基于Opensbi服务完成控制台输出 | TimerのBlog

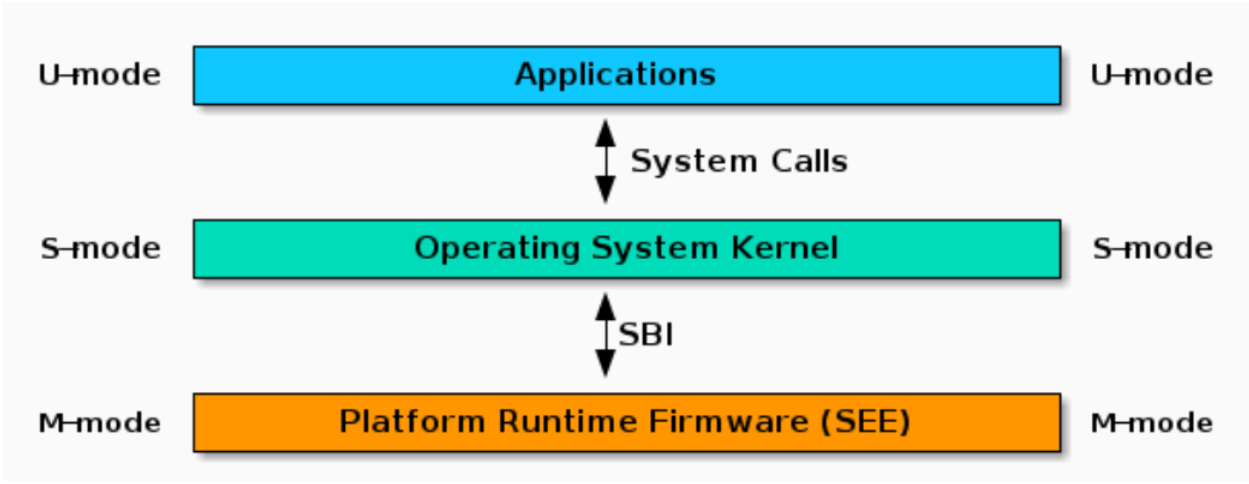
yanglianoo.github.io/2023/07/03/基于Opensbi服务完成控制台输出

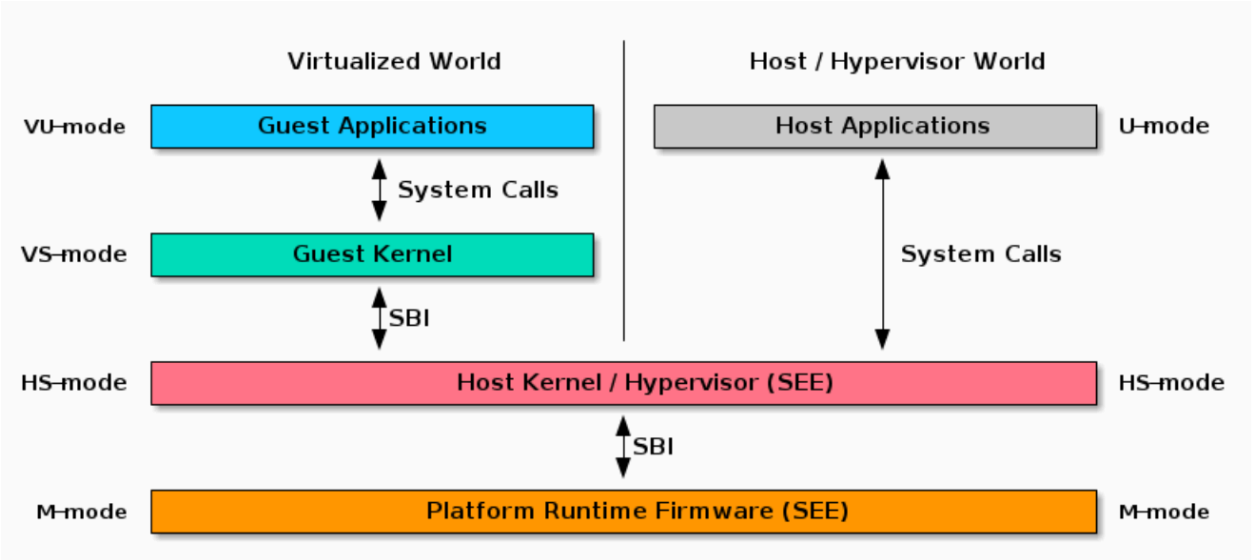
2023年7月3日

在上一篇文章中我们移植了Opensbi，大概了解了Opensbi是用来干嘛的，这一篇文章我们来详细介绍一下RISC-V Supervisor Binary Interface，即SBI。并且开启手写操作系统之路，首先利用Opensbi提供的服务来是实现串口打印。

1. RISC-V Supervisor Binary Interface

SBI允许在所有RISC-V实现上，通过定义平台（或虚拟化管理程序）特定功能的抽象，使监管者模式（S模式或VS模式）的软件具备可移植性。简单来说就是RISCV官方定义了一个规范接口，运行在S模式或VS模式的软件如os可以使用这些标准接口使得能够在不同的硬件平台上具有良好的移植性而不用去适配。SBI有两种架构，一种是CPU未启动虚拟化拓展，一种是启动了虚拟化功能的CPU。





如上图，SBI就是M模式和S模式之间的桥梁，是一套接口规范，我们已未启动虚拟化即未支持H拓展的CPU为例子。Opensbi就是上图中的SEE，向上给OS提供了接口，这些接口可以认为是不同的SBI函数，通过ecall指令来进行调用。所有的SBI函数共享一种二进制编码方式。

sbi规范到现在已经有两个大版本：v0.1 v0.2。为了保持兼容性，SBI扩展ID（EID）和SBI函数ID（FID）被编码为有符号的32位整数。新版本为0.2，在0.2版本中，函数调用的规定如下：

- 在监管者和SEE之间，使用ECALL作为控制传输指令，监管者就是S模式的软件程序
- a7编码SBI扩展ID（EID）
- a6编码SBI函数ID（FID），对于任何在a7中编码的SBI扩展，其定义在SBI v0.2之后。
- 在SBI调用期间，除了a0和a1寄存器外，所有寄存器都必须由被调用方保留。
- SBI函数必须在a0和a1中返回一对值，其中a0返回错误代码。类似于返回C结构体。

  c

```
struct
sbiret
 {
 long
error;
 long
value;
};
```

错误类型	值
SBI_SUCCESS 成功	0
SBI_ERR_FAILED 失败	-1
SBI_ERR_NOT_SUPPORTED 不支持操作	-2
SBI_ERR_INVALID_PARAM 非法参数	-3
SBI_ERR_DENIED 拒绝	-4
SBI_ERR_INVALID_ADDRESS 非法地址	-5
SBI_ERR_ALREADY_AVAILABLE (资源)已可用	-6
SBI_ERR_ALREADY_STARTED (操作)已启动	-7
SBI_ERR_ALREADY_STOPPED (操作)已停止	-8

EID和FID共同决定了调用的函数是什么，其中基本拓展函数如下：EID都为0x10

函数名	SBI 版本	FID	EID	用途
sbi_get_sbi_spec_version	0.2	0	0x10	获取SBI规范版本
sbi_get_sbi_impl_id	0.2	1	0x10	获取SBI实现标识符
sbi_get_sbi_impl_version	0.2	2	0x10	获取SBI实现版本
sbi_probe_extension	0.2	3	0x10	探测SBI扩展功能
sbi_get_mvendorid	0.2	4	0x10	获取机器供应商标识符
sbi_get_marchid	0.2	5	0x10	获取机器体系结构标识符
sbi_get_mimpid	0.2	6	0x10	获取机器实现标识符ID

传统的 SBI 扩展与 SBI v0.2（或更高版本）规范相比，遵循略微不同的调用约定，其中：

- a6 寄存器中的 SBI 函数ID 字段被忽略，因为这些被编码为多个 SBI 扩展 ID。
- a1寄存器中不返回任何值。
- 在 SBI 调用期间，除 a0 寄存器外的所有寄存器都必须由被调用者保留。
- a0 寄存器中返回的值是特定于 SBI 传统扩展的。
- SBI 实现在监管者访问内存时发生的页面和访问故障会被重定向回监管者，并且 sepc 寄存器指向故障的 ECALL指令。

函数名	SBI 版本	FID	EID	替代 EID	函数用途
sbi_set_timer	0.1	0	0x00	0x54494D45	设置时钟
sbi_console_putchar	0.1	0	0x01	N/A	控制台字符输出
sbi_console_getchar	0.1	0	0x02	N/A	控制台字符输入
sbi_clear_ipi	0.1	0	0x03	N/A	清除IPI
sbi_send_ipi	0.1	0	0x04	0x735049	发送IPI
sbi_remote_fence_i	0.1	0	0x05	0x52464E43	远程FENCE.I
sbi_remote_sfence_vma	0.1	0	0x06	0x52464E43	远程 SFENCE.VMA

函数名	SBI 版本	FID	EID	替代 EID	函数用途
sbi_remote_sfence_vma_asid	0.1	0	0x07	0x52464E43	远程 SFENCE.VMA（指定地址空间标识符）
sbi_shutdown	0.1	0	0x08	0x53525354	系统关闭
保留			0x09-0x0F		

我们使用到的sbi的函数不多，初步了解这些就够了，sbi的所有的详细规范定义请参考如下文档：

2. 基于Opensbi完成控制台输出

目标：在S模式下使用`ecall`指定调用`sbi_console_putchar`函数向控制台打印字符

2.1 untrusted-domain 起始地址修改

在上一篇文章中，我们为`quard_star`划分了`domain`，`opensbi`是运行在`untrusted-domain`中的，在`quard_star`的设备树文件中指定了两个`domain`的地址参数：

dts

```
next-arg1 = <0x0
0x82200000>;
next-addr = <0x0
0x82000000>;
```

这两个参数一个是下级程序的参数，一个是下级程序的起始地址，在前面提到我们的下级程序是uboot也可以直接是内核，为了使这个项目更有意义，我们来手写一个操作系统，就不使用uboot和linux系统了，关于如何移植uboot和linux内核请按照第一篇中参考博客中的方法继续走下去。在我的代码仓库中也移植成功了，可以参考一下移植uboot的commit。

> 这里需要说明一下在移植uboot时，使用riscv64-unknown-elf-gcc这个编译器是不行的，编译uboot需要riscv64-unknown-linux-gnu-gcc，关于交叉编译工具链的编译配置这里我就不详解了，网上有许多教程。我的项目中使用的uboot版本为`uboot-2023.04`

```
# # 编译uboot
# if [ ! -d "$SHELL_FOLDER/output/uboot" ]; then
# mkdir $SHELL_FOLDER/output/uboot
# fi
# cd $SHELL_FOLDER/u-boot-2023.04
# make CROSS_COMPILE=riscv64-unknown-linux-gnu- qemu-riscv64_smode_defconfig
# make CROSS_COMPILE=riscv64-unknown-linux-gnu-  -j16
# cp $SHELL_FOLDER/u-boot-2023.04/u-boot $SHELL_FOLDER/output/uboot/u-boot.elf
# cp $SHELL_FOLDER/u-boot-2023.04/u-boot.map $SHELL_FOLDER/output/uboot/u-boot.map
# cp $SHELL_FOLDER/u-boot-2023.04/u-boot.bin $SHELL_FOLDER/output/uboot/u-boot.bin
# riscv64-unknown-linux-gnu-objdump --source --demangle --disassemble --reloc --wide $SHELL_FOLDER/output/uboot/u-boot.elf > $SHELL_FOLDER/output/uboot/u-boot.lst

# # 生成uboot.dtb
# cd $SHELL_FOLDER/dts
# dtc -I dts -O dtb -o $SHELL_FOLDER/output/uboot/quard_star_uboot.dtb quard_star_uboot.dts
```

因此下级程序即为我们编写的OS，这里修改一下下级程序的地址和参数，将下级程序的起始地址改成了`0x80200000`，下级程序的参数随便给，这里先留着不修改吧

plaintext

```
next-arg1 = <0x0
0x82000000>;
next-addr = <0x0
0x80200000>;
```

2.2 创建OS

在quard_star目录下新建os文件夹，在此文件夹中编写我们的操作系统程序，然后新建了这些文件：

sh

```
timer@DESKTOP-JI9EVEH:~/quard-star/os$ ls
Makefile   entry.S  main.c  os.ld  sbi.c
sbi.h
```

2.2.1 entry.S

plaintext

```
      .section .text.entry
      .globl _start
_start:
    la sp, boot_stack_top
    call os_main

    .section .bss.stack
    .globl
boot_stack_lower_bound
boot_stack_lower_bound:
    .space 4096 * 16
    .globl boot_stack_top
boot_stack_top:
```

这段代码主要就是定义了一个大小为 4096 * 16 字节 = 64kb的连续内存空间，用作栈空间。将栈指针sp指向栈顶位置，然后调用os_main这个函数，os_main函数定义在main.c中

2.2.2 sbi.c 和 sbi.h

c

```c
/*sbi.h*/
#ifndef __SBI_H__
#define __SBI_H__

enum sbi_ext_id {
        SBI_EXT_0_1_SET_TIMER = 0x0,
        SBI_EXT_0_1_CONSOLE_PUTCHAR = 0x1,
        SBI_EXT_0_1_CONSOLE_GETCHAR = 0x2,
        SBI_EXT_0_1_CLEAR_IPI = 0x3,
        SBI_EXT_0_1_SEND_IPI = 0x4,
        SBI_EXT_0_1_REMOTE_FENCE_I = 0x5,
        SBI_EXT_0_1_REMOTE_SFENCE_VMA =
0x6,
        SBI_EXT_0_1_REMOTE_SFENCE_VMA_ASID
= 0x7,
        SBI_EXT_0_1_SHUTDOWN = 0x8,
        SBI_EXT_BASE = 0x10,
        SBI_EXT_TIME = 0x54494D45,
        SBI_EXT_IPI = 0x735049,
        SBI_EXT_RFENCE = 0x52464E43,
        SBI_EXT_HSM = 0x48534D,
        SBI_EXT_SRST = 0x53525354,
        SBI_EXT_PMU = 0x504D55,
};

/* sbi 返回结构体*/
struct sbiret {
        long error;
        long value;
};

#endif
```

c

```c
/*sbi.c*/
#include "sbi.h"
#include "stdint.h"
struct sbiret sbi_ecall(int ext, int fid, unsigned long arg0,
                        unsigned long arg1, unsigned long arg2,
                        unsigned long arg3, unsigned long arg4,
                        unsigned long arg5)
{
        struct sbiret ret;

    //使用GCC的扩展语法，用于将一个值存储到RISC-V架构中的寄存器a0中。
        register uintptr_t a0 asm ("a0") = (uintptr_t)(arg0);
        register uintptr_t a1 asm ("a1") = (uintptr_t)(arg1);
        register uintptr_t a2 asm ("a2") = (uintptr_t)(arg2);
        register uintptr_t a3 asm ("a3") = (uintptr_t)(arg3);
        register uintptr_t a4 asm ("a4") = (uintptr_t)(arg4);
        register uintptr_t a5 asm ("a5") = (uintptr_t)(arg5);
        register uintptr_t a6 asm ("a6") = (uintptr_t)(fid);
        register uintptr_t a7 asm ("a7") = (uintptr_t)(ext);
        asm volatile ("ecall"
                      : "+r" (a0), "+r" (a1)
                      : "r" (a2), "r" (a3), "r" (a4), "r" (a5), "r" (a6),
"r" (a7)
                      : "memory");
        ret.error = a0;
        ret.value = a1;

        return ret;
}


/**
 * sbi_console_putchar() - Writes given character to the console device.
 * @ch: The data to be written to the console.
 *
 * Return: None
 */
void sbi_console_putchar(int ch)
{
        sbi_ecall(SBI_EXT_0_1_CONSOLE_PUTCHAR, 0, ch, 0, 0, 0, 0, 0);
}
```

在sbi的头文件中定义了EID的枚举变量和sbi 的返回结构体，然后再sbi.c中定义了一个sbi_ecall的函数用于调用Opensbi提供的服务，最后定义了sbi_console_putchar函数传入想要输出的字符，然后传入EID和FID，去查上面的表EID=0x01,FID=0。

> 这里的代码我是抄的uboot的，有兴趣的可以去看一下uboot的源码
>
> 参考链接：OpenSBI - Messy Notes (chsgcxy.github.io)

2.2.3 main.c

c

```
extern
sbi_console_putchar(int ch);

void os_main()
{
    sbi_console_putchar('h');
    sbi_console_putchar('e');
    sbi_console_putchar('l');
    sbi_console_putchar('l');
    sbi_console_putchar('o');
    sbi_console_putchar('!');
}
```

main.c定义了os_main()函数，在os_main()函数中依次打印字符输出“hello!”

2.2.4 os.ld

plaintext

```
OUTPUT_ARCH(riscv)
ENTRY(_start)

MEMORY
{
        ram (rxai!w) : ORIGIN = 0x80200000, LENGTH
= 128M
}
SECTIONS
{
        .text : {
                *(.text .text.*)
        } >ram

        .rodata : {
                *(.rodata .rodata.*)
        } >ram

        .data : {
                . = ALIGN(4096);
                *(.sdata .sdata.*)
                *(.data .data.*)
                PROVIDE(_data_end = .);
        } >ram

        .bss :{
                *(.sbss .sbss.*)
                *(.bss .bss.*)
                *(COMMON)
        } >ram

}
```

链接脚本如上，其中os的可执行文件会被链接到0x80200000的位置

2.2.5 makefile

makefile

```
CROSS_COMPILE = riscv64-unknown-elf-
CFLAGS = -nostdlib -fno-builtin

# riscv64-unknown-elf-gcc 工具链可以同时编译汇编和
C 代码
CC = ${CROSS_COMPILE}gcc
OBJCOPY = ${CROSS_COMPILE}objcopy
OBJDUMP = ${CROSS_COMPILE}objdump

SRCS_ASM = \
        entry.S

SRCS_C = \
        sbi.c \
        main.c \

# 将源文件替换为 .o 文件
OBJS = $(SRCS_ASM:.S=.o)
OBJS += $(SRCS_C:.c=.o)


os.elf: ${OBJS}
        ${CC} ${CFLAGS} -T os.ld  -o os.elf $^
        ${OBJCOPY} -O binary os.elf os.bin

%.o : %.c
        ${CC} ${CFLAGS} -c -o $@ $<

%.o : %.S
        ${CC} ${CFLAGS} -c -o $@ $<


.PHONY : clean
clean:
        rm -rf *.o *.bin *.elf
```

编译链接生成os.bin

3. 测试

首先修改一下build.sh，先编译os，新增如下内容：

shell

```
# 编译os
if [ ! -d "$SHELL_FOLDER/output/os" ]; then
mkdir $SHELL_FOLDER/output/os
fi
cd $SHELL_FOLDER/os
make
cp $SHELL_FOLDER/os/os.bin
$SHELL_FOLDER/output/os/os.bin
make clean
```

合成固件：

```
# 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=$SHELL_FOLDER/output/opensbi/quard_star_sbi.dtb
# 写入 uboot.dtb,地址偏移量为 1K*1K = 0x100000
dd of=fw.bin bs=1k conv=notrunc seek=1K if=$SHELL_FOLDER/output/uboot/quard_star_uboot.dtb
# 写入 fw_jump.bin 地址偏移量为 2K*1K= 0x200000，因此 fw_jump.bin的地址偏移量为  0x200000
dd of=fw.bin bs=1k conv=notrunc seek=2k if=$SHELL_FOLDER/output/opensbi/fw_jump.bin
# 写入 trusted_domain.bin,地址偏移量为 1K*4K = 0x400000，因此 trusted_domain.bin的地址偏移量为  0x400000
dd of=fw.bin bs=1k conv=notrunc seek=4K if=$SHELL_FOLDER/output/trusted_domain/trusted_fw.bin
# 写入 uboot.bin,地址偏移量为 1K*8K =  0x800000
#dd of=fw.bin bs=1k conv=notrunc seek=8K if=$SHELL_FOLDER/output/uboot/u-boot.bin
# 写入 os.bin,地址偏移量为 1K*8K =  0x800000
dd of=fw.bin bs=1k conv=notrunc seek=8K if=$SHELL_FOLDER/output/os/os.bin
```

然后将修改boot/start.s将os.bin加载到0x80200000的位置

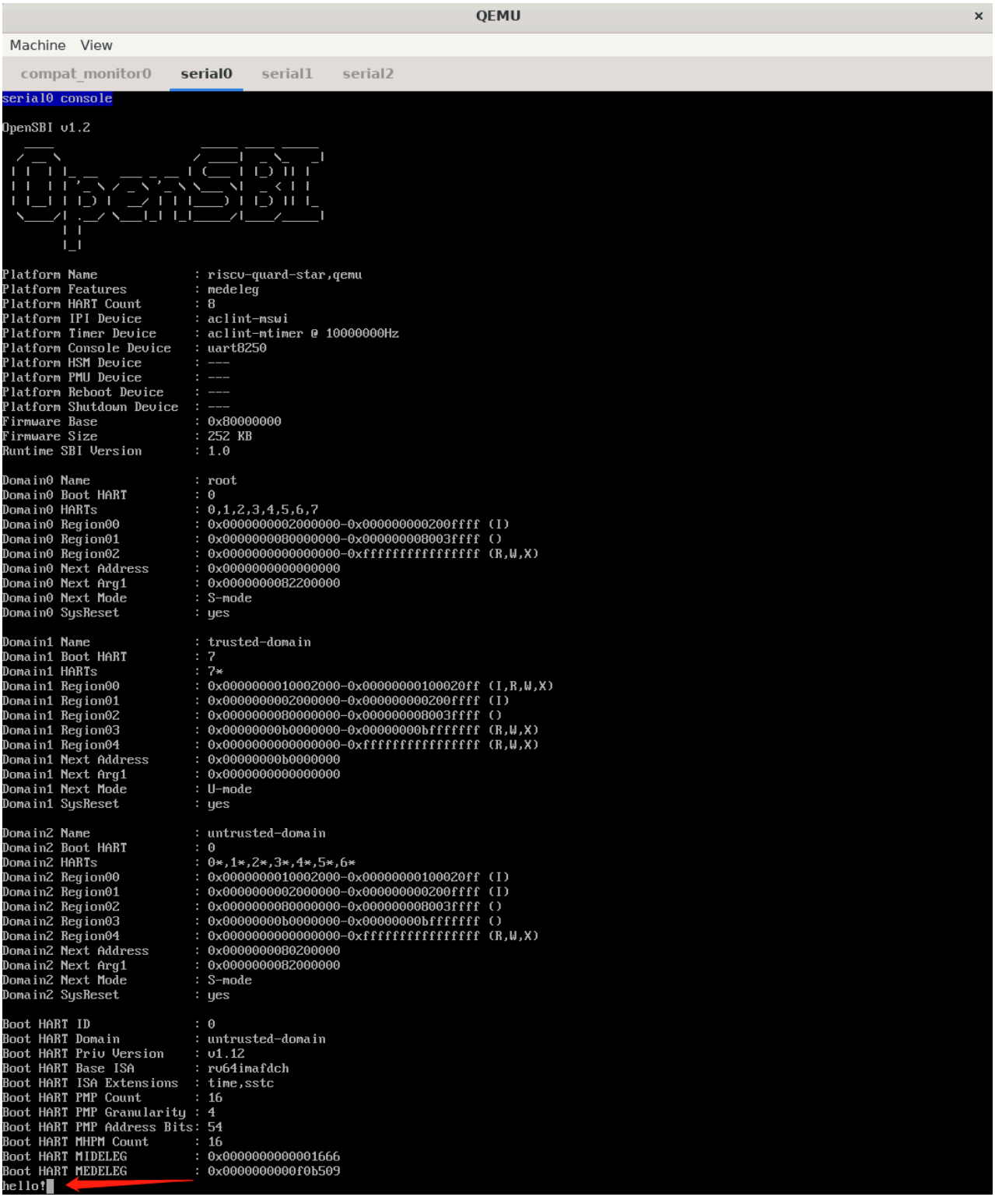
```
//load os.bin
//[0x20800000:0x20C00000] --> [0x80200000:0x80600000]
li      a0, 0x208
slli    a0, a0, 20      //a0 = 0x20800000
li      a1, 0x802
slli    a1, a1, 20      //a1 = 0x80200000
li      a2, 0x806
slli    a2, a2, 20      //a2 = 0x80600000
load_data a0,a1,a2
```

编译运行：

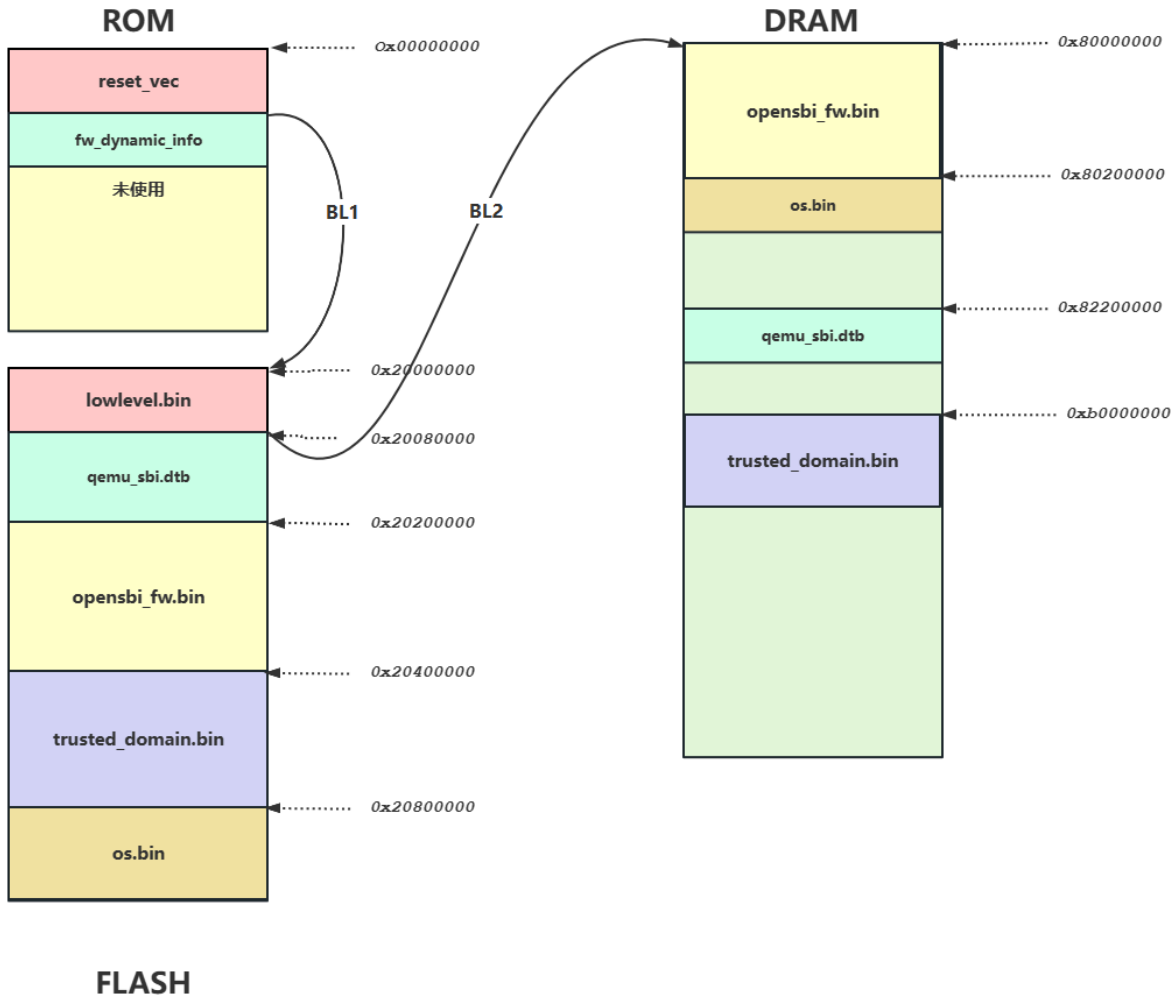
sh

```
./bu
ild.
sh
./ru
n.sh
```

运行结果如下，可以看见成功打印“hello！”



现在的内存布局如下：



相关推荐