基于opensbi为quard_star创建domain

yanglianoo.github.io/2023/06/24/基于opensbi为quard-star创建domain

2023年6月24日

1. Opensbi之domain机制

OpenSBI（Open Source Supervisor Binary Interface）的 domain 机制是一种用于管理和隔离不同软件实体（例如操作系统、虚拟机）的机制。它提供了一种在系统中划分资源和权限的方法，以确保软件实体之间的相互隔离和安全性。

在 OpenSBI 中，domain 是一种逻辑上的实体，它代表了一个软件实体，可以是一个操作系统、一个虚拟机或其他一些执行环境。每个 domain 都有自己的一组资源和权限，包括处理器（Hart）、内存、设备和中断等。domain 之间是相互隔离的，它们不能直接访问或干扰彼此的资源。

OpenSBI 的 domain 机制通过以下方式实现：

1. Domain ID：每个 domain 都有一个唯一的标识符，称为 Domain ID。它用于区分不同的 domain。
2. Hart Mask：OpenSBI 使用 Hart Mask 来表示哪些处理器属于特定的 domain。Hart Mask 是一个位图，每个位代表一个处理器，可以将相应的位设置为 1 表示该处理器属于某个 domain。
3. SBI 接口：OpenSBI 提供了一组 SBI（Supervisor Binary Interface）接口，用于 domain 之间的通信和资源管理。这些接口包括中断处理、内存管理、设备访问等，可以由 domain 使用来请求和管理资源。

通过 domain 机制，OpenSBI 可以实现不同软件实体的隔离和安全性。每个 domain 只能访问自己被授权的资源，并且不能越权访问其他 domain 的资源。这样可以确保系统的稳定性和安全性，并支持多个软件实体在同一硬件平台上共存和运行。

在opensbi的doc目录下的`domain_support.md`文档介绍了如何使用设备树来基于opensbi来划分domain，在文档中提到默认情况下，所有的 HART 都被分配给 `ROOT domain`。
OpenSBI 平台支持可以通过平台特定的回调函数提供 HART 到 domain 实例的分配。同时也可以使用设备树来定义`domain`，文档中举了一个设备树的例子如下：

plaintext

```
chosen {
    opensbi-domains {
        compatible = "opensbi,domain,config";

        tmem: tmem {
            compatible = "opensbi,domain,memregion";
            base = <0x0 0x80100000>;
            order = <20>;
        };

        tuart: tuart {
            compatible = "opensbi,domain,memregion";
            base = <0x0 0x10011000>;
            order = <12>;
            mmio;
            devices = <&uart1>;
        };

        allmem: allmem {
            compatible = "opensbi,domain,memregion";
            base = <0x0 0x0>;
            order = <64>;
        };

        tdomain: trusted-domain {
            compatible = "opensbi,domain,instance";
            possible-harts = <&cpu0>;
            regions = <&tmem 0x7>, <&tuart 0x7>;
            boot-hart = <&cpu0>;
            next-arg1 = <0x0 0x0>;
            next-addr = <0x0 0x80100000>;
            next-mode = <0x0>;
            system-reset-allowed;
        };

        udomain: untrusted-domain {
            compatible = "opensbi,domain,instance";
            possible-harts = <&cpu1 &cpu2 &cpu3 &cpu4>;
            regions = <&tmem 0x0>, <&tuart 0x0>, <&allmem
0x7>;
        };
    };
};

cpus {
    #address-cells = <1>;
    #size-cells = <0>;
    timebase-frequency = <10000000>;

    cpu0: cpu@0 {
        device_type = "cpu";
        reg = <0x00>;
        compatible = "riscv";
        opensbi-domain = <&tdomain>;
        ...
    };

    cpu1: cpu@1 {
```

```
        device_type = "cpu";
        reg = <0x01>;
        compatible = "riscv";
        opensbi-domain = <&udomain>;
        ...
    };

    cpu2: cpu@2 {
        device_type = "cpu";
        reg = <0x02>;
        compatible = "riscv";
        opensbi-domain = <&udomain>;
        ...
    };

    cpu3: cpu@3 {
        device_type = "cpu";
        reg = <0x03>;
        compatible = "riscv";
        opensbi-domain = <&udomain>;
        ...
    };

    cpu4: cpu@4 {
        device_type = "cpu";
        reg = <0x04>;
        compatible = "riscv";
        opensbi-domain = <&udomain>;
        ...
    };
};

uart1: serial@10011000 {
    ...
};
```

在这份设备树中需要定义如下三个节点：

- Domain Configuration Node

  这里就是要定义compatible = “opensbi,domain,config”;

- `Domain Memory Region Node`

  这里是定义和内存相关的节点，域内存区域设备树节点的属性如下：

  - compatible（必选）- 域内存区域的兼容字符串。该设备树属性的值应为“opensbi,domain,memregion”。
  - base（必选）- 域内存区域的基地址。该设备树属性应为2^order对齐的64位地址（即两个设备树单元）。
  - order（必选）- 域内存区域的阶数。该设备树属性应为32位值（即一个设备树单元），取值范围为 3 <= order <= __riscv_xlen。
  - mmio（可选）- 一个布尔标志，表示域内存区域是否为内存映射I/O（MMIO）区域。
  - devices（可选）- 设备列表，其中包含属于此域内存区域的设备设备树节点的句柄。
- `Domain Instance Node`

  - `compatible`（必选）-:域实例的兼容字符串。该设备树属性的值应为“opensbi,domain,instance”。

  - `possible-harts`（可选）- 域实例的CPU设备树节点句柄列表。该列表表示域实例的可能HART集合。

  - `regions`（可选）- 域实例的域内存区域设备树节点句柄和访问权限列表。每个列表条目都是一个设备树节点句柄和访问权限的对。访问权限以32位掩码表示，具有可读（BIT[0]）、可写（BIT[1]）、可执行（BIT[2]）和M模式（BIT[3]）的位。

  - `boot-hart`（可选）- 引导域实例的HART的设备树节点句柄。如果将冷启动HART分配给域实例，则忽略此设备树属性，并假定冷启动HART是域实例的引导HART。

  - `next-arg1`（可选）- 域实例的64位下一个引导阶段arg1。如果此设备树属性不可用且未将冷启动HART分配给域实例，则使用默认值0x0。如果此设备树属性不可用且将冷启动HART分配给域实例，则使用冷启动HART的下一个引导阶段arg1作为默认值。

  - `next-addr`（可选）- 域实例的64位下一个引导阶段地址。如果此设备树属性不可用且未将冷启动HART分配给域实例，则使用默认值0x0。如果此设备树属性不可用且将冷启动HART分配给域实例，则使用冷启动HART的下一个引导阶段地址作为默认值。

  - `next-mode`（可选）- 域实例的32位下一个引导阶段模式。该设备树属性的可能值为：0x1（s模式）和0x0（u模式）。如果此设备树属性不可用且未将冷启动HART分配给域实例，则使用默认值0x1。如果此设备树属性不可用且将冷启动HART分配给域实例，则使用冷启动HART的下一个引导阶段模式作为默认值。

  - `system-reset-allowed`（可选）- 一个布尔标志，表示是否允许域实例进行系统复位。

2. 为quard_star划分domain

2.1 修改quard_star设备树

dts

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

        opensbi-domains {  /* 定义opensbi-domains描述节点 */
            compatible = "opensbi,domain,config"; /* 节点名称 */

           tmem: tmem {   /* 定义内存节点 */
               compatible = "opensbi,domain,memregion";  /* 节点名称 */
               base = <0x0 0xb0000000>; /* 起始地址注意64位地址哦 */
               order = <28>; /* 内存大小即size=2^28 */
           };

           tuart: tuart {  /* 定义mmio节点 */
               compatible = "opensbi,domain,memregion";  /* 节点名称 */
               base = <0x0 0x10002000>; /* 起始地址 */
               order = <8>; /* size=2^8 */
               mmio;  /* mmio属性 */
               devices = <&uart2>; /* 关联到设备节点上 */
           };

            allmem: allmem { /* 定义内存节点，这个节点保护所有地址 */
                compatible = "opensbi,domain,memregion";
                base = <0x0 0x0>;
                order = <64>;
            };

           tdomain: trusted-domain { /* 定义domian节点 */
               compatible = "opensbi,domain,instance";  /* 节点名称 */
               possible-harts = <&cpu7>; /* dumian中允许使用的cpu core */
               regions = <&tmem 0x7>, <&tuart 0x7>, <&allmem 0x7>;/* 各个内存/mmio区域
的权限，3bit读写运行权限 0x7拥有全部权限 */
               boot-hart = <&cpu7>; /* domian中用于boot的core */
               next-arg1 = <0x0 0x00000000>; /* 下级程序的参数 */
                next-addr = <0x0 0xb0000000>; /* 下级程序的起始地址 */
                next-mode = <0x0>; /* 下级程序的允许模式 0为U模式，1为S模式 */
               system-reset-allowed; /* 允许复位 */
           };

            udomain: untrusted-domain {
                compatible = "opensbi,domain,instance";
                possible-harts = <&cpu0 &cpu1 &cpu2 &cpu3 &cpu4 &cpu5 &cpu6>;
                regions = <&tmem 0x0>, <&tuart 0x0>, <&allmem 0x7>;
                        boot-hart = <&cpu0>;
                next-arg1 = <0x0 0x82200000>;
                next-addr = <0x0 0x82000000>;
                next-mode = <0x1>;
                system-reset-allowed;
            };
        };
};
```

可以看见为quard_star划分了两个domain，一个为trusted-domain，使用了cpu7，下级程序的起始地址为0xb0000000，模式为U模式，这个domain可以用来运行类似freertos的实时操作系统；一个domain为untrusted-domain，这个domain使用了cpu0~cpu6，用于运行linux系统，下级程序的起始地址为0x82000000

2.2 编写domain测试代码

在quard_star目录下新建一个`trusted_domain`的文件夹，在此文件夹下新建`link.lds`和`startup.s`两个文件。

sh

```
timer@DESKTOP-JI9EVEH:~/quard-
star/trusted_domain$ ls
link.lds  startup.s
```

`link.lds`如下，这里需要注意的是运行起始地址和设备树中的保持一致，即：`0xb0000000`

plaintext

```
OUTPUT_ARCH( "riscv" )

ENTRY( _start )

MEMORY
{
        ddr (rxai!w) : ORIGIN = 0xb0000000, LENGTH
= 256M
}

SECTIONS
{
  .text :
  {
    KEEP(*(.text))
  } >ddr
}
```

`startup.s`如下：

avrasm

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

_start:
        li              t0,
0x100
        slli    t0,     t0, 20
        li              t1,
0x200
        slli    t1,     t1, 4
        add     t0, t0, t1
        li              t1,
'H'
        sb              t1,
0(t0)
        li              t1,
'e'
        sb              t1,
0(t0)
        li              t1,
'l'
        sb              t1,
0(t0)
        li              t1,
'l'
        sb              t1,
0(t0)
        li              t1,
'o'
        sb              t1,
0(t0)
        li              t1,
' '
        sb              t1,
0(t0)
        li              t1,
'Q'
        sb              t1,
0(t0)
        li              t1,
'u'
        sb              t1,
0(t0)
        li              t1,
'a'
        sb              t1,
0(t0)
        li              t1,
'r'
        sb              t1,
0(t0)
        li              t1,
'd'
        sb              t1,
0(t0)
        li              t1,
' '
```

```
        sb              t1,
0(t0)
        li              t1,
'S'
        sb              t1,
0(t0)
        li              t1,
't'
        sb              t1,
0(t0)
        li              t1,
'a'
        sb              t1,
0(t0)
        li              t1,
'r'
        sb              t1,
0(t0)
        li              t1,
' '
        sb              t1,
0(t0)
        li              t1,
'b'
        sb              t1,
0(t0)
        li              t1,
'o'
        sb              t1,
0(t0)
        li              t1,
'a'
        sb              t1,
0(t0)
        li              t1,
'r'
        sb              t1,
0(t0)
        li              t1,
'd'
        sb              t1,
0(t0)
        li              t1,
'!'
        sb              t1,
0(t0)
        li              t1,
'\r'
        sb              t1,
0(t0)
        li              t1,
'\n'
        sb              t1,
0(t0)
_loop:
        j               _loop

    .end
```

这里需要注意的是串口的输出地址为0x10002000，即UART2的地址，和设备树中tuart的地址保持一致

这里需要生成trusted_domain的固件并将其加载到0xb0000000处执行，所以先修改build.sh

先编译生成trusted_domain.bin

shell

```
#编译trusted_domain
if [ ! -d "$SHELL_FOLDER/output/trusted_domain" ]; then
mkdir $SHELL_FOLDER/output/trusted_domain
fi
cd $SHELL_FOLDER/trusted_domain
$CROSS_PREFIX-gcc -x assembler-with-cpp -c startup.s -o
$SHELL_FOLDER/output/trusted_domain/startup.o
$CROSS_PREFIX-gcc -nostartfiles -T./link.lds -Wl,-
Map=$SHELL_FOLDER/output/trusted_domain/trusted_fw.map -Wl,--gc-sections
$SHELL_FOLDER/output/trusted_domain/startup.o -o
$SHELL_FOLDER/output/trusted_domain/trusted_fw.elf
$CROSS_PREFIX-objcopy -O binary -S $SHELL_FOLDER/output/trusted_domain/trusted_fw.elf
$SHELL_FOLDER/output/trusted_domain/trusted_fw.bin
$CROSS_PREFIX-objdump --source --demangle --disassemble --reloc --wide
$SHELL_FOLDER/output/trusted_domain/trusted_fw.elf >
$SHELL_FOLDER/output/trusted_domain/trusted_fw.lst
```

然后合成fw.bin，新增一行如下：

```
# 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=$SHELL_FOLDER/output/opensbi/quard_star_sbi.dtb
# 写入 fw_jump.bin 地址偏移量为 2K*1K= 0x200000，因此 fw_jump.bin的地址偏移量为  0x200000
dd of=fw.bin bs=1k conv=notrunc seek=2k if=$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
```

将trusted_domain.bin写入到了地址偏移为：0x400000的地方，下一步就要将固件加载到0xb0000000，因此还需要修改一下boot下的start.s，修改如下：

avrasm

```
//load trusted_fw.bin
//[0x20400000:0x20800000] -->
[0xb0000000:0xb0400000]
   li           a0,     0x204
slli    a0,     a0, 20      //a0 = 0x20400000
   li           a1,     0xb00
slli    a1,     a1, 20      //a1 = 0xb0000000
   li           a2,     0xb04
slli    a2,     a2, 20      //a2 = 0xb0400000
load_data a0,a1,a2
```

2.3 测试

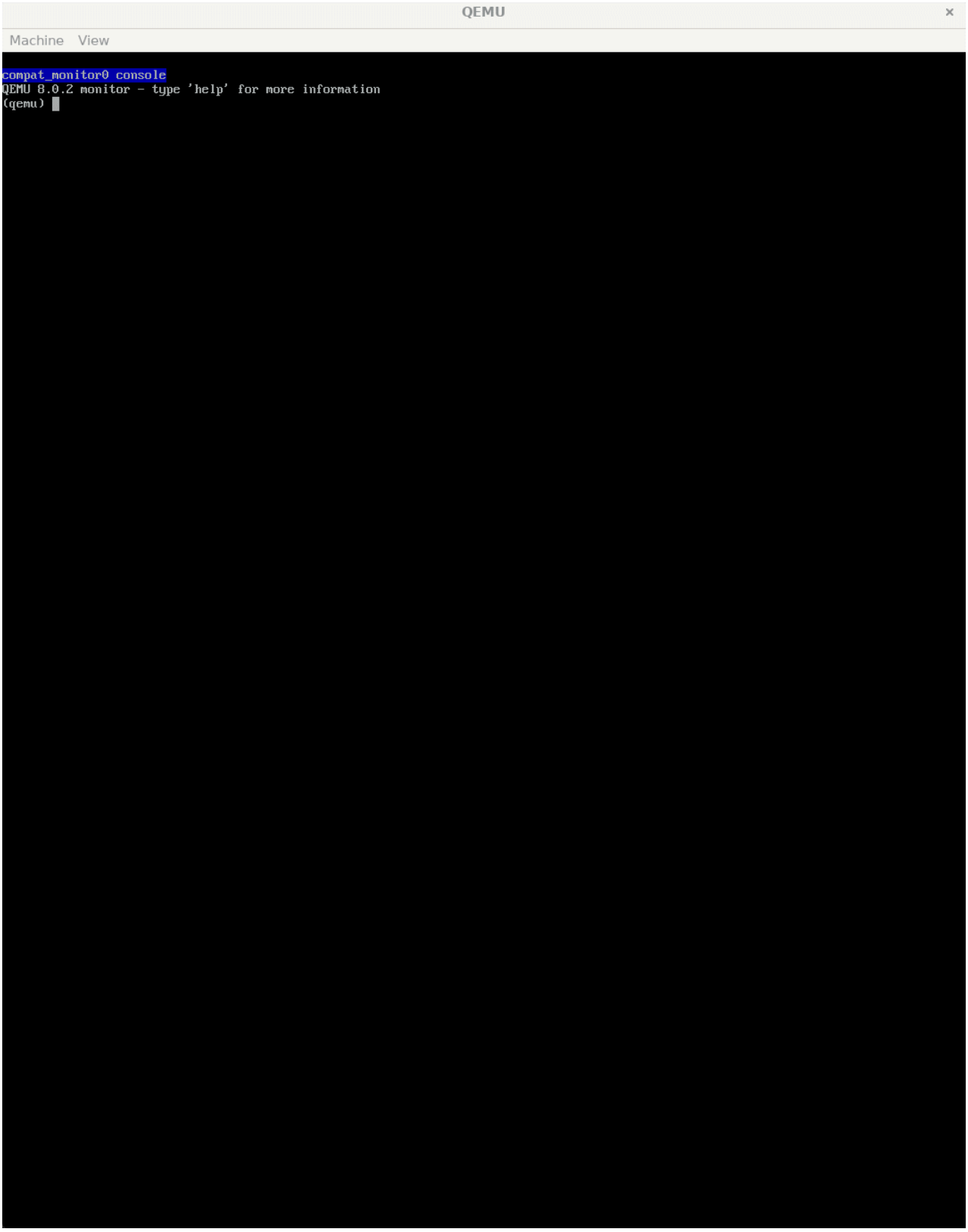
run.sh修改：

shell

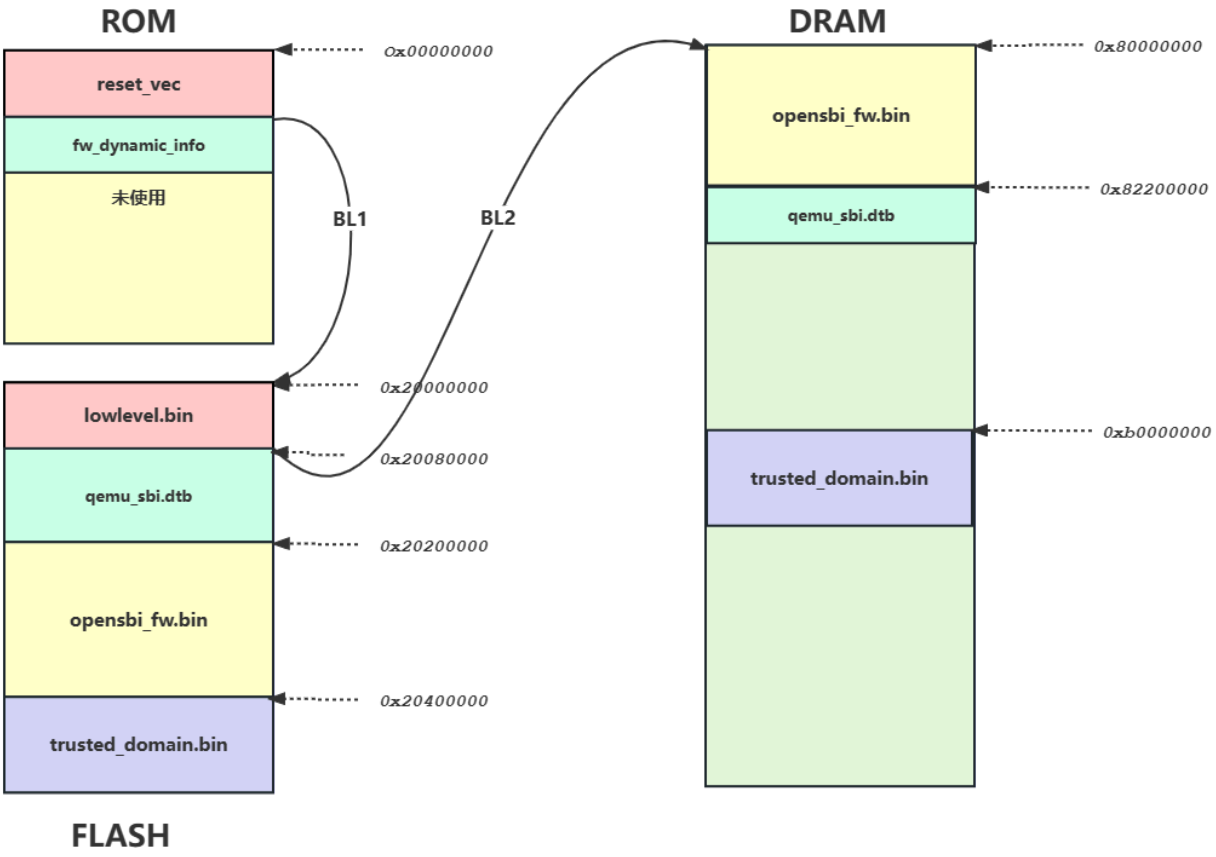
```
SHELL_FOLDER=$(cd "$(dirname "$0")";pwd)
DEFAULT_VC="1080x1920"

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

用DEFAULT_VC来指定了qemu显示的分辨率，这个分辨率我随便设置的，还需要新增三个-serial选项让qemu输出三个串口终端，运行结果如下：



所以现在的内存布局如下：



相关推荐