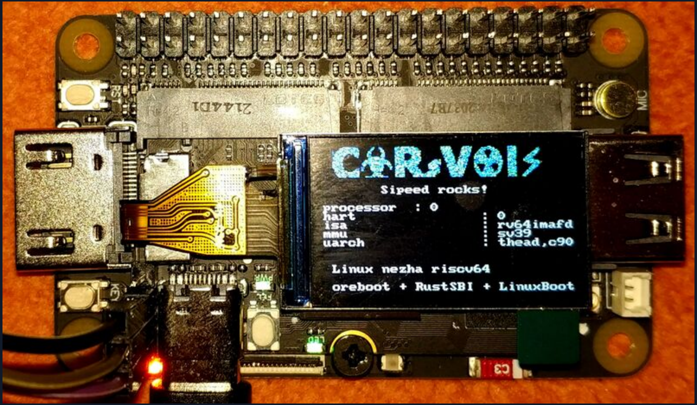
D1 Boards: Lichee RV + Dock



```
Sipeed rocks!
processor     : 0
hart          : 0
isa           : rv64imafd
mmu           : sv39
uarch         : thead,c90

Linux nezha riscv64
oreboot + RustSBI + LinuxBoot
```

The regular Dock has solder joints for a SPI flash, so I added one.

D1 Boards: Lichee RV + Dock



```
Sipeed rocks!
processor   : 0
hart        : 0
isa         : rv64imafd
mmu         : sv39
uarch       : thead,c90

Linux nezha riscv64
oreboot + RustSBI + LinuxBoot
```

The regular Dock has solder joints for a SPI flash, so I added one.

The Dock Pro already has a 16 MiB SPI flash, plus a USB serial converter.

D1 Boot Behavior

Mask ROM

D1 Boot Behavior

Mask ROM

It loads a blob from SPI flash, eMMC or SD card into SRAM (32K).

D1 Boot Behavior

Mask ROM

It loads a blob from SPI flash, eMMC or SD card into SRAM (32K).

The blob has to start with a specific `eGON` header:

D1 Boot Behavior

Mask ROM

It loads a blob from SPI flash, eMMC or SD card into SRAM (32K).
The blob has to start with a specific `eGON` header:

```
Disassembly of section .head:

0000000000020000 <head_jump>:
   20000: a5 a0          j       0x20068 <start+0x8>
   20002: 00 00          unimp

0000000000020004 <_ZN17oreboot_nezha_bt09EGON_HEAD17h5aa4b41b712905f
   20004: 65 47 4f 4e 2e 42 54 30        eGON.BT0
   2000c: 39 6c 0a 5f 00 00 00 00        9l._....
```

Note: The header is not documented in the manual.

D1 Boot Behavior

Mask ROM

It loads a blob from SPI flash, eMMC or SD card into SRAM (32K).
The blob has to start with a specific `eGON` header:

```
Disassembly of section .head:

0000000000020000 <head_jump>:
   20000: a5 a0          j       0x20068 <start+0x8>
   20002: 00 00          unimp

0000000000020004 <_ZN17oreboot_nezha_bt09EGON_HEAD17h5aa4b41b712905f
   20004: 65 47 4f 4e 2e 42 54 30         eGON.BT0
   2000c: 39 6c 0a 5f 00 00 00 00         9l._....
```

Note: The header is not documented in the manual.

SPI flash

We need to actively read from SPI flash and have no MMIO access to it.

D1 oreboot Flow

D1 oreboot Flow

SRAM Stage

Boot from flash and DRAM init were prototyped separately:
https://github.com/luojia65/test-d1-flash-bare/

D1 oreboot Flow

SRAM Stage

Boot from flash and DRAM init were prototyped separately:
https://github.com/luojia65/test-d1-flash-bare/
It has been copied and developed further in oreboot:
`src/mainboard/sunxi/nezha/bt0`

D1 oreboot Flow

SRAM Stage

Boot from flash and DRAM init were prototyped separately:
https://github.com/luojia65/test-d1-flash-bare/
It has been copied and developed further in oreboot:
`src/mainboard/sunxi/nezha/bt0`

Payloader Stage

D1 oreboot Flow

SRAM Stage

Boot from flash and DRAM init were prototyped separately:
https://github.com/luojia65/test-d1-flash-bare/
It has been copied and developed further in oreboot:
`src/mainboard/sunxi/nezha/bt0`

Payloader Stage

First RustSBI implementation in oreboot is for Allwinner Nezha (D1):
`src/mainboard/sunxi/nezha/main`

D1 DRAM init

D1 DRAM init

Remember: The DRAM controller is not documented in the manual, only its existence.

D1 DRAM init

Remember: The DRAM controller is not documented in the manual, only its existence.

- a bit less than 2000 lines of code
  - translated from C, which was translated from assembly dump

D1 DRAM init

Remember: The DRAM controller is not documented in the manual, only its existence.

- a bit less than 2000 lines of code
  - translated from C, which was translated from assembly dump

There are configurations per board. They could technically be determined at runtime.

D1 DRAM init

Remember: The DRAM controller is not documented in the manual, only its existence.

- a bit less than 2000 lines of code
  - translated from C, which was translated from assembly dump

There are configurations per board. They could technically be determined at runtime.

- community has knowledge on how it works
  - some documentation on https://linux-sunxi.org

D1 DRAM init

Remember: The DRAM controller is not documented in the manual, only its existence.

- a bit less than 2000 lines of code
  - translated from C, which was translated from assembly dump

There are configurations per board. They could technically be determined at runtime.

- community has knowledge on how it works
  - some documentation on https://linux-sunxi.org

I tried my best to name registers; reviews and help wanted!

We may be able to reuse at least parts, and apply them to other Allwinner SoCs.

D1 Development Status

D1 Development Status

- avilable in `main` branch
- boot from SPI flash
- SD card work in progress
- SBI is optional
- multiple boards supported

D1 Development Status

- avilable in `main` branch
- boot from SPI flash
- SD card work in progress
- SBI is optional
- multiple boards supported

We can boot

- Linux \o/
- xv6
- MnemOS
- r9

more to come: FreeBSD, Illumos…

D1 Development Status

- avilable in `main` branch
- boot from SPI flash
- SD card work in progress
- SBI is optional
- multiple boards supported

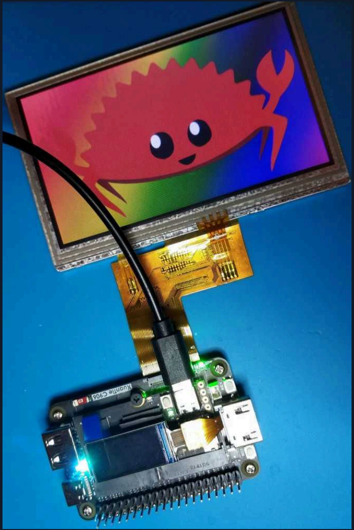
We can boot

- Linux \o/
- xv6
- MnemOS
- r9

more to come: FreeBSD, Illumos…

Bare metal testing app

https://github.com/adamgreig/d1rgb



StarFive JH7100

JH7100 SoC

JH7100 SoC

Production

no longer produced, *but* the successor JH7110 SoC appears to be using the same DRAM controller and parts, judging from very similar vendor code

JH7100 SoC

Production

no longer produced, *but* the successor JH7110 SoC appears to be using the same DRAM controller and parts, judging from very similar vendor code

Cores

- 2x U74 core, >1GHz
  - https://sifive.cdn.prismic.io/sifive/ad5577a0-9a00-45c9-a5d0-424a3d586060_u74_core_complex_manual_21G3.pdf
- 1x VP6

JH7100 SoC

Production

no longer produced, *but* the successor JH7110 SoC appears to be using the same DRAM controller and parts, judging from very similar vendor code

Cores

- 2x U74 core, >1GHz
  - https://sifive.cdn.prismic.io/sifive/ad5577a0-9a00-45c9-a5d0-424a3d586060_u74_core_complex_manual_21G3.pdf
- 1x VP6

Documentation

- no full manual publicly available
- instructions for firmware recovery with open tools
- sparse datasheet with list of peripheral blocks and suppliers (p23)
  - less than 140 pages

https://github.com/starfive-tech/JH7100_docs

JH7100 Boards

JH7100 Boards

BeagleV

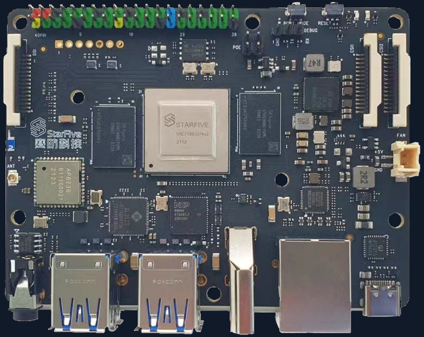
only a select few people received the board as a prototype

JH7100 Boards

BeagleV

only a select few people received the board as a prototype

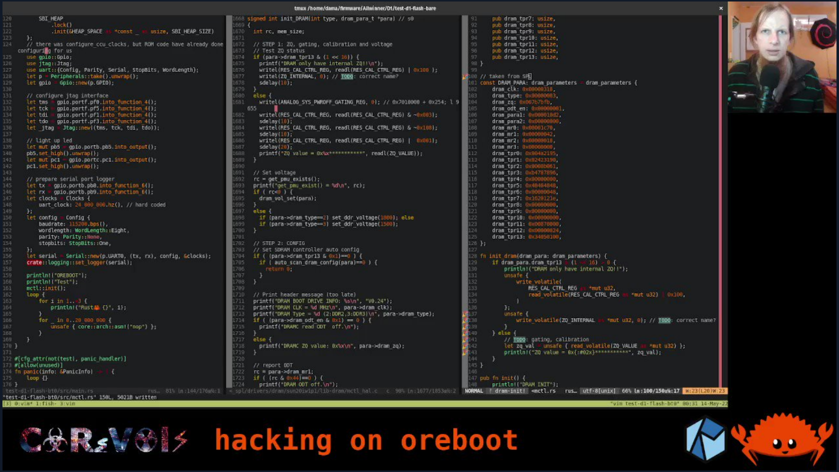
StarFive VisionFive 1



obtained via RISC-V International developer program
https://riscv.org/risc-v-developer-boards/details/

JH7100 Development Stream

Live: https://twitch.tv/cyrevolt



tmux /home/dama/firmware/Allwinner/D1/test-d1-flash-bare

```rust
        SBI_HEAP
            .lock()
            .init(&HEAP_SPACE as *const _ as usize, SBI_HEAP_SIZE)
    };
    // there was configure_ccu_clocks, but ROM code have already done
 configuring for us
    use gpio::Gpio;
    use jtag::Jtag;
    use uart::{Config, Parity, Serial, StopBits, WordLength};
    let p = Peripherals::take().unwrap();
    let gpio = Gpio::new(p.GPIO);

    // configure jtag interface
    let tms = gpio.portf.pf0.into_function_4();
    let tck = gpio.portf.pf5.into_function_4();
    let tdi = gpio.portf.pf1.into_function_4();
    let tdo = gpio.portf.pf3.into_function_4();
    let _jtag = Jtag::new((tms, tck, tdi, tdo));

    // light up led
    let mut pb5 = gpio.portb.pb5.into_output();
    pb5.set_high().unwrap();
    let mut pc1 = gpio.portc.pc1.into_output();
    pc1.set_high().unwrap();

    // prepare serial port logger
    let tx = gpio.portb.pb8.into_function_6();
    let rx = gpio.portb.pb9.into_function_6();
    let clocks = Clocks {
        uart_clock: 24_000_000.hz(), // hard coded
    };
    let config = Config {
        baudrate: 115200.bps(),
        wordlength: WordLength::Eight,
        parity: Parity::None,
        stopbits: StopBits::One,
    };
    let serial = Serial::new(p.UART0, (tx, rx), config, &clocks);
    crate::logging::set_logger(serial);

    println!("OREBOOT");
    println!("Test");
    mctl::init();
    loop {
        for i in 1..=3 {
            println!("Rust🦀 {}", i);
        }
        for _ in 0..20_000_000 {
            unsafe { core::arch::asm!("nop") };
        }
    }
}

#[cfg_attr(not(test), panic_handler)]
#[allow(unused)]
fn panic(info: &PanicInfo) -> ! {
    loop {}
}
```

```
test-d1-flash-bt0/src/main.rs        rus… 81% ln:144/176≡%:1
"test-d1-flash-bt0/src/mctl.rs" 150L, 5021B written
[3] 0:vim* 1:fish- 3:vim
```

```c
signed int init_DRAM(int type, dram_para_t *para) // s0
{
  int rc, mem_size;

  // STEP 1: ZQ, gating, calibration and voltage
  // Test ZQ status
  if (para->dram_tpr13 & (1 << 16)) {
    printf("DRAM only have internal ZQ!!\n");
    writel(RES_CAL_CTRL_REG, readl(RES_CAL_CTRL_REG) | 0x100 );
    writel(ZQ_INTERNAL, 0); // TODO: correct name?
    sdelay(10);
  }
  else {
    writel(ANALOG_SYS_PWROFF_GATING_REG, 0); // 0x7010000 + 0x254; l 9
655
    writel(RES_CAL_CTRL_REG, readl(RES_CAL_CTRL_REG) & ~0x003);
    sdelay(10);
    writel(RES_CAL_CTRL_REG, readl(RES_CAL_CTRL_REG) & ~0x100);
    sdelay(10);
    writel(RES_CAL_CTRL_REG, readl(RES_CAL_CTRL_REG) |  0x001);
    sdelay(20);
    printf("ZQ value = 0X%x***********", readl(ZQ_VALUE));
  }

  // Set voltage
  rc = get_pmu_exists();
  printf("get_pmu_exist() = %d\n", rc);
  if ( rc<0 ) {
    dram_vol_set(para);
  }
  else {
    if (para->dram_type==2) set_ddr_voltage(1800); else
    if (para->dram_type==3) set_ddr_voltage(1500);
  }

  // STEP 2: CONFIG
  // Set SDRAM controller auto config
  if ( (para->dram_tpr13 & 0x1)==0 ) {
    if ( auto_scan_dram_config(para)==0 ) {
      return 0;
    }
  }

  // Print header message (too late)
  printf("DRAM BOOT DRIVE INFO: %s\n", "V0.24");
  printf("DRAM CLK = %d MHz\n", para->dram_clk);
  printf("DRAM Type = %d (2:DDR2,3:DDR3)\n", para->dram_type);
  if ( (para->dram_odt_en & 0x1) == 0 ) {
    printf("DRAMC read ODT  off.\n");
  }
  else {
    printf("DRAMC ZQ value: 0x%x\n", para->dram_zq);
  }

  // report ODT
  rc = para->dram_mr1;
  if ( (rc & 0x44)==0 ) {
    printf("DRAM ODT off.\n");
```

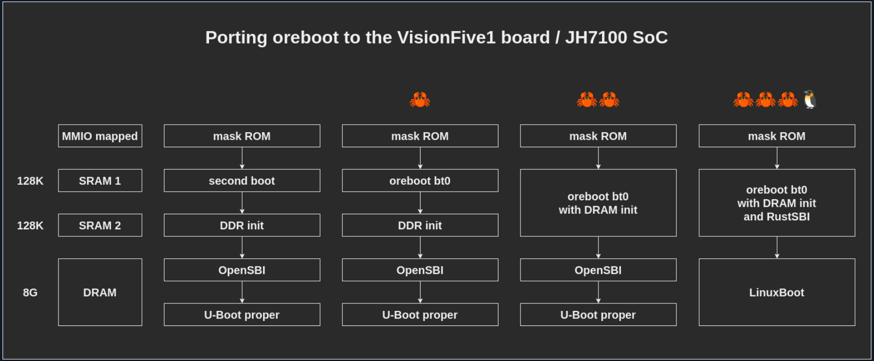
```
<_spl/drivers/dram/sun20iw1p1/lib-dram/mctl_hal.c   c  90% ln:1677/1853≡%:2
```

```rust
    pub dram_tpr7: usize,
    pub dram_tpr8: usize,
    pub dram_tpr9: usize,
    pub dram_tpr10: usize,
    pub dram_tpr11: usize,
    pub dram_tpr12: usize,
    pub dram_tpr13: usize,
}

// taken from SPL
const DRAM_PARA: dram_parameters = dram_parameters {
    dram_clk: 0x00000318,
    dram_type: 0x00000003,
    dram_zq: 0x007b7bfb,
    dram_odt_en: 0x00000001,
    dram_para1: 0x000010d2,
    dram_para2: 0x00000000,
    dram_mr0: 0x00001c70,
    dram_mr1: 0x00000042,
    dram_mr2: 0x00000018,
    dram_mr3: 0x00000000,
    dram_tpr0: 0x004a2195,
    dram_tpr1: 0x02423190,
    dram_tpr2: 0x0008b061,
    dram_tpr3: 0xb4787896,
    dram_tpr4: 0x00000000,
    dram_tpr5: 0x48484848,
    dram_tpr6: 0x00000048,
    dram_tpr7: 0x1620121e,
    dram_tpr8: 0x00000000,
    dram_tpr9: 0x00000000,
    dram_tpr10: 0x00000000,
    dram_tpr11: 0x00870000,
    dram_tpr12: 0x00000024,
    dram_tpr13: 0x34050100,
};

fn init_dram(dram_para: dram_parameters) {
    if dram_para.dram_tpr13 & (1 << 16) > 0 {
        println!("DRAM only have internal ZQ!!");
        unsafe {
            write_volatile(
                RES_CAL_CTRL_REG as *mut u32,
                read_volatile(RES_CAL_CTRL_REG as *mut u32) | 0x100,
            )
        };
        unsafe {
            write_volatile(ZQ_INTERNAL as *mut u32, 0); // TODO: correct name?
        }
    } else {
        // TODO: gating, calibration
        let zq_val = unsafe { read_volatile(ZQ_VALUE as *mut u32) };
        println!("ZQ value = 0x{:#02x}***********", zq_val);
    }
}

pub fn init() {
    println!("DRAM INIT");
```

```
NORMAL  ! dram-init!  <mctl.rs   rus…  utf-8[unix]  66% ln:100/150≡%:17  W:23(L20)W:23
                                             "vim test-d1-flash-bt0" 00:31 14-May-22
```

Archive: https://www.youtube.com/playlist?list=PLenOHeTI_A9PSGshDnEc4dYK-GSnCshk6

JH7100 Vendor Code and Transition Plan



JH7100 Boot Behavior

JH7100 Boot Behavior

Mask ROM

not documented; can be dumped via preflashed U-Boot

JH7100 Boot Behavior

Mask ROM

not documented; can be dumped via preflashed U-Boot
It loads a blob to SRAM, which has to be prefixed with a 4-byte value for its size.

JH7100 Boot Behavior

Mask ROM

not documented; can be dumped via preflashed U-Boot
It loads a blob to SRAM, which has to be prefixed with a 4-byte value for its size.

SPI flash

JH7100 Boot Behavior

Mask ROM

not documented; can be dumped via preflashed U-Boot

It loads a blob to SRAM, which has to be prefixed with a 4-byte value for its size.

SPI flash

MMIO access to the SPI flash is available, requiring little initialization.

JH7100 Boot Behavior

Mask ROM

not documented; can be dumped via preflashed U-Boot
It loads a blob to SRAM, which has to be prefixed with a 4-byte value for its size.

SPI flash

MMIO access to the SPI flash is available, requiring little initialization.
This means that loading from flash is just like copying from on area in memory to another.

JH7100 Boot Behavior

Mask ROM

not documented; can be dumped via preflashed U-Boot
It loads a blob to SRAM, which has to be prefixed with a 4-byte value for its size.

SPI flash

MMIO access to the SPI flash is available, requiring little initialization.
This means that loading from flash is just like copying from on area in memory to another.

Multicore

Note: Multiple cores allows for accessing peripherals in parallel.

JH7100 Boot Behavior

Mask ROM

not documented; can be dumped via preflashed U-Boot
It loads a blob to SRAM, which has to be prefixed with a 4-byte value for its size.

SPI flash

MMIO access to the SPI flash is available, requiring little initialization.
This means that loading from flash is just like copying from on area in memory to another.

Multicore

Note: Multiple cores allows for accessing peripherals in parallel.
Strategy: loop second hart; when done with peripherals, jump to OS.

DRAM init

DRAM init

We started with a case study, analyzing how the vendor code works.

https://github.com/starfive-tech/JH7100_ddrinit

DRAM init

We started with a case study, analyzing how the vendor code works.

https://github.com/starfive-tech/JH7100_ddrinit

- more than 4000 lines
- quite some registers have comments
- lots of magic values, little logic
- more than 50% is just writing `0`, probably unnecessary

DRAM init

We started with a case study, analyzing how the vendor code works.

https://github.com/starfive-tech/JH7100_ddrinit

- more than 4000 lines
- quite some registers have comments
- lots of magic values, little logic
- more than 50% is just writing `0`, probably unnecessary

We found and fixed a double bug in the vendor code:

writing back to the wrong register

DRAM init

We started with a case study, analyzing how the vendor code works.

https://github.com/starfive-tech/JH7100_ddrinit

- more than 4000 lines
- quite some registers have comments
- lots of magic values, little logic
- more than 50% is just writing `0`, probably unnecessary

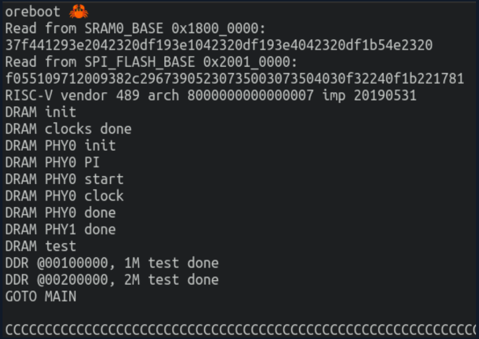
We found and fixed a double bug in the vendor code:

writing back to the wrong register

We reported the issue, with no reply so far.

https://github.com/starfive-tech/JH7100_ddrinit/issues/14

JH7100 Development Status



```
oreboot 🦀
Read from SRAM0_BASE 0x1800_0000:
37f441293e2042320df193e1042320df193e4042320df1b54e2320
Read from SPI_FLASH_BASE 0x2001_0000:
f055109712009382c2967390523073500307350403 0f32240f1b221781
RISC-V vendor 489 arch 8000000000000007 imp 20190531
DRAM init
DRAM clocks done
DRAM PHY0 init
DRAM PHY0 PI
DRAM PHY0 start
DRAM PHY0 clock
DRAM PHY0 done
DRAM PHY1 done
DRAM test
DDR @00100000, 1M test done
DDR @00200000, 2M test done
GOTO MAIN

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

- pull request open with DRAM init and jump to next stage https://github.com/oreboot/oreboot/pull/606
- we currently load and jump to the U-Boot + OpenSBI blob
  - which can then load Linux, e.g., via network

Approaches and Future Work

Talking to peripherals

Talking to peripherals

Register Blocks

Talking to peripherals

Register Blocks

A register block is a set of register that maps to a block in the SoC.

Talking to peripherals

Register Blocks

A register block is a set of register that maps to a block in the SoC.
The start is called the *base register*.

Talking to peripherals

Register Blocks

A register block is a set of register that maps to a block in the SoC. The start is called the *base register*.

```
const CCU_BASE: usize = 0x0200_1000;
const CCU_PLL_PERIO_CTRL: usize = CCU_BASE + 0x0020;
```

Talking to peripherals

Register Blocks

A register block is a set of register that maps to a block in the SoC. The start is called the *base register*.

```
const CCU_BASE: usize = 0x0200_1000;
const CCU_PLL_PERIO_CTRL: usize = CCU_BASE + 0x0020;
```

MMIO (memory-mapped input/output)

Writing to a peripheral register happens through a memory write instruction.

Talking to peripherals

Register Blocks

A register block is a set of register that maps to a block in the SoC.
The start is called the *base register*.

```
const CCU_BASE: usize = 0x0200_1000;
const CCU_PLL_PERIO_CTRL: usize = CCU_BASE + 0x0020;
```

MMIO (memory-mapped input/output)

Writing to a peripheral register happens through a memory write instruction.
To change a value, read first, apply a mask, and write back.

Talking to peripherals

Register Blocks

A register block is a set of register that maps to a block in the SoC.
The start is called the *base register*.

```rust
const CCU_BASE: usize = 0x0200_1000;
const CCU_PLL_PERI0_CTRL: usize = CCU_BASE + 0x0020;
```

MMIO (memory-mapped input/output)

Writing to a peripheral register happens through a memory write instruction.
To change a value, read first, apply a mask, and write back.
Example:

```rust
unsafe {
  let peri0_ctrl = read_volatile(CCU_PLL_PERI0_CTRL as *mut u32);
  let new_val = peri0_ctrl | 1 << 29; // set bit `29`
  write_volatile(CCU_PLL_PERI0_CTRL as *mut u32, new_val);
}
```

SVD -> PAC -> HAL

SVD -> PAC -> HAL

https://github.com/duskmoon314/aw-pac/tree/main/d1-pac

SVD -> PAC -> HAL

https://github.com/duskmoon314/aw-pac/tree/main/d1-pac

https://docs.rs/d1-pac/latest/d1_pac

SVD -> PAC -> HAL

https://github.com/duskmoon314/aw-pac/tree/main/d1-pac

https://docs.rs/d1-pac/latest/d1_pac

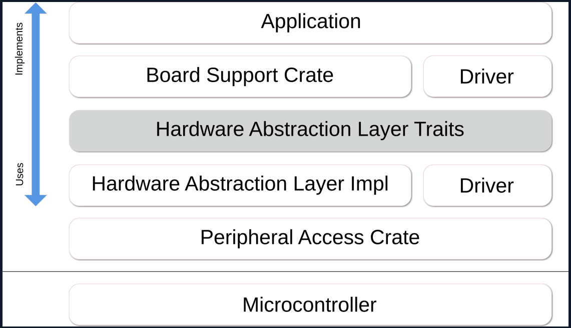


Image under CC BY 4.0

SVD -> PAC -> HAL

https://github.com/duskmoon314/aw-pac/tree/main/d1-pac

https://docs.rs/d1-pac/latest/d1_pac

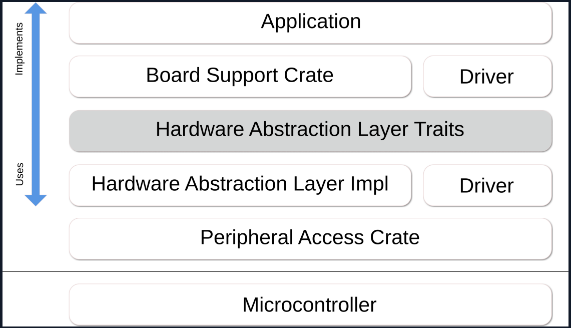


Image under CC BY 4.0

Layers in oreboot

1. App (mainboard)
2. HAL (“drivers”)
3. PAC (if available)

SVD -> PAC -> HAL

https://github.com/duskmoon314/aw-pac/tree/main/d1-pac

https://docs.rs/d1-pac/latest/d1_pac

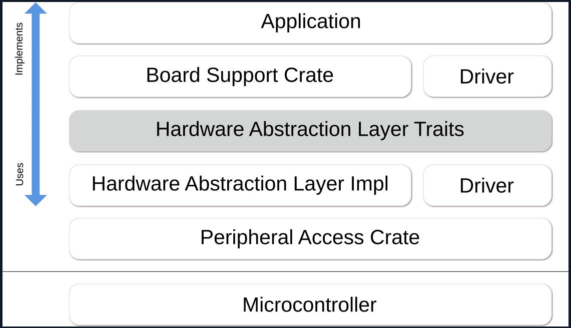


Image under CC BY 4.0

Layers in oreboot

1. App (mainboard)
2. HAL (“drivers”)
3. PAC (if available)

PACs are commonly generated from SVD files.

https://docs.rust-embedded.org/book/portability/index.html

Using a peripheral access crate (PAC)

Using a peripheral access crate (PAC)

Instead of using `write_volatile` directly, we call a *semantic function* from a library:

Using a peripheral access crate (PAC)

Instead of using `write_volatile` directly, we call a *semantic function* from a library:

```
// light up led
let mut pb5 = gpio.portb.pb5.into_output();
pb5.set_high().unwrap();
```

Using a peripheral access crate (PAC)

Instead of using `write_volatile` directly, we call a *semantic function* from a library:

```rust
// light up led
let mut pb5 = gpio.portb.pb5.into_output();
pb5.set_high().unwrap();
```

Depending on the API, we may need to use a writer interface and pass a function:

Using a peripheral access crate (PAC)

Instead of using `write_volatile` directly, we call a *semantic function* from a library:

```
// light up led
let mut pb5 = gpio.portb.pb5.into_output();
pb5.set_high().unwrap();
```

Depending on the API, we may need to use a writer interface and pass a function:

```
ccu.smhc0_clk.write(|w| w.clk_src_sel().pll_peri_1x());
```

RISC-V Runtime Services

RISC-V Runtime Services

Runtime Services are listed in **platform specs**, referencing the **SBI spec**.
https://github.com/riscv/riscv-platform-specs

RISC-V Runtime Services

Runtime Services are listed in **platform specs**, referencing the **SBI spec**.
https://github.com/riscv/riscv-platform-specs

The SBI (*Supervisor Binary Interface*) spec is a living document:
https://github.com/riscv-non-isa/riscv-sbi-doc

RISC-V Runtime Services

Runtime Services are listed in **platform specs**, referencing the **SBI spec**.
https://github.com/riscv/riscv-platform-specs

The SBI (*Supervisor Binary Interface*) spec is a living document:
https://github.com/riscv-non-isa/riscv-sbi-doc

It defines extensions and functions similar to system calls.

RISC-V Runtime Services

Runtime Services are listed in **platform specs**, referencing the **SBI spec**.
https://github.com/riscv/riscv-platform-specs

The SBI (*Supervisor Binary Interface*) spec is a living document:
https://github.com/riscv-non-isa/riscv-sbi-doc

It defines extensions and functions similar to system calls.

Ports need to be written per platform (core/SoC/board). We have one for the D1.

RISC-V Runtime Services

Runtime Services are listed in **platform specs**, referencing the **SBI spec**.
https://github.com/riscv/riscv-platform-specs

The SBI (*Supervisor Binary Interface*) spec is a living document:
https://github.com/riscv-non-isa/riscv-sbi-doc

It defines extensions and functions similar to system calls.

Ports need to be written per platform (core/SoC/board). We have one for the D1.

In oreboot, we use RustSBI.
https://github.com/rustsbi/rustsbi
https://docs.rs/rustsbi/latest/rustsbi/

A note on RISC-V customizability

A note on RISC-V customizability

The platform specs define sets of instructions necessary in order to run an OS.

A note on RISC-V customizability

The platform specs define sets of instructions necessary in order to run an OS.

There are *Control and Status Registers (CSRs)*, similar to x86 MSRs.

A note on RISC-V customizability

The platform specs define sets of instructions necessary in order to run an OS.

There are *Control and Status Registers (CSRs)*, similar to x86 MSRs.

They allow for vendor specific custom extensions.

A note on RISC-V customizability

The platform specs define sets of instructions necessary in order to run an OS.

There are *Control and Status Registers (CSRs)*, similar to x86 MSRs.

They allow for vendor specific custom extensions.

They *may* be required for full usage of the SoC.

A note on RISC-V customizability

The platform specs define sets of instructions necessary in order to run an OS.

There are *Control and Status Registers (CSRs)*, similar to x86 MSRs.

They allow for vendor specific custom extensions.

They *may* be required for full usage of the SoC.

Vendors may also implement custom instructions.

A note on RISC-V customizability

The platform specs define sets of instructions necessary in order to run an OS.

There are *Control and Status Registers (CSRs)*, similar to x86 MSRs.

They allow for vendor specific custom extensions.

They *may* be required for full usage of the SoC.

Vendors may also implement custom instructions.

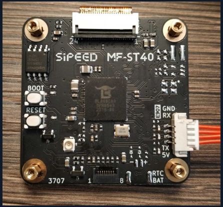
Both CSRs and custom instructions may be neglected, and a subset of the SoC’s capabilities be used.

Potential RISC-V SoCs for oreboot

Potential RISC-V SoCs for oreboot

BL808

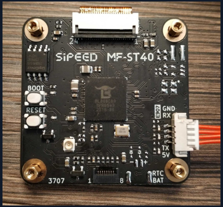
- already available (I have multiple boards)
- 1x C906 (512MHz), 1x E907, 1x low-power core



Potential RISC-V SoCs for oreboot

BL808

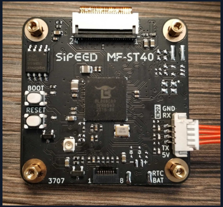
- already available (I have multiple boards)
- 1x C906 (512MHz), 1x E907, 1x low-power core



JH7110

- already available; some people received theirs from crowdfunding campaign
- *marketed* as *open* source
- no manual available
- initial U-Boot and Linux sources available
- https://github.com/starfive-tech/Tools is closed source

Potential RISC-V SoCs for oreboot



BL808

- already available (I have multiple boards)
- 1x C906 (512MHz), 1x E907, 1x low-power core

JH7110

- already available; some people received theirs from crowdfunding campaign
- *marketed* as *open* source
- no manual available
- initial U-Boot and Linux sources available
- https://github.com/starfive-tech/Tools is closed source

TH1520

- SoM with SPI flash placeholder
- coming soon
- multiple boards offered https://sipeed.com/licheepi4a
  - Lichee Pi 4A
  - Lichee Cluster 4A
  - Lichee Router 4A
  - Lichee Pad/Phone 4A

Further Work

Further Work

`layoutflash`

- library within oreboot
- idea: DTS for flash partitioning
- other ideas: add SBoM

Further Work

`layoutflash`

- library within oreboot
- idea: DTS for flash partitioning
- other ideas: add SBoM

xtask

Rust build framework used in oreboot
needs extension with more boards and common functions factored out

Further Work

`layoutflash`

- library within oreboot
- idea: DTS for flash partitioning
- other ideas: add SBoM

xtask

Rust build framework used in oreboot
needs extension with more boards and common functions factored out

ARM and other ISAs

We had some ARM and x86 boards and discarded them in favor of getting on.
However, there are issues tracking their status with starting points.
https://github.com/oreboot/oreboot/issues

Follow Me

Daniel Maslowski

https://github.com/orangecms
https://twitter.com/orangecms
https://twitch.tv/cyrevolt
https://youtube.com/@cyrevolt

https://github.com/oreboot/oreboot

https://metaspora.org/oreboot-comparison-riscv-d1-jh7100.pdf