SOPHON BSP 开发参考手册

发布 1.0.0

SOPHGO

2024 年 06 月 24 日

目录

CHAPTER 1

声明

法律声明

注意

您购买的产品、服务或特性等应受算能商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，算能对本文档内容不做任何明示或默示的声明或保证。由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

技术支持

地址 北京市海淀区丰豪东路 9 号院中关村集成电路设计园（ICPARK）1 号楼

邮编 100094

网址 https://www.sophgo.com/

邮箱 sales@sophgo.com

电话 +86-10-57590723 +86-10-57590724

CHAPTER 2

前言

2.1 文档概述

本文档详细介绍了 BM1684X 系列智算模组（含开发板）的外观特点、应用场景、设备参数、电气特性、配套软件、使用环境等，使得该设备的用户及开发者对 BM1684X 系列智算模组（含开发板）有比较全面深入的了解。设备用户及开发者可依据此手册，开展对该设备的安装、调试、部署、维护等一系列工作。

2.2 读者对象

本文档主要适用于如下人员：

- 算丰 FAE 工程师、售前工程师
- 生态合作伙伴的开发人员
- 用户企业研发工程师、售前工程师

2.3 约定的符号、标志、专用语解释

在本文中可能出现如下符号、标志，它们所代表的含义如下：

 危险	表示有高度危险，如果不能避免，可能导致人员伤亡或严重伤害
 警告	表示有中度或低度潜在危险，如果不能避免，可能导致人员轻微或中等伤害
 注意	表示有潜在风险，如果忽视这部分文本，可能导致设备损坏、数据丢失、设备性能降低或不可预知的结果
 防静电	防静电标识，表示静电敏感的设备或操作
 当心触电	电击防护标识，标识高压危险，需做好防护
 窍门	表示能帮助您解决某个问题或节省您的时间
 说明	表示是正文的附加信息，是对正文的强调和补充

2.4 缩略语

JPU	JPEG Process Unit	JPEG 处理单元
VPP	Video Post Process	图像后处理
VPU	Video Process Unit	视频编解码单元

2.5 修改记录

文档版本	发布日期	修订说明	对应硬件版本	对应软件版本
V0.1	2022-07-13	首次正式发布	BM1684X EVB:V1.1	0.2.3

2.6 声明

本产品系列将有不断地更新迭代，我们将定期检查本产品手册中的内容，在后续的版本中将出现不可避免的修正、删减、补充。

我们保留在不事先通知的情况下进行技术改进、文档变更、产品改进升级、增加或减少产品型号和功能的权利。

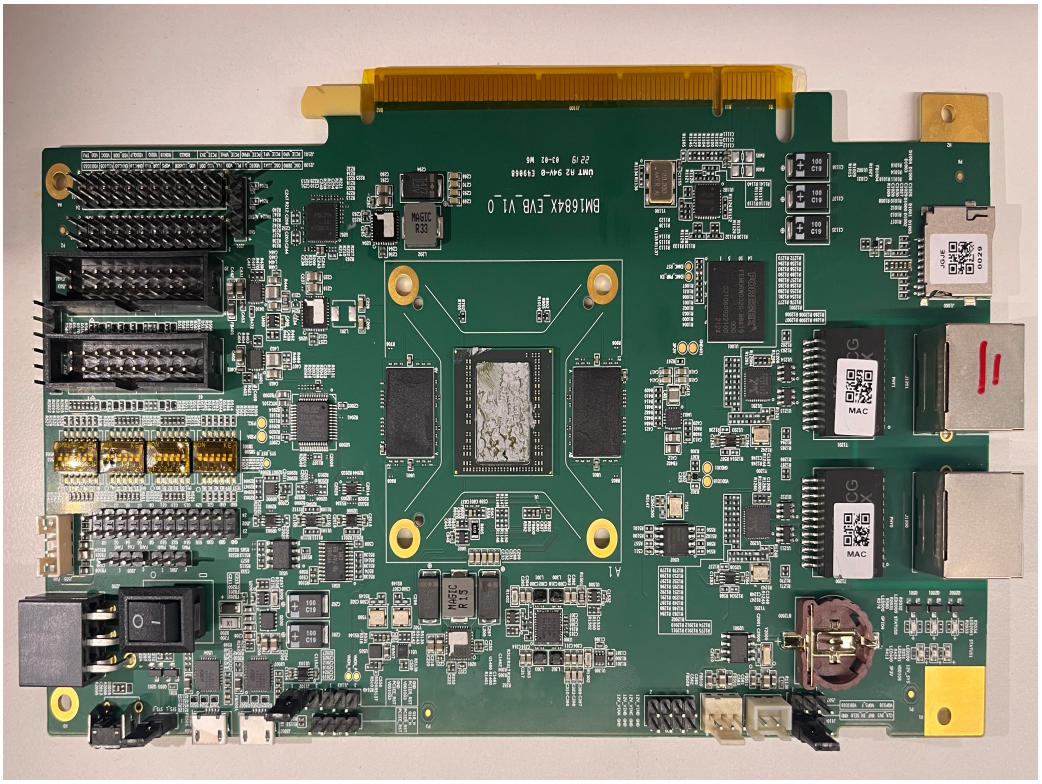
CHAPTER 3

硬件安装

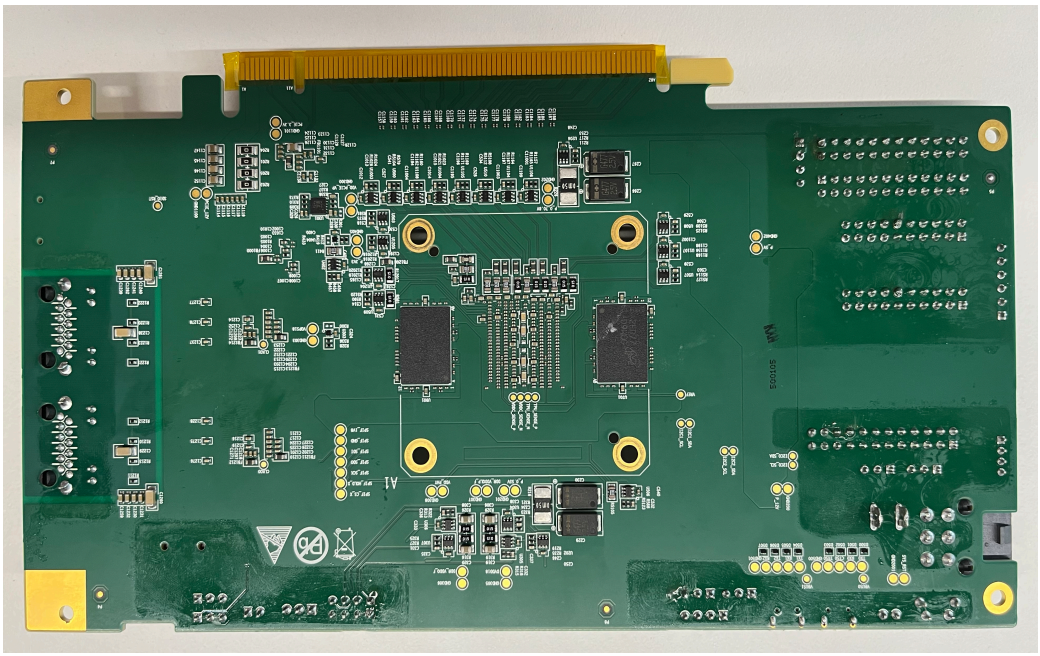
3.1 板卡安装

BM1684X 智算模组仅指包括 BM1684X、LPDDR4X、eMMC 等核心组件的板卡，如下图：

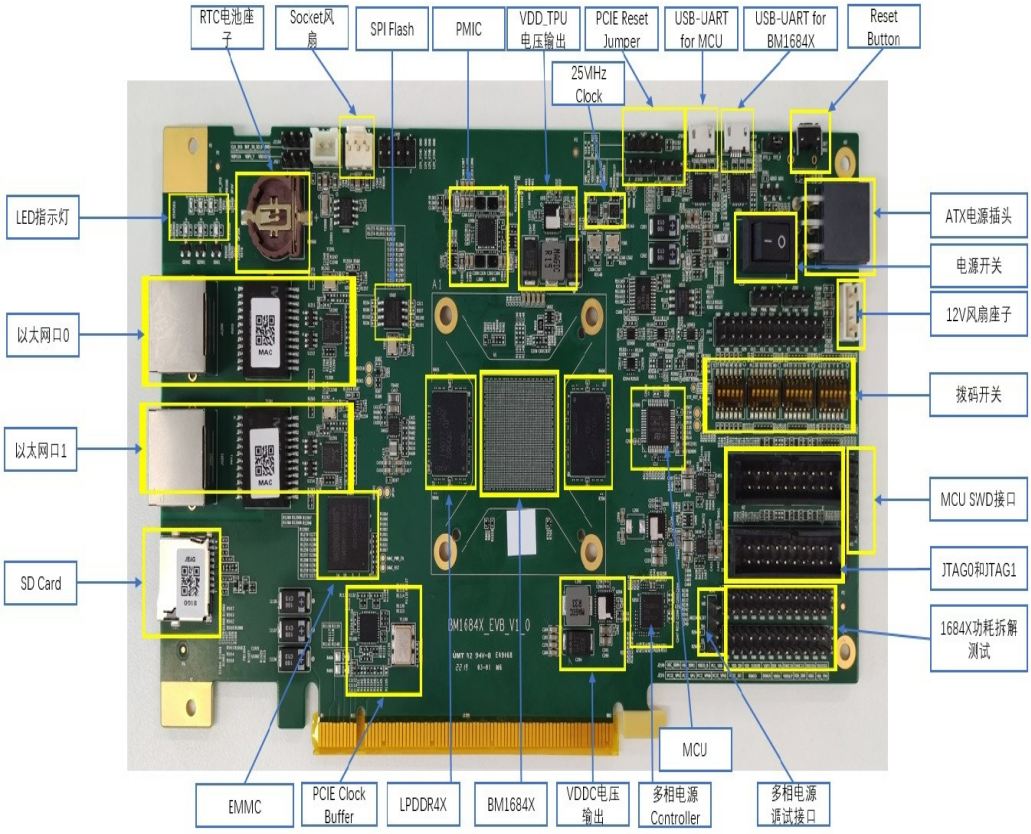
BM1684X EVB 正面



BM1684X EVB 背面



各部件位置指示图如下:



为方便后面的描述，下文以“核心板”指代这块板卡。

3.2 附件安装

为方便调试，建议您准备如下附件：

a. USB 转 UART 线缆一条：核心板引出的 UART0(UART for BM1684X) 为调试口，TTL 电平，波特率 115200，8 比特数据，1 比特停止位，无奇偶校验，无硬件流控。

b. 以太网线缆一条：接以太网口 0（eth0），预装系统默认设置为 DHCP，所以将 BM1684X 通过 eth0 和您的调试机都部署在同一路由器下比较方便。

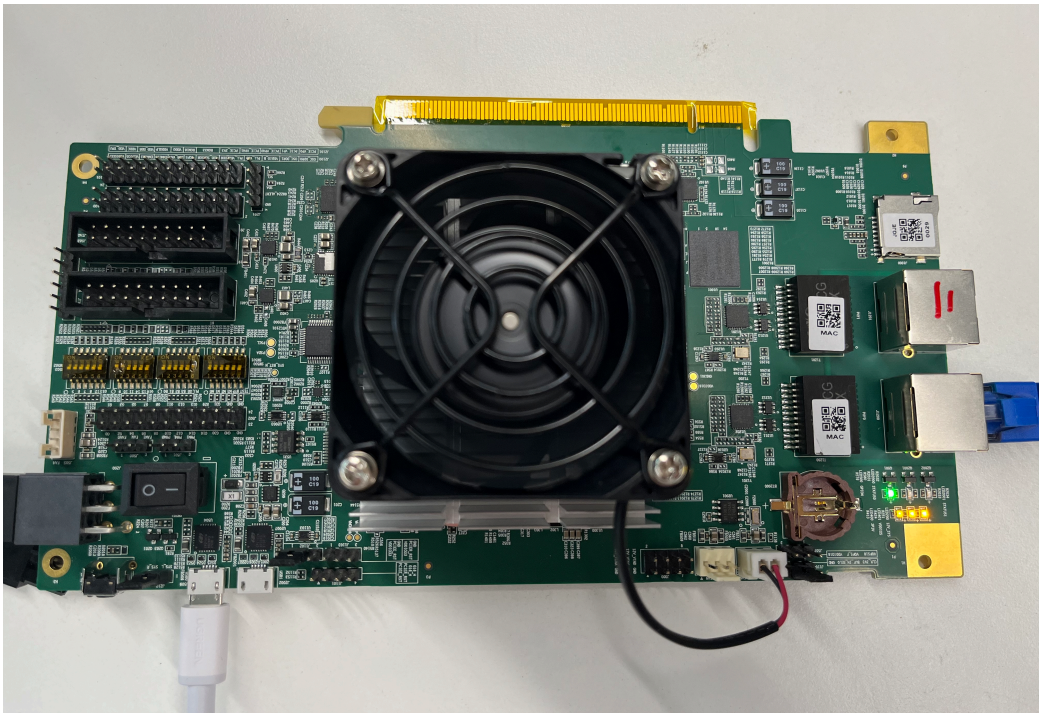
c. SD 卡一张：刷机或调试时使用，建议 8GB/class10 或更高规格。

d.  与您的底板设计相匹配的电源：如果您使用我们提供的参考底板，配套直流电源输出为 12V/10A，5.5*2.1mm 母头，中心为正极。

e. 散热：请安装散热片、风扇等必要的散热设备，以免出现过热关机等异常状况。

3.3 上电开机

一切就绪后，您就可以为底板加电了，如果您使用我们提供的参考底板，请先插上电源，然后拨电源键（此时从串口终端应该也应可以看到 log 打印了），指示灯正常状态如下：



请检查您的串口终端，BM1684X 出厂时已经预装 Ubuntu 20.04 系统，初始用户名和密码均为 linaro（root 账户无初始密码，使用前需要先用 linaro 账户做 sudo passwd root 设置密码）：

```
bm1684 login: linaro
Password:
Welcome to Ubuntu 20.04 LTS (GNU/Linux 5.4.202-bm1684 aarch64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

 * Super-optimized for small spaces - read how we shrank the memory
   footprint of MicroK8s to make it the smallest full K8s around.

   https://ubuntu.com/blog/microk8s-memory-optimisation
overlay / overlay rw,relatime,lowerdir=/media/root-ro,upperdir=/media/root-rw/
overlay,workdir=/media/root-rw/overlay-workdir 0 0
/dev/mmcblk0p5 /media/root-rw ext4 rw,relatime 0 0
/dev/mmcblk0p4 /media/root-ro ext4 ro,relatime 0 0

Last login: Mon Jul 11 11:30:26 CST 2022 from 192.168.0.105 on pts/0
linaro@bm1684:~$
```

检查 IP 地址请使用 ifconfig 或 ip a 命令:

```
ifconfig
ip a
```

如果需要手工配置静态 IP，可按如下方法修改/etc/netplan/01-netcfg.yaml 配置文件，并使能所修改的配置文件:

```
$ cat /etc/netplan/01-netcfg.yaml
network:
        version: 2
        renderer: networkd
        ethernets:
                eth0:
                        dhcp4: no                       # 静态 IP 需要改成 no, 动态 IP 则为 yes
                        addresses: [192.168.1.100/24]    # 加上 IP，动态 ip 则中括号内放空即
可
                        optional: yes
                        dhcp-identifier: mac            # 静态 IP 需要删掉这行
                eth1:
                        dhcp4: no
                        addresses: [192.168.150.1/24]
                        optional: yes
                enp3s0:
                        dhcp4: yes
                        addresses: []
                        dhcp-identifier: mac
                        optional: yes
$ sudo netplan try      # 测试配置是否可用
$ sudo netplan apply    # 使能最新配置
```

拿到 IP 地址后就可以使用 ssh 登录了，端口号为 22，用户名密码同样均为 linaro。

```
ssh linaro@your_ip
```

关机时建议使用 sudo poweroff 命令，尽量避免直接断电，以免文件系统损坏。

核心板有两个网卡，eth0 默认为 DHCP，故您需要通过上述方法获取 IP。eth1 默认配置为静态 IP：192.168.150.1。

CHAPTER 4

软件安装

4.1 检查预装版本

BM1684X 出厂时已经预装系统软件，在 Ubuntu 下可通过如下命令检查其版本：

a. 查看 Linux kernel 版本：bm_version

```
$ bm_version
sophon-mw-soc-sophon-ffmpeg : 0.2.3
sophon-mw-soc-sophon-opencv : 0.2.3
sophon-soc-libsophon : 0.2.3
boot_loader_version_bl1: v2.7(release):075b939dc-dirty Built : 14:30:23, Sep 15 2022
boot_loader_version_bl2: v2.7(release):075b939dc-dirty Built : 14:30:23, Sep 15 2022
boot_loader_version_bl31: v2.7(release):075b939dc-dirty Built : 14:30:23, Sep 15 2022
boot_loader_version_uboot: U-Boot 2022.07 075b939dc-dirty (Sep 15 2022 - 14:37:14␣
↪+0800) Sophon BM1684
KernelVersion : Linux bm1684 5.4.202-bm1684 #2 SMP PREEMPT Wed Jul 6 01:55:57
UTC 2022 aarch64 aarch64 aarch64 GNU/Linux
HWVersion: 0x00
MCUVersion: 0x03
```

sophon-mw-soc-sophon-ffmpeg、sophon-mw-soc-sophon-opencv 和 sophon-soc-libsophon 后面的信息为 SOPHON SDK 的版本号，boot_loader_version 后面的信息分别为 bl1、bl2、bl31 和 uboot 的 bootloader 版本号及 build 时间，KernelVersion 字段即为 Kernel 版本信息，5.4.202 表示官方 Linux Kernel 确切版本号，后半部分的时间戳代表 build 时间。MCUVersion 字段即为 MCU firmware 版本号。

4.2 软件更新

BM1684X 目前提供三种更新方式：SD 卡刷机，文件替换和 OTA 升级。其中 SD 卡刷机会重写整个 eMMC，也即您存储在 eMMC 的数据全部会丢失。这种方式最为干净可靠，理论上只要您的 BM1684X 没有硬件损坏，都可以进行 SD 卡刷机。文件替换方式是指在 Ubuntu 下通过替换对应文件的方式分别升级 bootloader、kernel 和其它软件。这种方式有一定的风险，如不同软件组件之间的版本匹配、文件损坏等。以下分别介绍三种软件更新方式的操作：

a. SD 卡刷机

请将 SD 卡格式化为 FAT32 格式（如果 SD 卡上有多个分区，只能使用第一个分区），大小为 1GB 以上。

请下载 BM1684X 最新刷机包，地址请见 FAQ 节：

请将下载的压缩包解压到 SD 卡根目录。确认文件如下（数量不一定相同）：

BOOT	boot_emmc-recovery.cmd	partition16G_ro.xml
boot.1-of-1.gz	boot_emmc-recovery.scr	recovery.1-of-1.gz
boot.cmd	boot_emmc-rootfs.cmd	rootfs.1-of-6.gz
boot.scr	boot_emmc-rootfs.scr	rootfs.2-of-6.gz
boot_emmc.cmd	boot_emmc-rootfs_rw.cmd	rootfs.3-of-6.gz
boot_emmc.scr	boot_emmc-rootfs_rw.scr	rootfs.4-of-6.gz
boot_emmc-boot.cmd	boot_emmc-system.cmd	rootfs.5-of-6.gz
boot_emmc-boot.scr	boot_emmc-system.scr	rootfs.6-of-6.gz
boot_emmc-data.cmd	boot_spif.cmd	rootfs_rw.1-of-1.gz
boot_emmc-data.scr	boot_spif.scr	spi_flash.bin
boot_emmc-gpt.cmd	data.1-of-1.gz	system.1-of-3.gz
boot_emmc-gpt.scr	fip.bin	system.2-of-3.gz
boot_emmc-misc.cmd	gpt.gz	system.3-of-3.gz
boot_emmc-misc.scr	misc.1-of-1.gz	

请将 BM1684X 断电，插入 SD 卡，并连接串口终端，然后给 BM1684X 上电。您将看到 BM1684X 自动进入刷机流程：

```
reading boot_emmc-gpt.scr
226 bytes read in 4 ms (54.7 KiB/s)
## Executing script at 310000000
reading gpt.gz
448 bytes read in 3 ms (145.5 KiB/s)
Uncompressed size: 17408 = 0x4400

MMC write: dev # 0, block # 0, count 34 ... 34 blocks written: OK

reading boot_emmc-boot.scr
240 bytes read in 4 ms (58.6 KiB/s)
## Executing script at 310000000
reading boot.1-of-1.gz
7754080 bytes read in 970 ms (7.6 MiB/s)
Uncompressed size: 134217728 = 0x8000000

MMC write: dev # 0, block # 8192, count 262144 ... 262144 blocks written: OK
```

刷机通常耗时约 3 分钟，结束后，会看到拔掉 SD 卡并重启 BM1684X 的提示，请依照操作即可：

```
eMMC update done
all done
Please remove the installation medium, then reboot
Please remove the installation medium, then reboot
Please remove the installation medium, then reboot
```

请注意：刷机后 Ubuntu 系统第一次启动时会进行文件系统初始化等关键动作，请勿随意断电，待开机进入命令行后使用 sudo poweroff 命令关机。

b. 文件替换

文件替换均在 Ubuntu 下执行，您可以选择使用串口或 SSH 终端。以下分别介绍如何替换各个组件。

替换 bootloader：请将您要更新的 spi_flash.bin 文件上传到 BM1684X ，然后执行 sudo flash_update -i ./spi_flash.bin -b 0x6000000，成功后可以看到如下 log：

```
linaro@bm1684:~$ sudo ./flash_update -i spi_flash.bin -b 0x6000000 -f 0x0
input file is: spi_flash.bin
write 0x112064 bytes to flash @ 0x6000000 + 0x0
0x6000000 mapped at address 0x7f853c9000.
INFO:  Start erasing 18 sectors, each 65536 bytes...
read flash:
  0  ff ff ff ff ff ff ff ff                          ........
INFO:  --program boot fw, page size 256
progress: 100%
INFO:  --program boot fw success
read flash:
  0  e0 3 1f aa e1 3 1f aa                           ........
unmapped 0x7f853c9000
```

可以执行 flash_update 查看帮助：

```
linaro@bm1684:/$ flash_update
BM1684/BM1684X update combo spi_flash.bin:
        sudo flash_update -i spi_flash.bin -b 0x6000000

BM1684 update fip.bin
        sudo flash_update -f fip.bin -b 0x6000000 -o 0x40000
BM1684 update spi_flash_bm1684.bin
        sudo flash_update -f spi_flash_bm1684.bin -b 0x6000000

BM1684X update fip.bin
        sudo flash_update -f fip.bin -b 0x6000000 -o 0x30000
BM1684X update spi_flash_bm1684.bin
        sudo flash_update -f spi_flash_bm1684x.bin -b 0x6000000

BM1684/BM1686X dump flash:
        sudo flash_update -d spi_flash_dump.bin -b 0x6000000 -o 0 -l 0x200000

BM1684/BM1686X write protection:
        sudo flash_update -p -b 0x6000000
```

替换 kernel：将您要更新的 emmcboot.itb 放入/boot 中替换同名文件，再 sudo reboot 即可。

替换 bmnnsdk2 运行时环境：bmnnsdk2 运行时环境位于/system 目录下，请将您拿到的更新包（通常是一个名为 system.tgz 的压缩包）整体替换即可，解压时请留意相对路径。替换 SophonSDK 运行时环境：SophonSDK 运行时环境位于/opt 目录下，请将您拿到的更新包（通常是一个名为 opt.tgz 的压缩包）整体替换即可，解压时请留意相对路径。

警告: 做完上述文件操作后不要马上暴力断电，否则可能会有文件损坏，请执行 sync、sudo reboot、sudo poweroff 等动作。

c. OTA 升级

按如下步骤可进行 OTA 升级:

1. 首先获取待更新版本的 SophonSDK 压缩包，获取其 sophon-img 子文件夹下的 bsp_update.tgz 和 system.tgz 压缩包。其中 bsp_update.tgz 主要包含升级脚本（bsp_update.sh）及内核镜像（emmcboot.itb）等内容，解压后的文件如下:

2. 将两个压缩包拷贝到模组的某一路径如家目录（/home/linaro）下，解压 bsp_update.tgz 压缩包并进入解压后的目录，执行 bsp_update.sh 升级脚本。可使用如下命令:

```
tar zxvf bsp_update.tgz
cd bsp_update
sudo ./bsp_update.sh
```

3. 回退至 system.tgz 所在目录，执行如下命令将 system.tgz 中的内容解压至/opt/sophon/libsophon-0.5.0 目录下:

```
sudo tar xzf system.tgz -C /opt/sophon/libsophon-0.5.0
sudo sync
```

4. 关机重启检查是否升级成功（可以通过 bm_version 查看 kernel 版本及 libsophon 的版本信息）。以下为升级前后的对比示例:

5. 如有内核开发的需求，需要升级内核开发软件包。同理，从对应版本的 SophonSDK 压缩包中获取 bsp-debs 将其拷贝至家目录（/home/linaro）下并在 bsp-debs 下创建 linux-headers-install.sh 脚本，脚本内容如下:

```
#!/bin/bash

cur_ver=$(uname -r)
echo ${cur_ver}
sudo mkdir -p /lib/modules/${cur_ver}
if [ -e /home/linaro/bsp-debs/linux-headers-${cur_ver}.deb ]; then
    if [ -d /lib/modules/${cur_ver} ]; then
```

（下页继续）

(续上页)

```
            sudo dpkg -i /home/linaro/bsp-debs/linux-headers-${cur_ver}.deb
            sudo mkdir -p /usr/src/linux-headers-${cur_ver}/tools/include/tools
            sudo cp /home/linaro/bsp-debs/*.h  /usr/src/linux-headers-${cur_ver}/tools/
↪include/tools
            cd /usr/src/linux-headers-${cur_ver}
            sudo make prepare0
            sudo make scripts
    else
            echo "/lib/modules not match"
    fi
else
    echo "linux header deb not match"
fi
```

如果遇到 linux-headers-install.sh 没有执行权限，使用如下命令增加权限：

```
chmod +x linux-headers-install.sh.sh
```

若脚本执行过程中出现缺少 flex 等错误，可执行如下命令安装相关环境：

```
sudo apt install flex bison libssl-dev
```

替换 MCU 固件：核心板上有一颗 MCU 负责 BM1684X 的上电时序等重要工作，它的固件只能通过下面的命令升级，不能通过 SD 卡升级。这颗 MCU 的固件如果烧写错误，会造成 BM1684X 无法上电，此时就只能通过专用的烧写器进行修复了，因此请谨慎操作，通常也并不需要对它进行升级。命令：sudo mcu-util-aarch64 upgrade 1 0x17 bm1686evb-mcu.bin。升级完成后请执行 sudo poweroff，待关机动作完成后（串口会打印 NOTICE: CPU0 bm_system_off，并且盒子的风扇声音会突然变大）对盒子进行断电后重新上电。

CHAPTER 5

系统软件构成

5.1 启动流程

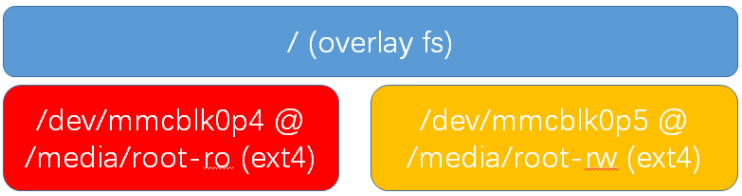
BM1684X 的系统软件属于典型的嵌入式 ARM64 Linux，由 bootloader、kernel、ramdisk 和 Ubuntu 20.04 构成，当开机后，依次执行如下：

其中：boot ROM、bootloader 基于 arm-trusted-firmware 和 u-boot 构建；kernel 基于 Linux 的 5.4 分支构建；Ubuntu 20.04 基于 Ubuntu 官方 arm64 源构建，不包含 GUI 相关组件。

5.2 eMMC 分区

分区设备文件	挂载点	文件系统	内容
/dev/mmcblk0p1	/boot	FAT32	存放 kernel 和 ramdisk 镜像
/dev/mmcblk0p2	/recovery	EXT4	存放 recovery mode 镜像
/dev/mmcblk0p3	无	无	存放配置信息，目前未使用
/dev/mmcblk0p4	/media/root-ro	EXT4	Ubuntu 20.04 系统的 read-only 部分
/dev/mmcblk0p5	/media/root-rw	EXT4	Ubuntu 20.04 系统的 read-write 部分
/dev/mmcblk0p6	/opt	EXT4	存放 sdk 的驱动和运行时环境
/dev/mmcblk0p7	/data	EXT4	存放用户数据，SOPHON 预装软件包未使用

关于第四和第五分区的说明：第四分区存储了 Ubuntu 20.04 系统的关键部分，挂载为只读；第五分区存储 Ubuntu 20.04 运行过程中产生的文件，挂载为可读可写。两个分区通过 overlayfs 聚合后挂载为系统的根目录，如下图所示：



用户通常无需关注此细节，对于日常使用来说是透明的，正常操作根目录下文件即可，但当用 df 等命令查看分区使用率等操作时请知悉此处，如下图：

```
linaro@linaro-developer:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
overlay         5.9G   31M  5.6G   1% /
devtmpfs        1.8G     0  1.8G   0% /dev
tmpfs           1.9G     0  1.9G   0% /dev/shm
tmpfs           1.9G   11M  1.9G   1% /run
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           1.9G     0  1.9G   0% /sys/fs/cgroup
/dev/mmcblk0p5  5.9G   31M  5.6G   1% /media/root-rw
/dev/mmcblk0p6  2.0G  6.0M  1.9G   1% /system
/dev/mmcblk0p4  1.7G  1.1G  429M  73% /media/root-ro
/dev/mmcblk0p7   18G   18M   18G   1% /data
/dev/mmcblk0p1  128M   16M  113M  13% /boot
/dev/mmcblk0p2  128M   84M   45M  66% /recovery
tmpfs           382M     0  382M   0% /run/user/1000
```

5.3 docker

核心板系统已经预装了 docker 服务，您可以用 docker info 命令查看状态。注意 docker 的根目录被配置到了/data/docker 目录下，与默认设置不同。

5.4 文件系统支持

如果您使用参考底板，当插入 U 盘或者移动硬盘后（需考虑 USB 供电能力），存储设备会被识别为/dev/sdb1 或类似节点，与桌面 PC Linux 环境下相同。文件系统支持 FAT、FAT32、EXT2/3/4、NTFS。BM1684X 不支持自动挂载，所以需要手工进行挂载：sudo mount /dev/sdb1 /mnt。当访问 NTFS 格式的存储设备时，预装的内核版本仅支持读取，如果需要写入，需要手工安装 ntfs-3g 软件包，请参考https://wiki.debian.org/NTFS。完成数据写入后，请及时使用 sync 或 umount 操作，关机时请使用 sudo poweroff 命令，避免暴力下电关机，以免数据丢失。

5.5 修改 SN 和 MAC 地址

BM1684X 的 SN 和 MAC 地址存放在 MCU 的 EEPROM 中，你可以通过如下方式进行修改。

首先需要解锁 MCU EEPROM：

```
sudo -i
echo 0 > /sys/devices/platform/5001c000.i2c/i2c-1/1-0017/lock
```

写入 SN：

```
echo "HQATEVBAIAIAI0001" > sn.txt
dd if=sn.txt of=/sys/bus/nvmem/devices/1-006a0/nvmem count=17 bs=1
```

count 参数需要根据实际写入的 SN 码长度进行修改。

读取 SN：

```
sudo -i
dd if=/sys/bus/nvmem/devices/1-006a0/nvmem count=17 bs=1 2>&1 | head -n 1 |␣
↪cut -c 1-17
```

其中 count 参数与 cut 参数需要根据实际写入的 SN 长度进行修改。

写入 MAC（双网卡各有一个 MAC）：

```
echo "E0A509261417" > mac0.txt
xxd -p -u -r mac0.txt > mac0.bin
dd if=mac0.bin of=/sys/bus/nvmem/devices/1-006a0/nvmem count=6 bs=1 seek=32
echo "E0A509261418" > mac1.txt
```

（下页继续）

(续上页)

```
xxd -p -u -r mac1.txt > mac1.bin
dd if=mac1.bin of=/sys/bus/nvmem/devices/1-006a0/nvmem count=6 bs=1 seek=64
```

最后重新对 MCU EEPROM 加锁，以避免意外改写：

```
echo 1 > /sys/devices/platform/5001c000.i2c/i2c-1/1-0017/lock
```

新的 MAC 地址将在重启系统后生效。

5.6 读写 eFuse

5.6.1 eFuse 寻址

BM1684X 内置的 eFuse 共 4096bit，按照 128×32bit 来组织，即地址范围 0~127，每个地址表示一个 32bit 的存储单元。eFuse 的每个 bit 初始值都是 0，用户可以将其从 0 修改成 1，但之后无法再从 1 修改成 0，比如您对地址 0 先写入 0x1，再写入 0x2，那么最后得到的是 0x1|0x2=0x3。

为了保证存储信息的可靠性，eFuse 中的信息通常都会存储两份，称为 double bit 机制，当两份拷贝中有任意一份为 1 时，即认为对应的 bit 为 1，即 result = copy_a OR copy_b。有两种存储形式：

1. 在一个 32bit 存储单元内进行 double bit，即奇数 bit（1、3、5、7……）和偶数 bit（0、2、4、6……）组成 double bit，比如约定地址 0 的 bit0 和 bit1，其中只要有一个为 1 就使能 secure firewall。这种形式的 double bit 用于硬件功能控制。
2. 若干个 32bit 存储单元与另外若干个存储单元组成 double bit。比如约定 SN = address[48] OR address[49]。这种形式的 double bit 用于软件定义信息的存储。

5.6.2 eFuse 分区

eFuse 里的一些地址有指定的用途，如下表：

地　址	内　容
0	bits[1] | [0]：使能 secure firewall bits[3] | [2]：禁用 JTAG bits[5] | [4]：禁止从片外 SPI flash 启动 bits[7] | [6]：使能 secure boot
1	bit[0] | bit[1]：使能 secure key
2~9	256bit secure key
10~17	256bit secure key 副本
18~25	256bit secure boot 使用的 root public key digest
26~33	256bit secure boot 使用的 root public key digest 副本
54~57	128bit 客户自定义 ID
58~61	128bit 客户自定义 ID 的副本
34~45	产品生产测试信息预留区域
64~82	产品生产测试信息预留区域

其余未注明区域目前没有特定用途，可以用作存储或实验之用。

5.6.3 eFuse 工具

BM1684X 中预装了一个 eFuse 读写工具，读写操作命令如下：

sudo efuse -r 0x? -l 0x? 即可以返回从该地址开始存储的若干个 32bit 值；

sudo efuse -w 0x? -v 0x? 即可在该地址写入指定的 32bit 值。

以上数值均只支持十六进制数。

CHAPTER 6

系统接口使用

BM1684X 的 CPU 占用率、内存使用率等信息均可使用标准的 Linux sysfs、procfs 节点，或 top 等工具读取。以下仅介绍 BM1684X 特有的一些接口或硬件使用方式。

6.1 读取核心板序列号

命令：

```
cat /sys/bus/i2c/devices/1-0017/information
```

返回（json 格式字符串）：


```
{
        "model": "BM1684X EVB",
        "chip": "BM1684X",
        "mcu": "STM32",
        "product sn": "",
        "board type": "0x20",
        "mcu version": "0x03",
        "pcb version": "0x00",
        "reset count": 0
}
```


命令：

```
cat /factory/OEMconfig.ini
```

返回：

```
linaro@bm1684:~/bsp-debs$ cat /factory/OEMconfig.ini
[BASE]
SN = BJSNS7MBCJGJA00WM
MAC0 = 58 c4 1e e0 1a 90
MAC1 = 58 c4 1e e0 1a 95
PRODUCT_TYPE = 0x01
AGING_FLAG = 0x01
DDR_TYPE = 16GB
BOARD_TYPE = V12
BOM = V12
MODULE_TYPE = SE6 DUO
EX_MODULE_TYPE = SE6 DUO
PRODUCT = SE6
VENDER = SOPHGO
ALGORITHM = 3RDPARTY
DEVICE_SN =
DATE_PRODUCTION =
PASSWORD_SSH = linaro
USERNAME = admin
PASSWORD = admin
```

命令：

```
bm_get_basic_info
```

返回：

```
linaro@bm1684:~/bsp-debs$ bm_get_basic_info
-------------------------------------------------------------------
chip sn: BJSNS7MBCJGJA00WM
device sn:
hostname: bm1684
uptimeInfo: up 14 minutes
boardtemperature: 41
coretemperature: 41
-------------------------------------------------------------------
```

6.2 读取 BM1684X 片上温度

命令：

```
cat /sys/class/thermal/thermal_zone0/temp
```

返回（单位为毫摄氏度）：

```
38745
```

即 38.745 摄氏度。

Linux 的 thermal 框架会使用这个温度做管理：

1. 普通版模组：当温度升到 85 度时，NPU 频率会降到 75%，CPU 降频到 1.15GHz；当温度回落到 80 度时，NPU 频率会恢复到 100%，CPU 频率恢复到 2.3GHz；当温度升到 90 度时，NPU 频率会降到最低挡位；当温度升到 95 度时，会自动关机。
2. 宽温版模组：当温度升到 95 度时，NPU 频率会降到 75%，CPU 降频到 1.15GHz；当温度回落到 90 度时，NPU 频率会恢复到 100%，CPU 频率恢复到 2.3GHz；当温度升到 105 度时，NPU 频率会降到最低挡位；当温度升到 110 度时，会自动关机。

另外，片外的 MCU 也会使用这个温度来做最后的保险机制：

1. 普通版模组：片上结温大于 95 度，并且板上温度大于 85 度时强制关机。
2. 宽温版模组：片上结温大于 120 度时强制关机。

6.3 读取核心板温度

命令：

```
cat /sys/class/thermal/thermal_zone1/temp
```

返回（单位为毫摄氏度）：

```
37375
```

即 37.375 摄氏度。

核心板温度通常会比前面读取的片上结温内部温度低。

6.4 读取功耗信息

命令：

```
sudo pmbus -d 0 -s 0x50 -i
```

返回：

```
I2C port 0, addr 0x50, type 0x3, reg 0x0, value 0x0
ISL68127 revision 0x33
ISL68127 switch to output 0, ret=0
ISL68127 output voltage: 749mV
ISL68127 output current: 2700mA
ISL68127 temperature 1: 59°C
ISL68127 output power: 2W → NPU 功耗
ISL68127 switch to output 1, ret=0
ISL68127 output voltage: 898mV
ISL68127 output current: 2900mA
ISL68127 temperature 1: 58°C
ISL68127 output power: 2W → CPU/Video 等功耗
```

第一组信息为 npu，第二组信息为 cpu 等。

pmbus 读取的是给 npu 和 cpu 供电的传感器的温度，所以更接近核心板温度，如果需要读取温度相关，请参考 4.2 和 4.3。

当使用 BM1684 设备时，命令如下：

```
sudo pmbus -d 0 -s 0x5c -i
```

返回结果格式与 BM1684X 相同。

6.5 使用 GPIO

BM1684X 包含 3 组 GPIO 控制器，每个控制 32 根 GPIO，与 Linux 的设备节点对应如下：

GPIO 控制器	Linux 设备节点	GPIO 物理编号	GPIO 逻辑编号
#0	/sys/class /gpio/gpiochip480	0 到 31	480 到 511
#1	/sys/class /gpio/gpiochip448	32 到 63	448 到 479
#2	/sys/class /gpio/gpiochip416	64 到 95	416 到 447

比如您需要操作电路图上标号为 GPIO29 的 pin，则需要：

```
sudo -i
echo 509 > /sys/class/gpio/export
```

然后就可以按照标准方式操作/sys/class/gpio/gpio509 下的节点了。

请注意，由于 pin 是复用的，并不是全部 96 根 GPIO 都可以使用，请与硬件设计结合确认。

6.6 使用 UART

BM1684X 的 144pin BTB 接口上提供了 3 组 UART，其中 UART0 已用作 bootloader 和 Linux 的 console 端口。

6.7 使用 I2C

BM1684X 的 144pin BTB 接口上提供了 1 组 I2C master，对应设备节点为/dev/i2c-2，可以使用标准的 I2C tools 和 API 操作。

在我们的参考底板上，BM1684X 通过这组 I2C 连接了底板上的 RTC 设备。

6.8 使用 PWM

警告： TODO: evb 板子风扇转速没法控制，需要更换硬件，待硬件完善后更新此章节

BM1684X 的 144pin BTB 接口上提供了 1 个 PWM 输出引脚，对应 PWM0：

```
sudo -i
echo 0 > /sys/class/pwm/pwmchip0/export
echo 0 > /sys/class/pwm/pwmchip0/pwm0/enable
```

配置：

```
echo %d > /sys/class/pwm/pwmchip0/pwm0/period
echo %d > /sys/class/pwm/pwmchip0/pwm0/duty_cycle
```

使能：

```
echo 1 > /sys/class/pwm/pwmchip0/pwm0/enable
```

在我们的参考底板上，这个 PWM 输出被用作风扇调速，Linux 的 thermal 框架会自动根据片上温度调整风扇转速。所以您会在第一步 export 时看到 Device or resource busy 错误，需要修改 device tree 把对应的 pwmfan 节点 disable 掉后才能自由使用：

```
pwmfan: pwm-fan {
  compatible = "pwm-fan";
  pwms = <&pwm 0 40000>, <&pwm 1 40000>; // period_ns
  pwm-names = "pwm0","pwm1";
  pwm_inuse = "pwm0";
  #cooling-cells = <2>;
  cooling-levels = <255 153 128 77 26 1>; //total 255
};
```

6.9 风扇测速

警告： EVB 板上需要更换风扇才能调节转速

BM1684X 的 144pin BTB 接口上提供了 1 个风扇测速输入引脚，可以采样风扇的转速输出方波频率，对照风扇手册上频率与转速的换算公式即可计算出实际转速。

使能：

```
sudo -i
echo 1 > /sys/class/bm-tach/bm-tach-0/enable
```

读取方波频率：

```
cat /sys/class/bm-tach/bm-tach-0/fan_speed
```

同时提供了一个 netlink 事件，当风扇停转时告警，示例代码可以从 bsp-sdk/linux-arm64/tools/fan_alert 获取。

6.10 查询内存用量

BM1684X 板载了 16GB DDR，可以分为三类：

1. kernel 管理的部分，即可以用 malloc、kmalloc 等常规 API 分配出来使用。
2. ION 管理的部分，预留给 NPU、VPU、VPP 使用，需要使用 ION 的 ionctl 接口，或使用 bmnnsdk2 中 bmlib 库提供的接口分配出来使用。
3. 预留给固件的部分，用户无法使用。

您可以使用如下方式检查各部分内存的用量：

1. 查看系统内存

```
linaro@bm1684:~$ free -h
               total        used        free      shared  buff/cache   available
Mem:           6.6Gi       230Mi       6.2Gi       1.0Mi       230Mi       6.3Gi
Swap:            0B          0B          0B
```

2. 查看 ION 内存

```
sudo -i
root@bm1684:~# cat /sys/kernel/debug/ion/bm_npu_heap_dump/summary  | head -2
Summary:
[0] npu heap size:4141875200 bytes, used:0 bytes        usage rate:0%, memory usage peak
0 bytes

root@bm1684:~# cat /sys/kernel/debug/ion/bm_vpu_heap_dump/summary  | head -2
Summary:
[2] vpu heap size:2147483648 bytes, used:0 bytes        usage rate:0%, memory usage peak
0 bytes

root@bm1684:~# cat /sys/kernel/debug/ion/bm_vpp_heap_dump/summary  | head -2
Summary:
[1] vpp heap size:3221225472 bytes, used:0 bytes        usage rate:0%, memory usage peak
0 bytes
```

如上，通常会有 3 个 ION heap（即三块预留的内存区域），如名字所示，分别供 NPU、VPU、VPP 使用。以上示例中只打印了每个 heap 使用信息的开头，如果完整地 cat summary 文件，可以看到其中分配的每块 buffer 的地址和大小信息。

CHAPTER 7

系统定制

因为 BM1684X 的底板可以由您自行设计，我们提供了一个 BSP SDK 以便您对内核和 Ubuntu 20.04 系统进行定制，然后生成自己的 SD 卡或 tftp 刷机包。由于从 V22.09.02 开始我们修改了 bootloader 的代码，导致无法使用 tftp 从 3.0.0 及以前的版本升级到 V22.09.02 及以后的版本，这种情况下请使用 SD 卡刷机升级。因为 BM1684X 核心板是制成品，故 bootloader 并未开放，如果需要定制请联系技术支持。

如果您只是希望部署自己的业务软件，并不涉及硬件修改，那么出于解耦的考虑，更推荐您把自己的业务软件打包成一个 deb 安装包。比如包含您的业务软件执行程序、依赖库、开机自启动服务等等，deb 安装包里还可以放一个安装时自动执行的脚本，在安装时做一些配置文件修改替换之类的操作。这样您可以单独安装、卸载、升级您的业务软件，避免与我们系统包版本的依赖问题，对产品部署后的批量更新等操作也更友好。deb 安装包的制作可以参考 Debian官方文档，或其他网上资料。

7.1 文件结构

BSP SDK 包含两部分：一部分为 github 网站（https://github.com/sophgo）上发布的源码文件，bootloarder-arm64 和 linux-arm64；另一部分基本不会改动的二进制文件，为避免影响 git 效率，是通过 NAS 发布的。请参考 bootloarder-arm64 源码文件的 README 中的描述将两部分合并，将看到如下文件：

```
top
├── bootloader-arm64
│   ├── scripts
│   │   └── envsetup.sh → 编译脚本入口
│   ├── trusted-firmware-a → TF-A 源代码
│   ├── u-boot → u-boot 源代码
```

（下页继续）

(续上页)

```
|   ├── ramdisk
|   |   └── build → ramdisk 的 cpio 文件和 u-boot 启动脚本文件
|   └── distro → Ubuntu 20.04 的定制部分
├── distro
|   └── distro_focal.tgz → Ubuntu 20.04 的原始底包
├── gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu → 交叉编译工具链
└── linux-arm64 → kernel 源代码
```

7.2 交叉编译

推荐在 Ubuntu 20.04 系统下进行交叉编译，不支持 X86_64 以外的架构。请预留至少 10GB 空闲磁盘空间，并请先安装必要的一些工具：

```
sudo apt install cmake libssl-dev fakeroot dpkg-dev device-tree-compiler u-boot-tools
```

进入到 BSP SDK 后，执行如下命令即可编译出 SD 卡和 tftp 刷机用的刷机包：

```
source bootloader-arm64/scripts/envsetup.sh
build_bsp
# build_update tftp 必须在 build_bsp 之后执行
build_update tftp
```

因为脚本中使用了 sudo，编译过程中可能会提示您输入当前用户密码。第一次编译时可能遇到各种问题，如结果不符合预期，请仔细检查编译 log，如果有遇到提示某某工具找不到的话，用 apt install 安装即可。

编译结果在 install/soc_bm1684 目录下，重点有如下几个文件（夹）：

```
sdcard → SD 卡刷机包，请参阅 2.2 节的使用说明 a；

tftp → tftp 刷机包，请参阅算能官方网站文档中心的 Sophon 设备和 SDK 使用
常见问题及解答相关章节的使用说明；

emmcboot.itb → kernel 和 ramdisk、dtb 打包在一起，请参阅 2.2 节的使用说明 b；

spi_flash.bin → bootloader，请参阅 2.2 节的使用说明 b；

rootfs → 根文件系统内容；
```

如果只是想更新内核的话，可以用如下命令：

```
build_kernel
build_ramdisk uclibc emmc
```

即可得到新的 emmcboot.itb。不建议您直接到 linux-arm64 目录下手敲 make，除非您非常清楚如何操作。

内核的编译结果在如下路径：

```
linux-arm64/build/bm1684/normal
```

编译出的 ko 可以在如下路径找到：

```
linux-arm64/build/bm1684/normal/modules/lib/modules/5.4.202-bm1684/kernel
```

两个路径下的内容是一样的，默认已经打进刷机包。

编译出的 linux-header 安装包（用于在板卡上编译 kernel module）可以在如下路径找到：

```
linux-arm64/build/bm1684/normal/debs
```

默认已经打进刷机包，即板卡上的/home/linaro/bsp-debs 目录。

7.3 修改 kernel

kernel 的配置文件在：

linux-arm64/arch/arm64/configs/bitmain_bm1684_normal_defconfig

请注意修改 kernel config 可能会造成您的 kernel 与我们通过二进制发布的驱动文件（板上/opt/sophon/libsophon-current/data/下的 bmtpu.ko、vpu.ko、jpu.ko）无法兼容。

标准版 BM1684X 使用的设备树文件在：

linux-arm64/arch/arm64/boot/dts/bitmain/bm1684x_evb_v0.0.dts

修改之后请执行：

```
build_kernel
build_ramdisk uclibc emmc
```

得到新的 emmcboot.itb 文件即包含了全部 kernel code 和 device tree 的修改。请替换到板卡的/boot 目录下并重启即可。

要注意的是，如果您把自己的 emmcboot.itb 部署到了板卡上，可能会造成板卡上预装的内容与您当前的内核镜像版本不一致。如果遇到兼容性问题，请把您编译主机上的 install/soc_bm1684/rootfs 下的/home/linaro/linux-dev 和/lib/module 两个目录也一起替换到板卡上即可。使用 tftp 或 SD 卡刷机包的话通常不会有这个问题，因为刷机包生成时总是会同步更新这些文件。

如果您使用的是 BM1684X 的某种变体，可以通过如下方式找到对应的 device tree 文件：

观察开机后 UART log 里 u-boot 打印的日志：

```
...
...
NOTICE:  BL31: Built : 07:47:33, Jun 29 2022
INFO:    ARM GICv2 driver initialized
INFO:    BL31: Initializing runtime services
INFO:    BL31: Preparing for EL3 exit to normal world
```

(下页继续)

(续上页)

```
INFO:    Entry point address = 0x308000000
INFO:    SPSR = 0x3c9
found dtb@130: bitmain-bm1684x-evb-v0.0
Selecting config 'bitmain-bm1684x-evb-v0.0'
...
...
```

关注 Selecting config 这一行，即可知道这块板子对应的 device tree 源文件是在 linux-arm64/arch/arm64/boot/dts/bitmain/目录下的 **bm1684x_evb_v0.0.dts**。

7.4 修改 Ubuntu 20.04

方式一：利用 Ubuntu 系统源码包进行修改

Ubuntu 20.04 系统的生成过程是这样：

1. distro/distro_focal.tgz 是 Ubuntu 官方纯净版底包。
2. bootloader-arm64/distro/overlay 下包含了 BM1684X 对底包的修改，会覆盖到底包的同名路径。
3. kernel 编译的过程中也会把 ko 等文件更新进去。
4. 如果 install/soc_bm1684 目录下有 system.tgz 文件，则刷机包生成过程中会把它作为/system 目录下的内容。
5. install/soc_bm1684 目录下有 data.tgz 文件，则刷机包生成过程中会把它作为/data 目录下的内容。

所以您可以在 overlay/bm1684 加入您自己的改动，比如放入一些工具软件，修改以太网配置文件等等，然后重新生成刷机包。

如果您有一个或多个 deb 想要预装到 Ubuntu 20.04，那么有两种做法：

a. 如果 deb 包比较简单，您可以直接将它解压缩后把里面的文件 copy 到 bootloader-arm64/distro/overlay/bm1684/rootfs 下的对应目录。

b. 将 deb 包直接放到 bootloader-arm64/distro/sophgo-fs/root/post_install/debs 目录，则 BM1684X 在刷机后第一次开机时会把这些 deb 包都安装上。

方式二：利用 qemu 虚拟机方式进行修改

（1）环境准备

1. 从官网获取 sdcard.tgz 基础软件包。
2. 解压 sdcard.tgz 到 sdcard 文件夹。

```
mkdir sdcard
tar -zxf sdcard.tgz -m -C sdcard
```

3. 将 bootloader-arm64/scripts/revert_package.sh 复制到 sdcard 目录下，然后制作 rootfs.tgz 软件包。

```
cd sdcard
sudo ./revert_package.sh rootfs
```

4. 在 sdcard 的同级目录中新建 rootfs 文件夹，并把 sdcard/rootfs.tgz 解压到 rootfs 文件夹下。

```
mkdir rootfs
sudo tar -zxf sdcard/rootfs.tgz -m -C rootfs
```

5. 安装 qemu 虚拟机。

```
sudo apt-get install qemu-user-static
```

（2）操作步骤

1. 进入 rootfs 目录，开启 qemu 虚拟机。

```
cd rootfs
sudo chroot . qemu-aarch64-static /bin/bash
```

2. 在虚拟机中安装好所需的 lib 以及工具后，例如 apt-get install nginx，安装完毕后执行 exit 退出虚拟机。

3. 在 rootfs 目录下打包修改后的 rootfs 文件系统，得到新的 rootfs.tgz 压缩包。

```
sudo tar -cvpzf rootfs.tgz ./*
```

4. 将新生成的 rootfs.tgz 软件包替换掉 install/soc_bm1684/下的 rootfs.tgz，然后根据需要重新编译刷机包。

```
build_update sdcard  // 重新编译 sdcard 刷机包
build_update tftp    // 重新编译 tftp 刷机包
```

7.5 定制化软件包

您可以通过以下操作获取您所需要的特定的软件包:

1. 从官网获取 sdcard.tgz 基础软件包。

2. 您需要参考文件结构一节准备相关文件，将 sdcard.tgz 软件包复制到 install/soc_bm1684 目录下，如果没有该目录，可以先执行以下命令

```
mkdir -p install/soc_bm1684
cp -rf {your_path}/sdcard.tgz install/soc_bm1684/  // {your_path}是您获取的 sdcard.tgz
基础软件包的本地路径
```

（下页继续）

(续上页)

```
source bootloader-arm64/scripts/envsetup.sh
revert_package
```

3. 执行完命令之后会在 install/soc_bm1684/下生成 boot.tgz，data.tgz，opt.tgz，recovery.tgz，rootfs.tgz，rootfs_rw.tgz 六个软件包，并且在 install/soc_bm1684/package_update/目录下生成 sdcard 和 update 两个文件夹，这里的 sdcard 文件夹是 sdcard.tgz 软件包解压出来的文件，update 文件夹保存了执行 revert_package 命令之后初始打包的六个软件包。

   boot.tgz 软件包主要用于 kernel。

   data.tgz 软件包主要用于 data 分区。

   opt.tgz 软件包包括运行时的 lib 库。

   recovery.tgz 软件包主要用于用户恢复出厂设置。

   rootfs.tgz 软件包可以用来制作您所需要的文件系统，参照修改 Ubuntu 20.04 一节中的利用 qemu 虚拟机方式进行修改部分的环境准备第 4 步及其之后的操作更新 rootfs.tgz 包，注意使用的 rootfs.tgz 原始包是 install/soc_bm1684/下的 rootfs.tgz。

   rootfs_rw.tgz 软件包文件系统 overlay 区，包括了所有系统安装的 app，lib，脚本，服务，/etc 下的设置，更新后都会清除。

4. 如果您要修改分区信息，您需要修改 bootloader-arm64/scripts/下的 partition32G.xml 文件。

5. 将修改后的 *.tgz 包替换掉 install/soc_bm1684/下的同名 *.tgz 包，然后根据需要重新编译刷机包。

```
build_update sdcard  // 重新编译 sdcard 刷机包
build_update tftp    // 重新编译 tftp 刷机包
```

7.6 如何通过 github 代码构建安装包

1. 从 http://219.142.246.77:65000/sharing/5ajzpas1H 下载工具链和 Ubuntu base。

2. 将它们放在与 bootloader-arm64 和 linux-arm64 同一级别的目录下，然后解压缩工具链，不需要解压缩发行版，你将得到以下文件夹：

```
.
├── bootloader-arm64
├── distro
│   └── distro_focal.tgz
├── gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu
└── linux-arm64
```

3. 执行以下命令

```
sudo apt install bison flex bc rsync kmod cpio sudo \
uuid-dev cmake libssl-dev fakeroot \
dpkg-dev device-tree-compiler u-boot-tools \
uuid-dev libxml2-dev debootstrap \
qemu-user-static kpartx
```

4. 编译 envsetup.sh 文件

```
source bootloader-arm64/scripts/envsetup.sh
```

5. 创建 bsp-debs 文件包

```
build_bsp_without_package
```

6. 从源码编译 SoC 版本。首先您需要编译 SoC BSP，请参考 BSP 的编译指导。

7. 在 GitHub 官网 https://github.com/sophgo/libsophon.git 下载 libsophon，参考 BSP 的编译指导编译 SoC BSP。在 SoC 模式下编译以获取 libsophon*.deb 文件包。

8. 在 https://developer.sophgo.com/site/index/material/all/all.html 网站下载 SDK，下载多媒体文件多媒体文件 sophon-mw-soc-sophon-ffmpeg*.deb ，sophon-mw-soc-sophon-opencv*.deb。

9. 拷贝 Sophon-soc-lib sophon*.deb、sophon-mw-soc-sophon-ffmpeg*.deb 和 Sophon-mw-soc-sophon-opencv*.deb 软件包到 soc_bm1684/bsp-debs 目录下。

10. 执行以下命令，即可得到 sdcard 刷机包。

```
build_ package
```

7.7 在 BM1684X 上编译内核模块

您也可以选择直接在 BM1684X 板卡上直接编译 kernel module，可以省去上述搭建交叉编译环境的麻烦。步骤如下：

1. uname -r 得到 kernel 版本号，与/home/linaro/bsp-debs 和/lib/modules 里面的文件名比较，确保一致
2. 因为 kernel 在交叉编译环境下做 make bindeb-pkg 的缺陷，需要再额外做如下处理：
   a. 用 date 命令检查当前系统时间，如果跟实际时间相差太多，请设置为当前时间，如

   ```
   sudo date -s "01:01:01 2021-03-01"
   ```

   b. 检查是否存在/home/linaro/bsp-debs/install.sh，如果有的话，执行它即可
   c. 如果没有的话，需要手工操作：

```
sudo dpkg -i /home/linaro/bsp-debs/linux-headers-*.deb
sudo mkdir -p /usr/src/linux-headers-$(uname -r)/tools/include/tools
sudo cp /home/linaro/linux-dev/*.h /usr/src/linux-headers-$(uname-r)/tools/include/
↪tools
cd /usr/src/linux-headers-$(uname -r)
sudo apt update
sudo apt-get install -y build-essential bc bison flex libssl-dev
sudo make scripts
```

3. 回到您的 driver 目录，make ko 吧

7.8 修改分区表

BM1684X 使用 GPT 分区表。分区表的配置文件在 bootloader-arm64/scripts/partition32G.xml，其中依次描述了每个分区的大小信息。不建议您修改分区的顺序和个数，以及 readonly 和 format 属性，以免与其它一些预装脚本中的写法发生冲突。您可以修改每个分区的大小。最后一个分区的大小不需要凑满 eMMC 实际容量，可以把它设成一个比较小的值，只要足够存放您准备预装的文件（即 data.tgz 解开后的内容）就可以。刷机后第一次开机时，会有一个脚本将这个分区自动扩大到填满 eMMC 的全部剩余可用空间。

7.9 修改 u-boot

u-boot 的配置文件在：

u-boot/configs/bitmain_bm1684_defconfig

板级头文件在：

u-boot/include/configs/bitmain_bm1684.h

板级 C 文件在：

u-boot/board/bitmain/bm1684/board.c

标准版 BM1684X 对应的 dts 文件是：

u-boot/arch/arm/dts/bitmain-bm1684x-evb-v0.0.dts

修改之后请执行：

```
build_fip
```

得到新的 spi_flash.bin，请将此文件放置到板卡上，参考 2.2.b 中的方式用 flash_update 工具更新后重启系统即可。

如果您使用的是 BM1684X 的某种变体，可以通过如下方式找到对应的 device tree 文件，请注意这个是 u-boot 自身使用的 device tree，并非 kernel 使用的 device tree：

观察开机后 UART log 里 u-boot 打印的日志：

```
...
...
NOTICE:  BL31: Built : 07:47:33, Jun 29 2022
INFO:    ARM GICv2 driver initialized
INFO:    BL31: Initializing runtime services
INFO:    BL31: Preparing for EL3 exit to normal world
INFO:    Entry point address = 0x308000000
INFO:    SPSR = 0x3c9
found dtb@130: bitmain-bm1684x-evb-v0.0
Selecting config 'bitmain-bm1684x-evb-v0.0'
...
...
```

关注 Selecting config 这一行，即可知道这块板子对应的 device tree 源文件是在 u-boot/arch/arm/dts/ 目录下的 **bitmain-bm1684x-evb-v0.0.dts**.

7.10 修改板卡预制的内存布局

本工具需要运行在 PC 机上，不可在板卡上直接运行推荐使用 Ubuntu 20.04 系统，Python 3.8 版本环境。如果您想直接修改当前板卡上的内存布局，请获取工具包，路径在 http://219.142.246.77:65000/fsdownload/5ajzpas1H/BSP%20SDK 的 memory_layout_modification_tool 目录下，仅用到这个文件夹，不需要其它完整的源码和交叉编译工具链等。其中包含如下文件：

├── dtc → device tree compiler

├── dumpimage → itb 解包工具

├── gen_mm_dts.py → 生成 memory layout 描述的脚本

├── gen_mm_dts.sh → 配合同名 python 脚本使用的 expect 脚本

├── gui_new_update_itb_its.py → 修改内存界面工具脚本

├── mkimage → itb 打包工具

├── new_update_itb_its.py → 修改内存命令行脚本

├── new_update_itb_its.sh → 配合同名 python 脚本使用的 expect 脚本

└── reassemble.sh → 打包 itb 的脚本

您直接用到的是 new_update_itb_its.py 和 gui_new_update_itb_its.py 脚本；其中 new_update_itb_its.py 是以命令行的方式进行修改内存布局操作，没有窗口界面；而 gui_new_update_itb_its.py 运行后会显示一个修改内存布局操作界面，适合带有桌面的 Ubuntu 系统。这两个脚本的具体操作步骤如下：

（一）new_update_itb_its.py 脚本操作步骤：

1. 从板卡的/boot 目录下 copy 出 emmcboot.itb 和 multi.its 两个文件放到脚本同级目录下 (即 mm_layout/目录下)。

2. 在 mm_layout 目录下，使用 Python 运行 new_update_itb_its.py 文件：

```
python3 new_update_itb_its.py
```

它会解开 emmcboot.itb，然后提示您选择要修改哪个 device tree；进入到所需要修改 dtb 文件的选择页面；标准版 SM5 对应的文件为 bm1684_asic_modm.dtb。

```
0): ./output/bm1684_cust_v1.1.dtb
1): ./output/bm1684_se5_v1.1.dtb
2): ./output/bm1684_se5_v1.1_mm0.dtb
3): ./output/bm1684_se5_v1.1_mm1.dtb
4): ./output/bm1684_evb_v1.2.dtb
5): ./output/bm1684_sm5_v1.1_rb.dtb
6): ./output/bm1684_sm5_v1.1_rb_mm0.dtb
7): ./output/bm1684_sm5_v1.1_rb_mm1.dtb
8): ./output/bm1684_sm5m_v0.0_rb.dtb
9): ./output/bm1684_sm5m_v0.0_tb.dtb
10): ./output/bm1684_sm5m_v0.1_rb.dtb
11): ./output/bm1684_sm5m_v0.1_tb.dtb
12): ./output/bm1684_sm5_v1.2_rb.dtb
13): ./output/bm1684_sm5_v1.2_tb.dtb
14): ./output/bm1684_sm5m_v0.2_rb.dtb
15): ./output/bm1684_sm5m_v0.2_tb.dtb
16): ./output/bm1684_se5_v2.0.dtb
17): ./output/bm1684_se5_v2.5.dtb
18): ./output/bm1684_se6_ctrl.dtb
19): ./output/bm1684_sm5m_v3.0_rb.dtb
20): ./output/bm1684_sm5m_v3.0_tb.dtb
21): ./output/bm1684_cust_v1.3.dtb
22): ./output/bm1684_sc5_ep.dtb
23): ./output/bm1684_sc5_mix.dtb
24): ./output/bm1684x_pld.dtb
25): ./output/bm1684x_fpga.dtb
26): ./output/bm1684x_evb_v0.0.dtb

Please choose dtb to be modified:
dtb index:
```

您可以通过在板卡上执行以下命令获取板卡对应的 dtb 文件名称：

```
cat /proc/device-tree/info/file-name
```

3. 输入所需要修改的板卡的 dtb 文件的序号，之后会列出当前这份 device tree 里的内存状况，并进入内存布局的功能操作选择流程：

```
Please choose dtb to be modified:
dtb index: 1
you choose ./output/bm1684_se5_v1.1.dtb
the information of memory...
DDR0 total size 0x100000000, 0x100000000 -> 0x1fffffff
DDR1 total size 0x100000000, 0x300000000 -> 0x3fffffff
DDR2 total size 0x100000000, 0x400000000 -> 0x4fffffff
new style node name
DDR0 available size 0xfaf00000, 0x105100000 -> 0x1fffffff
DDR1 available size 0xd7ec0000, 0x328140000 -> 0x3fffffff
DDR2 available size 0x100000000, 0x400000000 -> 0x4fffffff
0): ION0 for npu,on DDR0, current addr 0x105100000, size 0x510000000000000
1): ION2 for vpu,on DDR1, current addr 0x380000000, size 0x8000000000000000
2): ION1 for vpp,on DDR2, current addr 0x440000000, size 0x4000000000000000
['npu', 'vpu', 'vpp']


0) update
1) delete
2) add
3) finish all operation and generate new files
Please choose your operation:
operation index:
```

如上图，先列出了所有 DDR channel 的物理内存大小。然后列出了每个 DDR channel 上除去固定分配区域后还有多少可以调整的。接下来就会逐个列出 NPU、

VPP、VPU 三个区域所在 DDR channel 起始地址和大小；同时显示能够对内存布局的进行的操作。

0）update → 表示修改内存布局，注意这里不能调整区域所在的 DDR channel，不能删除或增加区域，只能修改大小，且大小不能超过上面开列的每个 DDR channel 的可用空间。

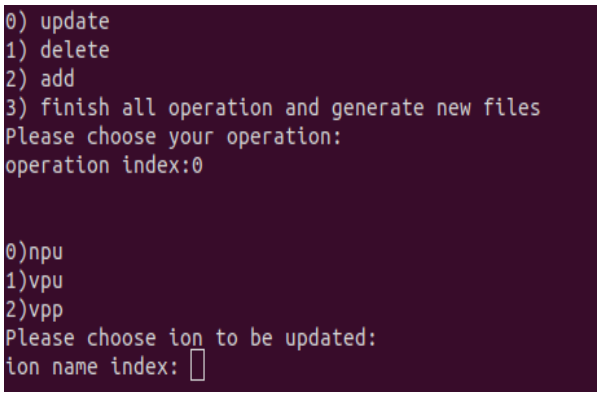
1）delete → 表示删除已有的内存区，例如删除 NPU、VPP、VPU 区域。

2）add → 表示增加板卡里没有的内存区，注意这里只能增加 NPU、VPP、VPU 区域，若 device tree 内已经存在该内存区域，则不能进行增加内存操作，且增加的内存大小不能超过上面开列的每个 DDR channel 的可用空间 (例如 device tree 内存在 NPU 区域，则无法再增加 NPU 内存区域)。

3）finish all operation and generate new files → 表示结束对内存布局的功能操作，生成新的 emmcboot.itb 文件。注意选择 update、delete、add 等操作之后，需要选择 finish all operation and generate new files 操作才会退出程序。

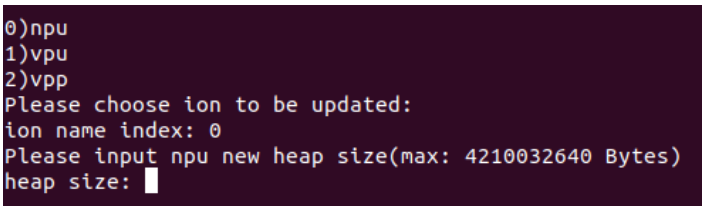
输入您所需要的内存操作序号后执行相应的操作，上述各操作的详细流程如下：

3.1 update 操作步骤：



如上图，输入 0 之后进入修改内存布局操作流程，然后会列出可供修改的内存区域名称及其序号。

3.1.1 选择您所需要修改的内存区域的序号：



选择您需要修改的内存区后，会让您输入该 ion 区域的新分配的内存大小 (10 进制和 16 进制都可以，16 进制数以 0x 开头)，并且列出了能够分配的最大内存大小。

3.1.2 输入您所需要修改的内存大小：

```
Please input npu new heap size(max: 4210032640 Bytes)
heap size: 1000000
Whether to continue to update memory?
[yes/no]:
```

输入所需要修改的内存大小后，会询问您是否需要继续进行修改内存区域操作。

3.1.3 选择您是否需要继续修改内存：

yes:(表示继续修改内存区域)

```
Whether to continue to update memory?
[yes/no]: yes


0)npu
1)vpu
2)vpp
Please choose ion to be updated:
ion name index:
```

选择 yes 后会继续进行修改内存操作，回到 3.1.1 步骤继续修改内存；注意，如果您继续修改内存选择了相同的 ion 区域操作会覆盖之前对该区域的操作，以最新的操作为准；如果您继续修改内存选择了不同的 ion 区域操作会记录之前的所有修改操作，执行所有的修改内存操作，如果您的更新的内存区域的起始地址不是所属 DDR channel 的可用起始基地址，系统会提示您相应的 warning，并把所更新的内存区域的起始地址重置到所属 DDR channel 的可用起始基地址。

no:(表示结束修改内存区域操作)

```
Whether to continue to update memory?
[yes/no]: no
you update the memory layout of npu!
update memory finished.


0) update
1) delete
2) add
3) finish all operation and generate new files
Please choose your operation:
operation index:
```

```
Warning: The current vpu ion start address is not an available start base address
```

选择 no 后会退出修改内存操作，显示您更新了哪个区域提示信息，回到 3 步骤继续选择对内存的功能操作；注意，如果您想结束所有的操作，需要选择 3 结束内存布局的功能操作，生成新的 emmcboot.itb 文件。

3.2 delete 操作步骤：

```
0) update
1) delete
2) add
3) finish all operation and generate new files
Please choose your operation:
operation index:1


0)npu
1)vpu
2)vpp
Please choose ion to be deleted:
ion name index:
```

如上图，输入 1 之后进入删除内存操作流程，然后列出可供删除的内存区域名称及序号。

3.2.1 选择您需要删除的内存区域序号：

```
0)npu
1)vpu
2)vpp
Please choose ion to be deleted:
ion name index: 0
Whether to continue to delete memory?
[yes/no]:
```

选择所需要删除的内存区域之后，会询问您是否需要继续进行删除内存区域操作。

3.2.2 选择您是否需要继续删除内存区域：

yes:(表示继续删除内存区域)

```
Whether to continue to delete memory?
[yes/no]: yes


0)vpu
1)vpp
Please choose ion to be deleted:
ion name index:
```

选择 yes 之后会继续进行删除内存操作，回到 3.2.1 步骤继续删除内存；注意，如果您继续删除内存区域，之前删除过的区域不可再删除，可供删除的内存区域也不再显示之前删除过的内存区。

no:(表示结束删除内存区域操作)


```
Whether to continue to delete memory?
[yes/no]: no
you delete the memory layout of npu!
delete memory finished.


0) update
1) delete
2) add
3) finish all operation and generate new files
Please choose your operation:
operation index:
```

选择 no 后会退出删除内存区域操作，显示您删除了哪个区域提示信息，回到 3 步骤继续选择对内存的功能操作；注意，如果您想结束所有的操作，需要选择 3 结束内存布局的功能操作，生成新的 emmcboot.itb 文件。

3.3 add 操作步骤：

```
0) update
1) delete
2) add
3) finish all operation and generate new files
Please choose your operation:
operation index:2
npu、vpu、vpp already exist, can not add memory!


0) update
1) delete
2) add
3) finish all operation and generate new files
Please choose your operation:
operation index:
```

注意：如果 device tree 内已经存在 NPU、VPU、VPP 内存区域，则不能进行增加内存操作，提示您 “npu、vpu、vpp already exist, can not add memory! “信息，并回到步骤 3 继续选择对内存的功能操作。

```
0) update
1) delete
2) add
3) finish all operation and generate new files
Please choose your operation:
operation index:2


0)npu
1)vpu
Please choose ion to be added:
ion name index:
```

如果 device tree 内不全存在 NPU、VPU、VPP 内存区域，则能进行增加内存操作，并显示可供增加的内存区域。

3.3.1 选择您所需要增加的内存区域的序号：

```
0)npu
1)vpu
Please choose ion to be added:
ion name index: 0
Please input add npu heap size(max: 4210032640 Bytes)
heap size:
```

选择您需要增加的内存区后，会让您输入该 ion 区域的新分配的内存大小（10 进制和 16 进制都可以，16 进制数以 0x 开头），并且列出了能够增加的最大内存大小。

3.3.2 输入您所需要增加的内存区域的大小：

```
0)npu
1)vpu
Please choose ion to be added:
ion name index: 0
Please input add npu heap size(max: 4210032640 Bytes)
heap size: 2000000
Whether to continue to add memory?
[yes/no]:
```

输入所需要增加的内存大小后，会询问您是否需要继续进行增加内存区域操作。

3.3.3 选择您是否需要继续增加内存区域

yes:(表示继续增加内存区域)

```
Whether to continue to add memory?
[yes/no]: yes


0)vpu
Please choose ion to be added:
ion name index:
```

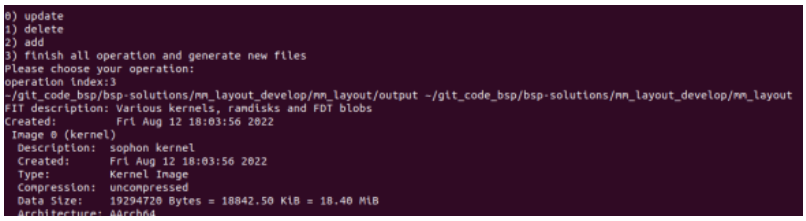
选择 yes 之后会继续进行增加内存区域操作，回到 3.3.1 步骤继续增加内存；注意，如果您继续增加内存区域，之前增加过的区域不可再增加，可供增加的内存区域也不再显示之前增加过的内存区。

no:(表示结束增加内存区域操作)

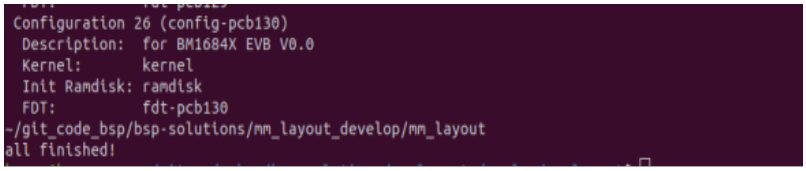
```
Whether to continue to add memory?
[yes/no]: no
you add the memory layout of npu!
you add the memory layout of vpu!
add memory finished.


0) update
1) delete
2) add
3) finish all operation and generate new files
Please choose your operation:
operation index:
```

选择 no 后会退出增加内存区域操作，显示您增加了哪个区域提示信息，回到 3 步骤继续选择对内存的功能操作；注意，如果您想结束所有的操作，需要选择 3 结束内存布局的功能操作，生成新的 emmcboot.itb 文件。

3.4 finish all operation and generate new files 操作步骤



……



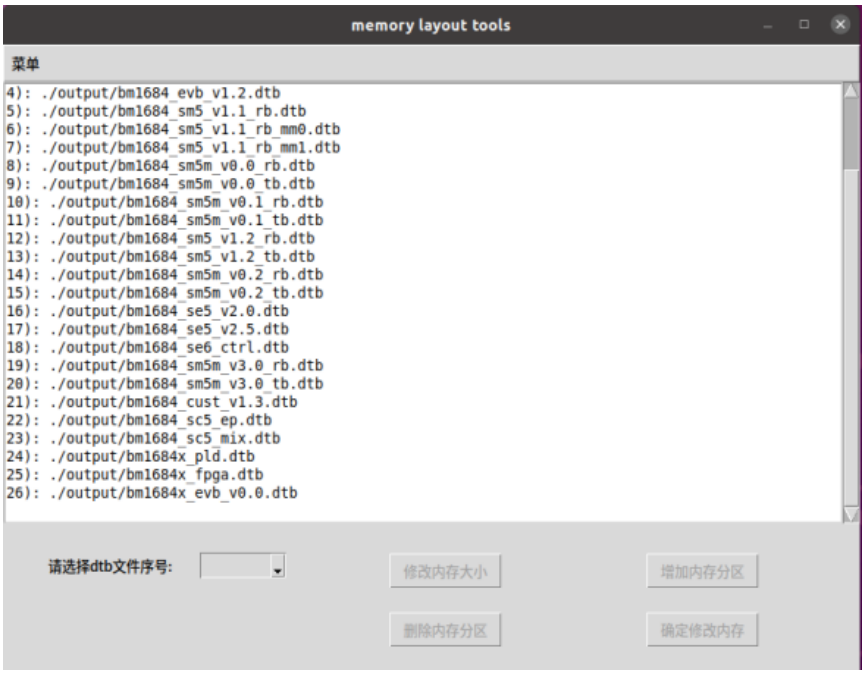
如上图，输入 3 之后结束所有的操作，提示您 “all finished!” 信息，生成新的 emmcboot.itb 文件。

（二）gui_new_update_itb_its.py 脚本操作步骤：

1. 从板卡的/boot 目录下 copy 出 emmcboot.itb 和 multi.its 两个文件放到脚本同级目录下 (即 mm_layout/目录下)。

2. 在 mm_layout 目录下，使用 Python(Python3.0 以上版本，建议使用 Python3.8) 运行 gui_new_update_itb_its.py 文件：

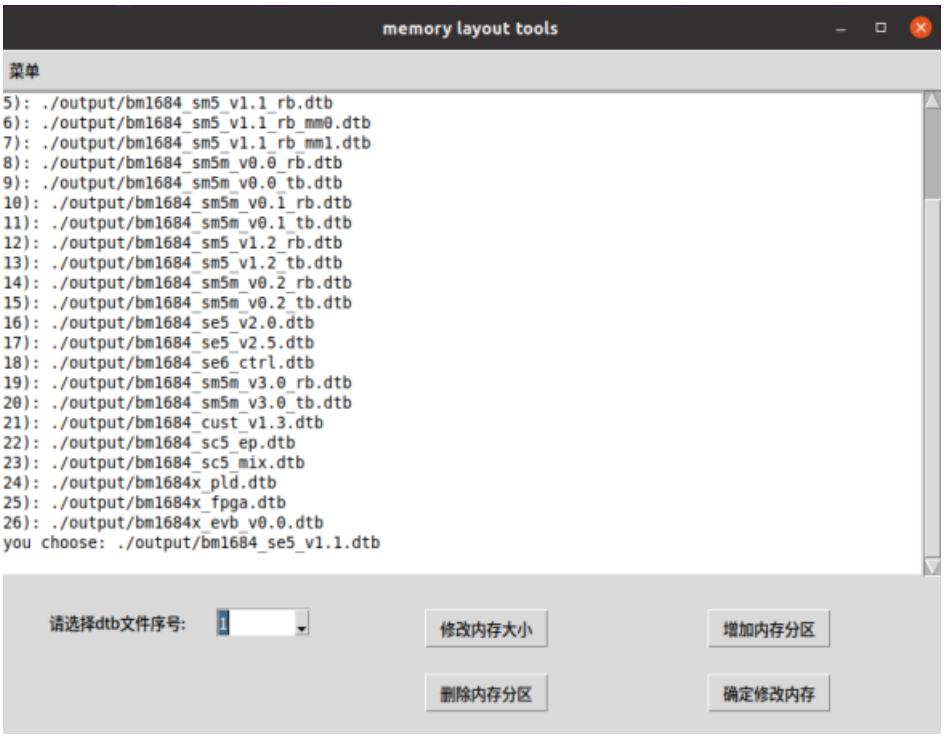
```
python3 gui_new_update_itb_its.py
```



运行该脚本之后会显示一个操作界面，它会解开 emmcboot.itb，然后显示您要修改哪个 device tree 的相关信息，标准版 SM5 对应的文件为 bm1684_asic_modm.dtb；您在进行相关操作之前需要先选择您所需要修改的 dtb 文件序号才可继续进行操作，您可以通过在板卡上执行以下命令获取板卡对应的 dtb 文件名称：

```
cat /proc/device-tree/info/file-name
```

3. 选择所需要修改的板卡的 dtb 文件序号，之后会激活相关功能的操作按钮，并提示您选择了哪个文件。



如上图，当您选择所需要操作的 dtb 文件之后，相关的功能操作按钮已经激活，有修改内存大小、删除内存分区、增加内存分区和确定修改内存四个功能按钮，当您进入修改内存相关操作的子页面时，主页面将暂时停止使用，子页面关闭后可继续使用；同时左上角的功能菜单中还有清空当前输出面板信息的功能。

0）修改内存大小 → 表示修改内存布局，注意这里不能调整区域所在的 DDR channel，不能删除或增加区域，只能修改大小，且大小不能超过所属 DDR channel 的可用空间。

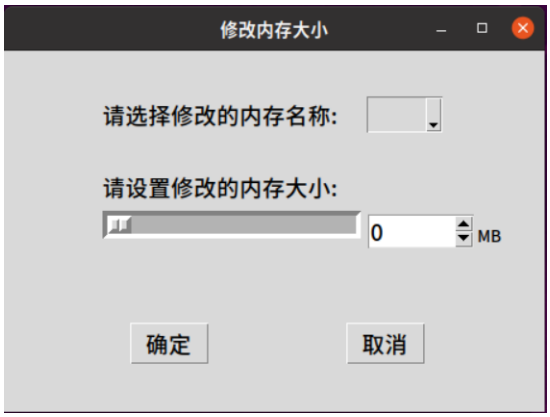
1）删除内存分区 → 表示删除已有的内存区，例如删除 NPU、VPP、VPU 区域。

2）增加内存分区 → 表示增加板卡里没有的内存区，注意这里只能增加 NPU、VPP、VPU 区域，若 device tree 内已经存在该内存区域，则不能进行增加内存操作，且增加的内存大小不能超过所属 DDR channel 的可用空间 (例如 device tree 内存在 NPU 区域，则无法再增加 NPU 内存区域)。

3）确定修改内存 → 表示确定修改内存布局的操作，注意在进行修改、增加、删除内存等操作之后，需要点击确定修改内存操作才会生成新的 emmcboot.itb 文件，且当您想要切换所修改的 dtb 文件时，也需要先点击此按钮。

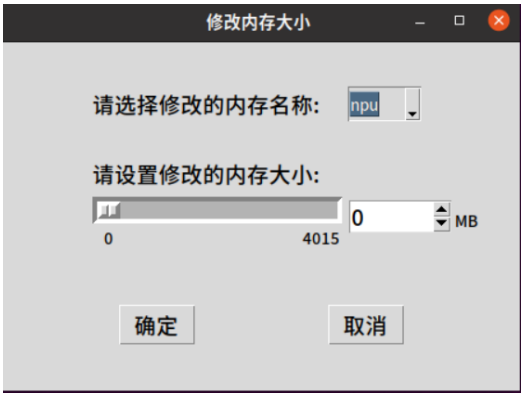
点击您所需要的内存操作按钮后将会执行相应的功能，上述各功能操作的详细流程如下：

3.1 修改内存大小操作步骤：



如上图，点击修改内存大小按钮后进入修改内存大小的子页面，在该页面可以选择修改内存区域的名称，可以拉动滑条设置所修改内存的大小，也可以手动输入和点击上下按钮设置所修改的内存大小，但是不能超过该内存区所属的 DDR channel 的可用空间大小。

3.1.1 选择您所需要修改的内存区域名称:



选择所修改的内存区域名称后，滑条会自动设置所能修改的最大内存大小。

3.1.2 设置所修改内存的大小，单位是 MB:

如上图，拉动滑条后，内存大小输入区会显示相应的大

小，也可以在该输入区输入所设置的新内存区域的大小，还可通过输入区右边的上下箭头微调大小；在输入区设置好大小后滑条也会调节到相应的位置，注意这里不能给所修改的内存区域设置 0MB 的新大小。

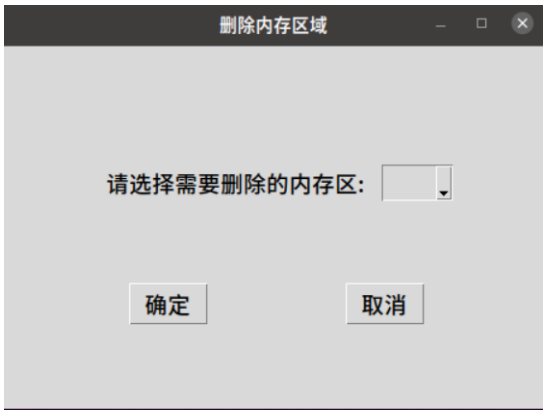
3.1.3 点击确定或者取消的操作按钮

点击确定按钮后，会弹出确认选择框，点击 OK 即可完成修改内存大小操作，回到主界面，点击确定修改内存按钮后完成本次修改内存大小操作，您也可以点击其余功能操作按钮，最后执行确定修改操作；若您对之前的内存大小调节操作有不满意的地方可以点击 Cancel 按钮返回修改内存大小子界面，重新设置相关信息。

点击取消按钮后，会返回主界面，取消本次修改内存大小操作。

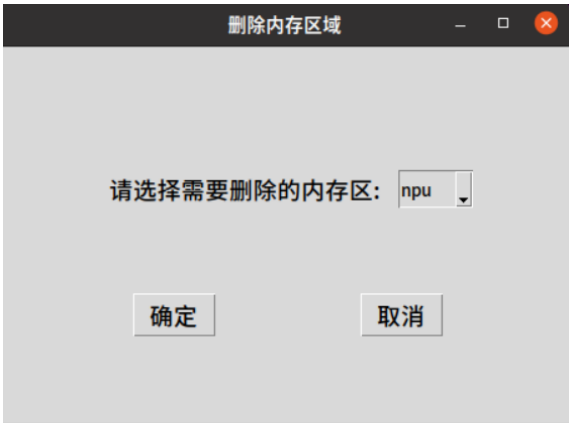
如果您执行确定修改操作后，您的更新的内存区域的起始地址不是所属 DDR channel 的可用起始基地址，系统会提示您相应的 warning，并把所更新的内存区域的起始地址重置到所属 DDR channel 的可用起始基地址。

3.2 删除内存分区操作步骤：



如上图，点击删除内存分区按钮后进入删除内存区域的子页面，在该页面可以选择删除内存区域的名称。

3.2.1 选择您所需要删除的内存区域名称：

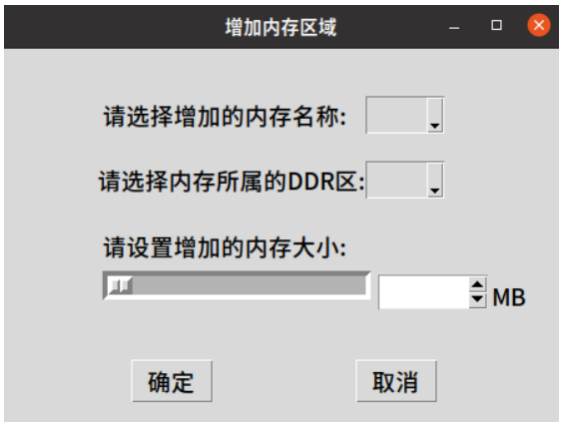


3.2.2 点击确定或者取消的操作按钮

点击确定按钮后，会弹出确认选择框，点击 OK 即可完成删除内存区域操作，回到主界面，点击确定修改内存按钮后完成本次删除内存分区操作，您也可以点击其余功能操作按钮，最后执行确定修改操作；若您对之前的删除内存操作有不满意的地方可以点击 Cancel 按钮返回删除内存区域子界面，重新设置相关信息。

击取消按钮后，会返回主界面，取消本次删除内存区域操作。

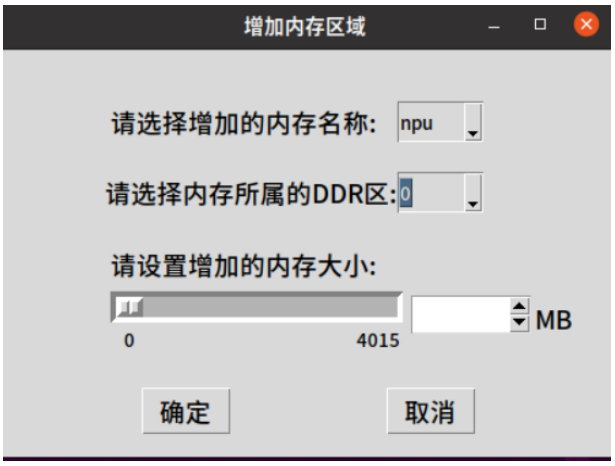
3.3 增加内存分区操作步骤：



如上图，点击增加内存分区按钮后进入增加内存区域子页面，在该页面可以选择增加内存区域的名称以及所属的 DDR channel，可以拉动滑条设置所增加内存的大小，也可以手动输入和点击上下按钮设置所增加的内存大小，但是不能超过该内存区所属的 DDR channel 的可用空间大小。注意，如果 device tree 内已经存在 NPU、VPU、VPP 内存区域，则不能进行增加内存操作。

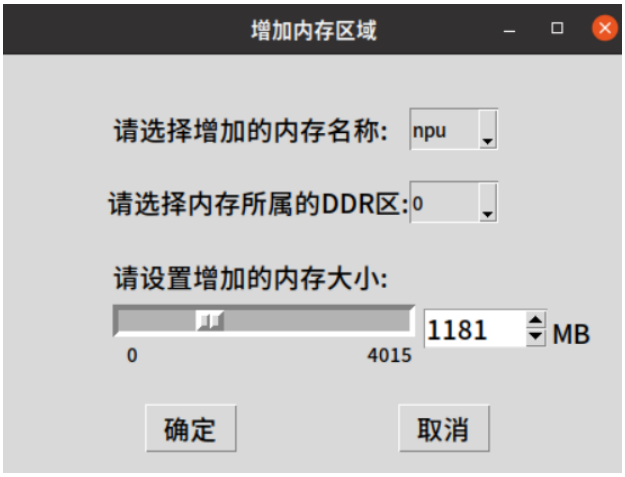
3.3.1 选择您所需要增加的内存区域名称：

3.3.2 选择所选内存区域的所属 DDR 区：



选择所属的 DDR 区后，滑条会自动设置所能增加的最大内存大小。

3.3.3 设置所增加内存的大小，单位是 MB：



如上图，拉动滑条后，内存大小输入区会显示相应的大小，也可以在该输入区输入所设置的新内存区域的大小，还可通过输入区右边的上下箭头微调大小；在输入区设置好大小后滑条也会调节到相应的位置，注意这里不能给所增加的内存区域设置 0MB 的新大小。

3.3.4 点击确定或者取消的操作按钮

点击确定按钮后，会弹出确认选择框，点击 OK 即可完成增加内存分区操作，回到主界面，点击确定修改内存按钮后完成本次增加内存分区操作，您也可以点击其余功能操作按钮，最后执行确定修改操作；若您对之前的内存大小调节操作有不满意的地方可以点击 Cancel 按钮返回增加内存分区子界面，重新设置相关信息。

点击取消按钮后，会返回主界面，取消本次增加内存分区操作。

3.4 确定修改内存操作步骤：

点击确定修改内存按钮后会在 output 目录下生成新的 emmcboot.itb 文件，结束之前对所选 dtb 文件的功能操作，您可以回到步骤 2 后选择其他 dtb 文件进行功能操作。

4. 最后界面关闭或者点击确定修改内存按钮后会在 output 目录下生成新的 emmcboot.itb 文件。将它替换回板卡的/boot 目录下，执行 sudo reboot 操作即可；另外，在 output 目录下会保留原始的 dts 和 dtb 文件供您比较。

注意事项：

（1）如果遇到 shell 提示” dtc not found”，在 Ubuntu 系统上可以通过执行以下命令解决：

```
sudo apt install device-tree-compiler
```

或者，mm_layout 文件夹里也提供了一个 dtc 执行文件，请把这个文件夹加入到 shell 的 PATH 变量即可，在 shell 里执行：

```
PATH=$PATH:/path/to/mm_layout/folder
```

然后再执行 new_update_itb_dts.py 或者 gui_new_update_itb_its.py 脚本。

（2）目前只能对 NPU、VPU、VPP 三个内存区域进行操作，其中 NPU 在 DDR0 上、VPU 在 DDR1 上、VPP 在 DDR2 上。

7.11 选择板卡预制的内存布局

BM1684 当前默认的内存布局可能不适合部分 yolo 等较大模型的精度测试，所以我们提供了一种定制的内存布局，让操作系统能够扩大使用的内存，方便客户进行精度测试。首先需要确认当前板卡使用的 pcb_version，可以通过

```
cat /proc/device-tree/info/file-name
```

获取当前板卡使用的 device-tree 的名字，然后打开/boot/multi.its 文件，搜索当前板卡的 device-tree 名字，找到当前 device-tree 对应的 fdt-pcb 后面的数字，这个数字就是 pcb_version 号。获取到 pcb_version 号后，可以通过如下方式进行切换内存布局，下述方式以 pcb_version 为 7 为例，如果使用其他型号的板卡，请自己将 extra-后面的数字换成真正的 pcb_verison。

```
sudo apt update
sudo apt install u-boot-tools
echo "set memory_model 0" > extra-7.cmd
mkimage -A arm64 -O linux -T script -C none -a 0 -e 0 -n \
   "Distro Boot Script" -d extra-7.cmd extra-7.scr
sudo cp extra-7.scr /boot
sudo reboot
```

如上会选中 u-boot 中 pcb_version 变量为 7 的板卡的第 0 个预制的内存布局，如果想要恢复此板卡默认的内存布局，删除/boot/extra-7.scr 后重启即可。

7.12 1684x kdump-crash 使用说明

本文记录了如何在 1684x ubuntu20.04 上使用 kexec/kdump-tools 生成 linux kernel coredump 文件，并用 crash 分析该 coredump 文件。

1. 环境准备

1）X86 主机

a）sd 卡 - 32G 以上容量, coredump 文件比较大，压缩的 coredump 文件，9GB 左右，非压缩的，16GB 左右（等于 ram 大小）

b）crash（https://github.com/crash-utility/crash/tags 选择 8.0 以上版本，x86 主机编译命令：make target=ARM64）或者使用随本文一起发布的 crash 命令，使用 crash 前需要在 Ubuntu 上安装 libncursesw6，libtinfo6，liblzma5，bison，libncurses-dev

c）vmlinux(与板子运行内核一致的，带有调试信息的内核文件，可以从 1684x 的/home/linaro/bsp-debs/linux-image-*-dbg.deb 中获取，在 linux 主机解压：dpkg-deb -R linux-image-*-dbg.deb linux-image-*-dbg，解压后在/linux-image-*-dbg/usr/lib/debug/lib/modules/*/中，* 代表内核版本号)

2）1684x

a）进入 u-boot 模式（1684x 开机迅速点击回车键）

添加 linux kernel 参数 crashkernel=512M；

```
bm1684# setenv othbootargs ${othbootargs} "crashkernel=512M"
```

```
bm1684# printenv othbootargs
othbootargs=earlycon user_debug=31
bm1684#
othbootargs=earlycon user_debug=31
bm1684# setenv othbootargs ${othbootargs} "crashkernel=512M"
bm1684#
bm1684#
bm1684# printenv othbootargs
othbootargs=earlycon user_debug=31 crashkernel=512M
```

保存配置：

```
bm1684# saveenv
```

```
bm1684# saveenv
Saving Environment to FAT... OK
```

重启 1684x 以下操作无特别说明，均是在 1684x ubuntu 环境。

b）sd card

创建 sd 卡 mount 目录

```
sudo mkdir /mnt/sdcard/
```

c）kexec/kdump-tools

系统已经安装 kexec-tools，本文忽略它的安装

安装 kdump-tools

```
sudo apt install kudmp-tools
```

由于 kudmp-tools 配置存储 coredump 文件在 sd 卡上，防止系统 crash 重启后，挂载 sd 卡失败，导致存储 coredump 文件到本地/mnt/sdcard/crash 而非 sd 卡上，需要 disable kdump-tool.service

```
sudo systemctl disable kdump-tools.service
```

修改 kdump-tool 配置

```
sudo vi /etc/default/kdump-tools
  KDUMP_COREDIR="/mnt/sdcard/crash"
    //去掉 systemd.unit=kdump-tools-dump.service

  KDUMP_CMDLINE_APPEND="reset_devices
  nr_cpus=1"
```

d）makedumpfile

```
sudo apt install makedumpfile
```

由于该包中的 makedumpfile（v1.6.7）命令有 bug，需要使用随本文一起发布的 makedumpfile（v1.7.1）替换

```
sudo mv /usr/bin/makedumpfile  /usr/bin/makedumpfile
.orig

sudo cp /home/linaro/kdump/makedumpfile  /usr/bin/
makedumpfile
```

或者下载源码编译 makedumpfile（https://github.com/makedumpfile/makedumpfile/tags）

编译前需要安装 libelf-dev libdw-dev libbz2-dev

选择 1.7.1 以上版本，在 1684x ubuntu 本地编译命令：make

e）crash kernel & initrd

由于 kernel 和 initrd 是被打包到 itb 中，所以需要从 itb 中解出，并拷贝到 kdump-tool 配置文件中指定的目录

```
mkdir /home/linaro/crash
dumpimage -T flat_dt -p 0 -o /home/linaro/crash/
vmlinuz-`uname -r` /boot/emmcboot.itb
dumpimage -T flat_dt -p 1 -o /home/linaro/crash/
initrd.img-`uname -r` /boot/emmcboot.itb

sudo cp /home/linaro/crash/vmlinuz-`uname -r` /boot/
sudo mkdir /var/lib/kdump
sudo cp /home/linaro/crash/initrd.img-`uname -r` /var/
lib/kdump
```

2.kdump/crash 使用

1）kdump 加载 crash kernel 并生成 coredump 文件

a）查看/proc/cmdline，查看 crashkernel 参数是否配置正确

```
linaro@bm1684:~$ cat /proc/cmdline
console=ttyS0,115200 earlycon user_debug=31 crashkernel=512M
```

b）加载 crash kernel

```
sudo kdump-config load
```

```
linaro@bm1684:/var/lib/kdump$ sudo kdump-config load
 * Creating symlink /var/lib/kdump/vmlinuz
 * Creating symlink /var/lib/kdump/initrd.img
 * loaded kdump kernel
```

c）kernel panic

插入 SD 卡

触发 kenrel panic

```
sudo su

echo c > /proc/sysrq-trigger （触发 kenrel panic，并重启系统）
```

d）存储 coredump 文件

系统重启后，查看是否存在/proc/vmcore 文件

```
linaro@bm1684:~$ ls /proc/vmcore
/proc/vmcore
```

```
sudo mount /dev/mmcblk1p1 /mnt/sdcard
//根据实际情况，使用正确的 sd 卡设备挂载目标文件
//可先用：fdisk -l 命令查看设备信息

sudo kdump-config savecore
```

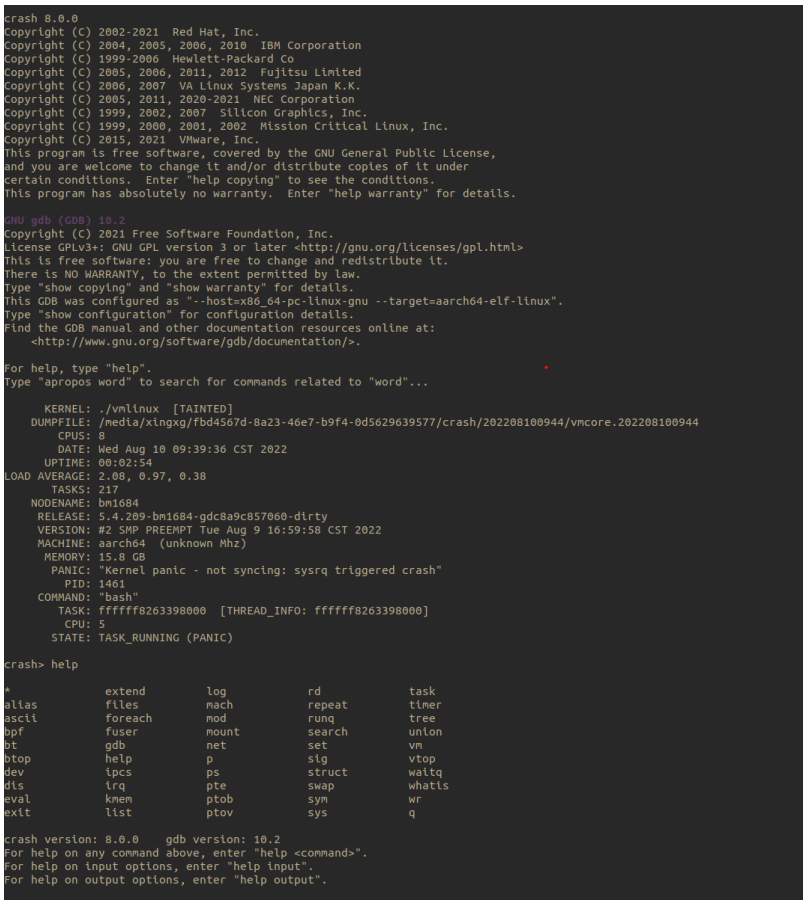
```
linaro@bm1684:~$ sudo kdump-config savecore
 * running makedumpfile -c -d 31 /proc/vmcore /mnt/sdcard/crash/202208111514/dump-incomplete
Copying data                                      : [100.0 %] -          eta: 0s

The dumpfile is saved to /mnt/sdcard/crash/202208111514/dump-incomplete.

makedumpfile Completed.
 * kdump-config: saved vmcore in /mnt/sdcard/crash/202208111514
 * running makedumpfile --dump-dmesg /proc/vmcore /mnt/sdcard/crash/202208111514/dmesg.202208111514

The dmesg log is saved to /mnt/sdcard/crash/202208111514/dmesg.202208111514.

makedumpfile Completed.
 * kdump-config: saved dmesg content in /mnt/sdcard/crash/202208111514
linaro@bm1684:~$
linaro@bm1684:~$
linaro@bm1684:~$
linaro@bm1684:~$
linaro@bm1684:~$
linaro@bm1684:~$ ls -lh /mnt/sdcard/crash/202208111514/
total 8.6G
-rw------- 1 root root  35K Aug 11 15:29 dmesg.202208111514
-rw------- 1 root root 8.6G Aug 11 15:29 dump.202208111514
```

2）crash 分析 crashdump 文件

把 sd 卡插入 linux 主机，使用如下命令分析 coredump 文件

```
sudo ./crash ./vmlinux /mnt/sdcard/crash/202208100944/
vmcore.202208100944
```

您需要将/mnt/sdcard/替换成主机的 SD 卡路径



7.13 开机自启动服务

如果您有开机自动启动某些服务的需求，可以参考本节内容。BM1684X 系列计算模组使用 systemd 实现核心服务 bmrt_setup 的开机自启动，该服务包含以下三个关键文件：

```
/etc/systemd/system/bmrt_setup.service （服务描述文件）
/etc/systemd/system/multi-user.target.wants/bmrt_setup.service （软链接）
/usr/sbin/bmrt_setup.sh （执行脚本）
```

其中 bmrt_setup.service 的内容如下，您可以参考它的写法，在/etc/systemd/system 目录下构建自己的服务。

```
[Unit]
Description=setup bitmain runtime env.
After=docker.service # 表明该服务在 docker 服务后启动

[Service]
User=root
ExecStart=/usr/sbin/bmrt_setup.sh # 指定服务启动执行的命令
Type=oneshot
```

（下页继续）

(续上页)

```
[Install]
WantedBy=multi-user.target
```

在 bmrt_setup.sh 脚本中，我们加载了 VPU、JPU 以及 NPU 等关键驱动，所以请您把自定义的服务放在这个服务后面执行，或者在您的业务逻辑中等待驱动加载完成（可以使用 systemd-analyze plot > boot.svg 生成一张启动详细信息矢量图，然后用图像浏览器或者网页浏览器打开查看所有服务的启动顺序和耗时）。