固件自由

RISC-V 架构下的 coreboot

王翔
wxjstz@126.com

个人简介

- 就职于广州腾御安
- HardenedLinux 社区贡献者
- 固件研究

话题

- coreboot 简介
- RISC-V 简介
- coreboot 移植到 HiFive1 的分析

coreboot 简介

- 以前被称为 LinuxBIOS
- 基于 GPL 协议开源
- 旨在替换专有固件
- 需要初始化裸机，必须针对具体主板移植

coreboot 历史

- 起源： 1991 ，冬，阿拉莫斯实验室（ LANL ）
- 贡献者：开源社区以及部分主板供应商
- google 的支持： chromebook 使用 coreboot
- 其他架构支持： ARM 、 MIPS 、 RISC-V 、 POWER8

coreboot 设计

- coreboot 可以加载 linux 内核及裸机 ELF
- coreboot 主要通过 C 语言实现，只有少量的汇编代码。
- coteboot 执行很少代码后就进入保护模式，大部分 BIOS 运行在实模式
- 控制权一但交给 payload 就不会执行 coreboot 的代码
- coreboot 可以通过 SeaBIOS 提供 BIOS 调用服务，来启动一些依赖 BIOS 服务的操作系统
- coreboot 设计成多个阶段，每个阶段完成部分初始化并引导下一个阶段
- coreboot 把多个阶段打包成一个文件

coreboot 变种

- librecore
  - 更注重自由，非 x86 固件开发框架
- libreboot
  - 目的在于删除私有的二进制 blob

RISC-V

- RISC-V 发音 RISC-five
- 起源于加州大学伯克利
- 开源精简指令集（BSD）
- 模块化设计的指令集，灵活、可扩展

HiFive1 处理器

- 微控制器：SiFive Freedom E310（FE310）
  - IP 核：E31 Coreplex
  - 架构：32BIT RV32IMAC
  - 主频：320+MHz
  - 性能：1.61DMIPs/MHz，2.73Coremark/MHz
    （Context-M3 1.50 DMIPS/MHz，3.34 CoreMark/MHz）
- 内存：16KB 指令缓存，16KB 数据存储器
- 其他功能：硬件乘除法器，调试模块，灵活到时钟（片上振荡器、PLLs）
- SPI 控制器：
- 外部中断引脚：19 个
- 外部唤醒引脚：1 个
- Flash 存储器：外接 SPI Flash，128Mbit
- 主机接口（microUSB）：编程、调试、UART 通讯（通过 FT2232 分别与芯片到 JTAG 和 UART 连接）

开发工具

- **获取：**
  - git clone --recursive https://github.com/sifive/freedom-e-sdk.git
- **构建工具链：**
  - make tools BOARD = freedom-e300-hifive1
- **编译目标项目：**
  - make softwore BOARD=freedom-e300-hifive1 PROGRAM=hello
- **下载目标项目到目标板：**
  - make upload BOARD=freedom-e300-hifive1 PROGRAM=hello

原代码结构

- 当前 coreboot 代码并没有支持真实硬件
- 只有模拟器的支持（ qemu/spike ）
- coreboot 启动过程被分为四个阶段，各个阶段为独立的可执行文件
  - bootblock
  - rom stage
  - ram stage
  - payload

原代码结构： bootblock

- **bootblock 为机器上电执行的代码，入可位于 src/arch/riscv/bootblock.S 中**
  - 初始化堆栈、初始化 hart 相关的数据结构、初始化异常、然后转入 C 程序入口
- **bootblock 有通用到 C 语言部分，它提供了一个通用到初始化引导流程。源代码位于 src/lib/bootblock.c 中**
  - 初始化过程主要通过四个函数实现，具体平台可以根据具体情况具体实现。
    - __attribute__((weak)) void bootblock_soc_early_init(void) { /* do nothing */
    - __attribute__((weak)) void bootblock_mainboard_early_init(void) { /* no-op */
    - __attribute__((weak)) void bootblock_soc_init(void) { /* do nothing */
    - __attribute__((weak)) void bootblock_mainboard_init(void) { /* do nothing */ }
- **下一个过程由框架提供的 run_ramstage 实现**

原代码结构： rom stage

- **入口 stage_entry 位于 src/arch/riscv/stages.c 中**
  - 不初始化 .data .bsss ，因为这部分工作被加载过程代替了
- **主程序位于：**
  - src/mainboard/emulation/spike-riscv/romstage.c
- **功能：初始化串口、打印内存信息、引导下一个过程**
  - 引导过程由框架代码 run_ramstage 实现

原代码结构： ram stage

- 入口 stage_entry 位于 src/arch/riscv/stages.c 中
  - 不初始化 .data .bsss ，因为这部分工作被加载过程代替了
- 有通用到 C 语言部分，它提供一个通用到初始化引导流程。源代码位于：
  - src/lib/hardwaremain.c 中
- ram stage 把启动过程分为 12 步，由枚举 boot_state_t 定义

```
tpedef enum {
        BS_PRE_DEVICE,  BS_DEV_INIT_CHIPS,  BS_DEV_ENUMERATE,
        BS_DEV_RESOURCES,  BS_DEV_ENABLE,  BS_DEV_INIT,
        BS_POST_DEVICE,  BS_OS_RESUME_CHECK,  BS_OS_RESUME,
        BS_WRITE_TABLES,  BS_PAYLOAD_LOAD,  BS_PAYLOAD_BOOT,
} boot_state_t;
```

原代码结构： ram stage

- 每个过程分为三步： Entry callbacks 、 State Actions 、 Exit callbacks
- 每个过程通过 boot_state 描述， State Actions 由框架实现代码固定
- Entry callbacks 、 Exit callbacks 用户可以根据具体平台实现
- 一个 callbacks 通过 boot_state_callback 结构描述， boot_state_callback 可以构成链表记录同一个过程中同一个步骤的所有回调函数
- boot_state_init_entry 结构体用于记录一个回调，以及其发生的过程和步骤，并定义了一个宏 BOOT_STATE_INIT_ENTRY ，用于把 boot_state_init_entry 类型的变量放入同一个段，这样便于初始化（建立回调函数与 boot_state 的关联）
- boot_state_schedule_static_entries 函数用于建立回调与 boot_state 到关联
- bs_walk_state_machine 用于遍历 boot_state 数组，一步一步执行初始化及引导过程

原代码结构： ram stage

- 初始化主流程主要负责处理设备树
- 设备树由两种数据结构组成： device / bus
- 设备树上有芯片配置信息、操作的句柄，主流程主要通过遍历设备树对设备初始化
- coreboot 自定义了一套语法处理设备树，解析器位于源码 util/sconfig

原代码结构：引导与加载

- coreboot 把 bootblock 、 rom stage 、 ram stage 以及 payload 打包成一个固件文件
- bootblock 由硬件加载进内存（类似 BIOS 加载 MBR ）
- 其他阶段通过上一个阶段解析固件文件找到并加载进内存执
- 跳转到下一个阶段的代码，需要具体架构实现（与权限有关）
  - 这个函数被命名为 arch_prog_run
  - riscv 源码位于 src/arch/riscv/boot.c 中

原代码结构：固件文件结构

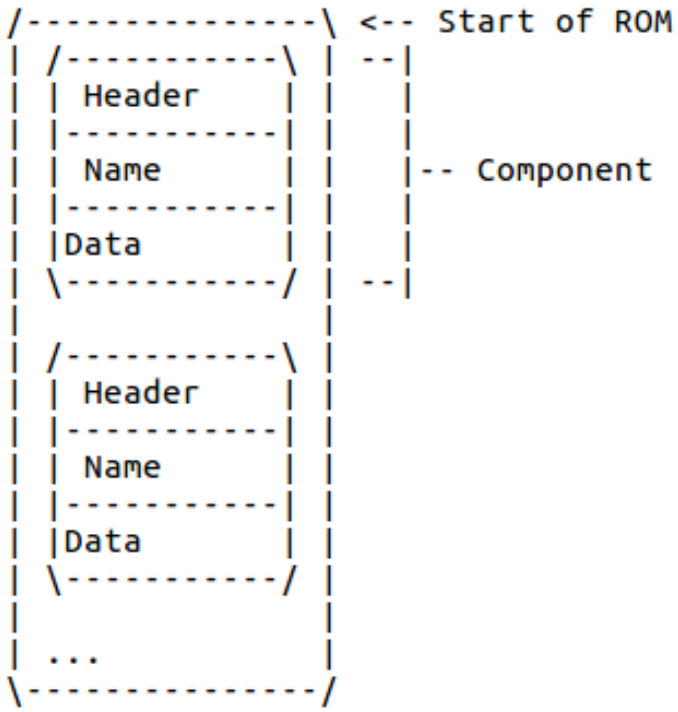
- 固件顶层是一个固定的块结构，此结构把固件文件分成固定大小到块，每一个块具有名字。
- 名字为 FMAP 的块存放了块布局信息。块信息通过两个结构体描述，fmap 、fmap_area

```
struct fmap {
    /* "__FMAP__" (0x5F5F464D41505F5F) */
    uint8_t  signature[8];
    /* major version */
    uint8_t  ver_major;
    /* minor version */
    uint8_t  ver_minor;
    /* address of the firmware binary */
    uint64_t base;
    /* size of firmware binary in bytes */
    uint32_t size;
    /* name of this firmware binary */
    uint8_t  name[FMAP_STRLEN];
    uint16_t nareas;
    struct fmap_area areas[];
} __attribute__((packed));
```

```
struct fmap_area {
    /* offset relative to base */
    uint32_t offset;
    /* size in bytes */
    uint32_t size;
    /* descriptive name */
    uint8_t  name[FMAP_STRLEN];
    /* flags for this area */
    uint16_t flags;
} __attribute__((packed));
```

原代码结构：固件文件结构

- FMAP 中名字为 COREBOOT 的块存放了 CBFS
- cbfs 内存试图如下，其中一个 component 对应一个文件



原代码结构：固件文件结构

component 中的 Header 对应结构体 cbfs_file

struct cbfs_file {

uint8_t magic[8];

uint32_t len;

uint32_t type;

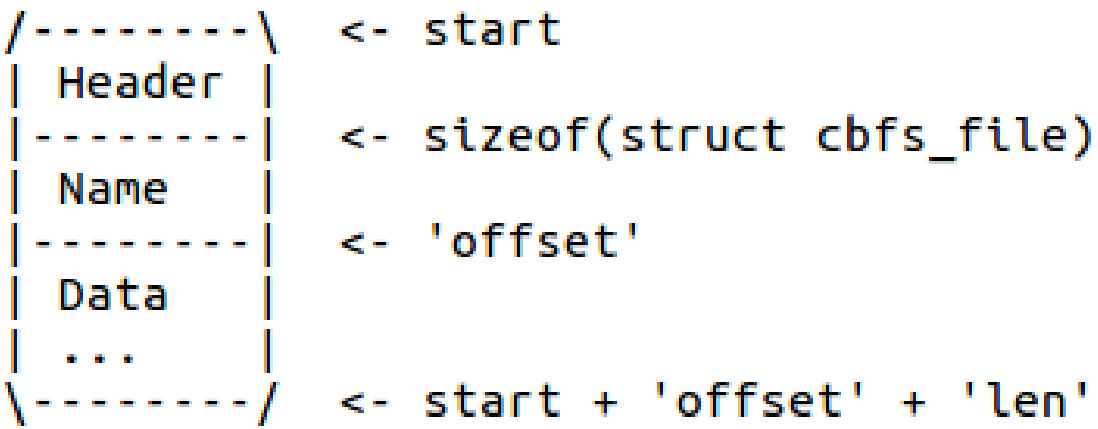
uint32_t attributes_offset;

uint32_t offset;

char filename[];

} __PACKED;

offset 、 len 描述 Data 的位置，如右图



原代码结构：固件文件结构

- **stage 、 payload 是封装在 component 中的数据结构**
- **涉及以下结构，这里不具体介绍**
  - cbfs_stage
  - cbfs_payload
  - cbfs_payload_segment

原代码结构：终端

- 终端代码位于 src/console 中
- console.c 是主要的框架代码，框架可以使用多种硬件
- init.c 主要导出了终端初始化函数
- vtxprintf.c 、 vsprintf.c 、 printk.c 几个文件通过 console.c 提供到接口，实现了类 C 语言 printf 的 printk 函数以及一些简单的辅助函数

移植：问题与策略

- **问题**
  - coreboot 源码是基于 RV64 的
  - RAM 空间太小，无法把固件加载进内存运行
- **策略**
  - 要支持 RV32 首先需要修改编译选项，让编译器汇编器生成 32 未代码，然后编译报错后对源码进行修改，主要涉及汇编文件中的汇编指令与 C 语言中 32 位 64 位数据类型问题
  - RAM 空间不足问题，这个暂时无法解决。 HiFive1 是一个评估板，更多是用于验证 IP 核的。这里，我剔除 cbfs 文件结构，直接在 Flash 中布局程序，上一个阶段直接跳转到下一个阶段

移植：编译选项

- 创建两个目录 src/soc/sifive 、 src/mainboard/sifive 。
- 在 SOC 目录下创建 Kconfig 定义宏
- 修改 src/arch/riscv/Makefine.ini ，把两个宏到值传递给编译器、汇编器

```
config SOC_SIFIVE_E300
    ......
    bool
    default n
if SOC_SIFIVE_E300
    config RISCV_ARCH
        string
        default "rv32imac"
    config RISCV_ABI
        string
        default "ilp32"
endif
```

移植：汇编指令相关

- **src/arch/riscv/include/bits.h**
  - 32 位跟 64 位模式下操作内存（ LOAD 、 STORE ）
  - 操作一个字长指令有差异
  - 这涉及一些汇编代码，可以根据编译错误修正
- **src/arch/riscv/include/vm.h**
  - 主要实现访问物理内存到方法，与数据类型和指令相关
  - 需要根据平台修正

移植：数据类型相关

- src/arch/riscv/include/stdin.h
  - 其中定义了机器字的类型，这需要根据平台修正

```
#if __riscv_xlen == 32
    - typedef s32    intptr_t;
    - typedef u32    uintptr_t;
#elif __riscv_xlen == 64
    - typedef s64    intptr_t;
    - typedef u64    uintptr_t;
#endif
```

  - 数据类型会影响很多 C 程序，这部分一般编译时会报错，可以根据编译器提示进行修改

HardenedLinux

移植：内存布局

- 这部分由链接脚本实现，coreboot平台实现了一些宏（src/include/memlayout.h），这些宏用于方便定义各个阶段的内存位置
- 但coreboot认为各个阶段都是加载进内存的（代码段与数据段是连续的），而HiFive1这种MCU结构可以直接在Flash中执行，需要对此文件进行修正
- HiFive1内存布局如下

```
#define RAM_START   0x80000000
#define FLASH_START 0x20400000
SECTIONS
{
   BOOTBLOCK(      FLASH_START, 64K)
   ROMSTAGE ( FLASH_START + 64K, 64K)
   RAMSTAGE (FLASH_START + 128K, 64K)

   SRAM_START        (RAM_START)
   STACK           (RAM_START + 4K, 4K)
   PAGETABLES        (RAM_START + 8K, 4K)
   PRERAM_CBMEM_CONSOLE(RAM_START + 12K, 4K)
   SRAM_END          (RAM_START + 16k)
}
```

HardenedLinux

移植：终端

- 这部分参照部分 SOC 的代码，以及 src/console/console.c 。
- 需要在 SOC 目录到 Kconfig 中添加 select HAVE_UART_SPECIAL
- 这样不使用平台实现的一些驱动（ 8250 等）

- 然后实现 5 个函数

```
void uart_init(int idx);                                  // 初始化
uint8_t uart_rx_byte(int idx);                            // 接受一个字符
void uart_tx_byte(int idx,unsigned char data); // 发送一个字符
void uart_tx_flush(int idx);                              // 刷新输出流
void uart_fill_lb(void *data);                            // 初始化一些平台数据
```

移植：执行下一阶段代码

- 由于知道每个阶段下一个阶段的起始位置，可以直接使用跳转
- 这可以通过把地址强制转换为函数指针再进行函数调用实现
- 例如：
  - *((void (*)(void))0x20410000)();*

移植：链接多个阶段

- **1、把各个阶段转换为 binary**
  - 通过 objcopy 实现
- **2、把 binary 封装成 elf**
  - 通过汇编器实现，汇编文件样式

```
.global _start
.section .text
_start:
.incbin "bootblock.bin"
```

- **3、链接**

```
ENTRY(_start)
SECTIONS {
    .text 0x20400000 : {
        . = 0x00000;
        bootblock-raw.o(.text)
        . = 0x10000;
        romstage-raw.o(.text)
        . = 0x20000;
        ramstage-raw.o(.text)
        . = 0x30000;
        hello-raw.o(.text)
    }
}
```

HardenedLinux

移植：下载

```
# 先启动 openocd
openocd -f openocd.cfg >/dev/null 2>&1 &

# 再通过 gdb 下载代码到目标板
riscv64-unknown-elf-gdb coreboot-raw.elf |
    --batch |
    -ex "set remotetimeout 500" |
    -ex "target extended-remote localhost:3333" |
    -ex "monitor reset halt" |
    -ex "monitor flash protect 0 64 last off" |
    -ex "load" |
    -ex "monitor resume" |
    -ex "monitor shutdown" |
    -ex "quit" && echo Successfully uploaded coreboot-raw.elf
```

HardenedLinux

移植：运行截图

minicom

文件(F)　编辑(E)　查看(V)　搜索(S)　终端(T)　帮助(H)

```
Press CTRL-A Z for help on special keys

core freq at 266420224 Hz
hart 0: HLS is 80001fc0
Time is 00000000 and timecmp is 00000000


coreboot-586246c Thu Aug  3 10:42:19 UTC 2017 bootblock starting...


coreboot-586246c Thu Aug  3 10:42:19 UTC 2017 romstage starting...


coreboot-586246c Thu Aug  3 10:42:19 UTC 2017 ramstage starting...


loading payload...

core freq at 266649600 Hz
hello world!


Progam has exited with code:0x00000000
```

HardenedLinux

参考

- RISC-V 网站
  https://riscv.org/
- RISC-V 邮件列表
  https://riscv.org/mailing-lists/
- RISC-V 架构分析
  https://github.com/hardenedlinux/embedded-iot_profile/blob/master/docs/riscv/riscv%E6%9E%B6%E6%9E%84.md
- coreboot 网站
  https://www.coreboot.org/
- coreboot 邮件列表
  https://www.coreboot.org/Mailinglist
- coreboot 源码分析
  https://github.com/hardenedlinux/embedded-iot_profile/blob/master/docs/riscv/coreboot%E5%88%86%E6%9E%90.md
- 移植的代码
  https://github.com/hardenedlinux/coreboot4HiFive1

问答

HardenedLinux