目录

ICE EVB 开发板简介

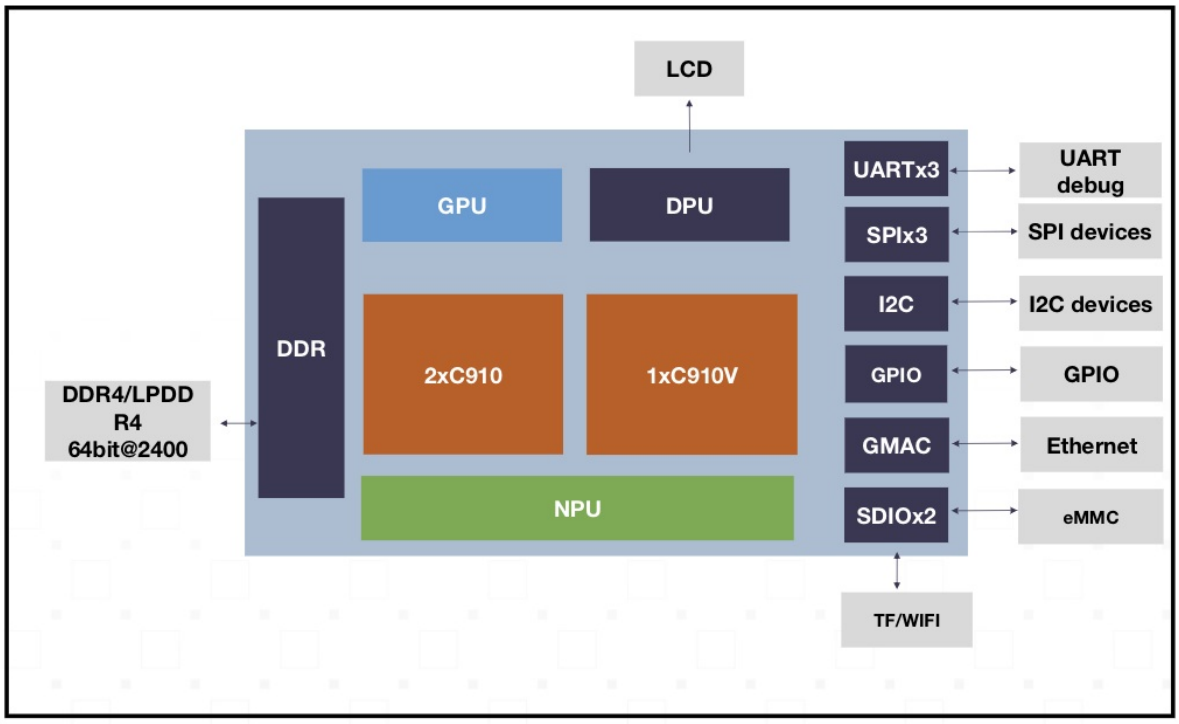
ICE EVB 是基于 T-Head 自主研发的 C910 CPU 的 ICE 高性能 SoC 开发板，以丰富的功能与外设，适合多种业务场景。

- 丰富的常见外设及接口
- 基于 OpenGL ES 的 3D 图形处理能力
- 先进的视觉AI处理器
- 音视频多媒体处理器

ICE 芯片

ICE 是一款通用智能数字 SoC 芯片，可以为每台设备提供高性价比的智算能力，赋予每台设备 AI 功能。内部集成 1个 3核 C910，1个 NN 核和 1个 GPU，可根据实际应用需要灵活配置运算能力，单片最高能提供 0.5TOPS@INT8 的运算能力。提供了 4K@60 帧的视频实时解码能力，支持 HEVC，AVC 和 JPEG 解码，支持 JPEG 编码以及1路 I2S 接口输入。同时提供了丰富的高速接口以及通用外设接口，用于与主控设备间的数据和命令交互。

- 内嵌平头哥双核 C910@1.2GHz
- 内嵌平头哥 C910V@1.2GHz
- 支持 vector 指令，位宽可达 128bit
- 支持 DDR4 up to 2400Mbps
- 支持 GMAC 接口
- 支持 GPU，支持 3D
- 支持 RGB888 LED 显示，最大 1080P
- 算力 0.5TOPS@INT8 数据类型
- 芯片尺寸：15x15mm
- 工艺： 28HPC+


LCD
GPU
DPU
DDR
2xC910
1xC910V
NPU
DDR4/LPDD
R4
64bit@2400
UARTx3
UART
debug
SPIx3
SPI devices
I2C
I2C devices
GPIO
GPIO
GMAC
Ethernet
SDIOx2
eMMC
TF/WIFI

EVB开发板

SDK简介

提供以 Linux 为操作系统，集成了 SoC 与板载外设的所有驱动并提供相应的测试用例，使开发者能够快速上手、快速开发、快速形成自己的产品。下表为 SDK 支持的重要模块的功能与参数：

模块	规格
BootROM	在烧写工具中，可对裸板进行 Flash 烧写 U-Boot 等启动镜像
U-Boot	可在 U-Boot 中烧写各类镜像，并启动 Linux 系统
Linux 系统	RISC-V 64GC 指令架构的 CPU，运行频率达 1.2GHz，内核版本为 5.4.36，提供多种内核调试方法
GPU	2 vec、4 shader cores 提供基于 OpenGL ES 2.0 的用户接口与测试用例
DPU	并行 pixel 输出，24-bit Data，Hsync，Vsync，DE (*Data Enable*) 支持 1080P 分辨率提供基于 fb 接口的用户接口与测试用例
GMAC	支持 RGMII/RMII 接口，支持 10/100/1000Mbps 速率
SD/eMMC	提供 SD 卡3.0 版本卡槽与读写能力提供 4.5 版本板载 4GB eMMC 颗粒
低速外设	3个 UART，1个 I2C，若干 GPIO

概述

本文介绍镜像烧写、运行预置程序、开发环境搭建、DEMO 程序、内核调试方法等多个方面，让用户快速上手。

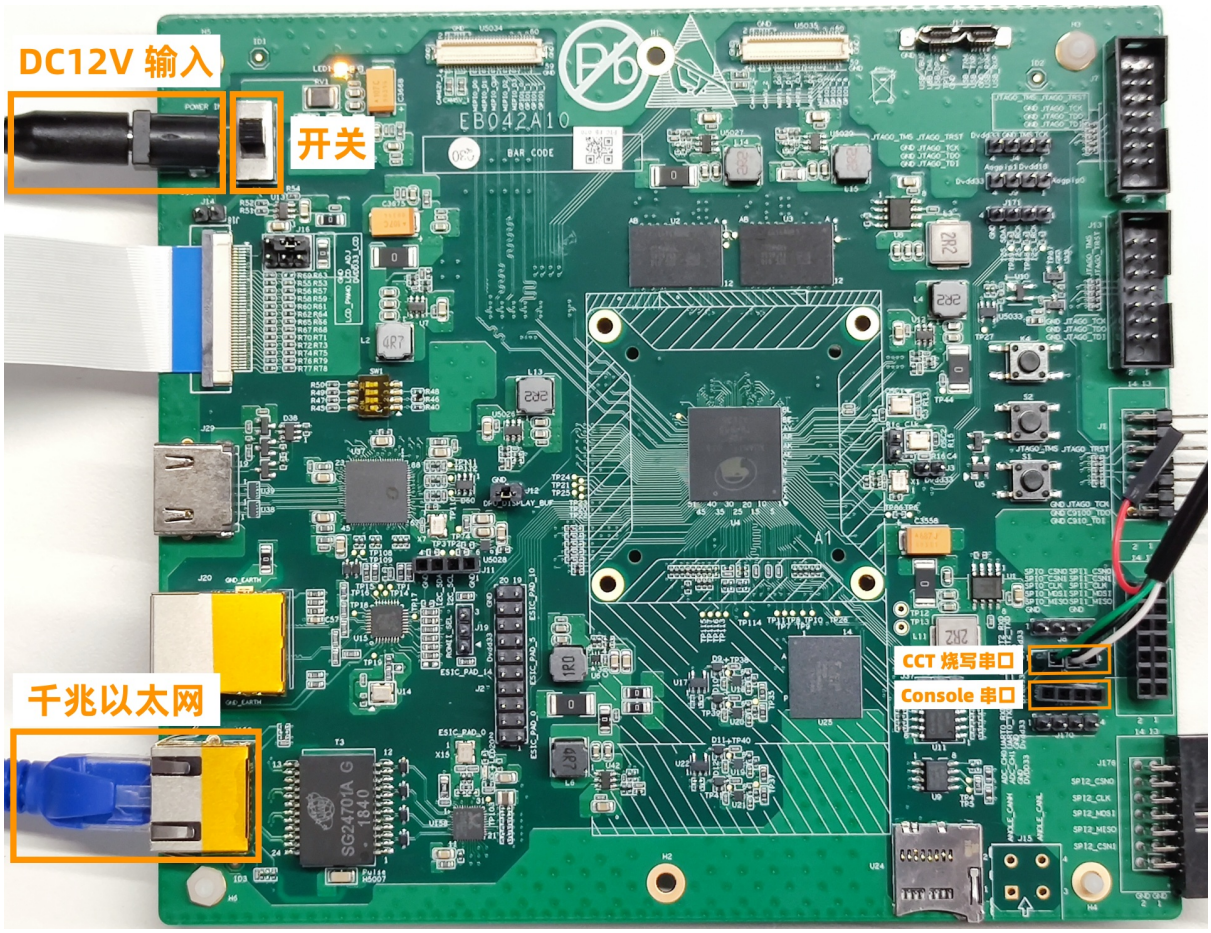
开发板接线

硬件连接

先按照系统连接图连接好开发板

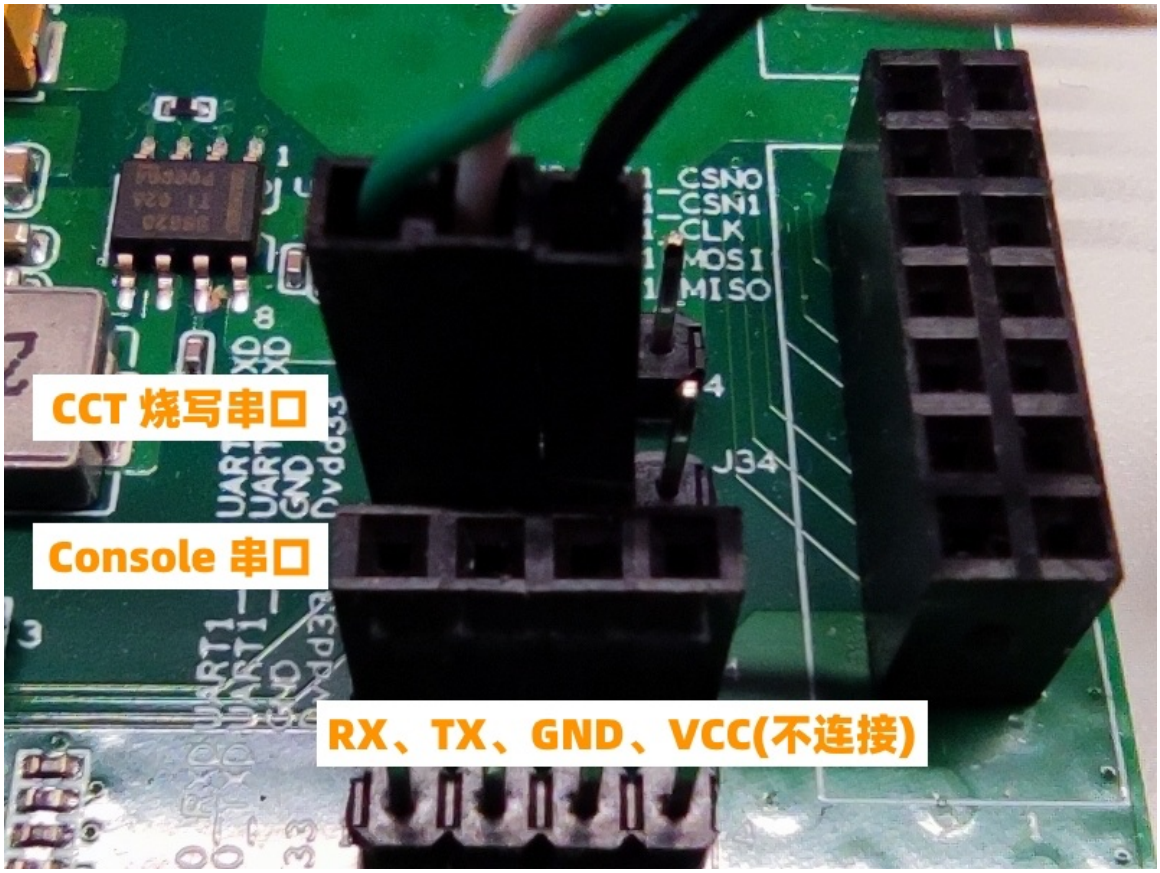
- 设备电源："DC12V 输入"，连接 12V 直流电源
- CCT 烧写串口："CCT 烧写串口"，J5 1-3 PIN，分别为RX、TX、GND，使用 thead-tools 或 CCT 工具烧写 uboot 镜像时使用的通信端口。
- Console 控制串口："Console 串口"，J169 1-3 PIN，分别为RX、TX、GND，log 输出，控制台时使用的通信端口。
- 网络接口："千兆以太网，连接 RJ45 千兆网线，10M/100M/1000M 自适应。

CCT 烧写系统连接图：



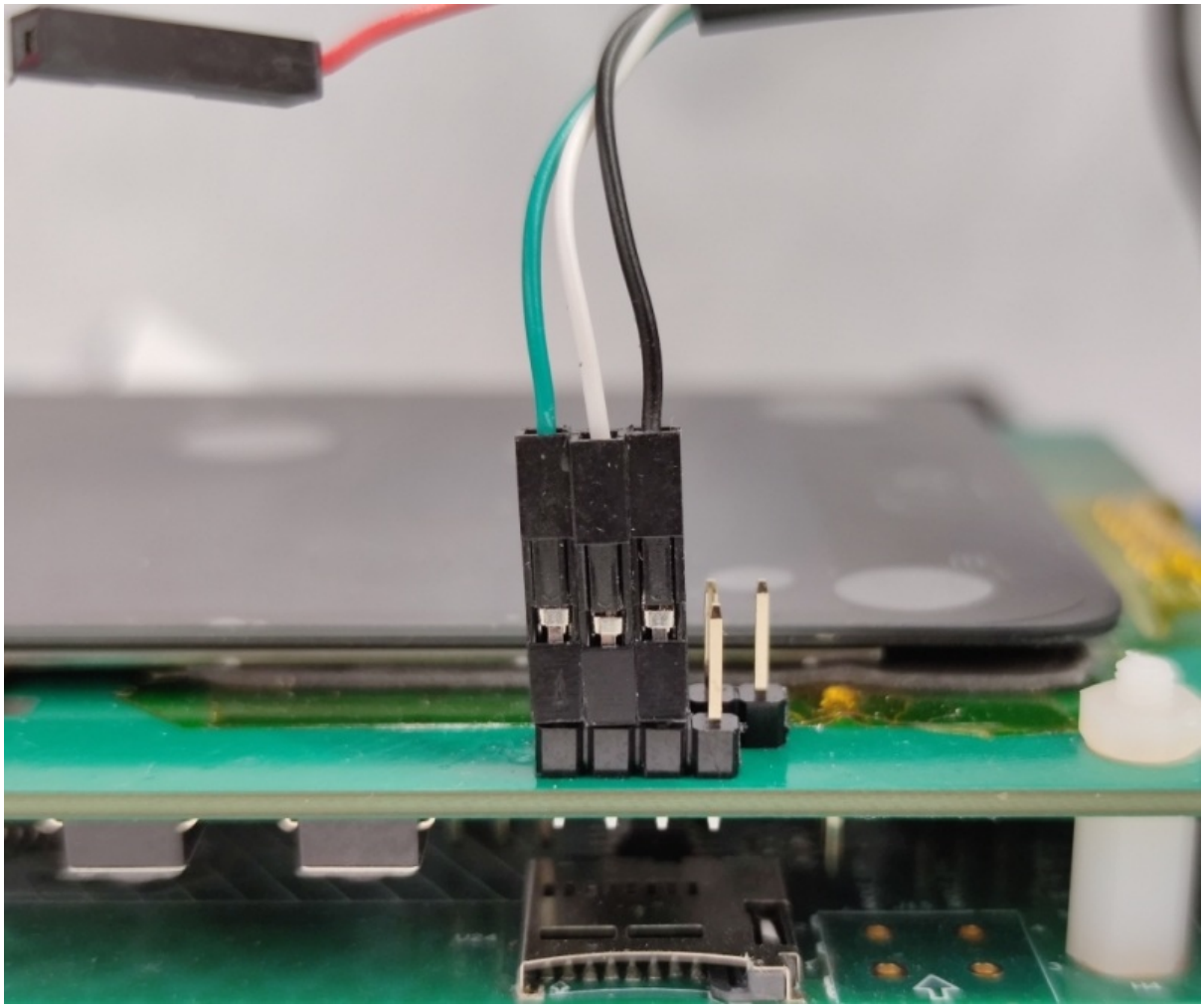
尤其注意串口的连接：

- 需要连2个串口，一个用来打印，一个用来烧录bootimg
- 连接的方式是绿，白，黑


CCT 烧写串口
Console 串口
RX、TX、GND、VCC(不连接)
CSN0
CSN1
CLK
MOSI
MISO
J34

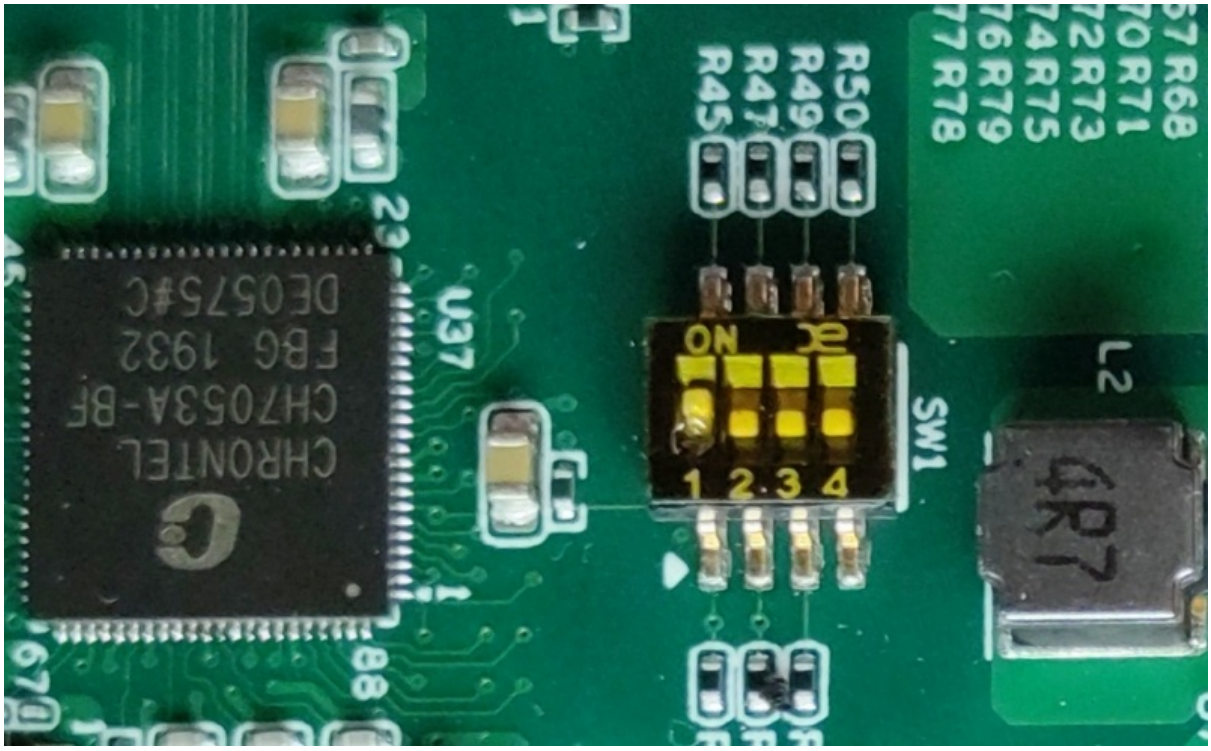
系统连接图：

Console 串口接线：



BOOT SEL 拨码：

BOOT SEL 拨码置为ON、OFF、OFF、OFF，eMMC 启动。


R50
R49
R47
R45
ON
1 2 3 4
SW1
L2
4R7
U37
CHRONTEL
CH7053A-BF
FBG 1932
DE0575#C

Linux 环境下烧写镜像

下载镜像

```
git clone https://gitee.com/thead-linux/ice_images.git
cd ice_images
ls -l
```

需要用到的，是如下三个文件：

```
-rw-r--r-- 1 root root  10485760 Dec 20 18:19 boot.ext4
-rw-r--r-- 1 root root 155189248 Dec 20 14:58 debian-rootfs.ext4
-rw-r--r-- 1 root root    599192 Dec 20 16:39 u-boot-with-spl.bin
```

- u-boot-with-spl.bin： uboot 引导镜像文件
- boot.ext4： /boot 分区文件系统，ext4格式，包含 fw_jump.bin、uImage、hw.dtb
- debian-rootfs.ext4： Debian 基础根文件系统镜像，ext4 格式

Thead-tools 工具

安装 thead-tools

通过 pip 命令来安装 yoctools 到你的系统中，yoctools 支持 python2.7+、python3.6+，建议使用 python2 作为默认 python。

```
sudo pip install thead-tools

# 如果使用官方地址下载过慢，可使用国内清华镜像源加速
sudo pip install -i https://pypi.tuna.tsinghua.edu.cn/simple thead-tools
```

如果未找到 `pip` 命令，请先安装 pythonp-pip，如： `sudo apt install python-pip` 。

thead-tools 使用

通过 thead-tools 命令可以烧写 uboot，更详细的 thead-tools 使用说明可以参见：thead-tools 使用说明，通过命令: `thead cct --help` 查使用说明：

```
root@linux > thead cct --help
Usage: thead cct <uart|list|download> [param]

Options:
  -h, --help            show this help message and exit
  -u UART               CCT serial port device
```

```
  -f FILE
  -o OFFSET             Device start address
  -b BLOCK, --block=BLOCK
  -d DEVICE, --device=DEVICE
                        Device name
  -c, --compress
  -D, --debug           Enable debbug trace info
```

烧写 uboot

查看电脑串口

参考 开发板接线 完成开发板的串口连接，通过 CCT 烧写 uboot 需要使用开发板的 CCT 烧写串口，通过命令 `thead cct uart` 命令看到电脑上安装的串口列表，并确认电脑上串口与开发板串的对应关系：

```
root@linux > thead cct uart
uart device list:
   /dev/ttyUSB0 - USB-Serial Controller
   /dev/ttyUSB1 - USB-Serial Controller
```

查找开发板存储器列表

通过命令：`thead cct -u /dev/ttyUSB0 list` 可以查看开发板支持的烧写存储器列表，如下：

```
root@linux > thead cct -u /dev/ttyUSB0 list
Wait .....................
CCT Version: 2
memory device list:
  dev = ram0   , size =  256.0KB
  dev = emmc0  , size =    2.0MB
  dev = emmc1  , size =    2.0MB
  dev = emmc2  , size =    3.7GB
```

注意：使用该命令时，先确定串口的连接是否正确，运行命令前先关闭开发板电源后再运行该命令，等到出现 `Wait ..........` 信息后再开启开发板电源，正常情况下电源开启后在1秒内会出现如下信息：

```
CCT Version: 2
memory device list:
  dev = ram0   , size =  256.0KB
  dev = emmc0  , size =    2.0MB
  dev = emmc1  , size =    2.0MB
  dev = emmc2  , size =    3.7GB
```

如果未出现上述打印信息，同时电脑是接有多个串口，请更新另一个串再尝试一次，如 `thead cct -u /dev/ttyUSB1 list` 。如果所有串口都尝试后均未出现上述打印机信息，请检查串口与开发板的连接是否符合，详细连接参见：开发板接线。

烧写 uboot

通过命令： `thead cct -u /dev/ttyUSB0 download -f u-boot-with-spl.bin -d emmc0` 将 uboot 烧写到 eMMC 的 0号分区，烧写过程信息如下：

```
CCT Version: 2
Send file 'u-boot-with-spl.bin' to 21:0 ...
Writing at 0x00009800... (3%)
```

待打印 `File u-boot-with-spl.bin download success.` 后，烧写成功后，可以通过开发板的 Console 串口看到正确的启动信息：

```
U-Boot 2020.01-g6cc5d59b0d (Dec 20 2020 - 08:37:37 +0000)

CPU:   rv64imafdcvsu
Model: T-HEAD c910 ice
DRAM:  4 GiB
GPU ChipDate is:0x20151217
GPU Frequency is:500000KHz
NPU ChipDate is:0x20190514
DPU ChipDate is:0x20161213
MMC:   mmc0@3fffb0000: 0
Loading Environment from MMC... OK
In:    serial@3fff73000
Out:   serial@3fff73000
Err:   serial@3fff73000
Net:
Warning: ethernet@3fffc0000 (eth0) using random MAC address - e6:e2:ea:7a:30:ce
eth0: ethernet@3fffc0000
Hit any key to stop autoboot:  1
```

烧写 Linux

安装 fastboot

发行版可通过包管理器安装 fastboot 工具，如 ubuntu下通过 `apt install fastboot` 安装或从官网下载二进制文件，下载地址：https://developer.android.com/studio/releases/platform-tools。

开发板uboot配置

开发板重新开机后进入 uboot 命令模式，当串口出现如下提示时，按任键即可进行命令模式。

```
Warning: ethernet@3fffc0000 (eth0) using random MAC address - a6:7d:bc:02:7d:4d
eth0: ethernet@3fffc0000
Hit any key to stop autoboot:  3
```

通过 `gpt` 命令配置 eMMC 分区大小，命令如下：

```
# 配置 eMMC 分区
setenv uuid_rootfs "80a5a8e9-c744-491a-93c1-4f4194fd690b"
setenv partitions "name=table,size=2031KB;name=boot,size=60MiB,type=bc13c2ff-59e6-4
262-a352-b275fd6f7172;name=root,uuid=$uuid_rootfs,size=-,type=linux"
gpt write mmc 0 $partitions

# 配置内核启动参数
setenv bootargs "console=ttyS0,115200 rdinit=/sbin/init rootwait rw earlyprintk roo
t=PARTUUID=$uuid_rootfs rootfstype=ext4 clk_ignore_unused loglevel=7 crashkernel=25
6M-:128M c910_mmu_v1"
setenv bootcmd "ext4load mmc 0:2 $opensbi_addr fw_jump.bin; ext4load mmc 0:2 $dtb_a
ddr hw.dtb; ext4load mmc 0:2 $kernel_addr uImage; bootm $kernel_addr - $dtb_addr"

# 配置网络
setenv ipaddr 192.168.1.100 # 要求 ipaddr 与主机的IP 在相同的网段并且与网络中其他IP不能冲突
setenv netmask 255.255.255.0
ping 192.168.1.1            # ping 主机的IP，检验网络是否正常
saveenv                     # 将配置保存到 eMMC,下次启动时无需再配置

# 开启fastboot模式
fastboot udp
```

uboot 模式下，可通过下列命令确认环境变量和分区信息：

```
# 打印环境变量
printenv

# 确认分区信息
mmc part

Partition Map for MMC device 0  --   Partition Type: EFI

Part    Start LBA       End LBA         Name
        Attributes
        Type GUID
        Partition GUID
  1     0x00000022      0x00000fff      "table"
        attrs:  0x0000000000000000
        type:   ebd0a0a2-b9e5-4433-87c0-68b6b72699c7
        type:   data
        guid:   95ad0d82-0adf-4dc0-94f8-280c65496c77
  2     0x00001000      0x0001efff      "boot"
```

```
        attrs:  0x0000000000000000
        type:   bc13c2ff-59e6-4262-a352-b275fd6f7172
        guid:   ee463c5a-5e1c-4c8e-9584-aca706c7e1d3
  3     0x0001f000      0x0075ffde      "root"
        attrs:  0x0000000000000000
        type:   0fc63daf-8483-4772-8e79-3d69d8477de4
        type:   linux
        guid:   80a5a8e9-c744-491a-93c1-4f4194fd690b
```

烧写镜像

在电脑上使用使用 fastboot 命令完成镜像烧写，192.168.1.100 是开发板的运行 u-boot 时配置的 IP 地址，使用 fastboot 命令前确认电脑与开发板在同一个子网。可在板子上使用 `ping 192.168.1.1` 检查网络情况。

```
# PC 烧写镜像
fastboot -s udp:192.168.1.100 -S 5M flash boot boot.ext4
fastboot -s udp:192.168.1.100 -S 5M flash root debian-rootfs.ext4
```

启动 Linux

完成 Debian Linux 烧写后，重新启动 ICE EVB 开发板，开发板 console 显示如下：

```
[  OK  ] Started Serial Getty on hvc0.
[  OK  ] Started Serial Getty on ttyS0.
[  OK  ] Reached target Login Prompts.
[  OK  ] Started System Logging Service.
[  OK  ] Finished Remove Stale Onli…ext4 Metadata Check Snapshots.
[  OK  ] Reached target Multi-User System.
[  OK  ] Reached target Graphical Interface.
         Starting Update UTMP about System Runlevel Changes...
[  OK  ] Finished Update UTMP about System Runlevel Changes.

Debian GNU/Linux bullseye/sid thead-910 ttyS0

thead-910 login: [   94.362252] random: crng init done
[   94.365674] random: 6 urandom warning(s) missed due to ratelimiting

thead-910 login:
```

登录 username: root，无密码，开发者可根据需要更改密码。

调整分区大小

debian-rootfs.ext4 烧写到 eMMC 后，需要通过 `resize2fs` 命令调整分区大小，命令如下：

```
# 调整根文件系统分区
resize2fs /dev/mmcblk0p3
# 调用 boot 分区
resize2fs /dev/mmcblk0p2
```

调整成功后，显示如下：

```
Filesystem at /dev/mmcblk0p3 is mounted on /; on-line resizing required
old_desc_blocks = 2, new_desc_blocks = 30
[  209.862639] EXT4-fs (mmcblk0p3): resized filesystem to 3803136
The filesystem on /dev/mmcblk0p3 is now 3803136 (1k) blocks long.
```

Windows 环境下烧写镜像

烧写工具 imgwriter

安装烧写工具

下载 imagewriter 并安装到 Windows。

```
git clone https://gitee.com/thead-linux/ice_images.git
cd ice_images
ls -l
```

需要用到的，是如下两个文件：

```
CCT_V0.95.zip        ice_ck910_imgwriter_asci_v5.bin
```

连接镜像传输串口

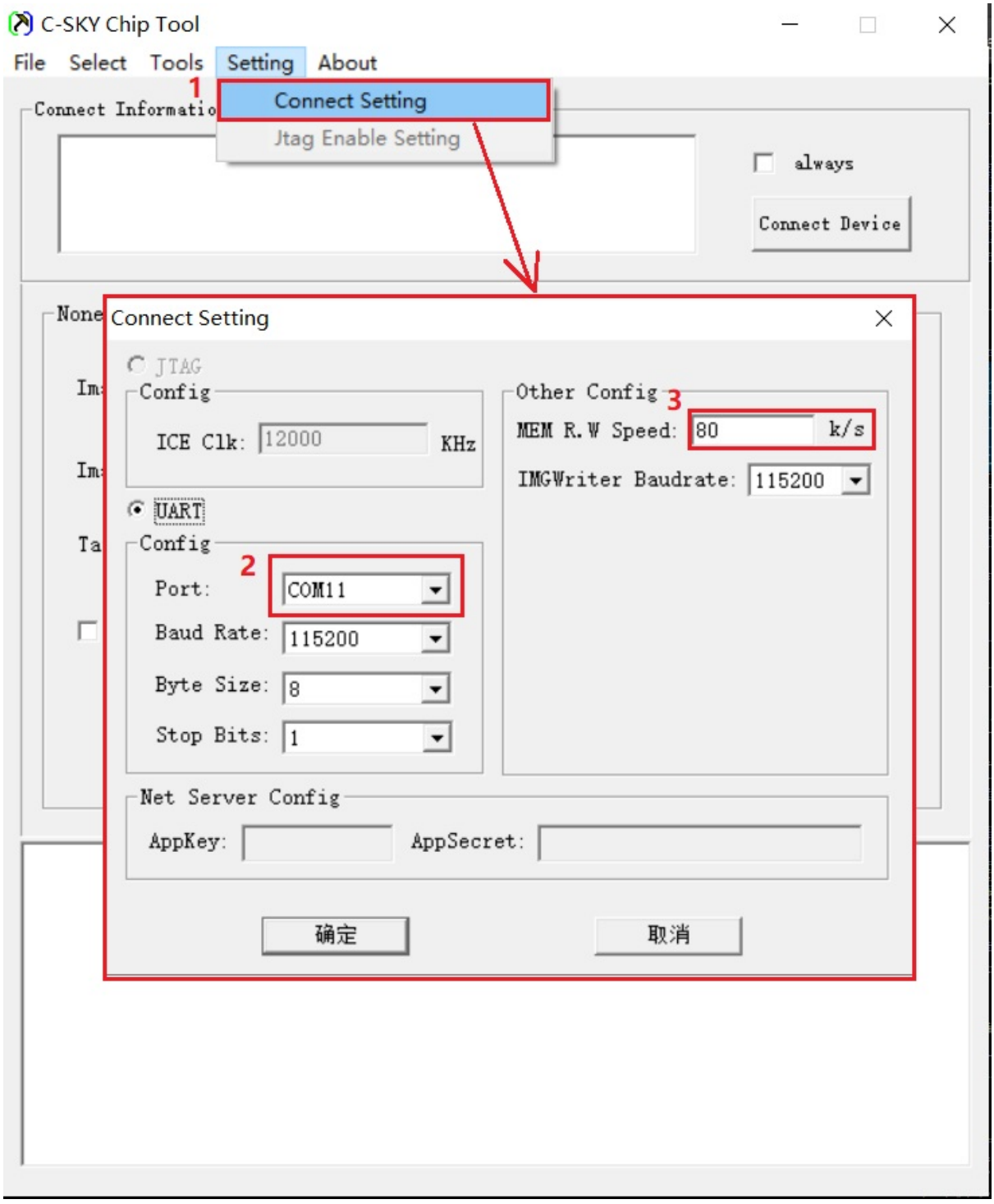
除了 Console 串口之外，CCT 需要用一个镜像传输专用串口。

连接配置

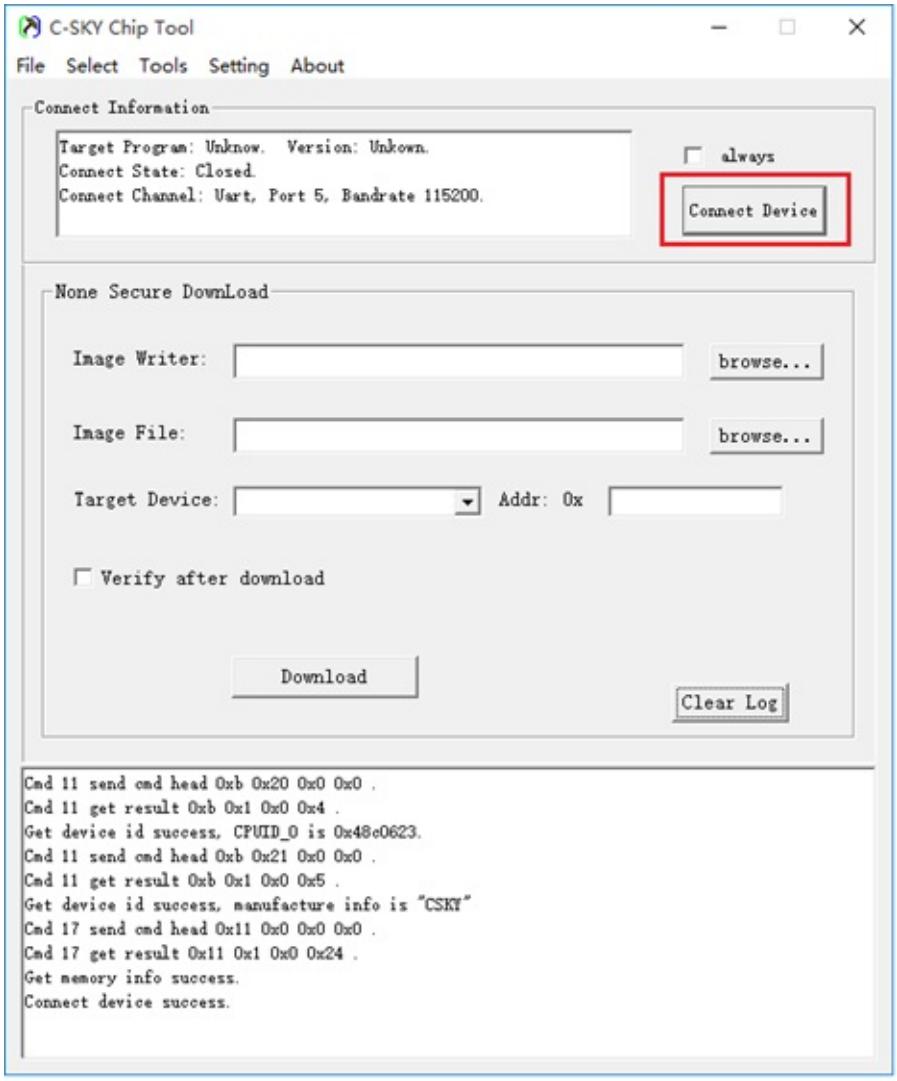
CCT 软件运行界面如下图所示，首先配置 CCT 的参数，其中：

• Port 配置的是 CCT 镜像传输串口

• Mem R.W Speed 是镜像传输协议速度

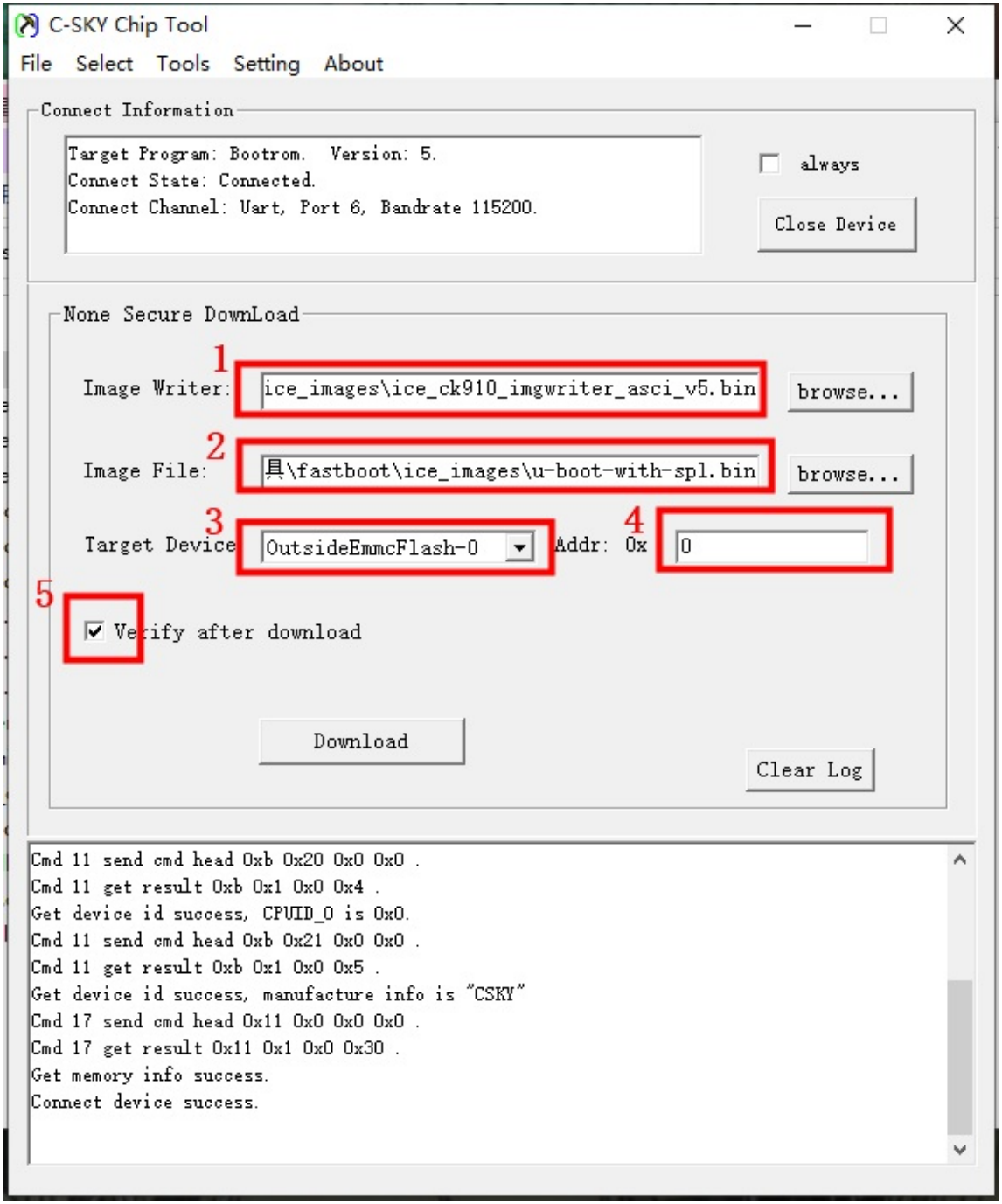

C-SKY Chip Tool
File
Select
Tools
Setting
About
1
Connect Setting
Jtag Enable Setting
Connect Information
always
Connect Device
Connect Setting
JTAG
Config
ICE Clk:
12000
KHz
Other Config
3
MEM R.W Speed:
80
k/s
IMGWriter Baudrate:
115200
UART
Config
2
Port:
COM11
Baud Rate:
115200
Byte Size:
8
Stop Bits:
1
Net Server Config
AppKey:
AppSecret:
确定
取消

连接设备



配置烧写目标参数

连接成功之后，需要配置镜像烧写参数，配置方法如下图所示：



1. Image Writer: 设置 ImageWriter 固件文件
2. Image File: 设置将要烧写到 EVB 板的镜像文件
3. Target Device: 镜像烧写的目标设备（常见选择为“OutsideEmmcFlash-0”，意为 eMMC 的 boot0 区域）
4. Addr: 镜像烧写到存储设备的偏移值。（FYI，图中文件名为示意文件名，具体视产品项目而定)

开始烧写

点击用户界面上的 "Download" 按钮，镜像就会开始自动下载与烧写。完成后，在完成之后，在用户界面的日志窗口中，将会出现“Download image .... success” 的内容提示，表示镜像完成了烧写。

烧写 Linux

安装 fastboot

最新 Fastboot 工具可从官网下载，下载地址：https://developer.android.com/studio/releases/platform-tools

安装 Android tools，内部包含 fastboot 命令。

在命令行下执行 fastboot 工具烧写镜像，烧写流程参考：1-Linux环境下烧写镜像

系统配置

1. 调整分区大小

debian-rootfs.ext4 烧写到 eMMC后，需要通过 `resize2fs` 命令调整分区大小，命令如下：

```
# 调整根文件系统分区
resize2fs /dev/mmcblk0p3
# 调用 boot 分区
resize2fs /dev/mmcblk0p2
```

调整成功后，显示如下：

```
Filesystem at /dev/mmcblk0p3 is mounted on /; on-line resizing required
old_desc_blocks = 2, new_desc_blocks = 30
[  209.862639] EXT4-fs (mmcblk0p3): resized filesystem to 3803136
The filesystem on /dev/mmcblk0p3 is now 3803136 (1k) blocks long.
```

可通过 `df -h` 查看磁盘空间使用情况。

2. 配置网络

动态分配IP

通过命令 `dhclient` 来获取 IP地址

```
# 配置网卡
cat > /etc/network/interfaces <<EOF
source-directory /etc/network/interfaces.d
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet dhcp
EOF

# 重启网络服务
systemctl restart networking.service
```

静态指定IP

```
# 配置网卡
cat > /etc/network/interfaces <<EOF
source-directory /etc/network/interfaces.d
auto lo
```

```
iface lo inet loopback

auto eth0
iface eth0 inet static
    address 192.168.1.100
    netmask 255.255.255.0
    gateway 192.168.1.1
EOF

# 重启网络服务
systemctl restart networking.service
```

3. 设置时区

```
$ ln -sf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime
# 或者：
timedatectl list-timezones | grep -i asia
timedatectl set-timezone Asia/Shanghai
```

4. 设置语言环境

```
# 安装
apt install -y locales
# 编辑 /etc/locale.gen
sed -i 's/# en_US.UTF-8 UTF-8/  en_US.UTF-8 UTF-8/g' /etc/locale.gen
sed -i 's/# zh_CN.UTF-8 UTF-8/  zh_CN.UTF-8 UTF-8/g' /etc/locale.gen
locale-gen

# 显示正在使用的 locale 和相关的环境变量
$ locale
LANG=en_US.utf8
LANGUAGE=
LC_CTYPE="en_US.utf8"
LC_NUMERIC="en_US.utf8"
LC_TIME="en_US.utf8"
LC_COLLATE="en_US.utf8"
LC_MONETARY="en_US.utf8"
LC_MESSAGES="en_US.utf8"
LC_PAPER="en_US.utf8"
LC_NAME="en_US.utf8"
LC_ADDRESS="en_US.utf8"
LC_TELEPHONE="en_US.utf8"
LC_MEASUREMENT="en_US.utf8"
LC_IDENTIFICATION="en_US.utf8"
LC_ALL=

# 查看已经生产的 locale
```

```
$ localedef --list-archive
en_US.utf8
zh_CN.utf8
# or
$ localectl list-locales
C.UTF-8
en_US.UTF-8
zh_CN.UTF-8

# 设置默认语言环境
$ localectl set-locale LANG=en_US.UTF-8
# 或者编辑 /etc/locale.conf 或 ~/.config/locale.conf
LANG=zh_CN.UTF-8
LC_COLLATE=C
LC_TIME=zh_CN.UTF-8
```

5. 配置SSH Server

```
# 安装
apt update
apt install -y ssh openssh-server

# 配置
vim /etc/ssh/sshd_config

# To disable tunneled clear text passwords, change to no here!
PasswordAuthentication yes
#PermitEmptyPasswords no

PermitRootLogin yes

# 启动
systemctl start sshd
systemctl restart sshd
```

6. 安装开发环境

```
apt update
apt install -y build-essential autoconf automake
```

7. 安装桌面

```
apt update

# 安装 xfce 桌面
apt install -y xfce4 dbus-x11
```

```
# 安装浏览器
apt install -y epiphany-browser

# 安装软键盘
apt install -y florence
```

概述

本章介绍如何通过源代码，进行 ICE 芯片的定制化开发。主要介绍内容包括：

第一步：搭建开发环境

第二步：编译 u-boot、OpenSBI、Kernel

第三步：制作根文件系统

搭建开发环境

方法一：使用 thead 工具安装

通过 thead 命令下载并安装 RISC-V 工具链，如果未安装 thead 工具，可通过 `sudo pip install thead_tools` 安装，thead 命令安装成功后，可通过如下命令安装工具链：

```
thead toolchain -r
```

安装成功后，控制台显示信息如下：

```
Start to download toolchain: riscv64-linux
 100.00% [#####################################################] Speed: 3.603MB/S
Start install, wait half a minute please.
Congratulations!
please run command:
  source /Users/zhuzhiguo/.zshrc
```

安装成功后，将 risc-v 工具链路径加入到 PATH环境变量中：

```
export PATH=$HOME/.thead/riscv64-linux/bin:$PATH
```

方法二：通过 tar 包安装

```
wget "http://yoctools.oss-cn-beijing.aliyuncs.com/\
riscv64-linux-x86_64-20201104.tar.bz2"

sudo mkdir -p /opt/riscv64-linux
sudo tar xf riscv64-linux-x86_64-20201104.tar.gz -C /opt/riscv64-linux

# 将 toolchain 加入到 PATH 环境变量中
export PATH=/opt/riscv64-linux/bin:$PATH
```

编译 u-boot

下载与编译

```
git clone "https://gitee.com/thead-linux/u-boot.git" -b master
make ice_evb_c910_defconfig
make -j ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu-
```

```
  LD      u-boot
  OBJCOPY u-boot.srec
  OBJCOPY u-boot-nodtb.bin
  SYM     u-boot.sym
  COPY    u-boot.bin
  CC      spl/lib/display_options.o
  LD      spl/lib/built-in.o
  LD      spl/u-boot-spl
  OBJCOPY spl/u-boot-spl-nodtb.bin
  COPY    spl/u-boot-spl.bin
  CAT     u-boot-with-spl.bin
```

烧写

方法一：使用 CCT 烧写，方法参见： Linux环境下烧写镜像

方法二：将 u-boot-with-spl.bin 传输到 evb 开发板上，执行 `dd if=u-boot-with-spl.bin of=/dev/mmcblk0boot0`

编译 OpenSBI

下载与编译

```
git clone "https://gitee.com/thead-linux/opensbi.git" -b master
make -j PLATFORM=thead/c910 \
CROSS_COMPILE=riscv64-unknown-linux-gnu- \
FW_TEXT_START=0x00000000 \
FW_JUMP_ADDR=0x00200000
```

烧写

```
scp opensbi/build/platform/thead/c910/firmware/fw_jump.bin \
  root@192.168.1.100:/boot/fw_jump.bin
```

将 `192.168.1.100` 地址换成 EVB 开发板的网络IP 地址

编译 Linux 内核

下载与编译

```
git clone "https://gitee.com/thead-linux/kernel.git" -b ice
make ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu- ice_defconfig
make -j ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-gnu-

mkimage -A riscv -O linux -T kernel -C none \
  -a 0x00200000 -e 0x00200000 -n Linux \
  -d arch/riscv/boot/Image uImage
```

烧写

```
scp uImage root@192.168.1.100:/boot/uImage
scp arch/riscv/boot/dts/thead/ice-evb.dtb root@192.168.1.100:/boot/hw.dtb
```

将 `192.168.1.100` 地址换成 EVB 开发板的网络 IP 地址。

Buildroot 构建

Buildroot 用来构建根文件系统。

下载与编译

```
git clone https://gitee.com/thead-linux/buildroot
cd buildroot
make thead_ice_evb_c910_br_defconfig
git clone https://gitee.com/thead-linux/dl.git
make source
make
```

编译过程较长，请耐心等待。

查看编译结果

```
ls output/images
```

显示如下：

```
root@linux:~/works/ice/buildroot > ls output/images -l
total 26804
-rw-r--r--. 1 thead thead 62914560 Dec 24 12:41 rootfs.ext2
```

烧写

EVB 开发板进入用fastboot 模式(`fastboot udp`)，使用 fastboot 工具烧写， 详见： Linux环境下烧写镜像

```
# 烧写 root 分区
fastboot -s udp:192.168.1.100 -S 5M flash root rootfs.ext2
```

启动镜像构建

制作 boot.ext4

方法1：使用 **make_ext4fs** 命令

如果未找到 `make_ext4fs` ，请使用 `sudo pip install thead-tools` 安装，或者通过源码编译得到 `make_ext4fs` ：

```
git clone https://github.com/superr/make_ext4fs.git
cd make_ext4fs
make
sudo cp make_ext4fs /usr/bin/
```

制作 boot.ext4（需要先编译opensbi和linux kernel）：

```
mkdir boot
cp opensbi/build/platform/thead/c910/firmware/fw_jump.bin boot/fw_jump.bin boot/.
cp kernel/uImage boot/hw.dtb boot/.
cp kernel/arch/riscv/boot/dts/thead/ice-evb.dtb boot/uImage boot/.

make_ext4fs -l 20M boot.ext4 boot
```

方法2：使用 **mkfs.ext4** 与 **mount**

该方法需要 sudo 权限

```
truncate -s 20M boot.ext4
mkfs.ext4 boot.ext4
mkdir boot
sudo mount boot.ext4 boot

cp opensbi/build/platform/thead/c910/firmware/fw_jump.bin boot/fw_jump.bin boot/.
cp kernel/uImage boot/hw.dtb boot/.
cp kernel/arch/riscv/boot/dts/thead/ice-evb.dtb boot/uImage boot/.

sudo umount boot.ext4
```

构建 Debian 根镜象

构建新的 debian 镜象，最好在已经安装好的 EVB 开发板的 Linux 环境下进行。

准备环境

```
apt update
apt install -y mmdebstrap
```

生成 rootfs目录

通过 mmdebstrap 命令制作 Debian 根文件系统：

```
# 生成 deiban rootfs
mkdir debian_rootfs

mmdebstrap --architectures=riscv64 sid debian_rootfs \
--include="debian-ports-archive-keyring,net-tools,wget,openssh-server,haveged" \
--dpkgopt='path-exclude=/usr/share/man/*' \
--dpkgopt='path-include=/usr/share/man/man[1-9]/*' \
--dpkgopt='path-exclude=/usr/share/locale/*' \
--dpkgopt='path-include=/usr/share/locale/locale.alias' \
--dpkgopt='path-exclude=/usr/share/doc/*' \
--dpkgopt='path-include=/usr/share/doc/*/copyright' \
--dpkgopt='path-include=/usr/share/doc/*/changelog.Debian.*' \
--dpkgopt='path-exclude=/usr/share/{doc,info,man,omf,help,gnome/help}/*' \
"deb http://mirrors.aliyun.com/debian-ports sid main" \
"deb http://mirrors.aliyun.com/debian-ports unreleased main" \
"deb http://mirrors.aliyun.com/thead sid main" \

# 添加 thead riscv 仓库
wget "http://mirrors.aliyun.com/thead/thead-keys/thead-repo.gpg" \
-O debian_rootfs/etc/apt/trusted.gpg.d/thead-repo.gpg

echo "" >> debian_rootfs/etc/apt/sources.list
echo "deb http://mirrors.aliyun.com/thead/debian-riscv64 sid main" \
>> debian_rootfs/etc/apt/sources.list

# 使能 root ssh 登录
echo "PermitRootLogin yes" >> debian_rootfs/etc/ssh/sshd_config

# 删除 root 登录密码
sed -i 's/^root:x:/root::/g' debian_rootfs/etc/passwd

# 查看文件夹大小
du -sm debian_rootfs
```

```
# 创建镜像文件
truncate -s 160M debian-rootfs.ext4
mkfs.ext4 debian-rootfs.ext4
mkdir tmp_rootfs
mount debian-rootfs.ext4 tmp_rootfs

mv debian_rootfs/* tmp_rootfs/

# umount
umount tmp_rootfs
```

烧写

方法一：使用 CCT 烧写，方法参见： Linux环境下烧写镜像

方法二：将 debian-rootfs.ext4 传输到 evb 开发板上， `dd if=debian-rootfs.ext4 of=/dev/mmcblk0p3`

使用玄铁 910-VECTOR 核

ICE 芯片内部有三个玄铁910处理器，其中 CPU0与 CPU1为同构处理器，不支持 VECTOR 指令，CPU2支持VECTOR 扩展指令集。在 ICE 芯片中使用 VECTOR 核时，可以通过 U-Boot 配置来实现两个环境的切换，具体操作如下：

切换到 **Vector** 核：

```
setenv boot_vector 1
saveenv
boot
```

切换到非 **VECTOR** 核：

```
setenv boot_vector 0
saveenv
boot
```

概述

功能演示

dhrystone

安装

```
apt install -y dhrystone
```

dhrystone 是一个用来测量 CPU 性能的工具

```
root@thead-910:~# dhrystone
Dhrystone Benchmark, Version 2.1 (Language: C)

Execution starts, 600000000 runs through Dhrystone
Execution ends

Final values of the variables used in the benchmark:

Int_Glob:            5
        should be:   5
Bool_Glob:           1
        should be:   1
Ch_1_Glob:           A
        should be:   A
Ch_2_Glob:           B
        should be:   B
Arr_1_Glob[8]:       7
        should be:   7
Arr_2_Glob[8][7]:    600000010
        should be:   Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:          481216
        should be:   (implementation-dependent)
  Discr:             0
        should be:   0
  Enum_Comp:         2
        should be:   2
  Int_Comp:          17
        should be:   17
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
  Ptr_Comp:          481216
        should be:   (implementation-dependent), same as above
  Discr:             0
        should be:   0
  Enum_Comp:         1
```

```
        should be:   1
  Int_Comp:          18
        should be:   18
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:           5
        should be:   5
Int_2_Loc:           13
        should be:   13
Int_3_Loc:           7
        should be:   7
Enum_Loc:            1
        should be:   1
Str_1_Loc:           DHRYSTONE PROGRAM, 1'ST STRING
        should be:   DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:           DHRYSTONE PROGRAM, 2'ND STRING
        should be:   DHRYSTONE PROGRAM, 2'ND STRING

Microseconds for one run through Dhrystone:    0.1
Dhrystones per Second:                      12329495.0
```

coremark

安装

```
apt install -y coremark
```

```
# coremark
2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 12970
Total time (secs): 12.970000
Iterations/Sec   : 8481.110254
Iterations       : 110000
Compiler version : GCC8.1.0
Compiler flags   :    -lrt
Memory location  : Please put data memory location here
                        (e.g. code in flash, data on heap etc)
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0x33ff
Correct operation validated. See readme.txt for run and reporting rules.
CoreMark 1.0 : 8481.110254 / GCC8.1.0    -lrt / Heap
```

latencytop

内核需要开启 **CONFIG_LATENCYTOP**

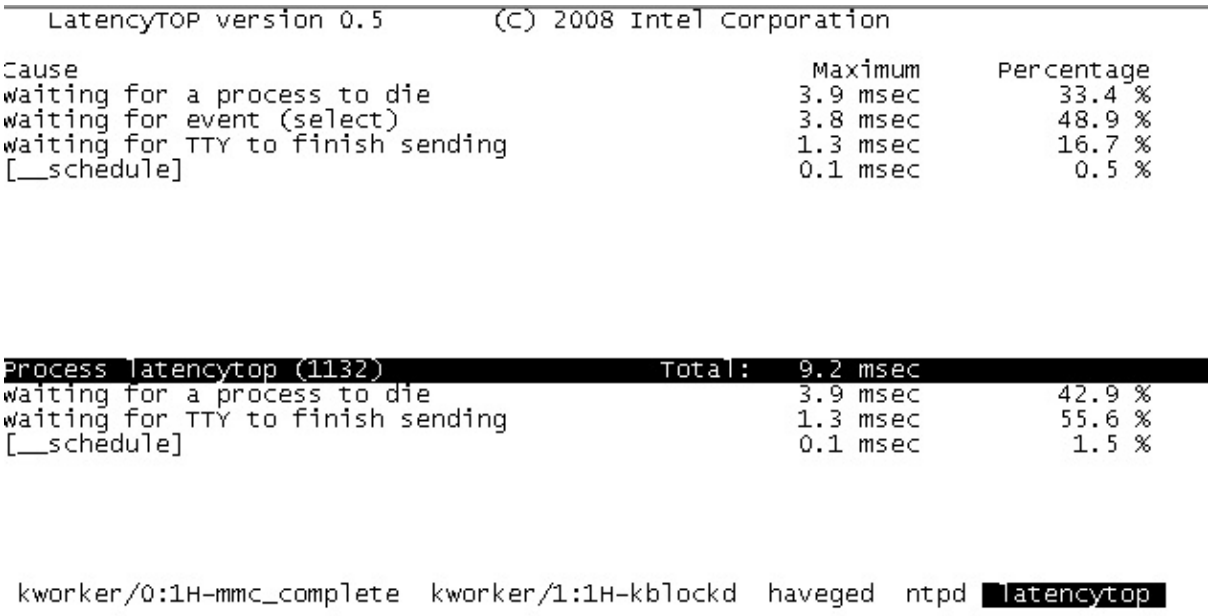
安装

```
apt install -y latencytop
```

latencytop 可以观察一定时间内，每一个进程在运行过程中发生的调度的次数和原因分布

```
   LatencyTOP version 0.5       (C) 2008 Intel Corporation

Cause                                                Maximum     Percentage
Waiting for a process to die                       3.8 msec         29.4 %
Waiting for event (select)                         3.7 msec         53.6 %
Waiting for TTY to finish sending                  1.3 msec         16.0 %
[__schedule]                                       0.2 msec          0.5 %

Process latencytop (1007)                 Total:   9.1 msec
Waiting for a process to die                       3.8 msec         42.3 %
Waiting for TTY to finish sending                  1.3 msec         55.7 %
[__schedule]                                       0.2 msec          1.9 %

 kworker/1:1-mm_percpu_wq  kworker/0:1H-mmc_complete  haveged  ntpd  latencytop
```

上面一部分是内核态的，下面一部分对应的是每一个进程



高亮的latencytop表示latencytop这个程序自身在运行过程中发生的进程切换次数和原因分布

memstat

安装

```
apt install -y memstat
```

memstat 详尽地分析了应用层内存使用的情况，并从匿名页和文件页两个角度给出了详尽的数据

```
# memstat
    288k: PID     1 (/bin/busybox)
    288k: PID    69 (/bin/busybox)
    288k: PID    73 (/bin/busybox)
   1752k: PID    78 (/bin/busybox)
   5988k: PID    94 (/usr/sbin/haveged)
  66536k: PID   106 (/usr/sbin/ntpd)
    572k: PID   111 (/usr/sbin/sshd)
    288k: PID   116 (/bin/busybox)
    672k: PID   117 (/bin/bash)
    300k: PID   894 (/usr/bin/memstat)
    536k(    516k): /bin/bash 117
   1284k(   1224k): /bin/busybox 1 69 73 78 116 1 69 73 78 116
    144k(     92k): /lib/ld-2.29.so 78 94 106 111 117 894 78 94 106 111 117 ...
     40k(     16k): /lib/libatomic.so.1.2.0 106 111 106 111 106 111
   1196k(   1028k): /lib/libc-2.29.so 78 94 106 111 117 894 78 94 106 111 11...
     36k(     24k): /lib/libcrypt-2.29.so 111
     32k(      8k): /lib/libdl-2.29.so 106 111 117 106 111 117 106 111 117
     76k(     68k): /lib/libgcc_s.so.1 106
    528k(    520k): /lib/libm-2.29.so 106
     24k(     16k): /lib/libnss_dns-2.29.so 106
     64k(     32k): /lib/libnss_files-2.29.so 78 106 111 117 78 106 111 117 ...
     96k(     80k): /lib/libpthread-2.29.so 106 111 106 111 106 111
     76k(     52k): /lib/libresolv-2.29.so 106 111 106 111 106 111
     16k(      8k): /lib/libutil-2.29.so 111
     16k(      8k): /usr/bin/memstat 894
   1868k(   1516k): /usr/lib/libcrypto.so.1.1 106 111 106 111 106 111
     92k(     84k): /usr/lib/libhavege.so.1.1.0 94
     32k(     24k): /usr/lib/libhistory.so.8.0 117
    264k(    240k): /usr/lib/libncursesw.so.6.1 117
    236k(    200k): /usr/lib/libreadline.so.8.0 117
    456k(    344k): /usr/lib/libssl.so.1.1 106 111 106 111 106 111
     68k(     60k): /usr/lib/libz.so.1.2.11 111
     24k(     16k): /usr/sbin/haveged 94
    572k(    536k): /usr/sbin/ntpd 106
    520k(    504k): /usr/sbin/sshd 111
```

ramsmp

ramsmp 测试了内存在操作浮点数和整数时的性能

```
# ramsmp -b6
RAMspeed/SMP (GENERIC) v3.5.0 by Rhett M. Hollander and Paul V. Bolotoff, 2002-09
```

```
8Gb per pass mode, 2 processes

FL-POINT  Copy:      5893.97 MB/s
FL-POINT  Scale:     6008.67 MB/s
FL-POINT  Add:       5067.11 MB/s
FL-POINT  Triad:     4047.71 MB/s
---
FL-POINT  AVERAGE:   5254.36 MB/s

# ramsmp -b3
RAMspeed/SMP (GENERIC) v3.5.0 by Rhett M. Hollander and Paul V. Bolotoff, 2002-09

8Gb per pass mode, 2 processes

INTEGER   Copy:      5920.51 MB/s
INTEGER   Scale:     5859.49 MB/s
INTEGER   Add:       3694.03 MB/s
INTEGER   Triad:     4998.79 MB/s
---
INTEGER   AVERAGE:   5118.20 MB/s
```

ramspeed

和 ramsmp 是一样的功能

stress_ng

安装

```
apt install -y stress-ng
```

```
stress-ng -c 2 --cpu-method all
```

stress-ng 会让 cpu 做一些 cpu 密集型的计算，产生 2个 worker（ICE上是 2个 core），使用 30多种不同的压力算法，如 pi, crc16, fft等，给 cpu 造成巨大压力

stress

stress 是 stress-ng 的低端版本，下面直接介绍 stess-ng

tinymembench

tinymembench 顾名思义，是一个 tiny 版本的专门针对 memory 的 benchmark，测量了各种带宽和延迟相关的性能参数

```
# tinymembench
tinymembench v0.4 (simple benchmark for memory throughput and latency)
... ...
 C copy backwards                                     :   3441.8 MB/s
 C copy backwards (32 byte blocks)                    :    641.2 MB/s (1.1%)
 C copy backwards (64 byte blocks)                    :    763.7 MB/s (0.2%)
 C copy                                               :   2410.1 MB/s (0.7%)
 C copy prefetched (32 bytes step)                    :   3481.1 MB/s
 C copy prefetched (64 bytes step)                    :   3482.3 MB/s
 C 2-pass copy                                        :   1770.6 MB/s (0.1%)
 C 2-pass copy prefetched (32 bytes step)             :   2399.3 MB/s
 C 2-pass copy prefetched (64 bytes step)             :   2402.0 MB/s
 C fill                                               :   6475.5 MB/s (0.4%)
 C fill (shuffle within 16 byte blocks)               :   6562.2 MB/s (0.4%)
 C fill (shuffle within 32 byte blocks)               :   1395.0 MB/s (0.3%)
 C fill (shuffle within 64 byte blocks)               :   1506.2 MB/s (0.2%)
 ---
 standard memcpy                                      :   3479.3 MB/s
 standard memset                                      :   6474.8 MB/s (0.4%)
... ...
block size : single random read / dual random read
      1024 :    0.0 ns          /     0.0 ns
      2048 :    0.0 ns          /     0.0 ns
      4096 :    0.0 ns          /     0.2 ns
      8192 :    0.4 ns          /     0.4 ns
     16384 :    0.6 ns          /     0.6 ns
     32768 :    0.7 ns          /     0.7 ns
     65536 :    0.8 ns          /     0.8 ns
    131072 :   11.7 ns          /    17.5 ns
    262144 :   17.2 ns          /    24.0 ns
    524288 :   20.0 ns          /    27.1 ns
   1048576 :   21.5 ns          /    28.8 ns
   2097152 :   25.8 ns          /    36.0 ns
   4194304 :  114.9 ns          /   169.7 ns
   8388608 :  172.3 ns          /   228.6 ns
  16777216 :  203.2 ns          /   254.4 ns
  33554432 :  222.5 ns          /   273.6 ns
  67108864 :  236.2 ns          /   291.9 ns
```

cache_calibrator

cache_calibrator 可以测量 cache 相关的性能参数

```
# cache_calibrator 1200 10M file

Calibrator v0.9e
(by Stefan.Manegold@cwi.nl, http://www.cwi.nl/~manegold/)
e16a9010 274364796944 4096    16
e16a9fff 274364801023 4096  4095
```

```
e16aa000 274364801024 4096     0

MINTIME = 10000

analyzing cache throughput...
      range      stride       spots     brutto-  netto-time
   10485760           8     1310720       17472         273

analyzing cache latency...
      range      stride       spots     brutto-  netto-time
   10485760           8     1310720      139776         273

analyzing TLB latency...
      range      stride       spots     brutto-  netto-time
    1474560        1152        1280       18176        1136


CPU loop + L1 access:       2.50 ns =   3 cy
             ( delay:       0.00 ns =   0 cy )

caches:
level  size    linesize   miss-latency        replace-time
  1     64 KB  128 bytes    9.19 ns =  11 cy    9.16 ns =  11 cy
  2      2 MB  128 bytes   19.05 ns =  23 cy   19.09 ns =  23 cy

TLBs:
level #entries  pagesize  miss-latency
  1       20       4 KB     3.34 ns =   4 cy

# ls
file.cache-replace-time.gp
file.TLB-miss-latency.data
file.TLB-miss-latency.gp
file.cache-miss-latency.data
file.cache-miss-latency.gp
file.cache-replace-time.data
```

其中 1200表示 1200MHz，也就是 1.2GHz，也即 ICE 开发板的 CPU 频率；跑完后生成了很多以“file”开头的文件，可以观测 cache 各个方面的性能

vmtouch

vmtouch 是一个可以操纵page cache的工具，它可以查看一个文件占用了多少 page cache，可以让一个文件完全浸泡在 page cache 里，也可以让一个文件完全从 page cache 中剥离出来

```
# vmtouch uImage
           Files: 1
     Directories: 0
  Resident Pages: 0/3433  0/13M  0%
```

```
         Elapsed: 0.000499 seconds

# free -m
              total        used        free      shared  buff/cache   available
Mem:           3448          67        3371           0           9        3368
Swap:             0           0           0
```

uImage 文件存放在 eMMC 上，通过 vmtouch 工具可见，uImage 大小为3433个pages，约13M，此时并没有占用 page cache

同时可以看到，此时的 free 内存为3371M

然后使用 vmtouch 将 uImage 完全纳入 page cache

```
# vmtouch -t uImage
           Files: 1
     Directories: 0
   Touched Pages: 3433 (13M)
         Elapsed: 0.61707 seconds
# free -m
              total        used        free      shared  buff/cache   available
Mem:           3448          67        3358           0          22        3363
Swap:             0           0           0
# vmtouch uImage
           Files: 1
     Directories: 0
  Resident Pages: 3433/3433  13M/13M  100%
         Elapsed: 0.001754 seconds
```

可以看到，uImage 已经完全浸泡在 page cache 中了，并且 free 内存也恰好少了13M

使用

```
# vmtouch -e uImage
           Files: 1
     Directories: 0
   Evicted Pages: 3433 (13M)
         Elapsed: 0.004634 seconds
# vmtouch uImage
           Files: 1
     Directories: 0
  Resident Pages: 0/3433  0/13M  0%
         Elapsed: 0.000446 seconds
```

可以将 uImage 从 page cache 中打捞出来

fio

安装

```
apt install -y fio
```

```
fio -filename=/dev/emcpowerb \
-iodepth 16 -thread -rw=randread \
-ioengine=psync -bs=4k -size=1G \
-numjobs=1 -runtime=180 \
-group_reporting -name=test1
```

输出：

```
test1: (g=0): rw=randread, bs=(R) 4096B-4096B, (W) 4096B-4096B, (T) 4096B-4096B, io
engine=psync, iodepth=16
fio-3.9
Starting 1 thread
Jobs: 1 (f=1): [r(1)][100.0%][r=330MiB/s][r=84.4k IOPS][eta 00m:00s]
test1: (groupid=0, jobs=1): err= 0: pid=167: Thu Jan  1 10:31:36 1970
   read: IOPS=84.9k, BW=332MiB/s (348MB/s)(1024MiB/3087msec)
    clat (usec): min=4, max=329, avg= 6.61, stdev= 3.91
     lat (usec): min=5, max=330, avg= 7.58, stdev= 4.26
    clat percentiles (usec):
     |  1.00th=[    6],  5.00th=[    6], 10.00th=[    6], 20.00th=[    7],
     | 30.00th=[    7], 40.00th=[    7], 50.00th=[    7], 60.00th=[    7],
     | 70.00th=[    7], 80.00th=[    7], 90.00th=[    8], 95.00th=[    8],
     | 99.00th=[    8], 99.50th=[    9], 99.90th=[   70], 99.95th=[  122],
     | 99.99th=[  151]
   bw (  KiB/s): min=334910, max=340294, per=99.46%, avg=337844.33, stdev=2086.47,
samples=6
   iops        : min=83727, max=85073, avg=84460.50, stdev=521.51, samples=6
  lat (usec)   : 10=99.57%, 20=0.25%, 50=0.02%, 100=0.10%, 250=0.06%
  lat (usec)   : 500=0.01%
  cpu          : usr=29.16%, sys=69.51%, ctx=772, majf=0, minf=1
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwts: total=262144,0,0,0 short=0,0,0,0 dropped=0,0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=16

Run status group 0 (all jobs):
   READ: bw=332MiB/s (348MB/s), 332MiB/s-332MiB/s (348MB/s-348MB/s), io=1024MiB (10
74MB), run=3087-3087msec
```

可以看到，平均带宽是 332MB/s，平均延迟在 10us，更多参数的设置和解读参见：https://tobert.github.io/post/2014-04-17-fio-output-explained.html

blktrace

内核需要开启**CONFIG_BLK_DEV_IO_TRACE**

安装

```
apt install -y blktrace
```

```
# dd if=/dev/mmcblk0p3 of=/dev/zero bs=512K count=1024000 &
# blktrace /dev/mmcblk0p3 -o - | blkparse -i -
179,3    0        1     0.000000000     9  C  RA 1164800 + 1536 [0]
179,0    0        2     0.000552333   136  A  RA 1166336 + 512 <- (179,3) 1039360
179,3    0        3     0.000554333   136  Q  RA 1166336 + 512 [dd]
179,3    0        4     0.000570000   136  G  RA 1166336 + 512 [dd]
179,3    0        5     0.000574000   136  P   N [dd]
179,3    0        6     0.000578000   136  U   N [dd] 1
179,3    0        7     0.000581333   136  I  RA 1166336 + 512 [dd]
179,3    0        8     0.000627000    45  D  RA 1166336 + 512 [kworker/0:2H]
179,0    0        9     0.001208000   136  A  RA 1166848 + 1024 <- (179,3) 1039872
179,3    0       10     0.001209000   136  Q  RA 1166848 + 1024 [dd]
179,3    0       11     0.001217333   136  G  RA 1166848 + 1024 [dd]
179,3    0       12     0.001220333   136  P   N [dd]
... ...
179,3    0      103     0.171033333    45  D  RA 1174528 + 512 [kworker/0:2H]
CPU0 (179,3):
 Reads Queued:          13,     4352KiB  Writes Queued:           0,        0KiB
 Read Dispatches:       13,     4352KiB  Write Dispatches:        0,        0KiB
 Reads Requeued:         0               Writes Requeued:         0
 Reads Completed:       12,     4608KiB  Writes Completed:        0,        0KiB
 Read Merges:            0,        0KiB  Write Merges:            0,        0KiB
 Read depth:             2               Write depth:             0
 IO unplugs:            13               Timer unplugs:           0

Throughput (R/W): 26947KiB/s / 0KiB/s
Events (179,3): 103 entries
Skips: 0 forward (0 -   0.0%)
```

其中第六个字段非常有用：每一个字母都代表了 IO 请求所经历的某个阶段

- Q – 即将生成 IO 请求
- G – IO 请求生成
- I – IO 请求进入 IO Scheduler 队列
- D – IO 请求进入 driver
- C – IO 请求执行完毕

而第四个字段表示从走到这一个阶段的时间戳，通过这一信息，可以详尽地分析一个 IO 完整的流程，并分析出瓶颈在哪里

最后，还可以通过 btt 来绘制流程图，这里就不表了

iostat

内核需要开启 **CONFIG_TASK_IO_ACCOUNTING**

```
# iostat -h
iostat v2.2, (C) 1999-2005 by Greg Franks, Zlatko Calusic, Rick Lindsley, Arnaud De
sitter
Distributed under the terms of the GPL (see LICENSE file)
Usage: iostat [-cdDpPxh] [disks...] [interval [count]]
options:

        c - print cpu usage info
        d - print basic disk info
        D - print disk utilization info
        p - print partition info also
        P - print partition info only
        x - print extended disk info
        h - this help
```

当前版本的 iostat 功能较弱，能看到的信息不多

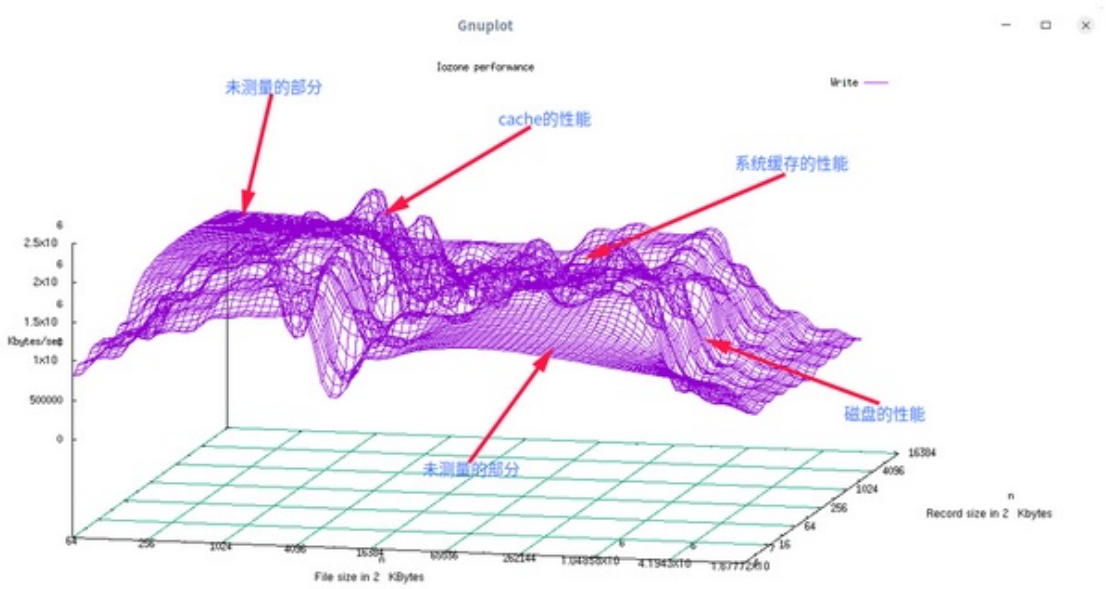
```
# iostat -cdDP
  mmcblk0       mmcblk0p3             cpu
 r/s  w/s   %b   r/s  w/s   %b   us  sy  wt  id
   5    0    2     5    0    2    0   0   7  92
```

iozone

iozone -a 可以给 ICE 上基于 eMMC 的文件系统进行一个全面测试

```
# iozone -a
        Iozone: Performance Test of File I/O
                Version $Revision: 3.483 $
                Compiled for 64 bit mode.
                Build: linux

        Contributors:William Norcott, Don Capps, Isom Crawford, Kirby Collins
                     Al Slater, Scott Rhine, Mike Wisner, Ken Goss
                     Steve Landherr, Brad Smith, Mark Kelly, Dr. Alain CYR,
                     Randy Dunlap, Mark Montague, Dan Million, Gavin Brebner,
                     Jean-Marc Zucconi, Jeff Blomberg, Benny Halevy, Dave Boone,
                     Erik Habbinga, Kris Strecker, Walter Wong, Joshua Root,
                     Fabrice Bacchella, Zhenghua Xue, Qin Li, Darren Sawyer,
                     Vangel Bojaxhi, Ben England, Vikentsi Lapa,
                     Alexey Skidanov, Sudhir Kumar.

        Run began: Thu Jan  1 09:39:29 1970

        Auto Mode
        Command line used: iozone -a
        Output is in kBytes/sec
        Time Resolution = 0.000001 seconds.
        Processor cache size set to 1024 kBytes.
        Processor cache line size set to 32 bytes.
        File stride size set to 17 * record size.
                                                            random    random     bkwd    record    stride
              kB  reclen    write  rewrite    read    reread    read     write     read   rewrite     read   fwrite frewrite   fread  freread
              64       4   158024   566371   955223   1122807  1066666  537815   914285   646464  1103448   516129   561403  1066666  1306122
              64       8   202531   659793  1488372   1422222  1488372  640000  1163636   810126  1523809   659793   673684  1391304  1599999
              64      16   213333   688172  1641025   1560975  1684210  673684  1391304   853333  1729729   771084   810126  1684210  1882352
              64      32   217687   719101  1729729   1641025  1882352  703296  1523809   842105  1882352  1015873  1207547  1828571  2064516
              64      64   219178   752941  1600000   2285714  2133333  744186  1641025   761904  1999999   771084   771084  1999999  2206896
             128       4   197836   606635  1174311   1174311  1163636  522448  1024000   707182  1103448   551724   547008  1230769  1292929
             128       8   212271   688172  1641025   1438202  1505882  663212  1319587   727272  1580246   656410   663212  1505882  1662337
             128      16   222222   744186  1828571   1599999  1828571  723163  1523809   962406  1828571   766467   766467  1753424  1828571
             128      32   226950   780487  2031746   1729729  1999999  643216  1777777   992248  1753424   934306   941176  1969230  2133333
             128      64   226950   785276  1855072   1969230  2133333  785276  1684210   876712  2098360   810126  1333333  2064516  2245614
             128     128   228980   795031  1753424   2327272  2327272  739884  1969230   805031  1523809   795031   805031  2245614  2370370
             256       4   198449   568888  1190697   1196261  1122807  573991  1003921   742028  1098712   537815   563876  1273631  1299492
```

iozone 本身可以生成 excel，结合 gnuplot 之类的画图工具还可以生成更复杂的剖面图



lmbench

lmbench 是一个很大的工具集合，下面介绍一些典型的功能

bw_mem

```
# bw_mem 1M rd
1.00 9565.06
```

表示测试 1MB数据量的读，速度为 9565MB/s

```
# bw_mem 10M cp
10.00 1218.92
```

表示测试 10MB数据量的拷贝，速度为 1218MB/s

lat_mem_rd

```
# lat_mem_rd 1M
"stride=64
0.00049 2.501
0.00098 2.501
0.00195 2.501
0.00293 2.501
0.00391 2.501
0.00586 2.501
0.00781 2.502
0.01172 2.503
```

```
0.01562 2.502
0.02344 2.503
0.03125 2.504
0.04688 3.790
0.06250 6.100
0.09375 6.198
0.12500 6.650
0.18750 6.741
0.25000 6.730
0.37500 6.710
0.50000 6.701
0.75000 6.695
1.00000 6.690
```

左边一列是读到的数据量（MB），右边一列是读的延时（ns）

mhz

```
# mhz
1199 MHz, 0.8340 nanosec clock
```

表示 cpu 运行的频率是 1.2GHz

当然，lmbench 还有很多测试用例，所以可以使用一个来自 ltp 的 lmbench 测试集 lmbench_tests.sh，该测试集涵盖了 lmbench 所有的测试用例，跑一遍下来，所有性能参数一目了然，如下：

```
# ./lmbench_test.sh
|TRACE LOG| ***** STARTING LMBENCH SCRIPT ***** |
|TRACE LOG| CREATING THE FILE OF 16MB SIZE FOR BENCHMARKING|
dd: error writing 'test1.txt': No space left on device
9+0 records in
8+0 records out
|TRACE LOG| ***** STARTING BANDWIDTH BENCHMARKS ***** |
|TRACE LOG| MEMORY BANDWIDTH BENCHMARKS |
|TEST START|bw_mem|
|TRACE LOG| Parameters      : |
|TRACE LOG| Operation       - rd|
|TRACE LOG| Memory Blk Size - 1M|
1.00 9563.10
|TEST RESULT|PASS|bw_mem|
|TEST END|bw_mem|
|TEST START|bw_mem|
|TRACE LOG| Parameters      : |
|TRACE LOG| Operation       - rd|
|TRACE LOG| Memory Blk Size - 2M|
2.00 9172.56
|TEST RESULT|PASS|bw_mem|
|TEST END|bw_mem|
```

```
|TEST START|bw_mem|
|TRACE LOG| Parameters      : |
|TRACE LOG| Operation       - rd|
|TRACE LOG| Memory Blk Size - 4M|
... ...
AF_UNIX sock stream bandwidth: 1903.41 MB/sec
|TEST RESULT|PASS|bw_unix|
|TEST END|bw_unix|
|TRACE LOG| BANDWIDTH OF FILE READ|
|TEST START|bw_file_rd|
|TRACE LOG| Parameters      : |
|TRACE LOG| Size            - 1MB|
|TRACE LOG| operation       - open2close|
|TRACE LOG| File            - test1.txt|
1.00 1658.93
|TEST RESULT|PASS|bw_file_rd|
|TEST END|bw_file_rd|
|TEST START|bw_file_rd|
|TRACE LOG| Parameters      : |
|TRACE LOG| Size            - 1MB|
|TRACE LOG| operation       - io_only|
|TRACE LOG| File            - test1.txt|
1.00 1652.89
... ...
"size=128k ovr=26.95
4 7.87
|TEST RESULT|PASS|lat_ctx|
|TEST END|lat_ctx|
|TEST START|lat_ctx|
|TRACE LOG| Parameters      : |
|TRACE LOG| procs           - 4|
|TRACE LOG| size_in_kbytes  - 256K|
|TRACE LOG| repetitions     - 100|
... ...
```

dstat

安装

```
apt install -y dstat
```

dstat 是一个整合了 vmstat、iostat、netstat、nfsstat 和 ifstat 的大杂烩

当前由于一些 python 颜色库的原因，还未能支持其跑起

nmon

安装

```
apt install -y nmon
```

nmon 也是一个综合性的性能观测工具

输入 nmon 会进入这个画面

```
+nmon-16g--------------------Hostname=buildroot----Refresh= 0secs ---09:01.14-+
|                                                                              |
|   ------------------------------                                             |
|    _ __  _ __ ___   ___  _ __          For help type H or ...                |
|   | '_ \| '_ ` _ \ / _ \| '_ \          nmon -?  - hint                      |
|   | | | | | | | | | (_) | | | |         nmon -h  - full details             |
|   |_| |_|_| |_| |_|\___/|_| |_|                                              |
|                                       To stop nmon type q to Quit            |
|   ------------------------------                                             |
|                                                                              |
|                                                                              |
|   Use these keys to toggle statistics on/off:                                |
|     c = CPU         l = CPU Long-term     - = Faster screen updates          |
|     C = " WideView  U = Utilisation       + = Slower screen updates          |
|     m = Memory      V = Virtual memory    j = File Systems                   |
|     d = Disks       n = Network           . = only busy disks/procs          |
|     r = Resource    N = NFS               h = more options                   |
|     k = Kernel      t = Top-processes     q = Quit                           |
+------------------------------------------------------------------------------+
```

再次输入“cUd”可以观测 cpu 使用率，io 使用率等性能参数

```
+nmon-16g------[H for help]---Hostname=buildroot----Refresh= 2secs ---09:03.30-+
| CPU Utilisation ------------------------------------------------------------|
|--------------------------+-----------------------------------------------+|
|CPU User%  Sys% Wait%  Idle|0          |25         |50          |75       100||
|  1  21.9   2.0   0.0  76.1|UUUUUUUUUU    >                                  ||
|  2   0.5   3.0   0.0  96.5|s         >                                     ||
|--------------------------+-----------------------------------------------+|
|Avg  11.0   2.2   0.0  86.8|UUUUUs >                                       ||
|--------------------------+-----------------------------------------------+|
| CPU Utilisation Stats ------------------------------------------------------|
|ALL    21.9    0.0    4.5  173.3    0.0    0.0    0.0    0.0    0.0    0.0    |
|CPU   User%  Nice%   Sys%  Idle%  Wait% HWirq% SWirq% Steal% Guest% GuestNice%|
|  1    21.9    0.0    2.0   76.2    0.0    0.0    0.0    0.0    0.0    0.0    |
|  2     0.5    0.0    3.0   96.6    0.0    0.0    0.0    0.0    0.0    0.0    |
| Disk I/O --/proc/diskstats----mostly in KB/s-----Warning:contains duplicates-|
|DiskName Busy  Read WriteKB|0          |25         |50          |75       100||
|mmcblk0    0%    0.0    0.0|>                                              ||
|mcblk0p3   0%    0.0    0.0|>                                              ||
|Totals Read-MB/s=0.0      Writes-MB/s=0.0      Transfers/sec=0.0             |
|------------------------------------------------------------------------------|
+------------------------------------------------------------------------------+
```

iperf3

安装

```
apt install -y iperf3
```

iperf 用于测量网络性能

在主机端启动一个 iperf3 server

```
Microsoft Windows [版本 10.0.18363.1256]
(c) 2019 Microsoft Corporation。保留所有权利。

D:\公司\工具\iperf-3.1.3-win64>iperf3.exe -s
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from 192.111.111.11, port 58022
[  5] local 192.111.111.123 port 5201 connected to 192.111.111.11 port 58024
[ ID] Interval           Transfer     Bandwidth
[  5]   0.00-1.00   sec  11.3 MBytes  94.8 Mbits/sec
[  5]   1.00-2.00   sec  11.1 MBytes  92.9 Mbits/sec
[  5]   2.00-3.00   sec  11.3 MBytes  94.4 Mbits/sec
[  5]   3.00-4.00   sec  11.3 MBytes  94.9 Mbits/sec
[  5]   4.00-5.00   sec  11.3 MBytes  94.9 Mbits/sec
[  5]   5.00-6.00   sec  11.3 MBytes  94.9 Mbits/sec
[  5]   6.00-7.00   sec  11.3 MBytes  94.9 Mbits/sec
[  5]   7.00-8.00   sec  11.3 MBytes  94.9 Mbits/sec
[  5]   8.00-9.00   sec  11.3 MBytes  94.9 Mbits/sec
[  5]   9.00-10.00  sec  11.3 MBytes  94.9 Mbits/sec
```

在板端启动一个 iperf3 测试程序

```
# iperf3 -c 192.111.111.123
Connecting to host 192.111.111.123, port 5201
[  5] local 192.111.111.11 port 58024 connected to 192.111.111.123 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  11.6 MBytes  96.8 Mbits/sec    0   92.7 KBytes
[  5]   1.00-2.00   sec  11.0 MBytes  92.5 Mbits/sec    0   92.7 KBytes
[  5]   2.00-3.00   sec  11.3 MBytes  94.6 Mbits/sec   54   94.1 KBytes
[  5]   3.00-4.00   sec  11.4 MBytes  95.6 Mbits/sec    0   98.4 KBytes
[  5]   4.00-5.00   sec  11.3 MBytes  94.6 Mbits/sec    0    115 KBytes
[  5]   5.00-6.00   sec  11.3 MBytes  95.1 Mbits/sec    0    115 KBytes
[  5]   6.00-7.00   sec  11.4 MBytes  95.6 Mbits/sec    0    115 KBytes
[  5]   7.00-8.00   sec  11.3 MBytes  94.6 Mbits/sec    0    137 KBytes
[  5]   8.00-9.00   sec  11.4 MBytes  95.6 Mbits/sec    0    137 KBytes
[  5]   9.00-10.00  sec  11.4 MBytes  95.6 Mbits/sec    0    137 KBytes
```

就可以测试网络性能了

dropwatch

内核需要开启 **CONFIG_NET_DROP_MONITOR**

```
# dropwatch -l kas
Initalizing kallsyms db
dropwatch> start
Enabling monitoring...
Kernel monitoring activated.
Issue Ctrl-C to stop monitoring
4 drops at __udp4_lib_rcv+670 (0xffffffe0004bf06e)
12 drops at __udp4_lib_rcv+670 (0xffffffe0004bf06e)
2 drops at __udp4_lib_rcv+6cc (0xffffffe0004bf0ca)
1 drops at __udp4_lib_rcv+6cc (0xffffffe0004bf0ca)
4 drops at __udp4_lib_rcv+670 (0xffffffe0004bf06e)
12 drops at __udp4_lib_rcv+670 (0xffffffe0004bf06e)
```

dropwatch 用来诊断网络丢包问题的，可以看到，在长期 ping 的过程中发生了 6次丢包，所在的函数均已列出，可以依据源代码去进行诊断

pv

安装

```
apt install -y pv
```

pv被用来显示命令在执行时的进度条

例如拷贝：

```
# pv messages > tmp
27.6KiB 0:00:00 [ 130MiB/s] [==================================>] 100%
```

而且pv还能限制传输的速度，例如1KB/s

```
# pv -L 1k messages > tmp
  10KiB 0:00:10 [1011 B/s] [===========>                          ] 36% ETA 0:00:17
```

或者dd：

```
# pv < /dev/mmcblk0p3 | dd of=/tmp/testfile bs=4M
 662MiB 0:00:02 [ 198MiB/s] [====>                                ] 17% ETA 0:00:09
```

whetstone

安装

```
apt install -y whetstone
```

whetstone 用来测量 cpu 的浮点运算能力

```
# whetstone 1000000
Loops: 1000000, Iterations: 1, Duration: 19.567 sec.
C Converted Double Precision Whetstones: 5110.747 MIPS
```

dieharder

安装

```
apt install -y dieharder
```

dieharder 是一款用于测试随机数生成能力的测试工具

```
# dieharder -a
#=============================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#=============================================================================#
   rng_name    |rands/second|   Seed   |
        mt19937|  3.34e+07  | 273137300|
#=============================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#=============================================================================#
   diehard_birthdays|   0|       100|     100|0.08654260|  PASSED
      diehard_operm5|   0|   1000000|     100|0.76268865|  PASSED
  diehard_rank_32x32|   0|     40000|     100|0.09896061|  PASSED
    diehard_rank_6x8|   0|    100000|     100|0.79845879|  PASSED
   diehard_bitstream|   0|   2097152|     100|0.90976807|  PASSED
        diehard_opso|   0|   2097152|     100|0.90149754|  PASSED
        diehard_oqso|   0|   2097152|     100|0.60943983|  PASSED
         diehard_dna|   0|   2097152|     100|0.28147819|  PASSED
diehard_count_1s_str|   0|    256000|     100|0.87571086|  PASSED
diehard_count_1s_byt|   0|    256000|     100|0.32803443|  PASSED
 diehard_parking_lot|   0|     12000|     100|0.43004782|  PASSED
    diehard_2dsphere|   2|      8000|     100|0.91557461|  PASSED
    diehard_3dsphere|   3|      4000|     100|0.10831747|  PASSED
     diehard_squeeze|   0|    100000|     100|0.58947448|  PASSED
        diehard_sums|   0|       100|     100|0.03804515|  PASSED
        diehard_runs|   0|    100000|     100|0.16852247|  PASSED
        diehard_runs|   0|    100000|     100|0.16901521|  PASSED
       diehard_craps|   0|    200000|     100|0.28860890|  PASSED
```

```
       diehard_craps|   0|    200000|     100|0.41000082|  PASSED

 marsaglia_tsang_gcd|   0|  10000000|     100|0.94040087|  PASSED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.98666060|  PASSED
         sts_monobit|   1|    100000|     100|0.99649756|   WEAK
            sts_runs|   2|    100000|     100|0.06977154|  PASSED
... ...
```

perf

perf 是一个全面的，可量化的性能分析工具，其中最有名的用法就是生成火焰图

例如，可以利用 perf 观察一下开发板在 iperf3 网络测试下 CPU 的负载

可以先启动一个 iperf3 测试，后台跑，无输出，不影响我们接下来的操作

```
iperf3 -c 192.111.111.123 -t 86400 > /dev/null 2>&1 &
```

然后再启动 perf，搜集当前 CPU 的数据

```
perf record -c 100000 -e cpu-cycles -g -a sleep 5
```

- -c 的参数代表采样间隔，100000表示 100000个 cycle，此时的时钟频率是 1.2GHz，一个 cycle 就是 1/1.2G秒，此时的采样间隔也就是 100000*1/1.2G* 1000 * 1000 = 83us，微秒级一般就可以了
- -e 的参数代表采样的事件，采的是一个 cpu-cycles 事件
- sleep 5 表示采样的时间为 5秒

执行完改命令，会生成一个 perf.data 文件，然后使用 perf script 命令处理该文件

```
perf script -i perf.data > perf.unfold
```

接下来，需要将这个 perf.unfold 文件拷贝到 pc 机的 linux 系统上来处理，因为后续需要进行 perl 脚本的处理并依赖浏览器打开火焰图文件

先下载 perl 相关的工具

```
git clone https://github.com/brendangregg/FlameGraph.git
```

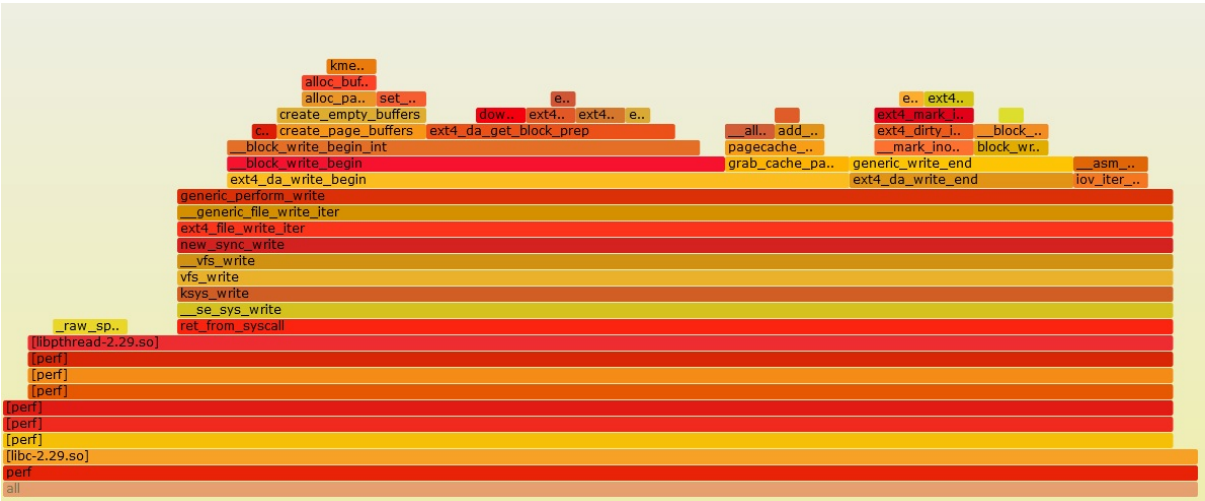
然后使用该工具处理 perf.data

```
./stackcollapse-perf.pl ../perf.unfold > perf.folded
./flamegraph.pl perf.folded > perf.svg
```

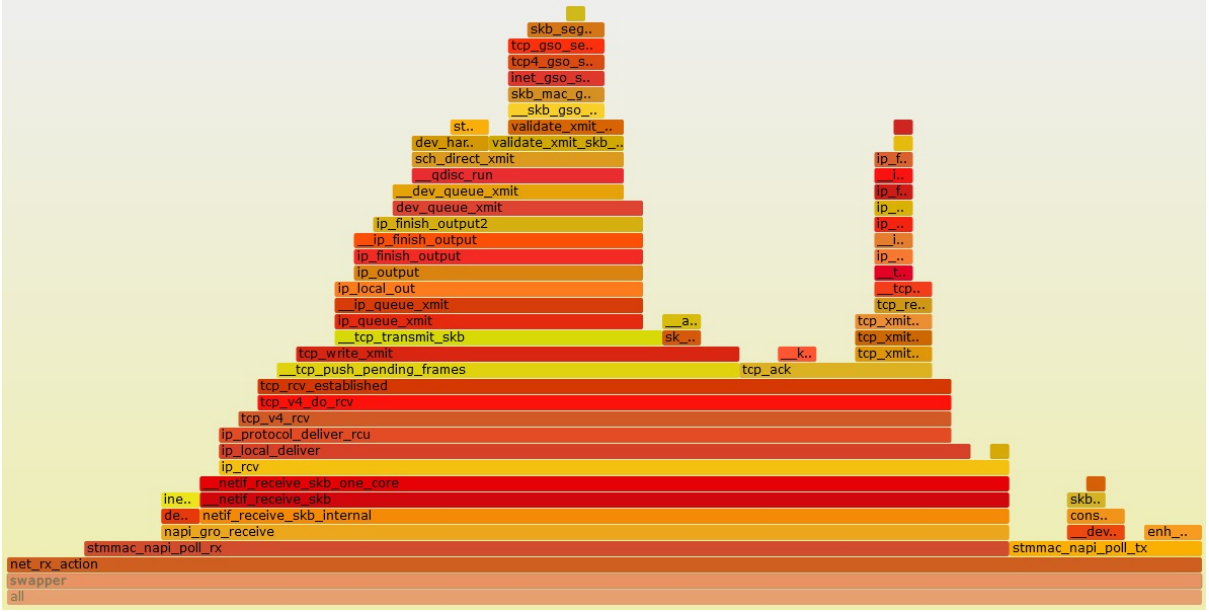
最后再用浏览器打开这个 perf.svg，就可以详细地看到这5秒钟内，CPU 的行为以及在各行为上花费的时间

可以看到，由于网络环境的原因，测试时的网速并不高，因此 CPU 在这5秒中内大部分也都处于 idle 状态

perf 的函数栈以及花费的时间比例可以通过点进 all 中的 perf 一栏看到



swapper 中用来处理网络接收的部分也可以通过进一步点击看到



ftrace

ftrace 可以记录每个函数极其子函数运行所花费的时间

```
echo nop > /sys/kernel/debug/tracing/current_tracer
echo 0 > /sys/kernel/debug/tracing/tracing_on
echo 42 > /sys/kernel/debug/tracing/set_ftrace_pid
# 此处探测的是stmmac_wq，42是它的pid
echo function_graph > /sys/kernel/debug/tracing/current_tracer
echo stmmac_napi_poll_rx > /sys/kernel/debug/tracing/set_graph_function
echo 1 > /sys/kernel/debug/tracing/tracing_on
cat /sys/kernel/debug/tracing/trace > trace.txt
```

打开 trace.txt

```
 0)               |  stmmac_napi_poll_rx() {
 0)   2.000 us    |    enh_desc_get_rx_status();
 0)   2.000 us    |    enh_desc_get_ext_status();
 0)   1.667 us    |    enh_desc_get_rx_frame_len();
 0)               |    __napi_alloc_skb() {
 0)   1.667 us    |      page_frag_alloc();
 0)               |      __build_skb() {
 0)               |        kmem_cache_alloc() {
 0)   1.333 us    |          should_failslab();
 0)   4.667 us    |        }
 0)   8.000 us    |      }
 0) + 14.667 us   |    }
 0)               |    dma_sync_single_for_cpu() {
 0)   1.666 us    |      arch_sync_dma_for_cpu();
 0)   4.333 us    |    }
 0)   1.667 us    |    skb_put();
```

```
  0)   1.667 us    |    page_pool_put_page();
  0)   1.667 us    |    eth_type_trans();
... ...
```

可以看到，trace.txt 里记录了每个函数运行所花费的时间

安装 GPU 驱动

安装驱动

```
apt update
apt install -y ice-drivers
```

运行测试程序

```
cd /opt/offical_tests/vdk/
./tutorial4_es20
```

GPU Demo 运行效果

Linux 小程序应用开发环境

下载并安装运行环境、Demo

```
apt update
apt install -y cube-miniapp
```

运行 Demo 程序

```
# 运行 Cube 小程序 Demo
cube-test 2021001142699009 pages/index/index
```

> - cube-test：cube 小程序框架
> - 2021001142699009： 小程序 appid，这个是由 cube-test 确定入口程序 id
> - pages/index/index：初始页面

注意：

1. 此 Demo 为离线方式小程序，实际小程序需要联网下载。
2. 小程序需要一个 HomeApp 调用，因此 Demo 直接调用，在初始页面再返回时，会返回到黑屏。

Cube 小程序简介

Cube 是一个可独立发布的跨平台 渲染引擎，在小程序这一技术体系内，仅作为一种不同于 `Web(WebView)` 的高性能渲染方式存在。

1. Cube 小程序特点
   - 支持使用 "类Web" 语言进行业务开发，快速开发体验
   - 兼顾性能和体验的跨平台 Native 渲染引擎，与 Native 应用相当的使用体验
2. 与 Web 的关系
   - 渲染能力支持上，基本实现了 w3c 样式规范的子集

- 在运行架构上，不基于 HTML 语言表示, 而基于节点级别的批量指令操作
- 在逻辑执行上，均通过 JavaScript 进行逻辑执行

示例：编译、运行 helloworld

1. 安装编译工具

```
# 全局安装
tnpm install -g  @ali/mini

# 如无权限或permission denied，请加sudo尝试安装
sudo tnpm install -g  @ali/mini
```

2. 初始化 helloworld 小程序项目

```
mkdir helloworld
cd helloworld
mini init
# 选择 Cube 小程序
? 请输入项目名称：  helloTinyApp
? 选择你的项目类型：  Cube 小程序

# 生成后的小程序项目目录
helloTinyApp
├── README.md
├── app.acss
├── app.js
├── app.json
├── mini.project.json
└── pages
    └── index
        ├── index.acss
        ├── index.axml
        ├── index.js
        └── index.json
```

3. 构建、打包小程序

```
# helloTinyApp Cube 小程序项目根目录下执行
mini build ./

...
Inject code for ./helloworld/helloTinyApp/miniapptools_dist/ng-main/index.html
failed because of file not exist
Inject code for ./helloworld/helloTinyApp/miniapptools_dist/ng-main/index.worke
r.js failed because of file not exist
>>> 构建完成
小程序产物包：./helloworld/helloTinyApp/miniapptools_dist/dist.tar
```

```
#出现上面 log 后，构建成功，构建产物为 miniapptools_dist/dist.tar
```

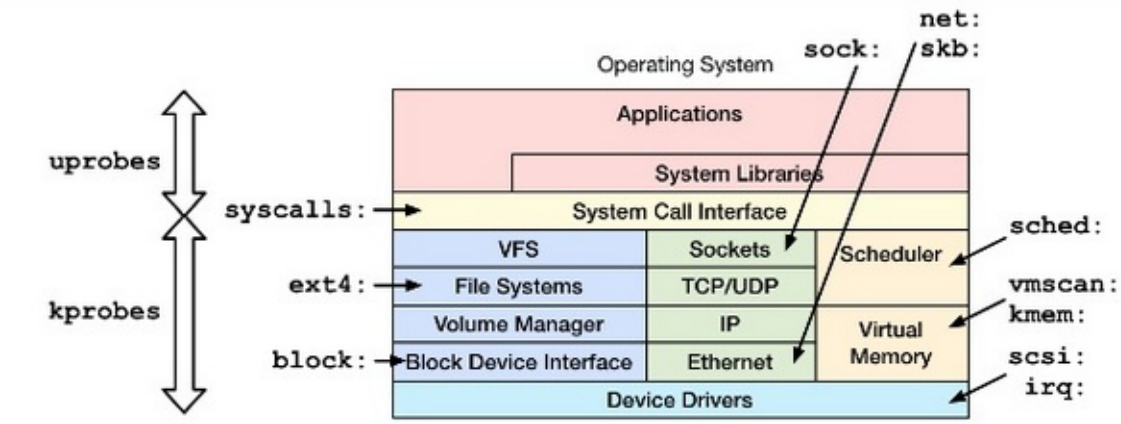
上传 dist.tar 到 ICE_EVB 的 `/resources/demo/pkg/2021001142699009` 目录下，并在 ICE EVB 板 Linux 环境下解压 dist.tar

```
# 在 PC 上执行 scp 命令，上传 dist.tar
scp miniapptools_dist/dist.tar root@192.168.1.100:/resources/demo/pkg/202100114
2699009/.

# 在 ICE_EVB 上解压 dist.tar
cd /resources/demo/pkg/2021001142699009/
tar xf dist.tar

# 启动更新后的小程序
cube-test 2021001142699009 pages/index/index
```

1. 可以使用支付宝的小程序开发者工具进行小程序模拟、调试。小程序开发者工具是针对 Web 小程序，Cube 小程序为其子集，不支持有些样式，造成构建时语法错误和显示效果不同。

Kprobe & Uprobe - Linux Tracing

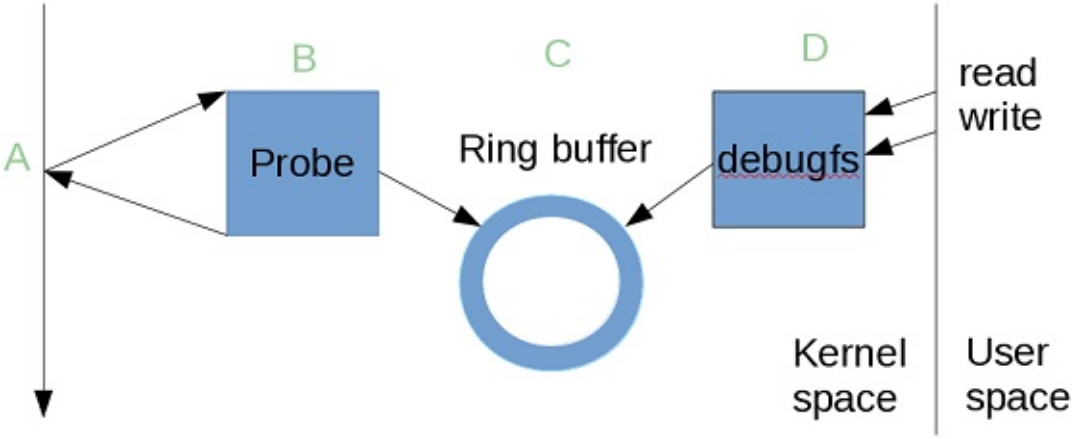
经过长期的发展, `kprobes/uprobes` 机制在事件(events)的基础上分别为内核态和用户态提供了追踪调试的功能, 这也构成了 tracepoint 机制的基础, 后期的很多工具, 比如 `perf_events` , `ftrace` 等都是在其基础上演化而来. 参考由 Brendan Gregg 提供的资料来看, `kprobes/uprobes` 在 Linux 动态追踪层面起到了基石的作用, 如下所示:



- kprobes: dynamic kernel tracing
  - function calls, returns, line numbers
- uprobes: dynamic user-level tracing

引用: https://blog.arstercz.com/introduction_to_linux_dynamic_tracing/

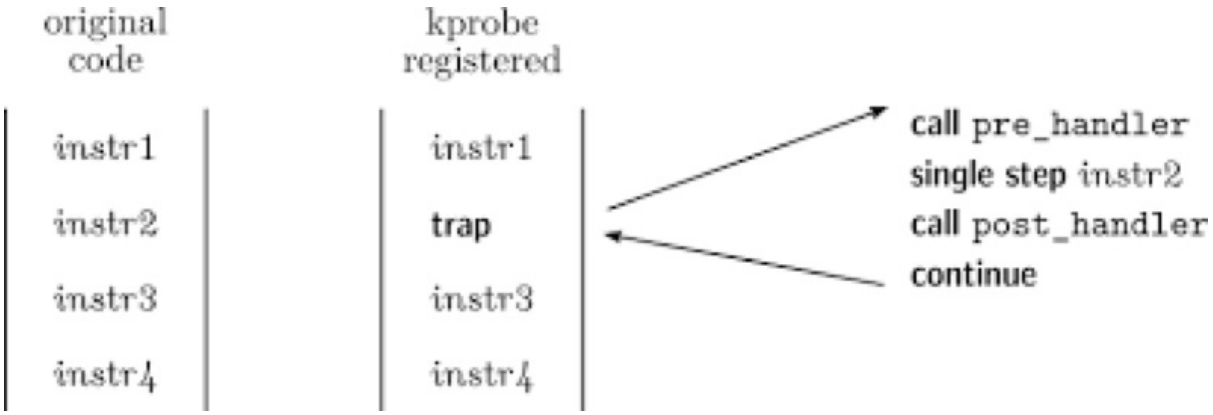
Kprobe 和 Uprobe 均可以通过 ftrace 的 /sys/debug/tracing interface (基于 debugfs 的用户空间层面的 API) 执行各种跟踪和分析，虽然 ftrace 的内部是复杂的, 不过输出的信息却以简单明了为主，更详细的使用示例可以参考 ftrace-lwn-365835, 如下图所示, 大致为 ftrace 的原理:



Krpobe 介绍

kprobe是linux内核的一个重要特性，是其他的内核调试工具（perf, systemtap）的“基础设施”，同时内核 BPF 也依赖 kprobe

它利用指令桩原理，截获指令流，并在指令执行前后插入hook函数：



如果需要知道内核函数是否被调用、被调用上下文、入参以及返回值，比较简单的方法是加printk，但是效率低。

利用kprobe技术，用户可以自定义自己的回调函数，可以再几乎所有的函数中动态插入探测点。

当内核执行流程执行到指定的探测函数时，会调用该回调函数，用户即可收集所需的信息了，同时内核最后还会回到原本的正常执行流程。如果用户已经收集足够的信息，不再需要继续探测，则同样可以动态的移除探测点。

kprobes技术包括的2种探测手段分别时kprobe 和 kretprobe:

- kprobe是最基本的探测方式，是实现后两种的基础，它可以在任意的位置放置探测点（就连函数内部的某条指令处也可以），它提供了探测点的调用前、调用后和内存访问出错3种回调方式，分别是pre_handler、post_handler和fault_handler，其中pre_handler函数将在被探测指令被执行前回调，post_handler会在被探测指令执行完毕后回调（注意不是被探测函数），fault_handler会在内存访问出错时被调用。
- retprobe从名字种就可以看出其用途了，它同样基于kprobe实现，用于获取被探测函数的返回值。

基本使用指南

开启内核：

```
Symbol: FTRACE [=y]

Type  : boolean

Prompt: Tracers

  Location:

(5) -> Kernel hacking
```

```
  Defined at kernel/trace/Kconfig:132

  Depends on: TRACING_SUPPORT [=y]

Symbol: KPROBE_EVENT [=y]

Type  : boolean

Prompt: Enable kprobes-based dynamic events

  Location:

    -> Kernel hacking

(1)   -> Tracers (FTRACE [=y])

  Defined at kernel/trace/Kconfig:405

  Depends on: TRACING_SUPPORT [=y] && FTRACE [=y] && KPROBES [=y] && HAVE_REGS_AND_
STACK_ACCESS_API [=y]
  Selects: TRACING [=y] && PROBE_EVENTS [=y]

Symbol: HAVE_KPROBES_ON_FTRACE [=y]

Type  : boolean

  Defined at arch/Kconfig:183

  Selected by: csky [=y]


Symbol: KPROBES_ON_FTRACE [=y]

Type  : boolean

  Defined at arch/Kconfig:79

  Depends on: KPROBES [=y] && HAVE_KPROBES_ON_FTRACE [=y] && DYNAMIC_FTRACE_WITH_RE
GS [=y]
```

终端运行:

首先通过 mount 获得 ftrace debug 接口，然后通过 kprobe_evets 注册你需要 probe 的内核函数，在 tracing/events/kprobes// 下可以控制该 kprobe 函数的 开启和关闭

```
# mount -t debugfs nodev /sys/kernel/debug/

# echo 'p:myprobe _do_fork dfd=%a0 filename=%a1 flags=%a2 mode=+4($stack)' > /sys/k
ernel/debug/tracing/kprobe_events
```

```
# echo 'r:myretprobe _do_fork $retval' >> /sys/kernel/debug/tracing/kprobe_events

# echo 1 > /sys/kernel/debug/tracing/events/kprobes/myprobe/enable
# echo 1 > /sys/kernel/debug/tracing/events/kprobes/myretprobe/enable

# cat /sys/kernel/debug/tracing/trace
# tracer: nop
#
# entries-in-buffer/entries-written: 8/8 #P:1
#
# _-----=> irqs-off
# / _----=> need-resched
# | / _---=> hardirq/softirq
# || / _--=> preempt-depth
# ||| / delay
# TASK-PID CPU# |||| TIMESTAMP FUNCTION
# | | | |||| | |
swapper/0-1 [000] dn.. 2.488544: Unknown type 599
swapper/0-1 [000] dn.. 2.489270: Unknown type 600
sh-121 [000] d... 408.113780: myprobe: (_do_fork+0x0/0x3ac) dfd=0xbe485f20 filename
=0x0 flags=0x0 mode=0x0
sh-121 [000] d... 408.117058: myretprobe: (sys_clone+0xa6/0xac <- _do_fork) arg1=0x
82
sh-121 [000] d... 409.816850: myprobe: (_do_fork+0x0/0x3ac) dfd=0xbe485f20 filename
=0x0 flags=0x0 mode=0x0
sh-121 [000] dn.. 409.817539: myretprobe: (sys_clone+0xa6/0xac <- _do_fork) arg1=0x
83
sh-121 [000] d... 411.202079: myprobe: (_do_fork+0x0/0x3ac) dfd=0xbe485f20 filename
=0x0 flags=0x0 mode=0x0
sh-121 [000] d... 411.202750: myretprobe: (sys_clone+0xa6/0xac <- _do_fork) arg1=0x
84
```

社区 kprobe 文档非常完善:

https://www.kernel.org/doc/Documentation/kprobes.txt

https://lwn.net/Articles/410200/

Trace-cmd 和 kernelshark:

Trace-cmd 是一个基于 ftrace 的用户态前端命令行工具，它的仓库在:

git://git.kernel.org/pub/scm/linux/kernel/git/rostedt/trace-cmd.git

很多发行版，都有 trace-cmd 的包，完善的 man 手册

具体使用参考:

https://lwn.net/Articles/410200/

Uprobe

ftrace 基础使用:

先准备一个应用程序，测试代码:

```
#include <stdio.h>
#include <unistd.h>

static void
print_curr_state_one(void)
{
    printf("This is the print current state one function\n");
}


static void
print_curr_state_two(void)
{
    printf("This is the print current state two function\n");
}


int main() {
    while(1) {
        print_curr_state_one();
        sleep(1);
        print_curr_state_two();
    }
}
```

编译并获取可执行文件和反汇编 (9 系列请使用 riscv-linux-gcc):

```
➜  csky-linux-gcc test.c -o test
➜  linux-next git:(linux-next-ftrace-kprobe-uprobe-simlutate-insn) ✗ ../artifacts/o
utput_860_next/images/host/bin/csky-linux-objdump -S test

00008518 <print_curr_state_one>:
    8518:       1422            subi            sp, sp, 8
    851a:       dd0e2000        st.w            r8, (sp, 0)
    851e:       ddee2001        st.w            r15, (sp, 0x4)
    8522:       6e3b            mov             r8, sp
    8524:       1006            lrw             r0, 0x8698      // 853c <print_curr
_state_one+0x24>
    8526:       eae00007        jsri            0x0     // from address pool at 0x8
540
    852a:       6c00            or              r0, r0
    852c:       6fa3            mov             sp, r8
```

```
    852e:       d9ee2001        ld.w            r15, (sp, 0x4)
    8532:       d90e2000        ld.w            r8, (sp, 0)
    8536:       1402            addi            sp, sp, 8
    8538:       783c            rts
    853a:       0000            .short  0x0000
    853c:       00008698        .long   0x00008698
    8540:       00000000        .long   0x00000000
00008544 <print_curr_state_two>:
    8544:       1422            subi            sp, sp, 8
    8546:       dd0e2000        st.w            r8, (sp, 0)
    854a:       ddee2001        st.w            r15, (sp, 0x4)
    854e:       6e3b            mov             r8, sp
    8550:       100d            lrw             r0, 0x86c8      // 8584 <main+0x1c>
    8552:       eae0000e        jsri            0x0     // from address pool at 0x8
588
    8556:       6c00            or              r0, r0
    8558:       6fa3            mov             sp, r8
    855a:       d9ee2001        ld.w            r15, (sp, 0x4)
    855e:       d90e2000        ld.w            r8, (sp, 0)
    8562:       1402            addi            sp, sp, 8
    8564:       783c            rts
        ...
00008568 <main>:
    8568:       1422            subi            sp, sp, 8
    856a:       dd0e2000        st.w            r8, (sp, 0)
    856e:       ddee2001        st.w            r15, (sp, 0x4)
    8572:       6e3b            mov             r8, sp
    8574:       e3ffffd2        bsr             0x8518  // 8518 <print_curr_state_o
ne>
    8578:       3001            movi            r0, 1
    857a:       eae00006        jsri            0x0     // from address pool at 0x8
590
    857e:       e3ffffe3        bsr             0x8544  // 8544 <print_curr_state_t
wo>
    8582:       07f9            br              0x8574  // 8574 <main+0xc>

➜  linux-next git:(linux-next-ftrace-kprobe-uprobe-simlutate-insn) ✗ ../artifacts/o
utput_860_next/images/host/bin/csky-linux-readelf -S test
There are 28 section headers, starting at offset 0x1cd8:
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        00008134 000134 00000d 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            00008144 000144 000020 00   A  0   0  4
  [ 3] .hash             HASH            00008164 000164 00003c 04   A  4   0  4
  [ 4] .dynsym           DYNSYM          000081a0 0001a0 0000a0 10   A  5   1  4
  [ 5] .dynstr           STRTAB          00008240 000240 000098 00   A  0   0  1
  [ 6] .gnu.version      VERSYM          000082d8 0002d8 000014 02   A  4   0  2
  [ 7] .gnu.version_r    VERNEED         000082ec 0002ec 000020 00   A  5   1  4
  [ 8] .rela.dyn         RELA            0000830c 00030c 00009c 0c   A  4   0  4
  [ 9] .init             PROGBITS        000083b0 0003b0 000030 00  AX  0   0 16
```

```
  [10] .text             PROGBITS        000083e0 0003e0 000282 00  AX  0   0 16
```

根据程序反汇编编插 uprobe 桩

```
echo 'p:enter_current_state_one /root/test:0x518 arg0=%a0 lr=%lr' >> /sys/kernel/de
bug/tracing/uprobe_events
echo 'r:exit_current_state_one /root/test:0x518 arg0=%a0' >> /sys/kernel/debug/trac
ing/uprobe_events
echo 'p:enter_current_state_two /root/test:0x544 arg0=%a0 lr=%lr' >> /sys/kernel/de
bug/tracing/uprobe_events
echo 'r:exit_current_state_two /root/test:0x544 arg0=%a0' >> /sys/kernel/debug/trac
ing/uprobe_events
echo 1 > /sys/kernel/debug/tracing/events/uprobes/enable
cat /sys/kernel/debug/tracing/trace
```

用户态任意位置都可以设置 uprobe 桩点

Perf probe 使用:

除了 ftrace，我们还可以使用 Perf probe -x 动态跟踪点是一种可观察性功能，可实现以下功能：甚至不需要重新编译就可以插装源代码的任意行。 有了程序源代码的副本，跟踪点可以在运行时放置在任何地方，并且每个变量的值都可以转储时间执行通过跟踪点。 这是一项非常强大的技术，用于检测其他人编写的复杂系统或代码。

```
perf probe -x /lib/libc.so.6 memcpy
perf record -e probe_libc:memcpy -aR ls
perf report
```

社区有大量 Uprobe 应用案例，和文档:

http://www.brendangregg.com/blog/2015-06-28/linux-ftrace-uprobe.html

https://www.kernel.org/doc/Documentation/trace/uprobetracer.txt

https://www.kernel.org/doc/html/latest/trace/uprobetracer.html

Perf 使用说明

Perf 简介

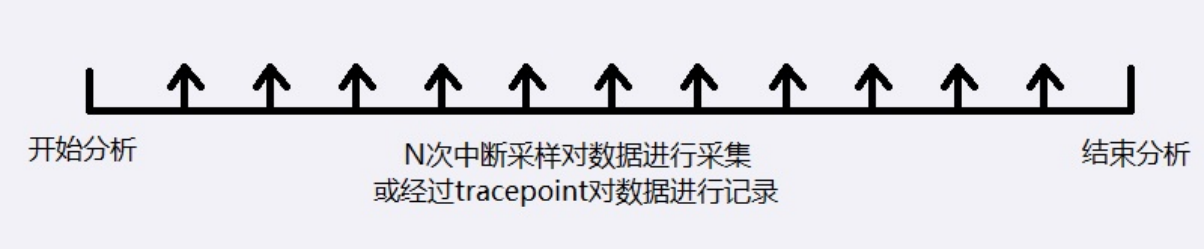
- Perf是一系列强大的性能分析工具集合。
- 在Linux 2.6.31版本引入，至今tool/perf目录拥有1万多个提交，是内核开发中最活跃的几个领域之一。
- Perf最初只负责处理系统性能事件，随着版本迭代和演进也引入了诸如probe，tracepotint，auxtrace，bpf等等各具特色的子工具。
- 通过perf可以使用一到两行命令就完成像程序热点采样，接口调用分析，阻塞分析这些以往需要插入大量分析代码才能完成的事情。
- 借助于内核日渐健全的tracepoint，perf拥有了一千多个linux内核预插桩点，可以对系统中调度，内存，文件系统，网络等各方面进行分析
- 围绕perf和系统性能事件，也有不少像perf-tool，火焰图，热点图，vtune等第三方功能扩展。
- 本说明主要针对perf stat/record/report，硬件PMU，火焰图几个部分重点进行介绍

Perf 分析流程

Perf 进行性能分析的方式通常有两种：

- 使用perf stat等命令对特定的事件计数器进行计算，并在程序结束后打印数值
- 使用perf record等命令以若干的事件为触发间隔对系统进行采样，将数据保存至perf.data文件以供后续分析

```
# perf stat ls
messages

 Performance counter stats for 'ls':

             42.83 msec task-clock                #    0.735 CPUs utilized
                 0      context-switches          #    0.000 K/sec
                 0      cpu-migrations            #    0.000 K/sec
                59      page-faults               #    0.001 M/sec
           2563283      cycles                    #    0.060 GHz
            714777      instructions              #    0.28  insn per cycle
            109487      conditionl-branch-instructions #    2.556 M/sec
             13290      conditional-branch-misspredict #   12.14% of all branches

       0.058298533 seconds time elapsed

       0.012837000 seconds user
       0.051349000 seconds sys


# perf record ls
messages       perf.data      perf.data.old
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.011 MB perf.data (260 samples) ]
#
```



Perf 事件支持

通过Perf list可以查看当前支持的所有事件包含硬件事件，软件事件，硬件cache事件，PMU事件以及预设Tracepoint事件

```
# perf list

List of pre-defined events (to be used in -e):

  L1-icache-access                                   [Hardware event]
  L1-icache-misses                                   [Hardware event]
  conditional-branch-instructions                    [Hardware event]
  conditional-branch-mispredict                      [Hardware event]
  cpu-cycles OR cycles                               [Hardware event]
  d-utlb-misses                                      [Hardware event]
  i-utlb-misses                                      [Hardware event]
  indirect-branch-instructions                       [Hardware event]
  indirect-branch-mispredict                         [Hardware event]
  instructions                                       [Hardware event]
  jtlb-misses                                        [Hardware event]
  lsu-cross-4k-stall-counter                         [Hardware event]
  lsu-other-stall-counter                            [Hardware event]
  lsu-speculation-fail                               [Hardware event]
  lsu-sq-data-discard-counter                        [Hardware event]
  lsu-sq-discard-counter                             [Hardware event]
  rf-instruction-counter                             [Hardware event]
  rf-launch-fail-counter                             [Hardware event]
  rf-reg-launch-fail-counter                         [Hardware event]
  store-instructions                                 [Hardware event]

  alignment-faults                                   [Software event]
  bpf-output                                         [Software event]
  context-switches OR cs                             [Software event]
  cpu-clock                                          [Software event]
  cpu-migrations OR migrations                       [Software event]
  dummy                                              [Software event]
  emulation-faults                                   [Software event]
  major-faults                                       [Software event]
  minor-faults                                       [Software event]
  page-faults OR faults                              [Software event]
  task-clock                                         [Software event]

  L1-dcache-load-misses                              [Hardware cache event]
  L1-dcache-loads                                    [Hardware cache event]
  L1-dcache-store-misses                             [Hardware cache event]
  L1-dcache-stores                                   [Hardware cache event]
  LLC-load-misses                                    [Hardware cache event]
```

使用 Perf 的准备工作

T-HEAD 平台支持通过buildroot来快速完成整个Linux系统的搭建，所以要在T-HEAD平台上体验perf相关功能需要通过 https://github.com/c-sky/buildroot/releases 获取最新的buildroot使用源代码编译或使用Quick Start中的命令下载预编译的镜像直接执行，具体可参照 buildroot用户手册，perf功能在 T-HEAD配置中默认开启，启动后可以输入perf命令确认环境

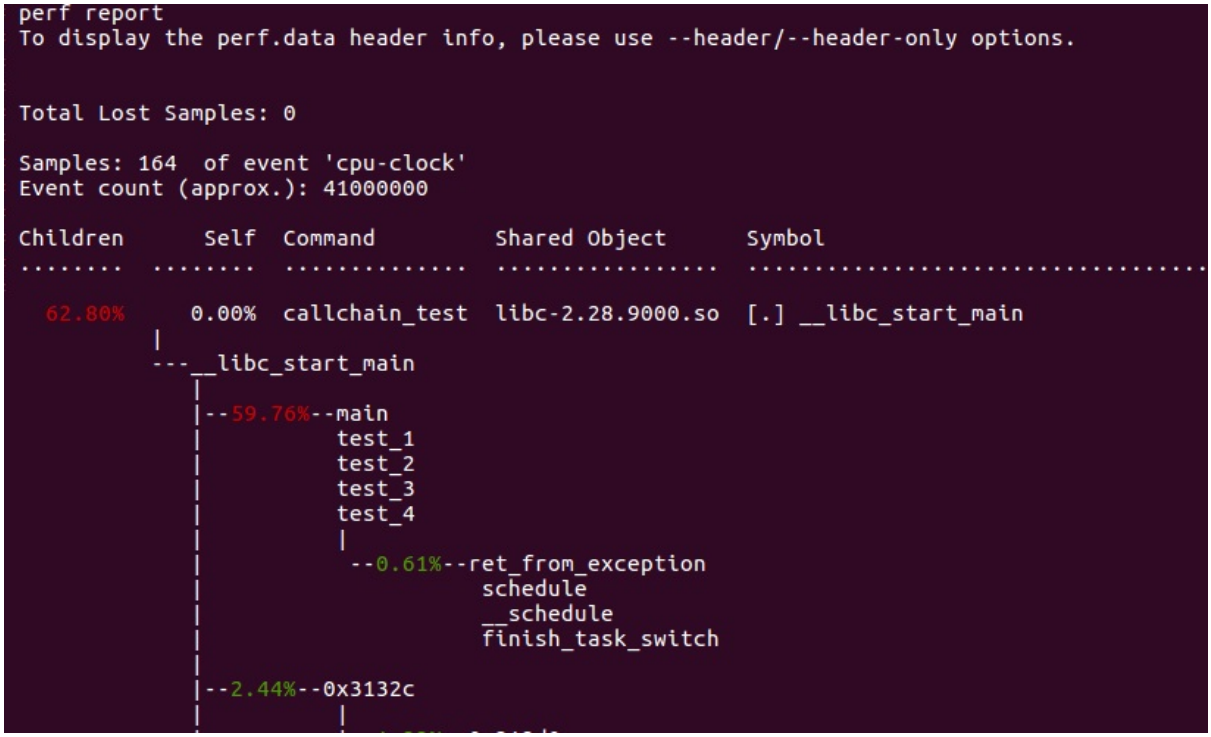
```
Starting network: OK
processor       : 0
C-SKY CPU model : ck810f
product info[0] : 0x0504000c
product info[1] : 0x10000000
product info[2] : 0x20000000
product info[3] : 0x30000000
hint (CPU funcs): 0x00000000
ccr  (L1C & MMU): 0x00000001
ccr2 (L2C)      : 0x00000000

arch-version : e68b1635ac2711d5fc939bce66318aa118c9484a

Skip the ci test

Welcome to Buildroot
buildroot login: root
# perf

 usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]

 The most commonly used perf commands are:
   annotate        Read perf.data (created by perf record) and display annotated code
   archive         Create archive with object files with build-ids found in perf.data file
   bench           General framework for benchmark suites
   buildid-cache   Manage build-id cache.
```

使用火焰图分析函数热点

以callchain_test为例如，这里将介绍如何使用perf命令和火焰图分析一个程序的函数热点

- 首先运行: perf record -g callchain_test 对 callchain_test程序的热点和调用栈进行采样
- 运行perf report可以直接观测函数热点，如果第一步中未使能 -g 选项则只显示热点不显示函数调用关系，可以看到右图中热点集中在 test_4 函数，函数调用入口为 test_1
- 当函数热点较为分散时可以通过火焰图更直观的看到函数调用关系

```
perf report
To display the perf.data header info, please use --header/--header-only options.


Total Lost Samples: 0

Samples: 154  of event 'cpu-clock'
Event count (approx.): 38500000

Overhead  Command          Shared Object      Symbol
........  ...............  .................  ....................................

  55.19%  callchain_test   callchain_test     [.] test_4
   1.95%  callchain_test   [kernel.kallsyms]  [k] __softirqentry_text_start
   1.95%  callchain_test   [kernel.kallsyms]  [k] flush_tlb_one
   1.95%  callchain_test   [kernel.kallsyms]  [k] flush_tlb_range
   1.30%  callchain_test   [kernel.kallsyms]  [k] flush_tlb_mm
   1.30%  callchain_test   [kernel.kallsyms]  [k] free_hot_cold_page
   1.30%  callchain_test   [kernel.kallsyms]  [k] page_add_file_rmap
   1.30%  callchain_test   [kernel.kallsyms]  [k] path_openat
   1.30%  callchain_test   [kernel.kallsyms]  [k] sys_mmap_pgoff
   1.30%  callchain_test   ld-2.28.9000.so    [.] $t
   1.30%  callchain_test   libc-2.28.9000.so  [.] __cxa_atexit
   1.30%  perf             [kernel.kallsyms]  [k] perf_event_exec
   0.65%  callchain_test   [kernel.kallsyms]  [k] __d_lookup_rcu
   0.65%  callchain_test   [kernel.kallsyms]  [k] __fdget_raw
   0.65%  callchain_test   [kernel.kallsyms]  [k] __fput
   0.65%  callchain_test   [kernel.kallsyms]  [k] __kunmap_atomic
```


perf report
To display the perf.data header info, please use --header/--header-only options.
Total Lost Samples: 0
Samples: 164 of event 'cpu-clock'
Event count (approx.): 41000000
Children Self Command Shared Object Symbol
62.80% 0.00% callchain_test libc-2.28.9000.so [.] __libc_start_main
__libc_start_main
59.76% main
test_1
test_2
test_3
test_4
0.61% ret_from_exception
schedule
__schedule
finish_task_switch
2.44% 0x3132c

The Kernel Address Sanitizer (KASAN)

KernelAddressSANitizer（KASAN）是动态内存错误检测器。它提供了一种快速而全面的解决方案，以查找无用后使用和越界错误。

KASAN使用编译时工具来检查每个内存访问，因此您将需要GCC 4.9.2或更高版本。需要GCC 5.0或更高版本才能检测对堆栈或全局变量的越界访问。

使用方法

在内核中开启 KASAN:

```
CONFIG_KASAN = y
```

并在CONFIG_KASAN_OUTLINE和CONFIG_KASAN_INLINE之间选择。 Outline和inline是编译器检测类型。 前者产生较小的二进制文件，而后者则快1.1-2倍。 内联检测需要GCC 5.0或更高版本。

KASAN可与SLUB和SLAB内存分配器一起使用。 为了更好地检测错误和更好地报告，请启用CONFIG_STACKTRACE。

要禁用特定文件或目录的检测，请在相应的内核Makefile中添加类似于以下内容的行：

- 对于单个文件 (譬如: main.o)

```
KASAN_SANITIZE_main.o := n
```

- 对于一个目录中的所有文件

```
KASAN_SANITIZE := n
```

出错报告

典型的越界访问报告如下所示：

```
==================================================================
BUG: AddressSanitizer: out of bounds access in kmalloc_oob_right+0x65/0x75 [test_ka
san] at addr ffff8800693bc5d3
Write of size 1 by task modprobe/1689
==============================================================================
BUG kmalloc-128 (Not tainted): kasan error
------------------------------------------------------------------------------

Disabling lock debugging due to kernel taint
INFO: Allocated in kmalloc_oob_right+0x3d/0x75 [test_kasan] age=0 cpu=0 pid=1689
 __slab_alloc+0x4b4/0x4f0
```

```
 kmem_cache_alloc_trace+0x10b/0x190
 kmalloc_oob_right+0x3d/0x75 [test_kasan]
 init_module+0x9/0x47 [test_kasan]
 do_one_initcall+0x99/0x200
 load_module+0x2cb3/0x3b20
 SyS_finit_module+0x76/0x80
 system_call_fastpath+0x12/0x17
INFO: Slab 0xffffea0001a4ef00 objects=17 used=7 fp=0xffff8800693bd728 flags=0x10000
0000004080
INFO: Object 0xffff8800693bc558 @offset=1368 fp=0xffff8800693bc720

Bytes b4 ffff8800693bc548: 00 00 00 00 00 00 00 00 5a 5a 5a 5a 5a 5a 5a 5a  .......
.ZZZZZZZZ
Object ffff8800693bc558: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkk
kkkkkkk
Object ffff8800693bc568: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkk
kkkkkkk
Object ffff8800693bc578: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkk
kkkkkkk
Object ffff8800693bc588: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkk
kkkkkkk
Object ffff8800693bc598: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkk
kkkkkkk
Object ffff8800693bc5a8: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkk
kkkkkkk
Object ffff8800693bc5b8: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b  kkkkkkkkk
kkkkkkk
Object ffff8800693bc5c8: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b a5  kkkkkkkkk
kkkkkk.
Redzone ffff8800693bc5d8: cc cc cc cc cc cc cc cc                          ........
Padding ffff8800693bc718: 5a 5a 5a 5a 5a 5a 5a 5a                          ZZZZZZZZ
CPU: 0 PID: 1689 Comm: modprobe Tainted: G    B          3.18.0-rc1-mm1+ #98
 ffff8800693bc000 0000000000000000 ffff8800693bc558 ffff88006923bb78
 ffffffff81cc68ae 00000000000000f3 ffff88006d407600 ffff88006923bba8
 ffffffff811fd848 ffff88006d407600 ffffea0001a4ef00 ffff8800693bc558
Call Trace:
 [<ffffffff81cc68ae>] dump_stack+0x46/0x58
 [<ffffffff811fd848>] print_trailer+0xf8/0x160
 [<ffffffffa00026a7>] ? kmem_cache_oob+0xc3/0xc3 [test_kasan]
 [<ffffffff811ff0f5>] object_err+0x35/0x40
 [<ffffffffa0002065>] ? kmalloc_oob_right+0x65/0x75 [test_kasan]
 [<ffffffff8120b9fa>] kasan_report_error+0x38a/0x3f0
 [<ffffffff8120a79f>] ? kasan_poison_shadow+0x2f/0x40
 [<ffffffff8120b344>] ? kasan_unpoison_shadow+0x14/0x40
 [<ffffffff8120a79f>] ? kasan_poison_shadow+0x2f/0x40
 [<ffffffffa00026a7>] ? kmem_cache_oob+0xc3/0xc3 [test_kasan]
 [<ffffffff8120a995>] __asan_store1+0x75/0xb0
 [<ffffffffa0002601>] ? kmem_cache_oob+0x1d/0xc3 [test_kasan]
 [<ffffffffa0002065>] ? kmalloc_oob_right+0x65/0x75 [test_kasan]
 [<ffffffffa0002065>] kmalloc_oob_right+0x65/0x75 [test_kasan]
 [<ffffffffa00026b0>] init_module+0x9/0x47 [test_kasan]
```

```
 [<ffffffff810002d9>] do_one_initcall+0x99/0x200
 [<ffffffff811e4e5c>] ? __vunmap+0xec/0x160
 [<ffffffff81114f63>] load_module+0x2cb3/0x3b20
 [<ffffffff8110fd70>] ? m_show+0x240/0x240
 [<ffffffff81115f06>] SyS_finit_module+0x76/0x80
 [<ffffffff81cd3129>] system_call_fastpath+0x12/0x17
Memory state around the buggy address:
 ffff8800693bc300: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
 ffff8800693bc380: fc fc 00 00 00 00 00 00 00 00 00 00 00 00 00 fc
 ffff8800693bc400: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
 ffff8800693bc480: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
 ffff8800693bc500: fc fc fc fc fc fc fc fc fc fc fc 00 00 00 00 00
>ffff8800693bc580: 00 00 00 00 00 00 00 00 00 00 03 fc fc fc fc fc
                                             ^
 ffff8800693bc600: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
 ffff8800693bc680: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
 ffff8800693bc700: fc fc fc fc fb fb fb fb fb fb fb fb fb fb fb fb
 ffff8800693bc780: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
 ffff8800693bc800: fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb fb
==================================================================
```

该报告的标题描述了发生了哪种错误以及导致这种错误的访问方式。后面是对访问的slub对象的描述（有关详细信息，请参见 Documentation/vm/slub.txt 中的 “ SLUB调试输出” 部分）以及对访问的内存页面的描述。

在最后一部分中，报告显示访问地址周围的内存状态。阅读此部分需要对KASAN的工作方式有一些了解。

存储器的每8个对齐字节的状态被编码在一个影子字节中。这8个字节可以访问，部分访问，释放或为redzone。对于每个影子字节，我们使用以下编码：0表示相应存储区的所有8个字节均可访问； N（1 <= N <= 7）表示前N个字节是可访问的，而其他（8-N）个字节则不可访问；任何负值表示整个8字节字不可访问。我们使用不同的负值来区分不同类型的不可访问的内存，例如Redzone或释放的内存（请参见mm / kasan / kasan.h）。

在报告上方的箭头指向影子字节03，这意味着被访问的地址可以部分访问。

本文参考： https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html

KGDB 内核调试指南

内核有两个不同的调试器前端（kdb和kgdb），它们连接到调试核心。如果在编译和运行时正确配置了内核，则可以使用两个调试器前端中的任何一个并在它们之间动态转换。

Kdb是简单化的shell风格的界面，可以在带有键盘或串行控制台的系统控制台上使用。您可以使用它来检查内存，寄存器，进程列表，dmesg，甚至设置断点以停在特定位置。尽管您可以设置断点并执行一些基本的内核运行控制，但Kdb并不是源代码级调试器。 Kdb主要旨在进行一些分析，以帮助开发或诊断内核问题。如果代码是使用CONFIG_KALLSYMS构建的，则可以按名称访问内核内置文件或内核模块中的某些符号。

Kgdb旨在用作Linux内核的源代码级调试器。它与gdb一起用于调试Linux内核。期望gdb可用于“侵入”内核以检查内存，变量并查看调用堆栈信息，类似于应用程序开发人员使用gdb调试应用程序的方式。可以在内核代码中放置断点并执行一些有限的执行步骤。

使用kgdb需要两台机器。其中一台机器是开发机器，另一台是目标机器。要调试的内核在目标计算机上运行。开发机器针对包含符号的vmlinux文件运行gdb实例（不是启动映像，如bzImage，zImage，uImage…）。在gdb中，开发人员指定连接参数并连接到kgdb。开发人员与gdb建立的连接类型取决于在测试计算机内核中编译为内置模块或可加载内核模块的kgdb I / O模块的可用性。

编译内核

在 menuconfig 中开启 CONFIG_KGDB

```
| Symbol: KGDB [=y]
                                                         |
| Type  : bool
                                                         |
| Defined at lib/Kconfig.kgdb:11
                                                         |
|   Prompt: KGDB: kernel debugger
```

```
|   Depends on: HAVE_ARCH_KGDB [=y] && DEBUG_KERNEL [=y]

                                                                |
|   Location:

                                                                |
|     -> Kernel hacking

| (1)   -> Generic Kernel Debugging Instruments
```

LOCKDEP

简介

验证程序操作的基本对象是锁的类型。

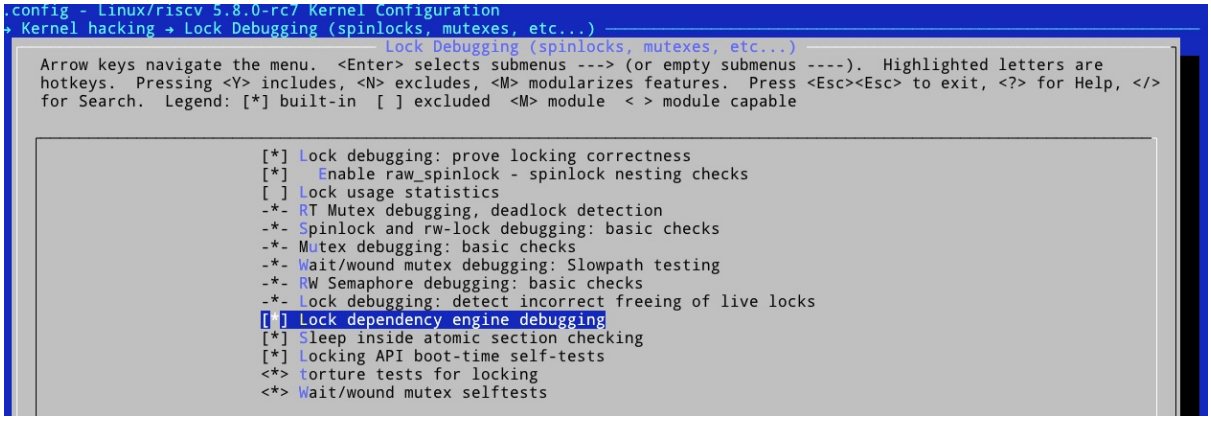
一类锁是一组在锁规则上逻辑上相同的锁，即使锁可能具有多个（可能是数万个）实例化。 例如，inode结构中的锁是一个类，而每个inode具有该锁类的自己的实例化。

验证器跟踪锁类的“状态”，并跟踪不同锁类之间的依赖关系。 验证器保持状态和依赖关系正确的滚动证明。

锁实例化不同，锁类本身永远不会消失：在启动后首次使用锁类时，它将被注册，并且该锁类的所有后续使用都将附加到该锁类上

使用介绍

在 Kernel hacking -> Lock Debuging 中开启 LOCK_DEP 测试，(注意不要开启 LOCK_STAT)



运行后，会启动自动测试

```
[    0.000424] Lock dependency validator: Copyright (c) 2006 Red Har
[    0.000443] ... MAX_LOCKDEP_SUBCLASSES:  8
[    0.000460] ... MAX_LOCK_DEPTH:          48
[    0.000476] ... MAX_LOCKDEP_KEYS:        8192
[    0.000492] ... CLASSHASH_SIZE:          4096
[    0.000508] ... MAX_LOCKDEP_ENTRIES:     32768
[    0.000524] ... MAX_LOCKDEP_CHAINS:      65536
[    0.000540] ... CHAINHASH_SIZE:          32768
[    0.000556]  memory used by lock dependency info: 6301 kB
[    0.000572]  memory used for stack traces: 4224 kB
[    0.000588]  per task-struct memory footprint: 1920 bytes
[    0.000604] ------------------------
[    0.000620] | Locking API testsuite:
[    0.000636] ----------------------------------------------------
[    0.000651]                                  | spin |wlock |rloc|
```

```
[    0.000667]   --------------------------------------------------
[    0.000698]                      A-A deadlock:  ok  |  ok  |  ok|
[    0.009065]                  A-B-B-A deadlock:  ok  |  ok  |  ok|
[    0.018270]              A-B-B-C-C-A deadlock:  ok  |  ok  |  ok|
[    0.028441]              A-B-C-A-B-C deadlock:  ok  |  ok  |  ok|
[    0.038546]          A-B-B-C-C-D-D-A deadlock:  ok  |  ok  |  ok|
[    0.049592]          A-B-C-D-B-D-D-A deadlock:  ok  |  ok  |  ok|
[    0.060594]          A-B-C-D-B-C-D-A deadlock:  ok  |  ok  |  ok|
[    0.071571]                    double unlock:  ok  |  ok  |  ok|
[    0.079825]                 initialize held:  ok  |  ok  |  ok|
[    0.087588]   --------------------------------------------------
[    0.087606]              recursive read-lock:             |  ok|
[    0.090094]           recursive read-lock #2:             |  ok|
[    0.092472]            mixed read-write-lock:             |  ok|
[    0.094807]            mixed write-read-lock:             |  ok|
[    0.097203]   mixed read-lock/lock-write ABBA:             |  ok|
[    0.099780]    mixed read-lock/lock-read ABBA:             |FAIL|
[    0.102444]  mixed write-lock/lock-write ABBA:             |  ok|
[    0.105050]   --------------------------------------------------
[    0.105079]      hard-irqs-on + irq-safe-A/12:  ok  |  ok  |  ok|
[    0.108638]      soft-irqs-on + irq-safe-A/12:  ok  |  ok  |  ok|
[    0.112183]      hard-irqs-on + irq-safe-A/21:  ok  |  ok  |  ok|
[    0.115749]      soft-irqs-on + irq-safe-A/21:  ok  |  ok  |  ok|
[    0.119282]        sirq-safe-A => hirqs-on/12:  ok  |  ok  |  ok|
[    0.122826]        sirq-safe-A => hirqs-on/21:  ok  |  ok  |  ok|
[    0.126384]          hard-safe-A + irqs-on/12:  ok  |  ok  |  ok|
[    0.129957]          soft-safe-A + irqs-on/12:  ok  |  ok  |  ok|
[    0.133488]          hard-safe-A + irqs-on/21:  ok  |  ok  |  ok|
[    0.137056]          soft-safe-A + irqs-on/21:  ok  |  ok  |  ok|
[    0.140635]     hard-safe-A + unsafe-B #1/123:  ok  |  ok  |  ok|
[    0.144589]     soft-safe-A + unsafe-B #1/123:  ok  |  ok  |  ok|
[    0.148522]     hard-safe-A + unsafe-B #1/132:  ok  |  ok  |  ok|
[    0.152483]     soft-safe-A + unsafe-B #1/132:  ok  |  ok  |  ok|
[    0.156465]     hard-safe-A + unsafe-B #1/213:  ok  |  ok  |  ok|
[    0.160438]     soft-safe-A + unsafe-B #1/213:  ok  |  ok  |  ok|
[    0.164415]     hard-safe-A + unsafe-B #1/231:  ok  |  ok  |  ok|
[    0.168325]     soft-safe-A + unsafe-B #1/231:  ok  |  ok  |  ok|
```

核心转储

在 Linux 系统中，常将“主内存”称为核心(core)，而核心映像(core image) 就是 “进程”(process)执行当时的内存内容。当进程发生错误或收到“信号”(signal) 而终止执行时，系统会将核心映像写入一个文件，以作为调试之用，这就是所谓的核心转储(core dump)。

通常在系统收到特定的信号时由操作系统生成。信号可以由程序执行过程中的异常触发，也可以由外部程序发送。动作的结果一般是生成一个某个进程的内存转储的文件，文件包含了此进程当前的运行堆栈信息。有时程序并未经过彻底测试，这使得它在执行的时候一不小心就会找到破坏。这可能会导致核心转储（core dump）。

当在一个程序崩溃时，系统会在指定目录下生成一个core文件，我们就可以通过 core文件来对造成程序崩贵的原因进行调试定位。

开启核心转储

默认情况下，Linux 没有打开core文件生成功能，通过以下命令打开core文件的生成：

```
# 不限制产生 core 的大小
ulimit -c unlimited
```

unlimited 意思是系统不限制core文件的大小，只要有足够的磁盘空间，会转存程序所占用的全部内存，如果需要限制系统产生 core 的大小，可以使用以下命令：

```
# core 最大限制大小为 409600 字节
ulimit -c 409600
```

如果需要关装核心转储功能，只需要将限制大小设为 `0` 即可：

```
ulimit -c 0
```

上述操作在一个终端中有效，退出或者新打开终端时无效，可以在将上述配置加入到 `/etc/profile` 中：

```
# 编辑 profile 文件
vi /etc/profile

# 将下行加到入profile 文件中
ulimit -c unlimited
```

使用GDB调试 core 文件

首先我们来编写一个产生错误的程序， `vi test.c`

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  int *p = NULL;

  // 给一个NULL指令赋值，会产生 Segmentation fault 错误
  *p = 100;

  return 0;
}
```

以上程序给一个空指令赋值，这是C语言编程中遇到的段错误，程序运行后，如果开启了核心转储后，就会产生一个 core 文件。

```
# 编译 test.c 生成 test 程序
gcc test.c -o test -g -MD

# 运行该程序
./test
```

运行后，我们可以看到 `Segmentation fault (core dumped)` 提示信息，表示已经在当前目录下产生了一 core 文件，通过 `ls -l core` 命令可以查看：

```
root@thead-910:~# ls -l core
-rw------- 1 root root 225280 Jan  6 10:51 core
```

下面我样就可以通过 core 来进行调试：

```
root@thead-910:~# gdb test core
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "riscv64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test...
```

```
[New LWP 1134]
Core was generated by `./test'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x0000002ae6775600 in main (argc=1, argv=0x3fffb22d18) at test.c:9
9        *p = 100;
```

通过 GDB 可以看到程序的第9行出错。

如果我们想一步步调试，那么可以在gdb中打断点(b)，然后运行(r)，或者一步步(s)调试，

```
(gdb) b test.c:9
Breakpoint 1 at 0x2ae67755f8: file test.c, line 9.
(gdb) r
Starting program: /root/test

Breakpoint 1, main (argc=1, argv=0x3fffffce8) at test.c:9
9        *p = 100;
(gdb) p p
$1 = (int *) 0x0
(gdb) n

Program received signal SIGSEGV, Segmentation fault.
0x0000002aaaaaa600 in main (argc=1, argv=0x3fffffce8) at test.c:9
9        *p = 100;
(gdb) quit
A debugging session is active.

        Inferior 1 [process 1198] will be killed.

Quit anyway? (y or n) y
```

使用 GDB 调式 core 的命令格式如下：

```
gdb <程序> <core文件>
```

概述

本章开始介绍在 ICE-EVB 开发板上进行软件开发，开发 Linux 应用程序有两种方式，第一次是直接在 ICE-EVB 开发板上建立开发环境，直接在ICE-EVB 的Linux 环境下进行软件开发。另一种是交叉开发环境，即在PC电脑上搭建 RISC-V 的应用开发环境，在PC机上完成应用开发，再传开发出来的程序上传到ICE-EVB 的Linux 下。

安装开发环境

开发板 Native 开发环境

在ICE-EVB 的开发板上安装开发环境：

```
apt update
apt install -y build-essential autoconf automake
```

示例：编写 helloworld 程序

1. 编程示例: helloworld.c `vi helloworld.c`

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    printf("Hello world!\n");

    return EXIT_SUCCESS;
}
```

2. 编译、执行：

```
# 编译
gcc helloworld.c -o helloworld

# 执行
./helloworld
Hello world!
```

交叉开发环境

方法一：使用 thead 工具安装

通过 thead 命令下载并安装 RISC-V 工具链，如果未安装 thead 工具，可通过 `sudo pip install thead_tools` 安装，thead 命令安装成功后，可通过如下命令安装工具链：

```
thead toolchain -r
```

安装成功后，控制台显示信息如下：

```
Start to download toolchain: riscv64-linux
 100.00% [##################################################] Speed: 3.603MB/S
Start install, wait half a minute please.
Congratulations!
please run command:
  source /Users/zhuzhiguo/.zshrc
```

安装成功后，将 risc-v 工具链路径加入到 PATH环境变量中：

```
export PATH=$HOME/.thead/riscv64-linux/bin:$PATH
```

方法二：通过 tar 包安装

```
wget "http://yoctools.oss-cn-beijing.aliyuncs.com/\
riscv64-linux-x86_64-20201104.tar.bz2"

sudo mkdir -p /opt/riscv64-linux
sudo tar xf riscv64-linux-x86_64-20201104.tar.gz -C /opt/riscv64-linux

# 将 toolchain 加入到 PATH 环境变量中
export PATH=/opt/riscv64-linux/bin:$PATH
```

交叉开发示例：编写 helloworld 程序

1. 在 PC 机编写代码: `vim helloworld.c`

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    printf("Hello world!\n");

    return EXIT_SUCCESS;
}
```

2. 编译与安装

```
# 在PC 机上完成编译
riscv64-linux-gnu-gcc helloworld.c -o helloworld

# 将程序安装到 ICE-DVB 开发板
scp helloworld root@192.168.1.100:~/
```

3. 在 ICE-EVB 上运行 helloworld 程序:

```
ssh root@192.168.1.100

./helloworld
Hello world!
```

Linux 下的 GPIO 编程

GPIO(General-purpose input/output) 通用型之输入输出的简称。在嵌入式系统中，经常需要控制许多结构简单的外部设备或者电路，这些设备有的需要通过CPU控制，有的需要CPU提供输入信号。并且，许多设备或电路只要求有开/关两种状态就够了，比如LED的亮与灭。对这些设备的控制，使用传统的串口或者并口就显得比较复杂，所以，在嵌入式微处理器上通常提供了一种“通用可编程I/O端口”，也就是GPIO。

Linux 下的GPIO设备

GPIO 设备

通过 sysfs 方式控制 GPIO，先访问 `/sys/class/gpio` 目录，向 export 文件写入 GPIO 编号，使得该 GPIO 的操作接口从内核空间暴露到用户空间，GPIO 的操作接口包括 direction 和 value 等，direction 控制 GPIO 方向，而 value 可控制 GPIO 输出或获得 GPIO 输入。文件 IO 方式操作 GPIO，使用到了4个函数 open、close、read、write。通过：`ls -l /sys/class/gpio` 可能以查看 GPIO的信息：

```
root@thead-910:~# ls -l /sys/class/gpio
total 0
--w------- 1 root root 4096 Jan  6 13:33 export
lrwxrwxrwx 1 root root    0 Dec 26 20:33 gpiochip352 -> ../../devices/platform/soc/
3fff72000.gpio/gpio/gpiochip352
lrwxrwxrwx 1 root root    0 Dec 26 20:33 gpiochip384 -> ../../devices/platform/soc/
3fff72000.gpio/gpio/gpiochip384
lrwxrwxrwx 1 root root    0 Dec 26 20:33 gpiochip416 -> ../../devices/platform/soc/
3fff72000.gpio/gpio/gpiochip416
lrwxrwxrwx 1 root root    0 Dec 26 20:33 gpiochip448 -> ../../devices/platform/soc/
3fff71000.gpio/gpio/gpiochip448
lrwxrwxrwx 1 root root    0 Dec 26 20:33 gpiochip480 -> ../../devices/platform/soc/
3fff71000.gpio/gpio/gpiochip480
--w------- 1 root root 4096 Dec 26 20:33 unexport
```

文件说明：

1. gpio_operation 通过 /sys/ 文件接口操作 IO 端口 GPIO 到文件系统的映射
2. 控制 GPIO 的目录位于 /sys/class/gpio
3. /sys/class/gpio/export 文件用于通知系统需要导出控制的 GPIO 引脚编号
4. /sys/class/gpio/unexport 用于通知系统取消导出
5. /sys/class/gpio/gpiochipXXX 目录保存系统中 GPIO 寄存器的信息，包括每个寄存器控制引脚的起始编号 base，寄存器名称，引脚总数

GPIO 导出

需要对一个GPIO 控制，首先需要对 GPIO进行导出，操作步骤如下：

第一步：确定GPIO 引脚编号

```
引脚编号 = 控制引脚的寄存器基数 + 控制引脚寄存器位数
```

例如：（具体 GPIO 需要参考数据手册），如果使想用 GPIO1_20，那么引脚编号就可能等于 1 x 32 + 20 = 54

第二步：导出引脚

向 /sys/class/gpio/export 写入此编号，比如20号引脚，在 shell 中可以通过以下命令实现：

```
# 导出20号GPIO
echo 20 > /sys/class/gpio/export

# 查看导出是否成功
ls /sys/class/gpio/gpio20
```

命令成功后生成 /sys/class/gpio/gpio20 目录，如果没有出现相应的目录，说明此引脚不可导出。

第三步：设置引导方向

通过 GPIO 的 direction文件，修改GPIO引脚的方向：

```
echo out > /sys/class/gpio/gpio20/direction
```

direction 接受的参数可以是：in、out、high、low。其中参数 high / low 在设置方向为输出的同时，将 value 设置为相应的 1 / 0

第四步：设置或者查看 GPIO电平

GPIO 的 value 文件是端口的数值，为1或0：

```
# 设置 GPIO 电平
echo 1 > /sys/class/gpio/gpio20/value

# 查看 GPIO 电平
cat /sys/class/gpio/gpio44/value
```

第五步：取消GPIO 导出

```
echo 20 > /sys/class/gpio/unexport
```

编程示例

```
#include <stdlib.h>
```

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>   //define O_WRONLY and O_RDONLY

static int write_file(const char *filename, const char *value)
{
    int fd = open(filename, O_WRONLY);
    if(fd != 0) {
        write(fd, value,strlen(value));
        close(fd);
        return 0;
    }

    return EXIT_FAILURE;
}

// 打开 GPIO
static int gpio_open(char *id)
{
    return write_file("/sys/class/gpio/export", id);
}

// 设置 GPIO 方向
static int gpio_direction(char *id, char *dir)
{
    char filename[128];
    sprintf(filename, "/sys/class/gpio/gpio%s/direction", id);

    return write_file(filename, dir);
}

// 设置 GPIO 电平
static int gpio_set(char *id, char *dir)
{
    char filename[128];
    sprintf(filename, "/sys/class/gpio/gpio%s/value", id);

    return write_file(filename, dir);
}

#define GPIO_PIN_48      "48"
int main(int argc, char **argv)
{
    if (gpio_open(GPIO_PIN_48) == 0) {
        gpio_direction(GPIO_PIN_48, "OUT");

        while(1) {
            gpio_set(GPIO_PIN_48,"1");
            usleep(1000000);
            gpio_set(GPIO_PIN_48,"0");
```

```
            usleep(1000000);
        }
    }

    return 0;
}
```

可以使用SHELL 脚本对GPIO的控制：

```
#!/bin/bash

echo 40 > /sys/class/gpio/export
echo out > /sys/class/gpio/gpio40/direction
while :
do
    echo 1 > /sys/class/gpio/gpio40/value
    usleep 1000
    echo 0 > /sys/class/gpio/gpio40/value
  usleep 1000
done
```

Linux 下的 I2C 编程

I2C 简介

IIC（Inter Integrated Circuit）总线是 PHILIPS 公司开发的一种半双工、双向二线制同步串行总线。IIC 总线包含两根信号线，双向数据线 SDA，时钟线 SCL）。

它的物理层有如下特点：

- 它是一个支持多设备的总线。“总线”指多个设备共用的信号线。 在一个I2C通讯总线中，可连接多个I2C通讯设备，支持多个通讯主机及多个通讯从机。
- 一个I2C总线只使用两条总线线路，一条双向串行数据线(SDA) ，一条串行时钟线 (SCL)。 数据线即用来表示数据，时钟线用于数据收发同步。
- 每个连接到总线的设备都有一个独立的设备地址，主机可以利用这个地址进行不同设备之间的访问。 其中地址是一个七位或十位的数字。
- 总线通过上拉电阻接到电源。当I2C设备空闲时，会输出高阻态，而当所有设备都空闲， 都输出高阻态时，由上拉电阻把总线拉成高电平。
- 多个主机同时使用总线时，为了防止数据冲突，会利用仲裁方式决定由哪个设备占用总线。
- 具有三种传输模式：标准模式传输速率为100kbit/s ，快速模式为400kbit/s ， 高速模式下可达3.4Mbit/s，但目前大多I2C设备尚不支持高速模式。

i2c-tools

Linux 系统提供 i2c-tools 工具来对系统中的 I2C 总线进行调试，下面重要介绍 i2c-tools 的使用方法。

安装i2c-tools

在开发板的Linux 命令环境中通过下面命令安装：

```
apt install i2c-tools
```

i2c-tools 提供 `i2cdetect` 、 `i2cdump` 、 `i2ctransfer` 、 `i2cset` 、 `i2cget` 命令，来操作I2C。

i2cdetect

- 显示所有可用的I2C总线： `i2cdetect -l`

  ```
  i2c-0   i2c             Synopsys DesignWare I2C adapter         I2C adapter
  ```

- I2C设备查询: `i2cdetect -y 0`

  ```
  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
  00:          -- -- -- -- -- -- -- -- -- -- -- -- --
  ```

```
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: 50 -- -- -- -- -- -- 57 -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

i2ctransfer

- 写：`i2ctransfer -f -y 0 w3@0x36 0x50 0x81 0x01`
- 读：`i2ctransfer -f -y 0 w2@0x36 0x30 0x0A r3`

i2cdump

通过i2cdump指令可导出I2C设备中的所有寄存器内容，例如输入i2cdump -y 0 0x50，可获得以下内容:

```
No size specified (using byte-data access)
0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: 01 00 0d 00 84 08 e8 03 04 4a 00 00 00 00 00 00    ?.?.?????J......
10: 00 00 35 02 36 39 39 2d 38 32 31 38 30 2d 31 30    ..5?699-82180-10
20: 30 30 2d 34 31 30 20 4a 2e 30 ff ff ff ff ff ff    00-410 J.0......
30: ff ff 35 2d 66 4b 04 00 36 2d 66 4b 04 00 00 00    ..5-fK?.6-fK?...
40: 00 00 00 00 37 2d 66 4b 04 00 30 33 32 33 32 31    ....7-fK?.032321
50: 36 31 33 30 35 36 35 ff ff ff ff ff ff ff ff ff    6130565.........
60: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
70: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
80: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
90: ff ff ff ff ff ff 4e 56 43 42 1c 00 4d 31 00 00    ......NVCB?.M1..
a0: 35 2d 66 4b 04 00 36 2d 66 4b 04 00 37 2d 66 4b    5-fK?.6-fK?.7-fK
b0: 04 00 ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ?...............
c0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
d0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
e0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff    ................
f0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff 96    ...............?
```

```
i2cdump -y 0 0x50指令中，
    -y        代表取消用户交互过程，直接执行指令；
    0         代表I2C总线编号；
    0x50      代表I2C设备从机地址，此处选择配置芯片的高256字节内容。
```