编译:

./build.sh all编译整个SDK

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

./build.sh all编译整个SDK

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

单独编译u-boot:

./build.sh u-boot

会生成两个镜像：

SDK/uboot/uboot.img

SDK/uboot/rk356x\_spl\_loader\_v1.13.112.bin也就是MiniLoaderAll.bin

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

单独编译内核kernel:

./build.sh kernel

编译成功会生成SDK/kernel/boot.img

该命令会编译Linux内核源码，包括内核设备树，内核模块。

如果只需要编译内核模块则可执行：

./build.sh modules

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

单独编译根文件系统rootfs，本开发板支持多种根文件系统(Buildroot、Yocto、Bebian)，推荐使用buildroot根文件系统

单独编译Buildroot根文件系统:

执行./build.sh buildroot

会在buildroot/output/rockchip\_rk3568/images/目录下生成 buildroot根文件系统镜像。

注意：编译会生成不同格式的rootfs镜像文件(如.cpio、.cpio.gz、.ext2、.ext4、.tar、.squashfs)，但对于RK3568使用ext4格式镜像rootfs.ext2，并且通常会重命名为rootfs.img

除此之外还可以执行：./build.sh rootfs编译根文件系统，但该命令不仅限于编译buildroot，还可以编译Yocto和Debian，只是默认是编译buildroot。可以通过环境变量RK\_ROOTFS\_SYSTEM指定需要编译的rootfs，例：

export RK\_ROOTFS\_SYSTEM＝buildroot

另外需要注意：编译根文件系统之前需要编译内核，因为编译根文件系统过程中会编译部分没集成在内核源码中的驱动模块，而且也会将内核源码目录下编译生成的.ko驱动模块拷贝到根文件系统。

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

单独编译recovery:

./build.sh recovery

会生成recovery.img 用于进入recovery模式，该镜像会烧录到recovery分区。

该镜像由多个镜像合并生成：ramdisk(recovery模式下挂载的根文件系统)、内核镜像、内核DTB、资源镜像resource.img。所以编译之前也需要先编译内核。

编译完成后，在buildroot/output/rockchip\_rk356x\_recovery/images/目录下生成 recovery.img镜像。

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

打包成update.img镜像：

执行./build.sh updateimg

RK工具将多个分立镜像(uboot.img、boot.img、MiniLoaderAll.bin、paramter.txt、misc.img、rootfs.img、oem.img、userdata.img、recovery.img等)打包成一个update.img固件。

打包成功后，会在rockdev目录下生成update.img

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SDK清理:

执行./build.sh cleanall

会清理uboot、kernel、buildroot(rootfs、recovery)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SDK开发：

1.SDK板及配置文件：

./build.sh lunch：

SDK板级配置文件:

位于SDK/device/rockchip/rk356x/ 目录。

BoardConfig-xxxx.mk 文件，这些.mk 文件便是板级配置文件。其中

BoardConfig-rk3568-atompi-evb1-ddr4-v10.mk 就是我们的 ATK-DLRK3568 开发板所使用的板级

配置文件；我们在 SDK 根目录下执行“./build.sh lunch”时所列举出来的文件就是从

<SDK>/device/rockchip/rk356x/目录来的。

这些.mk 文件其实是一个 sh 脚本文件，打开配置文件：

/\*

#!/bin/bash

# Target arch

export RK\_ARCH=arm64

# Uboot defconfig

export RK\_UBOOT\_DEFCONFIG=rk3568

# Uboot image format type: fit(flattened image tree)

export RK\_UBOOT\_FORMAT\_TYPE=fit

# Kernel defconfig

export RK\_KERNEL\_DEFCONFIG=rockchip\_linux\_defconfig

# Kernel defconfig fragment

export RK\_KERNEL\_DEFCONFIG\_FRAGMENT=

# Kernel dts

export RK\_KERNEL\_DTS=rk3568-atk-atompi-ca1

# boot image type

export RK\_BOOT\_IMG=boot.img

# kernel image path

export RK\_KERNEL\_IMG=kernel/arch/arm64/boot/Image

# kernel image format type: fit(flattened image tree)

export RK\_KERNEL\_FIT\_ITS=boot.its

# parameter for GPT table

export RK\_PARAMETER=parameter-buildroot-fit.txt

# Buildroot config

export RK\_CFG\_BUILDROOT=rockchip\_rk3568

# Recovery config

export RK\_CFG\_RECOVERY=rockchip\_rk356x\_recovery

# Recovery image format type: fit(flattened image tree)

export RK\_RECOVERY\_FIT\_ITS=boot4recovery.its

# ramboot config

export RK\_CFG\_RAMBOOT=

# Pcba config

export RK\_CFG\_PCBA=

# Build jobs

export RK\_JOBS=24

# target chip

export RK\_TARGET\_PRODUCT=rk356x

# Set rootfs type, including ext2 ext4 squashfs

export RK\_ROOTFS\_TYPE=ext4

# Set debian version (debian10: buster, debian11: bullseye)

export RK\_DEBIAN\_VERSION=buster

# yocto machine

export RK\_YOCTO\_MACHINE=rockchip-rk3568-evb

# rootfs image path

export RK\_ROOTFS\_IMG=rockdev/rootfs.${RK\_ROOTFS\_TYPE}

# Set ramboot image type

export RK\_RAMBOOT\_TYPE=

# Set oem partition type, including ext2 squashfs

export RK\_OEM\_FS\_TYPE=ext2

# Set userdata partition type, including ext2, fat

export RK\_USERDATA\_FS\_TYPE=ext2

#OEM config

export RK\_OEM\_DIR=oem\_normal

# OEM build on buildroot

#export RK\_OEM\_BUILDIN\_BUILDROOT=YES

#userdata config

export RK\_USERDATA\_DIR=userdata\_normal

#misc image

export RK\_MISC=wipe\_all-misc.img

#choose enable distro module

export RK\_DISTRO\_MODULE=

# Define pre-build script for this board

export RK\_BOARD\_PRE\_BUILD\_SCRIPT=app-build.sh

#only for atompi-ca1

\*/

RK\_ARCH：用于指定目标架构，rk3568 对应 arm64；

RK\_UBOOT\_DEFCONFIG：用于指定 U-Boot 的 defconfig 配置文件：rk3568。

RK\_UBOOT\_FORMAT\_TYPE：用于指定 uboot.img 镜像的格式，rk3568 平台默认使用的

是 fit（flattened image tree）格式镜像；

RK\_KERNEL\_DEFCONFIG：用于指定 Linux Kernel（内核）的 defconfig 配置文件；

rockchip\_linux\_defconfig。

RK\_KERNEL\_DEFCONFIG\_FRAGMENT：用于指定 Linux 内核的 defconfig fragment，

对于 rk3568 来说是空；

RK\_KERNEL\_DTS：用于指定内核设备树文件rk3568-atk-atompi-ca1。

RK\_BOOT\_IMG：设置为 boot.img 即可；

RK\_KERNEL\_IMG：用于指定内核镜像的路径；kernel/arch/arm64/boot/Image。

RK\_KERNEL\_FIT\_ITS：rk3568 平台 Linux 系统使用的启动镜像 boot.img 也是 FIT 格式

镜像，FIT 使用 its（image source file）文件来描述 image 的信息，RK\_KERNEL\_FIT\_ITS 用于

指定这个 its 文件，这个 its 文件必须要存放在<SDK>/device/rockchip/rk356x/目录下；boot.its。

RK\_PARAMETER：用于指定分区表文件，也就是前面给大家介绍的 parameter.txt 文件；

RK\_CFG\_BUILDROOT：用于指定 buildroot 根文件系统（普通模式）的 defconfig 配置文

件；rockchip\_rk3568\_defconfig。

RK\_CFG\_RECOVERY：用于指定recovery模式下根文件系统（recovery模式下使用ramdisk

根文件系统）的 defconfig 配置文件；rockchip\_rk356x\_recovery\_defconfig。

RK\_RECOVERY\_FIT\_ITS：用于指定 recovery.img 镜像对应的 its 文件。recovery.img 也

是 FIT 格式镜像，需要使用 its 文件来描述 image 的信息；boot4recovery.its。

RK\_CFG\_RAMBOOT：默认是空置，然不知其意；

RK\_CFG\_PCBA：用于指定 PCBA 的 defconfig 配置文件；

RK\_JOBS：用于指定 make 编译时的线程数，譬如：make -j24；

RK\_TARGET\_PRODUCT：用于指定目标产品名，对于 rk3568 平台来说，默认将其设置

为 rk356x；

RK\_ROOTFS\_TYPE：用于指定根文件系统的类型，譬如 ext2、ext4；

RK\_DEBIAN\_VERSION：用于指定 Debian 的版本，debian10: buster, debian11: bullseye，

默认使用的是 Debian 10，不要去改动它；

RK\_YOCTO\_MACHINE：编译 yocto 时，用于指定 machine；

RK\_ROOTFS\_IMG：用于指定根文件系统镜像的路径；

RK\_RAMBOOT\_TYPE：默认是空置，然不知其意；

RK\_OEM\_FS\_TYPE：用于指定 oem 分区的类型，保持默认就行；

RK\_USERDATA\_FS\_TYPE：用于指定 userdata 分区的类型，保持默认就行；

RK\_OEM\_DIR：用于指定 oem 对应的文件夹，<SDK>/device/rockchip/oem 目录下有多个

oem\_xxx 文件夹，这些文件夹中存放了厂家的 APP 或数据等，最终会编译成 oem.img 镜像；

RK\_USERDATA\_DIR：用于指定 userdata 对应的文件夹，<SDK>/device/rockchip/userdata

目录下有多个 userdata\_xxx 文件夹，这些文件夹中存放了最终用户的 APP 或数据等，最终会编

译成 userdata.img 镜像；

RK\_MISC：用于指定 misc 镜像。编译完 SDK 后，生成的<SDK>/rockdev/misc.img 镜像其

实就是 RK\_MISC 所指定的这个文件，只不过是进行了重命名而已；RK\_MISC 所指定的 misc

镜像必须要存放在<SDK>/device/rockchip/rockimg 目录下；

RK\_DISTRO\_MODULE：未使用到；

RK\_BOARD\_PRE\_BUILD\_SCRIPT：

关于这个板级配置文件就讲这么多，用户可以在**<SDK>/device/rockchip/rk356x** 目录下添加自己的板级配置文件，根据实际情况对配置文件中的变量进行修改、或添加新的变量。

1. U-boot开发：

源码位于：SDK/u-boot

看文档：

**<SDK>/docs/Common/UBOOT/Rockchip\_Developer\_Guide\_UBoot\_Nextdev\_CN.pdf**

**（或开发板光盘 A 盘-基础资料08、RK 官方文档LinuxCommonUBOOTRockchip\_Developer\_Guide\_UBoot\_Nextdev\_CN.pdf）**

**内包含：**Rockchip 平台 U-Boot 所涉及到的知识点、技术点， 包括 Rockchip 平台 U-Boot 基础简介、U-Boot 架构、U-Boot 启动流程、U-Boot 系统模块、驱动模块、Kernel-DTB、AB 系统、AVB 安全启动、TPL、SPL、U-Boot 快捷键等

* 1. U-boot设备树开发：

设备树分为 u-boot设备树 和 kernel设备树：

U-Boot 设备树负责初始化存储、调试串口等基础外设；

而 kernel 设备树初始化存储、调试 串口之外的外设，譬如 LCD 显示、千兆网等

U-Boot 中RK3568 的设备树文件是<U-Boot>/arch/arm/dts/rk3568-evb.dts

该设备树文件包含了 rk3568.dtsi 和 rk3568-u-boot.dtsi：

包含关系：

rk3568-evb.dts

rk3568.dtsi

rk3568-u-boot.dtsi

执行 U-Boot 代码前，先用 U-Boot 的设备树完成存储、调试串口的初始化操作，然后从存储上加载 kernel 的设备树并转而使用这个设备树继续初始化其余外设。

所以用户一般不需要去修改 U-Boot 的设备树文件（除非更换调试串口）。

* 1. U-boot 编译：

U-Boot 源码目录下提供了一个编译脚本 make.sh，可以直接使用该脚本编译 U-Boot 源码，譬如在 U-Boot 源码目录下执行如下命令编译 U-Boot：

./make.sh rk3568

编译完成后最终将会在该u\_boot目录下生成

**uboot.img** 和 **rk356x\_spl\_loader\_v1.13.112.bin** 两个镜像文件

* + 1. uboot.img镜像:

前面提到，uboot.img 是由多个镜像合并而成，包括 u-boot 镜像、u-boot dtb 以及 trust 镜 像（ARM Trusted Firmware + OP-TEE）。uboot.img 是一种 FIT（**flattened image tree**）格式镜像，支持任意多个 image 打包和校验。使用 file 命令查看 uboot.img，

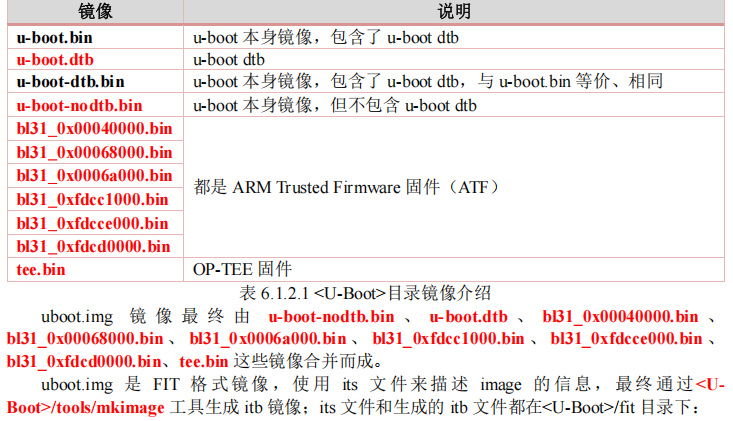
file uboot.img

FIT 使用 its（**image source file**）文件来描述 image 的信息，最后通过 mkimage 工具生成 itb（**flattened image tree blob**）镜像，那么这个 itb 镜像其实就是 uboot.img 镜像，uboot.img 镜像通常含有多份 itb 镜像，如下所示：

uboot.img = uboot.itb \* N（N 一般是 2）

这种设计也是为了避免如果第一份镜像启动失败、还可以尝试使用第二份镜像启动。

U-Boot 编译成功后，U-Boot 源码目录下会生成很多.bin 镜像以及.dtb 镜像：



* + 1. Rk356x\_spl\_loader\_v1.13.112.bin镜像

这个镜像文件就是前面给大家介绍的 MiniLoaderAll.bin 镜像

MiniLoaderAll.bin 由两部分构成：**TPL(Tiny Program Loader)** + **SPL(Secondary Program Loader)**构成。

**TPL 运行在 SRAM 中（片内内存），由 rk3568 芯片内部所固化的 Maskrom（BootROM）代码引导启动；其作用是负责完成 DRAM 的初始化工作、并启动 SPL**；**SPL 运行在 DDR，SPL 的作用是负责完成系统的 lowlevel 初始化、完成 uboot.img 的加载和引导工作**。

TPL、SPL 分别有两种实现方案：**开源版本**和**闭源版本**。

* + 1. 镜像启动顺序

Linux 系统的镜像启动顺序为：

Maskrom  MiniLoaderAll.bin  uboot.img  boot.img  rootfs.img

还可以进行细分：

Maskrom  TPL(ddr bin)  SPL(miniloader)  Trust(ATF + OP-TEE)  u-boot  kernel  rootfs

这个启动流程通过打印信息就可以分析出来。

* 1. U-boot的配置：

用户想要自己配置 U-Boot，可以在 U-Boot 源码目录下执行如下命令对其进行配置：

make rk3568\_defconfig //选择配置文件

make menuconfig //打开图形化配置界面

通过 menuconfig 图形化界面、用户可自行对 U-Boot 进行配置，配置完成后运行以下命令保存配置：

make savedefconfig //把配置信息保存到 defconfig 文件中

cp defconfig configs/rk3568\_defconfig //用 defconfig 文件替换 rk3568\_defconfig

保存配置后，我们可以直接在 U-Boot 源码目录下执行 make.sh 脚本编译 U-Boot 源码，也可以回到 SDK 根目录下执行 build.sh 脚本进行编译：

./make.sh rk3568 //在 U-Boot 源码目录下执行 make.sh 脚本编译

cd ../ //在 SDK 根目录下执行 build.sh 脚本编译

./build.sh uboot

* 1. defconfig配置文件

U-Boot中，RK3568平台使用的defconfig配置文件为：**<U-Boot>/configs/rk3568\_defconfig**

基本不用去动 U-Boot 源码。

如果不使用 make.sh（或 build.sh）脚本，直接通过 make 命令配置 U-Boot、然后编译 UBoot，可以这样做：

make rk3568\_defconfig -j24

make PYTHON=python2 CROSS\_COMPILE=<SDK>/prebuilts/gcc/linux-x86/aarch64/gcc-linaro-

6.3.1-2017.05-x86\_64\_aarch64-linux-gnu/bin/aarch64-linux-gnu- all --jobs=24

<SDK>表示 SDK 根目录，根据你自己的实际路径修改；编译内核、编译 U-Boot 用的都是这个交叉编译器 **<SDK>/prebuilts/gcc/linux-x86/aarch64/gcc-linaro-6.3.1-2017.05-**

**x86\_64\_aarch64-linux-gnu/bin/aarch64-linux-gnu-**。

使用make编译完成之后，并没有生成 uboot.img 和 rk356x\_spl\_loader\_v1.13.112.bin，因为这两个镜像属于 pack 打包镜像，需要通过专门的工具或脚本才能生成，而make.sh 脚本中已经添加了打包操作，所以使用 make.sh 脚本编译后会生成 uboot.img 和 rk356x\_spl\_loader\_v1.13.112.bin。

我们可以执行如下命令去打包生成这两个镜像：

./scripts/fit.sh --ini-loader ../rkbin/RKBOOT/RK3568MINIALL.ini --chip RK3568

scripts/fit.sh 脚本会调用 **tools/mkimage** 工具生成 uboot.img、调用 **tools/boot\_merger** 工具生成 rk356x\_spl\_loader\_v1.13.112.bin。

如果需要清理 U-Boot 源码工程，执行“**make distclean**”命令进行清理：

make distclean

1. Kernel内核开发

Linux 内核源码在**<SDK>/kernel** 目录下：

* 1. 内核编译

编译之前，我们先执行“**make distclean**”清理一下。

在内核源码目录下有一个 make.sh 脚 本文件，可以直接使用这个脚本文件来编译内核源码，直接运行：

./make.sh

编译完成后，会生成 boot.img 以及 resource.img

* + 1. Boot.img镜像：

boot.img 是一种 FIT 格式镜像，它由多个镜像合并而成，对于 RK3568 平台来说，烧录到开发板 boot 分区的 boot.img 包含了 内核镜像 Image、内核 DTB、resource.img 这三部分。

boot.img 是一种 FIT 格式镜像,以使用 file 命令查看 boot.img 文件发现该文件是一个 Android 格式启动镜像（Android bootimg），并非 FIT 格式。

**事实上，通过 make.sh 脚本编译后生成的 boot.img 确实是 Android 格式镜像； Android 格式 boot.img 是用于 RK3568 平台 Android 系统，**

**而 FIT 格式 boot.img 才用于 RK3568 平台 Linux 系统**

所以直接通过 make.sh 脚本编译生成的 boot.img 并不是最终使用的 boot.img。可通过在<SDK>顶层目录下如下执行如下命令生成 FIT 格式的 boot.img：

device/rockchip/common/mk-fitimage.sh kernel/boot.img device/rockchip/rk356x/boot.its

使用<SDK>/device/rockchip/common/mk-fitimage.sh 脚本生成 FIT 格式 boot.img 镜像。

第一个参数：表示输出文件 boot.img 的输出路径，如果使用相对地址、则必须相对于 SDK 源码根目录而言；

第二个参数：指定 its 源文件的路径，如果使用相对地址、则必须相对于 SDK 源码根目录而言；前面讲过，FIT 格式需要使用 its 文件来描述 image 的信息，所以这里需要指定一个 its 源文件。

执行上述命令，会将内核镜像、内核 DTB 以及资源镜像 resource.img 打包成一个 FIT 格式boot.img；命令中，并未指定内核镜像、内核 DTB 以及 resource.img 的路径，因为它们都有默认值（阅读 mk-fitimage.sh 脚本可知），如下：

内核镜像： 板级配置文件中 RK\_KERNEL\_IMG 变量所指定的镜像

内核 DTB： kernel/arch/arm64/boot/dts/rockchip/$RK\_KERNEL\_DTS.dtb

资源镜像 resource.img： kernel/resource.img

RK\_KERNEL\_IMG 和 RK\_KERNEL\_DTS 都是板级配置文件中定义的变量：

“**Device Tree Blob**”表示该文件是一个 FIT 格式镜像。通过 mk-fitimage.sh 脚本打包生成的 boot.img 才是最终烧录到开发板 boot 分区（Linux 系统）的 boot.img。

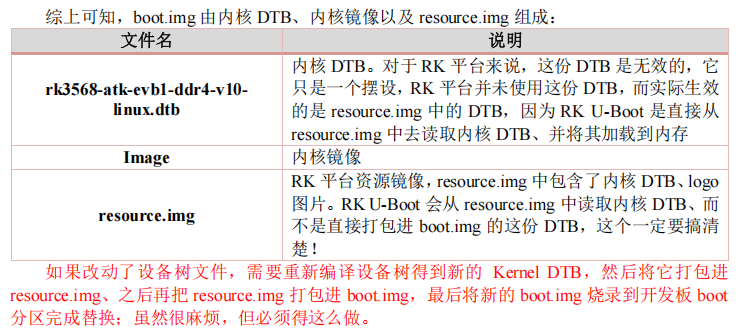
使用 mk-fitimage.sh 脚本生成 boot.img 镜像时，它会去<Kernel>/arch/arm64/boot/目录下寻找内核镜像 Image、<Kernel>/arch/arm64/boot/dts/rockchip/目录下寻找内核 DTB 以及<Kernel>/目录下寻找 resource.img：

**<Kernel>/arch/arm64/boot/Image**：内核镜像（由板级配置文件中的 RK\_KERNEL\_IMG 变量指定）。

**<Kernel>/arch/arm64/boot/dts/rk3568-atk-evb1-ddr4-v10-linux.dtb**：内核 DTB（由板级配置文件中的 RK\_KERNEL\_DTS 变量所决定！）。

**<Kernel>/resource.img**：resource 资源镜像（RK 自己定义的镜像类型，将 logo、dtb 等资源打包进 resource.img）。

所以运行脚本时，需确保这些镜像文件已经存在，否则会执行失败



* + 1. Resource.img镜像

resource.img 是 RK 自己设计的一种镜像，用于存放一些资源，譬如 u-boot logo 图片、内核logo 图片以及设备树镜像 DTB。resource.img 并不是单独烧录到开发板，而是将其打包进boot.img 中，最终烧录 boot.img。

通过<Kernel>/scripts/resource\_tool 工具可以生成 resource.img 镜像，可以将一个或多个资源 （DTB、图片资源等）打包进 resource.img 镜像中

mkdir temp\_dir

cd temp\_dir/

cp ../arch/arm64/boot/dts/rockchip/rk3568-atk-evb1-ddr4-v10-linux.dtb ./

cp ../logo.bmp ../logo\_kernel.bmp ./

../scripts/resource\_tool logo.bmp logo\_kernel.bmp rk3568-atk-evb1-ddr4-v10-linux.dtb

使用 scripts/resource\_tool 工具也可以将 resource.img 进行解包操作，执行如下命令进行解包：

rm -rf logo.bmp logo\_kernel.bmp rk3568-atk-evb1-ddr4-v10-linux.dtb

../scripts/resource\_tool –unpack

ls out/

解包后得到 u-boot logo 图片 logo.bmp、内核 logo 图片 logo\_kernel.bmp 以及一个 DTB 文件；因为制作 resource.img 时，会将 DTB 文件重命名为 rk-kernel.dtb。

用户可以将多个内核 DTB 打包进同一个 resource.img 镜像中，譬如 ATK-DLRK3568 开发 板出厂系统所使用的 resource.img 中就包含了 2 个内核 DTB，为了适配正点原子的两款 MIPI 屏（5.5 寸 720p 和 5.5 寸 1080p）。这两个 DTB 分别是：arch/arm64/boot/dts/rockchip/rk3568-atk-evb1-mipi-dsi-720p.dtb 和 arch/arm64/boot/dts/rockchip/rk3568-atk-evb1-mipi-dsi-1080p.dtb。

因为 RK 平台 U-Boot 支持 HW-ID DTB 功能，所以在 U-Boot 中可以根据用户接入的 MIPI 屏动态加载对应的内核 DTB；

用户接入的是 720p MIPI 屏，则加载 rk3568-atk-evb1-mipi-dsi-720p.dtb，

用户接入的是 1080p MIPI 屏，则加载 rk3568-atk-evb1-mipi-dsi-1080p.dtb。

制作包含多个内核 DTB 的 resource.img 时，可以使用 scripts/mkmultidtb.py 脚本去做，比直接使用 scripts/resource\_tool 工具更加方便； scripts/mkmultidtb.py 脚本依然是通过

scripts/resource\_tool 工具去制作 resource.img，不过在此之前，它还会对 DTB 文件名进行规则化，这是 U-Boot HW-ID DTB 所要求的，前面 6.1.5 小节已经给大家讲过。

使用 scripts/mkmultidtb.py 脚本制作 resource.img 之前，需要开发者提前配置好该脚本，打

开该文件

* + 1. 单独编译内核模块

运行 make.sh 脚本会编译整个内核源码，包括内核模块。当然我们也可以单独编译内核模块，这在调试内核模块时将很有用！单独编译内核模块之前需编译好内核源码，可执行如下命令编译内核模块：

make ARCH=arm64 modules -j24

Linux 系统下，内核模块最终需要拷贝到根文件系统的相应目录，譬如/system/lib/modules、 /usr/lib/modules 等；

对于 8852bs.ko（ATK-DLRK3568 开发板板载 WiFi RTL8852 的驱动模块）模块，编译 buildroot 根文件系统过程中，会自动将其拷贝到根文件系统 system/lib/modules 目录

下，最终打包生成根文件系统镜像（rootfs.img）。

编译 buildroot 根文件系统时，如果用户需要将自己的 ko 模块自动打包进 rootfs.img 镜像中，则 需要在 buildroot 中添加相应的配置（Config.in、xxx.mk），（编译时）让 buildroot 将用户指定的 ko 模块拷贝到根文件系统，一起打包生成 rootfs.img。

* 1. Kernel配置：

对内核源码进行配置，可执行如下命令打开 menuconfig 图形化配置界面（在内

核源码目录下执行）：

make ARCH=arm64 rockchip\_linux\_defconfig //加载配置文件

make ARCH=arm64 menuconfig //打开图形化配置界面

配置完成后可执行如下命令保存配置：

make ARCH=arm64 savedefconfig

cp defconfig arch/arm64/configs/rockchip\_linux\_defconfig

保存配置后，接下来我们可以回到 SDK 根目录下执行 build.sh 脚本编译内核源码：

cd ../ //回到 SDK 根目录下（当前处于内核源码目录下）

./build.sh kernel //执行 build.sh 脚本编译内核

除此之外，还可以在内核源码目录下执行 make.sh 脚本编译（在内核源码目录下执行）：

make clean //先清理工程

./make.sh //执行脚本进行编译

../device/rockchip/common/mk-fitimage.sh kernel/boot.img device/rockchip/rk356x/boot.its

* 1. 内核设备树和defconfig配置文件
     1. Rk3568-atk-evb1-ddr4-v10-linux.dts:

Linux Kernel 所使用的 defconfig 配置文件为 **<Kernel>/arch/arm64/configs/rockchip\_linux\_defconfig**

Rockchip 平台的所有设备树文件都存放在**<Kernel>arch/arm64/boot/dts/rockchip/**目录下。 对于 ATK-DLRK3568 开发板来说，使用的设备树文件为： **rk3568-atk-evb1-ddr4-v10-linux.dts**

**（rk3568-atk-atompi-ca1.dts）**

rk3568-atk-evb1-ddr4-v10-linux.dts 包含有多个.dtsi 设备树，如下所示：

**rk3568-atk-evb1-ddr4-v10-linux.dts**

rk3568-atk-evb1-ddr4-v10.dtsi

rk3568.dtsi

rk3568-evb.dtsi

rk3568-linux.dtsi

rk3568-screen\_choose.dtsi

rk3568-lcds.dtsi

\* rk3568.dtsi：该设备树文件是 RK3568 平台级设备树文件，与具体开发板硬件无关，纯

SoC 级别的设备树文件，由 RK 提供，开发者无需改动该文件。

\* rk3568-evb.dtsi：RK3568 板级通用设备树文件，通常被板级设备树文件所包含；

\* rk3568-linux.dtsi：该设备树包含 Linux 部分特有配置信息（与 Android 区分）； 仅找到一个rk3568-nvr-linux.dtsi文件

\* rk3568-atk-evb1-ddr4-v10.dtsi：ATK-DLRK3568 开发板板级设备树文件，该设备树文件包

含板级通用设备树文件 rk3568-evb.dtsi 以及 RK3568 平台级设备树文件 rk3568.dtsi；

\* rk3568-screen\_choose.dtsi：该设备树用于选择需要使能的 LCD 屏，譬如 HDMI、MIPI、

LVDS、VGA，支持单显、双显、三显。用户可在该文件中定义 ATK\_LCD\_TYPE\_XXX 宏（详

细内容请查看该文件）来使能对应的屏，譬如 ATK\_LCD\_TYPE\_HDMI 宏表示使能 HDMI

显示，没定义该宏表示禁用；

\* rk3568-lcds.dtsi：该设备树实现了 **ATK\_LCD\_TYPE\_XXX 宏控制 LCD 屏使能**的逻辑 ，

rk3568-lcds.dtsi 设备树会根据用户定义的 ATK\_LCD\_TYPE\_XXX 宏，使能对应的显示接口

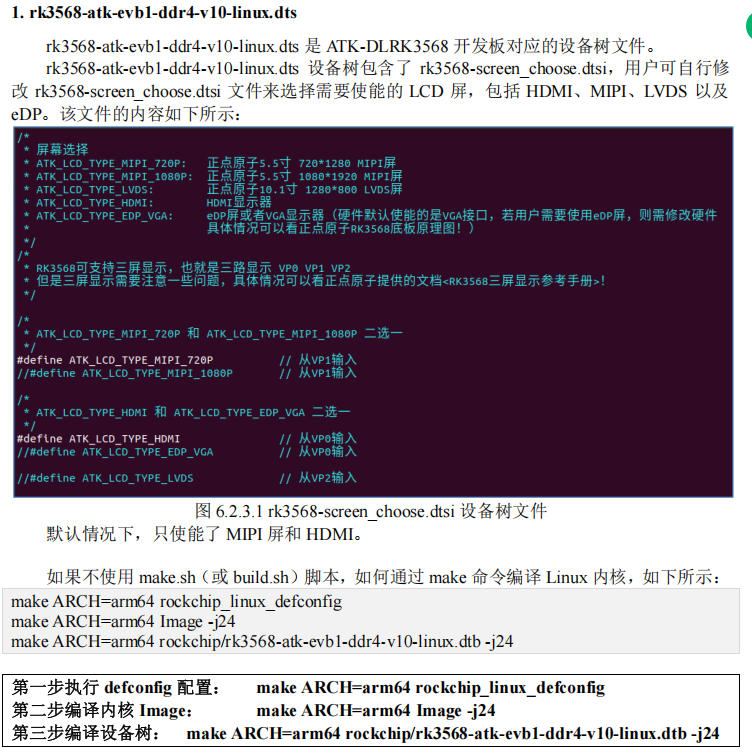
模块、并使能与该屏相关联的外设，譬如触摸屏（只有使能 MIPI 或 LVDS 屏时才会使

能触摸屏）、HDMI 音频（只有使能 HDMI 时才会使能 HDMI 音频）、背光（只有使能

MIPI 或 LVDS 时才会使能背光）。

rk3568-atk-evb1-ddr4-v10-linux.dts 设备树是提供给用户使用的，用户在编译内核源码时只需编译这个设备树即可！

rk3568-atk-atompi-ca1.dts：



执行上述命令编译成功后将会得到内核镜像 Image（arch/arm64/boot/Image）以及 Kernel DTB（arch/arm64/boot/dts/rockchip/rk3568-atk-evb1-ddr4-v10-linux.dtb）。

执行如下命令将 Kernel DTB、logo 图片打包到 resource.img 中：

cp arch/arm64/boot/dts/rockchip/rk3568-atk-evb1-ddr4-v10-linux.dtb ./

./scripts/resource\_tool logo.bmp logo\_kernel.bmp rk3568-atk-evb1-ddr4-v10-linux.dtb

rm -rf ./rk3568-atk-evb1-ddr4-v10-linux.dtb

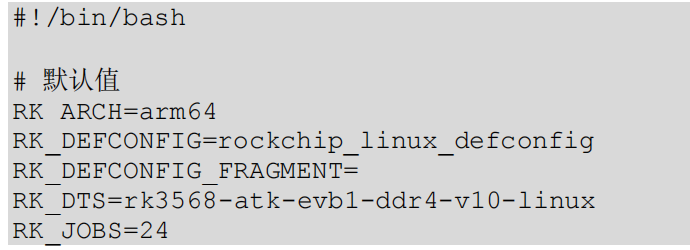
最后，将 Kernel DTB、resource.img（包含了 Kernel DTB）、内核镜像 Image 打包成一个 boot.img，如下所示：

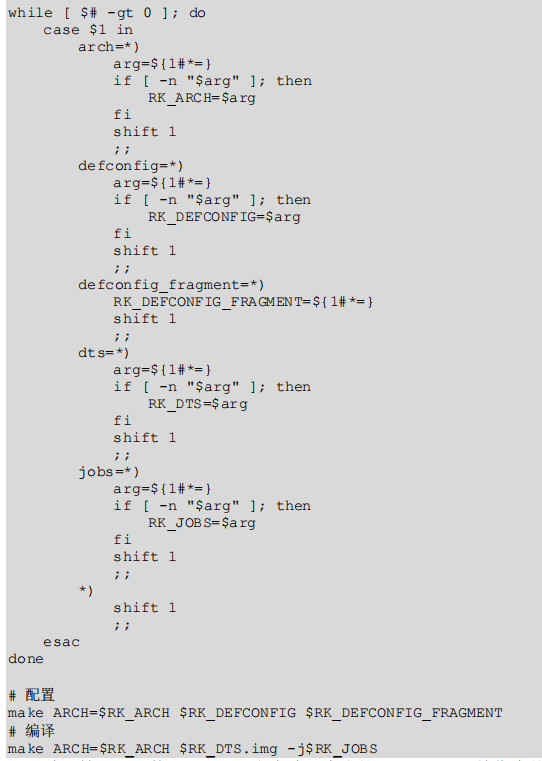
../device/rockchip/common/mk-fitimage.sh kernel/boot.img device/rockchip/rk356x/boot.its

直接通过 mk-fitimage.sh 脚本生成<Kernel>/boot.img。

* + 1. 修改make.sh脚本

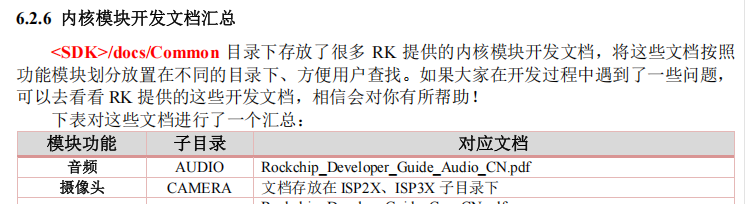
用户拿到 SDK 后，需对内核源码根目录下的 make.sh 脚本进行修改；打开<kernel>/make.sh 脚本文件，然后对其进行修改，修改之后的内容如下所示：

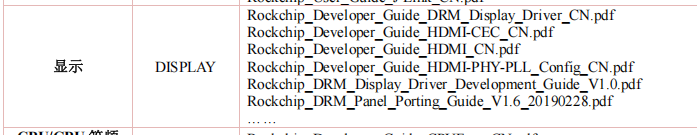




默认情况下，使用 make.sh 脚本编译生成的 resource.img 镜像中并未包含 rk3568-atk-evb1-ddr4-v10-linux.dtb，导致 rk3568-atk-evb1-ddr4-v10-linux.dts 设备树配置不会生效，所以需要对其进行修改。

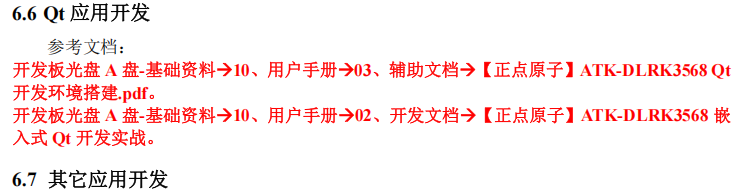
* + 1. 内核模块开发文档汇总

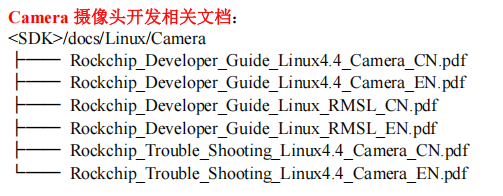




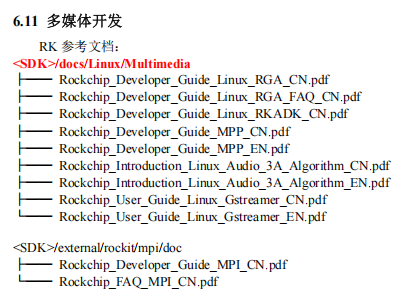
IMG_256

1. qt应用开发









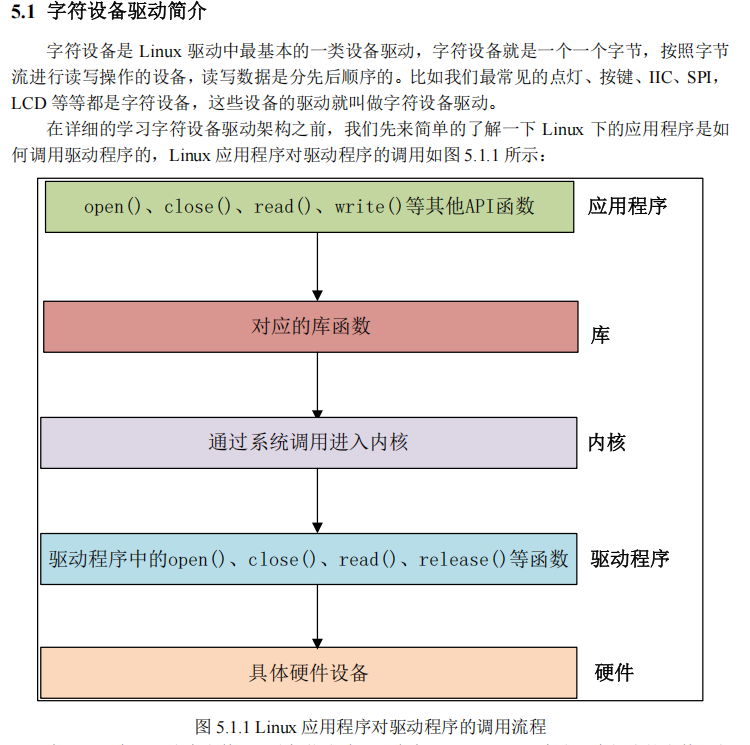
不能直接通过输入“aarch64-buildroot-linux-gnu-gcc -v”这样来使用交

叉编译器了，而是需要输入绝对路径，也就是：

/opt/atk-dlrk356x-toolchain/bin/aarch64-buildroot-linux-gnu-gcc -v

若要卸载交叉编译器，进入安装目录/opt 下直接删除即可，命令如下：

sudo rm -f atk-dlrk356x-toolchain/



在 Linux 内核文件 include/linux/fs.h 中有个 file\_operations 的结构体

简单介绍一下 file\_operation 结构体中比较重要的、常用的函数：

第 1771 行，owner 拥有该结构体的模块的指针，一般设置为 THIS\_MODULE。

第 1772 行，llseek 函数用于修改文件当前的读写位置。

第 1773 行，read 函数用于读取设备文件。

第 1774 行，write 函数用于向设备文件写入(发送)数据。

第 1779 行，poll 是个轮询函数，用于查询设备是否可以进行非阻塞的读写。

第 1780 行，unlocked\_ioctl 函数提供对于设备的控制功能，与应用程序中的 ioctl 函数对应。

第 1781 行，compat\_ioctl 函数与 unlocked\_ioctl 函数功能一样，区别在于在 64 位系统上，

32 位的应用程序调用将会使用此函数。在 32 位的系统上运行 32 位的应用程序调用的是

unlocked\_ioctl。

第 1782 行，mmap 函数用于将将设备的内存映射到进程空间中(也就是用户空间)，一般帧

缓冲设备会使用此函数，比如 LCD 驱动的显存，将帧缓冲(LCD 显存)映射到用户空间中以后应

用程序就可以直接操作显存了，这样就不用在用户空间和内核空间之间来回复制。

第 1784 行，open 函数用于打开设备文件。

第 1786 行，release 函数用于释放(关闭)设备文件，与应用程序中的 close 函数对应。

第 1788 行，fasync 函数用于刷新待处理的数据，用于将缓冲区中的数据刷新到磁盘中。

驱动加载：

insmod 命令不能解决模块的依赖关系，比如 drv.ko 依赖 first.ko 这个模块，就必须先使用 insmod 命令加载 first.ko 这个模块，然后再加载 drv.ko 这个模块。

但是 modprobe 就不会存在这个问题，modprobe 会分析模块的依赖关系，然后会将所有的依赖模块都加载到内核中，因此modprobe 命令相比 insmod 要智能一些。

modprobe 命令默认会去 /lib/modules/<kernel-version>目录中查找模块。

驱动卸载：

驱动模块的卸载使用命令“rmmod”即可。rmmod drv.ko

也可以使用“modprobe -r”命令卸载驱动modprobe -r drv

使用 modprobe 命令可以卸载掉驱动模块所依赖的其他模块，前提是这些依赖模块已经没有被其他模块所使用，否则就不能使用 modprobe 来卸载驱动模块。

**字符设备注册与注销**

对于字符设备驱动而言，当驱动模块加载成功以后需要注册字符设备，同样，卸载驱动模块的时候也需要注销掉字符设备。字符设备的注册和注销函数原型如下所示:

static inline int register\_chrdev(unsigned int major, const char \*name, const struct file\_operations \*fops)

static inline void unregister\_chrdev(unsigned int major,const char \*name)

register\_chrdev 函数用于注册字符设备，此函数一共有三个参数，这三个参数的含义如下:

**major：**主设备号，Linux 下每个设备都有一个设备号，设备号分为主设备号和次设备号两部分，关于设备号后面会详细讲解。

**name**：设备名字，指向一串字符串。

**fops：**结构体 file\_operations 类型指针，指向设备的操作函数集合变量。

unregister\_chrdev 函数用户注销字符设备，此函数有两个参数，这两个参数含义如下：

**major：**要注销的设备对应的主设备号。

**name：**要注销的设备对应的设备名。

命令：cat /proc/devices 查询已经使用的设备号

设备号

Linux 提供了一个名为 dev\_t 的数据类型表示设备号。设备号由主设备号和次设备号两部分组成。

在文件 include/linux/kdev\_t.h 中提供了几个关于设备号的操作函数(本质是宏)：

#define MINORBITS 20

#define MINORMASK **((**1U **<<** MINORBITS**) -** 1**)**

#define MAJOR**(**dev**)** **((**unsigned int**) ((**dev**) >>** MINORBITS**))**

#define MINOR**(**dev**)** **((**unsigned int**) ((**dev**) &** MINORMASK**))**

#define MKDEV**(**ma**,**mi**) (((**ma**) <<** MINORBITS**) | (**mi**))**

第 7 行，宏 MINORBITS 表示次设备号位数，一共是 20 位。

第 8 行，宏 MINORMASK 表示次设备号掩码。

第 10 行，宏 MAJOR 用于从 dev\_t 中获取主设备号，将 dev\_t 右移 20 位即可。

第 11 行，宏 MINOR 用于从 dev\_t 中获取次设备号，取 dev\_t 的低 20 位的值即可。

第 12 行，宏 MKDEV 用于将给定的主设备号和次设备号的值组合成 dev\_t 类型的设备号。

**1、静态分配设备号**

使用“cat /proc/devices”命令即可查看当前系统中所有已经使用了的设备号，对于没使用的设备号，我们手动分配给设备。

**2、动态分配设备号**

设备号的申请函数如下：

int alloc\_chrdev\_region(dev\_t \*dev, unsigned baseminor, unsigned count, const char \*name)

函数 alloc\_chrdev\_region 用于申请设备号，此函数有 4 个参数：

**dev**：保存申请到的设备号。

**baseminor：**次设备号起始地址，alloc\_chrdev\_region 可以申请一段连续的多个设备号，这些设备号的主设备号一样，但是次设备号不同，次设备号以 baseminor 为起始地址地址开始递增。一般 baseminor 为 0，也就是说次设备号从 0 开始。

**count：**要申请的设备号数量。

**name**：设备名字。

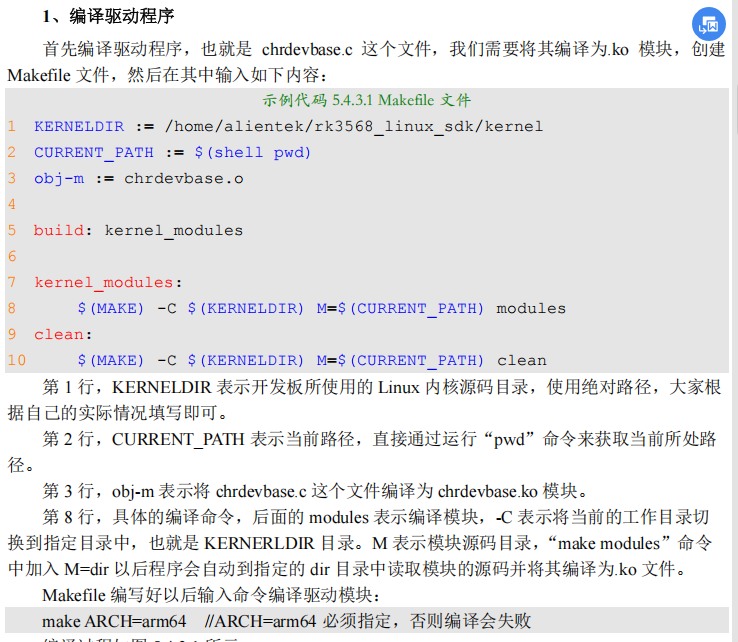
设备号释放函数如下：

void unregister\_chrdev\_region(dev\_t from, unsigned count)

此函数有两个参数：

**from**：要释放的设备号。

**count：**表示从 from 开始，要释放的设备号数量。





1. **手动创建设备节点文件：**

mknod /dev/chrdevbase c 200 0

“mknod”是创建节点命令，“/dev/chrdevbase”是要创建的节点文件，“c”表示这是个字符设备，“200”是设备的主设备号，“0”是设备的次设备号。创建完成以后就会存在/dev/chrdevbase 这个文件，可以使用“ls /dev/chrdevbase -l”命令查看

物理内存和虚拟内存之间的转换:

**1、ioremap 函数**

ioremap 函 数 用 于 获 取 指 定 物 理 地 址 空 间 对 应 的 虚 拟 地 址 空 间 ， 定 义 在arch/arm/include/asm/io.h 文件中。函数的实现是在 arch/arm/mm/ioremap.c 文件中：

void \_\_iomem **\***ioremap**(**resource\_size\_t res\_cookie**,** size\_t size**);**

ioremap 有两个参数：

**res\_cookie**：要映射的物理起始地址。

**size**：要映射的内存空间大小。

**返回值：**\_\_iomem 类型的指针，指向映射后的虚拟空间首地址。

(由函数实现的源码知：真正起作用的是函数 arch\_ioremap\_caller。 )

**2、iounmap 函数**

卸载驱动的时候需要使用 iounmap 函数释放掉 ioremap 函数所做的映射。

void iounmap **(**volatile void \_\_iomem **\***addr**)**

iounmap 只有一个参数 addr，就是 要取消映射的 虚拟地址空间首地址。

**6.1.2 I/O 内存访问函数**

I/O 是输入/输出的意思，不是 GPIO 引脚。

这里涉及到两个概念：

I/O 端口和 I/O 内存。

当外部寄存器或内存映射到 IO 空间时，称为 I/O 端口。

当外部寄存器或内存映射到内存空间时，称为 I/O 内存。

但是对于 ARM 来说没有 I/O 空间这个概念，因此 ARM 体系下只有 I/O 内存(可以直接理解为内存)。

使用 ioremap 函数将寄存器的物理地址映射到虚拟地址以后，我们就可以直接通过指针访问这些地址，但是 Linux 内核不建议这么做，而是推荐使用一组操作函数来对映射后的内存进行读写操作，而不是直接 指针访问 映射后的虚拟地址。

**1、读操作函数**

u8 readb**(**const volatile void \_\_iomem **\***addr**)**

u16 readw**(**const volatile void \_\_iomem **\***addr**)**

u32 readl**(**const volatile void \_\_iomem **\***addr**)**

readb、readw 和 readl 这三个函数分别对应 8bit、16bit 和 32bit 读操作，参数 addr 就是要读取写内存地址，返回值就是读取到的数据。

**2、写操作函数**

void writeb**(**u8 value**,** volatile void \_\_iomem **\***addr**)**

void writew**(**u16 value**,** volatile void \_\_iomem **\***addr**)**

void writel**(**u32 value**,** volatile void \_\_iomem **\***addr**)**

writeb、writew 和 writel 这三个函数分别对应 8bit、16bit 和 32bit 写操作，参数 value 是要写入的数值，addr 是要写入的地址。

**7.1 新字符设备驱动原理**

register\_chrdev 函数注册字符设备的时候只需要给定一个主设备号，但是这样会带来两个问题：

①、需要我们事先确定好哪些主设备号没有使用。

②、会将一个主设备号下的所有次设备号都使用掉，比如现在设置 LED 这个主设备号为

200，那么 0~1048575(2^20-1)这个区间的次设备号就全部都被 LED 一个设备分走了。

解决这两个问题最好的方法就是在使用设备号的时候向 Linux 内核申请，需要几个就申请几个，由 Linux 内核分配设备可以使用的设备号。

因此：

没有指定设备号则使用如下函数来申请设备号：

int alloc\_chrdev\_region(dev\_t \*dev, unsigned baseminor, unsigned count, const char \*name)

如果给定了设备的主设备号和次设备号就使用如下所示函数来注册设备号即可：

int register\_chrdev\_region(dev\_t from, unsigned count, const char \*name)

参数 from 是要申请的起始设备号，也就是给定的设备号；参数 count 是要申请的数量，一 般都是一个；参数 name 是设备名字。

注 销 字 符 设 备 之 后 要 释 放 掉 设 备 号 ， 不 管 是 通 过

alloc\_chrdev\_region 函数 还是 register\_chrdev\_region 函数 申请的设备号

统一使用如下释放函数：

void unregister\_chrdev\_region(dev\_t from, unsigned count)

**7.1.2 新的字符设备注册方法**

在 Linux 中使用 cdev 结构体表示一个字符设备，cdev 结构体在 include/linux/cdev.h 文件中，定义如下：

struct cdev **{**

struct kobject kobj**;**

struct module **\***owner**;**

const struct file\_operations **\***ops**;**

struct list\_head list**;**

dev\_t dev**;**

unsigned int count**;**

**}** \_\_randomize\_layout;

在 cdev 中有两个重要的成员变量：ops 和 dev，这两个就是字符设备文件操作函数集合

file\_operations 以及设备号 dev\_t。编写字符设备驱动之前需要定义一个 cdev 结构体变量，这个变量就表示一个字符设备，

**2、cdev\_init 函数**

定义好 cdev 变量以后就要使用 cdev\_init 函数对其进行初始化。

cdev 就是要初始化的 cdev 结构体变量，参数 fops 就是字符设备文件操作函数集合。

struct cdev testcdev**;**

/\* 设备操作函数 \*/

static struct file\_operations test\_fops **= {**

**.**owner **=** THIS\_MODULE**,**

/\* 其他具体的初始项 \*/

**};**

testcdev**.**owner **=** THIS\_MODULE**;**

cdev\_init**(&**testcdev**, &**test\_fops**);** /\* 初始化 cdev 结构体变量 \*/

**3、cdev\_add 函数**

cdev\_add 函数用于向 Linux 系统添加字符设备(cdev 结构体变量)，首先使用 cdev\_init 函数 完成对 cdev 结构体变量的初始化，然后使用 cdev\_add 函数向 Linux 系统添加这个字符设备。

int cdev\_add(struct cdev \*p, dev\_t dev, unsigned count)

参数 p 指向要添加的字符设备(cdev 结构体变量)，

参数 dev 就是设备所使用的设备号，

参数 count 是要添加的设备数量。

**4、cdev\_del 函数**

卸载驱动的时候一定要使用 cdev\_del 函数从 Linux 内核中删除相应的字符设备。

void cdev\_del(struct cdev \*p)

参数 p 就是要删除的字符设备

**7.2 自动创建设备节点**

在前面的 Linux 驱动实验中，当我们使用 modprobe 加载驱动程序以后还需要使用命令 “mknod”手动创建设备节点。

本节就来讲解一下如何实现自动创建设备节点，在驱动中实现自动创建设备节点的功能以后，使用 modprobe 加载驱动模块成功的话就会自动在/dev 目录下创建对应的设备文件。

**7.2.0 mdev 机制**

udev 是一个用户程序，在 Linux 下通过 udev 来实现设备文件的创建与删除。

udev 可以检测系统中硬件设备状态，可以根据系统中硬件设备状态来创建或者删除设备文件。

比如使用modprobe 命令成功加载驱动模块以后就自动在/dev 目录下创建对应的设备节点文件,使用rmmod 命令卸载驱动模块以后就删除掉/dev 目录下的设备节点文件。

开发板启动的时候会启动 udev，

**7.2.1 创建和删除类**

**创建类：**

自动创建设备节点的工作是在驱动程序的入口函数中完成的，一般在 cdev\_add 函数后面添加自动创建设备节点相关代码。

首先要创建一个 class 类，class 是个结构体，定义在文件include/linux/device.h 里面。class\_create 是类创建函数，class\_create 是个宏定义，内容如下：

/\*

extern struct class **\*** \_\_must\_check \_\_class\_create**(**struct module **\***owner**,**const char **\***name**,**struct lock\_class\_key **\***key**);**

extern void class\_destroy**(**struct class **\***cls**);**

#define class\_create**(**owner**,** name**)** **({**static struct lock\_class\_key \_\_key**;**\_\_class\_create**(**owner**,** name**, &**\_\_key**);** **})**

**\*/**

根据上述代码，将宏 class\_create 展开以后内容如下：

struct class \*class\_create (struct module \*owner, const char \*name)

class\_create 一共有两个参数：

参数 owner 一般为 THIS\_MODULE，

参数 name 是 类名字。

返回值是个指向结构体 class 的指针，也就是创建的类。

删除类：

卸载驱动程序的时候需要删除掉类，类删除函数为 class\_destroy，函数原型如下：

void class\_destroy(struct class \*cls);

参数 cls 就是要删除的类。

**7.2.2 创建设备**

上一小节创建好类以后还不能实现自动创建设备节点，我们还需要在这个类下创建一个设备。

使用 device\_create 函数在类下面创建设备，device\_create 函数原型如下：

struct device \*device\_create(struct class \*class,

struct device \*parent,

dev\_t devt,

void \*drvdata,

const char \*fmt, ...)

device\_create 是可以 变参数 的函数。

参数 class 就是设备要到创建哪个类下面；

参数 parent 是父设备，一般为 NULL，也就是没有父设备；

参数 devt 是设备号；

参数 drvdata 是设备可能会使用的一些数据，一般为 NULL；

参数 fmt 是设备名字，如果设置 fmt=xxx 的话，就会生成/dev/xxx这个设备文件。

返回值就是创建好的设备。

删除设备：

卸载驱动的时候需要删除掉创建的设备，设备删除函数为 device\_destroy，函数原型如下：

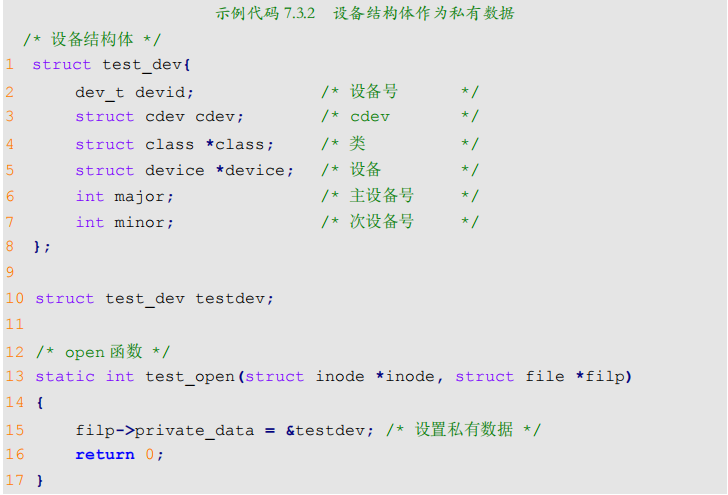
void device\_destroy(struct class \*cls, dev\_t devt)

参数 classs 是要删除的设备所处的类，参数 devt 是要删除的设备号。



**7.3 设置文件私有数据**

每个硬件设备都有一些属性，比如主设备号(dev\_t)，类(class)、设备(device)、开关状态(state) 等等，在编写驱动的时候 你可以将这些属性 全部写成 结构体成员 的形式，如下所示：



在 open 函数里面设置好私有数据以后，在 write、read、close 等函数中直接读取 private\_data 即可得到设备结构体。

————————————————————————————————————

**第八章 Linux 设备树**

我们所使用的 Linux 版本为 4.19.232

**8.1 什么是设备树？**

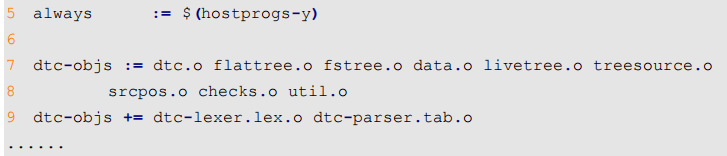
描述设备树的文件叫做 DTS(Device Tree Source)，这个 DTS 文件采用树形结构描述板级设备，也就是开发板上的设备信息，比如CPU 数量、 内存基地址、IIC 接口上接了哪些设备、SPI 接口上接了哪些设备等。

**8.2 DTS、DTB 和 DTC**

设备树 源文件 扩展名为.dts，但是我们在前面移植 Linux 的时候却一直在使用.dtb 文件，那么 DTS 和 DTB 这两个文件是什么关系呢？

DTS 是设备树源码文件，DTB 是将DTS 编译以后得到的二进制文件。将.c 文件编译为.o 需要用到 gcc 编译器，那么将.dts 编译为.dtb 需要用到 DTC 工具！DTC 工具源码在 Linux 内核的 scripts/dtc 目录下，scripts/dtc/Makefile 文件内容如下：

IMG_256



可以看出，DTC 工具依赖于 dtc.c、flattree.c、fstree.c 等文件，最终编译并链接出 DTC 这个主机文件。

如果要编译 DTS 文件，只需要进入到 Linux 源码根目录下，然后执行如下命

令：(对于 RK3568，需要指定 ARCH=arm64)

make ARCH=arm64 all

(编译 Linux 源码中的所有东西，包括 uImage/zImage，.ko 驱动模块以及设备树)

或者：

make ARCH=arm64 dtbs

编译选中的所有设备树文件。

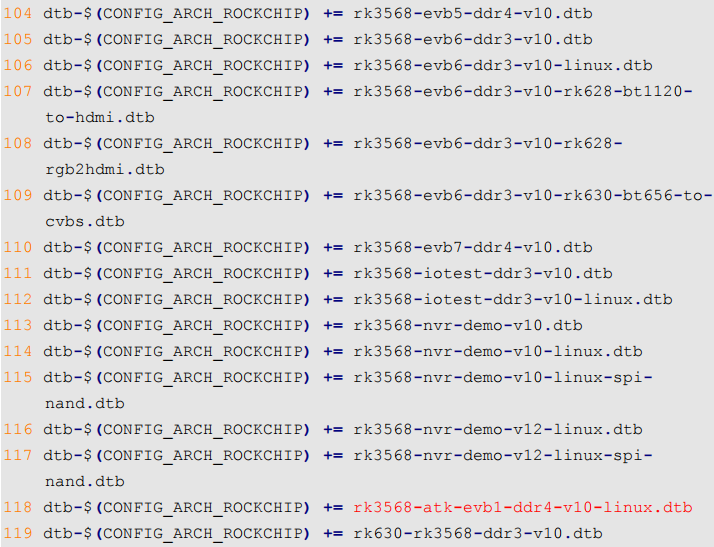
或者

make ARCH=arm64 rockchip/rk3568-atk-evb1-ddr4-v10-linux.dtb

只编译指定的某个设备树。

基于 ARM 架构的 SOC 有很多种，一种 SOC 又可以制作出很多款板子，每个板子都有一个对应的 DTS 文件，那么如何确定编译哪一个 DTS 文件呢？我们就以 RK3568 这款芯片对应的板子为例来看一下，打开 arch/arm64/boot/dts/rockchip/Makefile，有如下内容:





可以看出，比如 这个 Makefile 下有许多 RK3326、RK3566、RK3568 等不同的.dtb 文件。

如果我们使用 RK3568 新做了一个板子，只需要新建一个此板子对应的.dts 文件，然后将对应的.dtb 文件名添加到这个 Makefile 下。

这样在编译设备树的时候就会将对应的.dts 编译为二进制的.dtb 文件。

示例代码 8.2.2 中 118 行就是我们在给正点原子的开发板移植 Linux 系统的时候添加的设备树。

**8.3 DTS 语法**

以 rk3568-atk-evb1-ddr4-v10-linux.dts 这个文件为例来讲解一下 DTS 语法。

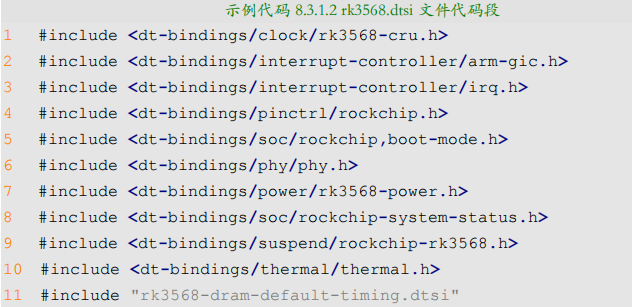
关于设备树详细的语法规则参考《Devicetree SpecificationV0.2.pdf》和《Power\_ePAPR\_APPROVED\_v1.12.pdf》这两份文档。

**8.3.1 .dtsi 头文件**

和 C 语言一样，设备树也支持头文件，设备树的头文件扩展名为.dtsi。设备树里面除了可以通过“#include”来引用.dtsi 文件，也可以引用.h 头文件甚至也可以引用.dts 文件。

因此在.dts 设备树文件中，可以通过“#include”来引用.h、.dtsi 和.dts 文件。只是，我们在编写设备树头文件的时候最好选择.dtsi 后缀。

在 rk3568-atkevb1-ddr4-v10.dtsi 中有如下所示内容：



一般.dtsi 文件用于描述 SOC 的内部外设信息，比如 CPU 架构、主频、外设寄存器地址范围，比如 UART、IIC 等。

**8.3.2 设备节点**

设备树是采用树形结构来描述板子上的设备信息的文件，每个设备都是一个节点，叫做设备节点，每个节点都通过一些属性信息来描述节点信息，属性就是键—值对。

“/”是根节点，每个设备树文件只有一个根节点。

aliases、cpus 和 i2c0 是根节点“/”的三个子节点，节点命名格式为：

1.

node-name@unit-address 例如：“i2c@ff3f0000

“node-name”是节点名字。

“unit-address”一般表示设备的地址或寄存器首地址。

2.

label: node-name@unit-address 例如：“i2c0: i2c@ff3f0000”

**label是节点标签**（类似于重命名），目的是方便访问节点（使用&label来访问这个节点）。例如使用 “&i2c0 ”就可以访问 “i2c@ff3f0000”这个节点。

“node-name”还是节点名字。

“unit-address”仍表示设备的地址或寄存器首地址。

每个节点都有不同属性，不同的属性又有不同的内容，属性都是键值对，值可以为空或任意的字节流。设备树源码中常用的几种数据形式如下所示：

**1、字符串**

compatible = "rockchip,rk3568";

上述代码设置 compatible 属性的值为字符串“rockchip,rk3568”。

**2、32 位无符号整数**

reg = <0>;

上述代码设置 reg 属性的值为 0，reg 的值也可以设置为一组值，比如：

reg = <0 0x123456 100>;

**3、字符串列表**

属性值也可以为 字符串 列表，字符串和字符串之间采用“,”隔开，如下所示：

compatible = "rockchip,rk3568-evb ", "rockchip,rk3568";

上述代码设置属性 compatible 的值为“rockchip,RK3568-evb”和“rockchip,rk3568”

**8.3.3 标准属性**

节点是由一堆的属性组成，节点都是具体的设备，不同的设备需要的属性不同，**用户可以自定义属性。除了用户自定义属性，有很多属性是标准属性**，Linux 下的很多外设驱动都会使用这些标准属性:

**1、compatible 属性**

compatible 属性也叫做“兼容性”属性,属性的值是一个字符串列表，compatible 属性用于将设备和驱动绑定起来。字符串列表用于选择设备所要使用的驱动程序。属性格式为：

"manufacturer,model"

manufacturer 表示厂商，model 一般是模块对应的驱动名字，

例如：compatible ="sony,imx415";

compatible 也可以多个属性值。比如：

compatible = "ilitek,ili9881d", "simple-panel-dsi";

这样我们的设备就有两个属性值，这个设备首先使用第一个兼容值在 Linux 内核里面查找，看看能不能找到与之匹配的驱动文件，如果没有找到的话就使用第二个兼容值查，以此类推。

一般驱动程序文件 会有一个 OF 匹配表，此 OF 匹配表保存着一些 compatible 值，如果设备节点的 compatible 属性值和 OF 匹配表中的任何一个值相等，那么就表示设备可以使用这个驱动。

例如：

示例代码 8.3.3.1 imx415.c 文件代码段

static const struct of\_device\_id imx415\_of\_match**[] = {**

**{ .**compatible **=** "sony,imx415" **},**

**{},**

**};**

imx415\_of\_match 就是 imx415.c 这个驱动文件的匹配表，此匹配表只有一个匹配值

“sony,imx415”。如果在设备树中 有哪个节点的 compatible 属性值与此相等，那么这个节点就会使用此驱动文件。

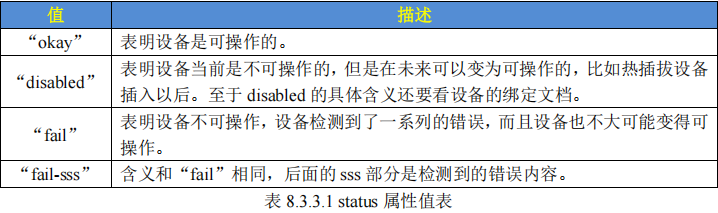
**2、model 属性**

model 属性值也是一个字符串，一般 model 属性描述开发板的名字或者设备模块信息，比如名字什么的，比如：

model = "Rockchip rk3568 EVB DDR4 V10 Board";

**3、status 属性**

status 属性看名字就知道是和设备状态有关的，status 属性值也是字符串，字符串是设备的状态信息，可选的状态如下：



**4、#address-cells 和#size-cells 属性**

这两个属性的值都是无符号 32 位整形，这两个属性可以用在任何拥有子节点的设备中，用于描述子节点的地址信息。

#address-cells 属性值决定了子节点 reg 属性中 地址信息 所占用的字长(32 位)，

#size-cells 属性值决定了子节点 reg 属性中 长度信息 所占的字长(32 位)。

#address-cells 和#size-cells 表明了子节点应该如何编写 reg 属性值，一般 reg 属性

都是和地址有关的内容，和地址相关的信息有两种：起始地址和地址长度，reg 属性的格式为：

reg = <address1 length1 address2 length2 address3 length3……>

每个“address length”组合表示一个地址范围，其中

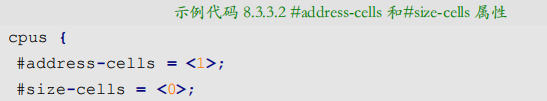
address 是起始地址，

length 是地址长度

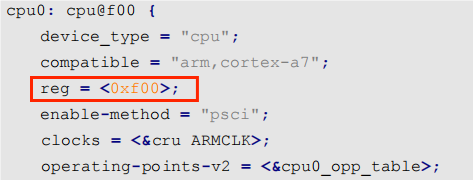
#address-cells 表明 address 这个数据所占用的字长。

#size-cells 表明 length 这个数据所占用的字长，

例如：



说明 cpus 的子节点 reg 属性中 起始地址所占用的字长为1，地址长度所占用的字长为 0。

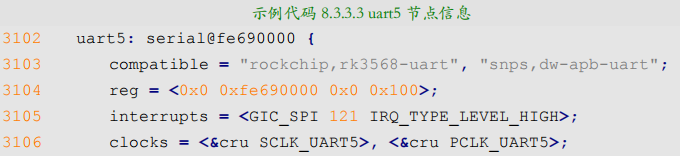


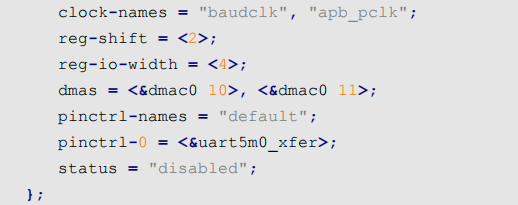
因为父节点设置了#address-cells = <1>，#size-cells = <0>，

因此 addres=0xf00，没有 length 的值，相当于设置了起始地址，没有设置地址长度。

**5、reg 属性**

reg 属性的值一般是(address，length)对，reg 属性一般用于描述设备地址空间资源信息或者设备地址信息。比如某个外设的寄存器地址范围信息，或者 IIC器件的设备地址等：比如在 rk3568.dtsi 中：





由于 uart5 的父节点“/” 设置了#address-cells = <1>、#size-cells = <1>，因此 reg 属性中address=0xfe690000，length=0x100。

**6、ranges 属性**

ranges属性值可以为空 或者 按照(child-bus-address,parent-bus-address,length)格式编写的数字矩阵，ranges 是一个地址映射/转换表，ranges 属性每个项目由子地址、父地址和地址空间长度 这三部分组成：

**child-bus-address**：子总线地址空间的物理地址，由父节点的#address-cells 确定此物理地址所占用的字长。

**parent-bus-address：**父总线地址空间的物理地址，由父节点的#address-cells 确定此物理地址所占用的字长。

**length：**子地址空间的长度，由父节点的#size-cells 确定此地址长度所占用的字长。

如果 ranges 属性值为空值，说明子地址空间和父地址空间完全相同，不需要进行地址转换。

对于我们所使用的 RK3568 来说，子地址空间和父地址空间完全相同，因此会在 rk3568.dtsi 中有大量的值为空的 ranges 属性。

**7、name 属性**

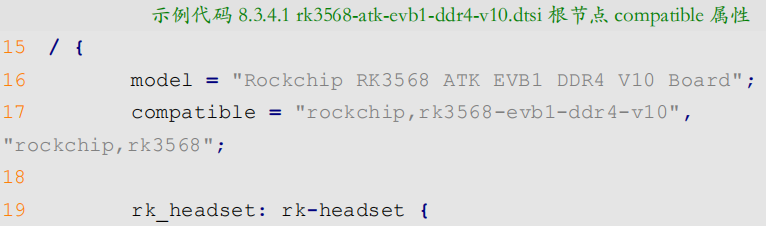
name 属性值为字符串，name 属性用于记录节点名字，name 属性已经被弃用，不推荐使用name 属性，一些老的设备树文件可能会使用此属性。

**8、device\_type 属性**

device\_type 属性值为字符串，IEEE 1275 会用到此属性，用于描述设备的 FCode，但是设备树没有 FCode，所以此属性也被抛弃了。

**8.3.4 根节点 compatible 属性**

每个节点都有 compatible 属性，根节点“/”也是。



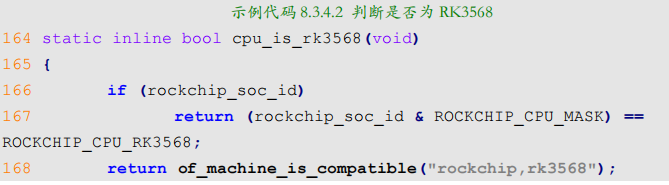
设备节点的 compatible 属性值是为了匹配 Linux 内核中的驱动程序

根节点的 compatible 属性可以知道我们所使用的设备。

一般第一个值描述了所使用的硬件设备名字，比如这里使用的是“rk3568-evb1- ddr4-v10”这个设备。

第二个值描述了设备所使用的 SOC，比如这里使用的是“RK3568”这颗SOC。

Linux 内核会通过根节点的 compoatible 属性查看是否支持此设备，如果支持的话设备就会启动 Linux 内核，例如在开 include/linux/rockchip/cpu.h 文件，有如下内容：



函数 cpu\_is\_rk3568 用于判断当前是否为 RK3568，

of\_machine\_is\_compatible 函数判断根节点 compatible 值里面是否有“rockchip,rk3568”。

**8.3.5 向节点追加或修改内容**

产品开发过程中可能面临着-频繁的需求更改，比如第一版硬件上有一个 IIC 接口的六轴芯片 MPU6050，第二版硬件又要把这个 MPU6050 更换为 MPU9250 等。一旦硬件修改了，我们就要同步的修改设备树文件，毕竟设备树是描述板子硬件信息的文件。

假如现在于需要在 i2c5 这个节点上添加一个 fxls8471 子节点。先看一下 I2C5 接口对应的节点，打开文件 rk3568.dtsi 文件：

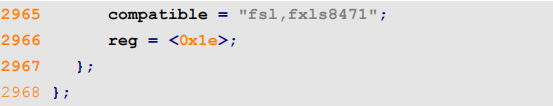
/\*

直接在定义定义I2C5的 rk3568.dtsi 文件中直接添加fxls8471 节点（是不可以的）：



现在要在 i2c5 节点下创建一个 fxls8471子节点，最简单的方法就是在 i2c5 下直接添加一个名为 fxls8471 的子节点：





但是i2c5 节点是定义在 rk3568.dtsi 文件中的，而 rk3568.dtsi 是共有的设备树头文件，其他所有使用到 rk3568 这颗SOC 的板子都会引用 rk3568.dtsi 这个文件。就相当于在

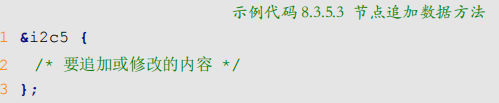
其他的所有板子上都添加了 fxls8471 这个设备。

\*/

因此我们现在需要向 i2c5 节点追加一个名为 fxls8471 的子节点，同时不能影响到其他使用到 RK3568 的板子。

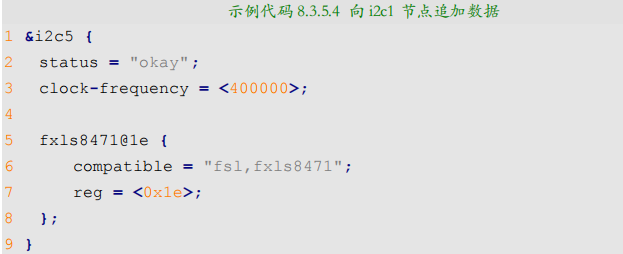
（Atompi开发板使用的设备树文件为 rk3568-atk-atompi-ca1.dts）

ATK-DLRK3568 开发板使用的设备树文件为 rk3568-atk-evb1-ddr4-v10.dtsi 和 rk3568-linux.dtsi。因此我们需要在rk3568-atk-evb1-ddr4-v10.dtsi文件中完成追加的内容：



&i2c5 表示要访问 i2c5 这个 label 所对应的节点，也就是 rk3568.dtsi 中的“i2c5:i2c@fe5e0000”。花括号内就是要向 i2c5 这个节点添加的内容，包括修改某些属性的值。

正确的做法就是在rk3568-atk-evb1-ddr4-v10.dtsi中，向i2c5节点追加fxls8471相关的信息:

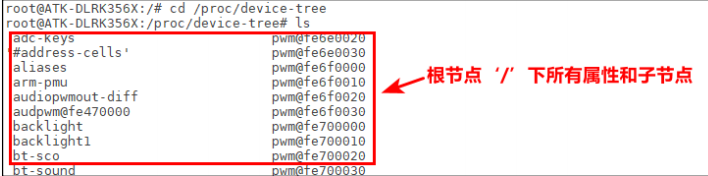


i2c5 子节点 fxls8471，表示 I2C5 上连接的 fxls8471，“fxls8471”子节点里面描述了 fxls8471 这颗芯片的相关信息。

因为示例代码 8.3.5.4 中的内容是 rk3568-atk-evb1-ddr4-v10.dtsi 这个文件内的，所以不会对使用 RK3568 这颗 SOC 的其他板子造成任何影响。

**8.5 设备树在系统中的体现**

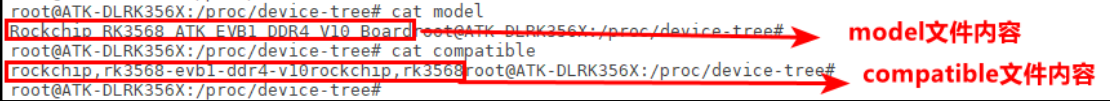
Linux 内核启动的时候会解析设备树中各个节点的信息，并且在根文件系统的/proc/devicetree 目录下根据节点名字创建不同文件夹：



/proc/device-tree 目录下是根节点“/”的所有属性和子节点。

**1、根节点“/”各个属性**

根节点属性表现为一个个的文件，比如图 8.5.1 中的“#address-cells”、“#sizecells”、“compatible”、“model”和“name”这 5 个文件，它们在设备树中就是根节点的 5 个属性。既然是文件那么肯定可以查看其内容，输入 cat 命令来查看 model 和 compatible 这两个文件的内容：



文件model的内容是“Rockchip RK3568 ATK EVB1 DDR4 V10 Board”，

文件 compatible 的内容为“rockchip,rk3568-evb1-ddr4-v10rockchip,rk3568”。

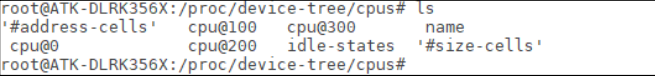
正好对应 rk3568-atk-evb1-ddr4-v10.dtsi 里，根节点“/”的 model 和 compatible 属性值

**2、根节点“/”下各子节点**

/proc/device-tree 目录下各个文件夹 就是 根节点“/”的各个子节点，比如“aliases”、“chosen”和“cpus” 等等。

大家可以查看一下 rk3568-atk-evb1-ddr4-v10.dtsi 和其所引用的所有.dtsi 文件，看看根节点的子节点都有哪些。

/proc/device-tree 目录就是设备树在根文件系统中的体现，同样是按照树形结构组织的，进入/proc/device-tree/cpus 目录中就可以看到 cpus 节点的所有子节点：



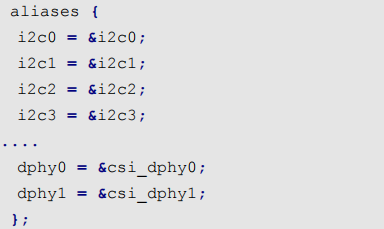
和根节点“/”一样，图中的所有文件分别为 soc 节点的 属性文件和子节点文件夹。大家可以自行查看一下这些属性文件的内容是否和 rk3568.dtsi 中 cpus 节点的属性值相同。

**8.6 特殊节点**

在根节点“/”中有两个特殊的子节点：aliases 和 chosen，我们接下来看一下这两个特殊的子节点。

**8.6.1 aliases 子节点**

打开 rk3568.dtsi 文件，aliases 节点内容如下：

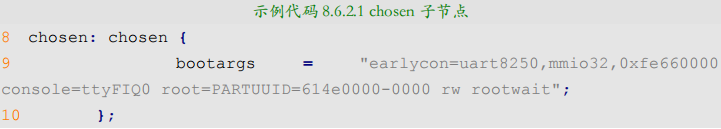


aliases 的意思是“别名”，因此 aliases 节点的主要功能就是定义别名，定义别名的目

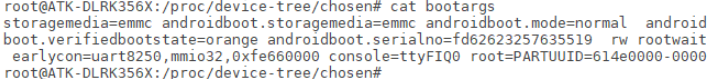
的就是为了方便访问节点。不过我们一般会在节点命名的时候会加上 label，然后通过&label 来访问节点，这样也很方便，而且设备树里面大量使用的是&label 的形式来访问节点。

**8.6.2 chosen 子节点**

chosen 并不是一个真实的设备。chosen 节点主要是为了 uboot 向 Linux 内核传递数据，重点是 bootargs 参数。一般.dts 文件中 chosen 节点通常为空或者内容很少，rk3568-linux.dtsi 中chosen 节点内容如下所示：



从示例代码知：chosen 节点设置了属性“bootargs”的值。但是当我们进入到/proc/device-tree/chosen 目录里面查看 bootargs 的值：



可以看到：bootargs文件的内容 与 示例代码的bootargs内容不一致。前面多了“storagemedia=emmc androidboot.storagemedia=emmc androidboot.mode=normal rootwait”

那么它是怎么来的？

uboot 里面的 bootargs 环境变量会传递给内核，RK3568 里面的 bootargs 值如下：

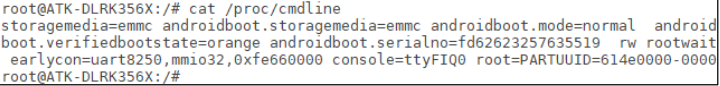
IMG_256

从图可知：uboot 里面的 bootargs 环境变量值为：

storagemedia=emmc androidboot.storagemedia=emmc androidboot.mode=normal

刚好就是bootargs多出来的部分。也就是说最终传递给 Linux 内核的bootargs 是 uboot 下 bootargs 环境变量的值 加 设备树里面 bootargs 属性的值。

这个通过查看 linux 内核的 cmdline(命令行)参数来得到

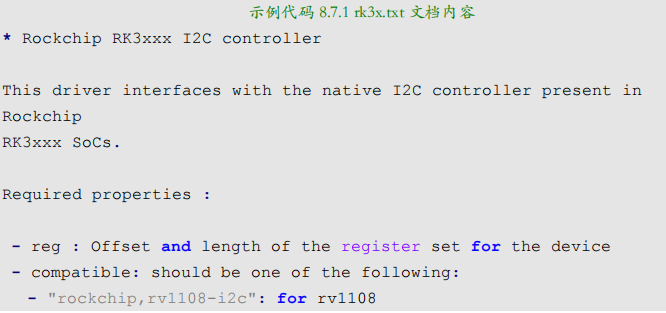


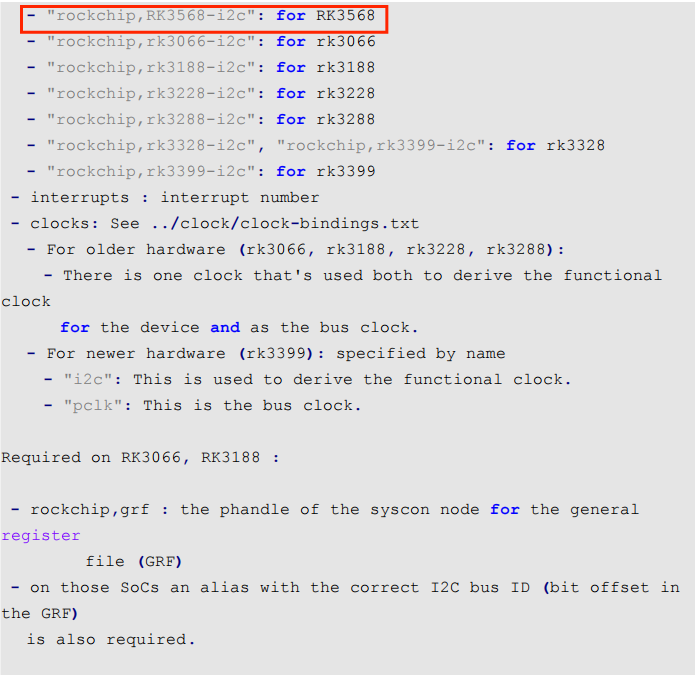
**8.7 绑定信息文档**

设备树是用来描述板子上的设备信息的，不同的设备其信息不同，反映到设备树中就是属性不同。那么我们在设备树中添加一个硬件对应的节点的时候 从哪里查阅相关的说明呢？在Linux 内核源码中有详细的 TXT 文档描述了如何添加节点，这些 TXT 文档叫做绑定文档，路径为：Linux 源码目录/Documentation/devicetree/bindings，

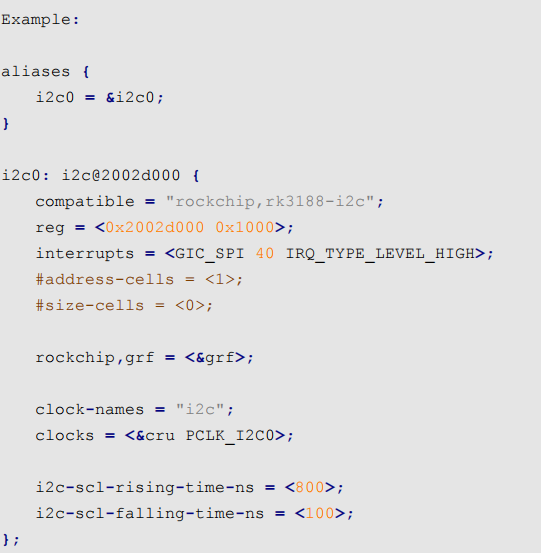
比如我们现在要想在 RK3568 这颗 SOC 的 I2C 下添加一个节点，那么就可以查看

Documentation/devicetree/bindings/i2c/i2c-rk3x.txt，此文档详细的描述了瑞芯微出品的 SOC 如何在设备树中添加 I2C 设备节点。





案例：



**8.8 设备树常用 OF 操作函数**

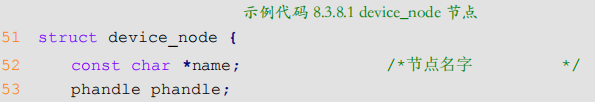
设备树描述了设备的详细信息，这些信息包括数字类型的信息、字符串类型的信息、数组类型的信息，我们在编写驱动的时候需要获取到这些信息。

/\*

比如设备树使用 reg 属性描述了某个外设的寄存器地址为 0X02005482，长度为 0X400，我们在编写驱动的时候就需要获取到 reg 属性的0X02005482 和 0X400 这两个值来初始化外设。

\*/

Linux 内核给我们提供了一系列的函数来获取设备树中的节点或者属性信息，这一系列的函数都有一个统一的前缀“of\_”，所以在很多资料里面也被叫做 OF 函数。这些 OF 函数原型都定义在 include/linux/of.h 文件中。





与查找节点有关的 OF 函数有 5 个：

**1、of\_find\_node\_by\_name 函数**

通过节点名字查找指定的节点。

struct device\_node \*of\_find\_node\_by\_name(struct device\_node \*from, const char \*name);

**from**：开始查找的节点，如果为 NULL 表示 从根节点开始查找整个设备树。

**name**：要查找的节点名字。

**返回值：**找到的节点，如果为 NULL 表示查找失败。

**2、of\_find\_node\_by\_type 函数**

函数通过 device\_type 属性查找指定的节点。

struct device\_node \*of\_find\_node\_by\_type(struct device\_node \*from, const char \*type)

**from**：开始查找的节点，如果为 NULL 表示从根节点开始查找整个设备树。

**type**：要查找的节点对应的 type 字符串，也就是 device\_type 属性值。

**返回值：**找到的节点，如果为 NULL 表示查找失败。

**3、of\_find\_compatible\_node 函数**

函数根据 device\_type 和 compatible 这两个属性查找指定的节点。

struct device\_node \*of\_find\_compatible\_node(struct device\_node \*from, const char \*type, const char \*compat)

**from**：开始查找的节点，如果为 NULL 表示从根节点开始查找整个设备树。

**type**：要查找的节点对应的 type 字符串，也就是 device\_type 属性值，可以为 NULL，表示忽略掉 device\_type 属性。

**compat：**要查找的节点所对应的 compatible 属性列表。

**返回值：**找到的节点，如果为 NULL 表示查找失败。

**4、of\_find\_matching\_node\_and\_match 函数**

函数通过 of\_device\_id 匹配表来查找指定的节点

struct device\_node \*of\_find\_matching\_node\_and\_match(struct device\_node \*from, const struct of\_device\_id \*matches, const struct of\_device\_id \*\*match)

**from**：开始查找的节点，如果为 NULL 表示从根节点开始查找整个设备树。

**matches**：of\_device\_id 匹配表，也就是在此匹配表里面查找节点。

**match：**找到的匹配的 of\_device\_id。

**返回值：**找到的节点，如果为 NULL 表示查找失败

**5、of\_find\_node\_by\_path 函数**

函数通过路径来查找指定的节点。

inline struct device\_node \*of\_find\_node\_by\_path(const char \*path)

**path**：带有全路径的节点名，可以使用节点的别名，比如“/backlight”就是 backlight 这个

节点的全路径。

**返回值：**找到的节点，如果为 NULL 表示查找失败

**8.8.2 查找父/子节点的 OF 函数**

Linux 内核提供了几个查找节点对应的父节点或子节点的 OF 函数:

**1、of\_get\_parent 函数**

用于获取指定节点的父节点(如果有父节点的话)。

struct device\_node \*of\_get\_parent(const struct device\_node \*node)

**node**：要查找的父节点的节点。

**返回值：**找到的父节点。

**2、of\_get\_next\_child 函数**

用迭代的方式查找子节点。

struct device\_node \*of\_get\_next\_child(const struct device\_node \*node, struct device\_node \*prev)

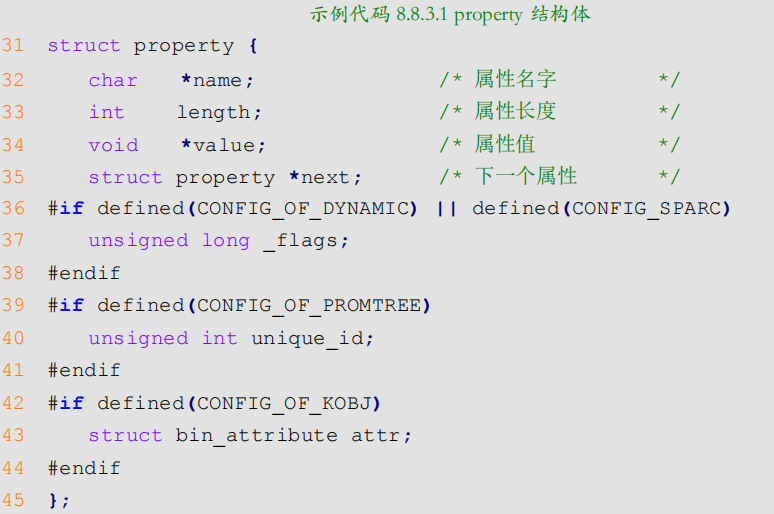
**node**：父节点。

**prev**：前一个子节点，也就是从哪一个子节点开始迭代的查找下一个子节点。可以设置为NULL，表示从第一个子节点开始。

**返回值：**找到的下一个子节点。

**8.8.3 提取属性值的 OF 函数**

节点的属性信息里面保存了驱动所需要的内容，因此对于属性值的提取非常重要，Linux 内核中使用结构体 property 表示属性，此结构体同样定义在文件 include/linux/of.h 中



Linux 内核也提供了提取属性值的 OF 函数。

**1、of\_find\_property 函数**

of\_find\_property 函数用于查找指定的属性

struct property \*of\_find\_property(const struct device\_node \*np,const char \*name,int \*lenp)

**np**：设备节点。

**name**： 属性名字。

**lenp**：属性值的字节数

**返回值：**找到的属性。

**2、of\_property\_count\_elems\_of\_size 函数**

用于获取属性中元素的数量，比如 reg 属性值是一个数组，那么使用此函数可以获取到这个数组的大小。

int of\_property\_count\_elems\_of\_size(const struct device\_node \*np, const char \*propname, int elem\_size)

**np**：设备节点。

**proname**： 需要统计元素数量的属性名字。

**elem\_size**：元素长度。

**返回值：**得到的属性元素数量。

**3、of\_property\_read\_u32\_index 函数**

用于从属性中获取指定标号的 u32 类型数据值(无符号 32 位)

int of\_property\_read\_u32\_index(const struct device\_node \*np, const char \*propname, u32 index, u32 \*out\_value)

**np**：设备节点。

**proname**： 要读取的属性名字。

**index**：要读取的值标号。

**out\_value**：读取到的值

**返回值：**0 读取成功，

负值，读取失败，

-EINVAL 表示属性不存在，

-ENODATA 表示没有要读取的数据，

-EOVERFLOW 表示属性值列表太小。

**4、 of\_property\_read\_u8\_array 函数**

**of\_property\_read\_u16\_array 函数**

**of\_property\_read\_u32\_array 函数**

**of\_property\_read\_u64\_array 函数**

分别是读取属性中 u8、u16、u32 和 u64 类型的数组数据

int of\_property\_read\_u8\_array(const struct device\_node \*np, const char \*propname, u8 \*out\_values, size\_t sz)

int of\_property\_read\_u16\_array(const struct device\_node \*np, const char \*propname, u16 \*out\_values, size\_t sz)

int of\_property\_read\_u32\_array(const struct device\_node \*np, const char \*propname, u32 \*out\_values, size\_t sz)

int of\_property\_read\_u64\_array(const struct device\_node \*np, const char \*propname, u64 \*out\_values, size\_t sz)

**np**：设备节点。

**proname**： 要读取的属性名字。

**out\_value**：读取到的数组值，分别为 u8、u16、u32 和 u64。

**sz：**要读取的数组元素数量

**返回值：**0，读取成功，

负值，读取失败，

-EINVAL 表示属性不存在，

-ENODATA 表示没有要读取的数据，

-EOVERFLOW 表示属性值列表太小。

**5、of\_property\_read\_u8 函数**

**of\_property\_read\_u16 函数**

**of\_property\_read\_u32 函数**

**of\_property\_read\_u64 函数**

有些属性只有一个整形值，这四个函数就是用于读取这种只有一个整形值的属性，分别用于读取 u8、u16、u32 和 u64 类型属性值。

int of\_property\_read\_u8(const struct device\_node \*np, const char \*propname, u8 \*out\_value)

int of\_property\_read\_u16(const struct device\_node \*np, const char \*propname, u16 \*out\_value)

int of\_property\_read\_u32(const struct device\_node \*np, const char \*propname, u32 \*out\_value)

int of\_property\_read\_u64(const struct device\_node \*np, const char \*propname, u64 \*out\_value)

**np**：设备节点。

**proname**： 要读取的属性名字。

**out\_value**：读取到的数组值。

**返回值：**0，读取成功，

负值，读取失败，

-EINVAL 表示属性不存在，

-ENODATA 表示没有要读取的数据，

-EOVERFLOW 表示属性值列表太小。

**6、of\_property\_read\_string 函数**

of\_property\_read\_string 函数用于读取属性中字符串值，函数原型如下：

int of\_property\_read\_string(struct device\_node \*np, const char \*propname, const char \*\*out\_string)

**np**：设备节点。

**proname**： 要读取的属性名字。

**out\_string**：读取到的字符串值。

**返回值：**0，读取成功，负值，读取失败。

**7、of\_n\_addr\_cells 函数**

of\_n\_addr\_cells 函数用于获取#address-cells 属性值，函数原型如下：

int of\_n\_addr\_cells(struct device\_node \*np)

**np**：设备节点。

**返回值：**获取到的#address-cells 属性值。

**8、of\_n\_size\_cells 函数**

用于获取#size-cells 属性值。

int of\_n\_size\_cells(struct device\_node \*np)

**np**：设备节点。

**返回值：**获取到的#size-cells 属性值。

其它of函数见手册。