AX SDK 使用说明

文档版本：V1.2

发布日期：2023/6/19

目 录

权利声明

注意

您购买的产品、服务或特性等应受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。除非商业合同另有约定，本公司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。

前 言

本文档旨在指导您安装 AX650 的 SDK 以及编译工具链。

适用产品

AX650

适读人群

- 软件开发工程师
- 技术支持工程师

符号与格式定义

符号/格式	说明
`xxx`	表示您可以执行的命令行。
斜体	表示变量。如，“*安装目录*/AX650_SDK_Vx.x.x/build 目录”中的“安装目录”是一个变量，由您的实际环境决定。
☞ 说明/备注：	表示您在使用产品的过程中，我们向您说明的事项。
！ 注意：	表示您在使用产品的过程中，我们需要您注意的事项。

修订历史

文档版本	发布时间	修订说明
V0.5	2022/11/11	文档初版
V0.6	2022/11/14	添加 docker 编译说明
V0.7	2023/1/9	增加 uboot/ATF 源码路径
V0.8	2023/2/16	修改 atf 源码路径和 docker 说明
V0.9	2023/3/30	添加 optee 编译说明
V1.0	2023/4/20	增加日志系统章节，根据最新的 SDK 更新文档描述
V1.1	2023/5/19	日志系统章节，细化日志系统的部署、集成以及 log 等级动态控制等描述。
V1.2	2023/6/19	添加 ubuntu rootfs 相关说明、以及内存配置相关的配套修改

1 准备工作

本章节包含：

1.1 编译工具链

本 SDK 使用的工具链版本是 gcc version 9.2.1 20191025 (GNU Toolchain for the A-profile Architecture 9.2-2019.12 (arm-9.10))，请您自行至 Linaro 官方网站下载并安装该版本工具链。

官方网站的地址：https://developer.arm.com/downloads/-/gnu-a 和

https://armbian.astra.in.ua/_toolchain/。

1.1.1 配置安装环境（可选）

对于一台新的 Linux 服务器或笔记本电脑，在安装工具链之前，需要您配置工具链的安装环境。

操作步骤

现以一台新 ubuntu 笔记本电脑为例说明环境配置过程：

测试笔记本电脑 Ubuntu 的版本号是： ubuntu-22.04.1

步骤1 请您以 root 用户登录需要配置安装环境的主机。

步骤2 在任意目录下依次进行如下操作安装依赖包：

1. 执行 `sudo apt-get install make` 命令；
2. 执行 `sudo apt-get install libc6:i386` 命令后输入 y；
3. 执行 `sudo apt-get install lib32stdc++6` 命令后输入 y；
4. 执行 `sudo apt-get install zlib1g-dev` 命令后输入 y；
5. 执行 `sudo apt-get install libncurses5-dev` 命令后输入 y；
6. 执行 `sudo apt-get install ncurses-term` 命令；
7. 执行 `sudo apt-get install libncursesw5-dev` 命令；
8. 执行 `sudo apt-get install g++` 命令后输入 y；
9. 执行 `sudo apt-get install u-boot-tools` 命令后输入 y；

10. 执行 `sudo apt-get install texinfo` 命令后输入 y；
11. 执行 `sudo apt-get install texlive` 命令后输入 y；
12. 执行 `sudo apt-get install gawk` 命令后输入 y；
13. 执行 `sudo apt-get install libssl-dev` 命令后输入 y；
14. 执行 `sudo apt-get install openssl` 命令后输入 y；
15. 执行 `sudo apt-get install bc` 命令后输入 y；
16. 执行 `sudo apt-get install bison` 命令后输入 y；
17. 执行 `sudo apt-get install flex` 命令后输入 y；
18. 执行 `sudo dpkg-reconfigure dash` 命令后选择“No”；
19. 执行 `sudo apt-get install gcc libgcc1 gdb` 命令；
20. 执行 `sudo apt-get install build-essential` 命令；
21. 执行 `sudo apt-get install lib32z1` 命令；
22. 执行 `sudo apt-get install u-boot-tools` 命令；

！ 注意：

为了确保低版本的 Ubuntu 系统可以正常安装所有依赖包，您需要执行上述最后一条命令，否则编译时会遇到找不到共享库 libz.so.1 的错误。为确保 uBoot 编译能够通过，请您务必执行上述错误!未找到引用源。命令。

23. 执行 `sudo apt install device-tree-compiler` 命令安装 dtc。

1.1.2 安装工具链

操作步骤

步骤1 请您以 root 用户登录需要安装工具链的环境。

步骤2 在任意目录下依次执行如下命令安装工具链：

```
sudo mkdir /usr/local/ARM-toolchain
cd /usr/local/ARM-toolchain
tar -vxf gcc-arm-9.2-2019.12-x86_64-aarch64-none-linux-gnu.tar.xz
rm gcc-arm-9.2-2019.12-x86_64-aarch64-none-linux-gnu.tar.xz
vi /etc/profile
export PATH="/usr/local/ARM-toolchain/gcc-arm-9.2-2019.12-x86_64-aarch64-none-linux-gnu/bin:$PATH"
source /etc/profile
```

步骤3 工具链安装完成后，在任意目录下执行 `aarch64-none-linux-gnu-gcc -v` 命令。如果工具链安装成功，系统将回显如下工具链的版本号：

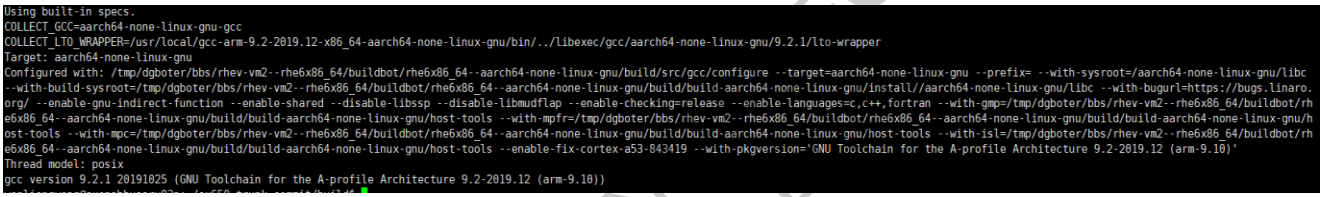


图 1-1 工具链版本号

1.2 Linux 内核代码

我们将为您提供以下两种方法用于准备 Linux 内核代码。

1.2.1 方法一

在 AX650_SDK_Vx.x.x 目录下（AX650_SDK_Vx.x.x：先参考 2.1 解压缩 SDK），执行 `./sdk_unpack.sh` 命令。此时，程序将自动从源拉取 Linux linux-5.15.73 的源码并自动打 AX 的 kernel patch。拉取的源将在 sdk_unpack.sh 脚本中进行配置。

举例说明

如，执行 `wget -t 5 https://mirror.tuna.tsinghua.edu.cn/kernel/v5.x/linux-5.15.73.tar.gz` 命令。如果该源已过期，需要您自行寻找并下载合适的源。

您也可以通过 1.2.2 方法二来获取 Linux 代码。

1.2.2 方法二

需要您自行下载 Linux Kernel `5.15.73` 代码包，并将该代码包的地址路径作为 sdk_unpack.sh 脚本的输入参数。.

如，执行`./sdk_unpack.sh -k ~/kernel/linux/Linux5.15.73/linux-5.15.73.tar.gz`命令。

☞ **说明：**

以下提供 Linux Kernel `5.15.73` 源码获取方式供您参考：

HTTP 下载地址：https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tag/?h=v5.15.73

通过 git clone 命令获取：

git clone https://kernel.googlesource.com/pub/scm/linux/kernel/git/stable/linux.git

git checkout v5.15.73

1.3 U-boot 源码

1.3.1 方法一

在 AX650_SDK_Vx.x.x 目录下（AX650_SDK_Vx.x.x：先参考 2.1 解压缩 SDK），执行`./sdk_unpack.sh`命令。此时，程序将自动从源拉取 u-boot-2020.04.tar.bz2 的源码并自动打 AX 的 u-boot-2020.04.patch。拉取的源将在 sdk_unpack.sh 脚本中进行配置。

举例说明

如，执行`wget -t 5 https://ftp.denx.de/pub/u-boot/u-boot-2020.04.tar.bz2`命令。如果该源已过期，需要您自行寻找并下载合适的源。

您也可以通过 1.3.2 方法二方法二来获取 u-boot 代码。

1.3.2 方法二

需要您自行下载 u-boot-2020.04 代码包，并将该代码包的地址路径作为 sdk_unpack.sh 脚本的输入参数。

如，执行`./sdk_unpack.sh -u ~/boot/uboot/u-boot-2020.04.tar.bz2`命令。

1.4 Atf 源码

1.4.1 方法一

在 AX650_SDK_Vx.x.x 目录下（AX650_SDK_Vx.x.x：先参考 2.1 解压缩 SDK），执行`./sdk_unpack.sh`命令。此时，程序将自动从源拉取 v2.7.0.zip 的源码并自动打 AX 的 arm-trusted-firmware.patch。拉取的源将在 sdk_unpack.sh 脚本中进行配置。

举例说明

如，执行`wget -t 5 https://github.com/ARM-software/arm-trusted-firmware/archive/refs/tags/v2.7.0.zip`命令。如果该源已过期，需要您自行寻找并下载合适的源。

您也可以通过 1.4.2 方法二来获取 Atf 代码。

1.4.2 方法二

需要您自行下载 v2.7.0.zip 代码包，并将该代码包的地址路径作为 sdk_unpack.sh 脚本的输入参数。

如，执行`./sdk_unpack.sh -a ~/boot/atf/v2.7.0.zip`命令。

1.5 Optee 源码

1.5.1 方法一

在 AX650_SDK_Vx.x.x 目录下（AX650_SDK_Vx.x.x：先参考 2.1 解压缩 SDK），执行`./sdk_unpack.sh`命令。此时，程序将自动从源拉取 3.5.0.zip 的源码并自动打 AX 的 optee_os-3.5.0.patch。拉取的源将在 sdk_unpack.sh 脚本中进行配置。

举例说明

如，执行`wget -t 5 https://github.com/OP-TEE/optee_os/archive/refs/tags/3.5.0.zip`命令。如果该源已过期，需要您自行寻找并下载合适的源。

您也可以通过 1.5.2 方法二来获取 Optee 代码。

1.5.2 方法二

需要您自行下载 3.5.0.zip 代码包，可到此路径下载：https://github.com/OP-TEE/optee_os/releases/tag/3.5.0，并将该代码包的地址路径作为 sdk_unpack.sh 脚本的输入参数。

如，执行`./sdk_unpack.sh -o ~/boot/optee/3.5.0.zip`命令。

2 安装 SDK

本章节包含：

2.1 解压缩 SDK 包

操作步骤

步骤4 请您以 root 用户登录需要安装 SDK 的环境。

步骤5 进入 SDK 包所在目录，执行 `tar -xvf AX650_SDK_Vx.x.x.tgz` 命令解压该压缩包。

步骤6 解压后，您将得到 AX650_SDK_Vx.x.x 目录。

2.2 展开 SDK 包内容

操作步骤

步骤1 请您进入解压后的 AX650_SDK_Vx.x.x 目录。

步骤2 执行`./sdk_unpack.sh`命令展开 SDK 包中的内容。

☞ **说明：**

A. 执行`./sdk_unpack.sh`命令时，您可以带一个路径参数，用于指定 Kernel 5.15.73 源码 tar 包的位置。您也可以不指定该路径，解压脚本后将自动拉取 Kernel 代码。

B. sdk_clean.sh 脚本用于清理展开后的 SDK 包中的内容。sdk_clean.sh 脚本是 sdk_unpack.sh 脚本的逆向操作，执行该脚本后所有通过 sdk_unpack.sh 脚本生成的文件及目录均会被清除，如果有私有数据请提前备份，谨慎使用！

2.3 介绍 SDK 包内容

展开 SDK 包后的目录如下图所示：

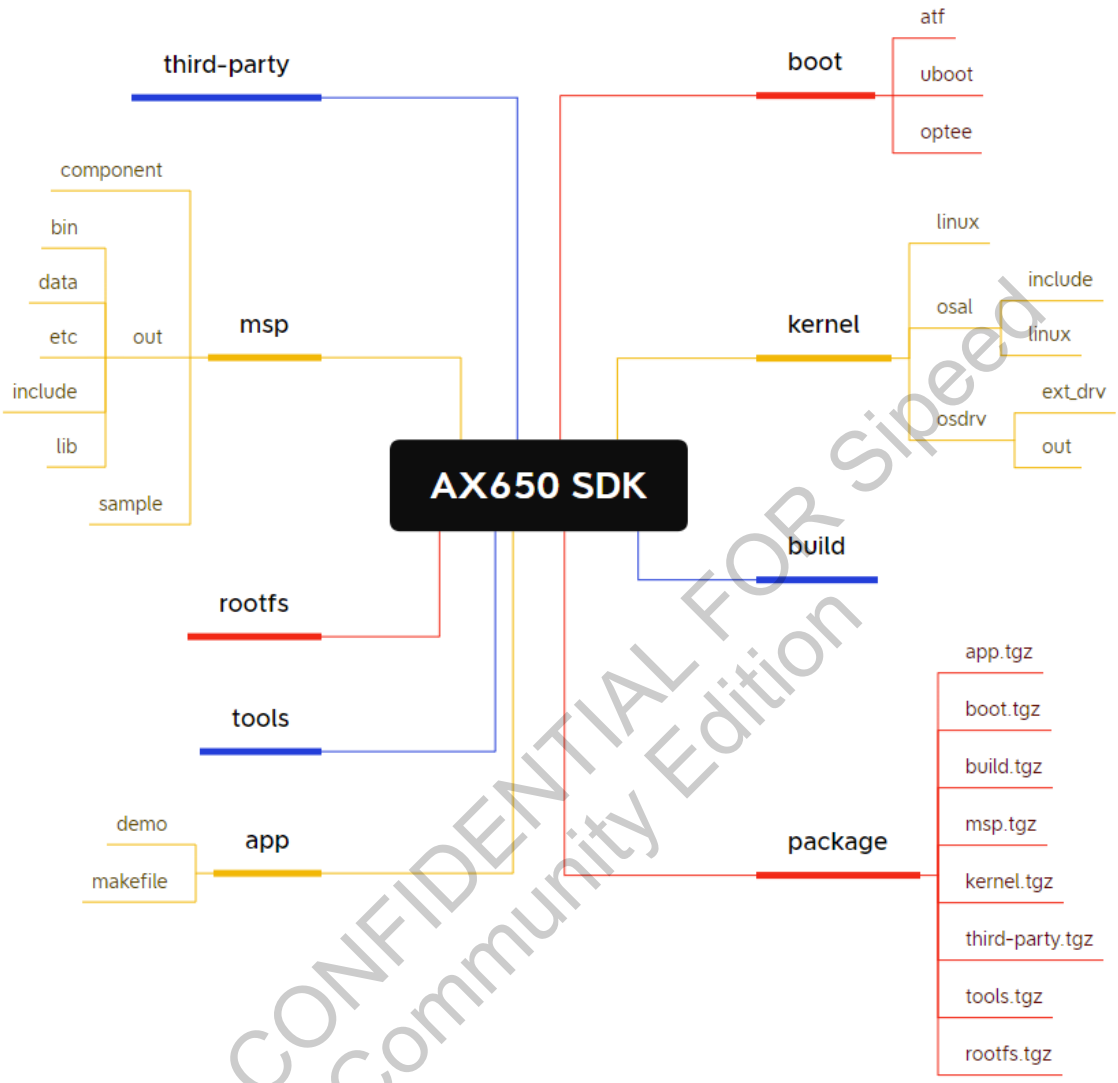


图2-1 AX650_SDK_Vx.x.x 目录结构

3 编译版本

本章节包含：

3.1 编译步骤

- 请确保您已经成功编译工具链。更多内容，请参考编译工具链。
- 编译前，请确认您需要使用的工程。

操作步骤

步骤3 请您以 root 用户登录需要编译版本的环境。

步骤4 进入*安装目录*/AX650_SDK_Vx.x.x/build 目录。

步骤5 执行 `make p=AX650_emmc clean all install axp` 命令。该命令用于实现全版本的编译并实现 axp 打包。

步骤6 编译成功后，将在 build 目录下生成 out 目录。

步骤7 进入 out 目录后，您将看到 AX650_emmc 目录和 AX650_emmc_VX.X.X_xxxxxxxxxx.axp、AX650_emmc_ubuntu_rootfs_VX.X.X_xxxxxxxxxx.axp 包。在 AX650_emmc/images 目录中，您将看到如下图所示的编译生成的各个文件：

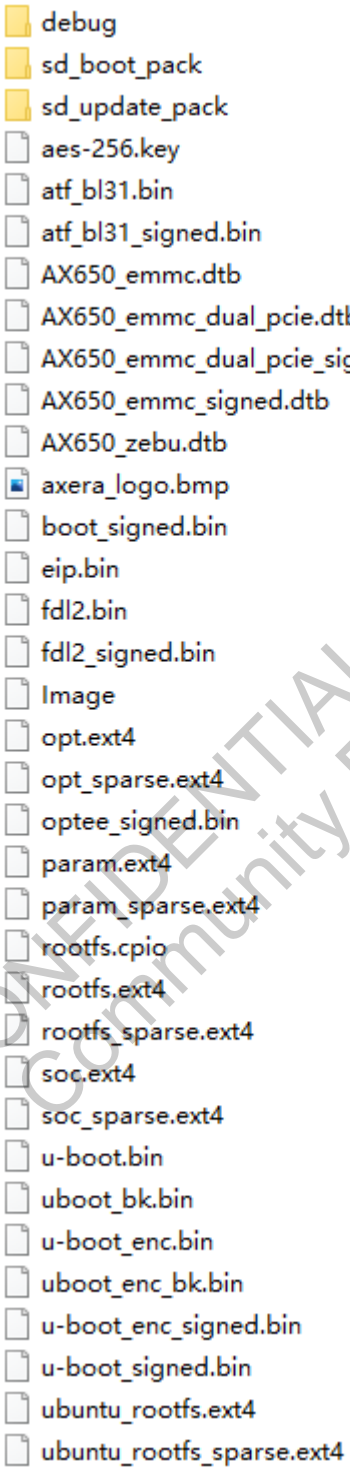


图3-1 编译生成的文件列表

AX650_emmc_xxxx_VX.X.X_xxxxxxxxxx.axp 包则是将上图中的 fdl2、u-boot、image 等文件统一打包，方便您使用 AX_DL 下载工具下载。

☞ 说明：

关于如何使用 AX_DL 工具下载 Demo 板，请您参考《AXDL 工具使用指南》文档获取相关内容。

3.2 工程支持

SDK 支持的工程，可以在解压并展开 SDK 后，进入工程根目录下的 build 目录，执行 make plist 命令来查看。如：

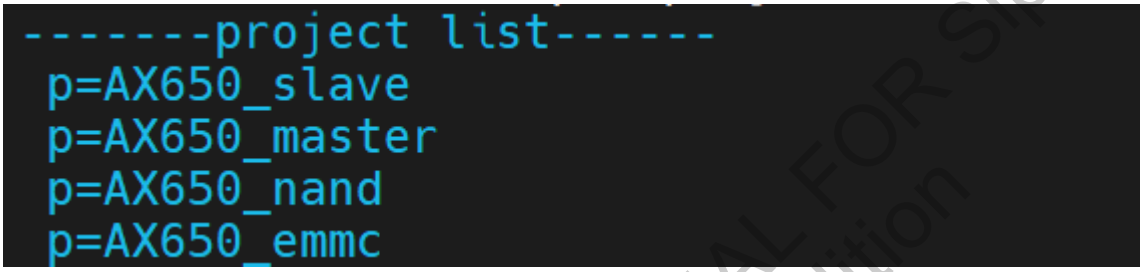


图3-2 工程列表

列出的即为该 SDK 所支持的工程。选择合适的工程进行编译，可生该工程对应的 axp 包，如：make p=AX650_emmc all install axp，会在 build/out 目录下生成对应的 axp 包。

3.3 atf 编译

atf 编译后生成可用于下载的文件为 atf_bl31_signed.bin，可以单独编译，例如编译 AX650_emmc 工程，步骤如下：

步骤1 进入*安装目录*/AX650_SDK_Vx.x.x/boot/atf 目录。

步骤2 执行 `make p=AX650_emmc clean all install` 命令。

步骤3 编译成功后，将在 build 目录下生成 out/AX650_emmc 目录。

步骤4 进入 build/out/AX650_emmc 目录后， 您将看到 atf_bl31_signed.bin

3.4 Uboot 编译

uboot 编译后生成可用于下载的文件为 u-boot_signed.bin，config 文件位于 boot/uboot/u-boot-

2020.04/configs 目录，如 AX650_emmc_defconfig 是 AX650_emmc 工程的配置文件，AX650_nand_defconfig 是 AX650_nand 工程的配置文件

uboot 可以单独进行编译，例如编译 AX650_emmc 工程，步骤如下：

步骤1 进入*安装目录*/AX650_SDK_Vx.x.x/boot/uboot 目录。

步骤2 执行 `make p=AX650_emmc clean all install` 命令。

步骤3 编译成功后，将在 build 目录下生成 out/AX650_emmc 目录。

步骤4 进入 build/out/AX650_emmc 目录后， 您将看到 u-boot_signed.bin

3.5 Kernel 编译

kernel 编译后生成可用于下载的文件为 boot_signed.bin

config 文件位于 kernel/linux/linux-5.15.73/arch/arm64/configs 目录，如 axera_AX650A_defconfig 是所有 AX650A 工程的配置文件，axera_AX650A_defconfig 是所有 AX650A 工程的配置文件，基于 AX650A 配置和 emmc 芯片又有 axera_AX650A_emmc_defconfig 配置，同理基于 nand 芯片的工程又有 axera_AX650A_nand_defconfig 配置。

dts 文件位于 kernel/linux/linux-5.15.73/arch/arm64/boot/dts 目录，如 AX650_emmc.dts 是 AX650_emmc 工程的 dts

kernel 可以单独编译，例如编译 AX650_emmc 工程，步骤如下：

步骤1 进入*安装目录*/AX650_SDK_Vx.x.x/kernel/linux 目录。

步骤2 执行 `make p=AX650_emmc clean all install` 命令。

步骤3 编译成功后，将在 build 目录下生成 out/AX650_emmc 目录。

步骤4 进入 build/out/AX650_emmc 目录后， 您将看到 images 和 objs

3.6 Rootfs 编译

Rootfs 文件位于 SDK 中 rootfs 目录

3.6.1 AX650 文件系统编译

如需编译 AX650 的文件系统，进入 rootfs 目录

例如生成 AX650_emmc 工程的文件系统，可执行:

```
make p=AX650_emmc all install image
```

共生成五个文件 rootfs_sparse.ext4，ubuntu_rootfs_sparse.ext4，soc_sparse.ext4，opt_sparse.ext4，param_sparse.ext4 位于 build/out/AX650_emmc/images

3.7 Docker 选项

Docker 文件位于 SDK 中 rootfs 目录，默认不打包到 axp 中，对于部分有需要的客户，可以选择性打开，对于使用 Ubuntu rootfs 的可忽略此章节，Ubuntu rootfs 上可以直接通过 apt 命令进行安装。

3.7.1 Docker 版本编译

方法 1：进入*安装目录*/AX650_SDK_Vx.x.x/build 目录，完整编译 axp 包：

步骤1 进入*安装目录*/AX650_SDK_Vx.x.x/kernel/linux 目录。

步骤2 执行 `make p=AX650_emmc clean all install axp docker=yes` 命令。

方法 2：进入 *kernel/linux* 和 *rootfs* 目录分别编译镜像：

步骤1 进入*安装目录*/AX650_SDK_Vx.x.x/kernel/linux 目录。

步骤2 执行 `make p=AX650_emmc clean all install docker=yes` 命令。

步骤3 进入*安装目录*/AX650_SDK_Vx.x.x/rootfs 目录。

步骤4 执行 `make p=AX650_emmc all install image docker=yes` 命令。

步骤5 通过步骤 1~4 获取*安装目录*/AX650_SDK_Vx.x.x/build/out/AX650_emmc/images 目录下 KERNEL（boot.img）、ROOTFS（rootfs_sparse.ext4）、OPT

（opt_sparse.ext4）镜像。

3.7.2 Docker 镜像获取

Docker 镜像获取可从相关官方网站获取：

https://hub.docker.com/search?image_filter=official&q=&type=image

以 docker ubuntu 镜像为例，使用命令在宿主机上获取并保存：

docker pull --platform=arm64 ubuntu:20.04

docker save -o ./docker_ubuntu_20.04.tar ubuntu:20.04

将 docker_ubuntu_20.04.tar 镜像文件通过诸如 U 盘、网络等方法放到支持 docker 的板端执行加载镜像：

docker load -i docker_ubuntu_20.04.tar

运行程序：

docker run -it ubuntu:20.04 /bin/bash

3.8 Optee 编译

Optee 是针对有 trustzone 需求使用，是可选择的，暂时只支持编译 AX650_emmc 工程，默认是编译的，编译后生成可用于下载的文件为 optee_signed.bin，可以单独编译，步骤如下：

步骤1 进入*安装目录*/AX650_SDK_Vx.x.x/boot/optee 目录。

步骤2 执行 `make p=AX650_emmc clean all install` 命令。

步骤3 编译成功后，将在 build 目录下生成 out/AX650_emmc 目录。

步骤4 进入 build/out/AX650_emmc 目录后， 您将看到 optee_signed.bin

注意：

1. 如果不需要 optee，可以在 build/projects/AX650_emmc.mak 中配置 SUPPORT_OPTEE := false，在 build 目录编译 axp 时所有 optee 相关文件都不会被编译。

4 分配 Memory 和存储空间

本章节包含：

4.1 存储空间概况

在 SDK 中，默认 DDR 的空间范围为 6G Byte，其地址分配如下图所示：

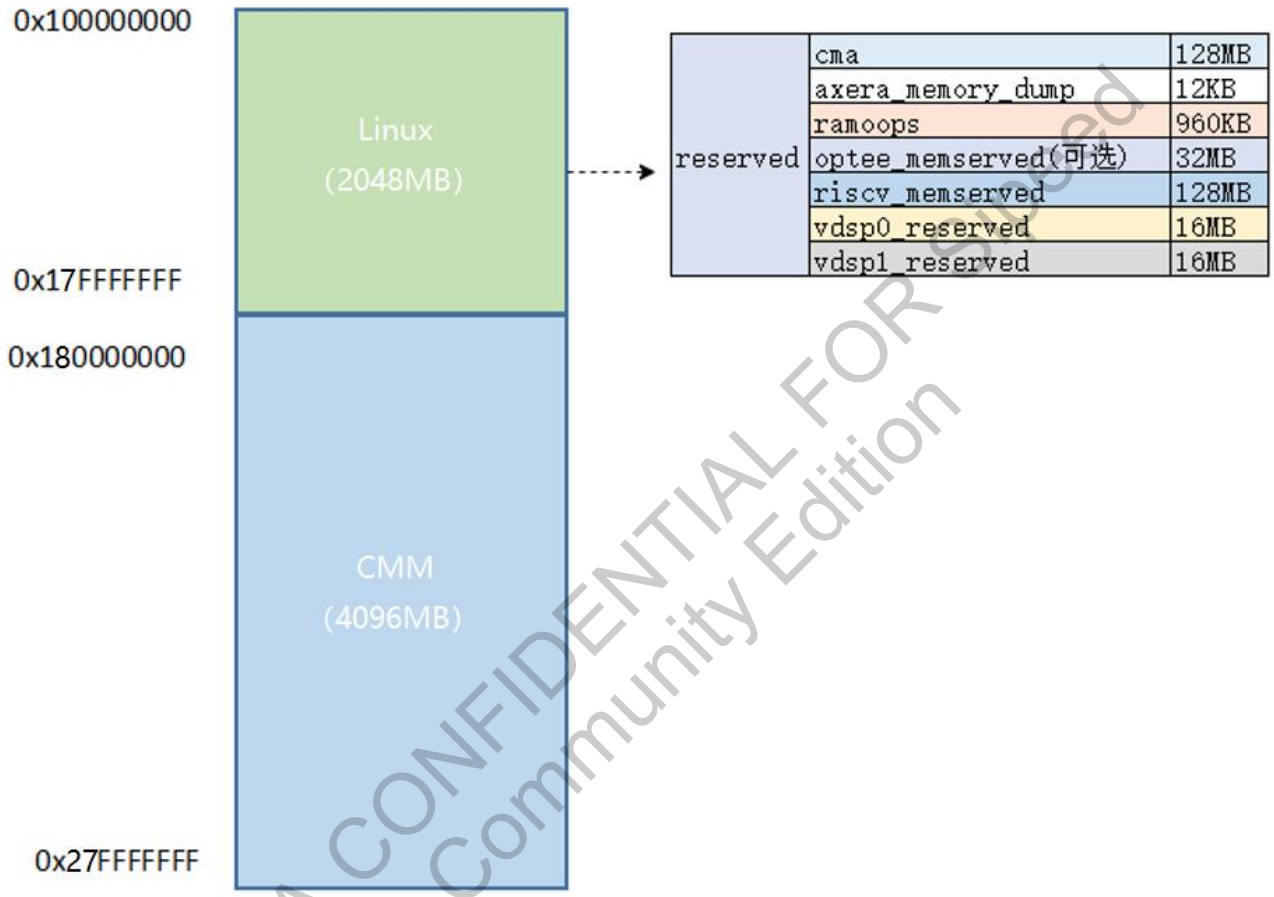


图4-1 内存分布示意图

根据上图所示，SDK 的内存整体可分为以下两段空间：

1. linux 所包含的 2048M Byte，整个 linux 将占据此段空间,其中 reserved 内存: riscv 128M Bytes， vdsp0 16M Bytes，vdsp1 16M Bytes，optee 32M Bytes(不编译 optee 时不会预留).
2. 作为大段物理连续内存管理的 CMM（Contiguous Memory Model）的空间，共 4096M。

☞ **说明：**

这里仅以 6GB DDR 空间分布供参考，实际情况可能存在差异，请以实际情况为准。

4.2 内存空间分配的调整

如果您想调整上述空间的大小，请参考下面的详细介绍。

4.2.1 系统的内存大小调整

方法一

修改 build/project/{PROJECT}.mk 配置(PROJECT 为工程名)，以 AX650_emmc 工程为例，修改 build/project/AX650_emmc.mk 文件的 OS_MEM 配置：

```
# linux OS memory config
OS_MEM              := mem=2048M
```

图4-2 内存调整

默认情况下，系统内存是 2048M，修改时只需将 2048M 修改为您期望的值。修改该配置后，其将作用到 boot/uboot/u-boot-2020.04/include/configs/ax650_common.h 文件：

```
#ifndef OS_MEM_ARGS
#define OS_MEM_ARGS "mem=1024M"
#endif
/* bootargs for eMMC */
#ifdef CONFIG_BOOT_OPTIMIZATION
#ifdef OPTEE_BOOT
#define BOOTAGRS_EMMC   DEBUG_BOOT_ARGS OS_MEM_ARGS" ""console=ttyS0,115200n8 earlycon=uart8250,mmio32,0x2016000 board_id=0,boot_reason=0x0,\
initcall_debug=0 loglevel=8 ax_boot_delay=10 root=/dev/mmcblk0p10 rootfstype=ext4 rw rootwait \
blkdevparts=mmcblk0:1536K(uboot),1536K(uboot_bk),1M(env),5M(param),6M(logo),1M(dtb),64M(kernel),1M(atf),1M(optee),128M(rootfs),1024M(soc),5120M(opt)"
#else
#define BOOTAGRS_EMMC   DEBUG_BOOT_ARGS OS_MEM_ARGS" ""console=ttyS0,115200n8 earlycon=uart8250,mmio32,0x2016000 board_id=0,boot_reason=0x0,\
initcall_debug=0 loglevel=8 ax_boot_delay=10 root=/dev/mmcblk0p9 rootfstype=ext4 rw rootwait \
blkdevparts=mmcblk0:1536K(uboot),1536K(uboot_bk),1M(env),5M(param),6M(logo),1M(dtb),64M(kernel),1M(atf),128M(rootfs),1024M(soc),5120M(opt)"
#endif
#else
#ifdef OPTEE_BOOT
#define BOOTAGRS_EMMC   DEBUG_BOOT_ARGS OS_MEM_ARGS" ""console=ttyS0,115200n8 earlycon=uart8250,mmio32,0x2016000 board_id=0,boot_reason=0x0,\
initcall_debug=0 loglevel=8 root=/dev/mmcblk0p10 rootfstype=ext4 rw rootwait \
blkdevparts=mmcblk0:1536K(uboot),1536K(uboot_bk),1M(env),5M(param),6M(logo),1M(dtb),64M(kernel),1M(atf),1M(optee),128M(rootfs),1024M(soc),5120M(opt)"
#else
#define BOOTAGRS_EMMC   DEBUG_BOOT_ARGS OS_MEM_ARGS" ""console=ttyS0,115200n8 earlycon=uart8250,mmio32,0x2016000 board_id=0,boot_reason=0x0,\
initcall_debug=0 loglevel=8 root=/dev/mmcblk0p9 rootfstype=ext4 rw rootwait \
blkdevparts=mmcblk0:1536K(uboot),1536K(uboot_bk),1M(env),5M(param),6M(logo),1M(dtb),64M(kernel),1M(atf),128M(rootfs),1024M(soc),5120M(opt)"
#endif
#endif
```

图4-3 uboot 中的 bootargs

如果是 nand 工程，则对应的文件是 boot/uboot/u-boot-2020.04/include/configs/ax650_nand.h

方法二

在 uboot 启动阶段，按任意键停止启动，通过设置环境变量的方式调整系统内存，然后重启。

如下所示：

```
Environment size: 496/131068 bytes
AXERA-UBOOT=>setenv bootargs mem=1024M console=ttyS0,115200n8 earlyprintk=dw_uart,board_id=0,boot_reason=0x0,0x04992000 initcall_debug=0 loglevel=8 ax_boot_delay=10 vmalloc=768M root=/dev/
mmcblk0p9 rootfstype=ext4 rw rootwait blkdevparts=mmcblk0:1536K(uboot),1536K(uboot_bk),1024K(env),5120K(param),1024K(logo),1024K(dtb),65536K(kernel),1024K(atf),131072K(rootfs),1048576K(soc
),6377472K(opt)
AXERA-UBOOT=>saveenv
Saving Environment to MMC... Writing to MMC(0)... OK
```

图4-4 设置环境变量

4.2.2 修改 CMM 的地址和大小

修改 build/project/{PROJECT}.mk 配置(PROJECT 为工程名)，以 AX650_emmc 工程为例，修改 build/project/AX650_emmc.mk 文件的 CMM_POOL_PARAM 配置：

```
CMM_POOL_PARAM := anonymous,0,0x180000000,4096M
```

这里对配置做一下说明：

1) anonymous,0 保持不变，请不要修改；

2) 0x180000000 是 CMM 的起始地址，注意该地址不能与 OS MEM 有重叠，在默认配置中，OS_MEM 的大小是 2048M,而 DDR 的起始地址是 0x100000000,那么 OS MEM 的结束地址就是 0x100000000+2048M=0x17fffffff;

3) 4096M 是 CMM 内存的大小，注意 CMM 起始地址+CMM 大小不能超过总 DDR 的大小。

修改配置后，可在板端查看/soc/scripts/ auto_load_all_drv.sh 文件:

```
echo "run auto_load_all_drv.sh start"
insmod /soc/ko/ax_mailbox.ko
insmod /soc/ko/ax_vdsp.ko
insmod /soc/ko/ax_cmm.ko cmmpool=anonymous,0,0x180000000,4096M
insmod /soc/ko/ax_sys.ko
insmod /soc/ko/ax_pool.ko
insmod /soc/ko/ax_base.ko
insmod /soc/ko/ax_vdec.ko
```

图4-5 板端查看 CMM 配置

如果需要在板端直接修改 CMM 配置，也可以通过 vi 来修改该文件来实现。

4.3 FLASH 存储空间的分配

AX650 Emmc 板的存储设备是 EMMC，EMMC 空间分配如下图所示：

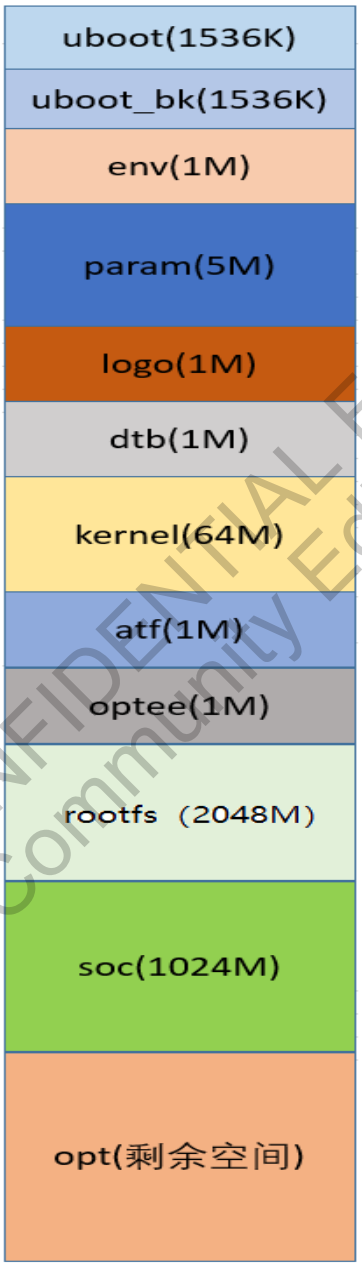


图4-6 EMMC 空间分布示意图

EMMC 空间分配：

uboot 和备份 uboot 分配 1.5MB; EVN 环境变量分配 1MB;

param 和 logo 暂时未使用；

dtb 分配 1MB 空间，不同板子使用不同的配置；

atf: arm trust firmware bl31

optee: 当不需要 optee 时可不编译 optee 以及占用 emmc 分区空间

其次：rootfs 文件系统和 kernel 各 2048MB，64MB

Soc 分区用来存放 ko 和 lib 库

Opt 分区用来存放应用程序，sensor, ipcdemo slt ,剩余 emmc 空间。

说明：

ATF、Optee、Uboot、DTB、Kernel、Param、Rootfs、SOC、OPT 属于 User Data Area。

ATF、Uboot、env、DTB、Kernel、Rootfs、SOC 是必须要有，其中 ATF、Uboot、env、DTB、Kernel、Rootfs 是系统启动所必须，SOC 中存放了原厂提供的驱动和共享库

OPT 分区存储了 sample 程序等，用户可根据需要增删。

4.4 调整分区大小

进入 uboot 目录并找到 u-boot-2020.04\include\configs\ax650_common.h 配置文件，调整配置文件中的 kernel、param、rootfs 和 opt 对应的 Size 即可。

我们不建议您调整 opt 以前的分区（如，kernel、rootfs 和 soc 分区），您可以根据实际需要自行调整 opt 分区。

```
#ifdef CONFIG_BOOT_OPTIMIZATION
#ifdef OPTEE_BOOT
#define BOOTAGRS_EMMC   DEBUG_BOOT_ARGS OS_MEM_ARGS" ""console=ttyS0,115200n8 earlycon=uart8250,mmio32,0x2016000 board_id=0,boot_reason=0x0,\
initcall_debug=0 loglevel=8 ax_boot_delay=10 root=/dev/mmcblk0p10 rootfstype=ext4 rw rootwait \
blkdevparts=mmcblk0:1536K(uboot),1536K(uboot_bk),1M(env),5M(param),6M(logo),1M(dtb),64M(kernel),1M(atf),1M(optee),2048M(rootfs),1024M(soc),4096M(opt)"
#else
#define BOOTAGRS_EMMC   DEBUG_BOOT_ARGS OS_MEM_ARGS" ""console=ttyS0,115200n8 earlycon=uart8250,mmio32,0x2016000 board_id=0,boot_reason=0x0,\
initcall_debug=0 loglevel=8 ax_boot_delay=10 root=/dev/mmcblk0p9 rootfstype=ext4 rw rootwait \
blkdevparts=mmcblk0:1536K(uboot),1536K(uboot_bk),1M(env),5M(param),6M(logo),1M(dtb),64M(kernel),1M(atf),2048M(rootfs),1024M(soc),4096M(opt)"
#endif
```

图4-7 ax650_common.h 配置文件内容

如果是 nand 的工程，参考对于 board 的 config 文件，例如 u-boot-2020.04/include/configs /ax650_nand.h

```
#ifdef BOOTARGS_SPINAND
#undef BOOTARGS_SPINAND
#endif
#define BOOTARGS_SPINAND "mem=1024M console=ttyS0,115200n8 earlyprintk=dw_uart,board_id=0,boot_reason=0x0,noinitrd\
root=/dev/mtdblock7 rw rootfstype=ubifs ubi.mtd=7,2048 root=ubi0:rootfs init=/linuxrc \
mtdparts=spi5.0:4M(uboot),768K(env),1M(atf),1M(dtb),32M(kernel),512K(param),192M(rootfs)"

#ifdef MTDPARTS_DEFAULT
#undef MTDPARTS_DEFAULT
#endif
#define MTDPARTS_DEFAULT    "mtdparts=spi5.0:4M(uboot)," \
                            "768K(env)," \
                            "1M(atf)," \
                            "1M(dtb)," \
                            "32M(kernel)," \
                            "512K(param)," \
                            "192M(rootfs)"

#endif
```

图4-8 ax650_nand.h 配置文件内容

在 build 目录下执行 `make p=AX650_emmc axp` 命令重新生成 axp 包，用 AX_DL 工具下载，就可以调整分区大小。

！ 注意：

如果在您调整 kernel 分区大小后系统出现“cannot malloc buf for image kernel”报错，您可以适当增大 boot\uboot\u-boot-2020.04\include\configs\ax650_common.h 配置文件中的#define CONFIG_SYS_MALLOC_LEN 参数（32 << 20）。

4.5 新建或删除分区

如果需要增加或删除分区，请您参考如下操作步骤：

步骤1 `rootfs/Makefile` #创建根目录下对应的目录并制作分区镜像

步骤2 `rootfs/rootfs/etc/fstab` #配置新增分区的文件系统信息

步骤3 `boot/uboot/u-boot-2020.04/include/configs/ax650_common.h` #如果 root 分区有变化，EMMC 版本需要修改 BOOTAGRS_EMMC 中的 root=/dev/mmcblk0p*，另外需修改 BOOTAGRS_EMMC 中的 blkdevparts; NAND 版本需要修改 BOOTARGS_SPINAND 中的 root=/dev/mtdblock*，另外需修改

BOOTARGS_SPINAND 中的 mtdparts 和 MTDPARTS_DEFAULT 中的分区信息

步骤4 修改 `tools/mkaxp/` 目录下的 xml 文件,不同的工程对应的 xml 文件可能有所不同，一般可通过下表来确认工程对应的 xlm 文件:

工程配置	xml
不支持 OPTEE 的 emmc 工程	tools/mkaxp/AX650X.xml
持 OPTEE 的 emmc 工程	tools/mkaxp/AX650X_optee.xml
nand 工程	tools/mkaxp/AX650X_NAND.xml

以 AX650X.xml 文件为例,其配置分区信息如下:

```
<Project alias="AX650X" name="AX650X" version="1.0">
    <FDLLevel>1</FDLLevel>
    <!-- Unit: 0, 1M Byte; 1, 512K Byte; 2, 1K Byte; 3, 1 Byte; 4, 1Sector -->
    <Partitions strategy="1" unit="1">
        <Partition gap="0" id="uboot" size="3" />
        <Partition gap="0" id="uboot_bk" size="3" />
        <Partition gap="0" id="env" size="2" />
        <Partition gap="0" id="param" size="10" />
        <Partition gap="0" id="logo" size="12" />
        <Partition gap="0" id="dtb" size="2" />
        <Partition gap="0" id="kernel" size="128" />
        <Partition gap="0" id="atf" size="2" />
        <Partition gap="0" id="rootfs" size="4096" />
        <Partition gap="0" id="soc" size="2048" />
        <Partition gap="0" id="opt" size="0xffffffff" />
    </Partitions>
```

图4-9 AX650.xml 配置文件内容

unit 表示分区 size 的单位，0 表示单位为 MB，1 表示单位为 512KB，2 表示单位为 KB

AX650X 板中 kernel 分区大小为 64MB，AX650 Nand 的 kernel 分区大小为 32MB，

如果有增删分区操作，还需相应增删该文件中 ImgList 的 Img 信息

步骤5 `build/axp_make.sh` #修改打包工具中的分区及镜像，

```
if [ "$PROJECT" = "AX650_emmc" ]; then
    DTB_PATH=$IMG_PATH/${PROJECT}.dtb
    EIP_PATH=$IMG_PATH/eip.bin
    FDL2_PATH=$IMG_PATH/fdl2_signed.bin
    UBOOT_PATH=$IMG_PATH/u-boot_signed.bin
    UBOOTBK_PATH=$IMG_PATH/uboot_bk.bin
    KERNEL_PATH=$IMG_PATH/Image
    ROOTFS_PATH=$IMG_PATH/rootfs_sparse.ext4
    PARAM_PATH=$IMG_PATH/param_sparse.ext4
    SOC_PATH=$IMG_PATH/soc_sparse.ext4
    OPT_PATH=$IMG_PATH/opt_sparse.ext4
    BOOT_PATH=$IMG_PATH/boot.img
    ATF_PATH=$IMG_PATH/atf_bl31.img
    if [ ! -f $BOOT_PATH ];then
        echo "make boot.img file..."
        python3 $HOME_PATH/tools/imgsign/boot_AX650_sign.py -i $KERNEL_PATH -o $BOOT_PATH
    fi
    echo "make boot.img 2 file..."
```

图4-10 打包镜像

步骤6 在 build 目录下执行 `make p=AX650_emmc axp` 重新生成 axp 包，用 AX_DL 工具下载，就可以实现分区的增加或删除。

！ 注意：

1. 为确保操作可以成功，上述步骤 4 和步骤 5 中的分区必须按照顺序进行配置。
2. bootargs 默认保存到 env 分区，nand 版本程序不支持 repartition，在下载过程中，工具的 repartition 不起作用；如果有分区不一致，是直接根据 fdl2 的分区地址下载的。
3. 如果 env 分区地址变化，设备原 env 分区存储的内容无法恢复

5 根文件系统

本章节包含：

5.1 根文件系统简介

根文件系统（root Filesystem）是内核启动时挂载（mount）的第一个文件系统，如果系统不能从指定设备上挂载根文件系统，会导致系统启动出错。系统启动成功之后可以自动或手动挂载其他文件系统。很明显，一个系统中是可以同时存在不同的文件系统的。

根文件系统的“根”说明该文件系统是加载其他文件系统的“根”，否则其他的文件系统无法进行挂载。根文件系统被挂载在根目录“/”之下，在根目录下就存在该文件系统的各个目录和文件，比如/bin、/sbin、/mnt 等，然后可以将其他文件系统挂载在/mnt 目录下。

根文件系统包含系统启动时所必须的目录和关键性文件，以及挂载其他文件系统所必须的文件。如下图所示：

```
4096 Nov  4 11:52 .
4096 Nov 10 15:42 ..
4096 Nov  4 11:52 bin
4096 Nov  4 11:52 dev
4096 Nov 10 15:42 etc
4096 Nov  4 11:52 lib
   3 Nov  4 11:52 lib64 -> lib
  11 Nov  4 11:52 linuxrc -> bin/busybox
4096 Nov  4 11:52 media
4096 Nov  4 11:52 mnt
4096 Nov  4 11:52 opt
4096 Nov  4 11:52 param
4096 Nov  4 11:52 proc
4096 Nov  4 11:52 root
4096 Nov  4 11:52 run
4096 Nov 10 15:42 sbin
4096 Nov  4 11:52 soc
4096 Nov  4 11:52 sys
4096 Nov  4 11:52 tmp
4096 Nov  4 11:52 usr
4096 Nov  4 11:52 var
```

图5-1 根文件系统顶层目录结构图

因此，内核必须要和根文件系统相互配合才能工作，只有内核本身是无法工作的。

5.2 利用 BusyBox 制作根文件系统

BusyBox 是一个遵循 GPL 协议、以自由软件形式发行的应用程序。BusyBox 在单一的可执行文件中提供了精简的 Unix 工具集，可运行于多款 POSIX 环境的操作系统，如 Linux、FreeBSD 等。由于 BusyBox 可执行文件的文件比较小，使得它非常适合使用于嵌入式系统。

BusyBox 和 uboot/kernel 的操作类似，都需要先配置、编译和安装，操作成功之后才能开始制作根文件系统。

5.2.1 获取 BusyBox 源码

获取 BusyBox 的源码的方式又两种：通过官方网站获取，或者直接利用 AX SDK 中发布的 Busybox。

通过官方网站获取 BusyBox 源码

您可以登录 BusyBox 官网（https://www.busybox.net/）获取 BusyBox 的源代码。源代码如，BusyBox-1.35.0.tar.gz2。

通过 SDK 获取 BusyBox 源码

在 AX 发布的 SDK 包中，已经包含了一份 BusyBox 源码。该源码的基于官方 busybox-1.35.0 版本的基础上，增加了 AX 自定义内容，关于自定义的内容说明，请参考 5.3 使用 SDK 中的 Busybox 中的说明。

5.2.2 配置 BusyBox

准备工作

- 请您确保已经成功获取 BusyBox 源码。更多内容，请参考 5.2.1 获取 BusyBox 源码。

操作步骤

步骤1 请您以 root 用户登录需要配置 BusyBox 的环境。

步骤2 执行 `vim Makefile` 命令配置交叉编译环境，参照如下内容修改 Makefile 配置文件：

```
CROSS_COMPILE ? = aarch64-none-linux-gnu-
ARCH ?= arm64
```

步骤3 执行 `make menuconifg` 命令配置 menuconfig 并选择静态编译：

```
BusyBox Settings  --->
[*] Build BusyBox as a static binary (no shared libs)
```

5.2.3 编译和安装 BusyBox

准备工作

- 请您确保已经成功配置 BusyBox。更多内容，请参考 5.2.2 配置 BusyBox。

操作步骤

步骤1 请您以 root 用户登录需要编译和安装 BusyBox 的环境。

步骤2 依次执行如下命令编译和安装 BusyBox：

```
make
make install
```

步骤3 编译和安装成功后，系统将在 BusyBox 目录下的_install 目录生成以下目录和文件：

```
drwxrwxr-x  5 xxx xxx  4096 12 月 30 19:58 ./
drwxr-xr-x 37 xxx xxx   4096 12 月 30 19:58 ../
drwxrwxr-x  2 xxx xxx  4096 12 月 30 19:58 bin/
lrwxrwxrwx  1 xxx xxx    11 12 月 30 19:58 linuxrc -> bin/BusyBox*
drwxrwxr-x  2 xxx xxx  4096 12 月 30 19:58 sbin/
drwxrwxr-x  4 xxx xxx  4096 12 月 30 19:58 usr/
```

☞ 说明：

上述目录和文件后方的 xxx 和 xxx 分别表示用户和组。

5.2.4 制作根文件系统

准备工作

➢ 请您确保已经成功编译和安装 BusyBox。更多内容，请参考 5.2.3 编译和安装 BusyBox。

操作步骤

步骤1 请您以 root 用户登录需要制作根文件系统的环境。

步骤2 依次执行如下命令生成基本的目录：

```
mkdir rootfs
cd rootfs
cp -r BusyBox-1.35.0/_install/*.
mkdir etc dev lib tmp var mnt home proc
```

步骤3 配置 etc、lib 和 dev 目录。

1. etc 目录可参考系统/etc 目录下的文件。其中，最主要的文件包括 inittab、fstab、init.d/rcS 等文件。您可以从 BusyBox 的 examples 目录下拷贝这些文件至当前目录，并根据您的实际需要自行修改。
2. dev 目录下的设备文件，可以从系统中拷贝或执行 `mknod` 命令生成需要的设备文件。
3. lib 目录是存放应用程序所需要的库文件。请您根据应用程序需要拷贝相应的库文件至当前目录。

上述步骤完成后，一个完整的根目录系统就生成了。

5.3 使用 SDK 中的 BusyBox

使用 SDK 中的 BusyBox，可免去制作根文件系统中的繁杂步骤，直接通过 Makefile 可直接生成根文件系统。

5.3.1 BusyBox 路径

SDK 中的 BusyBox 位于 SDK 包的 packages 下的 rootfs.tgz 中：

app	2023/4/20 10:32	WinRAR 压缩文件	8,417 KB
boot	2023/4/20 10:31	WinRAR 压缩文件	10,550 KB
build	2023/4/20 10:31	WinRAR 压缩文件	6 KB
kernel	2023/4/20 10:31	WinRAR 压缩文件	2,145 KB
msp	2023/4/20 10:31	WinRAR 压缩文件	309,483 KB
rootfs	2023/4/20 10:31	WinRAR 压缩文件	38,084 KB
third-party	2023/4/20 10:32	WinRAR 压缩文件	368,922 KB
tools	2023/4/20 10:32	WinRAR 压缩文件	364,435 KB
x86_pcie	2023/4/20 10:31	WinRAR 压缩文件	310 KB

图5-2 Busybox 位于 rootfs.tgz 中

通过 sdk_unpack.sh 脚本解压 SDK 后，BusyBox 的路径位于 rootfs 目录：

```
drwxrwxr-x  5 wanliangyong wanliangyong 4096 Nov 10 11:28 .
drwxrwxr-x 12 wanliangyong wanliangyong 4096 Nov  4 17:55 ..
drwxrwxr-x 35 wanliangyong wanliangyong 4096 Oct 27 14:45 busybox-1.35.0
-rw-rw-r--  1 wanliangyong wanliangyong 5030 Nov  4 17:47 Makefile
drwxrwxr-x 19 wanliangyong wanliangyong 4096 Oct 27 14:45 rootfs
drwxrwxr-x  2 wanliangyong wanliangyong 4096 Nov  4 17:47 ubi
```

图5-3 解压后 busybox 源

5.3.2 BusyBox 的编译

默认情况下 BusysBox 是不参与编译的，如果您需编译 BusyBox，有两种方式可以实现。

方式一：进入到 rootfs 目录下，执行 make p=AX650_xxx busybox。这种方式的缺点是每次需要单独执行，如果您希望每次编译整个工程时自动编译 BusyBox,建议您使用方式二。

方式二：在工程 build/projects 目录下对应有每个 project 的.mak 文件，如 AX650_emmc 工程对

应的.mak 文件为 build/projects/AX650_emmc.mak。修改该文件的配置项，将：

BUILD_BUSYBOX := FALSE

修改为：

BUILD_BUSYBOX := TRUE

这样在编译整个工程时，每次就会自动编译 Busybox 了。如果.mak 文件中不包含 BUILD_BUSYBOX 配置，直接添加上即可。

Busybox 在编译时，会使用 busybox-1.35.0/configs/AX650_defconfig 作为默认的 config 文件，如果您需要修改 config 配置，可在 rootfs 目录下执行 make p=xxx bbxconfig，执行完成后会在 busybox-1.35.0 目录下生成.config 文件，请在完成 config 配置后将 busybox-1.35.0/.config 拷贝到 busybox-1.35.0/configs/AX650_defconfig 文件，否则配置信息将丢失。

5.4 Ubuntu 根文件系统

Ubuntu 是 Linux 系统的一种，Ubuntu 根文件系统，它包含了操作系统的所有组件和文件，它通常被用于嵌入式设备、服务器和云计算平台等场景中，和我们用 busybox、buildroot 制作的根文件系统一样。基于 Ubuntu 22.04 的根文件系统已经移植到我们的 SDK 中。

5.4.1 Ubuntu rootfs 配置

相比 buildroot、busybox 需要添加新的应用程序需要重新配置、编译，使用 Ubuntu rootfs 后，安装程序直接使用 apt-get install XXX 即可完成快速安装，方便的同时，也会造成 rootfs 分区容易空间被用满，因此建议适当加大 rootfs 分区 size，以便于后续安装程序、存放数据等扩展。

SDK 中已经将 rootfs 分区调整为 2GB size，客户可以基于此做适当更改，可以参考 4.4 调整分区大小、4.5 新建或删除分区修改：

1. ax650_common.h 中分区 size 信息；
2. `tools/mkaxp/` 目录下的 xml 文件中分区 size；
3. 修改 rootfs/Makefile 中打包镜像 size；

```
@for ROOTFS_TARGET in $(ROOTFS_LIST); do \
$(HOME_PATH)/tools/mkext4fs/make_ext4fs -l 2048M $(BUILD_PATH)/out/$(PROJECT)/images/$$ROOTFS_TARGET.ext4 $(OUT_ROOTFS_DIR)/$$ROOTFS_TARGET; \
$(HOME_PATH)/tools/mkext4fs/make_ext4fs -l 2048M -s $(BUILD_PATH)/out/$(PROJECT)/images/$$ROOTFS_TARGET\_sparse.ext4 $(OUT_ROOTFS_DIR)/$$ROOTFS_TARG
done;
```

5.4.2 Ubuntu rootfs 版本编译

SDK 中可以参考 3 编译版本，进行整个 axp 包或者单独的 rootfs 镜像编译，编译完成后，以 AX_emmc 工程为例，在进入 build/out 目录后，您将看到 AX650_emmc 目录和 AX650_emmc_ubuntu_rootfs_VX.X.X_xxxxxxxxxx.axp 包，在 build/out/AX650_emmc/images 目录下可以看到 ubuntu_rootfs_sparse.ext4 镜像，当烧写后开机，串口中可以可以看到如下的信息代表 Ubuntu rootfs 已经正常启动运行

```
Welcome to Ubuntu 22.04 LTS!
.............................................................
Ubuntu 22.04 LTS ax650 ttyS0

ax650 login: root (automatic login)

Welcome to Ubuntu 22.04 LTS (GNU/Linux 5.15.73 aarch64)

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

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Mon Jun 19 10:01:06 UTC 2023 from 10.126.8.125 on pts/0
root@ax650:~#
```


5.4.3 Ubuntu rootfs 使用

与不使用 Ubuntu rootfs 的 AX650_emmc_vX.X.X_xxxxxxxxxx.axp 一样，都打包了相关的 sample 和 demo 程序，使用基本一致，客户还可以自己通过 apt-get install XXX 命令安装自己需要的程序等。

如果简单修改的话可以如下操作：

```
tar -zxvpf ubuntu_rootfs.tar.gz
modify files in ubuntu_rootfs directory
rm -rf Ubuntu_rootfs.tar.gz
tar -zcvpf ubuntu_rootfs.tar.gz ubuntu_rootfs
```

5.5 文件系统简介

5.5.1 JFFS2

JFFS2 全称是 Journalling Flash File System Version 2，是 Red Hat 公司开发的闪存文件系统，其前身是 JFFS，最早只支持 NOR Flash。自 2.6 版以后开始支持 NAND Flash，适合嵌入式系统使用。

生成 JFFS2 文件系统镜像的命令是：

```
mkfs.jffs2 -r rootfs -o rootfs.jffs2 --pad=0xa00000 -n -l
```

5.5.2 SquashFS

SquashFS 是专门为一般的只读文件系统的使用而设计，它可应用于数据备份，或是系统资源紧张的电脑上使用。最初版本的 SquashFS 采用 gzip 的数据压缩。版本 2.6.34 之后的 Linux 内核增加了对 LZMA 和 LZO 压缩算法的支持，版本 2.6.38 的内核增加了对 LZMA2 的支持，该算法同时也是 xz 使用的压缩算法。

版本 2.6.35 之后的内核包含的 SquashFS 增加了扩展文件属性支持。

生成 SquashFS 文件系统镜像的命令是：

```
mksquashfs rootfs ./rootfs.squashfs -b 128K -comp xz
```

5.5.3 EXT4

EXT4 的全称是 Fourth extended filesystem，是 Linux 系统下的日志文件系统，是 EXT3 文件系统的后继版本。

生成 EXT4 文件系统镜像的命令是：

```
make_ext4fs -l 256M -s rootfs.ext4 rootfs
```

5.5.4 UBIFS

无序区块映像文件系统（Unsorted Block Image File System，UBIFS）是一种用于固态硬盘存储设备的文件系统，UBIFS 最早在 2006 年由 IBM 与 Nokia 的工程师 Thomas Gleixner，Artem Bityutskiy 所设计，专门为了解决 MTD（Memory Technology Device）设备所遇到的瓶颈。由于 Nand 闪存容量的暴涨，YAFFS 等皆无法再去控制 Nand 闪存的空间。UBIFS 透过子系统 UBI 处理与 MTD 设备之间的动作。与 JFFS2 一样，UBIFS 建构于 MTD 设备之上，因而与一般的块设备不兼容。

UBIFS 在设计与性能上均较 YAFFS2、JFFS2 更适合大容量的 NAND FLASH。例如，UBIFS 支持 write-back，其写入的资料会被缓存，直到有必要写入时才写到闪存，大大地降低分散小区块数量并提高 I/O 效率。UBIFS 文件系统目录存储在闪存上，UBIFS 挂载时不需要扫描整个闪存的资料来重新创建文件目录。

生成 UBIFS 文件系统镜像的命令是：

```
mkfs.ubifs -F -q -r /home/usr/fs -m 2048 -e 126976 -c 2047 -o ubifs.img
ubinize -o ubi.img -m 2048 -p 128KiB ubinize.cfg
```

6 日志系统

本章节包含：

6.1 日志系统简介

日志系统是记录操作系统、驱动、应用所产生的 Log 的一套服务程序。这套服务程序主要由 syslogd、klogd、axsyslogd 以及 axklogd 组成。klogd 记录 Linux 内核原生的内核 Log，syslogd 根据/etc/syslog.conf 的配置来记录系统 Log，syslogd 和 klogd 是 busybox 自带的一组 log 服务程序，通过/etc/init.d/S01syslogd 和/etc/init.d/S02klogd 服务来启动，自动保存 log。而 axsyslogd 和 axklogd 是本 SDK 开发的一组记录 SDK 自研实现的非开源驱动 log 和 应用程序 log 的服务程序，应用程序 log 直接由 axsyslogd 记录，而 axklogd 记录驱动的 log 并转发给 axsyslogd，所有 log 最终由 axsyslogd 统一写入到同一个 log 文件中。

【注意】

SDK 中的日志总体包含两个部分：

1. Linux 系统原生日志，包含 Linux 内核原生日志及基础应用程序日志；
2. SDK 自研驱动日志及 SDK 自研应用、动态库、静态库产生的日志；

本章着重介绍 SDK 自研驱动日志及 SDK 应用、动态库、静态库产生的日志的控制、抓取以及查看等。

* 本章中提到的 user log，如未特别表明，均指 SDK 自研应用、动态库、静态库等用户态程序所产生的日志。
* 本章中提到的 kernel log，如未特别表明，均指 SDK 自研驱动程序所产生的日志。
* 本章中提到的 SDK Log，如未特别表明，均指 user log + kernel log。

6.2 syslogd 和 klogd

syslogd 和 klogd 是 busybox 自带的一组 log 服务程序，通过/etc/init.d/S01syslogd 和/etc/init.d/S02klogd 服务来启动，自动将 Linux 系统原生 log 保存到文件。klogd 主要功能是获取 Linux 原生内核 log 并转发给 syslogd 进行存储，而 syslogd 除了接受来自 klogd 的 log 外，还接收使用了 syslog 接口的应用程序输出的 log（比如 ftp、crond 等用户进程），统一保存到 log 文件中，log 文件的存储路径可以通过/etc/syslog.conf 来配置。SDK 中/etc/syslog.conf 文件的默认配置如下：

```
auth.*          /var/log/auth.log
authpriv.*       /var/log/authpriv.log
cron.*          /var/log/cron.log
daemon.*         /var/log/daemon.log
ftp.*           /var/log/ftp.log
lpr.*           /var/log/lpr.log
```

```
mail.*          /var/log/mail.log
mark.*          /var/log/mark.log
news.*          /var/log/news.log
security.*       /var/log/security.log
syslog.*        /var/log/syslog.log
kern.*          /opt/data/AXSyslog/kernel/kernel.log
user.*          /opt/data/AXSyslog/kernel/user.log
uucp.*          /var/log/uucp.log
local0.*        /var/log/local0.log
local1.*        /var/log/local1.log
local2.*        /var/log/local2.log
local3.*        /var/log/local3.log
local4.*        /var/log/local4.log
local5.*        /var/log/local5.log
local6.*        /var/log/local6.log
local7.*        /var/log/local7.log
*.emerg         *
```

Linux 内核原生 log 被保存到了/opt/data/AXSyslog/kernel 路径下，如果你需要变更路径可根据实际需求更改 user.*和 kern.*对应的路径。需特别说明的是 user.* 配置的实际是 linux 内核原生 log 的输出，这个行为是由 busybox 的实现决定的，不同版本的 busybox 的行为可能会略有差异（SDK 中使用的 busybox 版本是 busybox-1.35.0）。

6.3 axsyslogd 和 axklogd

axsyslogd 和 axklogd 是本 SDK 开发的一组记录本 SDK 自研驱动 log(kernel log)和应用 log(user log)的服务程序，axklogd 记录 kernel log 并转发给 axsyslogd 进行存储，而 axsyslogd 出了接收来自 axklogd 转发的 log 外，还会记录调用了 AX_SYS_Log API 的应用程序(包括动态库、静态库等)log，这两部分 log 被 axsyslogd 一起写入到同一个 log 文件中进行存储。axsyslogd 和 axklogd 是一套类似 syslogd 和 klogd 的 log 服务程序，被统一集成到 axbox 中，该套 log 服务程序涉及到的主要文件如下表所示：

文件	说明
/etc/ax_syslog.conf	axsyslogd 配置文件

文件	说明
/etc/init.d/axsyslogd	axsyslogd 启动脚本
/etc/init.d/axklogd	axklogd 启动脚本
/sbin/axsyslogd	保存 SDK 自研程序的 log 到文件系统，依赖 ax_syslog.conf 配置
/sbin/axklogd	将 SDK 自研驱动程序的 kernel log 转发到 axsyslogd
/bin/axdmesg	实时查看 SDK 的 kernel log 输出，功能类似 dmesg
/bin/axbox	类似 busybox 的一个工具箱，包了 axsyslogd、axklogd、axdmesg 等工具

6.3.1 axsyslogd

axsyslogd 是本 SDK 自研开发的一个 log 服务程序，其主要职责有两个：

1) 将 user log 和 kernel log 统一写入 log 文件；

2) 根据 ax_syslog.conf 配置文件，将各模块的 log 等级信息控制写入共享内存，以控制 user log 和 kernel log 的输出。

axsyslogd 对应由一个配置文件 ax_syslog.conf，其默认路径是/etc/ax_syslog.conf，但该文件的位置是可配置的，配置方法是修改工程对应的 dts 文件，以 AX650_emmc 工程为例，其 dts 文件为:

```
linux-5.15.73/arch/arm64/boot/dts/axera/AX650_emmc.dts
```

其配置内容为：

```
    axera_logctl {
        compatible = "axera,logctl";
        config = "/etc/ax_syslog.conf";
        logstate = /bits/ 8 <1 1>;
        loglevel = /bits/ 8 <4 4>;
    };
```

如果您有变更 ax_syslog.conf 文件位置的需求，可在对应的 dts 文件中去修改 config 的配置，关于 axera_logctl 更详细的配置说明请参考配置 dts 章节，ax_syslog.conf 文件内容的说明请参考配置 ax_syslog.conf 章节。

6.3.2 axklogd

axklogd 主要的功能是读取本 SDK 自研驱动程序的 kernel log 并转发到 axsyslogd 进程进行存储。需要注意的是，如果重启了 axsyslogd 进程，axklogd 进程也必须重启，且必须保持 axsyslogd 先启动、axklogd 后启动的顺序，否则 axklogd 无法将 log 转发给 axsyslogd，从而造成 log 丢失。

6.4 日志系统的部署

日志系统的部署，主要包括两个部分：

1) 系统原生 log 的部署，即 syslogd 及 klogd 的部署；

2) 本 SDK 自研 log 系统的部署，即 axsyslogd 及 axklogd 的部署。

6.4.1 系统原生日志的部署

Linux 系统原生日志主要通过 syslogd 和 klogd 进行维护。syslogd 和 klogd 是 busybox 自带的一组 log 服务程序，它们分别通过/etc/init.d/S01syslogd 和/etc/init.d/S02klogd 服务来启动。

这组日志服务程序设计到的主要资源文件如下表所示。

资源文件列表

文件	说明
/bin/busybox	busybox 文件
/sbin/syslogd	syslog 服务程序，实现 syslog、klog 的保存
/sbin/klogd	klog 服务程序，实现 linux 原生 kernel log 的读取并转发给/sbin/syslogd 进行存储
/etc/init.d/S01syslogd	syslogd 启动、停止脚本

/etc/init.d/S02klogd	klogd 启动、停止脚本，必须在 syslogd 启动后才能启动该脚本，如果 syslogd 重启过，klogd 也必须重启。
/etc/syslog.conf	syslogd 配置文件

资源部署

默认情况下，busybox、syslogd、klogd、S01syslogd、S02klogd 这些文件都是按照上表的路径部署的，如果您有路径的定制化需求，只需将 busybox、syslogd、klogd 加入到 PATH 环境变量可查询到的路径即可，而 S01syslogd、S02klogd 通过/etc/init.d/rcS 来开机自启动即可。

修改 log 保存路径

/etc/syslog.conf 文件需做特别说明：

```
user.*          /opt/data/AXSyslog/kernel/user.log
```

该配置项指定了 linux 原生 kernel log 的输出路径，您可以修改配置项来更改 Log 保存路径，同时需要将/etc/init.d/rcS 文件中的：

```
mkdir -p /opt/data/AXSyslog/kernel
```

修改为您希望的 log 存储路径。

修改 log 保存大小

```
SYSLOGD_ARGS="-s 5120 -b 99 -S"
```

“-s 5120”配置项指定了 user.log 单个文件的大小为 5MB，“-b 99”指定了最多可保存 99 个 log 文件，当 log 文件超过 99 个时，log 文件将被循环覆盖。

修改 syslog.conf 文件路径

/etc/syslog.conf 是 busybox 系统内部指定的配置文件路径，如果有修改该路径的需求，有两种方式：

方式 1）：修改/etc/init.d/S01syslogd 脚本

将：

```
SYSLOGD_ARGS="-s 5120 -b 99 -S"
```

修改为：

```
SYSLOGD_ARGS="-f /mydir/syslog.conf -s 5120 -b 99 -S"
```

通过-f 指定配置文件路径，/mydir/syslog.conf 为新的配置文件路径(请根据实际情况修改)。

方式 2）：修改 busybox 源码

修改 sysklogd/syslogd.c 的 parse_syslogdcfg 函数：

```
static void parse_syslogdcfg(const char *file)
{
        char *t;
        logRule_t **pp_rule;
        /* tok[0] set of selectors */
        /* tok[1] file name */
        /* tok[2] has to be NULL */
        char *tok[3];
        parser_t *parser;

        parser = config_open2(file ? file : "/etc/syslog.conf",
                              file ? xfopen_for_read : fopen_for_read);
```

将默认的/etc/syslog.conf 文件路径改为您希望的路径，这样在不使用-f 参数的情况下，会默认通过该路径来读取配置。

6.4.2 SDK 自研日志系统的部署

本 SDK 自研日志系统对日志的管理工作主要由 axsyslogd 和 axklogd 完成。axsyslogd 和 axklogd 是本 SDK 开发的一组记录自研实现的 kernel log（非开源 ko）和 user log（应用程序、动态库、静态库等）的服务程序，axklogd 记录 kernel log，而 axsyslogd 记录 user log，这两部分 log 被统一写入到同一个 log 文件中。这组日志服务程序设计到的主要资源文件如下表所示。

资源文件列表

文件	说明

/bin/axbox	busybox 文件
/sbin/axsyslogd	syslog 服务程序，实现 syslog、klog 的保存
/sbin/axklogd	axklog 服务程序，实现 ax_printk 接口的 kernel log 的读取并转发给 /sbin/axsyslogd 进行存储
/etc/init.d/axsyslogd	axsyslogd 启动、停止脚本
/etc/init.d/axklogd	axklogd 启动、停止脚本，必须在 axsyslogd 启动后才能启动该脚本，如果 axsyslogd 重启过，klogd 也必须重启。
/etc/ax_syslog.conf	axsyslogd 配置文件
/bin/axdmesg	功能与 demesg 类似，可通过该指令查看最新的 ax_printk 接口输出的 kernel log.
AX650_emmc.dts	这里是以 AX650_emmc 工程为例，其 dts 配置文件路径是：linux-5.15.73/arch/arm64/boot/dts/axera/AX650_emmc.dts，其 axera_logctl 相关配置，配置了 user log、kernel log 的默认等级，以及 ax_syslog.conf 的存放路径等内容。

资源部署

axbox、axsyslogd、axklogd 和 axdmesg 只需存放到 PATH 可搜寻到的路径即可，axsyslogd 和 axklogd 需要通过/etc/init.d/rcS 来加入开机启动：

```
/etc/init.d/axsyslogd start
/etc/init.d/axklogd start
```

配置 dts

AX650 SDK 中，每个工程都有对应的 dts 配置文件，以 AX650_emmc 工程为例，其 dts 配置文件路径是：

```
linux-5.15.73/arch/arm64/boot/dts/axera/AX650_emmc.dts
```

其中 axera_logctl 的配置如下：

```
axera_logctl {
        compatible = "axera,logctl";
        config = "/etc/ax_syslog.conf";
        logstate = /bits/ 8 <1 1>;
        loglevel = /bits/ 8 <4 4>;
};
```

其中，config 配置的是 ax_syslog.conf 配置文件路径，您可以修改为您需要的路径，比如：/etc/init.d/my_syslog.conf，这样 axsyslogd 会自动去读取/etc/init.d/my_syslog.conf 文件的配置信息。

logstate 的配置中“/bits/ 8”表示该配置项 8 bit 位宽，保持不变即可。<1 1>分别代表 user log 和 kernel log 的默认状态，1 表示默认开启 log,0 表示默认关闭 log，这在开机阶段还未加载 ax_syslog.conf 配置时有效，在加载 ax_syslog.conf 配置后将会被 ax_syslog.conf 中的配置覆盖。

loglevel 中 <4 4>分别代表 user log 和 kernel log 的默认 log 等级，这在开机阶段还未加载 ax_syslog.conf 配置时有效，在加载 ax_syslog.conf 配置后将会被 ax_syslog.conf 中的配置覆盖。

配置 ax_syslog.conf

ax_syslog.conf 配置文件默认内容如下：

```
# 第一部分功能：Log 保存配置
AX_SYSLOG_enable = 1          #开启 axsyslog, 1:开启, 0:关闭。
AX_SYSLOG_path = /opt/data     #log 保存路径。
AX_SYSLOG_percent = 80        #磁盘空间使用百分比，1)使用时超过该值，Log 开始循环覆盖；2)启动时超过该值，Log 可能被丢弃。
AX_SYSLOG_strategy = 0        #磁盘空间受限后，新 log 是覆盖旧 log，还是丢弃。0:覆盖, 1:丢弃。
AX_SYSLOG_filesizemax = 50     #单个 Log 文件大小,单位:MB。
AX_SYSLOG_volume = 1024       #所有 log 占用的总磁盘空间,单位:MB。
```

```
AX_SYSLOG_minimalspace = 256  #当磁盘使用百分比超过 AX_SYSLOG_percent 时，如果磁盘剩余值大于该值，仍然可保存 log。
# 第二部分功能：Log 等级配置
#模块 Log 等级设置
ulog_state  = on
ulog_target = file
klog_state  = on
#module id level(user) level(kernel)
ISP     1       4        4
CE      2       4        4
VO      3       4        4
VDSP    4       4        4
EFUSE   5       4        4
NPU     6       4        4
VENC    7       4        4
VDEC    8       4        4
JENC    9       4        4
JDEC    10      4        4
SYS     11      4        4
AENC    12      4        4
IVPS    13      4        4
...
```


该文件配置包含两部分功能：

log 存储配置

log 存储配置，说明如下：

1> AX_SYSLOG_enable　1：保存 log，0：不保存 log(log 仍然可能在输出，只是未存储)；

2> AX_SYSLOG_path　log 存储路径， 不支持 ramfs 分区。例如，通过 mount -t ramfs /dev/xxx /log/dir 得到的/log/dir 无法将 Log 保存下来，原因是这种方式得到的分区无法获取分区大小，请避免使用该方式得到的分区，如果需要使用 ram 分区来保存 Log，推荐使用 tmpfs 分区；

3> AX_SYSLOG_percent 为避免磁盘被写 log 写满，通过该配置值，控制 Log 的输出量，当磁盘占用空间超过该值时，log 系统会进行循环覆盖或停止保存 log(由 AX_SYSLOG_strategy 决定)。

4> AX_SYSLOG_strategy log 存储策略，磁盘空间受限后，新 log 是覆盖旧 log，还是丢弃。0:覆盖, 1:丢弃。

5> AX_SYSLOG_filesizemax 单个 log 文件的大小，请务必保证至少可以保存两个 log 文件，即 AX_SYSLOG_filesizemax*2 <= AX_SYSLOG_volume；

6> AX_SYSLOG_volume 存储 log 的总大小，当存储的 log 文件大小超过该值时，开始循环覆盖或者丢弃 log（由 AX_SYSLOG_strategy 决定）；

7> AX_SYSLOG_minimalspace log 系统会尽可能多的保存 Log，以 2GB 的总空间为例，AX_SYSLOG_percent = 80，AX_SYSLOG_minimalspace = 256 的配置，当磁盘写到 1638.4MB 时，剩余的可用空间为 409.6MB，这个值大于 AX_SYSLOG_minimalspace，所以 Log 会继续保存，直到可以空间小于 256MB，Log 开始循环覆盖。相反，如果总空间为 1GB，AX_SYSLOG_percent = 80，AX_SYSLOG_minimalspace = 256 的配置，那么当可用空间小于 AX_SYSLOG_minimalspace 时，Log 仍然会保存到磁盘的 80%，然后开始循环覆盖。请根据你的实际磁盘空间来配置这两项值。

log 等级配置

log 等级配置,包含 user log 和 kernel log 等级的配置说明如下：

```
#模块 Log 等级设置
ulog_state  = on
ulog_target = file
klog_state  = on
```

ulog_state：配置 user log 的输出与否，可选配置：on/off， on：输出 Log，off：关闭 Log。

ulog_target：配置 user log 的输出目的地，可选配置：file/console/null， file：Log 保存到文件，console：Log 输出到终端，null:Log 被丢弃，相当于配置了 ulog_state=off。

Klog_state：配置 kernel log 的输出与否，可选配置：on/off， on：输出 Log，off：关闭 Log。

各模块 Log 等级配置：

```
#module id level(user) level(kernel)
  ISP     1        4         4
  CE      2        4         4
```

```
VO      3       4       4
...
```

第一列：module 是模块名称,最大长度 11 字符(超过将会被截断),可自定义命名;

第二列：是模块 id，模块的唯一标识，取值范围：0~255，其中 0~127 为 SDK 内部使用 id， 128~255 共客户自定义使用；

第三列：level(user)是模块对应的 user log 输出等级，可配置的等级为 0-7，值越小，等级越高，0-7 对应的 Log 等级如下：

值	等级
0	EMERGENCY
1	ALERT
2	CRITICAL
3	ERROR
4	WARN
5	NOTICE
6	INFO
7	DEBUG

第四列：level(kernel) 是模块对应的 kernel log 输出等级，可配置的等级为 0-7，值越小，等级越高，0-7 对应的 log 等级与 user log 一致。

6.4.3 只读文件系统的日志部署

SDK Log 存储的位置，根据配置/etc/ax_syslog.config 文件：

```
AX_SYSLOG_path = /opt/data
```

配置可知位于/opt/data/ASyslog/syslog 目录，log 文件是以“日期_时间.log”形式命名的，如：2023-05-11_15-22-26.log。Log 文件的大小根据配置：

```
AX_SYSLOG_filesizemax = 50
```

SDK 中提供的默认值是 50MB，当 log 文件超过 50MB 时，会建立一个新的文件。当总的日志文件大小达到配置：

```
AX_SYSLOG_volume = 1024
```

时，将会循环覆盖，删除最早创建的 log 文件，并创建新的 log 文件，如此循环。

user log 输出说明

user log 输出的格式如下：

```
系统时间 Axera: [内核时间] [模块名][log 类型][进程 id] Log 内容
```

其中：

系统时间：log 输出时的系统时间

[内核 Log 时间] 从开机开始计算的内核时间

[CPU] AX650 CPU 为 8 核，取值区间为 C0-C7

[log 类型]

M: eMergency

A: Alert

C: Critical

E: Error

W: Warning

N: Notic

I: Information

D: Debug

[进程 id] 输出 Log 所属线程 ID

[模块名] 输出 Log 所属模块

示例 Log 如下：

```
2023-05-09 19:43:36 AXera: [ 6783.744241] [AX_VDEC][W][1570] [tid:1613,
AxVdDot10][AX_DecPicInfo_Print][711] all pp disabled! 265
2023-05-09 19:43:36 AXera: [ 6783.744262] [AX_VDEC][W][1570] [tid:1618,
AxVdDewt11][AX_VsiDecProcess][1769] VdGrp=11, 1 DEC_STRM_PROCESSED
```

kernel log 输出说明

通过 ax_printk 输出的 Log 格式如下：

```
系统时间 kernel: [内核时间] [CPU][模块名][log 类型][线程 id] Log 内容
```

其中：

系统时间：log 输出时的系统时间

[内核 Log 时间] 从开机开始计算的内核时间

[CPU] AX650 CPU 为 8 核，取值区间为 C0-C7

[log 类型]

M: eMergency

A: Alert

C: Critical

E: Error

W: Warning

N: Notic

I: Information

D: Debug

[线程 id] 输出 Log 所属线程 ID

[模块名] 输出 Log 所属模块

示例 Log 如下：

```
2023-05-09 19:43:36 kernel: [ 6783.908171] [C5][VO][E][1570] [__draw_chn_disable:176] layer0-chn58 status(0x81) invalid
2023-05-09 19:43:36 kernel: [ 6784.059040] [C5][ivps][E][1570] [axvpp_wq_destroy:1436] error result:0x7ff
```

根据以上 user log 和 kernel log 的输出说明可知，我们保持了 user log 与 kernel log 时间的一致，包括系统时间的一致，以及内核时间的一致(这里的内核时间，是指从开机开始计算的时间)，这对对比分析 user log 与 kernel log 非常有帮助。

6.5 集成自定义模块 Log 到日志系统

6.5.1 集成应用程序 Log

要集成应用程序 Log 到 SDK 自研日志系统，主要涉及到 4 部分内容：

1) .h 头文件引用，主要包括：ax_global_type.h、ax_sys_api.h 和 ax_sys_log.h；

2) ax_syslog.conf 配置文件，添加对应的配置；

3) API 调用，主要包括：AX_SYS_Init()、AX_SYS_Deinit()和 AX_SYS_LogPrint_Ex()；

4) 库文件引用: libax_sys.so 或 libax_sys.a。

为更好说明如何集成应用程序 Log 到我们的日志系统，下面举例演示。

步骤 1：在 ax_global_type.h 文件的 AX_MOD_ID_E 枚举中添加自定义模块 ID；

ax_global_type.h 文件位于 SDK 的 msp/out/include 目录，其 AX_MOD_ID_E 枚举包含了所有模块 ID 的定义：

```
typedef enum
```

```
{
    AX_ID_MIN        = 0x00,
    AX_ID_ISP        = 0x01,
    AX_ID_CE         = 0x02,
    AX_ID_VO         = 0x03,
    AX_ID_VDSP       = 0x04,
    ......
    /* for customer*/
    AX_ID_CUST_MIN = 0x80, /* 128 */
    AX_ID_MAX        = 0xFF   /* 255 */
} AX_MOD_ID_E;
```

AX_ID_CUST_MIN 之前的 ID 是供 SDK 内部使用的，AX_ID_CUST_MIN 及之后是供客户客制化使用的，所以增加模块的时候，请在 AX_ID_CUST_MIN 之后增加，避免与 SDK 产生冲突。

步骤 2：在 ax_syslog.conf 文件中添加配置；

步骤 3：在源文件中调用 AX_SYS API；

API 的调用共涉及到 3 个：AX_SYS_Init()、AX_SYS_Deinit()和 AX_SYS_LogPrint_Ex()。其中 AX_SYS_Init()和 AX_SYS_Deinit()是系统初始化和去初始化，AX_SYS_LogPrint_Ex()是 Log 输出 API。

步骤 4：在 Make file 中指定 libax_sys 库路径。

libax_sys.so/libax_sys.a 位于 SDK 的 msp\out\lib 目录，步骤 2 中引用的 AX_SYS API 都位于 libax_sys 库中。

demo 程序代码(文件路径：msp\sample\ulog)如下：

ax_global_type.h 文件,新增 AX_ID_CUST1、AX_ID_CUST2、AX_ID_CUST3 共 3 个 ID，其 ID 值从 128 开始递增：

```
typedef enum
{
    AX_ID_MIN        = 0x00,
    AX_ID_ISP        = 0x01,
```

```
    AX_ID_CE          = 0x02,
    AX_ID_VO          = 0x03,
    AX_ID_VDSP        = 0x04,
    ......
    /* for customer*/
    AX_ID_CUST_MIN = 0x80, /* 128 */
    AX_ID_CUST1 = AX_ID_CUST_MIN,
    AX_ID_CUST2,
    AX_ID_CUST3,

    AX_ID_MAX         = 0xFF   /* 255 */
} AX_MOD_ID_E;
```

ax_syslog.conf 文件：

```
#module id level(user) level(kernel)
ISP      1        4          4
CE       2        4          4
VO       3        4          4
VDSP     4        4          4
...
CUST1   128       4          4
CUST2   129       4          4
CUST3   130       4          4
```

【注意】module 列字符长度不能超过 11 字节，超出将被截断。

sample_ulog.c 文件：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <errno.h>
#include <unistd.h>
```

```
#include <pthread.h>
#include "ax_sys_log.h"
#include "ax_sys_api.h"

int id_list[] = {
    AX_ID_CUST1,
    AX_ID_CUST2,
    AX_ID_CUST3,
};

char* tag_list[] = {
    "CUST1",
    "CUST2",
    "CUST3",
};

int level_list[] = {
    SYS_LOG_EMERGENCY,
    SYS_LOG_ALERT,
    SYS_LOG_CRITICAL,
    SYS_LOG_ERROR,
    SYS_LOG_WARN,
    SYS_LOG_NOTICE,
    SYS_LOG_INFO,
    SYS_LOG_DEBUG,
};

static int gLoopExit = 0;

static AX_VOID __sigint(int iSigNo)
{
    printf("Catch signal %d\n", iSigNo);
```

```
    gLoopExit = 1;

    return ;
}

int main(int argc, char **argv)
{
    int ret = 0;
    int i;
    int cnt = sizeof(id_list)/sizeof(id_list[0]);
    int level;
    signal(SIGINT, __sigint);

    // init
    ret = AX_SYS_Init();
    if (AX_SUCCESS != ret) {
        printf("AX_SYS_Init failed! Error Code:0x%X\n", ret);
        return -1;
    }

    while(gLoopExit==0){
        for (level=SYS_LOG_DEBUG; level>=SYS_LOG_EMERGENCY; level--){
            for (i=0; i< cnt; i++){
                AX_SYS_LogPrint_Ex(level, tag_list[i], id_list[i], "This log's id: %d leve:%d\n", id_list[i],
level);
                if (gLoopExit) break;
            }
            if (gLoopExit) break;
            usleep(1000000);
        }
    }
    //deinit
```

```
    ret = AX_SYS_Deinit();
    if (AX_SUCCESS != ret) {
        printf("AX_SYS_Deinit failed! Error Code:0x%X\n", ret);
        return -1;
    }
    return 0;
}
```

Makefile 文件：

```
CUR_PATH        := $(shell pwd)
HOME_PATH       := $(abspath $(CUR_PATH)/../../..)

include $(HOME_PATH)/build/color.mk
include $(HOME_PATH)/build/config.mak

PRJ_OUT_HOME        := $(HOME_PATH)/build/out/$(PROJECT)
OBJ_OUT_PATH        := $(PRJ_OUT_HOME)/objs
MSP_OUT_PATH        := $(HOME_PATH)/msp/out
SRC_PATH           := $(CUR_PATH)
COMMON_PATH         := $(CUR_PATH)/../common
LIB_PATH           := $(MSP_OUT_PATH)/lib
SRC_RELATIVE_PATH   := $(subst $(HOME_PATH)/,,$(SRC_PATH))
TARGET_OUT_PATH     := $(OBJ_OUT_PATH)/$(SRC_RELATIVE_PATH)
ROOTFS_TARGET_PATH  := $(MSP_OUT_PATH)
HEADER_EXT         := $(HOME_PATH)/header/external
OUT_PATH           := $(HOME_PATH)/msp/out

# output
MOD_NAME              := sample_ulog
OUTPUT                := .obj

# source
```

```
    SRCS                    := $(wildcard $(SRC_PATH)/*.c)

    CINCLUDE                += -I$(SRC_PATH) \
                               -I$(OUT_PATH)/include \
                               -I$(BASE_PATH)/sys \
                               -I$(COMMON_PATH)/include \
                               -I$(HEADER_EXT)

    OBJS                    := $(addprefix $(OUTPUT)/,$(notdir
$(patsubst %.c,%.o,$(SRCS))))
    DEPS                    := $(OBJS:%.o=%.d)
    # exec
    TARGET                  := $(MOD_NAME)

    # build flags
    ifeq ($(debug),yes)
    CFLAGS                  += -Wall -O0 -ggdb3
    else
    CFLAGS                  += -Wall -O2
    endif

    # dependency
    CLIB                    += -L$(OUT_PATH)/lib
    CLIB                    += -lax_sys
    # CLIB                  += -lm -lpthread

    # install
    INSTALL_BIN     := $(TARGET)

    # link
    LINK = $(CC)
```

```
include $(HOME_PATH)/build/rules.mak
```

编译后生成 sample_ulog 可执行程序，运行后的 Log 输出：

```
2023-05-12 13:52:05 AXera: [62574.756935] [CUST1][W][5130] This log's id: 128 leve:4
2023-05-12 13:52:05 AXera: [62574.756995] [CUST2][W][5130] This log's id: 129 leve:4
2023-05-12 13:52:05 AXera: [62574.757014] [CUST3][W][5130] This log's id: 130 leve:4
2023-05-12 13:52:06 AXera: [62575.757934] [CUST1][E][5130] This log's id: 128 leve:3
2023-05-12 13:52:06 AXera: [62575.757977] [CUST2][E][5130] This log's id: 129 leve:3
2023-05-12 13:52:06 AXera: [62575.757995] [CUST3][E][5130] This log's id: 130 leve:3
2023-05-12 13:52:07 AXera: [62576.758932] [CUST1][C][5130] This log's id: 128 leve:2
2023-05-12 13:52:07 AXera: [62576.758972] [CUST2][C][5130] This log's id: 129 leve:2
2023-05-12 13:52:07 AXera: [62576.758989] [CUST3][C][5130] This log's id: 130 leve:2
2023-05-12 13:52:08 AXera: [62577.759942] [CUST1][A][5130] This log's id: 128 leve:1
2023-05-12 13:52:08 AXera: [62577.759998] [CUST2][A][5130] This log's id: 129 leve:1
2023-05-12 13:52:08 AXera: [62577.760017] [CUST3][A][5130] This log's id: 130 leve:1
2023-05-12 13:52:09 AXera: [62578.760941] [CUST1][M][5130] This log's id: 128 leve:0
2023-05-12 13:52:09 AXera: [62578.760996] [CUST2][M][5130] This log's id: 129 leve:0
2023-05-12 13:52:09 AXera: [62578.761013] [CUST3][M][5130] This log's id: 130 leve:0
```

cat /proc/ax_proc/logctl 查看当前 Log 等级配置：

module	id	level(U)	level(K)
ISP	1	4	4
CE	2	4	4
VO	3	4	4
VDSP	4	4	4
EFUSE	5	4	4
NPU	6	4	4
VENC	7	4	4
VDEC	8	4	4
JENC	9	4	4
JDEC	10	4	4
SYS	11	4	4
AENC	12	4	4
IVPS	13	4	4
MIPI	14	4	4
ADEC	15	4	4
DMA	16	4	4
VIN	17	4	4
USER	18	4	4
IVES	19	4	4
SKEL	20	4	4
IVE	21	4	4
3A	25	4	4
AI	32	4	4
AO	33	4	4
SENSOR	34	4	4
CUST1	128	4	4
CUST2	129	4	4
CUST3	130	4	4

至此，应用程序 Log 的集成完成。

6.5.2 集成驱动程序 Log

要集成驱动程序 Log 到 SDK 自研日志系统，则相对更加简单，只需要三个步骤：

1) 在 ax_global_type.h 的 AX_MOD_ID_E 中添加自定义模块 ID；

2) ax_syslog.conf 配置文件，添加对应的配置；

3) 在驱动程序中调用 ax_printk API 打印 Log。

为更好说明如何集成驱动程序 Log 到我们的日志系统，下面举例演示。

步骤 1：在 ax_global_type.h 文件的 AX_MOD_ID_E 枚举中添加自定义模块 ID，方式同集成应用程序 Log 介绍的一样，如果应用程序已经添加过对应的模块，则不用重复添加；

步骤 2：在驱动程序源码中调用 ax_printk 接口输出 Log;

为更好说明如何集成驱动程序 Log 到我们的日志系统，下面举例演示。

demo 程序代码(文件路径：kernel\osdrv\klog)如下：

ax_global_type.h 文件，新增 AX_ID_CUST1、AX_ID_CUST2、AX_ID_CUST3 共 3 个 ID，操作同集成应用程序 Log 中介绍的一样，如果应用程序已经添加过对应的模块，则不用重复添加。

Makefile 文件

```
MKFILE_PATH       := $(abspath $(lastword $(MAKEFILE_LIST)))
HOME_PATH         := $(abspath $(dir $(MKFILE_PATH))../../..)
BUILD_PATH        := $(HOME_PATH)/build
ROOTFS_PATH       := $(HOME_PATH)/rootfs

include $(BUILD_PATH)/config.mak

OBJ_OUT_DIR       := $(BUILD_PATH)/out/$(PROJECT)/objs
KSRC              := $(abspath $(dir $(MKFILE_PATH)))
SRC_RELATIVE_PATH  := $(subst $(HOME_PATH)/,,$(KSRC))
KBUILD_DIR        := $(OBJ_OUT_DIR)/$(SRC_RELATIVE_PATH)
KBUILD_MAKEFILE    := $(KBUILD_DIR)/Makefile
MSP_OUT_PATH      := $(HOME_PATH)/msp/out

EXTRA_CFLAGS += -I$(MSP_OUT_PATH)/include/

KDIR ?= $(HOME_PATH)/kernel/linux/$(KERNEL_DIR)
MODULE_NAME := test_klog

SRCS := testklog.c

$(MODULE_NAME)-objs := $(SRCS:%.c=%.o)

obj-m := $(MODULE_NAME).o

clean-objs := $(SRCS:%.c=%.o)
clean-objs += $(join $(dir $(SRCS)), $(patsubst %.c, .%.o.cmd, $(notdir
```

```
$(SRCS))))

    include $(BUILD_PATH)/krules.mak
```

testklog.c 文件

```
    #include <linux/module.h>
    #include <linux/proc_fs.h>
    #include <linux/seq_file.h>
    #include <linux/slab.h>
    #include <linux/delay.h>
    #include "ax_global_type.h"

    #define TEST_KLOG_PROC_NAME             "ax_proc/test_klog"

    int id_list[] = {
        AX_ID_CUST1,
        AX_ID_CUST2,
        AX_ID_CUST3,
    };

    char* tag_list[] = {
        "CUST1",
        "CUST2",
        "CUST3",
    };

    static int test_klog(void)
    {
        int ret = 0;
        int i,level;
        int cnt = sizeof(id_list)/sizeof(id_list[0]);
        for (level=SYS_LOG_DEBUG; level>=SYS_LOG_EMERGENCY; level--){
```

```
        for(i=0; i < cnt; i++){
            ax_printk(id_list[i], tag_list[i], level, "This log's id:%d, level:%d", id_list[i], level);
        }
    }
    return ret;
}


static ssize_t logctl_proc_write(struct file *file, const char *buffer, size_t count, loff_t *f_pos)
{
    char *tmp = kzalloc((count + 1), GFP_KERNEL);
    char cmd[64];
    int ret;

    if (!tmp)
        return -ENOMEM;
    if (copy_from_user(tmp, buffer, count)) {
        pr_err("@%s copy_from_user failed!!!\n", __FUNCTION__);
        kfree(tmp);
        return -EFAULT;
    }

    ret = sscanf(tmp, "%s", cmd);
    if (0 == strcmp(cmd, "test")){
        test_klog();
    }

    kfree(tmp);
    tmp = NULL;

    return count;
}
```

```
static int axlogctl_proc_show(struct seq_file *p, void *v)
{
    return 0;
}

static int logctl_proc_open(struct inode *inode, struct file *file)
{
    return single_open(file, axlogctl_proc_show, NULL);
}

static const struct proc_ops proc_axlogctl_operations = {
    .proc_flags = PROC_ENTRY_PERMANENT,
    .proc_open    = logctl_proc_open,
    .proc_read    = seq_read,
    .proc_write = logctl_proc_write,
    .proc_release = single_release,
};

static int __init test_klog_proc_init(void)
{
    proc_create(TEST_KLOG_PROC_NAME, S_IRUSR, NULL, &proc_axlogctl_operations);
    return 0;
}

static void __exit test_klog_proc_exit(void)
{
        remove_proc_entry(TEST_KLOG_PROC_NAME, NULL);
}

module_init(test_klog_proc_init);
module_exit(test_klog_proc_exit);
```

```
MODULE_AUTHOR("axera");
MODULE_LICENSE("GPL");
```

编译后生成 test_klog.ko 驱动程序，insmod 后会生成/proc/ax_proc/test_klog 节点，执行:

echo test > /proc/ax_proc/test_klog

即可看到/opt/data/AXSyslog/syslog 目录下的 Log 文件中查看到 Log 输出：

```
2023-05-12 14:26:32 kernel: [   61.314683] [C5][CUST1][W][538] This log's id:128, level:4
2023-05-12 14:26:32 kernel: [   61.314693] [C5][CUST2][W][538] This log's id:129, level:4
2023-05-12 14:26:32 kernel: [   61.314695] [C5][CUST3][W][538] This log's id:130, level:4
2023-05-12 14:26:32 kernel: [   61.314696] [C5][CUST1][E][538] This log's id:128, level:3
2023-05-12 14:26:32 kernel: [   61.314698] [C5][CUST2][E][538] This log's id:129, level:3
2023-05-12 14:26:32 kernel: [   61.314700] [C5][CUST3][E][538] This log's id:130, level:3
2023-05-12 14:26:32 kernel: [   61.314702] [C5][CUST1][C][538] This log's id:128, level:2
2023-05-12 14:26:32 kernel: [   61.314703] [C5][CUST2][C][538] This log's id:129, level:2
2023-05-12 14:26:32 kernel: [   61.314705] [C5][CUST3][C][538] This log's id:130, level:2
2023-05-12 14:26:32 kernel: [   61.314707] [C5][CUST1][A][538] This log's id:128, level:1
2023-05-12 14:26:32 kernel: [   61.314708] [C5][CUST2][A][538] This log's id:129, level:1
2023-05-12 14:26:32 kernel: [   61.314710] [C5][CUST3][A][538] This log's id:130, level:1
2023-05-12 14:26:32 kernel: [   61.314711] [C5][CUST1][M][538] This log's id:128, level:0
2023-05-12 14:26:32 kernel: [   61.314713] [C5][CUST2][M][538] This log's id:129, level:0
2023-05-15 14:26:32 kernel: [   61.314715] [C5][CUST3][M][538] This log's id:130,
level:0
```

至此，驱动程序 Log 的集成完成。

6.6 动态修改 Log 等级

6.6.1 动态修改 Linux 内核原生 Log 等级

对于 Linux 原生内核 log，即通过 printk 输出的 log，可以通过/proc/sys/kernel/printk 节点来动态控制输出到串口的 linux log 等级。默认情况下，Linux 原生内核 Log 的等级配置为

4(warning)，这可以通过 cat /proc/sys/kernel/printk 来查看：

```
# cat /proc/sys/kernel/printk
 4       4       1       7
```

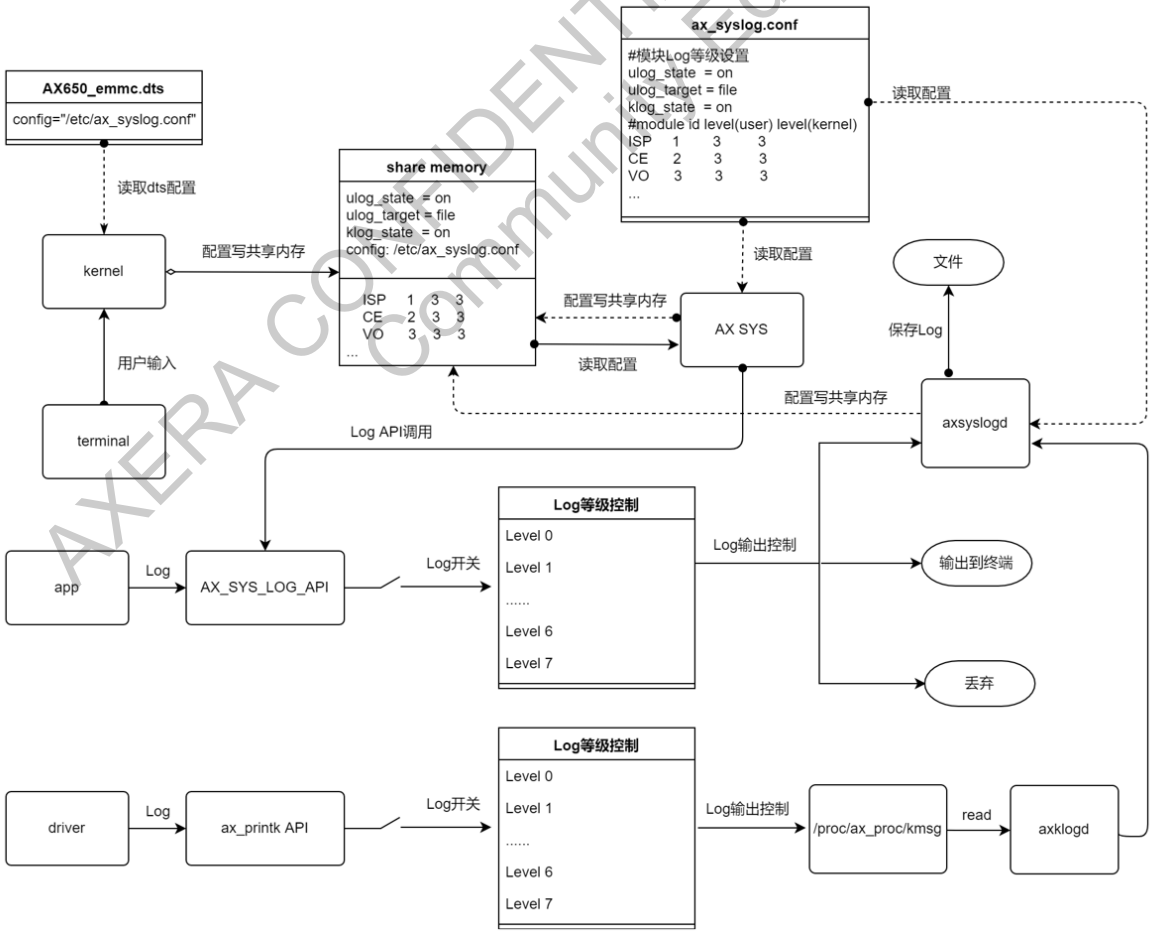
如果要修改输出到终端的 Log 等级，比如修改到等级 7，可以通过下面指令来实现：

```
echo 7 > /proc/sys/kernel/printk
```

关于 printk log 等级设置的更多用法，可参考内核文档 linux-5.15.73/Documentation/core-api/printk-basics.rst。

6.6.2 SDK 自研日志系统 log 等级控制流程

本 SDK 自研日志系统 Log 等级控制的基本流程如下图所，配置文件 ax_syslog.conf 文件包含了 log 等级配置基本信息。



6.6.3 动态修改 user log 等级

我们的 log 系统支持按模块进行 log 等级修改，每个模块都有自己对应的 ID，比如 ISP、VO 和 NPU 他们对应的模块 ID 分别是 1 、 3 、 6 。确定模块对应的 ID 值的方式有两种：

1) 通过 cat /etc/ax_syslog.conf 文件查看；

2) 通过 cat /proc/ax_proc/logctl 节点查看。

在 ax_syslog.conf 配置文件中，我们有一段这样的配置：

```
#模块 Log 等级设置
ulog_state  = on
ulog_target = file
klog_state  = on
#module id level(user) level(kernel)
  ISP     1        4         4
  CE      2        4         4
  VO      3        4         4
  VDSP    4        4         4
  ....
```

以 ISP 模块为例，其 id 为 1，user log 等级为 4，这表示 ISP 模块 user log 等级为 0-4 的 Log 能够被输出，而等级为 5-7 的 Log 将被忽略。如果您需要修改模块的默认等级，可以在 ax_syslog.conf 配置文件修改并保存。

为了方便对 user log 等级控制，我们提供了根据模块 id 来控制 log 等级的指令。指令格式如下：

```
echo ulog [id] [level] > /proc/ax_proc/logctl
```

其中 ulog 表示控制 user log 等级， id 为模块 ID，比如 ISP 的模块 ID 为 1，level 为控制 log 输出的等级，而/proc/ax_proc/logctl 为控制 log 等级的设备节点。为更好地展示如何控制 user log 的等级，我们举例说明。

示例 1：把 ISP 模块的 user log 输出等级设置为 7，可通过指令：

```
echo ulog 1 7 > /proc/ax_proc/logctl
```

来实现，修改后可通过:

```
cat /proc/ax_proc/logctl
```

查看设置是否生效：

```
 module    id  level(U)  level(K)
    ISP     1         7         4
     CE     2         4         4
     VO     3         4         4
   VDSP     4         4         4
  EFUSE     5         4         4
    NPU     6         4         4
   VENC     7         4         4
   VDEC     8         4         4
   JENC     9         4         4
```

user log 等级设置为 7 后，所有等级的 ISP user log 输出均可看到。

示例 2： 将所有模块的 user log 等级修改为 2

```
echo ulog all 2 > /proc/ax_proc/logctl
```

查看修改结果：

```
 module    id  level(U)  level(K)
    ISP     1         2         4
     CE     2         2         4
     VO     3         2         4
   VDSP     4         2         4
  EFUSE     5         2         4
    NPU     6         2         4
   VENC     7         2         4
   VDEC     8         2         4
   JENC     9         2         4
   JDEC    10         2         4
    SYS    11         2         4
   AENC    12         2         4
   IVPS    13         2         4
   MIPI    14         2         4
   ADEC    15         2         4
    DMA    16         2         4
    VIN    17         2         4
   USER    18         2         4
   IVES    19         2         4
   SKEL    20         2         4
    IVE    21         2         4
     3A    25         2         4
     AI    32         2         4
     AO    33         2         4
```

示例 3：关闭/打开 user log 输出

```
klog state:  on
ulog state:  off
ulog_target: console
```

这种情况下，user log 将不再输出（不只是 log 不保存到文件，在应用程序中 log 的就已经不输出了）。

打开 user log 输出：

```
echo ulog on > /proc/ax_proc/logctl
```

示例 4：切换 user log 输出到 console 终端

```
echo ulog console > /proc/ax_proc/logctl
```

如果要切换输出到文件，可通过指令

```
echo ulog file> /proc/ax_proc/logctl
```

来实现。

6.6.4 动态修改 kernel log 等级

kernel log 等级的控制与 user log 等级的控制方式基本一致，只需将命令中的 ulog 改成 klog，最大的差异在于 user log 能够控制输出到 console 终端，但 kernel log 只能输出到日志文件。

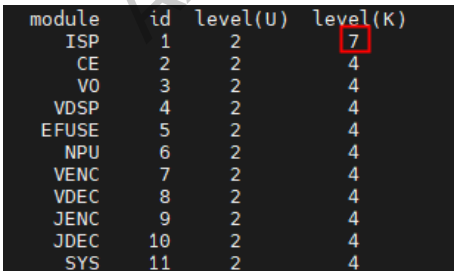
以下举例演示如何控制 kernel log 的输出。

示例 1：把 ISP 模块的 log 输出等级设置为 7，可通过指令：

```
echo klog 1 7 > /proc/ax_proc/logctl
```

来实现，修改后可通过:

```
cat /proc/ax_proc/logctl
```



module	id	level(U)	level(K)
ISP	1	2	7
CE	2	2	4
VO	3	2	4
VDSP	4	2	4
EFUSE	5	2	4
NPU	6	2	4
VENC	7	2	4
VDEC	8	2	4
JENC	9	2	4
JDEC	10	2	4
SYS	11	2	4

示例 2： 将所有模块的 kernel log 等级修改为 3

```
echo klog all 3 > /proc/ax_proc/logctl
```

查看修改结果：

```
module    id  level(U)  level(K)
   ISP     1     2         3
    CE     2     2         3
    VO     3     2         3
  VDSP     4     2         3
 EFUSE     5     2         3
   NPU     6     2         3
  VENC     7     2         3
  VDEC     8     2         3
  JENC     9     2         3
  JDEC    10     2         3
   SYS    11     2         3
  AENC    12     2         3
  IVPS    13     2         3
  MIPI    14     2         3
  ADEC    15     2         3
   DMA    16     2         3
   VIN    17     2         3
  USER    18     2         3
  IVES    19     2         3
  SKEL    20     2         3
   IVE    21     2         3
    3A    25     2         3
    AI    32     2         3
    AO    33     2         3
SENSOR    34     2         3
```

示例 3：关闭/打开 kernel log 输出

```
echo klog off > /proc/ax_proc/logctl
```

查看结果

```
----------------------------------------
klog_state:  off
ulog_state:  on
ulog_target: file
----------------------------------------
```

关闭 kernel log 后将不会有任何 kernel log 输出。
重新打开 kernel log 可以执行

```
echo klog on > /proc/ax_proc/logctl
```