

虚拟soc使用手册

0. 概述

- 0.1 为什么需要虚拟soc
- 0.2 项目简介
- 0.3 重要约定
- 0.4 总结
- 0.5 修订记录

1. 虚拟soc的使用

- 1.1 安装
- 1.2 启动命令
 - 1.2.0 选择虚拟板卡型号，及常用配置
 - 1.2.1 虚拟网卡
 - 1.2.2 挂载虚拟磁盘
 - 1.2.3 挂载sd卡
 - 1.2.4 加载数据到内存
 - 1.2.5 引导linux
 - 1.2.6 串口重定向到tcp端口
- 1.3 控制快捷键
- 1.4 地址空间
- 1.5 总结

2. 调试baremetal应用

- 2.1 加载并运行应用程序
- 2.2 启用gdb服务
- 2.3 使用vscode调试应用
- 2.5 总结

3. 调试linux

- 3.1 引导linux启动
- 3.2 ssh连接&scp传输文件
- 3.3 tftp文件传输
- 3.4 nfs文件共享
- 3.5 访问host文件夹
- 3.6 gdb调试linux内核
- 3.7 使用vscode调试linux内核
- 3.8 总结

0. 概述

0.1 为什么需要虚拟soc

在传统的soc开发/验证的流程中（图1），soc开发版（验证平台）是必不可少的硬件条件，且整个流程遵循以下几个环节：

- 1-> 在host上编辑代码、交叉编译程序，生成可执行文件
- 2-> 在开发板上执行代码
- 3-> jlink、T32、openocd、ds等，下载、调试代码
- 4-> 串口、网口、USB、SPI等，用于host与soc通信

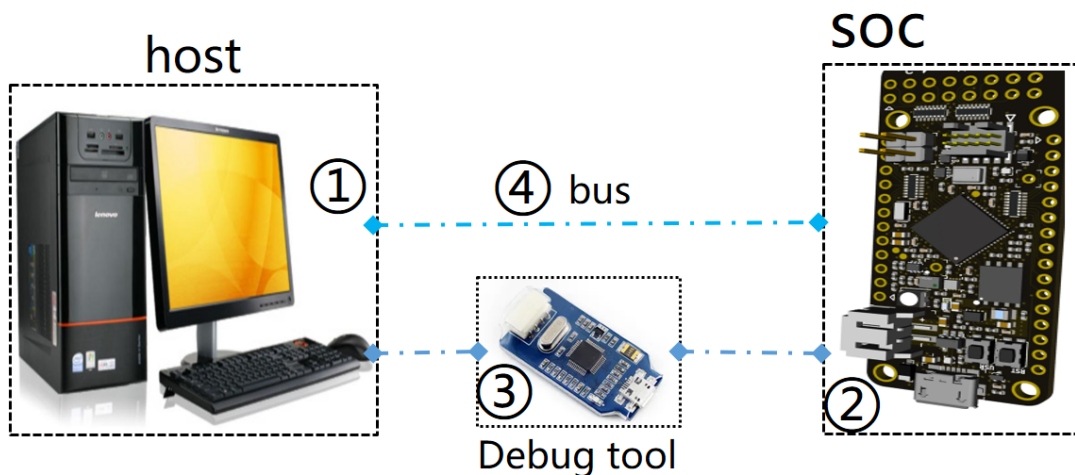


图1 SoC软件开发（调试）流程图

软件开发、调试必须等硬件环境完成之后才能进行，因此存在两个痛点：

- 对于soc的研发，软件开发、调试不能先行，必须等硬件环境完成后才能开始，这样会延长soc软件的研发周期。
- 对于用户，必须具备开发版、调试工具才能使用，增加了成本。

为了解决以上问题，本项目使用软件去模拟SoC各个模块的功能，提供调试服务、虚拟网络服务、虚拟存储设备挂载，以达到在没有硬件环境的条件下能开发、调试对应软件的功能。

0.2 项目简介

本项目基于开源项目QEMU(Quick Emulator)，结合al9000硬件设计进行二次开发。该项目的主页为：<http://confluence.anlogic.com/x/HQOIaG>，主页中包含了一些更加详细的信息。

已支持的虚拟设备

- cpu（双核A53、mmu、gic-v3）
- 内存(DDR、ram、rom)
- npu(hard npu、software npu)
- uart
- 存储设备（sd、flash、虚拟磁盘）
- 网络服务（网卡）
- ...

调试服务：

- gdb server，可以自定义端口

操作系统支持：

- Linux
- Windows（win10, win11）

使用虚拟soc，可以完成以下事情：

- 运行aarch64应用程序
- 运行riscv应用程序（支持中）
- 挂载存储设备，在程序中使用文件系统
- 挂载虚拟网卡，在程序中使用网络服务
- 使用gdb调试应用程序
- ...

0.3 重要约定

为了方便描述，我们约定：

- 下文中的**host**指用户的主机，可以为linux系统（示例中使用的是ubuntu20.04）、windows系统（win10、win11）。
- 下文**虚拟soc**指基于qemu开发的适用于a19000的设备模拟器，它是一个应用程序，模拟apu时其名称为qemu-system-aarch64。

0.4 总结

虚拟soc是使用软件模拟硬件行为的模拟器，支持多种架构、支持多个操作系统。使用虚拟soc，能让大家在没有硬件条件下进行软件开发与调试，缩短研发周期、节约成本、方便调试。

0.5 修订记录

日期	章节	内容
2022/10/1	所有章节	初始版本

1. 虚拟soc的使用

1.1 安装

虚拟soc的安装有两种方式：源码编译、直接使用编译好的可执行文件。

- （1）源码编译

linux系统下：

```
1 # 安装依赖
2 sudo apt install pkg-config libglib2.0-dev libpixman-1-dev bison flex
  libbfd-dev libstdc++-dev re2c
3
4 # 安装ninja
5 git clone git://github.com/ninja-build/ninja.git && cd ninja
6 ./configure.py --bootstrap
7 sudo cp ninja /usr/bin
8
9 # 下载源码
10 git clone git@10.8.10.6:/home/git/alsoc-qemu.git
11 cd alsoc-qemu
12 git submodule init
13 git submodule update --recursive
14
15 # aarch64配置 只安装arm64平台
16 # --enable-virtfs 可以支持9p文件共享，与主机进行直接文件夹共享，但需要手动安装libcap-
  ng-dev and libattr1-dev 依赖
17 # 其中aarch64-softmmu用于启动系统，例如linux，编译后对应的程序是qemu-system-aarch64
```

```

18 # aarch64-linux-user 用于启动程序的运行，即不用启动linux系统也可以模拟程序的运行，例如
    自己写的helloworld应用程序，编译后对应的软件名为qemu-aarch64
19 # --enable-trace-backends=simple,log : 设置trace的后端
20 # --disable-werror : 禁用编译时出现warning而报错，例如定义变量而未使用，定义函数而未使
    用。
21 # --static : 使用静态编译，保证编译后的程序运行不依赖共享库，--disable-xkbcommon --
    disable-libudev --disable-sdl --disable-gtk等必须加上后才能静态编译成功。
22 ./configure --target-list=aarch64-softmmu --enable-virtfs --enable-trace-
    backends=simple,log --disable-werror --static --disable-xkbcommon --disable-
    libudev --disable-sdl --disable-gtk
23 # build, 在build目录下会生成可执行文件: qemu-system-aarch64
24 make -j8

```

验证:

```

1 | ./build/qemu-system-aarch64 --version

```

windows系统下: 参考: <http://confluence.anlogic.com/x/KgOlAg>

- (2) 直接使用编译好的可执行文件

我们会将编译好的可执行文件放在共享目录:

\\192.168.11.15\fileserver\file_share\AI\SoC\soc_qemu, 每次更新之后都会实时同步。该目录包含了win和linux版, 且使用静态编译, 可以直接使用。详细描述见: <http://confluence.anlogic.com/x/KgOlAg>

1.2 启动命令

虚拟soc的启动需要设置很多参数, 建议写到脚本中后运行, 常见脚本如下, 根据自身需求修改参数便可:

```

1 | qemu-system-aarch64 -M a19000 [...]

```

虚拟soc启动参数总结:

```

1 | path/of/qemu -param_name1 param_value1[,p1=v1,p2=v2,...] -param_name2 -
    param_name3 param_value3

```

虚拟soc启动参数有三种类型:

- a、只需指定参数, 例如: -nographic
- b、参数必须指定值, 例如-monitor stdio
- c、参数还需指定其他配置项, 例如: -drive
file=./image/rootfs.ext4,if=none,format=raw,id=hd0

参数规则: 以短横线 (-) 开头的为新的参数, 它前面必须为空格, 参数与参数值之间为空格, 用逗号 (,) 连接所有的配置项, 配置项赋值用等号 (=), 逗号 (,) 和等号 (=) 两边不能有空格。

虚拟soc的参数众多, 但有很多是重复的, 请大家一定按照本教程中的规则使用各种参数。

以下为详细参数及说明。

1.2.0 选择虚拟板卡型号，及常用配置

```
1 # 选择a19000, 支持e12, 支持e13, (e12、e13默认都启用)
2 -M a19000,secure=on,virtualization=on
3 # 将虚拟soc的uart重定向到启动虚拟soc的终端上
4 -nographic
5 # 启动gdb服务, 7777 为gdb server的端口号, 如果多人在同一host上使用, 则每个人的端口号不能相同。
6 # -S表示虚拟soc在启动gdb server后会停住, 等待gdb client的连接。
7 -S -gdb tcp::7777
```

1.2.1 虚拟网卡

```
1 -netdev user,id=eth0,hostfwd=tcp::8001-:22,restrict=off \
2 -device virtio-net-device,netdev=eth0 \
```

此时虚拟soc中添加了虚拟网卡eth0,在虚拟soc中访问网络的能力与host一样,只是ip不一样。从虚拟soc中能通过host的ip访问host,但在**host不能通过虚拟soc的ip访问虚拟soc,必须通过hostfwd参数添加端口转发**,访问host上的转发端口从而访问虚拟soc (ssh登录即使用这种方式)。

hostfwd指定端口转发,其基本设置格式为:

```
# host端的8001转发到虚拟soc内部22端口,实现ssh登录
# host端的8002转发到虚拟soc内部8002端口,实现gdbserver
-netdev user,id=eth0,hostfwd=tcp::8001-:22,hostfwd=tcp::8002-:8002 \
-device virtio-net-device,netdev=eth0 \
```



hostfwd的数量不限,根据实际需求添加便可,例如:

```
1 # host端的8001转发到虚拟soc内部22端口,实现ssh登录
2 # host端的8002转发到虚拟soc内部8002端口,实现gdbserver
3 -netdev user,id=eth0,hostfwd=tcp::8001-:22,hostfwd=tcp::8002-:8002 \
4 -device virtio-net-device,netdev=eth0 \
```

网络配置模型为:

```
1 guest (10.0.2.15) <-----> Firewall/DHCP server <-----> Internet
2 | (10.0.2.2)
3 |
4 ----> DNS server (10.0.2.3)
5 |
6 ----> SMB server (10.0.2.4)
```

1.2.2 挂载虚拟磁盘

有两种形式,与被挂载的文件形式有关系。

(1) 当被挂载虚拟磁盘为一个文件,格式支持: ext2、ext3、ext4、fat32,例如根文件系统(ext4格式)。

```
1 -drive file=rootfs.ext4,index=0,media=disk,format=raw \
```

此时虚拟soc中会增加一个存储设备，设备路径为：/dev/vda，设备名与参数中的index有关，0、1、2、...分别对应vda、vdb、vdc、...。在虚拟soc的linux系统中挂载设备的时候一定要注意该对应关系。

(2) 当被挂载虚拟磁盘为host上的文件夹，此时虚拟soc会将该文件夹虚拟成一个fat文件系统。

```
1 | -drive file=fat:floppy:rw:/host/path/of/share,index=1,media=disk,format=raw
```

此时index与虚拟soc中的设备名的对应关系与（1）中一样。在虚拟soc的linux系统中便可以使用挂载fat系统的命令挂载该设备，在虚拟soc中对于该设备的文件操作都会同步到host的目录：/host/path/of/share。在虚拟soc的linux系统中挂载fat的命令为：

```
1 | # /dev/vdb为设备路径，/virt_soc/为被挂载的路径。
2 | mount -t vfat /dev/vdb /virt_soc/
```

注意事项：

- 只能使用英语路径（ascii编码）
- 当虚拟soc中挂载了该路径后，host上不要在路径下进行文件操作。

1.2.3 挂载sd卡

```
1 | -drive file=sd_img,if=sd,format=raw,index=0
```

在apu-sdk中有使用示例，参照：<http://confluence.anlogic.com/x/Clp8Aw>

1.2.4 加载数据到内存

```
1 | # 加载elf到内存，addr为内存的起始地址，file为加载的文件，会自动识别格式，cpu-num指定cpu
   | 来运行该程序。
2 | -device loader,addr=0x00100000,file=app.elf,cpu-num=0 \
3 | # 加载二进制文件到内存，与elf一样，只是没有cpu-num参数
4 | -device loader,file=data.bin,addr=0x12000000
```

1.2.5 引导linux

虚拟soc具备自动引导linux启动的功能，参数为：

```
1 | # 启动linux镜像，并添加启动参数
2 | -kernel kernel_image_path -append "rootwait root=/dev/vda console=ttyAMA0"
```

该功能需要准备好linux镜像，根文件系统，参考第三章。

1.2.6 串口重定向到tcp端口

```
1 | # 将串口重定向到8005端口
2 | # server: 开启服务
3 | # nowait: 不等待client的连接而运行程序
4 | -serial tcp::8005,server,nowait
```

1.3 控制快捷键

qemu运行后，可以通过快捷键控制它，例如退出等。

所有快捷键都是**先按ctrl + a后松开，再按第三个键**。此时键盘不能处于大写状态。

快捷键	功能	备注
ctrl+a + x	退出qemu	先按ctrl + a后松开，再按x
ctrl+a + c	进入qemu的命令行，可以执行qemu的内置命令。使用info显示所有命令	先按ctrl + a后松开，再按c

通过ctrl+a+c进入qemu命令行后，常用以下命令

命令	功能	备注
q	退出qemu	
info qtree	显示设备树	一般用于查看设备是否被挂载

1.4 地址空间

与al9000的地址空间一样，请参照al9000的地址空间说明。

1.5 总结

根据加载的设备不一样，启动虚拟soc的命令需要搭配各种参数。在使用的过程中，合理搭配各种参数能提升开发效率。

2. 调试baremetal应用

可以使用虚拟soc运行、调试baremetal应用，支持直接启动、使用gdb调试、使用vscode调试、使用eclipse调试。这几种模式在启动命令上没有区别（使用gdb的时候会加上-S -gdb tcp::7777）。

2.1 加载并运行应用程序

常见启动参数：

```
1 ./qemu-system-aarch64 \  
2 -M al9000,secure=on,virtualization=on \  
3 -nographic \  
4 -device loader,addr=0x00100000,file=app.elf,cpu-num=0
```

其中./qemu-system-aarch64为虚拟soc可执行文件的路径，app.elf为要运行的程序，支持elf格式、bin文件，addr为程序的链接地址（程序在sdk中开发时link的地址）。cpu-num为使用哪个cpu，apu有两个cpu，一般指定为0。

使用该命令则在虚拟soc中运行应用程序，如果要加载数据到内存、挂载sd卡，启用gdb调试则加上对应的参数。

使用该功能的前提是编译好了可执行的应用，且知道链接地址。请参考：<http://confluence.analogic.com/x/GpH1Ag>

2.2 启用gdb服务

```
1 | ./qemu-system-aarch64 \  
2 | -M al9000,secure=on,virtualization=on \  
3 | -nographic \  
4 | -device loader,addr=0x00100000,file=app.elf,cpu-num=0 \  
5 | -S -gdb tcp::7777
```

启动后会自动启动gdb 服务，等待gdb的连接。7777为要使用的端口，可以任意填写2000~65536之间的任意数值。

之后另起一个终端，终端运行gdb客户端（这里的gdb必须使用编译被调试程序的交叉编译工具链里的gdb）。

```
1 | # 启用gdb，并加载符号表  
2 | aarch64-linux-gnu-gdb app.elf  
3 | # 以下命令都在gdb命令行里输入  
4 | # 连接gdb 服务端,7777为端口号  
5 | tar remote:7777  
6 | # 设置断点  
7 | b main  
8 | # ... 其他命令
```

常用gdb命令请参考：<http://confluence.anlogic.com/x/TZcbAw>

2.3 使用vscode调试应用

vscode的配置与3.7节一样，虚拟soc的配置与 2.3节一样，这里不在赘述。

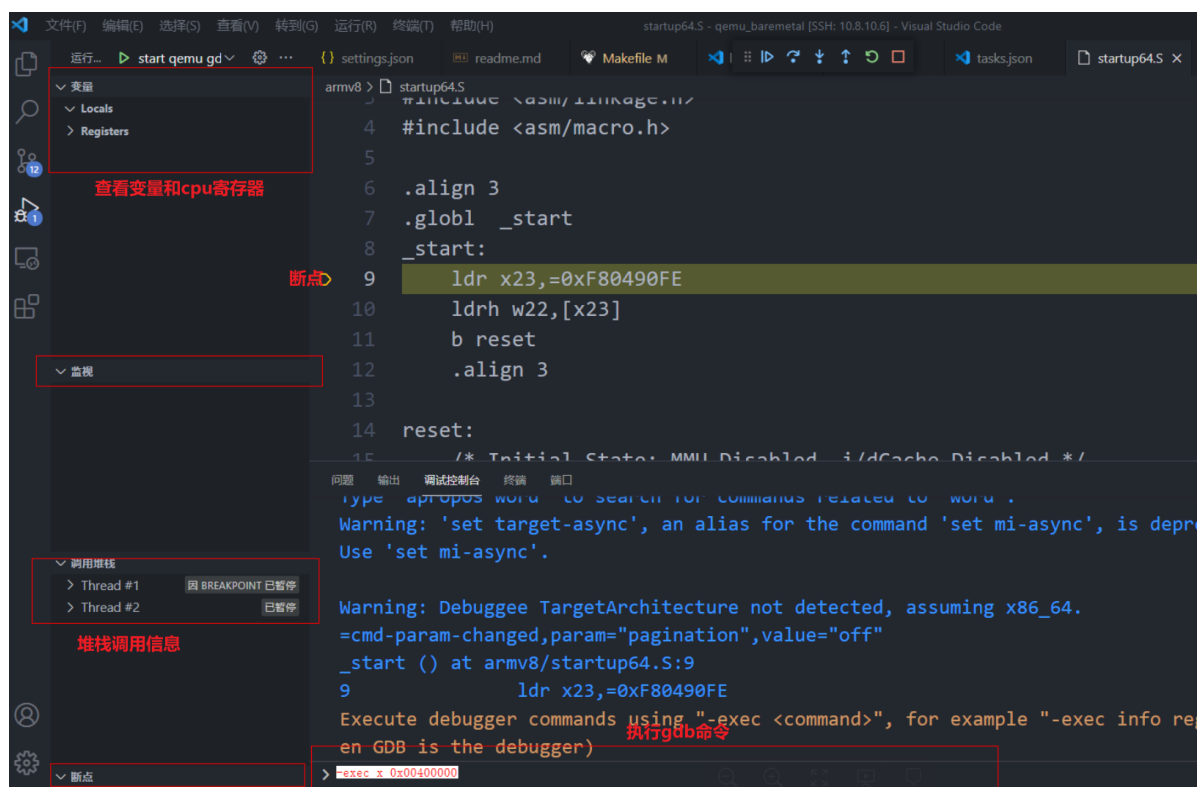
注意事项：

- vscode终端会显示启动qemu的命令，有以下输出的时候代表vscode启动gdb成功：

```
[info]: Link all objects to qemu_gdb.elf  
[info]: Launch ./bin/qemu-system-aarch64  
./bin/qemu-system-aarch64 \  
-M al9000 \  
-nographic \  
-device loader,addr=0x00400000,file=/data/workspace/alsoc-embedded-sw/build/qemu_gdb.elf,cpu-num=0 \  
-S -gdb tcp::7777 \  
-drive file=./apps/sd_test/sdcard_fs.fat32.img,if=sd,format=raw,index=0 \  

```

- 在vscode 中按 “f5”， 如果提示 “无法追踪 run qemu”，点击anyway。
- 在终端中会显示启动虚拟soc命令的参数，gdb连上qemu的gdb server后在调试控制端可以使用gdb 命令，需要注意的是需要在gdb命令前加上 -exec， 例如： -exec x 0x00400000



2.5 总结

使用虚拟soc运行或调试baremetal应用是其基本功能，基本思想是使用loader将应用代码加载到内存区域，并用指定cpu从指定地址开始运行。结合gdb服务、vscode等IDE能让调试过程更加方便快捷。

3. 调试linux

可以使用虚拟soc开发或调试linux相关的程序，包括：内核、驱动、应用。对于驱动而言，只能使用虚拟soc已经包含的模块。

使用该功能必备条件：

- 编译了虚拟soc的运行程序（3.1中的qemu-path）

- 编译了linux镜像（3.1中的kernel_image_path）
- 准备好了rootfs文件（3.1中的rootfs_path）
- 编译了设备树文件（3.1中的device_tree.dtb）

以上文件在服务器上都有提供（源码+编译过后的文件），可以直接拉取后使用。

3.1 引导linux启动

```
1  #!/bin/bash
2
3  qemu_path="/data/share/anlogic-qemu/build/qemu-system-aarch64"
4  linux_img="Image"
5  rootfs_path="rootfs.ext4"
6  hd1_img="./virt_soc"
7
8  ${qemu_path} \
9      -M a19000,secure=off,virtualization=off \
10     -kernel ${linux_img} \
11     -append "rootwait root=/dev/vda console=ttyAMA0" \
12     -dtb a19000_t.dtb \
13     -nographic \
14     -netdev user,id=eth0,hostfwd=tcp::8001-:22,restrict=off \
15     -device virtio-net-device,netdev=eth0 \
16     -drive file=${rootfs_path},index=0,media=disk,format=raw \
17     -drive file=fat:floppy:rw:${hd1_img},index=1,media=disk,format=raw
```

脚本解析:

行号	示意	备注
3	指定虚拟soc可执行文件路径，在第8行被使用，必须参数。	
4	指定linux镜像的路径，在第10行被使用。必须参数。	
5	根文件系统的路径，在第16行被使用。必须参数	
6	用于共享的host的文件夹路径，在第17行被使用。非必须，可以将该行与第17行同时删除。	
9	指定虚拟soc名称，关闭el2和el3。使用虚拟soc内部的loader引导linux的前提条件是关闭el2、el3	
11	传递给linux内核的启动参数，root为挂载根文件系统的路径，与第16行中的index要对应。	
12	指定设备树	
13	命令行模式	
14-15	虚拟网卡eth0，将host端口8001转发到虚拟soc的22，用于ssh登录	
16	rootfs的路径，支持ext4等格式。linux中设备为：/dev/vda	
17	将host上的路径虚拟为fat文件系统挂载到虚拟soc中，linux中设备为：/dev/vdb	

温馨提醒：如果启动了linux，最好使用命令 'poweroff' 关闭虚拟soc（而不是使用ctrl+A + x来直接退出虚拟soc），以保护挂载的文件不受到损坏。

3.2 ssh连接&scp传输文件

在虚拟soc启动之后，可以在host上远程连接，或者用scp进行文件传输。

使用该功能的前提条件：

- 网卡中进行了端口转发，见3.1中的14-15行命令。
- **虚拟soc中运行的linux系统的根文件系统中启用了ssh服务。**
- 虚拟soc中启用了ssh服务。查看 /etc/ssh/sshd_config,修改下面三项后重启：

```
1 Port 22
2 AddressFamily any
3 PermitRootLogin yes
```

在host上使用ssh登录以及传输文件：

```
1 ssh root@localhost -p 8001
2 # scp命令中指定端口使用大写P
3 # 复制当前目录下的file1.txt到虚拟中的/目录下
4 scp -P 8001 file1.txt root@localhost:/
```

3.3 tftp文件传输

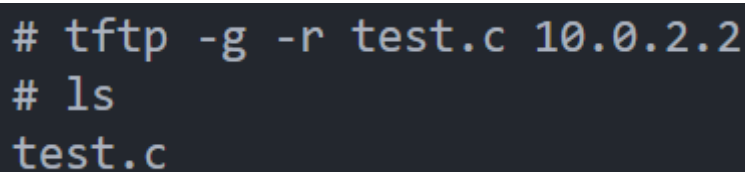
在虚拟soc中可以使用tftp与host传输文件。

该功能的使用前提是在启动虚拟网卡的参数中加上tftp=/path/of/host/tftp/dir（3.1节中启动参数的第14行）。加上参数后，虚拟soc会启动内建的tftp服务，ip地址为10.0.2.2，映射到host的文件夹路径为：/path/of/host/tftp/dir。

```
1 -netdev
   user,id=eth0,hostfwd=tcp::8001-:22,restrict=off,tftp=/data/share/tftp \
```

在虚拟soc中则使用tftp命令：

```
1 Usage: tftp [OPTIONS] HOST [PORT]
2
3 Transfer a file from/to tftp server
4     -l FILE Local FILE
5     -r FILE Remote FILE
6     -g      Get file
7     -p      Put file
8     -b SIZE Transfer blocks in bytes
```



```
# tftp -g -r test.c 10.0.2.2
# ls
test.c
```

其中10.0.2.2为虚拟soc内建的tftp服务的地址。如果在host上建立了tftp服务，则将此ip改为host的ip后也能实现与host进行文件传输的功能，此时无需tftp=/share/dir的参数。

3.4 nfs文件共享

在虚拟soc中可以使用挂载nfs实现与host进行文件共享。

使用该功能的前提：

- host上搭建了nfs服务

```
1  sudo apt-get install nfs-kernel-server # 安装 NFS服务器端
2  sudo apt-get install nfs-common      # 安装 NFS客户端
3  # 配置nfs服务
4  sudo vim /etc/exports 填入以下内容
5
6  /data/share/nfs *(insecure,rw,sync,no_root_squash)
7
8  # 其中/data/share/nfs 为host上的nfs服务的根目录
9  # insecure是必须的，否则会报权限错误
10 # *表示任何ip都可以登录
11
12 # 重启nfs服务
13 sudo /etc/init.d/nfs-kernel-server restart
```

可以在主机上测试nfs是否成功。以下为测试代码，将路径改为自己的实际路径。

```
1  # /data/share/nfs为host上的nfs服务根目录
2  # /data/share/nfs_test 挂载目录
3  sudo mount -t nfs localhost:/data/share/nfs /data/share/nfs_test -o nolock
```

在虚拟soc中，使用mount挂载nfs

```
1  # /data/share/nfs为host上的nfs服务根目录
2  # /nfs 为虚拟soc上的挂载目录
3
4  mkdir /nfs
5  mount -t nfs 10.8.10.52:/data/share/nfs /nfs -o nolock
```

如果没有报错，便实现了文件共享。

3.5 访问host文件夹

在虚拟soc中可以使用挂载虚拟磁盘的方式来与访问并操作host上某文件夹的文件。注意这种方式不能实时共享文件，在虚拟soc中修改文件会实时同步到host上，**禁止当该文件夹被虚拟soc挂载时在host上操作该文件夹以及文件夹中的任意文件。**

虚拟soc的启动参数见3.1的第17行，**参数中index与linux系统的磁盘设备名对应关系一定要正确**（见1.2.2）。



启动虚拟soc，进入linux系统，后挂载该磁盘。

```
1 | mkdir /virt_soc
2 | mount -t vfat /dev/vdb /virt_soc/
```

这种方式注意事项见：1.2.2

3.6 gdb调试linux内核

调试linux内核，需要在linux内核编译时设置以下几个选项：

```
1 | # 开启kernel debug info
2 | kernel hacking --->
3 |     [*] kernel debugging
4 |     Compile-time checks and compiler options --->
5 |         [*] Compile the kernel with debug info
6 |         [*] Provide GDB scripts for kernel debuggin
7 | # 关闭地址随机化
8 | Processor type and features ---->
9 |     [] Randomize the address of the kernel image (KASLR)
10 |
```

qemu启动脚本相对于3.1的脚本多了一行：

```
1 | # 7777 为gdb server的端口号，如果多人在同一host上使用，则每个人的端口号不能相同。
2 | -s -gdb tcp::7777
```

加上上述参数后，虚拟soc在启动后会等待gdb 客户端的连接。

在host上启动gdb客户端后连接虚拟soc提供的gdb服务（终端不会有输出），命令如下：


```

25         "ignoreFailures": true
26     }
27 ],
28
29     }
30 ]
31 }

```

- 配置.vscode/tasks.json

```

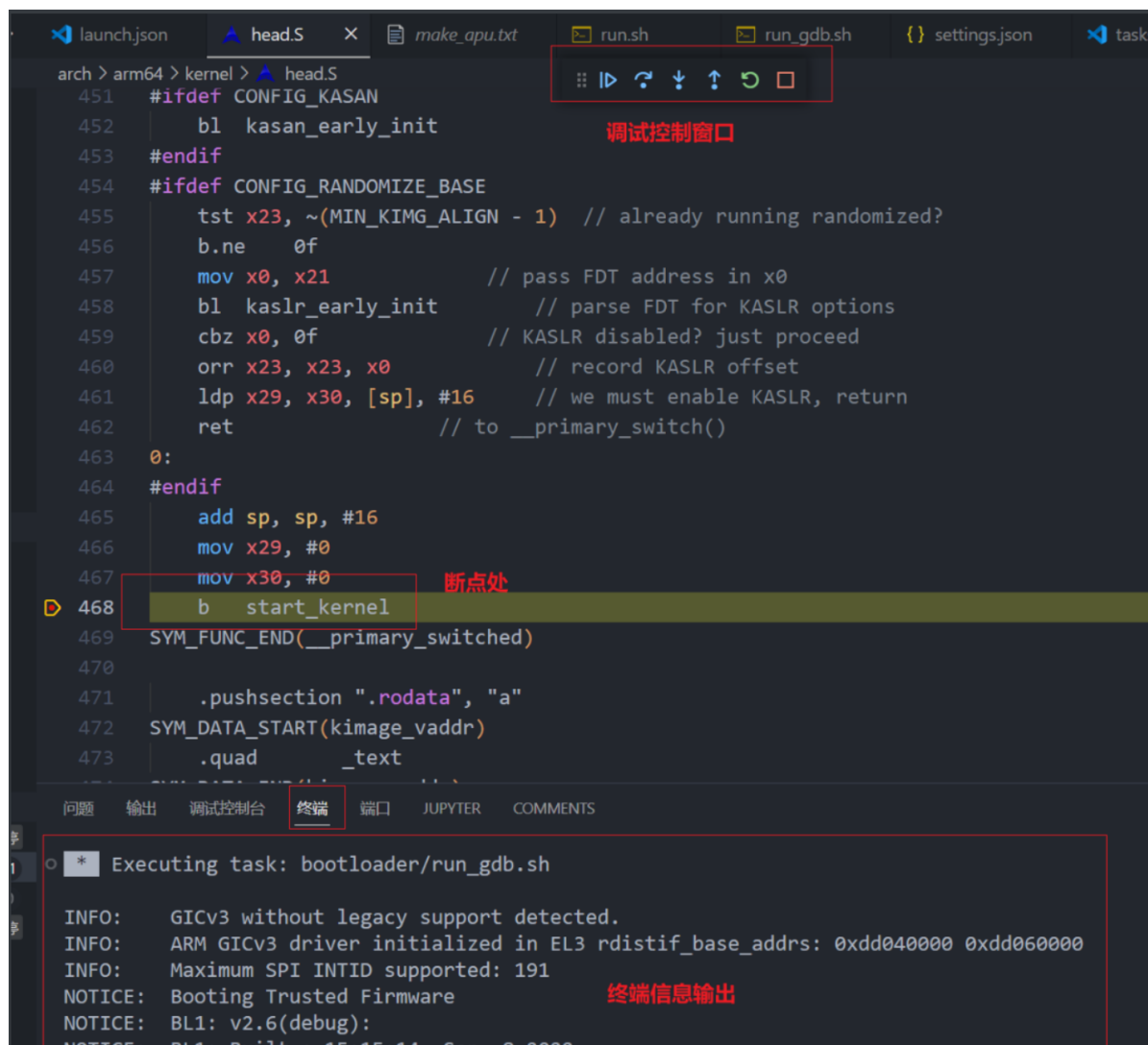
1  {
2      "version": "2.0.0",
3      "tasks": [
4          {
5              // start a qemu with gdb server for vscode debug
6              "label": "run qemu",
7              "type": "shell",
8              // start virt soc script
9              "command": "/path/of/qemu_start.sh",
10             "isBackground": true,
11         },
12         {
13             // forcibly stop qemu which opened by preLaunchTask after
14             debugger.
15             "label": "stop qemu",
16             "type": "shell",
17             // gdb server port
18             "command": "kill -9 `lsof -i:7777 | awk 'NR==2{print $2}'` ",
19             "isBackground": true,
20         }
21     ]
22 }

```

以上两个文件中的gdb的端口号要与qemu启动脚本中的端口号一样。

之后按F5后便可以进入调试模式，等待qemu启动。

在arch/arm64/kernel/head.S文件的start_kernel（466行）打个断点后，按f5，便可以运行到此处，并停在断点处。



3.8 总结

文件共享（传输）的四种方法对比

方法	优点	缺点
scp	没有其他依赖	不能实时操作，需要在host上手敲命令
tftp	内建tftp服务	不能实时共享，需要在虚拟soc里手敲tftp命令
nfs	能实时共享	需要在host端搭建nfs服务
虚拟fat磁盘挂载	虚拟soc操作文件能实时更新到host端	虚拟soc启动后在host最好不要操作共享文件夹