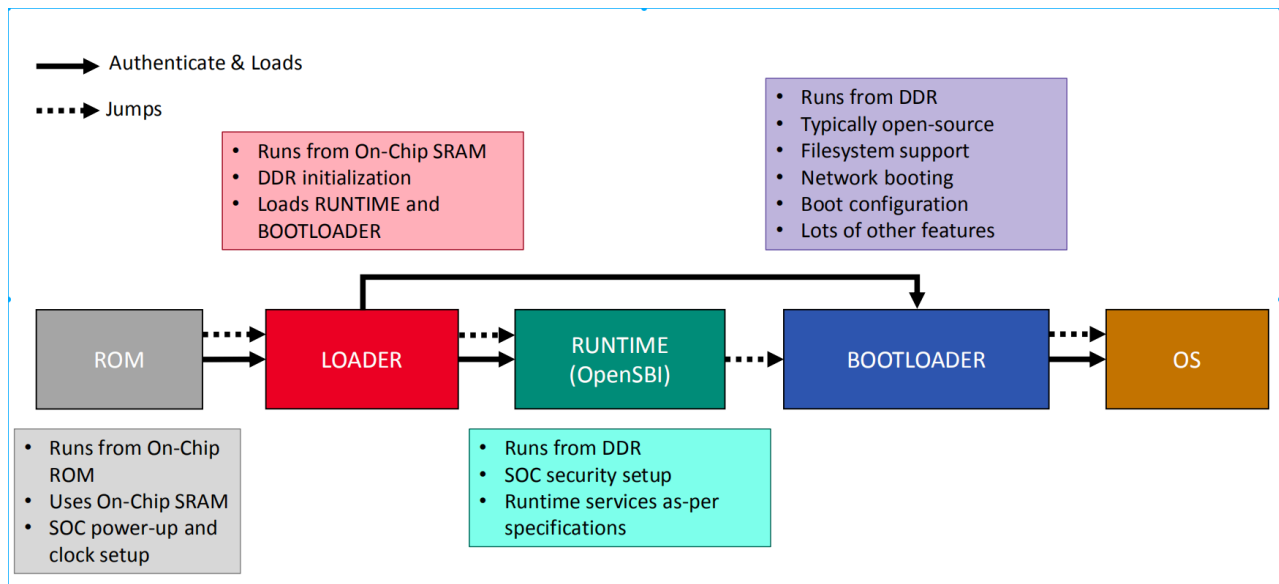


# 为quard-star移植opensbi-1.2

yanglianoo.github.io/2023/06/22/为quard-star移植opensbi-1-2

2023年6月22日

## 1. RISC-V的多级启动流程



如上图RISC-V的多级启动流程从ROM上的代码开始，实心黑箭头代表加载操作，虚线黑箭头代表跳转操作，因此rom上的代码负责把LOADER的代码加载到SARM上然后跳转到LOADER处执行，LOADER的代码会初始化DDR然后加载Opensbi固件到DDR，然后跳转到Opensbi处执行，或者直接加载BOOTLOADER，然后跳转到BOOTLOADER处执行，最后BOOTLOADER会加载OS然后跳转到OS处启动操作系统

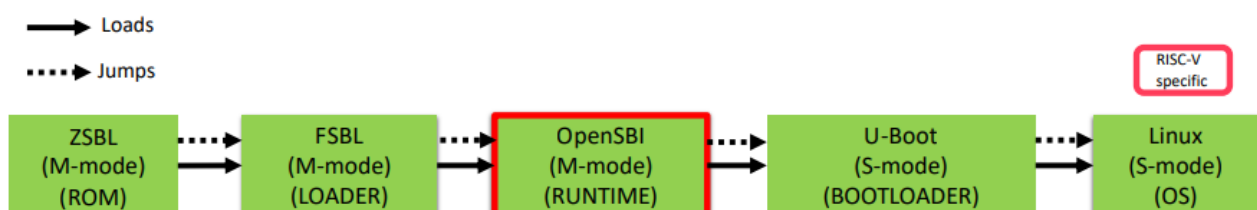
第一阶段为ZSBL，运行在M模式下，对应到quard-star的ROM上的代码就是复位向量代码就是那个reset\_vec里的代码，由于我们通过qemu将固件通过-drive命令直接加载到了flash的地方，所以rom上的代码不用执行load操作。

第二阶段为FSBL，运行在M模式下，对应到quard-star就是flash上的代码，在这段代码里我们需要加载Opensbi固件，加载设备树，然后跳转到Opensbi处执行

第三阶段就是Opensbi了，Opensbi是运行在M模式下的一段运行时代码

第四阶段为U-boot

第五阶段为OS



## 2. Opensbi简介

参考链接：OpenSBI三种固件的区别 - 知乎 (zhihu.com)

QEMU 启动方式分析（1）：QEMU 及 RISC-V 启动流程简介 - yjmstr - 博客园 (cnblogs.com)

RISC-V 的 Runtime 通常是 OpenSBI，它是运行在 M 模式下的程序，但能够为 S 模式提供一些特定的服务，这些服务由 SBI (Supervisor Binary Interface) 规范定义。

SBI 是指 Supervisor Binary Interface，它是运行在 M 模式下的程序，操作系统通过 SBI 来调用 M 模式的硬件资源。而 OpenSBI 是指西数开发的一种开源 SBI 实现。

OpenSBI 有三种 Firmware：

- FW\_PAYLOAD：下一引导阶段被作为 payload 打包进来，通常是 U-Boot 或 Linux。这是兼容 Linux 的 RISC-V 硬件所使用的默认 firmware。
- FW\_JUMP：跳转到一个固定地址，该地址上需存有下一个加载器。QEMU 的早期版本曾经使用过它。
- FW\_DYNAMIC：根据前一个阶段传入的信息加载下一个阶段。通常是 U-Boot SPL 使用它。现在 QEMU 默认使用 FW\_DYNAMIC。

下载Opensbi源码：Releases · riscv-software-src/opensbi (github.com)

Dec 24, 2022

avpatel

v1.2

6b5188c

Compare

### OpenSBI Version 1.2




Latest

This release has:

- Menuconfig support using Kconfiglib v14.1.0
- RISC-V AIA v1.0.0 support
- Cadence UART driver
- Platform specific PMU device operations
- Trap handling improvements for platforms with H-extension
- Semihosting support
- T-HEAD C9XX PMU and CLINT support
- FDT based drivers for Andes AE350 platform
- Andes AE350 platform improvements
- Allow enabling/disabling of SBI extensions via Kconfig
- Renesas SCIF serial driver
- Renesas RZ/Five platform support

Overall, this release mainly adds Kconfig support, new drivers and new platforms along with other improvements.

#### Assets

 opensbi-1.2-rv-bin.tar.xz	12.3 MB	Dec 24, 2022
 Source code (zip)		Dec 24, 2022
 Source code (tar.gz)		Dec 24, 2022

截至笔者下载的时候，最新的版本为1.2，不同的版本有一定的区别，0.9和1.2版本在配置方面有许多差距，下载完成源码后解压到quard-star目录下。

sh

```
timer@DESKTOP-JI9EVEH:~/quard-star$ ls
README.md  boot  build.sh  dts  opensbi-1.2  output  qemu-8.0.2  qemu.log
run.sh
```

### 3. 移植Opensbi-1.2到quard-star上

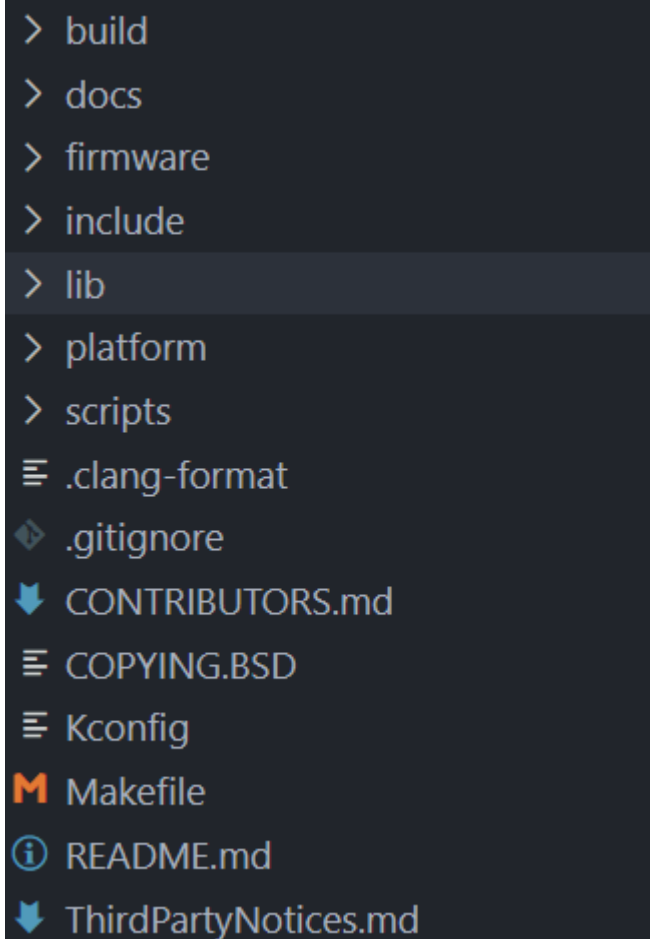
---

在移植之前需要明确几个东西:

- 我们采用Opensbi的固件是FW\_JUMP 类型的, 会被加载到DRAM处执行即0x80000000
- 需要自行编写设备树编译将设备树的地址传递给Opensbi, rom上的fw\_dynamic\_info用不到
- 需要编写在flash上运行的代码来将opensbi的固件加载到DRAM处, 然后跳转执行

#### 3.1 在opensbi中新增quard-star支持

---



```
> build
> docs
> firmware
> include
> lib
> platform
> scripts
≡ .clang-format
🔹 .gitignore
📄 CONTRIBUTORS.md
≡ COPYING.BSD
≡ Kconfig
📄 Makefile
📄 README.md
📄 ThirdPartyNotices.md
```

可以看到opensbi的源码并不是特别多, 我们先看看docs下的platform\_guide.md, 里面有这么一段话:

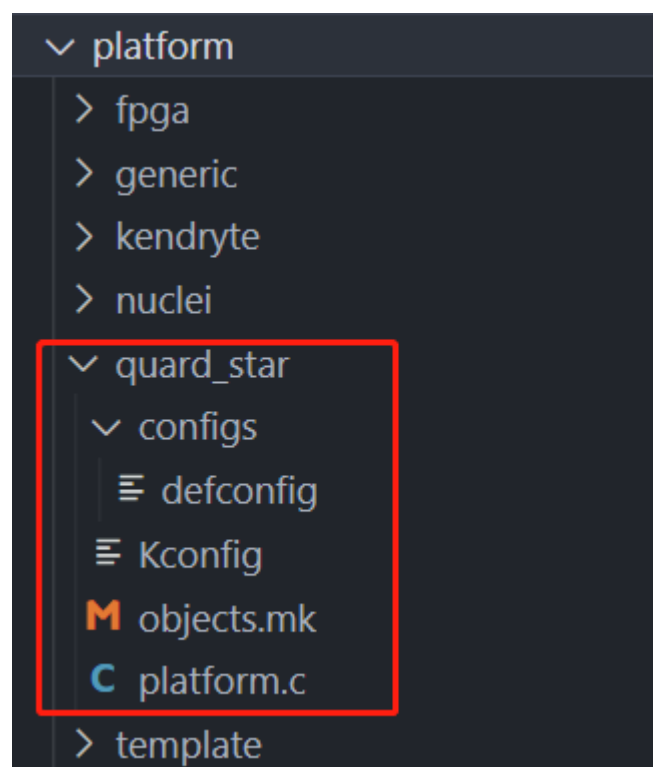
## Adding support for a new platform

Support for a new platform named `<xyz>` can be added as follows:

1. Create a directory named `<xyz>` under the `platform/` directory.
2. Create platform configuration files named `Kconfig` and `configs/defconfig` under the `platform/<xyz>/` directory. These configuration files will provide the build time configuration for the sources to be compiled.
3. Create a `platform/<xyz>/objects.mk` file for listing the platform object files to be compiled. This file also provides platform-specific compiler flags, and select firmware options.
4. Create a `platform/<xyz>/platform.c` file providing a `struct sbi_platform` instance.

这里告诉了我们如何在为`quard_star`新增`opensbi`的支持，如下：

- 在platform文件夹下新建一个名为`quard_star`的文件夹
- 在`quard_star`文件夹下新增三个文件`Kconfig`，`objects.mk`，`platform.c`
- 在`quard_star`文件夹下新增一个文件夹`configs`，在`configs`目录下新建一个名为`defconfig`的文件



可以看到在platform文件夹下还有一些其他平台，比如`kendryte-k210`，还有一个名为`generic`的平台，这是一个通用的板级平台，我们的`quard_star`就是参考`generic`的代码实现的，还有一个叫`template`的平台，这是`opensbi`官方提供的默认工程。

`Kconfig`文件的内容，这里照着`generic`中的内容copy了过来，看样子是选择使用FDT以及支持fdt的domain配置和fdt的pmu配置。

## plaintext

```
# SPDX-License-Identifier: BSD-2-  
Clause
```

```
config PLATFORM_QUARD_STAR  
    bool  
    select FDT  
    select FDT_DOMAIN  
    select FDT_PMU  
    default y
```

objects的内容比较简单，配置固件为JUMP类型，同时指定jump的地址

## makefile

```
#  
# SPDX-License-Identifier: BSD-2-  
Clause  
#  
  
# Compiler flags  
platform-cppflags-y =  
platform-cflags-y =  
platform-asflags-y =  
platform-ldflags-y =  
  
# Objects to build  
platform-objs-y += platform.o  
  
# Blobs to build  
FW_JUMP=y  
FW_TEXT_START=0x80000000  
FW_JUMP_ADDR=0x0
```

**defconfig**的内容是指定配置支持哪些硬件，这里也是把**generic**中的内容**copy**了过来，虽然现在**quard\_star**的硬件比较少，但是不管，先把所有的都添加进去

## makefile

```
CONFIG_PLATFORM_ALLWINNER_D1=
y
CONFIG_PLATFORM_ANDES_AE350=y
CONFIG_PLATFORM_RENESAS_RZFIV
E=y
CONFIG_PLATFORM_SIFIVE_FU540=
y
CONFIG_PLATFORM_SIFIVE_FU740=
y
CONFIG_FDT_GPIO=y
CONFIG_FDT_GPIO_SIFIVE=y
CONFIG_FDT_I2C=y
CONFIG_FDT_I2C_SIFIVE=y
CONFIG_FDT_IPI=y
CONFIG_FDT_IPI_MSWI=y
CONFIG_FDT_IPI_PLICSW=y
CONFIG_FDT_IRQCHIP=y
CONFIG_FDT_IRQCHIP_APLIC=y
CONFIG_FDT_IRQCHIP_IMSIC=y
CONFIG_FDT_IRQCHIP_PLIC=y
CONFIG_FDT_RESET=y
CONFIG_FDT_RESET_ATCWD200=y
CONFIG_FDT_RESET_GPIO=y
CONFIG_FDT_RESET_HTIF=y
CONFIG_FDT_RESET_SIFIVE_TEST=
y
CONFIG_FDT_RESET_SUNXI_WDT=y
CONFIG_FDT_RESET_THEAD=y
CONFIG_FDT_SERIAL=y
CONFIG_FDT_SERIAL_CADENCE=y
CONFIG_FDT_SERIAL_GAISLER=y
CONFIG_FDT_SERIAL_HTIF=y
CONFIG_FDT_SERIAL_RENESAS_SCI
F=y
CONFIG_FDT_SERIAL_SHAKTI=y
CONFIG_FDT_SERIAL_SIFIVE=y
CONFIG_FDT_SERIAL_LITEX=y
CONFIG_FDT_SERIAL_UART8250=y
CONFIG_FDT_SERIAL_XILINX_UART
LITE=y
CONFIG_FDT_TIMER=y
CONFIG_FDT_TIMER_MTIMER=y
CONFIG_FDT_TIMER_PLMT=y
CONFIG_SERIAL_SEMIHOSTING=y
```

platform.c:

在platform.c中第一个比较重要的函数为fw\_platform\_init

```

unsigned long fw_platform_init(unsigned long arg0, unsigned long
arg1,
                                unsigned long arg2, unsigned long
arg3,
                                unsigned long arg4)
{
    const char *model;
    void *fdt = (void *)arg1;
    u32 hartid, hart_count = 0;
    int rc, root_offset, cpus_offset, cpu_offset, len;

    root_offset = fdt_path_offset(fdt, "/");
    if (root_offset < 0)
        goto fail;

    model = fdt_getprop(fdt, root_offset, "model", &len);
    if (model)
        sbi_strncpy(platform.name, model,
sizeof(platform.name));

    cpus_offset = fdt_path_offset(fdt, "/cpus");
    if (cpus_offset < 0)
        goto fail;

    fdt_for_each_subnode(cpu_offset, fdt, cpus_offset) {
        rc = fdt_parse_hart_id(fdt, cpu_offset, &hartid);
        if (rc)
            continue;

        if (SBI_HARTMASK_MAX_BITS <= hartid)
            continue;

        quard_star_hart_index2id[hart_count++] = hartid;
    }

    platform.hart_count = hart_count;

    /* Return original FDT pointer */
    return arg1;

fail:
    while (1)
        wfi();
}

```

传入的五个arg依次对应着上一启动阶段传递过来的参数，为a0~a4寄存器，其中a0中放的是hard id，a1中放的是设备树的地址，arg1 被强制转换为一个指向设备树（Device Tree）的指针，即 fdt。函数主要的逻辑如下：

1. 首先，通过解析设备树来获取平台的模型名称（"model" 属性），并将其存储在 platform.name 变量中。

2. 接下来，在设备树的“/cpus”路径下遍历处理器节点，获取每个处理器的 hartid（处理器标识符）。
3. 根据获取的 hartid，将其存储在 quard\_star\_hart\_index2id 数组中，并增加 hart\_count 变量的计数。
4. 最后，设置 platform.hart\_count 变量为 hart\_count，表示平台上处理器的数量。
5. 函数返回 arg1，即原始的设备树指针。

第二个重要的定义为platform\_ops

C

```
const struct sbi_platform_operations platform_ops = {
    .early_init      = quard_star_early_init,      //早期初始化,
    不需要
    .final_init      = quard_star_final_init,      //最终初始化,
    需要
    .early_exit      = quard_star_early_exit,      //早期退出, 不
    需要
    .final_exit      = quard_star_final_exit,      //最终退出, 不
    需要
    .domains_init    = quard_star_domains_init,    //从设备树填充域,
    需要
    .console_init    = fdt_serial_init,            //初始化控制台
    .irqchip_init    = fdt_irqchip_init,           //初始化中断
    .irqchip_exit    = fdt_irqchip_exit,           //中断退出
    .ipi_init        = fdt_ipi_init,               //中断通信
    .ipi_exit        = fdt_ipi_exit,
    .pmu_init        = quard_star_pmu_init,         //电源配置
    .pmu_xlate_to_mhpmevent = quard_star_pmu_xlate_to_mhpmevent,
    .get_tlbr_flush_limit = quard_star_tlbr_flush_limit, //需要
    .timer_init      = fdt_timer_init,
    .timer_exit      = fdt_timer_exit,
};
```

这段代码定义了一个名为“platform\_ops”的结构体变量，类型为“const struct sbi\_platform\_operations”，用于指定平台相关的操作函数。

这个结构体包含了多个成员，每个成员对应一个平台相关的操作函数。以下是每个成员和对应的函数名：

- early\_init: quard\_star\_early\_init
- final\_init: quard\_star\_final\_init
- early\_exit: quard\_star\_early\_exit
- final\_exit: quard\_star\_final\_exit
- domains\_init: quard\_star\_domains\_init
- console\_init: fdt\_serial\_init
- irqchip\_init: fdt\_irqchip\_init
- irqchip\_exit: fdt\_irqchip\_exit
- ipi\_init: fdt\_ipi\_init
- ipi\_exit: fdt\_ipi\_exit



- `pmu_init`: `quard_star_pmu_init`
- `pmu_xlate_to_mhpmevent`: `quard_star_pmu_xlate_to_mhpmevent`
- `get_tlbr_flush_limit`: `quard_star_tlbr_flush_limit`
- `timer_init`: `fdt_timer_init`
- `timer_exit`: `fdt_timer_exit`

这些函数是平台特定的操作函数，用于在 OpenSBI 初始化过程中进行特定的操作和配置。每个函数在相应的阶段被调用，以完成与平台相关的初始化、配置和资源管理等工作。

通过定义这个结构体并填充相应的函数指针，OpenSBI 可以根据平台的特性和需求，调用适当的操作函数，以确保其不同平台上的正确运行和适配性。

其中这些操作函数的代码如下：



```

static int quard_star_early_init(bool cold_boot)
{
    return 0;
}

static int quard_star_final_init(bool cold_boot)
{
    void *fdt;

    if (cold_boot)
        fdt_reset_init();
    if (!cold_boot)
        return 0;

    fdt = sbi_scratch_thishart_arg1_ptr();

    fdt_cpu_fixup(fdt);
    fdt_fixups(fdt);
    fdt_domain_fixup(fdt);

    return 0;
}

static void quard_star_early_exit(void)
{
}

static void quard_star_final_exit(void)
{
}

static int quard_star_domains_init(void)
{
    return fdt_domains_populate(fdt_get_address());
}

static int quard_star_pmu_init(void)
{
    return fdt_pmu_setup(fdt_get_address());
}

static uint64_t quard_star_pmu_xlate_to_mhpmevent(uint32_t event_idx,
                                                    uint64_t data)
{
    uint64_t evt_val = 0;

    /* data is valid only for raw events and is equal to event
    selector */
    if (event_idx == SBI_PMU_EVENT_RAW_IDX)
        evt_val = data;
    else {
        /**
         * Generic platform follows the SBI specification
         recommendation

```

```

        * i.e. zero extended event_idx is used as mhpmevent value
for
        * hardware general/cache events if platform doesn't define
one.

        */
    evt_val = fdt_pmu_get_select_value(event_idx);
    if (!evt_val)
        evt_val = (uint64_t)event_idx;
}

    return evt_val;
}
static u64 quard_star_tlb_flush_limit(void)
{
    return SBI_PLATFORM_TLB_RANGE_FLUSH_LIMIT_DEFAULT;
}

```

最后定义了platform结构体，代码如下：

C

```

struct sbi_platform platform = {
    .opensbi_version      = OPENSBI_VERSION,
    .platform_version     = SBI_PLATFORM_VERSION(0x0,
0x01),
    .name                 = "Quard-Star",
    .features             = SBI_PLATFORM_DEFAULT_FEATURES,
    .hart_count           = SBI_HARTMASK_MAX_BITS,
    .hart_index2id        = quard_star_hart_index2id,
    .hart_stack_size      =
SBI_PLATFORM_DEFAULT_HART_STACK_SIZE,
    .platform_ops_addr    = (unsigned long)&platform_ops
};

```

这段代码定义了一个名为“platform”的结构体变量，类型为“struct sbi\_platform”，用于描述平台的相关信息和配置。

这个结构体包含了多个成员，每个成员用于存储特定的平台属性和配置。以下是每个成员和对应的值或指针：

- **opensbi\_version**: OPENSBI\_VERSION, 指定了 OpenSBI 的版本号。
- **platform\_version**: SBI\_PLATFORM\_VERSION(0x0, 0x01), 指定了平台的版本号。
- **name**: "Quard-Star", 指定了平台的名称。
- **features**: SBI\_PLATFORM\_DEFAULT\_FEATURES, 指定了平台的默认特性。
- **hart\_count**: SBI\_HARTMASK\_MAX\_BITS, 指定了平台上处理器 (Hart) 的数量。
- **hart\_index2id**: quard\_star\_hart\_index2id, 一个指向处理器标识符数组的指针，用于将处理器索引映射到唯一的处理器标识符。

- `hart_stack_size`: `SBI_PLATFORM_DEFAULT_HART_STACK_SIZE`, 指定了每个处理器的默认堆栈大小。
- `platform_ops_addr`: `(unsigned long)&platform_ops`, 指向平台操作函数结构体的指针, 用于指定平台操作函数的地址。

通过定义这个结构体并填充相应的值或指针, OpenSBI 可以了解和配置特定平台的相关信息, 以在运行时正确地适配和操作该平台。这些信息包括版本号、特性支持、处理器数量、堆栈大小和操作函数等。

## 3.2 编写设备树

---

在`quard_star`目录下新建`dtb`文件夹, 在此文件夹下新建`quard_star_sbi.dts`, 设备树文件如下:



```
/dts-v1/;
```

```
/ {
```

```
    #address-cells = <0x2>;  
    #size-cells = <0x2>;  
    compatible = "riscv-quard-star";  
    model = "riscv-quard-star,qemu";
```

```
    chosen {  
        stdout-path = "/soc/uart0@10000000";  
    };
```

```
    memory@80000000 {  
        device_type = "memory";  
        reg = <0x0 0x80000000 0x0 0x40000000>;  
    };
```

```
    cpus {  
        #address-cells = <0x1>;  
        #size-cells = <0x0>;  
        timebase-frequency = <0x989680>;
```

```
        cpu0: cpu@0 {  
            phandle = <0xf>;  
            device_type = "cpu";  
            reg = <0x0>;  
            status = "okay";  
            compatible = "riscv";  
            riscv,isa = "rv64imafdcsv";  
            mmu-type = "riscv,sv48";  
  
            interrupt-controller {  
                #interrupt-cells = <0x1>;  
                interrupt-controller;  
                compatible = "riscv,cpu-intc";  
                phandle = <0x10>;  
            };  
        };
```

```
        cpu1: cpu@1 {  
            phandle = <0xd>;  
            device_type = "cpu";  
            reg = <0x1>;  
            status = "okay";  
            compatible = "riscv";  
            riscv,isa = "rv64imafdcsv";  
            mmu-type = "riscv,sv48";  
  
            interrupt-controller {  
                #interrupt-cells = <0x1>;  
                interrupt-controller;  
                compatible = "riscv,cpu-intc";  
                phandle = <0xe>;  
            };  
        };
```

```
        cpu2: cpu@2 {  
            phandle = <0xb>;
```

```

        device_type = "cpu";
        reg = <0x2>;
        status = "okay";
        compatible = "riscv";
        riscv,isa = "rv64imafdcsv";
        mmu-type = "riscv,sv48";

        interrupt-controller {
            #interrupt-cells = <0x1>;
            interrupt-controller;
            compatible = "riscv,cpu-intc";
            phandle = <0xc>;
        };
};

cpu3: cpu@3 {
    phandle = <0x9>;
    device_type = "cpu";
    reg = <0x3>;
    status = "okay";
    compatible = "riscv";
    riscv,isa = "rv64imafdcsv";
    mmu-type = "riscv,sv48";

    interrupt-controller {
        #interrupt-cells = <0x1>;
        interrupt-controller;
        compatible = "riscv,cpu-intc";
        phandle = <0xa>;
    };
};

cpu4: cpu@4 {
    phandle = <0x7>;
    device_type = "cpu";
    reg = <0x4>;
    status = "okay";
    compatible = "riscv";
    riscv,isa = "rv64imafdcsv";
    mmu-type = "riscv,sv48";

    interrupt-controller {
        #interrupt-cells = <0x1>;
        interrupt-controller;
        compatible = "riscv,cpu-intc";
        phandle = <0x8>;
    };
};

cpu5: cpu@5 {
    phandle = <0x5>;
    device_type = "cpu";
    reg = <0x5>;
    status = "okay";
    compatible = "riscv";
    riscv,isa = "rv64imafdcsv";
    mmu-type = "riscv,sv48";

    interrupt-controller {
        #interrupt-cells = <0x1>;

```



```

        interrupt-controller;
        compatible = "riscv,cpu-intc";
        phandle = <0x6>;
    };
};

cpu6: cpu@6 {
    phandle = <0x3>;
    device_type = "cpu";
    reg = <0x6>;
    status = "okay";
    compatible = "riscv";
    riscv,isa = "rv64imafdcsv";
    mmu-type = "riscv,sv48";

    interrupt-controller {
        #interrupt-cells = <0x1>;
        interrupt-controller;
        compatible = "riscv,cpu-intc";
        phandle = <0x4>;
    };
};

cpu7: cpu@7 {
    phandle = <0x1>;
    device_type = "cpu";
    reg = <0x7>;
    status = "okay";
    compatible = "riscv";
    riscv,isa = "rv64imafdcsv";
    mmu-type = "riscv,sv48";

    interrupt-controller {
        #interrupt-cells = <0x1>;
        interrupt-controller;
        compatible = "riscv,cpu-intc";
        phandle = <0x2>;
    };
};

cpu-map {

    cluster0 {

        core0 {
            cpu = <0xf>;
        };

        core1 {
            cpu = <0xd>;
        };

        core2 {
            cpu = <0xb>;
        };

        core3 {
            cpu = <0x9>;
        };
    };
};

```

```

        core4 {
            cpu = <0x7>;
        };

        core5 {
            cpu = <0x5>;
        };

        core6 {
            cpu = <0x3>;
        };

        core7 {
            cpu = <0x1>;
        };
    };
};

soc {
    #address-cells = <0x2>;
    #size-cells = <0x2>;
    compatible = "simple-bus";
    ranges;

    uart0: uart0@10000000 {
        interrupts = <0xa>;
        interrupt-parent = <0x11>;
        clock-frequency = <0x384000>;
        reg = <0x0 0x10000000 0x0 0x1000>;
        compatible = "ns16550a";
    };

    uart1: uart1@10001000 {
        interrupts = <0xa>;
        interrupt-parent = <0x11>;
        clock-frequency = <0x384000>;
        reg = <0x0 0x10001000 0x0 0x1000>;
        compatible = "ns16550a";
    };

    uart2: uart2@10002000 {
        interrupts = <0xa>;
        interrupt-parent = <0x11>;
        clock-frequency = <0x384000>;
        reg = <0x0 0x10002000 0x0 0x1000>;
        compatible = "ns16550a";
    };

    plic@c000000 {
        phandle = <0x11>;
        riscv,ndev = <0x35>;
        reg = <0x0 0xc000000 0x0 0x210000>;
        interrupts-extended = <0x10 0xb 0x10 0x9 0xe 0xb 0xe 0x9 0xc
0xb 0xc 0x9 0xa 0xb 0xa 0x9 0x8 0xb 0x8 0x9 0x6 0xb 0x6 0x9 0x4 0xb 0x4 0x9 0x2 0xb 0x2
0x9>;

        interrupt-controller;
        compatible = "riscv,plic0";
        #interrupt-cells = <0x1>;
        #address-cells = <0x0>;

```

```

};

clint@2000000 {
    interrupts-extended = <0x10 0x3 0x10 0x7 0xe 0x3 0xe 0x7 0xc
0x3 0xc 0x7 0xa 0x3 0xa 0x7 0x8 0x3 0x8 0x7 0x6 0x3 0x6 0x7 0x4 0x3 0x4 0x7 0x2 0x3 0x2
0x7>;

    reg = <0x0 0x2000000 0x0 0x10000>;
    compatible = "riscv,clint0";
};

};

```

关于设备树的语法和详细解释请参考我另外一篇博客：[设备树详解 | TimerのBlog](https://timer.blog.yanglianoo.github.io/)  
([yanglianoo.github.io](https://yanglianoo.github.io))

### 3.3 重新编写start.s

---

avrasm

```

        .macro loop,cunt
li        t1,    0xffff
li        t2,    \cunt
1:
        nop
        addi    t1, t1, -1
        bne     t1, x0, 1b
li        t1,    0xffff
        addi    t2, t2, -1
        bne     t2, x0, 1b
        .endm

        .macro load_data,_src_start,_dst_start,_dst_end
bgeu      \_dst_start, \_dst_end, 2f
1:
        lw      t0, (\_src_start)
        sw      t0, (\_dst_start)
        addi    \_src_start, \_src_start, 4
        addi    \_dst_start, \_dst_start, 4
        bltu    \_dst_start, \_dst_end, 1b
2:
        .endm

        .section .text
        .globl _start
        .type _start,@function

_start:
        //load opensbi_fw.bin
        //[0x20200000:0x20400000] -->
[0x80000000:0x80200000]
        li      a0,    0x202
        slli    a0,    a0, 20      //a0 = 0x20200000
        li      a1,    0x800
        slli    a1,    a1, 20      //a1 = 0x80000000
        li      a2,    0x802
        slli    a2,    a2, 20      //a2 = 0x80200000
        load_data a0,a1,a2

        //load qemu_sbi.dtb
        //[0x20080000:0x20100000] -->
[0x82200000:0x82280000]
        li      a0,    0x2008
        slli    a0,    a0, 16      //a0 = 0x20080000
        li      a1,    0x822
        slli    a1,    a1, 20      //a1 = 0x82200000
        li      a2,    0x8228
        slli    a2,    a2, 16      //a2 = 0x82280000
        load_data a0,a1,a2

        csrr    a0, mhartid
        li      t0,    0x0
        beq     a0, t0, _no_wait
        loop    0x1000
_no_wait:
        li      a1,    0x822
        slli    a1,    a1, 20      //a1 = 0x82200000

```

```

li      t0, 0x800
      slli      t0,      t0, 20      //t0 = 0x80000000
jr      t0

.end

```

这段代码会被加载到flash执行，主要工作是加载opensbi的固件和设备树，然后跳转到DRAM处执行opensbi的代码

#### 1. 宏定义：

- `loop,cunt` 宏用于执行循环，根据给定的次数进行空操作。
- `load_data` 宏用于从源地址加载数据，并将其存储到目标地址，直到目标地址达到结束地址。

#### 2. `_start` 函数：

- 加载 `opensbi_fw.bin` 文件的内容到指定内存区域 `[0x80000000:0x80200000]`。
- 加载 `qemu_sbi.dtb` 文件的内容到指定内存区域 `[0x82200000:0x82280000]`。
- 获取当前处理器的 ID，并与零进行比较。如果相等，执行 `_no_wait` 标签处的代码。
- `_no_wait` 标签处的代码加载一个地址到寄存器 `a1`，然后将控制权跳转到寄存器 `t0` 指向的地址。

这里固件的地址和在build.sh合成的固件的地址相关，下面就来修改build.sh

## 3.4 合成固件

---

在build.sh中新增如下代码：

## shell

```
#编译 opensbi
if [ ! -d "$SHELL_FOLDER/output/opensbi" ]; then
mkdir $SHELL_FOLDER/output/opensbi
fi
cd $SHELL_FOLDER/opensbi-1.2
make CROSS_COMPILE=$CROSS_PREFIX- PLATFORM=quard_star
cp -r $SHELL_FOLDER/opensbi-1.2/build/platform/quard_star/firmware/*.bin
$SHELL_FOLDER/output/opensbi/

# 生成sbi.dtb
cd $SHELL_FOLDER/dts
dtc -I dts -O dtb -o $SHELL_FOLDER/output/opensbi/quard_star_sbi.dtb quard_star_sbi.dts

# 合成firmware固件
if [ ! -d "$SHELL_FOLDER/output/fw" ]; then
mkdir $SHELL_FOLDER/output/fw
fi
cd $SHELL_FOLDER/output/fw
rm -rf fw.bin
# 填充 32K的0
dd of=fw.bin bs=1k count=32k if=/dev/zero
# 写入 lowlevel_fw.bin 偏移量地址为 0
dd of=fw.bin bs=1k conv=notrunc seek=0
if=$SHELL_FOLDER/output/lowlevelboot/lowlevel_fw.bin
# 写入 quard_star_sbi.dtb 地址偏移量为 512K, 因此 fdt的地址偏移量为 0x80000
dd of=fw.bin bs=1k conv=notrunc seek=512
if=$SHELL_FOLDER/output/opensbi/quard_star_sbi.dtb
# 写入 fw_jump.bin 地址偏移量为 2K*1K= 0x2000000, 因此 fw_jump.bin的地址偏移量为
0x2000000
dd of=fw.bin bs=1k conv=notrunc seek=2k if=$SHELL_FOLDER/output/opensbi/fw_jump.bin
```

- 首先编译`Opensbi`，在编译时需要指定`PLATFORM=quard_star`
- 然后编译生成`dtb`
- 最后合成`firmware`固件

根据dd命令的使用可以看见将`quard_star_sbi.dtb`写入到了偏移地址为`0x80000`的地方，用于这个固件会被加载到flash处，所以设备树的地址为：`0x20080000`

`fw_jump.bin` 地址偏移量为  $2K \times 1K = 0x2000000$ ，因此 `fw_jump.bin`的地址偏移量为 `0x2000000`，所以`opensbi`固件的地址为：`0x20200000`

`lowlevel_fw.bin` 偏移量地址为 0，因此`lowlevel_fw.bin` 的地址为：`0x20000000`

这里要说明一下为啥要使用`dd of=fw.bin bs=1k count=32k if=/dev/zero` 来填充填充 32K 的0，这是qemu使用-drive加载的固件的大小要大于等于定义的flash的大小，不然会报错。

## 3.5 编译运行

---

run.sh修改

shell

```
SHELL_FOLDER=$(cd "$(dirname "$0");pwd)

$SHELL_FOLDER/output/qemu/bin/qemu-system-riscv64 \
-M quard-star \
-m 1G \
-smp 8 \
-bios none \
-drive
if=pflash,bus=0,unit=0,format=raw,file=$SHELL_FOLDER/output/fw/fw.bin \
-d in_asm -D qemu.log \
-nographic --parallel none \
```

这里添加了一个qemu.log会生成汇编代码，可以根据这个log来查看固件是否有正确的加载、跳转。

依次执行.build.sh，.run.sh，出现如下就代表移植成功了：



```
timer@DESKTOP-JI9EVEH:~/quard-star$ ./run.sh
```

OpenSBI v1.2



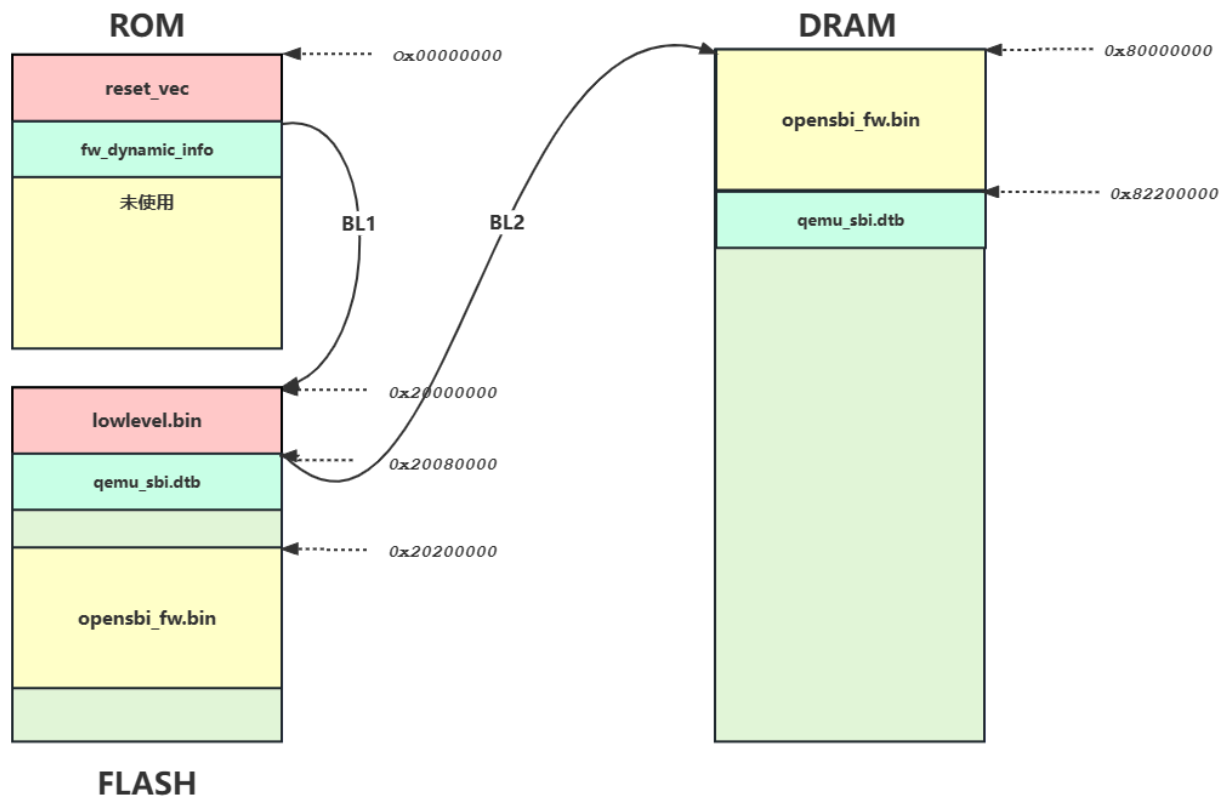
```
Platform Name           : riscv-quard-star,qemu
Platform Features       : medeleg
Platform HART Count     : 8
Platform IPI Device     : aclint-mswi
Platform Timer Device   : aclint-mtimer @ 100000000Hz
Platform Console Device : uart8250
Platform HSM Device     : ---
Platform PMU Device     : ---
Platform Reboot Device  : ---
Platform Shutdown Device : ---
Firmware Base          : 0x80000000
Firmware Size          : 252 KB
Runtime SBI Version    : 1.0

Domain0 Name           : root
Domain0 Boot HART      : 0
Domain0 HARTs          : 0*,1*,2*,3*,4*,5*,6*,7*
Domain0 Region00       : 0x00000000002000000-0x0000000000200ffff (I)
Domain0 Region01       : 0x000000000080000000-0x00000000008003ffff ()
Domain0 Region02       : 0x00000000000000000-0xffffffffffffffff (R,W,X)
Domain0 Next Address   : 0x0000000000000000
Domain0 Next Arg1      : 0x00000000008220000
Domain0 Next Mode      : S-mode
Domain0 SysReset       : yes

Boot HART ID           : 0
Boot HART Domain       : root
Boot HART Priv Version : v1.12
Boot HART Base ISA     : rv64imafdch
Boot HART ISA Extensions : time,sstc
Boot HART PMP Count    : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count   : 16
Boot HART MIDELEG      : 0x00000000000001666
Boot HART MEDELEG      : 0x0000000000f0b509
```

可以看到显示Opensbi的版本为v1.2，固件的起始地址为：0x80000000，大小为252kb。

所以现在的内存布局如下：



## 4. 结语

关于Opensbi还有很多值得学习的地方，后续有时间可深入分析其源码，这里先把它跑起来。

文章作者: Timer

文章链接: <https://yanglianoo.github.io/2023/06/22/为quard-star移植opensbi-1-2/>

版权声明: 本博客所有文章除特别声明外，均采用 CC BY-NC-SA 4.0 许可协议。转载请注明来自 TimerのBlog!

相关推荐