

# U-Boot v2017(next-dev) 开发指南

---

文件标识：RK-KF-YF-45

发布版本：2.11.0

日期：2021-05-13

文件密级：绝密 秘密 内部资料 公开

## 免责声明

本文档按“现状”提供，瑞芯微电子股份有限公司（“本公司”，下同）不对本文档的任何陈述、信息和内容的准确性、可靠性、完整性、适销性、特定目的性和非侵权性提供任何明示或暗示的声明或保证。本文档仅作为使用指导的参考。

由于产品版本升级或其他原因，本文档将可能在未经任何通知的情况下，不定期进行更新或修改。

## 商标声明

“Rockchip”、“瑞芯微”、“瑞芯”均为本公司的注册商标，归本公司所有。

本文档可能提及的其他所有注册商标或商标，由其各自拥有者所有。

## 版权所有 © 2021 瑞芯微电子股份有限公司

超越合理使用范畴，非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd.

地址：福建省福州市铜盘路软件园A区18号

网址：[www.rock-chips.com](http://www.rock-chips.com)

客户服务电话：+86-4007-700-590

客户服务传真：+86-591-83951833

客户服务邮箱：[fae@rock-chips.com](mailto:fae@rock-chips.com)

---

## 前言

### 概述

本文主要指导读者如何在 U-Boot v2017(next-dev) 版本进行项目开发。

### 各芯片 feature 支持状态

芯片名称	Distro Boot	RKIMG Boot	SPL/TPL	Trust(SPL)	AVB
RV1108	Y	Y	Y	N	N
RK3036	Y	Y	Y	Y	N
RK3126C	Y	Y	Y	Y	Y
RK3128	Y	Y	Y	Y	Y
RK3229	Y	Y	Y	Y	Y
RK3288	Y	Y	Y	Y	Y
RK3308	Y	-	Y	Y	-
RK3326/PX30	Y	Y	Y	Y	Y
RK3328	Y	Y	Y	Y	Y
RK3368/PX5	Y	Y	Y	Y	Y
RK3399	Y	Y	Y	Y	Y
RK1808	Y	-	Y	Y	-
RV1126/RV1109	Y	Y	Y	Y	N
RK3566/RK3568	Y	N	Y	Y	Y

## 读者对象

本文档（本指南）主要适用于以下工程师：

技术支持工程师

软件开发工程师

## 修订记录

版本号	作者	修改日期	修改说明
V1.00	陈健洪	2018-02-28	初始版本
V1.01	朱志展	2018-06-22	fastboot 说明, OPTEE Client 说明
V1.10	陈健洪	2018-07-23	完善文档, 更新和调整大部分章节
V1.11	林鼎强	2018-07-26	完善 Nand、SFC SPI Flash 存储驱动部分
V1.12	陈亮	2018-08-08	增加 HW-ID 使用说明
V1.13	张晴	2018-09-20	增加 CLK 使用说明
V1.20	陈健洪	2018-11-06	增加/更新 defconfig/rktest/probe/interrupt/kernel dtb/uart/atags
V1.21	陈健洪	2019-01-21	增加 dtbo/amp/dvfs 宽温/fdt 命令说明
V1.22	林平	2019-03-05	增加 optee client 说明
V1.23	陈健洪 朱志展	2019-03-25	增加 kernel cmdline 说明
V1.30	陈健洪	2019-03-25	精简和整理文档、纠正排版问题、完善和调整部分章节内容
V1.31	朱志展	2019-04-23	增加硬件 CRYPTO 说明
V1.32	朱志展	2019-05-14	补充 kernel cmdline 说明
V1.33	朱志展	2019-05-29	增加 MMC 命令小节、AVB 与 A/B 系统说明, 术语说明
V1.40	陈健洪	2019-06-20	增加/更新: memblk/sysmem/bi dram/statcktrace/hotkey/ fdt param/run_command/distro/led/reset/ env/wdt/spl/amp/crypto/ efuse/Android compatible/io-domain/bootflow/pack image
V1.41	朱志展	2019-08-21	增加 secure otp 说明

版本号	作者	修改日期	修改说明
V1.42	朱志展	2019-08-27	增加存储设备/MTD 设备说明
V1.43	朱志展	2019-10-08	增加 BCB 说明
V1.44	朱志展	2019-10-15	增加 SPL 驱动与功能支持说明
V1.45	朱志展	2019-11-15	增加 SPL pinctrl 使用说明
V2.0.0	陈健洪	2020-05-02	大版本升级：重构格式、内容、排版等
V2.1.0	陈健洪	2020-05-29	增加FIT方案
V2.1.1	林鼎强	2020-06-07	添加开源框架存储支持说明
V2.1.2	黄涛	2020-07-08	调整格式
V2.2.0	吴达超	2020-07-09	添加以太网网络支持说明
V2.3.0	陈有敏	2020-07-13	添加TPL支持说明
V2.4.0	陈健洪	2020-09-23	更新FIT和编译烧写章节
V2.5.0	朱志展	2020-12-28	更新FIT章节
V2.6.0	朱志展	2020-12-30	添加快速开机章节
V2.7.0	林涛	2021-01-25	添加PCIe支持说明
V2.8.0	陈健洪	2021-03-12	添加U-Boot固件格式、内存容量修改、AMP、RK3568支持说明
V2.9.0	陈健洪	2021-04-13	更新FIT、U盘升级章节
V2.10.0	朱志展	2021-05-06	添加安全操作step-by-step
V2.11.0	陈健洪	2021-05-13	FIT章节增加：recovery.img打包和签名、pss签名参数

## 目录

## U-Boot v2017(next-dev) 开发指南

### Chapter-1 基础简介

- Feature
- Version
- DM
- Security
- Boot-order
- Driver-probe
- Shell
- Boot-Command
- TPL/SPL/U-Boot-proper
- Build-Output
- Environment-Variables
- U-Boot DTS
- Relocation

### Chapter-2 RK架构

- 平台文件
- 平台配置
- 启动流程
- 内存布局
- 存储布局
- Kernel-DTB
- Aliases
- Stacktrace
- ATAGS传参
- U-Boot固件
- kernel固件
- 快捷键
- MMU-Cache
- 内核解压
- bidram/sysmem
- 分区表
- HW-ID DTB
- make.sh
- vendor storage
- AMP
- SD/U盘
- SysReset
- Interrupt
- Timestamp
- Relocation
- 总体耗时
- 详细耗时
- fuse.programmed

### Chapter-3 编译烧写

- 前期准备
- 固件编译
- 固件烧写
- 固件大小
- 特殊打包

### Chapter-4 系统模块

- Android BCB
- AArch32
- DTBO/DTO
- 原理介绍
- DTO 启用
- DTO 结果
- Cmdline

数据来源  
数据含义  
ENV 操作  
    框架支持  
    相关接口  
    高级接口  
    存储位置  
    通用选项  
    fw\_printenv工具  
HW-ID DTB  
    设计原理  
    硬件参考  
    DTB命名  
    DTB打包  
    功能启用  
    加载结果  
AB系统  
    芯片支持  
    配置项  
    分区表  
    注意事项  
AVB安全启动  
    Feature  
    配置  
    参考  
Fastboot  
    配置选项  
    触发方式  
    命令支持  
    命令详解  
SD和U盘  
    机制原理  
    固件制作  
    SD 配置  
    USB 配置  
    功能生效  
    注意事项  
Chapter-5 驱动模块  
    Interrupt  
        框架支持  
        相关接口  
    Clock  
        框架支持  
        相关接口  
        时钟初始化  
        CPU提频  
        时钟树  
    GPIO  
        框架支持  
        相关接口  
    Pinctrl  
        框架支持  
        相关接口  
I2C  
    框架支持  
    相关接口  
Display  
    框架支持

相关接口

DTS 配置

defconfig

LOGO分区

Pmic/Regulator

框架支持

相关接口

init 电压

跳过初始化

Charge

框架支持

打包图片

DTS 配置

系统休眠

更换图片

充电灯

Storage

框架支持

相关接口

DTS 配置

Uart

单独更换

全局更换

关闭打印

相关接口

Key

框架支持

相关接口

Vendor Storage

原理概述

框架支持

相关接口

功能自测

OPTEE Client

框架支持

固件说明

接口文档

共享内存

测试命令

常见错误打印

DVFS

宽温策略

框架支持

相关接口

启用宽温

宽温结果

AMP

框架支持

功能启用

IO-Domain

框架支持

相关接口

Watchdog

框架支持

相关接口

Crypto

框架支持

相关接口

DTS 配置  
Reset  
    框架支持  
    相关接口  
    DTS 配置  
Led  
    框架支持  
    相关接口  
    DTS 节点  
Efuse/Otp  
    框架支持  
    相关接口  
    DTS 配置  
    调用示例  
    Secure-Otp  
MTD  
    框架支持  
    相关接口  
以太网  
    框架支持  
    相关接口  
    DTS 配置  
    使用示例  
    网络故障排查  
PCIe  
    框架支持  
    DTS 配置  
    使用示例  
Chapter-6 进阶原理  
kernel-DTB  
    设计背景  
    Live device tree  
    机制实现  
    U-Boot  
内核传参  
    cmdline  
    内存容量  
    其它方式  
AB系统  
    AB 数据格式  
    AB 启动模式  
        successful-boot  
        reset-retry  
        模式对比  
    启动流程  
    升级和异常  
    验证方法  
        successful-boot  
        reset-retry  
    引用参考  
AVB安全启动  
    引用参考  
    术语  
    简介  
    加密示例  
    AVB  
        AVB 特性  
        key+签名+证书

- AVB lock
- AVB unlock
- kernel 配置
- Android SDK
- Cmdline 新内容
- 分区参考
- fastboot 命令
  - 命令速览
  - 命令使用
- 固件烧写
- Pre-loader verified
- U-boot verified
- 系统校验启动
- Linux AVB
  - 操作流程
  - 验证流程
- SD启动和升级
  - 简介
  - 分类
    - 常规卡
    - 升级卡
    - 启动卡
    - 修复卡
  - 固件标志
  - 启动流程
    - pre-loader 启动
    - U-Boot 启动
    - Recovery和PCBA
- 注意事项
- Chapter-7 配置裁剪
- Chapter-8 调试手段
  - DEBUG
  - Initcall
  - io命令
  - iomem命令
  - i2c命令
  - gpio命令
  - fdt命令
  - mmc命令
  - 时间戳
  - dm tree
  - dm uclass
  - stacktrace.sh
  - 系统卡死
  - CRC 校验
  - HASH校验
  - 修改DDR容量
  - 跳转信息
  - 启动信息
    - RK固件
    - Didstro固件
    - 无有效固件
- Chapter-9 测试用例
- Chapter-10 SPL
  - 固件引导
    - FIT 固件
    - RKFW 固件
    - 存储优先级

编译打包

代码编译

固件打包

系统模块

GPT

A/B system

启动优先级

ATAGS

kernel boot

pinctrl

secure boot

驱动模块

MMC

MTD block

OTP

Crypto

Uart

Chapter-11 TPL

编译打包

配置

编译

打包

Chapter-12 FIT

前言

简介

基础介绍

范例介绍

itb结构

平台配置

支持列表

代码配置

镜像文件

its 文件

相关工具

非安全启动

uboot.img

boot.img

安全启动

原理

校验流程

key存放

key使用

签名存放

防回滚

前期准备

Key

配置

固件

编译打包

启动信息

远程签名

实现思路

被签名数据

具体步骤

其它方案

固件解包

固件替换

安全校验Step-by-Step

## Chapter-13 快速开机

- 芯片支持
- 存储支持
- bootrom 支持
- U-Boot SPL 支持
- mcu配置
- kernel 支持
- 快速开机流程

## Chapter-14 注意事项

- SDK 兼容
  - androidboot.mode 兼容
  - misc 兼容

## Chapter-15 工具

- trust\_merger
- boot\_merger
- loaderimage
- resource\_tool
- mkimage
- stacktrace.sh
- mkbootimg
- unpack\_bootimg
- repack-bootimg
- pack\_resource.sh
- buildman
- patman

## Chapter-16 FAQ

## Chapter-17 附录

- 下载地址
  - rkbin
  - GCC
- 术语

---

# Chapter-1 基础简介

---

## Feature

v2017(next-dev) 是 RK 从 U-Boot 官方的 v2017.09 正式版本中切出来进行开发的版本，目前已经支持 RK 所有主流在售芯片。支持的功能主要有：

- 支持 RK Android 固件启动；
- 支持 Android AOSP 固件启动；
- 支持 Linux Distro 固件启动；
- 支持 Rockchip miniloader 和 SPL/TPL 两种 Pre-loader 引导；
- 支持 LVDS、EDP、MIPI、HDMI、CVBS、RGB 等显示设备；
- 支持 eMMC、Nand Flash、SPI Nand flash、SPI NOR flash、SD 卡、U 盘等存储设备启动；
- 支持 FAT、EXT2、EXT4 文件系统；
- 支持 GPT、RK parameter 分区表；
- 支持开机 LOGO、充电动画、低电管理、电源管理；
- 支持 I2C、PMIC、CHARGE、FUEL GUAGE、USB、GPIO、PWM、GMAC、eMMC、NAND、Interrupt 等；
- 支持 Vendor storage 保存用户的数据和配置；
- 支持 RockUSB 和 Google Fastboot 两种 USB gadget 烧写 eMMC；
- 支持 Mass storage、ethernet、HID 等 USB 设备；

- 支持通过硬件状态动态选择 kernel DTB；

## Version

RK 的 U-Boot 一共有两个版本：v2014旧版本和v2017新版本，内部名称分别为rkdevelop和next-dev。用户有两个方式确认当前U-Boot是否为v2017版本。

方式1：确认根目录Makefile的版本号是否为2017。

```
#  
## Chapter-1 SPDX-License-Identifier:      GPL-2.0+  
  
#  
  
VERSION = 2017  
PATCHLEVEL = 09  
SUBLEVEL =  
EXTRAVERSION =  
NAME =  
.....
```

方式2：确认开机第一行正式打印是否为 U-Boot 2017.09。

```
U-Boot 2017.09-01818-g11818ff-dirty (Nov 14 2019 - 11:11:47 +0800)
```

```
.....
```

项目开源：v2017已开源且定期更新到Github：<https://github.com/rockchip-linux/u-boot>

内核版本：v2017要求RK内核版本 >= 4.4

## DM

DM (Driver Model) 是 U-Boot 标准的 device-driver 开发模型，跟 kernel 的 device-driver 模型非常类似。v2017版本也遵循 DM 框架开发各功能模块。建议读者先阅读DM文档，了解DM架构原理和实现。

README：

```
./doc/driver-model/README.txt
```

### Terminology

---

**Uclass** - a group of devices which operate in the same way. A uclass provides a way of accessing individual devices within the group, but always using the same interface. For example a GPIO uclass provides operations for get/set value. An I2C uclass may have 10 I2C ports, 4 with one driver, and 6 with another.

**Driver** - some code which talks to a peripheral and presents a higher-level interface to it.

**Device** - an instance of a driver, tied to a particular port or peripheral.

简要概括：

- uclass：设备驱动模型
- driver：驱动

- device: 设备

核心代码:

```
./drivers/core/
```

## Security

U-Boot在ARM TrustZone的安全体系中属于Non-Secure World，无法直接访问任何安全的资源（如：安全 memory、安全 otp、efuse...），需要借助 trust 间接访问。RK平台上U-Boot的CPU运行模式：

```
32位平台: Non-Secure PL1
64位平台: EL2(Always be Non-Secure)
```

## Boot-order

RK平台根据前级Loader代码是否开源，目前有两套启动方式：

```
// 前级loader闭源
BOOTROM => ddr bin => Miniloader => TRUST => U-BOOT => KERNEL
// 前级loader开源
BOOTROM => TPL => SPL => TRUST => U-BOOT => KERNEL
```

TPL相当于ddr bin，SPL相当于miniloader。TPL+SPL的组合实现了跟RK闭源ddr.bin+miniloader一致的功能，可相互替换。

## Driver-probe

U-Boot虽然引入了device-driver开发模型，但初始化阶段不会像kernel那样自动发起已注册device-driver的probe。driver的probe必须由用户主动调用发起。接口如下：

```
int uclass_get_device(enum uclass_id id, int index, struct udevice **devp);
int uclass_get_device_by_name(enum uclass_id id, const char *name,
                             struct udevice **devp);
int uclass_get_device_by_seq(enum uclass_id id, int seq, struct udevice **devp);
int uclass_get_device_by_of_offset(enum uclass_id id, int node, struct udevice
**devp);
int uclass_get_device_by_ofnode(enum uclass_id id, ofnode node, struct udevice
**devp);
int uclass_get_device_by_phandle_id(enum uclass_id id,
                                    int phandle_id, struct udevice **devp);
int uclass_get_device_by_phandle(enum uclass_id id,
                                 struct udevice *parent, struct udevice **devp);
int uclass_get_device_by_driver(enum uclass_id id,
                               const struct driver *drv, struct udevice
**devp);
int uclass_get_device_tail(struct udevice *dev, int ret, struct udevice **devp);
.....
```

上述接口的核心调用：

```
int device_probe(struct udevice *dev); // 建议用户一定要了解内部实现!
```

# Shell

U-Boot的Shell叫CLI(cmdline line interface), 即命令行模式, 用户可以根据自己需求自定义CMD。CMD除了通过Shell调用, 还能通过 `run_command()` 和 `run_command_list()` 以代码的形式调用。

```
int run_command(const char *cmd, int flag)
int run_command_list(const char *cmd, int len, int flag)
```

## Boot-Command

U-Boot 最终通过 `CONFIG_BOOTCOMMAND` 定义的启动命令引导kernel。在执行 `CONFIG_BOOTCMD` 之前还会执行 `CONFIG_PREBOOT` 预启动命令, 通常这个命令定义为空。

## TPL/SPL/U-Boot-proper

U-Boot 通过使用**不同的编译条件**可以用同一套代码获取三种不同功能的Loader: TPL/SPL/U-Boot-proper。

TPL(Tiny Program Loader)和SPL(Secondary Program Loader)是比 U-Boot 更早阶段的 Loader:

- TPL: 运行在 sram 中, 负责完成 ddr 初始化;
- SPL: 运行在 ddr 中, 负责完成系统的 lowlevel 初始化、后级固件加载 (`trust.img` 和 `uboot.img`) ;
- U-Boot proper: 运行在ddr中, 即我们通常所说的"U-Boot", 它负责引导kernel;

说明: U-Boot proper 这一说法主要是为了和 SPL 区分开。出于习惯, 后续章节提到的 U-Boot proper 我们都简称为 U-Boot。

启动流程:

```
BOOTROM => TPL(ddr bin) => SPL(miniloader) => TRUST => U-BOOT => KERNEL
```

更多参考: doc/README.TPL 和 doc/README.SPL

## Build-Output

U-Boot编译成功后会在根目录下生成一些重要文件 (支持TPL/SPL编译时才有TPL/SPL的生成文件) :

```
// U-Boot阶段
./u-boot.map          // MAP表文件
./u-boot.sym          // SYMBOL表文件
./u-boot              // ELF文件, 类同内核的vmlinuX (重要!)
./u-boot.dtb          // u-boot自己的dtb文件
./u-boot.bin          // 可执行二进制文件, 会被打包成uboot.img用于烧写

// SPL阶段
./spl/u-boot-spl.map  // MAP表文件
./spl/u-boot-spl.sym  // SYMBOL表文件
./spl/u-boot-spl     // ELF文件, 类同内核的vmlinuX (重要!)
./spl/u-boot-spl.dtb // spl自己的dtb文件
./spl/u-boot-spl.bin // 可执行二进制文件, 会被打包成loader用于烧写

// TPL阶段
./tpl/u-boot-tpl.map  // MAP表文件
./tpl/u-boot-tpl.sym  // SYMBOL表文件
```

```
./tpl/u-boot-tpl      // ELF文件，类同内核的vmlinux（重要！）  
./tpl/u-boot-tpl.dtb // tpl自己的dtb文件  
./tpl/u-boot-tpl.bin // 可执行二进制文件，会被打包成loader用于烧写
```

## Environment-Variables

ENV(Environment-Variables) 是U-Boot支持的一种全局数据管理和传递方式，原理是构建一张HASH映射表，把用户的数据以"键值-数据" 作为表项进行管理。

ENV 通常用于定义平台配置参数：固件加载地址、网络配置 (ipaddr、serverip) 、bootcmd、bootargs等，用户可以在命令行下使用 `printenv` 命令打印出来。

- 用户可选择是否把ENV数据保存到本地存储上
- ENV数据仅限于U-Boot使用，无法直接传递给内核、内核也无法直接解析
- 用户层可以通过U-Boot提供的fw\_printenv工具访问ENV数据

RK 平台上 ENV 数据的存储地址和大小定义如下 (单位：字节)：

```
if ARCH_ROCKCHIP  
config ENV_OFFSET  
    hex  
    depends on !ENV_IS_IN_UBI  
    depends on !ENV_IS_NOWHERE  
    default 0x3f8000  
    help  
        offset from the start of the device (or partition)  
  
config ENV_SIZE  
    hex  
    default 0x8000  
    help  
        Size of the environment storage area  
endif
```

## U-Boot DTS

U-Boot有自己的DTS文件，编译时会自动生成相应的DTB文件，被添加在u-boot.bin末尾。文件目录：

```
arch/arm/dts/
```

各平台具体使用哪个DTS文件，通过defconfig中的 `CONFIG_DEFAULT_DEVICE_TREE` 指定。

## Relocation

通常在开机阶段，U-Boot由前一级的bootloader加载到DRAM的低地址。U-Boot完成board\_f.c 的流程后会把自己重定向到内存末尾某个预留的地址上（称为relocation，这个地址根据U-Boot内存布局而定），完成relocation再继续完成board\_r.c 流程。可以通过开机信息识别：

```
U-Boot 2017.09-gabfd1c5e3d-210202-dirty #cjh (Mar 08 2021 - 16:57:31 +0800)
```

```
Model: Rockchip RK3568 Evaluation Board
PreSerial: 2, raw, 0xfe660000
DRAM: 2 GiB
Sysmem: init
// relocate到ddr首地址偏移0x7d304000的地址。如果为0，则没有做relocation。
Relocation Offset: 7d304000, fdt: 7b9f8ed8
Using default environment
.....
```

## Chapter-2 RK架构

本章主要向用户介绍RK平台上一些重要的基础情况、feature等。

### 平台文件

平台目录：

```
./arch/arm/include/asm/arch-rockchip/
./arch/arm/mach-rockchip/
./board/rockchip/
./include/configs/
```

defconfig目录：

```
./configs/
```

核心公共板级文件！

```
./arch/arm/mach-rockchip/board.c
```

### 平台配置

#### 配置文件

各平台的配置选项、参数通常位于如下几个位置：

```
// 各平台公共文件（开发者通常不需要修改）
./arch/arm/mach-rockchip/Kconfig
./include/configs/rockchip-common.h

// 各平台独有，以RK3399为例
./include/configs/rk3399_common.h
./include/configs/evb_rk3399.h
./configs/rk3399_defconfig
```

#### 配置说明：

如下针对 rockchip-common.h、rkxxx\_common.h、evb\_rkxxx.h 定义的重要配置给出说明。

- RKIMG\_DET\_BOOTDEV：存储类型探测命令，以逐个扫描的方式探测当前的存储设备类型；
- RKIMG\_BOOTCOMMAND：kernel 启动命令；

- ENV\_MEM\_LAYOUT\_SETTINGS: 固件加载地址, 包括 ramdisk/fdt/kernel;
- PARTS\_DEFAULT: 默认的 GPT 分区表, 在某些情况下, 当存储中没有发现有效的 GPT 分区表时被使用;
- ROCKCHIP\_DEVICE\_SETTINGS: 外设相关命令, 主要是指定 stdio (一般会包含显示模块启动命令) ;
- BOOTENV: distro 方式启动 linux 时的启动设备探测命令;
- CONFIG\_SYS\_MALLOC\_LEN: malloc 内存池大小;
- CONFIG\_SYS\_TEXT\_BASE: U-Boot 运行的起始地址;
- CONFIG\_BOOTCOMMAND: 启动命令, 一般定义为 RKIMG\_BOOTCOMMAND;
- CONFIG\_PREBOOT: 预启动命令, 在 CONFIG\_BOOTCOMMAND 前被执行;
- CONFIG\_SYS\_MMC\_ENV\_DEV: MMC 作为 ENV 存储介质时的 dev num, 一般是 0;

如下以 RK3399 为例进行说明:

./include/configs/rockchip-common.h:

```
.....
#define RKIMG_DET_BOOTDEV \
    "rkimg_bootdev=" \
    "if mmc dev 1 && rkimgtest mmc 1; then " \
        "setenv devtype mmc; setenv devnum 1; echo Boot from SDcard;" \
    "elif mmc dev 0; then " \
        "setenv devtype mmc; setenv devnum 0;" \
    "elif rknand dev 0; then " \
        "setenv devtype rknand; setenv devnum 0;" \
    "elif rksfc dev 0; then " \
        "setenv devtype rksfc; setenv devnum 0;" \
    "fi; \0"

#define RKIMG_BOOTCOMMAND \
    "boot_android ${devtype} ${devnum};" \
    "bootrkp;" \
    "run distro_bootcmd;" \
.....
// 动态探测当前的存储类型
// 启动android格式固件
// 启动RK格式固件
// 启动linux固件
```

./include/configs/rk3399\_common.h:

```
.....
#ifndef CONFIG_SPL_BUILD
#define ENV_MEM_LAYOUT_SETTINGS \
    "scriptaddr=0x00500000\0" \
    "pxefile_addr_r=0x00600000\0" \
    "fdt_addr_r=0x01f00000\0" \
    "kernel_addr_r=0x02080000\0" \
    "ramdisk_addr_r=0x0a200000\0"

#include <config_distro_bootcmd.h>
#define CONFIG_EXTRA_ENV_SETTINGS \
    ENV_MEM_LAYOUT_SETTINGS \
    "partitions=" PARTS_DEFAULT \
    ROCKCHIP_DEVICE_SETTINGS \
    RKIMG_DET_BOOTDEV \
    BOOTENV \
#endif
.....
// 固件的加载地址
// 默认的GPT分区表
// 启动linux时的启动设备探测命令
```

```
#define CONFIG_PREBOOT // 在CONFIG_BOOTCOMMAND之前被执行的预启动命令  
令  
.....
```

./include/configs/evb\_rk3399.h:

```
.....  
#ifndef CONFIG_SPL_BUILD  
#undef CONFIG_BOOTCOMMAND  
#define CONFIG_BOOTCOMMAND RKIMG_BOOTCOMMAND // 定义启动命令（设置为  
RKIMG_BOOTCOMMAND）  
#endif  
.....  
#define ROCKCHIP_DEVICE_SETTINGS \  
"stdout=serial,vidconsole\0" \  
"stderr=serial,vidconsole\0"  
.....
```

## 启动流程

RK平台的U-Boot 启动流程如下，仅列出一些重要步骤：

```
start.s  
// 汇编环境  
=> IRQ/FIQ/lowlevel/vbar/errata/cp15/gic // ARM架构相关的lowlevel初始化  
=> _main  
    => stack // 准备好C环境需要的栈  
    // 【第一阶段】C环境初始化，发起一系列的函数调用  
    => board_init_f: init_sequence_f[]  
        initf_malloc  
        arch_cpu_init // 【SoC的lowlevel初始化】  
        serial_init // 串口初始化  
        dram_init // 【获取ddr容量信息】  
        reserve_mmu // 从ddr末尾开始往低地址reserve内存  
        reserve_video  
        reserve_uboot  
        reserve_malloc  
        reserve_global_data  
        reserve_fdt  
        reserve_stacks  
        dram_init_banksizes  
        sysmem_init  
        setup_reloc // 确定U-Boot自身要reloc的地址  
// 汇编环境  
=> relocate_code // 汇编实现U-Boot代码的relocation  
// 【第二阶段】C环境初始化，发起一系列的函数调用  
=> board_init_r: init_sequence_r[]  
    initr_caches // 使能MMU和I/Dcache  
    initr_malloc  
    bidram_initr  
    sysmem_initr  
    initr_of_live // 初始化of_live  
    initr_dm // 初始化dm框架  
    board_init // 【平台初始化，最核心部分】  
        board_debug_uart_init // 串口iomux、clk配置  
        init_kernel_dtb // 【切到kernel dtb】！
```

<code>clks_probe</code>	// 初始化系统频率
<code>regulators_enable_boot_on</code>	// 初始化系统电源
<code>io_domain_init</code>	// <b>io-domain</b> 初始化
<code>set_armclk_rate</code>	// <b>__weak</b> , ARM提频(平台有需求才实现)
<code>dvfs_init</code>	// 宽温芯片的调频调压
<code>rk_board_init</code>	// <b>__weak</b> , 由各个具体平台进行实现
<code>console_init_r</code>	
<code>board_late_init</code>	// 【 <b>platform late</b> 初始化】
<code>rockchip_set_ethaddr</code>	// 设置mac地址
<code>rockchip_set_serialno</code>	// 设置 <b>serialno</b>
<code>setup_boot_mode</code>	// 解析"reboot xxx"命令、 // 识别按键和 <b>loader</b> 烧写模式、
 <b>recovery</b>	
<code>charge_display</code>	// U-Boot充电
<code>rockchip_show_logo</code>	// 显示开机 <b>logo</b>
<code>soc_clk_dump</code>	// 打印 <b>clk tree</b>
<code>rk_board_late_init</code>	// <b>__weak</b> , 由各个具体平台进行实现
<code>run_main_loop</code>	// 【进入命令行模式, 或执行启动命令】

## 内存布局

U-Boot 由前级 Loader 加载到 `CONFIG_SYS_TEXT_BASE` 地址，初始化时会探明当前系统的总内存容量，32位平台上认为最大4GB可用（但是不影响内核对容量的识别），64位平台上认为所有内存都可用。然后通过一系列`reserve_xxx()`接口从内存末尾往前预留需要的内存，最后把自己`relocate`到某段`reserve`的空间上。内存整体使用布局如下，以ARM64为例（常规情况）：

Name	Start Addr Offset	Size	Usage	Secure
ATF	0x00000000	1M	ARM Trusted Firmware	Yes
SHM	0x00100000	1M	SHM, Pstore	No
OP-TEE	0x08400000	2M~30M	参考 TEE 开发手册	Yes
FDT	fdt_addr_r	-	kernel dtb	No
KERNEL	kernel_addr_r	-	kernel 镜像	No
RAMDISK	ramdisk_addr_r	-	ramdisk 镜像	No
.....	-	-	-	-
FASTBOOT	-	-	Fastboot buffer	No
.....	-	-	-	-
SP	-	-	stack	No
FDT	-	sizeof(dtb)	U-Boot dtb	No
GD	-	sizeof(gd)	-	No
Board	-	sizeof(bd_t)	-	No
MALLOC	-	CONFIG_SYS_MALLOC_LEN	系统的堆空间	No
U-Boot	-	sizeof(mon)	u-boot 镜像	No
Video FB	-	fb size	32M	No
TLB Table	RAM_TOP-64K	32K	MMU 页表	No

上表中的 `start Addr offset` 一栏表示基于 DDR base 的地址偏移；

Fastboot地址和大小由配置决定：`CONFIG_FASTBOOT_BUF_ADDR`, `CONFIG_FASTBOOT_BUF_SIZE`。

- Video FB/U-Boot/Malloc/Board/Gd/Fdt/Sp 由顶向下根据实际需求大小来分配；
- 64 位平台：ATF 是 ARMv8 必需的，OP-TEE 是可选项；32 位平台：只有 OP-TEE；
- kernel fdt/kernel/ramdisk 是 U-Boot 需要加载的固件地址，由 `ENV_MEM_LAYOUT_SETTINGS` 定义；
- Fastboot 功能需要的 buffer 地址和大小在 defconfig 中定义；
- OP-TEE 占据的空间需要根据实际需求而定，最大为 30M；其中 RK1808/RK3308 上 OP-TEE 放在低地址，不在 0x8400000；

## 存储布局

RK linux方案的存储布局如下，Android方案除了boot/rootfs的定义跟linux平台有差异，其它基本一致，可借鉴参考。

Partition	Start Sector		Number of Sectors		Partition Size		Requirements
MBR	0	00000000	1	00000001	512	0.5KB	
Primary GPT	1	00000001	63	0000003F	32256	31.5KB	
<b>loader1</b>	<b>64</b>	<b>00000040</b>	<b>7104</b>	<b>00001bc0</b>	<b>4096000</b>	<b>2.5MB</b>	<b>preloader (miniloader or U-Boot SPL)</b>
Vendor Storage	7168	00001c00	512	00000200	262144	256KB	SN, MAC and etc.
Reserved Space	7680	00001e00	384	00000180	196608	192KB	Not used
reserved1	8064	00001f80	128	00000080	65536	64KB	legacy DRM key
U-Boot ENV	8128	00001fc0	64	00000040	32768	32KB	
reserved2	8192	00002000	8192	00002000	4194304	4MB	legacy parameter
<b>loader2</b>	<b>16384</b>	<b>00004000</b>	<b>8192</b>	<b>00002000</b>	<b>4194304</b>	<b>4MB</b>	<b>U-Boot or UEFI</b>
<b>trust</b>	<b>24576</b>	<b>00006000</b>	<b>8192</b>	<b>00002000</b>	<b>4194304</b>	<b>4MB</b>	<b>trusted-os like ATF, OP-TEE</b>
<b>boot ( bootable must be set )</b>	<b>32768</b>	<b>00008000</b>	<b>229376</b>	<b>00038000</b>	<b>117440512</b>	<b>112MB</b>	<b>kernel, dtb, extlinux.conf, ramdisk</b>
<b>rootfs</b>	<b>262144</b>	<b>00040000</b>	-	-	-	<b>-MB</b>	<b>Linux system</b>
Secondary GPT	16777183	0FFFFDFD	33	00000021	16896	16.5KB	

图片引用：[http://opensource.rock-chips.com/wiki\\_Partitions](http://opensource.rock-chips.com/wiki_Partitions)

## Kernel-DTB

原生的U-Boot只支持使用U-Boot自己的DTB，RK平台增加了kernel DTB机制的支持，即使用kernel DTB去初始化外设。主要目的是为了兼容外设板级差异，如：power、clock、display等。

二者的作用：

- U-Boot DTB：负责初始化存储、打印串口等设备；
- Kernel DTB：负责初始化存储、打印串口以外的设备；

U-Boot初始化时先用U-Boot DTB完成存储、打印串口初始化，然后从存储上加载Kernel DTB并转而使用这份DTB继续初始化其余外设。Kernel DTB的代码实现在函数：`init_kernel_dtb()`。

开发者一般不需要修改U-Boot DTB（除非更换打印串口），各平台发布的SDK里使用的defconfig都已启用kernel DTB机制。所以通常对于外设的DTS修改，用户应该修改kernel DTB。

### 关于U-Boot DTB：

DTS目录：

```
./arch/arm/dts/
```

启用kernel DTB机制后：编译阶段会把U-Boot DTS里带u-boot,dm-pre-reloc和u-boot,dm-spl属性的节点过滤出来，在此基础上再剔除defconfig中`CONFIG_OF_SPL_REMOVE_PROPS`指定的property，最终生成u-boot.dtb文件并且追加在u-boot.bin的末尾。

用户编译完U-Boot后可以通过fdtdump命令检查DTB内容：

```
fdtdump ./u-boot.dtb | less
```

更多参考：进阶原理章节。

## Aliases

U-Boot中有些特殊的aliases有别于kernel DTS里的定义。

eMMC/SD在U-Boot中统称为mmc设备，使用编号0、1作区分；SD的启动优先级高于eMMC。

```
mmc1: 表示sd  
mmc0: 表示emmc
```

## Stacktrace

原生的U-Boot不支持调用栈回溯机制，RK平台增加了该功能。目前一共有3种方式触发调用栈打印：

- 系统崩溃时自动触发；
- 用户主动调用 `dump_stack()`；
- 使能 `CONFIG_ROCKCHIP_DEBUGGER`；

例如系统abort：

```
"Synchronous Abort" handler, esr 0x96000010

// abort的原因、pc、lr、sp
* Reason:      Exception from a Data abort, from current exception level
* PC           = 000000000028f430
* LR           = 00000000002608d0
* SP           = 00000000f3dceb30

...
// 重点突出PC和LR
call trace:
PC:    [< 0028f430 >]
LR:    [< 002608d0 >]

// 函数调用关系
Stack:
    [< 0028f430 >]
    [< 0028da24 >]
    [< 00211600 >]
    [< 002117b0 >]
    [< 00202910 >]
    [< 00202aa8 >]
    [< 0027698c >]
    [< 002151ec >]
    [< 00201b2c >]

// 指导用户转换上述调用栈信息
Copy info from "call trace..." to a file(eg. dump.txt), and run
command in your U-Boot project: ./scripts/stacktrace.sh dump.txt
```

用户根据上述说明，把调用栈信息复制到任意txt文件（比如dump.txt）后执行如下命令：

```
cjh@Ubuntu:~/u-boot$ ./scripts/stacktrace.sh dump.txt

// 符号表来源
SYMBOL File: ./u-boot.sym

// 重点列出PC和LR对应的代码位置
call trace:
PC:    [< 0028f430 >] strncpy+0xc/0x20      ./lib/string.c:98
LR:    [< 002608d0 >] on_serialno+0x10/0x1c ./drivers/usb/gadget/g_dnl.c:217
```

```
// 转换后得到真实函数名
Stack:
[< 0028f430 >] strncpy+0xc/0x20
[< 0028da24 >] hdelete_r+0xcc/0xf0
[< 00211600 >] _do_env_set.isra.0+0x70/0x1b8
[< 002117b0 >] env_set+0x3c/0x58
[< 00202910 >] rockchip_set_serialno+0x54/0x140
[< 00202aa8 >] board_late_init+0x5c/0xa0
[< 0027698c >] initcall_run_list+0x58/0x94
[< 002151ec >] board_init_r+0x20/0x24
[< 00201b2c >] relocation_return+0x4/0x0
```

### 注意事项：

- 转换命令有三种，具体用哪种请根据调用栈打印之后的指导说明。

```
./scripts/stacktrace.sh ./dump.txt          // 解析来自U-Boot的调用栈信息
./scripts/stacktrace.sh ./dump.txt tpl      // 解析来自tpl的调用栈信息
./scripts/stacktrace.sh ./dump.txt spl       // 解析来自spl的调用栈信息
```

执行该命令时，**当前机器上的固件必须和当前代码环境匹配才有意义！**否则会得到错误的转换。

## ATAGS传参

RK平台的启动流程：

```
BOOTROM => ddr-bin => Miniloader => TRUST => U-BOOT => KERNEL
```

RK平台的各级固件之间可以通过ATAGS机制传递一些配置信息。

- 适用范围：ddr-bin、miniloader、trust、U-Boot，不包含Kernel。
- 传递内容：串口配置、存储类型、ATF 和 OP-TEE 占用的内存、ddr 容量等。

代码实现：

```
./arch/arm/include/asm/arch-rockchip/rk_atags.h
./arch/arm/mach-rockchip/rk_atags.c
```

## U-Boot固件

RK平台的U-Boot和trust有两种固件格式：RK和FIT格式分别由Miniloader和SPL负责引导。目前Rockchip发布的SDK以RV1126为分界点，RV1126开始的平台采用FIT格式，之前的平台采用RK格式。

- RK 格式

Rockchip自定义的固件格式，U-Boot和trust分别打包为uboot.img和trust.img。如下：

uboot.img 和32位 trust.img 镜像文件的magic为“LOADER”

```
00000000  4c 4f 41 44 45 52 20 20  00 00 00 00 00 00 00 00 | LOADER
.....| 00 00 20 00 78 d0 0f 00  06 99 c2 a8 20 00 00 00 | ... .x.....
...| 00000020  09 8a b0 e1 89 7a c2 89  0d e8 da ef 86 3e f2 24
| .....z.....>.$|
```

64位 trust.img 镜像文件的magic为“BL3X”

```
00000000 42 4c 33 58 00 01 00 00 23 00 00 00 f8 00 04 00  
| BL3X....#..... |  
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
| ..... |
```

- FIT 格式

U-Boot mainline支持的一种灵活性极高的固件格式。 U-Boot、 trust以及mcu等固件一起打包为 uboot.img。

uboot.img 的镜像文件的magic 为“d0 0d fe ed”，用命令 `fdtdump uboot.img` 可以查看固件头。

```
00000000 d0 0d fe ed 00 00 06 00 00 00 00 58 00 00 04 c4  
| .....x.... |  
00000010 00 00 00 28 00 00 00 11 00 00 00 10 00 00 00 00 | ...  
| ..... |
```

| 更多FIT介绍请参考FIT章节。

- 备份打包

通常为了应对OTA升级过程断电等可能导致固件损坏的情况， uboot.img和trust.img都做了多备份打包。

固件	单份大小	打包份数
RK uboot.img	1MB	4
RK 32位trust.img	1MB	4
RK 64位trust.img	2MB	2
FIT uboot.img	2MB	2

| 从上述表格可知， uboot.img 和 trust.img 的大小默认都是4MB。

单份大小和份数的修改方法：

- RK 格式： 编译命令追加参数。例如：`--sz-uboot 2048 1` 和 `--sz-trust 4096 1`， 表示 uboot.img单份2M， 打包1份； trust.img单份4M， 打包1份。
- FIT 格式： 更改配置参数： `CONFIG_SPL_FIT_IMAGE_KB` 和 `CONFIG_SPL_FIT_IMAGE_MULTIPLE`。 分别表示单份大小(单位： KB)和打包份数。

## kernel固件

RK平台的U-Boot支持三种格式的内核固件引导：

- RK格式

镜像文件的 magic 为“KRNL”：

```
00000000 4B 52 4E 4C 42 97 0F 00 1F 8B 08 00 00 00 00 00  
KRNL..y.....  
00000010 00 03 A4 BC 0B 78 53 55 D6 37 BE 4F 4E D2 A4 69  
.....xsu.7.ON..i
```

```
kernel.img = kernel;  
resource.img = dtb + logo.bmp + logo_kernel.bmp;  
boot.img = ramdisk;  
recovery.img = ramdisk(for recovery) ;
```

- Android格式

镜像文件的 magic 为"ANDROID!"：

```
00000000 41 4E 44 52 4F 49 44 21 24 10 74 00 00 80 40 60  
ANDROID!$.t...@`  
00000010 F9 31 CD 00 00 00 00 62 00 00 00 00 00 00 F0 60  
.1.....b.....`
```

```
boot.img = kernel + ramdisk+ resource + <dtb>;  
recovery.img = kernel + ramdisk(for recovery) + resource + <recovery_dtbo> + <dtb>;
```

说明：recovery\_dtbo：从Android-9.0才开始新增的镜像；dtb：从Android-10.0才开始新增的镜像；

- Distro格式

目前开源 Linux 的一种通用固件打包格式：将 ramdisk、dtb、kernel 打包成一个镜像。这个镜像通常以某种文件系统格式存在，例如 ext2、ext4、fat 等，U-Boot需要通过文件系统才能访问到它的内容。更多参考：

```
./doc/README.distro  
./include/config_distro_defaults.h  
./include/config_distro_bootcmd.h
```

- 引导优先级：android > rk > distro，每一类固件都有对应的启动命令，三个命令会按优先级逐个执行，直到把固件引导起来。如果所有命令都失败，则停在U-Boot命令行模式。

启动优先级定义：

```
#define RKIMG_BOOTCOMMAND \  
"boot_android ${devtype} ${devnum};" \  
"bootrk;" \  
"run distro_bootcmd;"
```

## 快捷键

RK平台提供串口组合键触发一些事件用于调试、烧写（如果无法触发，请多尝试几次；启用secure-boot时无效）。**开机时长按**：

- ctrl+c: 进入 U-Boot 命令行模式；
- ctrl+d: 进入 loader 烧写模式；
- ctrl+b: 进入 maskrom 烧写模式；
- ctrl+f: 进入 fastboot 模式；
- ctrl+m: 打印 bidram/system 信息；
- ctrl+i: 使能内核 initcall\_debug；
- ctrl+p: 打印 cmdline 信息；
- ctrl+s: "Starting kernel..."之后进入 U-Boot 命令行；

## MMU-Cache

RK平台默认使能MMU、Dcache、Icache，MMU采用1:1线性映射，Dcache采用write-back策略。相关接口：

```
// Icache接口:  
void icache_enable (void);  
void icache_disable (void);  
void invalidate_icache_all(void);  
  
// Dcache接口:  
void dcache_disable (void);  
void dcache_enable(void);  
void flush_dcache_range(unsigned long start, unsigned long stop);  
void flush_cache(unsigned long start, unsigned long size);  
void flush_dcache_all(void);  
void invalidate_dcache_range(unsigned long start, unsigned long stop);  
void invalidate_dcache_all(void);  
// 重新映射某块内存区间的Dcache属性  
void mmu_set_region_dcache_behaviour(phys_addr_t start, size_t size,  
                                     enum dcache_option option)
```

## 内核解压

- 64位平台的机器通常烧写Image，由U-Boot加载到目标运行地址。但是RK平台的U-Boot还可支持对64位LZ4格式的压缩内核进行解压。但是用户必须使能：

```
CONFIG_LZ4=y
```

64位LZ4压缩内核的解压前、后地址必须定义在各平台的 rkxxx\_common.h 文件中：

```
#define ENV_MEM_LAYOUT_SETTINGS \  
    "scriptaddr=0x00500000\0" \  
    "pxefile_addr_r=0x00600000\0" \  
    "fdt_addr_r=0x01f00000\0" \  
    "kernel_addr_no_b132_r=0x00280000\0" \  
    "kernel_addr_r=0x00680000\0"           // LZ4解压内核的地址  
    "kernel_addr_c=0x02480000\0"           // LZ4压缩内核的地址  
    "ramdisk_addr_r=0x04000000\0"
```

- 32位平台的机器通常烧写zImage，由U-Boot加载到 kernel\_addr\_r 地址上，再由内核完成自解压。但是RK平台的U-Boot还可支持Image格式，由U-Boot加载到目标运行地址。

目前各平台的 rkxxx\_common.h 文件只定义了 kernel\_addr\_r 而没有定义 kernel\_addr\_c 地址。但是用户不需要更改配置，U-Boot会判断当前是zImage还是Image，动态处理这2个地址。但是用户必须关闭：

```
CONFIG_SKIP_RELOCATE_UBOOT
```

32位内核的加载地址定义：

```

#define ENV_MEM_LAYOUT_SETTINGS \
    "scriptaddr=0x60000000\0" \
    "pxefile_addr_r=0x60100000\0" \
    "fdt_addr_r=0x68300000\0" \
    "kernel_addr_r=0x62008000\0" // zImage压缩内核的地址
    "ramdisk_addr_r=0x6a200000\0"

```

## bidram/sysmem

U-Boot可以使用系统的所有内存，且从高地址往低地址预留系统所需的内存，预留完后通常还剩余较大的内存空间。U-Boot没有机制去管理这块空间，因此RK平台引入bidram、sysmem内存块机制对这块内存进行管理。

由此，加上U-Boot已有的malloc管理机制，RK平台就把系统所有内存通过sysmem + bidram + malloc管理起来了，防止出现内存冲突等问题。



- bidram：管理u-boot、kernel阶段不可用、需要剔除的内存块，例如：ATF、OP-TEE 占用的空间；
- sysmem：管理kernel 可见、可用的内存块。例如：fdt、ramdisk、kernel、fastboot 占用的空间。

相关代码：

```

./lib/sysmem.c
./lib/bidram.c
./include/membblk.h
./arch/arm/mach-rockchip/membblk.c

```

如下是 bidram 和 sysmem 的内存管理信息表，当出现内存块初始化或分配异常时会被 dump 出来。如下做出简单介绍。

bidram 内存信息表：

```

bidram_dump_all:
-----  

// <1> 这里显示了U-Boot从前级loader获取的ddr的总容量信息，一共有2GB
memory.rgn[0].addr      = 0x00000000 - 0x80000000 (size: 0x80000000)

memory.total            = 0x80000000 (2048 MiB. 0 KiB)
-----  

// <2> 这里显示了被预留起来的各固件内存信息，这些空间对kernel不可见
reserved.rgn[0].name   = "ATF"
                     .addr = 0x00000000 - 0x00100000 (size: 0x00100000)
reserved.rgn[1].name   = "SHM"
                     .addr = 0x00100000 - 0x00200000 (size: 0x00100000)
reserved.rgn[2].name   = "OP-TEE"
                     .addr = 0x08400000 - 0x0a200000 (size: 0x01e00000)

```

```

reserved.total      = 0x02000000 (32 MiB. 0 KiB)
-----
// <3> 这里是核心算法对上述<2>进行的预留信息整理, 例如: 会对相邻块进行合并
LMB.reserved[0].addr = 0x00000000 - 0x00200000 (size: 0x00200000)
LMB.reserved[1].addr = 0x08400000 - 0x0a200000 (size: 0x01e00000)

reserved.core.total = 0x02000000 (32 MiB. 0 KiB)

```

sysmem 内存信息表:

```

sysmem_dump_all:

// <1> 这里是sysmem可管理的总内存容量, 即bidram<3>之外的可用ddr容量, 对kernel可见。
memory.rgn[0].addr = 0x00200000 - 0x08400000 (size: 0x08200000)
memory.rgn[1].addr = 0x0a200000 - 0x80000000 (size: 0x75e00000)

memory.total       = 0x7e000000 (2016 MiB. 0 KiB)
-----
// <2> 这里显示了各个固件alloc走的内存块信息
allocated.rgn[0].name = "U-Boot"
    .addr = 0x71dd6140 - 0x80000000 (size: 0x0e229ec0)
allocated.rgn[1].name = "STACK"      <Overflow!> // 表明栈溢出
    .addr = 0x71bd6140 - 0x71dd6140 (size: 0x00200000)
allocated.rgn[2].name = "FDT"
    .addr = 0x08300000 - 0x08316204 (size: 0x00016204)
allocated.rgn[3].name = "KERNEL"     <Overflow!> // 表明内存块溢出
    .addr = 0x00280000 - 0x014ce204 (size: 0x0124e204)
allocated.rgn[4].name = "RAMDISK"
    .addr = 0x0a200000 - 0x0a3e6804 (size: 0x001e6804)
// <3> malloc_r/f的大小
malloc_r: 192 MiB, malloc_f: 16 KiB

allocated.total     = 0xf874acc (248 MiB. 466 KiB)
-----
// <4> 这里是核心算法对上述<2>进行的信息整理, 显示被占用走的内存块信息
LMB.reserved[0].addr = 0x00280000 - 0x014ce204 (size: 0x0124e204)
LMB.reserved[1].addr = 0x08300000 - 0x08316204 (size: 0x00016204)
LMB.reserved[2].addr = 0x0a200000 - 0x0a3e6804 (size: 0x001e6804)
LMB.reserved[3].addr = 0x71bd6140 - 0x80000000 (size: 0x0e429ec0)

reserved.core.total = 0xf874acc (248 MiB. 466 KiB)

```

如下是一些常见的错误打印, 当出现这些异常时, 请结合上述 bidram 和 sysmem dump 内存信息进行分析。

```

// 期望申请的内存已经被其他固件占用了，存在内存重叠。这说明当前系统的内存块使用规划不合理
Sysmem Error: "KERNEL" (0x00200000 - 0x02200000) alloc is overlap with existence
"RAMDISK" (0x00100000 - 0x01200000)

// 期望申请的内存因为一些特殊原因无法申请到（分析sysmem和bidram信息）
Sysmem Error: Failed to alloc "KERNEL" expect at 0x00200000 - 0x02200000 but at
0x00400000 - 0x0420000

// sysmem管理的空间起始地址为0x200000，所以根本申请不到0x100000起始的空间
Sysmem Error: Failed to alloc "KERNEL" at 0x00100000 - 0x02200000

// 重复申请"RAMDISK"内存块
Sysmem Error: Failed to double alloc for existence "RAMDISK"

```

## 分区表

RK平台的U-Boot支持两种分区表：RK parameter格式（旧）和标准GPT格式（新），当机器上同时存在两种分区表时，优先使用GPT分区表。无论是GPT还是RK parameter，烧写用的分区表文件都叫parameter.txt。用户可以通过“TYPE: GPT”属性确认是否为GPT。

```

FIRMWARE_VER:8.1
MACHINE_MODEL:RK3399
MACHINE_ID:007
MANUFACTURER: RK3399
MAGIC: 0x5041524B
ATAG: 0x00200800
MACHINE: 3399
CHECK_MASK: 0x80
PWR_HLD: 0,0,A,0,1
TYPE: GPT // 当前是GPT格式的分区表，否则为RK parameter
CMDLINE:mtdparts=rk29xxnand:0x00002000@0x00004000(uboot),0x00002000@0x00006000(trust),0
x00002000@0x00008000(misc),0x00008000@0x000a000(resource),0x00010000@0x0012000(kerne
l),0x00010000@0x0022000(boot),0x00020000@0x0032000(recovery),0x00038000@0x0005200(bac
kup),0x00002000@0x0008a000(security),0x00100000@0x0008c000(cache),0x00500000@0x0018c000
(system),0x00008000@0x0068c000(metadata),0x00100000@0x00694000(vendor),0x00100000@0x000796000(oem),0x00000400@0x00896000(frp),-@0x00896400(userdata:grow)

```

## HW-ID DTB

RK平台的U-Boot可以根据GPIO或者ADC的硬件状态，从多份DTB文件中筛选与硬件状态匹配的DTB进行加载。

更多参考：系统模块章节。

## make.sh

make.sh既是一个编译脚本，也是一个打包、调试工具。可用于反汇编、打包固件。

```

// 帮助命令
./make.sh --help

```

```

// 打包固件的功能
./make.sh trust          // 打包trust
./make.sh loader          // 打包loader
./make.sh trust <ini-file> // 打包trust时指定ini文件
./make.sh loader <ini-file> // 打包loader时指定ini文件
./make.sh spl              // 用tpl+spl替换ddr和miniloader, 打包成loader
./make.sh spl-s            // 用spl替换miniloader, 打包成loader
./make.sh itb              // 打包u-boot.itb (64位平台只支持打包ATF和U-Boot, OP-TEE不打包)
./make.sh env              // 生成fw_printenv工具

// 反汇编的功能
./make.sh elf-[x] [type]   // 反汇编: 使用-[x]参数, [type]可选择是否反汇编SPL或TPL
./make.sh elf              // 反汇编u-boot文件, 默认使用-D参数
./make.sh elf-s            // 反汇编u-boot文件, 使用-S参数
./make.sh elf-d            // 反汇编u-boot文件, 使用-D参数
./make.sh elf spl          // 反汇编tpl/u-boot-tpl文件, 默认使用-D参数
./make.sh elf tpl          // 反汇编spl/u-boot-tpl文件, 默认使用-D参数
./make.sh <addr>           // 需要addr对应的函数名和代码位置
./make.sh map              // 打开u-boot.map
./make.sh sym              // 打开u-boot.sym

```

## vendor storage

RK平台的U-Boot提供了Vendor storage区域给用户保存SN、MAC等信息。存储偏移如下（详见 vendor.c）：

```

#define EMMC_VENDOR_PART_OFFSET      (1024 * 7)
/* --- Spi Nand/SLC/MLC large capacity case define --- */
#define NAND_VENDOR_PART_OFFSET     0
/* --- Spi/Spi Nand/SLC/MLC small capacity case define --- */
#define FLASH_VENDOR_PART_OFFSET    8
.....

```

用户一般不需要关注和修改存储偏移，只需要关注读写接口：

```

int vendor_storage_read(u16 id, void *pbuff, u16 size)
int vendor_storage_write(u16 id, void *pbuff, u16 size)

```

## AMP

RK平台的U-Boot支持AMP(Asymmetric Multi-Processing) 固件引导。

更多参考：驱动模块章节。

## SD/U盘

RK平台的U-Boot支持SD/U盘的固件启动或升级。其中：

- SD启动/升级是从bootrom这一级开始支持
- U盘启动/升级是从U-Boot这一级开始支持

更多参考：系统模块章节。

## SysReset

- U-Boot的复位和kernel一样，最终需要陷入trust里完成
- U-Boot 命令行模式可以支持跟kernel一样的reboot xxx命令（依赖于kernel dts中的定义）

## Interrupt

U-Boot的原生代码没有完整支持中断，RK平台完善了该功能，支持GIC-V2、GIC-V3。

更多参考：驱动模块章节。

## Timestamp

Kernel的打印信息默认带有时间戳，方便用户关注时间。U-Boot的打印信息默认没有带时间戳，用户有需要的话可以使能配置 CONFIG\_BOOTSTAGE\_PRINTF\_TIMESTAMP。如下：

```
[    0.324987] U-Boot 2017.09-00019-g9b55ed0-dirty (Dec 26 2019 - 14:31:44
+0800)

[    0.327215] Model: Evb-RK3288
[    0.330039] PreSerial: 2
[    0.332526] DRAM: 2 GiB
[    0.336454] Relocation offset: 00000000, fdt: 7be22c38
[    0.346981] Using default environment

[    0.351075] dwmmc@ff0c0000: 1, dwmmc@ff0f0000: 0
[    0.394136] Bootdev(atags): mmc 0
[    0.394272] MMC0: High Speed, 52Mhz
[    0.395276] PartType: EFI
[    0.400347] Android 9.0, Build 2019.6
[    0.402070] boot mode: None
[    0.405213] Found DTB in boot part
[    0.407833] DTB: rk-kernel.dtb
[    0.418211] ANDROID: fdt overlay OK
[    0.432128] I2C0 speed: 400000Hz
[    0.435916] PMIC: RK808
[    0.439113] vdd_arm 1100000 uv
[    0.444148] vdd_gpu 1100000 uv
.....
[    1.005018] ## Booting Android Image at 0x02007800 ...
[    1.009917] Kernel load addr 0x02008000 size 8062 KiB
[    1.014981] ## Flattened Device Tree blob at 08300000
[    1.019970]   Booting using the fdt blob at 0x8300000
[    1.025185]   XIP Kernel Image ... OK
[    1.035469]   'reserved-memory' dma-unusable@fe000000: addr=fe000000
size=1000000
[    1.037448]   'reserved-memory' ramaoops@00000000: addr=80000000 size=f0000
[    1.044412]   Using Device Tree in place at 08300000, end 08316ed1
[    1.064363] Adding bank: 0x00000000 - 0x08400000 (size: 0x08400000)
[    1.064976] Adding bank: 0x09200000 - 0x80000000 (size: 0x76e00000)
[    1.075259] Total: 812.613 ms

[    1.075279] Starting kernel ...
.....
```

注意：时间戳打印的是相对时间，而非绝对时间。

## Relocation

U-Boot会在完成board\_f.c的流程后把自己relocate到内存末尾的某个地址上，具体地址视U-Boot内存布局而定。RK的U-Boot默认：

- 32位平台：`CONFIG_SKIP_RELOCATE_UNBOOT=y`时不做relocation，否则有做。
- 64位平台有做relocation。

## 总体耗时

U-Boot 初始化结束默认会打印本阶段的总耗时：

```
### Booting Android Image at 0x02007800 ...
Kernel load addr 0x02008000 size 8062 KiB
### Flattened Device Tree blob at 08300000
  Booting using the fdt blob at 0x8300000
  XIP Kernel Image ... OK
'reserved-memory' dma-unusable@fe000000: addr=fe000000 size=1000000
'reserved-memory' ramoops@00000000: addr=8000000 size=f0000
  Using Device Tree in place at 08300000, end 08316ed1
Adding bank: 0x00000000 - 0x08400000 (size: 0x08400000)
Adding bank: 0x09200000 - 0x80000000 (size: 0x76e00000)
Total: 812.613 ms // U-Boot阶段的总耗时

Starting kernel ...
```

## 详细耗时

用户可以打开lib/initcall.c的`debug()`和`DEBUG`获得如下的流程耗时，函数地址可借助./make.sh进行反汇编获得。

```
U-Boot 2017.09-00019-g9b55ed0-dirty (Dec 26 2019 - 14:45:33 +0800)

          #      5212 us # 137.868 ms
initcall: 0020de1f
          #      1 us # 142.636 ms
initcall: 0020e015
Model: Evb-RK3288
          #      1646 us # 149. 48 ms
initcall: 0020dd61
PreSerial: 2
          #      1213 us # 155. 28 ms
initcall: 0020ddcd
DRAM:
          #      606 us # 160.401 ms
initcall: 00203719
          // 如下 187 us 是 initcall: 00203719 调用的耗时
          // 如下 165.355 ms 是 initcall: 00203719 为止的U-Boot启动耗时
          #      187 us # 165.355 ms
initcall: 0020de81
          #      2 us # 169.938 ms
initcall: 0020dc29
          #      1 us # 174.703 ms
initcall: 0020dc3d
          #      1 us # 179.469 ms
initcall: 0020ddad
          #      2 us # 184.237 ms
initcall: 0020de27
          #      1 us # 189.  2 ms
```

.....

## fuse.programmed

RK平台为了方便调试secure-boot功能，只需要对固件签名就能开启secure-boot模式（可不烧写efuse/otp）。Miniloader会通过U-Boot向kernel追加cmdline表明当前的efuse/otp使能是否被烧写：

- "fuse.programmed=1"：开启了secure-boot，efuse/otp已经被烧写。
- "fuse.programmed=0"：开启了secure-boot，efuse/otp没有被烧写。
- cmdline中没有fuse.programmed：没有开启secure-boot（Miniloader不传递），或者Miniloader太旧没有支持传递。

U-Boot需要包含如下提交：

```
83c9bd4 board: rockchip: pass fuse programmed state to kernel
```

## Chapter-3 编译烧写

### 前期准备

- 下载rkbin

这是一个工具包仓库，用于存放RK不开源的bin、脚本、打包工具。U-Boot编译时会从该仓库索引相关文件，打包生成loader、trust、uboot固件。rkbin和U-Boot工程必须保持同级目录关系。

仓库下载：请参考附录章节。

- 下载GCC

GCC编译器使用gcc-linaro-6.3.1，放置于prebuilts目录之内。prebuilts和U-Boot保持同级目录关系。如下：

```
// 32位:  
prebuilts/gcc/linux-x86/arm/gcc-linaro-6.3.1-2017.05-x86_64_arm-linux-  
gnueabihf  
// 64位:  
prebuilts/gcc/linux-x86/aarch64/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-  
linux-gnu/
```

GCC下载：请参考附录章节

- 选择defconfig

芯片	defconfig	支持kernel dtb	说明
rv1108	evb-rv1108_defconfig	N	\
rk1808	rk1808_defconfig	Y	\
rk1806	rk1806_defconfig	Y	\
rk3036	rk3036_defconfig	Y	\
rk3128x	rk3128x_defconfig	Y	\
rk3128	evb-rk3128_defconfig	N	\
rk3126	rk3126_defconfig	Y	\
rk322x	rk322x_defconfig	Y	\
rk3288	rk3288_defconfig	Y	\
rk3368	rk3368_defconfig	Y	\
rk3328	rk3328_defconfig	Y	\
rk3399	rk3399_defconfig	Y	\
rk3399pro	rk3399pro_defconfig	Y	\
rk3399pro-npu	rknpu-lion_defconfig	Y	\
rk3308	rk3308_defconfig	Y	\
rk3308-aarch32	rk3308-aarch32_defconfig	Y	\
px30	px30_defconfig	Y	\
rk3326	rk3326_defconfig	Y	\
rk3326-aarch32	rk3326-aarch32_defconfig	Y	\
rv1126	rv1126_defconfig	Y	通用版本
rv1126	rv1126-ab.config	Y	通用版本+支持A/B
rv1126	rv1126-emmc-tb.config rv1126-lp3-emmc-tb.config rv1126-spi-nor-tb.config	Y	eMMC+DDR3 快速开机 eMMC+LP3 快速开机 Spi Nor+DDR3 快速开机
rv1126	rv1126-spi-nor-tiny_defconfig rv1126-ramboot.config rv1126-usbplug.config	Y	Spi Nor 小容量 无存储器件(内存启动) usbplug功能

芯片	defconfig	支持kernel dtb	说明
rk3566	rk3566.config rk3566-eink.config	Y	通用版本 电子书版本
rk3568	rk3568_defconfig rk3568-dfu.config rk3568-nand.config rk3568-spl-spi- nand_defconfig rk3568-aarch32.config rk3568-usbplug.config	Y	通用版本 支持dfu 支持MLC/TLC/ eMMC SPI-nand专用SPL 支持aarch32模式 支持usbplug模式

注意：如果表格和SDK发布的defconfig不同，请以SDK为准。

- config fragment介绍

由于单个平台上产品的差异化需求，一个defconfig已经无法满足。所以从RV1126开始支持config fragment，即对defconfig进行overlay。

例如：rv1126-emmc-tb.config里指定了`CONFIG_BASE_DEFCONFIG="rv1126_defconfig"`，当执行`./make.sh rv1126-emmc-tb`命令时会先用rv1126\_defconfig生成.config，然后再使用rv1126-emmc-tb.config里的配置对.config进行overlay。该命令可等效为：

```
make rv1126_defconfig rv1126-emmc-tb.config && make
```

如果要对config fragment文件进行更新，只需要借助`./scripts/sync-fragment.sh`。例如：

```
./scripts/sync-fragment.sh configs/rv1126-emmc-tb.config
```

命令效果：把当前.config和rv1126\_defconfig的配置差异项diff到rv1126-emmc-tb.config文件中。

## 固件编译

编译命令：

```
./make.sh [board] // [board]: configs/[board]_defconfig文件。
```

**首次编译：**无论32位或64位平台，第一次或想重新指定defconfig，则编译命令必须指定[board]。例如：

```
./make.sh rk3399 // build for rk3399_defconfig
./make.sh evb-rk3399 // build for evb-rk3399_defconfig
./make.sh firefly-rk3288 // build for firefly-rk3288_defconfig
```

**二次编译：**无论32位或64位平台，如果要基于当前".config"二次编译，则编译命令不用指定[board]：

```
./make.sh
```

注意：如果编译时出现奇怪的问题导致编译失败，请尝试`make distclean`后重新编译。

**固件生成**: 编译完成后会在U-Boot根目录下打包生成: trust、uboot、loader。如下是打包生成时的信息:

```
// 编译...
....
// uboot打包过程
load addr is 0x60000000!
pack input u-boot.bin
pack file size: 478737
crc = 0x840f163c
uboot version: v2017.12 Dec 11 2017
pack uboot.img success!
pack uboot okay! Input: u-boot.bin

// loader打包过程及引用的ini文件
out:rk3126_loader_v2.09.247.bin
fix opt:rk3126_loader_v2.09.247.bin
merge success(rk3126_loader_v2.09.247.bin)
pack loader okay! Input: /home/cjh/rkbin/RKBOOT/RK3126MINIALL.ini

// trust打包过程及引用的ini文件
Load addr is 0x68400000!
pack file size: 602104
crc = 0x9c178803
trustos version: Trust os
pack ./trust.img success!
trust.img with ta is ready
pack trust okay! Input: /home/cjh/rkbin/RKTRUST/RK3126TOS.ini

// 提示编译成功。注意: 即使上述的trust和loader打包失败也会提示这句话, 说明至少生成了
uboot.img
Platform RK3126 is build OK, with new .config(make rk3126_defconfig)
```

最终在根目录下生成可烧写的固件:

```
./uboot.img
./trust.img // 注意: 如果是fit格式的固件, 则没有trust.img。trust的二进制被打包在
uboot.img里。
./rk3126_loader_v2.09.247.bin
```

固件打包工具: 请参考工具章节。

## 固件烧写

**烧写工具:**

Windows/Linux的固件烧写工具建议使用SDK发布的工具版本或最新版本。

**烧写模式:**

RK平台一共有两种烧写模式: Maskrom模式、Loader模式(U-Boot)。

(1) 进入Loader烧写模式的方法:

- 开机时, 机器长按音量+
- 开机时, 上位机长按ctrl+d组合键

- U-Boot命令行输入: download 或者 rockusb 0 \$devtype \$devnum

(2) 进入Maskrom烧写模式的方法:

- 开机时, 上位机长按ctrl+b组合键
- U-Boot命令行输入: rbrom

#### 注意事项:

- 目前 U-Boot 支持两种分区表: RK parameter 分区表 (旧) 和 GPT 分区表 (新)。如果想从当前的分区表替换成另外一种分区表类型, 则 Nand 机器必须整套固件重新烧写; eMMC 机器可以支持单独替换分区表。
- 如果机器上同时存在两种分区表, 则优先识别GPT分区表。可通过开机信息确认:

```
...
PartType: EFI // 当前为GPT分区表, 否则打印"PartType: RKPARM"
...
```

## 固件大小

请参考章节: RK架构 => U-Boot固件。

## 特殊打包

./make.sh 除了编译代码, 还集成了固件打包功能。提供了一些额外的独立打包命令供开发者使用。但是使用的前提是已经编译过一次 U-Boot。

非FIT格式:

```
./make.sh trust          // 打包trust
./make.sh loader          // 打包loader
./make.sh trust <ini-file> // 打包trust时指定ini文件, 否则使用默认ini文件
./make.sh loader <ini-file> // 打包loader时指定ini文件, 否则使用默认ini文件
```

FIT格式:

```
// 旧脚本:
./make.sh spl              // 用tpl+spl替换ddr和miniloader, 打包成loader
./make.sh spl-s             // 用spl替换miniloader, 打包成loader

// 新脚本:
./make.sh --spl             // 用spl替换miniloader, 打包成loader
./make.sh --tpl              // 用tpl替换ddr, 打包成loader
./make.sh --tpl --spl        // 用tpl、spl替换ddr、miniloader, 打包成loader
./make.sh --spl-new          // ./make.sh --spl 命令只打包但不编译, 此命令会重新编译再打包。
```

如何鉴别新旧脚本? 如果新命令生效, make.sh就是新脚本。

## Chapter-4 系统模块

### Android BCB

BCB (Bootloader Control Block) 是Android控制系统启动流程而设计的一种和bootloader交互的机制。数据结构定义在 misc 分区偏移 16KB 或者 0 位置。

数据结构：

```
struct android_bootloader_message {
    char command[32];
    char status[32];
    char recovery[768];

    /* The 'recovery' field used to be 1024 bytes. It has only ever
     * been used to store the recovery command line, so 768 bytes
     * should be plenty. We carve off the last 256 bytes to store the
     * stage string (for multistage packages) and possible future
     * expansion. */
    char stage[32];

    /* The 'reserved' field used to be 224 bytes when it was initially
     * carved off from the 1024-byte recovery field. Bump it up to
     * 1184-byte so that the entire bootloader_message struct rounds up
     * to 2048-byte. */
    char reserved[1184];
};
```

command：启动命令，目前支持以下三个：

参数	功能
bootonce-bootloader	启动进入 U-Boot fastboot
boot-recovery	启动进入 recovery
boot-fastboot	启动进入 recovery fastboot (简称 fastbootd)

recovery：为进入 recovery mode 的附带命令，开头为 "recovery\n"，后面可以带多个参数，以"--"开头，以"\n"结尾，例如 "recovery\n--wipe\_ab\n--wipe\_package\_size=345\n--reason=wipePackage\n"：

参数	功能
update_package	OTA 升级
retry_count	进 recovery 升级次数，比如升级时意外掉电，依据该值重新进入 recovery 升级
wipe_data	erase user data (and cache), then reboot
wipe_cache	wipe cache (but not user data), then reboot
show_text	show the recovery text menu, used by some bootloader
sideload	
sideload_auto_reboot	an option only available in user-debug build, reboot the device without waiting
just_exit	do nothing, exit and reboot
locale	save the locale to cache, then recovery will load locale from cache when reboot
shutdown_after	return shutdown
wipe_all	擦除整个 userdata 分区
wipe_ab	wipe the current A/B device, with a secure wipe of all the partitions in RECOVERY_WIPE
wipe_package_size	wipe package size
prompt_and_wipe_data	prompt the user that data is corrupt, with their consent erase user data (and cache), then reboot
fw_update	SD 卡固件升级
factory_mode	工厂模式，主要用于做一些设备测试，如 PCBA 测试
pcba_test	进入 PCBA 测试
resize_partition	重新规划分区大小， android Q 的动态分区支持
rk_fwupdate	指定 rk SD/USB 固件升级，作用域仅限于 U-Boot

U-Boot阶段一般不需要用到和关心上述参数，仅供用户学习参考。

## AArch32

ARMv8 的 64 位芯片支持从 AArch64 退化到 AArch32 模式运行（跟 ARMv7 兼容），代码必须用 32 位编译。

用户可以通过这个宏来确认当前是否为 ARMv8 的 AArch32 模式：

```
CONFIG_ARM64_BOOT_AARCH32=y
```

## DTBO/DTO

为了便于用户对本章节内容的理解，这里先阅读附录章节，明确专业术语：DTB, DTBO, DTC, DTO, DTS, FDT。

它们之间的关系可以描述为：

- DTS 是用于描述 FDT 的文件；
- DTS 经过 DTC 编译后可生成 DTB/DTBO；
- DTB 和 DTBO 通过 DTO 操作可合并成一个新的 DTB；

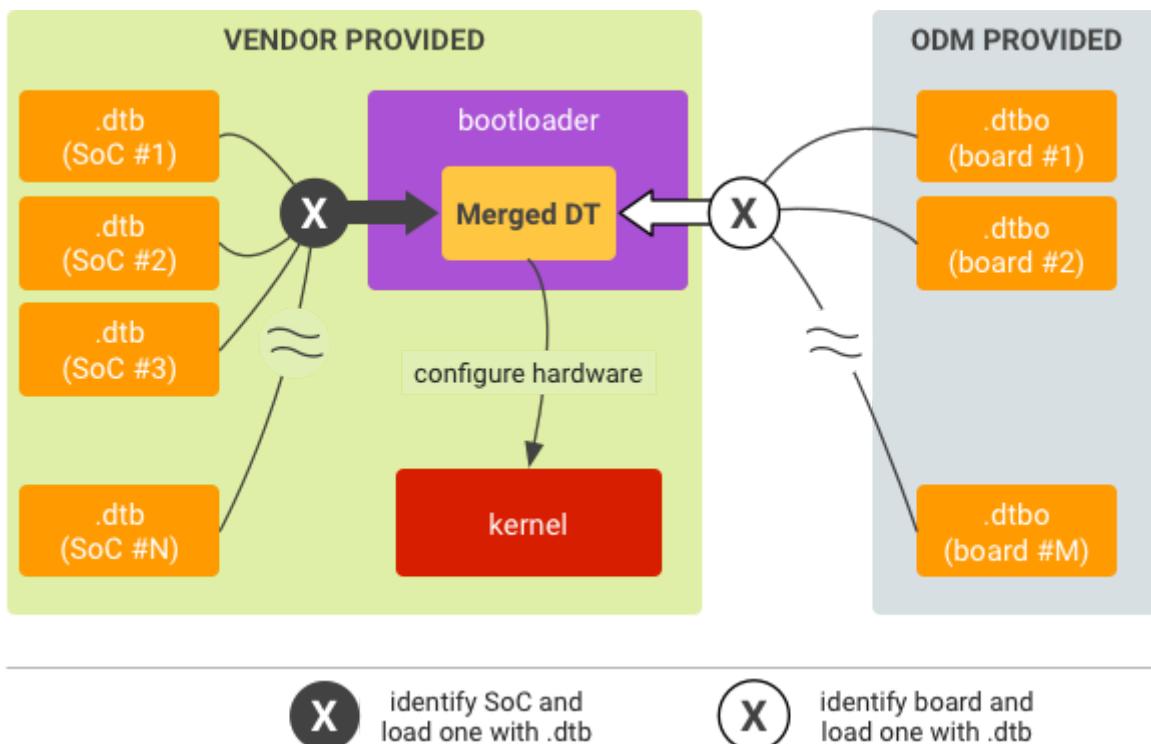
通常情况下很多用户习惯把“DTO”这个词的动作含义用“DTBO”来替代，下文中我们避开这个概念混用，明确：DTO 是一个动词概念，代表的是操作；DTBO 是一个名词概念，指的是用于叠加的次 dtb。

本章节更多知识可参考：<https://source.android.google.cn/devices/architecture/dto>。

## 原理介绍

DTO(Device Tree Overlay) 是 Android P 后引入且必须强制启用的功能，可让次设备树 Blob (DTBO) 叠加在已有的主设备树 Blob 上。DTO 可以维护系统芯片 Soc 设备树，并动态叠加针对特定设备的设备树，从而向树中添加节点并对现有树中的属性进行更改。

主设备树 Blob (\*.dtb) 一般由 Vendor 厂商提供，次设备树 Blob (\*.dtbo) 可由 ODM/OEM 等厂商提供，最后通过 bootloader 合并后再传递给 kernel。如下图：



图片来自：<https://source.android.google.cn/devices/architecture/dto>

需要注意：DTO 操作使用的 DTB 和 DTBO 的编译跟普通的 DTB 编译有区别，语法上有特殊区别：

使用 dtc 编译.dts 时，您必须添加选项-@以在生成的.dtbo 中添加\_symbols\_ 节点。\_symbols\_ 节点包含带标签的所有节点的列表，DTO 库可使用这个列表作为参考。如下示例：

1. 编译主.dts 的示例命令：

```
dtc -@ -o dtb -o my_main_dt.dtb my_main_dt.dts
```

2. 编译叠加层 DT .dts 的示例命令：

```
dtc -@ -o dtb -o my_overlay_dt.dtbo my_overlay_dt.dts
```

## DTO 启用

1. 配置使能：

```
CONFIG_CMD_DTIMG=y  
CONFIG_OF_LIBFDT_OVERLAY=y
```

2. board\_select\_fdt\_index()函数的实现。这是一个`_weak`函数，用户可以根据实际情况重新实现它。函数作用是在多份 DTBO 中获取用于执行 DTO 操作的那份 DTBO（返回 index 索引，最小从 0 开始），默认的 weak 函数返回的 index 为 0。

```
/*  
 * Default return index 0.  
 */  
__weak int board_select_fdt_index(ulong dt_table_hdr)  
{  
    /*  
     * User can use "dt_for_each_entry(entry, hdr, idx)" to iterate  
     * over all dt entry of DT image and pick up which they want.  
     *  

```

## DTO 结果

1. DTO 执行完成后，在 U-Boot 的开机信息中可以看到结果：

```
// 成功时的打印  
ANDROID: fdt overlay OK  
  
// 失败时的打印  
ANDROID: fdt overlay failed, ret=-19
```

通常引起失败的原因一般都是因为主/次设备书 blob 的内容存在不兼容引起，所以用户需要对它们的生成语法和兼容性要比较清楚。

2. DTO 执行成功后在给 kernel 的 cmdline 里追加如下信息，表明当前使用哪份 DTBO 进行 DTO 操作：

```
androidboot.dtbo_idx=1 // idx从0开始，这里表示选取idx=1的那份DTBO进行DTO操作
```

3. DTO 执行成功后可以在U-Boot命令行使用 `fdt` 命令查看DTB内容，确认改动是否生效。

## Cmdline

cmdline 是 U-Boot 向 kernel 传递参数的一个重要手段，诸如传递启动存储，设备状态等。目前 cmdline 参数有多个来源，由 U-Boot 进行拼接、过滤重复数据之后再传给 kernel。U-Boot 阶段的 cmdline 被保存在 bootargs 环境变量中。

U-Boot 最终是通过修改的 kernel DTB 里的 `/chosen/bootargs` 实现 cmdline 传递。

## 数据来源

- parameter.txt 文件

如果是 RK 格式的分区表，可以在 parameter.txt 里存放 cmdline 信息，例如：

```
CMDLINE: console=ttyFIQ0 androidboot.baseband=N/A  
androidboot.selinux=permissive androidboot.hardware=rk30board  
androidboot.console=ttyFIQ0 init=/init  
mtdparts=rk29xxnand:0x00002000@0x00002000(uboot),0x00002000@0x00004000(trust  
) ,
```

如果是 GPT 格式的分区表，在 parameter.txt 里存放 cmdline 信息是无效的。

- kernel dts 的 `/chosen/bootargs`，例如：

```
chosen {  
    bootargs = "earlyprintk=uart8250,mmio32,0xff30000 swiotlb=1  
    console=ttyFIQ0  
        androidboot.baseband=N/A androidboot.veritymode=enforcing  
        androidboot.hardware=rk30board androidboot.console=ttyFIQ0  
        init=/init kpti=0";  
};
```

- U-Boot：根据当前运行的状态，U-Boot 会动态追加一些内容到 cmdline。比如：

```
storagemedia=emmc androidboot.mode=emmc .....
```

- boot/recovery.img 固件头里的通常也会有 cmdline 字段信息。

## 数据含义

下面列出 RK 平台常用的 cmdline 参数含义。更多可参考内核文档：Documentation/admin-guide/kernel-parameters.txt。

- sdfwupdate：sd 升级卡标志，recovery 程序需要；
- root=PARTUUID：指定 rootfs(system) 分区的 UUID，仅 GPT 表支持
- skip\_initramfs：kernel 不使用 uboot 加载的 ramdisk，而使用 rootfs(system) 里的 ramdisk
- storagemedia：存储启动类型；
- console：kernel 打印口的配置信息；
- earlycon：在串口节点未建立之前，指定串口及其配置
- loop.max\_part：max\_part 用来设定每个 loop 的设备所能支持的分区数目
- rootwait：用于文件系统不能立即可用的情况，例如 emmc 初始化未完成，这个时候如果不设置 root\_wait 的话，就会 mount rootfs failed，而加上这个参数的话，则可以等待 driver 加载完成后，在从存储设备中 copy 出 rootfs，再 mount 的话，就不会提示失败了
- ro/rw：加载 rootfs 的属性，只读/读写
- firmware\_calss.path：指定驱动位置，如 wifi、bt、gpu 等

- dm="lroot none 0,0 4096 linear 98:3 0,4096 4096 linear 98:32" root=/dev/dm-0: Will boot to a rw dm-linear target of 8192 sectors split across two block devices identified by their major:minor numbers. After boot, udev will rename this target to /dev/mapper/lroot (depending on the rules). No uuid was assigned. 参考链接<https://android.googlesource.com/kernel/common/+/android-3.18/Documentation/device-mapper/boot.txt>
- androidboot.slot\_suffix: AB System 时为 kernel 指定从哪个 slot 启动
- androidboot.serialno: 为 kernel 及上层提供序列号, 例如 adb 的序列号等
- androidboot.verifiedbootstate: 安卓需求, 为上层提供 uboot 校验固件的状态, 有三种状态, 如下
  - green: If in LOCKED state and the key used for verification was not set by the end user
  - yellow: If in LOCKED state and the key used for verification was set by the end user
  - orange: If in the UNLOCKED state
- androidboot.hardware: 启动设备, 如 rk30board
- androidboot.verifymode: 指定验证分区的真实模式/状态 (即验证固件的完整性)
- androidboot.selinux: SELinux 是一种基于域-类型模型 (domain-type) 的强制访问控制 (MAC) 安全系统。有三种模式:
  - enforcing: 强制模式, 代表 SELinux 运作中, 且已经正确的开始限制 domain/type 了
  - permissive: 宽容模式: 代表 SELinux 运作中, 不过仅会有警告讯息并不会实际限制 domain/type 的存取。这种模式可以运来作为 SELinux 的 debug 之用
  - disabled: 关闭, SELinux 并没有实际运作
- androidboot.mode: 安卓启动方式, 有 normal 与 charger。
  - normal: 正常开机启动
  - charger: 关机后接电源开机, androidboot.mode 被设置为 charger, 这个值由 uboot 检测电源充电后设置到 bootargs 环境变量内
- androidboot.wificountrycode: 设置 wifi 国家码, 如 US, CN
- androidboot.baseband: 配置基带, RK 无此功能, 设置为 N/A
- androidboot.console: android 信息输出口配置
- androidboot.vbmeta.device=PARTUUID: 指定 vbmeta 在存储中的位置
- androidboot.vbmeta.hash\_alg: 设置 vbmeta hash 算法, 如 sha512
- androidboot.vbmeta.size: 指定 vbmeta 的 size
- androidboot.vbmeta.digest: 给 kernel 上传 vbmeta 的 digest, kernel 加载 vbmeta 后计算 digest, 并与此 digest 对比
- androidboot.vbmeta.device\_state: avb2.0 指定系统 lock 与 unlock

## ENV 操作

### 框架支持

ENV 是 U-Boot 框架中非常重要的一种数据管理方式, 通过 hash table 构建"键值"和"数据"进行映射管理, 支持"增/删/改/查"操作。通常, 我们把它管理的键值和数据统称为: 环境变量。U-Boot 支持把 ENV 数据保存在各种存储介质: NOWHERE/eMMC/FLASH/EEPROM/NAND/SPI\_FLASH/UBI...

配置:

```
// 默认配置: ENV保存在内存
CONFIG_ENV_IS_NOWHERE

// ENV保存在各种存储介质
```

```
CONFIG_ENV_IS_IN_MMC
CONFIG_ENV_IS_IN_NAND
CONFIG_ENV_IS_IN_EEPROM
CONFIG_ENV_IS_IN_FAT
CONFIG_ENV_IS_IN_FLASH
CONFIG_ENV_IS_IN_NVRAM
CONFIG_ENV_IS_IN_ONENAND
CONFIG_ENV_IS_IN_REMOTE
CONFIG_ENV_IS_IN_SPI_FLASH
CONFIG_ENV_IS_IN_UBI

// 任意已经接入到BLK框架层的存储介质（mmc除外），RK平台推荐使用！
CONFIG_ENV_IS_IN_BLK_DEV
```

框架代码：

```
./env/nowhere.c
./env/env_blk.c
./env/mmc.c
./env/nand.c
./env/eeprom.c
./env/embedded.c
./env/ext4.c
./env/fat.c
./env/flash.c
.....
```

## 相关接口

```
// 获取环境变量
char *env_get(const char *varname);
ulong env_get_ulong(const char *name, int base, ulong default_val);
ulong env_get_hex(const char *varname, ulong default_val);

// 修改或创建环境变量，value为NULL时等同于删除操作
int env_set(const char *varname, const char *value);
int env_set_ulong(const char *varname, ulong value);
int env_set_hex(const char *varname, ulong value);

// 把保存在存储介质上的ENV信息全部加载出来
int env_load(void);

// 把当前所有ENV信息保存到存储介质上
int env_save(void);
```

- env\_load(): 用户不需要调用，U-Boot 框架会在合适的启动流程里调用；
- env\_save(): 用户在需要的时刻主动调用，会把所有的 ENV 信息保存到 CONFIG\_ENV\_IS\_NOWHERE\_XXX 指定的存储介质；

## 高级接口

RK 提供了两个统一处理 ENV 的高级接口，具有创建、追加、替换的功能。主要是为了处理 bootargs 环境变量，但同样适用于其他环境变量操作。

```
/**
```

```

* env_update() - update sub value of an environment variable
*
* This add/append/replace the sub value of an environment variable.
*
* @varname: Variable to adjust
* @valude: value to add/append/replace
* @return 0 if OK, 1 on error
*/
int env_update(const char *varname, const char *varvalue);

/** 
* env_update_filter() - update sub value of an environment variable but
* ignore some key word
*
* This add/append/replace/igore the sub value of an environment variable.
*
* @varname: Variable to adjust
* @valude: Value to add/append/replace
* @ignore: Value to be ignored that in varvalue
* @return 0 if OK, 1 on error
*/
int env_update_filter(const char *varname, const char *varvalue, const char
*ignore);

```

1 env\_update()使用规则：

- 创建：如果 varname 不存在，则创建 varname 和 varvalue；
- 追加：如果 varname 已存在，varvalue 不存在，则追加 varvalue；
- 替换：如果 varname 已存在，varvalue 已存在，则用当前的 varvalue 替换原来的。比如：原来存在“storagemedia=emmc”，当前传入 varvalue 为“storagemedia=rknand”，则最终更新为“storagemedia=rknand”。

2 env\_update\_filter()是 env\_update()的扩展版本：在更新 env 的同时把 varvalue 里的某个关键字剔除；

3 特别注意：env\_update()和 env\_update\_filter()都是以空格和“=”作为分隔符对 ENV 内容进行单元分割，所以操作单元是：单个词、"key=value"组合词：

- 单个词：sdfwupdate、.....
- "key=value"组合词：storagemedia=emmc、init=/init、androidboot.console=ttyFIQ0、.....
- 上述两个接口无法处理长字符串单元。比如无法把“console=ttyFIQ0 androidboot.baseband=N/A androidboot.selinux=permissive”作为一个整体单元进行操作。

## 存储位置

env\_save()可以把 ENV 保存到存储介质，RK 平台的 ENV 的存储位置和大小定义如下：

```

if ARCH_ROCKCHIP
config ENV_OFFSET
    hex
    depends on !ENV_IS_IN_UBI
    depends on !ENV_IS_NOWHERE
    default 0x3f8000
    help
        offset from the start of the device (or partition)

config ENV_SIZE

```

```
hex
default 0x8000
help
    size of the environment storage area
endif
```

- 通常，ENV\_OFFSET 和 ENV\_SIZE 都不建议修改。

## 通用选项

目前常用的存储介质一般有：eMMC/sdmmc/Nandflash/Norflash 等。但 U-Boot 原生的 Nand、Nor 类 ENV 驱动都走 MTD 框架，而 RK 所有已支持的存储都是走 BLK 框架层，因此这些 ENV 驱动无法使用。

为此，RK 为接入 BLK 框架层的存储提供 `CONFIG_ENV_IS_IN_BLK_DEV` 配置选项：

- eMMC/sdmmc 的情况，依然选择 `CONFIG_ENV_IS_IN_MMC`；
- Nand、Nor 的情况，可以选择 `CONFIG_ENV_IS_IN_BLK_DEV`；

请用户使用前先阅读 Kconfig 的 `CONFIG_ENV_IS_IN_BLK_DEV` 定义。

```
// 已经默认被指定好，不需要修改
CONFIG_ENV_OFFSET
CONFIG_ENV_SIZE

// 通常不需要使用到
CONFIG_ENV_OFFSET_REDUND (optional)
CONFIG_ENV_SIZE_REDUND (optional)
CONFIG_SYS_MM_ENV_PART (optional)
```

注意：无论选择哪个 `CONFIG_ENV_IS_IN_XXX` 配置，请先阅读 Kconfig 中的定义说明，里面有子配置说明。

## fw\_printenv工具

`fw_printenv` 是 U-Boot 提供的一个给 linux 使用的 env 工具。通过这个工具，用户可以在 linux 上访问、修改 env 的内容。使用该工具要求 env 区域必须位于一个 kernel 可见的分区上（建议独立分区），本质上是通过 kernel sys 下的存储节点访问到 env 区域。

工具获取方式：

```
./make.sh env
```

执行完命令后获得：

```
./tools/env/fw_printenv      // env 读写工具
./tools/env/fw_env.config    // env 配置文件
./tools/env/README            // env 读写工具说明文档
```

使用方法请参考 README 文档。

## HW-ID DTB

RK 平台的 U-Boot 支持检测硬件上的 GPIO 或者 ADC 状态动态加载不同的 Kernel DTB，暂称为 HW-ID DTB (Hardware id DTB) 功能。

## 设计原理

通常硬件设计会经常更新版本和一些元器件，比如：屏幕、wifi 模组等。如果每一个硬件版本都要对应一套软件，维护起来就比较麻烦。所以需要 HW\_ID 功能实现一套软件可以适配不同版本的硬件。

针对不同硬件版本，软件上需要提供对应的 dtb 文件，同时还要提供 ADC/GPIO 硬件唯一值用于表征当前硬件版本（比如：固定的 adc 值、固定的某 GPIO 电平）。

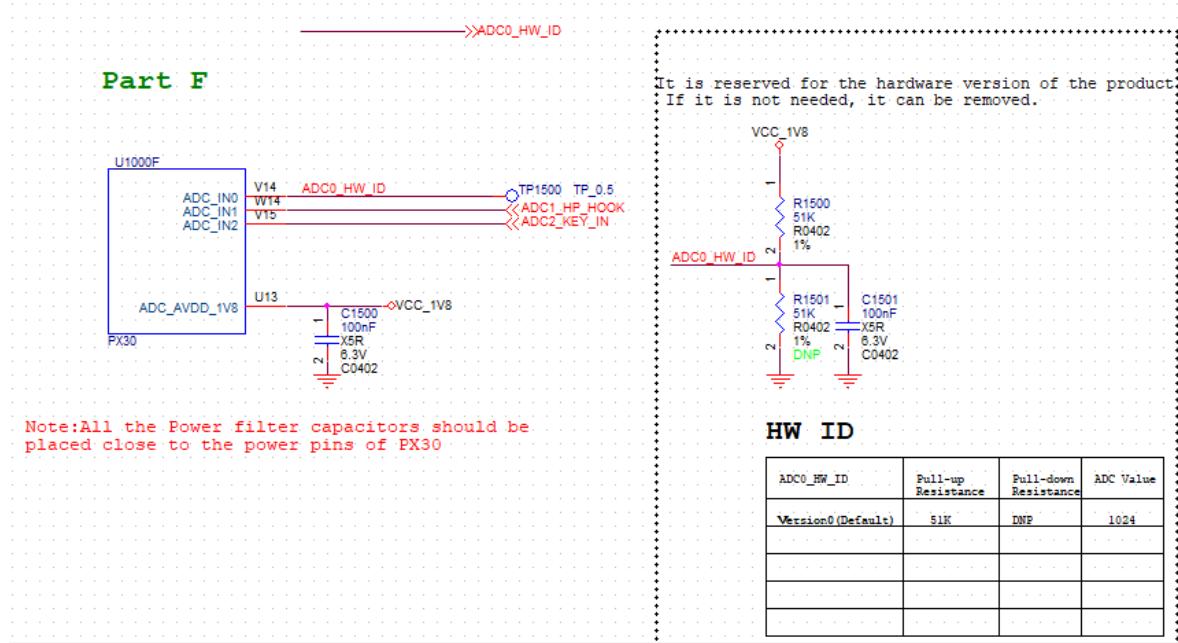
用户把这些和硬件版本对应的 dtb 文件全打包进同一个 resource.img，U-Boot 引导 kernel 时会检测硬件唯一值，从 resource.img 里找出和当前硬件版本匹配的 dtb 传给 kernel。

## 硬件参考

目前支持 ADC 和 GPIO 两种方式确定硬件版本。

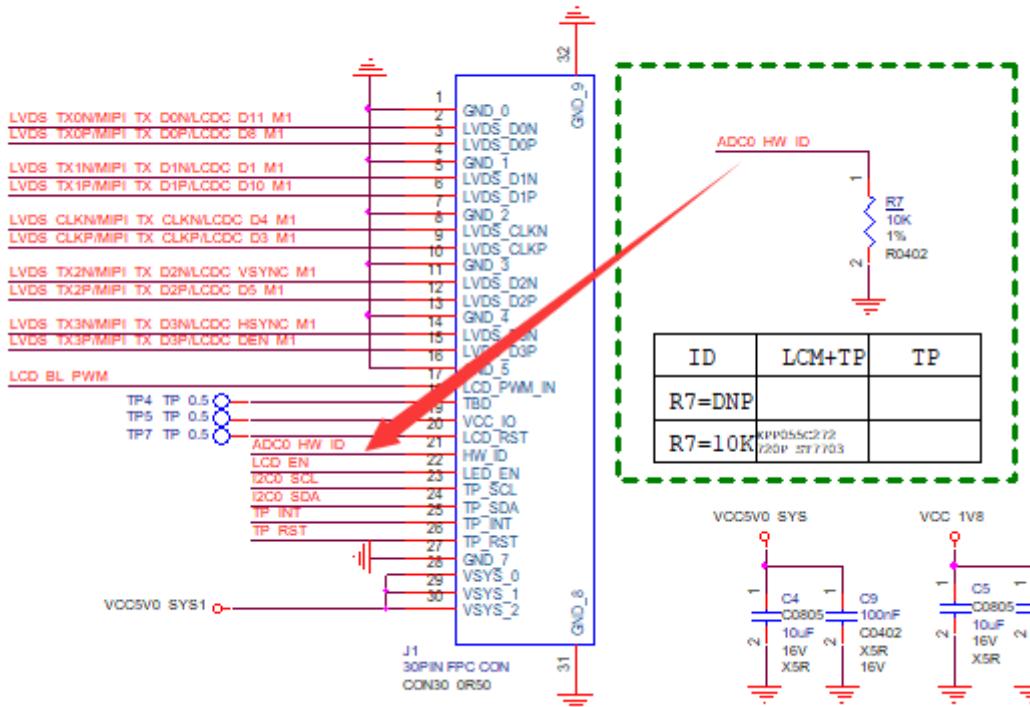
### ADC 参考设计

RK3326-EVB/PX30-EVB 主板上有预留分压电阻，不同的电阻分压有不同的 ADC 值，这样可以确定不同硬件版本：



配套使用的 MIPI 屏小板预留有另外一颗下拉电阻：

# LCD/TP Adapter Board



不同的 mipi 屏会配置不同的阻值，配合 EVB 主板确定一个唯一的 ADC 参数值。

目前 V1 版本的 ADC 计算方法：ADC 参数最大值为 1024，对应着 ADC\_IN0 引脚被直接上拉到供电电压 1.8V，MIPI 屏上有一颗 10K 的下拉电阻，接通 EVB 板后  $ADC = 1024 * 10K / (10K + 51K) = 167.8$ 。

## GPIO 参考设计

目前没有 GPIO 的硬件参考设计，用户可自己定制。

## DTB命名

用户需要将ADC/GPIO 的硬件唯一值信息体现在 dtb 文件名里。命名规则如下：

### ADC 作为 HW\_ID DTB：

- 文件名以“.dtb”结尾；
- HW\_ID 格式：#[controller]\_ch[channel]=[adcval]，称为一个完整单元  
 [controller]: dts 里面 ADC 控制器的节点名字。  
 [channel]: ADC 通道。  
 [adcval]: ADC 的中心值，实际有效范围是：adcval+-30。
- 每个完整单元必须使用小写字母，内部不能有空格；
- 多个单元之间通过#进行分隔，最多支持 10 个单元；

范例：

```
rk3326-evb-1p3-v10#saradc_ch2=111#saradc_ch1=810.dtb
rk3326-evb-1p3-v10#_saradc_ch2=569.dtb
```

### GPIO作为 HW\_ID DTB：

- 文件名以“.dtb”结尾；

- HW\_ID 格式: #gpio[pin]=[level], 称为一个完整单元  
 [pin]: GPIO 脚, 如 0a2 表示 gpio0a2  
 [level]: GPIO 引脚电平。
- 每个完整单元必须使用小写字母, 内部不能有空格;
- 多个单元之间通过#进行分隔, 最多支持 10 个单元;

范例:

```
rk3326-evb-lp3-v10#gpio0a2=0#gpio0c3=1.dtb
```

## DTB打包

kernel 仓库: scripts/mkmultidtb.py。通过该脚本可以把多个 dtb 打包进同一个 resource.img。

用户需要打开脚本文件把要打包的 dtb 文件写到 DTBS 字典里面, 并填上对应的 ADC/GPIO 的配置信息。

```
...
DTBS = {}
DTBS['PX30-EVB'] = OrderedDict([('rk3326-evb-lp3-v10', '#_saradc_ch0=166'),
                                 ('px30-evb-ddr3-lvds-v10', '#_saradc_ch0=512')])
...
...
```

上例中, 执行 scripts/mkmultidtb.py PX30-EVB 就会生成包含 3 份 dtb 的 resource.img:

- rk-kernel.dtb: rk 默认的 dtb, 不体现在上述字典中。所有 dtb 都没匹配成功时默认被使用。打包脚本会使用 DTBS 的第一个 dtb 作为默认的 dtb;
- rk3326-evb-lp3-v10#\_saradc\_ch0=166.dtb: 包含 ADC 信息的 rk3326 dtb 文件;
- px30-evb-ddr3-lvds-v10#\_saradc\_ch0=512.dtb: 包含 ADC 信息的 px30 dtb 文件;

## 功能启用

配置选项:

```
CONFIG_ROCKCHIP_HWID_DTB=y
```

驱动代码:

```
./arch/arm/mach-rockchip/resource_img.c // 具体实现: rockchip_read_hwid_dtb()
```

DTS 配置:

如果用GPIO作为硬件识别, 必须在rkxx-u-boot.dtsi中保留对应的pinctrl和gpio节点; ADC默认已使能。

例如: gpio0和gpio1作为识别:

```
...
&pinctrl {
    u-boot,dm-spl; // 追加该属性, 让该节点被保留在U-Boot DTB中。下同。
};

&gpio0 {
```

```
    u-boot,dm-spl;
};

&gpio1 {
    u-boot,dm-spl;
};

...
```

## 加载结果

```
.....
mmc0(part 0) is current device
boot mode: None
DTB: rk3326-evb-lp3-v10#_saradc_ch0=166.dtb // 打印匹配的DTB, 否则默认"rk-
kernel.dtb"
Using kernel dtb
.....
```

## AB系统

所谓的 A/B System 即把系统固件分为两份，分别称为slot-a, slot-b。系统可以从任意一个 slot 启动，当一个 slot 启动失败后还可以从另一个启动，同时升级时可以直接将固件拷贝到另一个 slot 上而无需进入系统升级模式。详细原理和流程请参考进阶原理章节。

目前RK平台的pre-loader和U-Boot都可以支持A/B系统。

## 芯片支持

芯片	Android 支持	Linux 支持
rk3128x	N	Y
rk3126	N	Y
rk322x	Y	Y
rk3288	N	Y
rk3368	N	Y
rk3328	N	Y
rk3399	N	Y
rk3308	N	Y
px30	Y	Y
rk3326	Y	Y

## 配置项

A/B System需要依赖LIBAVB，如下：

```
// A/B依赖的库
CONFIG_AVB_LIBAVB=y
CONFIG_AVB_LIBAVB_AB=y
CONFIG_AVB_LIBAVB_ATX=y
CONFIG_AVB_LIBAVB_USER=y
CONFIG_RK_AVB_LIBAVB_USER=y
// 使能A/B功能
CONFIG_ANDROID_AB=y
```

## 分区表

A/B System对分区表有要求：需要支持 A/B 的分区必须增加后缀 \_a 和 \_b。parameter.txt 参考如下：

```
FIRMWARE_VER:8.1
MACHINE_MODEL:RK3326
MACHINE_ID:007
MANUFACTURER: RK3326
MAGIC: 0x5041524B
ATAG: 0x00200800
MACHINE: 3326
CHECK_MASK: 0x80
PWR_HLD: 0,0,A,0,1
TYPE: GPT
CMDLINE:
mtdparts=rk29xxnand:0x00002000@0x00004000(uboot_a),0x00002000@0x00006000(uboot_b),
0x00002000@0x00008000(trust_a),0x00002000@0x0000a000(trust_b),0x00001000@0x0000
0c000(misc),0x00001000@0x0000d000(vbmeta_a),0x00001000@0x0000e000(vbmeta_b),0x00
020000@0x0000e000(boot_a),0x00020000@0x0002e000(boot_b),0x00100000@0x0004e000(sy
stem_a),0x00300000@0x0032e000(system_b),0x00100000@0x0062e000(vendor_a),0x001000
00@0x0072e000(vendor_b),0x00002000@0x0082e000(oem_a),0x00002000@0x00830000(oem_b)
,0x00100000@0x00832000(factory),0x00008000@0x842000(factory_bootloader),0x000800
00@0x008ca000(oem),-@0x0094a000(userdata)
```

## 注意事项

旧的U-Boot使能A/B系统之后，用户如果访问带a/b的分区，则传递给 `part_get_info_by_name()` 的分区名必须带slot后缀，例如：`"boot_a"` 或 `"boot_b"`。这样会增加很多冗余代码：用户必须先获取当前系统的slot，再进行字符串拼接，最终得到分区名。

新的代码优化了这个问题。如果用户的代码版本在下面这个提交点之后，则访问a/b的分区时可带、可不带slot后缀，框架层会自动探测当前系统使用哪个slot。例如：上述情况可直接使用 `"boot"`。

```
commit c6666740ee3b51c3e102bfba1ab95b78df29246
Author: Joseph Chen <chenjh@rock-chips.com>
Date:   Thu Oct 24 15:48:46 2019 +0800

common: android/rkimg: remove/clean android a/b (slot) code

- the partition disk layer takes over the responsibility of slot suffix
appending, we remove relative code to make file clean;
- put android a/b code together and name them to be easy understood,
this makes file easy to read.

Change-Id: Id8c838da682ce6098bd7192d7d7c64269f4e86ba
Signed-off-by: Joseph Chen <chenjh@rock-chips.com>
```

# AVB安全启动

AVB 为 Android Verified Boot，谷歌设计的一套固件校验流程，主要用于校验 boot system 等固件。Rockchip Secure Boot 参考通信中的校验方式及 AVB，实现一套完整的 Secure Boot 校验方案。

## Feature

- 安全校验
- 完整性校验
- 防回滚保护
- persistent partition 支持
- chained partitions 支持，可以与 boot, system 签名私钥一致，也可以由 oem 自己保存私钥，但必须由 PRK 签名

## 配置

开启 AVB 需要 trust 支持：

```
CONFIG_OPTEE_CLIENT=y
CONFIG_OPTEE_V1=y
CONFIG_OPTEE_ALWAYS_USE_SECURITY_PARTITION=y // 安全数据存储到security分区
```

- `CONFIG_OPTEE_V1`：适用平台有 312x,322x,3288,3228H,3368,3399。
- `CONFIG_OPTEE_V2`：适用平台有 3326,3308。
- `CONFIG_OPTEE_ALWAYS_USE_SECURITY_PARTITION`：eMMC 的 rpmb 不能用时才开这个宏，默认不开。

开启 AVB 相关配置：

```
CONFIG_AVB_LIBAVB=y
CONFIG_AVB_LIBAVB_AB=y
CONFIG_AVB_LIBAVB_ATX=y
CONFIG_AVB_LIBAVB_USER=y
CONFIG_RK_AVB_LIBAVB_USER=y
// 上面几个为必选，下面选择为支持 AVB 与 A/B 特性，两个特性可以分开使用。
CONFIG_ANDROID_AB=y //这个支持 A/B
CONFIG_ANDROID_AVB=y //这个支持 AVB
// 下面宏为仅有 efuse 的平台使用
CONFIG_ROCKCHIP_PRELOADER_PUB_KEY=y
// 下面宏需要严格unlock校验时候打开
CONFIG_RK_AVB_LIBAVB_ENABLE_ATH_UNLOCK=y
// 安全校验开启
CONFIG_AVB_VBMETA_PUBLIC_KEY_VALIDATE=y
// 如果需要cpuid作为challenge number，开启以下宏
CONFIG_MISC=y
CONFIG_ROCKCHIP_EFUSE=y
CONFIG_ROCKCHIP OTP=y
```

## 参考

因为AVB涉及的内容比较多，其余原理、配置请参考进阶原理章节。

## Fastboot

Fastboot是Android提供的一种借助USB和U-Boot进行交互的方式，一般用于获取设备信息、烧写固件等。

## 配置选项

```
// 使能配置
CONFIG_FASTBOOT
CONFIG_FASTBOOT_FLASH
CONFIG_USB_FUNCTION_FASTBOO

// 参数配置
CONFIG_FASTBOOT_BUF_ADDR
CONFIG_FASTBOOT_BUF_SIZE
CONFIG_FASTBOOT_FLASH_MMC_DEV
CONFIG_FASTBOOT_USB_DEV
```

## 触发方式

Fastboot 默认使用 Google adb 的 VID/PID，有如下几种触发方式：

- kernel的命令行执行：reboot fastboot
- U-Boot的命令行执行：fastboot usb 0
- 开机长按组合键：ctrl+f

## 命令支持

```
fastboot flash < partition > [ < filename > ]
fastboot erase < partition >
fastboot getvar < variable > | all
fastboot set_active < slot >
fastboot reboot
fastboot reboot-bootloader
fastboot flashing unlock
fastboot flashing lock
fastboot stage [ < filename > ]
fastboot get_staged [ < filename > ]
fastboot oem fuse at-perm-attr-data
fastboot oem fuse at-perm-attr
fastboot oem at-get-ca-request
fastboot oem at-set-ca-response
fastboot oem at-lock-vboot
fastboot oem at-unlock-vboot
fastboot oem at-disable-unlock-vboot
fastboot oem fuse at-bootloader-vboot-key
fastboot oem format
fastboot oem at-get-vboot-unlock-challenge
fastboot oem at-reset-rollback-index
```

## 命令详解

- fastboot flash < partition > [ < filename > ]  
功能：分区烧写  
举例：fastboot flash boot boot.img
- fastboot erase < partition >

功能: 擦除分区

举例: fastboot erase boot

- fastboot getvar < variable >

功能: 获取设备信息

举例: fastboot getvar version-bootloader

< variable > 参数:

version	/* fastboot 版本 */
version-bootloader	/* uboot 版本 */
version-baseband	
product	/* 产品信息 */
serialno	/* 序列号 */
secure	/* 是否开启安全校验 */
max-download-size	/* fastboot 支持单次传输最大字节数 */
logical-block-size	/* 逻辑块数 */
erase-block-size	/* 擦除块数 */
partition-type : < partition >	/* 分区类型 */
partition-size : < partition >	/* 分区大小 */
unlocked	/* 设备lock状态 */
off-mode-charge	
battery-voltage	
variant	
battery-soc-ok	
slot-count	/* slot 数目 */
has-slot: < partition >	/* 查看slot内是否有该分区名 */
current-slot	/* 当前启动的slot */
slot-suffixes	/* 当前设备具有的slot, 打印出其name */
slot-successful: < _a   _b >	/* 查看分区是否正确校验启动过 */
slot-unbootable: < _a   _b >	/* 查看分区是否被设置为unbootable */
slot-retry-count: < _a   _b >	/* 查看分区的retry-count次数 */
at-attest-dh	
at-attest-uuid	
at-vboot-state	

- fastboot getvar all

功能: 获取所有设备信息

- fastboot set\_active < slot >

功能: 设置重启的 slot

举例: fastboot set\_active \_a

- fastboot reboot

功能: 重启设备, 正常启动

举例: fastboot reboot

- fastboot reboot-bootloader

功能: 重启设备, 进入 fastboot 模式

举例: fastboot reboot-bootloader

- fastboot flashing unlock

功能: 解锁设备, 允许烧写固件

举例: fastboot flashing unlock

- fastboot flashing lock  
功能：锁定设备，禁止烧写  
举例：fastboot flashing lock
- fastboot stage [ < filename > ]  
功能：下载数据到设备端内存，内存起始地址为 CONFIG\_FASTBOOT\_BUF\_ADDR  
举例：fastboot stage permanent\_attributes.bin
- fastboot get\_staged [ < filename > ]  
功能：从设备端获取数据  
举例：fastboot get\_staged raw\_unlock\_challenge.bin
- fastboot oem fuse at-perm-attr  
功能：烧写 permanent\_attributes.bin 及 hash  
举例：

```
fastboot stage permanent_attributes.bin
fastboot oem fuse at-perm-attr
```
- fastboot oem fuse at-perm-attr-data  
功能：只烧写 permanent\_attributes.bin 到安全存储区域（RPMB）  
举例：

```
fastboot stage permanent_attributes.bin
fastboot oem fuse at-perm-attr-data
```
- fastboot oem at-get-ca-request
- fastboot oem at-set-ca-response
- fastboot oem at-lock-vboot  
功能：锁定设备  
举例：fastboot oem at-lock-vboot
- fastboot oem at-unlock-vboot  
功能：解锁设备，现支持 authenticated unlock  
举例：

```
fastboot oem at-get-vboot-unlock-challenge
fastboot get_staged raw_unlock_challenge.bin
./make_unlock.sh (见 make_unlock.sh 参考)

fastboot stage unlock_credential.bin
fastboot oem at-unlock-vboot
```
- 可以参考《how-to-generate-keys-about-avb.md》
- fastboot oem fuse at-bootloader-vboot-key  
功能：烧写 bootloader key hash  
举例：

```
fastboot stage bootloader-pub-key.bin
fastboot oem fuse at-bootloader-vboot-key
```
- fastboot oem format

功能：重新格式化分区，分区信息依赖于\$partitions

举例：fastboot oem format

- fastboot oem at-get-vboot-unlock-challenge

功能：authenticated unlock，需要获得 unlock challenge 数据

举例：参见 16. fastboot oem at-unlock-vboot

- fastboot oem at-reset-rollback-index

功能：复位设备的 rollback 数据

举例：fastboot oem at-reset-rollback-index

- fastboot oem at-disable-unlock-vboot

功能：使 fastboot oem at-unlock-vboot 命令失效

举例：fastboot oem at-disable-unlock-vboot

## SD和U盘

本章节主要介绍RK平台上的SD 和U盘的固件启动、升级。

### 机制原理

启动卡和升级卡制作完成后，都会在固件头部的固定存储偏移位置打上固定的tag，用于标记当前是启动卡还是升级卡。U-Boot 识别到这个标记后就会走启动或升级流程。其中：

- 启动卡：卡内只有一份完整的固件。U-Boot直接使用这份完成固件正常启动系统；
- 升级卡：卡内有两份固件。制作升级卡时，PC工具会烧录两份固件：一份仅含进入recovery模式必须的分区镜像（记为A固件），一份是完整的update.img 固件（记为B固件）。U-Boot使用A固件引导系统进入recovery模式，然后由recovery程序使用B固件完成升级工作。

### 特别注意：

- SD卡启动/升级是从bootrom这一级就开始支持；
- U盘启动/升级仅从U-Boot这一级开始支持，即用户至少要保证U-Boot能正常工作！

## 固件制作

RK平台上的SD和U盘启动卡、升级卡的制作流程是完全一致的，仅需两个步骤：

- 使用SDK目录下的 `RKTools/linux/Linux_Pack_Firmware/rockdev/` 工具生成 update.img。
- 使用SDDiskTool 工具把update.img 烧录到SD或U盘。如图：
  - 选择可移动磁盘
  - 选择 固件升级 或者 SD启动
  - 点击 开始创建



## SD 配置

SD启动/升级：各平台SDK发布的U-Boot已经默认使能该功能，用户不需要额外配置。

## USB 配置

U盘启动/升级：各平台SDK发布的U-Boot 默认没有使能。因为U-Boot原生的USB扫描命令很耗时，所以用户自己按需开启：

- 步骤1：烧写升级用的整套固件到本地存储 (eMMC/Nand/...等)，确认这套固件正常可用。
- 步骤2：插上U盘开机进入U-Boot命令行模式。执行 `usb start` 和 `usb info` 命令确认能正常识别U盘，否则请先调通U盘的识别。
- 步骤3：将满足步骤1的kernel DTB 拷贝一份命名成kern.dtb放到U-Boot的 `./dts/` 目录下。这份 kern.dtb会在编译U-Boot时被自动打包进uboot.img。

kern.dtb 用途：当本地存储分区的kernel dtb 有损坏时，U-Boot 使用kern.dtb 确保USB 能被正常初始化。

- 步骤4：U-Boot使能U盘启动/升级配置

```
CONFIG_ROCKCHIP_USB_BOOT=y
```

重新编译烧写uboot.img。

如果该过程提示uboot的固件过大无法打包生成，是因为步骤3加入kern.dtb引起的，请先裁掉一些不用的U-Boot配置。

## 功能生效

如何确认SD、U盘启动或升级功能生效。

用户可以擦除本地存储 (eMMC、Nand...) 上的kernel、resource、boot、recovery 等关键分区，确认插上SD/U盘后能进入kernel。

## 注意事项

- U盘初始化时会调用 `usb start` 命令，整个过程相对耗时；
- 如果启动/升级卡要支持GPT分区表，则SDDiskTool工具的版本要求 >= v1.59；
- 如果启动/升级卡要支持AB系统，则SDDiskTool工具的版本要求 >= v1.61；
- 因为U盘启动/升级功能是2019.11才增加的功能，所以相关仓库需要满足如下条件：

1. U-Boot 仓库要更新至如下提交点（建议）：

```
commit 369e944c844f783508b7839ae86a3418e2f63bc7
Author: Joseph Chen <chenjh@rock-chips.com>
Date:   Thu Dec 12 18:07:07 2019 +0800

fdt/Makefile: make u-boot-dtb.bin 8-byte aligned

The dts/kern.dtb is appended after u-boot-dtb.bin for u-disk boot.

Make sure u-boot-dtb.bin is 8-byte aligned to avoid data-abort on
calling: fdt_check_header(gd->fdt_blob_kern).

Signed-off-by: Joseph Chen <chenjh@rock-chips.com>
Change-Id: Id5f2daf0c5446e7ea828cb970d3d4879e3acda86
```

或者单独增加如下几个补丁的改动（估计比较困难）：

```
369e944 fdt/Makefile: make u-boot-dtb.bin 8-byte aligned
b3b57ac rockchip: board: fix always entering recovery on normal boot u-disk
e0cee41 rockchip: resource: add sha1/256 verify for kernel dtb
5e817a0 tools: rockchip: resource_tool: add sha1 for file entry
fc474da lib: sha256: add sha256_csum()
0ed06f1 rockchip: support boot from u-disk
01f0422 common: bootm: skip usb_stop() if usb is boot device
5704c89 fdtdec: support pack "kern.dtb" to the end of u-boot.bin
3bdef7e gpt: return 1 directly when test the mbr sector
```

1. rkbin 仓库要包含这个提交：

```
commit f9c0b0b72673a65865b00a8824908ca6f12ecc32
Author: Joseph Chen <chenjh@rock-chips.com>
Date:   Thu Nov 7 09:21:36 2019 +0800

tools: resource: add sha1 for file entry

Base on U-Boot next-dev branch:
(5e817a0 tools: rockchip: resource_tool: add sha1 for file entry)

Change-Id: Ife061cabacab488dbecf2a3245d58cc660091dbd
Signed-off-by: Joseph Chen <chenjh@rock-chips.com>
```

1. kernel 仓库要包含这个提交：

```
commit 078785057478c789bb033ba06925fa3a07e3130a
Author: Tao Huang <huangtao@rock-chips.com>
Date: Thu Nov 7 17:53:38 2019 +0800

    rk: scripts/resource_tool: add sha1 for file entry

    From u-boot 5e817a0ea427 ("tools: rockchip: resource_tool: add sha1 for
    file entry").
    Merge all c files to one resource_tool.c

    Change-Id: If63ba77d1f5a3660bd6ef87769bb456fa086ae71
    Signed-off-by: Tao Huang <huangtao@rock-chips.com>
```

- 如果用户手上的SDK比较旧，除了单独增加上述的补丁，建议跟负责recovery的工程师确认是否recovery有相关补丁。

## Chapter-5 驱动模块

### Interrupt

#### 框架支持

U-Boot 原生代码没有中断框架，RK 自己实现了一套用于支持 GICv2/v3，默认使能。

目前用到中断的场景：

- Pwrkey: U-Boot 充电时 CPU 会进入低功耗休眠，需要通过Pwrkey 中断唤醒 CPU；
- Timer: U-Boot 充电和测试用例中会用到 Timer 中断；
- Debug: 使能 CONFIG\_ROCKCHIP\_DEBUGGER 调试功能；

配置：

```
CONFIG_IRQ
CONFIG_GICV2
CONFIG_GICV3
```

框架代码：

```
./drivers/irq/irq-gpio-switch.c
./drivers/irq/irq-gpio.c
./drivers/irq/irq-generic.c
./drivers/irq/irq-gic.c
./drivers/irq/virq.c
./include/irq-generic.h
```

### 相关接口

```
// CPU本地中断开关
void enable_interrupts(void);
int disable_interrupts(void);

// GPIO转换成中断号
int gpio_to_irq(struct gpio_desc *gpio);
int phandle_gpio_to_irq(u32 gpio_phandle, u32 pin);
```

```

int hard_gpio_to_irq(unsigned gpio);

// 注册/释放中断回调
void irq_install_handler(int irq, interrupt_handler_t *handler, void *data);
void irq_free_handler(int irq);

// 使能/关闭中断
int irq_handler_enable(int irq);
int irq_handler_disable(int irq);

// 中断触发类型
int irq_set_irq_type(int irq, unsigned int type);

```

## 申请 IRQ

- 拥有独立硬件中断号的外设不需要额外转换，例如：pwm, timer等。
- GPIO 的 pin 脚没有独立的硬件中断号，需要额外转换申请。

一共有 3 种方式申请 GPIO 的 pin 脚中断号：

(1) 传入 `struct gpio_desc` 结构体

```

// 此方法可以动态解析dts配置，比较灵活、常用。
int gpio_to_irq(struct gpio_desc *gpio);

```

范例：

```

battery {
    compatible = "battery,rk817";
    .....
    dc_det_gpio = <&gpio2 7 GPIO_ACTIVE_LOW>;
    .....
};

```

```

struct gpio_desc dc_det;
int ret, irq;

ret = gpio_request_by_name_nodev(dev_ofnode(dev), "dc_det_gpio", 0,
                                &dc_det, GPIOD_IS_IN);
// 为了示例简单，省去返回值判断
if (!ret) {
    irq = gpio_to_irq(&dc_det);
    irq_install_handler(irq, ...);
    irq_set_irq_type(irq, IRQ_TYPE_EDGE_FALLING);
    irq_handler_enable(irq);
}

```

(2) 传入 gpio 的 phandle 和 pin

```

// 此方法可以动态解析dts配置，比较灵活、常用。
int phandle_gpio_to_irq(u32 gpio_phandle, u32 pin);

```

范例 (rk817 的中断引脚为 GPIO0\_A7) :

```
rk817: pmic@20 {
    compatible = "rockchip,rk817";
    reg = <0x20>;
    .....
    interrupt-parent = <&gpio0>;           // "&gpio0": 指向gpio0节点的phandle;
    interrupts = <7 IRQ_TYPE_LEVEL_LOW>;    // "7": pin脚;
    .....
};
```

```
u32 interrupt[2], phandle;
int irq, ret;

phandle = dev_read_u32_default(dev->parent, "interrupt-parent", -1);
if (phandle < 0) {
    printf("failed get 'interrupt-parent', ret=%d\n", phandle);
    return phandle;
}

ret = dev_read_u32_array(dev->parent, "interrupts", interrupt, 2);
if (ret) {
    printf("failed get 'interrupt', ret=%d\n", ret);
    return ret;
}

// 为了示例简单，省去返回值判断
irq = phandle_gpio_to_irq(phandle, interrupt[0]);
irq_install_handler(irq, pwrkey_irq_handler, dev);
irq_set_irq_type(irq, IRQ_TYPE_EDGE_FALLING);
irq_handler_enable(irq);
```

### (3) 强制指定 gpio

```
// 此方法直接强制指定 gpio，传入的 gpio 必须通过特殊的宏来声明才行，不够灵活，不建议使用。
int hard_gpio_to_irq(unsigned gpio);
```

范例 (GPIO0\_A0 申请中断) :

```
int gpio0_a0, irq;

// 为了示例简单，省去返回值判断
gpio0_a0 = RK_IRQ_GPIO(RK_GPIO0, RK_PA0);
irq = hard_gpio_to_irq(gpio0_a0);
irq_install_handler(irq, ...);
irq_handler_enable(irq);
```

## Clock

### 框架支持

clock 驱动使用 clk-uclass 框架和标准接口。

配置:

```
CONFIG_CLK
```

框架代码：

```
./drivers/clk/clk-uclass.c
```

平台驱动代码：

```
./drivers/clk/rockchip/...
```

## 相关接口

```
// 申请时钟
int clk_get_by_index(struct udevice *dev, int index, struct clk *clk);
int clk_get_by_name(struct udevice *dev, const char *name, struct clk *clk);

// 使能/关闭时钟
int clk_enable(struct clk *clk);
int clk_disable(struct clk *clk);

// 配置/获取频率
ulong (*get_rate)(struct clk *clk);
ulong (*set_rate)(struct clk *clk, ulong rate);

// 配置/获取相位
int (*get_phase)(struct clk *clk);
int (*set_phase)(struct clk *clk, int degrees);
```

## 时钟初始化

一共有三类接口涉及时钟初始化，如下先列出cru节点的信息，方便后续介绍。

```
cru: clock-controller@ff2b0000 {
    compatible = "rockchip,px30-cru";
    .....
    assigned-clocks =
        <&pmucru PLL_GPLL>, <&pmucru PCLK_PMU_PRE>,
        <&pmucru SCLK_WIFI_PMU>, <&cru ARMCLK>,
        <&cru ACLK_BUS_PRE>, <&cru ACLK_PERI_PRE>,
        <&cru HCLK_BUS_PRE>, <&cru HCLK_PERI_PRE>,
        <&cru PCLK_BUS_PRE>, <&cru SCLK_GPU>;
    assigned-clock-rates =
        <12000000000>, <1000000000>,
        <26000000>, <6000000000>,
        <2000000000>, <2000000000>,
        <1500000000>, <1500000000>,
        <1000000000>, <2000000000>;
    .....
}
```

### 第一类，平台基础时钟默认初始化： rkclk\_init()

各平台cru驱动probe会调用 `rkclk_init()` 完成 pll/cpu/bus 频率初始化，频率定义在 `cru_rkxxx.h`。例如 RK3399：

```
#define APLL_HZ      (600 * MHz)
#define GPLL_HZ       (800 * MHz)
```

```

#define CPLL_HZ          (384 * MHz)
#define NPLL_HZ          (600 * MHz)
#define PPLL_HZ          (676 * MHz)
#define PMU_PCLK_HZ     ( 48 * MHz)
#define ACLKM_CORE_HZ   (300 * MHz)
#define ATCLK_CORE_HZ   (300 * MHz)
#define PCLK_DBG_HZ      (100 * MHz)
#define PERIHP_ACLK_HZ  (150 * MHz)
#define PERIHP_HCLK_HZ  ( 75 * MHz)
#define PERIHP_PCLK_HZ  (37500 * KHz)
#define PERILP0_ACLK_HZ (300 * MHz)
#define PERILP0_HCLK_HZ (100 * MHz)
#define PERILP0_PCLK_HZ ( 50 * MHz)
#define PERILP1_HCLK_HZ (100 * MHz)
#define PERILP1_PCLK_HZ ( 50 * MHz)

.....

```

### 第二类，平台基础时钟二次初始化: `clk_set_defaults()`

各平台cru驱动probe可能会调用`clk_set_defaults()`解析且配置cru节点内`assigned-clocks/assigned-clock-parents/assigned-clock-rates`指定的频率（即重新配置频率），但是不包括arm频率。仅当实现`set_armclk_rate()`时才会配置。

除了cru之外，有需求的外设也可以在自己的probe里主动调用`clk_set_defaults()`，例如vop、gmac。

### 第三类，各模块时钟初始化: `clk_set_rate()`

大部分外设模块都是调用`clk_set_rate()`配置自己的频率。

## CPU提频

cpu开机提频的实现流程：

- 步骤1：cru节点的`assigned-clocks`里指定arm目标频率；
- 步骤2：cru驱动probe时调用`clk_set_defaults()`获取（但不会配置）步骤1的arm目标频率；
- 步骤3：实现`set_armclk_rate()`，这是一个`_weak`函数，它会配置从步骤2获取的arm目标频率；
- 步骤4：arm的regulator节点增加`regulator-init-microvolt = <...>`指定init电压避免电压不足；

## 时钟树

U-Boot框架没有提供时钟树管理，各平台增加了`soc_clk_dump()`用于简单打印时钟信息。例如：

```
CLK: (sync kernel. arm: enter 1200000 KHz, init 1200000 KHz, kernel 800000 KHz)
    apll 800000 KHz
    dpll 392000 KHz
    cpll 1000000 KHz
    gpll 1188000 KHz
    npll 24000 KHz
    ppll 100000 KHz
    hsclk_bus 297000 KHz
    mscclk_bus 198000 KHz
    lscclk_bus 99000 KHz
    mscclk_peri 198000 KHz
    lscclk_peri 99000 KHz
```

第一行打印的含义：

- `sync kernel`: cru 驱动通过 `clk_set_defaults()` 配置了 kernel cru 节点内指定的各总线频率 (arm 频率除外)；否则显示为 `sync uboot`；
- `enter 1200000 KHz`: 前级 Loader 进入 U-Boot 时的 arm 频率；
- `init 1200000 KHz`: U-Boot 的 arm 初始化频率，即 `APLL_HZ` 定义的频率；
- `kernel 800000 KHz`: 实现了 `set_armclk_rate()` 并设置了 kernel cru 节点里 `assigned-clocks` 指定的 arm 频率；否则显示: "kernel ON/A"；

## GPIO

### 框架支持

GPIO 驱动使用 `gpio-uclass` 框架和标准接口，用户必须通过 `struct gpio_desc` 才能访问到 `gpio`。

配置：

```
CONFIG_DM_GPIO
CONFIG_ROCKCHIP_GPIO
```

框架代码：

```
./drivers/gpio/gpio-uclass.c
```

驱动代码：

```
./drivers/gpio/rk_gpio.c
```

### 相关接口

```
// 申请/释放GPIO
int gpio_request_by_name(struct udevice *dev, const char *list_name,
                         int index, struct gpio_desc *desc, int flags);
int gpio_request_by_name_nodev(ofnode node, const char *list_name, int index,
                               struct gpio_desc *desc, int flags);
int gpio_request_list_by_name(struct udevice *dev, const char *list_name,
                            struct gpio_desc *desc_list, int max_count, int
flags);
int gpio_request_list_by_name_nodev(ofnode node, const char *list_name,
                                   struct gpio_desc *desc_list, int max_count,
                                   int flags);
```

```

int dm_gpio_free(struct udevice *dev, struct gpio_desc *desc)

// 配置GPIO方向。@flags: GPIO_IS_OUT (输出) 和 GPIO_IS_IN (输入)
int dm_gpio_set_dir_flags(struct gpio_desc *desc, ulong flags);

// 设置/获取GPIO电平
int dm_gpio_get_value(const struct gpio_desc *desc)
int dm_gpio_set_value(const struct gpio_desc *desc, int value)

```

### 注意事项：

`dm_gpio_get_value()` 的返回值表示active状态，而非电平的高或低。例如：

- gpios = <&gpio2 RK\_PD0 GPIO\_ACTIVE\_LOW>, 低电平时返回值为 1, 高电平为 0。
- gpios = <&gpio2 RK\_PD0 GPIO\_ACTIVE\_HIGH>, 低电平时返回值为 0, 高电平为 1。

`dm_gpio_set_value()` 的参数value也是同理，1: active, 0: inactive。

## Pinctrl

### 框架支持

pinctrl 驱动使用 pinctrl-uclass 框架和标准接口。

配置：

```

CONFIG_PINCTRL_GENERIC
CONFIG_PINCTRL_ROCKCHIP

```

框架代码：

```
./drivers/pinctrl/pinctrl-uclass.c
```

驱动代码：

```
./drivers/pinctrl/pinctrl-rockchip.c
```

## 相关接口

```

int pinctrl_select_state(struct udevice *dev, const char *statename) // 设置状态
int pinctrl_get_gpio_mux(struct udevice *dev, int banknum, int index) // 获取状态

```

pinctrl 框架会在各个driver probe 时自动为其设置"default"状态，用户一般不需要调用pinctrl接口。

## I2C

### 框架支持

i2c 驱动使用 i2c-uclass 框架和标准接口。

配置：

```
CONFIG_DM_I2C  
CONFIG_SYS_I2C_ROCKCHIP
```

框架代码:

```
./drivers/i2c/i2c-uclass.c
```

驱动代码:

```
./drivers/i2c/rk_i2c.c  
./drivers/i2c/i2c-gpio.c // gpio模拟i2c通讯, 目前用不到
```

## 相关接口

```
// i2c 读写  
int dm_i2c_read(struct udevice *dev, uint offset, uint8_t *buffer, int len)  
int dm_i2c_write(struct udevice *dev, uint offset, const uint8_t *buffer, int  
len)  
  
// 对上面接口的封装  
int dm_i2c_reg_read(struct udevice *dev, uint offset)  
int dm_i2c_reg_write(struct udevice *dev, uint offset, unsigned int val);
```

## Display

### 框架支持

RK U-Boot 目前支持的显示接口包括: RGB、LVDS、EDP、MIPI 和 HDMI, 未来还会加入 CVBS、DP 等。U-Boot 显示的 logo 图片来自 kernel 根目录下的 logo.bmp 和 logo\_kernel.bmp, 它们被打包在 resource.img 里。

对图片的要求:

- 8bit 或者 24bit BMP 格式;
- logo.bmp 和 logo\_kernel.bmp 的图片分辨率大小一致;
- 对于 rk312x/px30/rk3308 这种基于 vop lite 结构的芯片, 由于 VOP 不支持镜像, 而 24bit 的 BMP 图片是按镜像存储, 所以如果发现显示的图片做了 y 方向的镜像, 请在 PC 端提前将图片做好 y 方向的镜像。

配置:

```
CONFIG_DM_VIDEO  
CONFIG_DISPLAY  
CONFIG_DRM_ROCKCHIP  
CONFIG_DRM_ROCKCHIP_PANEL  
CONFIG_DRM_ROCKCHIP_DW_MIPI_DSI  
CONFIG_DRM_ROCKCHIP_DW_HDMI  
CONFIG_DRM_ROCKCHIP_LVDS  
CONFIG_DRM_ROCKCHIP_RGB  
CONFIG_DRM_ROCKCHIP_RK618  
CONFIG_ROCKCHIP_DRM_TVE  
CONFIG_DRM_ROCKCHIP_ANALOGIX_DP  
CONFIG_DRM_ROCKCHIP_INNO_VIDEO_COMBO_PHY  
CONFIG_DRM_ROCKCHIP_INNO_VIDEO_PHY
```

框架代码:

```
drivers/video/drm/rockchip_display.c  
drivers/video/drm/rockchip_display.h
```

驱动文件:

```
vop:  
    drivers/video/drm/rockchip_crtc.c  
    drivers/video/drm/rockchip_crtc.h  
    drivers/video/drm/rockchip_vop.c  
    drivers/video/drm/rockchip_vop.h  
    drivers/video/drm/rockchip_vop_reg.c  
    drivers/video/drm/rockchip_vop_reg.h  
  
rgb:  
    drivers/video/drm/rockchip_rgb.c  
    drivers/video/drm/rockchip_rgb.h  
  
lvds:  
    drivers/video/drm/rockchip_lvds.c  
    drivers/video/drm/rockchip_lvds.h  
  
mipi:  
    drivers/video/drm/rockchip_mipi_dsi.c  
    drivers/video/drm/rockchip_mipi_dsi.h  
    drivers/video/drm/rockchip-inno-mipi-dphy.c  
  
edp:  
    drivers/video/drm/rockchip_analogix_dp.c  
    drivers/video/drm/rockchip_analogix_dp.h  
    drivers/video/drm/rockchip_analogix_dp_reg.c  
    drivers/video/drm/rockchip_analogix_dp_reg.h  
  
hdmi:  
    drivers/video/drm/dw_hdmi.c  
    drivers/video/drm/dw_hdmi.h  
    drivers/video/drm/rockchip_dw_hdmi.c  
    drivers/video/drm/rockchip_dw_hdmi.h  
  
panel:  
    drivers/video/drm/rockchip_panel.c  
    drivers/video/drm/rockchip_panel.h
```

## 相关接口

```

// 显示 U-Boot logo 和 kernel logo:
void rockchip_show_logo(void);

// 显示 bmp 图片, 目前主要用于充电图片显示:
void rockchip_show_bmp(const char *bmp);

// 将 U-Boot 中的一些变量通过 dtb 传给内核。
// 包括 kernel logo 的大小、地址、格式, crtcon 输出扫描时序以及过扫描的配置。
// 未来还会加入 BCSH 等相关变量配置。
void rockchip_display_fixup(void *blob);

```

## DTS 配置

```

reserved-memory {
    #address-cells = <2>;
    #size-cells = <2>;
    ranges;

    drm_logo: drm-logo@00000000 {
        compatible = "rockchip,drm-logo";
        // 预留buffer用于kernel logo的存放, 具体地址和大小在U-Boot中会修改
        reg = <0x0 0x0 0x0 0x0>;
    };
};

&route-edp {
    status = "okay"; // 使能U-Boot logo显示功能
    logo_uboot = "logo.bmp"; // 指定U-Boot logo显示的图片
    logo_kernel = "logo_kernel.bmp"; // 指定kernel logo显示的图片
    logo_mode = "center"; // center: 居中显示, fullscreen: 全屏显示
    charge_logo_mode = "center"; // center: 居中显示, fullscreen: 全屏显示
    connect = <&vopb_out_edp>; // 确定显示通路, vopb->edp->panel
};

&edp {
    status = "okay"; // 使能edp
};

&vopb {
    status = "okay"; // 使能vopb
};

&panel {
    "simple-panel";
    ...
    status = "okay";

    disp_timings: display-timings {
        native-mode = <&timing0>;
        timing0: timing0 {
            ...
        };
    };
};

```

## defconfig

目前除了 RK3308 之外的其他平台，U-Boot 的 defconfig 已经默认支持显示，只要在 dts 中将显示相关的信息配置好即可。RK3308 考虑到启动速度等一些原因默认不支持显示，需要在 defconfig 中加入如下修改：

```
--- a/configs/evb-rk3308_defconfig
+++ b/configs/evb-rk3308_defconfig
@@ -4,7 +4,6 @@ CONFIG_SYS_MALLOC_F_LEN=0x2000
CONFIG_ROCKCHIP_RK3308=y
CONFIG_ROCKCHIP_SPL_RESERVE_IRAM=0x0
CONFIG_RKIMG_BOOTLOADER=y
-# CONFIG_USING_KERNEL_DTB is not set
CONFIG_TARGET_EVB_RK3308=y
CONFIG_DEFAULT_DEVICE_TREE="rk3308-evb"
CONFIG_DEBUG_UART=y
@@ -55,6 +54,11 @@ CONFIG_USB_GADGET_DOWNLOAD=y
CONFIG_G_DNL_MANUFACTURER="Rockchip"
CONFIG_G_DNL_VENDOR_NUM=0x2207
CONFIG_G_DNL_PRODUCT_NUM=0x330d
+CONFIG_DM_VIDEO=y
+CONFIG_DISPLAY=y
+CONFIG_DRM_ROCKCHIP=y
+CONFIG_DRM_ROCKCHIP_RGB=y
+CONFIG_LCD=y
CONFIG_USE_TINY_PRINTF=y
CONFIG_SPL_TINY_MEMSET=y
CONFIG_ERRNO_STR=y
```

### 关于 upstream defconfig 配置的说明：

upstream 维护了一套 Rockchip U-Boot 显示驱动，目前主要支持 RK3288 和 RK3399 两个平台：

```
./drivers/video/rockchip/
```

如果要使用这套驱动，可以打开 CONFIG\_VIDEO\_ROCKCHIP，同时关闭 CONFIG\_DRM\_ROCKCHIP。跟我们目前 SDK 使用的显示驱动对比，后者的优势有：

- 支持的平台和显示接口更全面；
- HDMI、DP 等显示接口可以根据用户的设定输出指定分辨率，过扫描效果，显示效果调节效果等。
- U-Boot logo 可以平滑过渡到 kernel logo 直到系统起来；

## LOGO分区

用户如果有动态更新开机LOGO的需求（一般在应用层发起更新），可以通过独立的LOGO分区实现。

操作步骤：

- 分区表中增加独立的LOGO分区
- 用户根据需要以某种方式动态更新LOGO分区中的图片。更新时，用户直接把原始图片更新到 LOGO分区中即可，不需要任何打包。当LOGO分区的图片无效时，则仍旧使用resource文件中默认的图片。

### LOGO分区支持情况：

如果代码只包含如下提交，则LOGO分区仅支持1张图片，只能替换默认的 logo.bmp：

```
1d30bcc rockchip: resource: support parse "logo" partition picture
```

如果代码包含如下提交，LOGO分区支持2张图片：图片1用于替换logo.bmp，图片2用于替换logo\_kernel.bmp。两张图片紧挨着，图片之间保持512字节对齐，顺序不可更换。

```
commit 07f987d8d495380787203e2bc2accd44100e6051
Author: Joseph Chen <chenjh@rock-chips.com>
Date:   Sun Dec 8 18:00:37 2019 +0800

    rockchip: resource: support parse logo_kernel.bmp from logo partition

    "logo" partition layout, not change order:

    |-----| 0x00
    | raw logo.bmp      |
    |-----| N*512-byte aligned
    | raw logo_kernel.bmp |
    |-----|


    N: the sector count of logo.bmp

Signed-off-by: Joseph Chen <chenjh@rock-chips.com>
Change-Id: I2deba013d3963c99664c5bfd69693835a46ba48f
```

假设图片分别为logo.bmp和logo\_kernel.bmp。logo.img 打包命令：

```
cat logo.bmp > logo.img && truncate -s %512 logo.img && cat logo_kernel.bmp >> logo.img
```

把生成的logo.img烧写到logo分区即可，开机后看到"LOGO: "打印：

```
U-Boot 2017.09-g042c01531e-210512dirty #cjh (May 14 2021 - 11:25:03 +0800)

Model: Rockchip RK3568 Evaluation Board
PreSerial: 2, raw, 0xfe660000
DRAM: 2 GiB
Sysmem: init
Relocation Offset: 7d34f000, fdt: 7b9f8758
Using default environment

dwmmc@fe2b0000: 1, dwmmc@fe2c0000: 2, sdhci@fe310000: 0
Bootdev(atags): mmc 0
MMC0: HS200, 200Mhz
PartType: EFI
boot mode: normal
FIT: No fdt blob
Android 11.0, Build 2021.4, v2
Found DTB in boot part
// 如下打印说明logo.img分区的图片被正确识别到。
LOGO: logo.bmp
LOGO: logo_kernel.bmp
DTB: rk-kernel.dtb
HASH(c): OK
.....
```

## Pmic/Regulator

## 框架支持

PMIC/Regulator 驱动使用 pmic-uclass、regulator-uclass 框架和标准接口。

PMIC支持:

```
rk805/rk808/rk809/rk816/rk817/rk818
```

Regulator支持:

```
rk805/rk808/rk809/rk816/rk817/rk818/syr82x/tcs452x/fan53555/pwm/gpio/fixed
```

配置:

```
CONFIG_DM_PMIC
CONFIG_PMIC_CHILDREN
CONFIG_PMIC_RK8XX          // 适用于目前所有RK8XX系列芯片
CONFIG_DM_REGULATOR
CONFIG_REGULATOR_PWM
CONFIG_REGULATOR_RK8XX      // 适用于目前所有RK8XX系列芯片
CONFIG_REGULATOR_FAN53555
```

框架代码:

```
./drivers/power/pmic/pmic-uclass.c
./drivers/power/regulator/regulator-uclass.c
```

驱动文件:

```
./drivers/power/pmic/rk8xx.c
./drivers/power/regulator/rk8xx.c
./drivers/power/regulator/fixed.c
./drivers/power/regulator/gpio-regulator.c
./drivers/power/regulator/pwm_regulator.c
./drivers/power/regulator/fan53555_regulator.c
```

## 相关接口

```
// 获取regulator。@platname: “regulator-name”指定的名字，如: vdd_arm、vdd_logic;
int regulator_get_by_platname(const char *platname, struct udevice **devp);

// 使能/关闭
int regulator_get_enable(struct udevice *dev);
int regulator_set_enable(struct udevice *dev, bool enable);
int regulator_set_suspend_enable(struct udevice *dev, bool enable);
int regulator_get_suspend_enable(struct udevice *dev);

// 配置/获取电压
int regulator_get_value(struct udevice *dev);
int regulator_set_value(struct udevice *dev, int uv);
int regulator_set_suspend_value(struct udevice *dev, int uv);
int regulator_get_suspend_value(struct udevice *dev);
```

## init 电压

目前有两种方式为某路regulator设置初始化电压输出，前提是必须配置 regulator-boot-on：

- 配置 regulator-min-microvolt 和 regulator-max-microvolt 为相同值；
- 配置 regulator-init-microvolt = <...>

```
vdd_arm: DCDC_REG1 {  
    regulator-name = "vdd_arm";  
    regulator-boot-on; // 必须配置  
    regulator-min-microvolt = <712500>;  
    regulator-max-microvolt = <1450000>;  
    regulator-init-microvolt = <1100000>; // 设置初始化电压为1.1v  
    ....  
};
```

## 跳过初始化

如果想跳过某路regulator的初始化，可增加 regulator-loader-ignore。

```
vdd_arm: DCDC_REG1 {  
    regulator-name = "vdd_arm";  
    regulator-loader-ignore; // 仅对U-Boot阶段的regulator初始化有效，kernel无效  
    ....  
};
```

## Charge

### 框架支持

U-Boot 原生代码不支持充电功能，RK自己实现了一套。

充电涉及的模块较多：Display、PMIC、电量计、充电动画、pwrkey、led、CPU低功耗休眠、Timer等。

电量计支持：

```
RK809/RK816/RK817/RK818/cw201x.
```

配置：

```
// 框架  
CONFIG_DM_CHARGE_DISPLAY  
CONFIG_CHARGE_ANIMATION  
CONFIG_DM_FUEL_GAUGE  
// 驱动  
CONFIG_POWER_FG_CW201X  
CONFIG_POWER_FG_RK818  
CONFIG_POWER_FG_RK817  
CONFIG_POWER_FG_RK816
```

充电动画：

```
./drivers/power/charge-display-uclass.c
```

充电动画驱动：

```
// 负责管理整个充电过程，它会获取电量、充电类型、按键事件，发起低功耗休眠等。  
./drivers/power/charge_animation.c
```

电量计框架：

```
./drivers/power/fuel_gauge/fuel_gauge_uclass.c
```

电量计驱动：

```
./drivers/power/fuel_gauge/fg_rk818.c  
./drivers/power/fuel_gauge/fg_rk817.c // rk809复用  
./drivers/power/fuel_gauge/fg_rk816.c  
.....
```

逻辑流程：

```
charge-display-uclass.c  
=> charge_animation.c  
=> fuel_gauge_uclass.c  
=> fg_rkxx.c
```

## 打包图片

充电图片位于 `./tools/images/` 目录，需要打包进 `resource.img` 才能被充电框架显示。

内核编译得到的 `resource.img` 默认没打包充电图片，需要在 U-Boot 里另外单独打包。

```
$ ls tools/images/  
battery_0.bmp battery_1.bmp battery_2.bmp battery_3.bmp battery_4.bmp  
battery.bmp battery_fail.bmp
```

打包命令：

```
./pack_resource.sh <input resource.img> 或  
.scripts/pack_resource.sh <input resource.img>
```

打包信息：

```
./pack_resource.sh /home/cjh/3399/kernel/resource.img  
  
Pack ./tools/images/ & /home/guest/3399/kernel/resource.img to resource.img ...  
Unpacking old image(/home/guest/3399/kernel/resource.img):  
rk-kernel.dtb logo.bmp logo_kernel.bmp  
Pack to resource.img successed!  
Packed resources:  
rk-kernel.dtb battery_1.bmp battery_2.bmp battery_3.bmp battery_4.bmp  
battery.bmp battery_fail.bmp logo.bmp logo_kernel.bmp battery_0.bmp  
  
resource.img is packed ready
```

成功后会在 U-Boot 根目录下生成包含图片的 `resource.img`，通过 `hd` 命令确认内容：

```
hd resource.img | less
```

```

00000000  52 53 43 45 00 00 00 00  01 01 01 00 0a 00 00 00 |RSCE.....|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |.....|
*.
-----
*.
00000400  45 4e 54 52 62 61 74 74  65 72 79 5f 31 2e 62 6d |ENTRbattery_1.bm|
// 图片1
00000410  70 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |p.....|
00000420  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |.....|
*.
00000500  00 00 00 00 4d 00 00 00  9c 18 00 00 00 00 00 00 |....M.....|
00000510  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |.....|
*.
00000600  45 4e 54 52 62 61 74 74  65 72 79 5f 32 2e 62 6d |ENTRbattery_2.bm|
// 图片2
00000610  70 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |p.....|
00000620  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |.....|
.....
```

## DTS 配置

DTS充电节点:

```

charge-animation {
    compatible = "rockchip,uboot-charge";
    status = "okay";

    rockchip,uboot-charge-on = <0>;           // 是否开启U-Boot充电
    rockchip,android-charge-on = <1>;          // 是否开启Android充电

    rockchip,uboot-exit-charge-level = <5>;     // U-Boot充电时，允许开机的最低电量
    rockchip,uboot-exit-charge-voltage = <3650>; // U-Boot充电时，允许开机的最低电压
    rockchip,screen-on-voltage = <3400>;        // U-Boot充电时，允许点亮屏幕的最低电压

    rockchip,uboot-low-power-voltage = <3350>; // U-Boot无条件强制进入充电模式的最低电压

    rockchip,system-suspend = <1>;              // 是否灭屏时进入trust低功耗待机（要ATF支持）
    rockchip,auto-off-screen-interval = <20>;   // 自动灭屏超时，单位秒，默认15s
    rockchip,auto-wakeup-interval = <10>;       // 休眠自动唤醒时间，单位秒。如果值为0或没
                                                // 有这个属性，则禁止休眠自动唤醒，一般用于
                                                // 压力测试使用
    rockchip,auto-wakeup-screen-invert = <1>;   // 休眠自动唤醒时是否需要亮/灭屏
};
```

## 系统休眠

Pwrkey按键:

- 短按 pwrkey 可以亮/灭屏，灭屏时系统会进入低功耗模式；
- 长按 pwrkey 可开机进入系统

低功模式有2种，通过 `rockchip,system-suspend = <VAL>` 选择：

- VAL为0：cpu wfi 模式。此时不处理外设，仅仅cpu 进入低功耗模式；
- VAL为1：system suspend 模式，需要ATF 支持才有效。类同kernel的系统深度待机，整个SoC进入待机。

ATF支持低功耗的版本要求：

芯片	最低版本
RK3399	rk3399_bl31_v1.32.elf
RK3368	rk3368h_bl31_v2.22.elf
PX30	px30_bl31_v1.05.elf
RK3326	rk3326_bl31_v1.05.elf
RK3308	rk3308_bl31_v2.00.elf、rk3308_bl31_aarch32_v2.20.elf
RK312X	rk3126_tee_ta_v1.39.bin
RK3288	rk3288_tee_ta_v1.43.bin

## 更换图片

1. 更换 `./tools/images/` 目录下的图片（采用 8bit 或 24bit bmp），使用命令 `ls | sort` 确认图片排列顺序是低电量到高电量，使用 `pack_resource.sh` 脚本把图片打包进 `resource.img`；
2. 修改 `./drivers/power/charge_animation.c` 里的图片和电量关系；

```
/*
 * IF you want to use your own charge images, please:
 *
 * 1. Update the following 'image[]' to point to your own images;
 * 2. You must set the failed image as last one and soc = -1 !!!
 */

static const struct charge_image image[] = {
    { .name = "battery_0.bmp", .soc = 5, .period = 600 },
    { .name = "battery_1.bmp", .soc = 20, .period = 600 },
    { .name = "battery_2.bmp", .soc = 40, .period = 600 },
    { .name = "battery_3.bmp", .soc = 60, .period = 600 },
    { .name = "battery_4.bmp", .soc = 80, .period = 600 },
    { .name = "battery.bmp", .soc = 100, .period = 600 },
    { .name = "battery_fail.bmp", .soc = -1, .period = 1000 },
};

// @name: 图片的名字;
// @soc: 图片对应的电量;
// @period: 图片刷新时间 (单位: ms);
// 注意: 最后一张图片必须是 fail 图片, 且“soc=-1”不可改变 !!
```

## 充电灯

实际产品中用户对 led 的控制需求各不相同，因此充电框架仅支持 2 个灯。充电时刻 led、充满时刻 led：

- 充满时刻 led：充电时候，电量有变化的时候，才会翻转 led 显示；

- 充满时刻 led：电量 100%充满时，才会点亮 led 灯；

上述2个Led 仅是一个demo，用户请根据自己的需求修改代码。

配置选项：

```
CONFIG_LED_CHARGING_NAME
CONFIG_LED_CHARGING_FULL_NAME
```

这两个配置选项用于指定 led 的 label 属性，请参考Led章节。

## Storage

存储驱动使用标准的存储框架，访问接口对接到 BLK 层用于支持文件系统。目前支持的存储设备：eMMC、Nand flash、SPI Nand flash、SPI Nor flash，其中 flash 相关的框架如下：

简称	主要支持的颗粒类型	主控驱动	flash 框架	注册设备类型	主要支持文件系统
rknand 方案	MLC TLC Nand	drivers/rkand	drivers/rkand	block 设备	FAT、EXT、SquashFS
rkflash 方案	SLC Nand、SPI Nand	drivers/rkflash	drivers/rkflash	block 设备	FAT、EXT、SquashFS
rkflash 方案 (SPI Nor 支持)	SPI Nor	drivers/rkflash	drivers/rkflash	block 或 mtd 设备	SquashFS、JFFS2
SLC Nand 开源方案	SLC Nand	drivers/mtd/nand/raw	drivers/mtd/nand/raw	mtd 设备	UBI
SPI Nand 开源方案	SPI Nand	drivers/spi/rockchip_sfc.c	drivers/mtd/nand/raw	mtd 设备	UBI
SPI Nor 开源方案	SPI Nor	drivers/spi/rockchip_sfc.c	drivers/mtd/spi	mtd 或 mtd block 设备	SquashFS、JFFS2

说明：

- rkflash 与 开源方案中关于 Nand flash 的支持主要区别在于：rkflash 集成 rk ftl (Flash Transfer Layer) 在存储驱动中，而开源方案 ftl 部分则依赖于文件系统自身的 flash 的管理，例如 UBI 文件系统支持坏块管理、磨损均衡等适合 Nand flash 的文件系统特性。

## 框架支持

### rknand

rknand 是针对大容量 Nand flash 设备所设计的存储驱动，通过 Nandc host 与 Nand flash device 通信，具体适用颗粒选型参考《RKNandFlashSupportList》，适用以下颗粒：

- SLC、MLC、TLC Nand flash

配置：

```
CONFIG_RKNAND
```

驱动文件:

```
./drivers/rknand/
```

### rkflash

rkflash 是针对选用小容量存储的设备所设计的存储驱动，其中 Nand flash 的支持是通过 Nandc host 与 Nand flash device 通信完成，SPI flash 的支持是通过 SFC host 与 SPI flash devices 通信完成，具体适用颗粒选型参考《RK SpiNor and SLC Nand SupportList》，适用以下颗粒：

- 128MB、256MB 和 512MB 的 SLC Nand flash
- 部分 SPI Nand flash
- 部分 SPI Nor flash 颗粒

配置：

```
CONFIG_RKFLASH

CONFIG_RKNANDC_NAND /* 小容量并口Nand flash */
CONFIG_RKSFC_NOR    /* SPI Nor flash */
CONFIG_RKSFC_NAND   /* SPI Nand flash */
```

驱动文件:

```
./drivers/rkflash/
```

注意：

1. SFC (serial flash controller) 是 Rockchip 为简便支持 spi flash 所设计的专用模块；
2. 由于 rknand 驱动与 rkflash 驱动 Nand 代码中 ftl 部分不兼容，所以
  - CONFIG\_RKNAND 与 CONFIG\_RKNANDC\_NAND 不能同时配置
  - CONFIG\_RKNAND 与 CONFIG\_RKSFC\_NAND 不能同时配置

### MMC & SD

MMC为多媒体卡，比如 eMMC；SD为是一种基于半导体快闪记忆器的新一代记忆设备。在rockchip平台，它们共用一个 dw\_mmc 控制器（除了rk3399，rk3399pro）。

配置：

```
CONFIG_MMC_DW=y
CONFIG_MMC_DW_ROCKCHIP=y
CONFIG_CMD_MMC=y
```

驱动文件:

```
./drivers/mmc/
```

### SLC Nand & SPI Nand & SPI Nor 开源方案

由于开源社区的不断完善及 UBI 文件系统的可行性，RK 也完善 flash 结合较多开源代码的方案，且开源方案默认选用 pre loader 为 SPL 的启动方案，所以大部分配置都是结合 SPL 相关配置来完成。

配置：

1. SPL、MTD、开源存储驱动及 MTD block 配置：参考 CH10-SPL 4.2 章节
2. 去除 rkflash 宏配置：

```
CONFIG_RKFLASH=n
```

驱动文件：

```
./drivers/mtd/nand/raw           //SLC Nand 主控驱动及协议层  
./drivers/mtd/nand/spi          //SPI Nand 协议层  
.drivers/spi/rockchip_sfc.c    //SPI Flash 主控驱动  
.drivers/mtd/spi                //SPI Nor 协议层
```

## 相关接口

存储驱动的访问接口都对挂到BLK层，所以无论何种存储都通过如下接口访问：

```
// 获取存储句柄  
struct blk_desc *rockchip_get_bootdev(void)  
  
// 访问接口  
unsigned long blk_dread(struct blk_desc *block_dev, lbaint_t start,  
                         lbaint_t blkcnt, void *buffer)  
unsigned long blk_dwrite(struct blk_desc *block_dev, lbaint_t start,  
                         lbaint_t blkcnt, const void *buffer)  
unsigned long blk_derase(struct blk_desc *block_dev, lbaint_t start,  
                         lbaint_t blkcnt)
```

## DTS 配置

eMMC配置：

```
// rkxxxx.dtsi配置  
emmc: dwmmc@ff390000 {  
    compatible = "rockchip,px30-dw-mshc", "rockchip,rk3288-dw-mshc";  
    reg = <0x0 0xff390000 0x0 0x4000>;           // 控制器寄存器base address及长度  
    max-frequency = <150000000>;                  // eMMC普通模式时钟为50MHz,当配置为eMMC  
                                                // HS200模式, 该max-frequency生效  
    clocks = <&cru HCLK_EMMC>, <&cru SCLK_EMMC>,  
             <&cru SCLK_EMMC_DRV>, <&cru SCLK_EMMC_SAMPLE>; // 控制器对应时钟编号  
    clock-names = "biu", "ciu", "ciu-drv", "ciu-sample"; // 控制器时钟名  
    fifo-depth = <0x100>;                           // fifo深度, 默认配置  
    interrupts = <GIC_SPI 53 IRQ_TYPE_LEVEL_HIGH>; // 中断配置  
    status = "disabled";  
};  
  
// rkxxxx-u-boot.dtsi  
&emmc {  
    u-boot,dm-pre-reloc;  
    status = "okay";  
}  
  
// rkxxxx.dts  
&emmc {
```

```

bus-width = <8>; // 设备总线位宽
cap-mmc-highspeed; // 标识此卡槽支持highspeed mmc
mmc-hs200-1_8v; // 支持HS200
supports-emmc; // 标识此插槽为eMMC功能，必须添加，否则无法初始化外设
disable-wp; // 对于无物理WP管脚，需要配置
non-removable; // 此项表示该插槽为不可移动设备。此项为必须添加项
num-slots = <1>; // 标识为第几插槽
status = "okay";
};

```

Nandc 配置：

```

&nandc {
    u-boot,dm-pre-reloc;
    status = "okay";
};

```

SFC 配置：

```

&sfc {
    u-boot,dm-pre-reloc;
    status = "okay";
    spi_nand: flash@0 {
        u-boot,dm-spl;
        compatible = "spi-nand";
        reg = <0>;
        spi-tx-bus-width = <1>;
        spi-rx-bus-width = <4>;
        spi-max-frequency = <96000000>;
    };
    spi_nor: flash@1 {
        u-boot,dm-spl;
        compatible = "jedec,spi-nor";
        reg = <0>;
        spi-tx-bus-width = <1>;
        spi-rx-bus-width = <4>;
        spi-max-frequency = <96000000>;
    };
};

```

注意：

- 考虑到软件的兼容性，u-boot 下仅支持 spi-tx-bus-width = <1> 的一线 SPI flash 传输；

## Uart

serial 使用 serial-uclass.c 框架和标准接口，目前主要是UART debug在使用。

配置：

```
// 使能配置
CONFIG_DEBUG_UART
CONFIG_SYS_NS16550

// 参数配置
CONFIG_DEBUG_UART_BASE
CONFIG_DEBUG_UART_CLOCK
CONFIG_BAUDRATE
```

框架代码：

```
./drivers/serial/serial-uclass.c
```

驱动代码：

```
./drivers/serial/ns16550.c
```

## 单独更换

单独更换 U-Boot 阶段的UART debug 流程如下 (uart2 为例) :

- `CONFIG_ROCKCHIP_PRELOADER_SERIAL` 禁用;
- `board_debug_uart_init()` 里配置 uart iomux (注意：某些平台有m0、m1...模式要配置) ;
- `board_debug_uart_init()` 里配置 uart clock , 保证时钟源是 24Mhz;
- `defconfig` 更新 `CONFIG_BAUDRATE` ;
- `defconfig` 更新 `CONFIG_DEBUG_UART_BASE` ;
- U-Boot uart 节点中增加 2 个必要属性并且使能:

```
&uart2 {
    u-boot,dm-pre-reloc;
    clock-frequency = <24000000>;
    status = "okay";
};
```

- U-Boot chosen 节点中指定 stdout-path:

```
chosen {
    stdout-path = &uart2;
};
```

## 全局更换

Pre-loader serial 是实现前级固件共享UART debug 配置的机制，这些固件包括：ddr、miniloader、bl31、op-tee、U-Boot。原理：由最早阶段的 ddr bin 配置好UART debug 并且通过 ATAGS 传参机制逐级传递下去，各级固件获取 UART debug 配置进行使用（不包括 kernel）。

用户可以通过修改 ddr bin里的串口配置实现UART debug的全局替换，步骤：

### DDR bin 配置

rkbin 仓库里提供了工具给用户配置不同的参数，包括串口更换：

```
tools/ddrbin_tool  
tools/ddrbin_param.txt  
tools/ddrbin_tool_user_guide.txt
```

## U-Boot 配置

1 使能配置：

```
CONFIG_ROCKCHIP_PRELOADER_SERIAL // 已经默认使能
```

2 rkxx-u-boot.dtsi 中把使用到的 uart 节点加上属性“u-boot,dm-pre-reloc;”；

3 aliases 建立 serial 别名，因为 U-Boot 是通过 aliaes 找到目标节点并初始化它的。

例如：./arch/arm/dts/rk1808-u-boot.dtsi 里为了方便，为所有 uart 都建立别名；

```
aliases {  
    mmc0 = &emmc;  
    mmc1 = &sddmmc;  
  
    // 必须创建别名  
    serial0 = &uart0;  
    serial1 = &uart1;  
    serial2 = &uart2;  
    serial3 = &uart3;  
    serial4 = &uart4;  
    serial5 = &uart5;  
    serial6 = &uart6;  
    serial7 = &uart7;  
};  
  
.....  
  
// 必须增加u-boot,dm-pre-reloc属性  
&uart0 {  
    u-boot,dm-pre-reloc;  
};  
&uart1 {  
    u-boot,dm-pre-reloc;  
};  
&uart2 {  
    u-boot,dm-pre-reloc;  
    clock-frequency = <24000000>;  
    status = "okay";  
};  
&uart3 {  
    u-boot,dm-pre-reloc;  
};  
&uart4 {  
    u-boot,dm-pre-reloc;  
};
```

## 关闭打印

```
CONFIG_DISABLE_CONSOLE=y
```

## 相关接口

```
// UART debug接口
void putc(const char c);
void puts(const char *s);
int printf(const char *fmt, ...);
void flushc(void);

// 跟外设通信功能的普通UART接口
int serial_dev_getc(struct udevice *dev);
int serial_dev_tstc(struct udevice *dev);
void serial_dev_putc(struct udevice *dev, char ch);
void serial_dev_puts(struct udevice *dev, const char *str);
void serial_dev_setbrg(struct udevice *dev, int baudrate);
void serial_dev_clear(struct udevice *dev);
```

## Key

### 框架支持

U-Boot 框架默认没有支持按键功能，RK 自己实现了一套按键框架。

实现规则：

- 所有按键都通过 kernel 和 U-Boot 的 DTS 指定，U-Boot 不使用 hard code 的方式定义任何按键；
- U-Boot 优先查找 kernel dts 中的按键，找不到再查找 U-Boot dts 中的按键。
- U-Boot dts里仅定义了烧写按键。
- 如果用户要更新烧写按键定义，请同时更新kernel和U-Boot的dts。

配置：

```
CONFIG_DM_KEY
CONFIG_RK8XX_PWRKEY
CONFIG_ADC_KEY
CONFIG_GPIO_KEY
CONFIG_RK_KEY
```

框架代码：

```
./include/dt-bindings/input/linux-event-codes.h
./drivers/input/key-uclass.c
./include/key.h
```

驱动代码：

```
./drivers/input/rk8xx_pwrkey.c      // 支持PMIC的pwrkey(RK805/RK809/RK816/RK817)
./drivers/input/rk_key.c            // 支持compatible = "rockchip,key"
./drivers/input/gpio_key.c         // 支持compatible = "gpio-keys"
./drivers/input/adc_key.c          // 支持compatible = "adc-keys"
```

pwrkey 仅以中断方式被识别，其余 gpio 按键以轮询方式被识别。

## 相关接口

接口：

```
int key_read(int code)
```

code 定义：

```
/include/dt-bindings/input/linux-event-codes.h
```

返回值：

```
enum key_state {
    KEY_PRESS_NONE,           // 非完整的短按（没有释放按键）或非完整长按（按下时间不够长）;
    KEY_PRESS_DOWN,           // 一次完整的短按（按下=>释放）;
    KEY_PRESS_LONG_DOWN,      // 一次完整的长按（可以不释放）;
    KEY_NOT_EXIST,            // 按键不存在
};
```

KEY\_PRESS\_LONG\_DOWN 默认时长 2000ms，目前只用于 U-Boot 充电的 pwrkey 长按事件。

```
#define KEY_LONG_DOWN_MS     2000
```

范例：

```
int ret;

ret = key_read(KEY_VOLUMEUP);
...
```

## Vendor Storage

Vendor Storage 用于存放 SN、MAC 等不需要加密的小数据。数据存放在 NVM (eMMC、NAND 等) 的保留分区中，有多个备份，更新数据时数据不丢失，可靠性高。

详细的资料参考文档《appnote rk vendor storage》。

### 原理概述

一共把 vendor 的存储块分成 4 个分区，vendor0、vendor1、vendor2、vendor3。每个 vendorX (X=0、1、2、3) 的 hdr 里都有一个单调递增的 version 字段用于表明 vendorX 被更新的时刻点。每次读操作只读取最新的 vendorX (即 version 最大)，写操作的时候会更新 version 并且把整个原有信息和新增信息搬到 vendorX+1 分区里。例如当前从 vendor2 读取到信息，经过修改后再回写，此时写入的是 vendor3。这样做只是为了起到一个简单的安全防护作用。

### 框架支持

U-Boot 框架没有支持 Vendor Storage 功能，Rockchip 自己实现了一套 Vendor Storage 驱动。

配置：

```
CONFIG_ROCKCHIP_VENDOR_PARTITION
```

驱动文件：

```
./arch/arm/mach-rockchip/vendor.c  
./arch/arm/include/asm/arch-rockchip/vendor.h
```

## 相关接口

```
int vendor_storage_read(u16 id, void *pbuff, u16 size)  
int vendor_storage_write(u16 id, void *pbuff, u16 size)
```

关于 id 的定义和使用, 请参考《appnote rk vendor storage》。

## 功能自测

U-Boot 串口命令行下使用"rktest vendor"命令可以进行 Vendor Storage 功能自测。

## OPTEE Client

U-Boot在ARM TrustZone里属于Non-Secure World, 需要借助OPTEE Client才能访问安全资源。

### 框架支持

U-Boot 框架默认没有支持OPTEE Client功能, RK 自己实现了一套。

配置:

```
// 总使能  
CONFIG_OPTEE_CLIENT  
  
// 旧平台使用, 如 RK312x、RK322x、RK3288、RK3228H、RK3368、RK3399  
CONFIG_OPTEE_V1  
// 新平台使用, 如 RK3326、RK3308  
CONFIG_OPTEE_V2  
// 当 eMMC 的 RPMB 不能用时必须开启此配置, 即启用security分区!  
CONFIG_OPTEE_ALWAYS_USE_SECURITY_PARTITION
```

框架和驱动:

```
lib/optee_clientApi/
```

## 固件说明

使用的 trust.img 必须启用 TA 功能, 否则无法跟OPTEE Client交互。

## 接口文档

Optee client 驱动在 lib/optee\_client 目录下, Optee Client Api 请参考  
《TEE\_Client\_API\_Specification-V1.0\_c.pdf》。

下载地址为: <https://globalplatform.org/specs-library/tee-client-api-specification/>

## 共享内存

U-Boot 与 Optee 通信时的数据需放在共享内存中。用户可通过 `TEEC_AllocateSharedMemory()` 申请共享内存, 但建议不超过 1M。若超过则建议分割数据进行多次传递, 使用完需调用 `TEEC_ReleaseSharedMemory()` 释放共享内存。

## 测试命令

作用：测试安全存储功能。U-Boot 命令行：

```
=> mmc testsecurestorage
```

该测试用例将循环测试安全存储读写功能，当硬件使用 emmc 时将测试 rpmb 与 security 分区两种安全存储方式；当硬件使用 nand 时只测试 security 分区安全存储。

## 常见错误打印

- 没有找到 emmc 或者 nand 设备。此时请检查 U-Boot 是否缺少配置，或者硬件是否损坏。

```
"TEEC: Could not find device"
```

- 没有找到 security 分区。当没有RPMB可用时，需要在 parameter.txt 中定义 security 分区。

```
"TEEC: Could not find security partition"
```

- 第一次使用 security 分区进行安全存储或 security 分区数据被非法篡改时会出现该打印。

```
"TEEC: verify [%d] fail, cleanning ...."
```

- 安全存储的空间不足。请检查存储的数据是否过大，或者之前存储过大量的数据但没有删除。

```
"TEEC: Not enough space available in secure storage!"
```

## DVFS

本章节的DVFS不同于kernel，是专门针对宽温芯片的动态调频调压机制。

### 宽温策略

U-Boot 框架没有支持 DVFS，为了支持某些芯片的宽温功能，RK 实现了一套 DVFS 宽温驱动根据芯片温度调整 cpu/dmc 频率-电压。但有别于内核 DVFS 驱动，这套宽温驱动仅仅在触发最高/低温度阈值时进行控制。

#### 宽温策略：

1. 宽温驱动用于调整 cpu/dmc 的频率-电压，控制策略可同时对 cpu 和 dmc 生效，也可只对其中一个生效，由 dts 配置决定；cpu 和 dmc 的控制策略是一样的；
2. 宽温驱动会解析 cpu/dmc 节点的 opp table、regulator、clock、thermal zone 的"trip-point-0"，获取频率-电压档位、最高/低温度阈值、允许的最高电压等信息；
3. 若 cpu/dmc 的 opp table 里指定了 rockchip,low-temp = <...> 或 rockchip,high-temp = <...>，又或者 cpu/dmc 引用了 thermal zone 的 trip 节点，那么 cpu/dmc 宽温控制策略就会生效；
4. 关键属性：
  - rockchip,low-temp：最低温度阈值，下述用 TEMP\_min 表示；
  - rockchip,high-temp 和 thermal zone：最高温度阈值，下述用 TEMP\_max 表示（如果二者都有效，策略上都会拿当前温度进与之比较）；
  - rockchip,max-volt：允许设置的最高电压值，下述用 V\_max 表示；
5. 阈值触发的处理：
  - 如果温度高于 TEMP\_max，把频率和电压都降到最低档位；

- 如果温度低于 TEMP\_min， 默认抬压 50mv。若抬压 50mv 会导致电压超过 V\_max，则电压设定为 V\_max，同时把频率降低 2 档；

6. 目前宽温策略应用在 2 个时刻点：

- regulator 和 clk 框架初始化完成后，宽温驱动进行初始化并且执行一次宽温策略，具体位置在 board.c 文件的 board\_init()中调用；
- preboot 阶段（即加载固件之前）再执行一次宽温策略：如果 dts 节点中指定了"repeat"等相关属性（见下文），当执行完本次宽温策略后芯片温度依然不在温度阈值范围内，那就停止系统启动并且不断执行宽温策略，直到芯片温度回归到阈值范围内才继续启动系统。如果没有"repeat"等相关属性，则执行完本次宽温策略后就直接启动系统，目前一般不需要 repeat 属性。

## 框架支持

框架代码：

```
./drivers/power/dvfs/dvfs-uclass.c
./include/dvfs.h
./cmd/dvfs.c
```

驱动代码：

```
./drivers/power/dvfs/rockchip_wtemp_dvfs.c
```

## 相关接口

```
// 执行一次dvfs策略
int dvfs_apply(struct udevice *dev);

// 如果存在repeat属性，当温度不在阈值范围内时循环执行dvfs策略
int dvfs_repeat_apply(struct udevice *dev);
```

## 启用宽温

1. 配置使能：

```
CONFIG_DM_DVFS=y
CONFIG_ROCKCHIP_WTEMP_DVFS=y

CONFIG_DM_THERMAL=y
CONFIG_ROCKCHIP_THERMAL=y
CONFIG_USING_KERNEL_DTB=y
```

2. 指定 CONFIG\_PREBOOT：

```
#ifdef CONFIG_DM_DVFS
#define CONFIG_PREBOOT "dvfs repeat"
#else
#define CONFIG_PREBOOT
#endif
```

3. kernel dts 配置宽温节点

```
uboot-wide-temperature {
```

```

compatible = "rockchip,uboot-wide-temperature";

// 可选项。表示是否在U-Boot阶段触发cpu的最高/低温度阈值时让宽温驱动停止启动系统,
// 且不断执行宽温处理策略, 直到芯片温度回归到阈值范围内才继续启动系统。
cpu,low-temp-repeat;
cpu,high-temp-repeat;

// 可选项。表示是否在U-Boot阶段触发dmc的最高/低温度阈值时让宽温驱动停止启动系统,
// 且不断执行宽温处理策略, 直到芯片温度回归到阈值范围内才继续启动系统。
dmc,low-temp-repeat;
dmc,high-temp-repeat;

status = "okay";
};

```

一般情况下不需要配置上述的 repeat 相关属性。

## 宽温结果

当 cpu 温控启用时有如下打印:

```

// <NULL>表明没有指定低温阈值
DVFS: cpu: low=<NULL>'c, high=95'c, vmax=1350000uv, tz_temp=88.0'c, h_repeat=0,
l_repeat=0

```

当 cpu 温控触发高温阈值时会有调整信息:

```

DVFS: 90.352'c
DVFS: cpu(high): 600000000->408000000 Hz, 1050000->950000 uv

```

当 cpu 温控触发低温阈值时会有调整信息:

```

DVFS: 10.352'c
DVFS: cpu(low): 600000000->600000000 Hz, 1050000->1100000 uv

```

同理, 当 dmc 触发高低温阈值时, 也会有上述信息打印, 信息前缀为"dmc":

```

DVFS: dmc: .....
DVFS: dmc(high): .....
DVFS: dmc(low): .....

```

## AMP

### 框架支持

U-Boot 框架默认没有 AMP(Asymmetric Multi-Processing) 支持, RK 自己实现了一套 AMP 机制: 不同的CPU运行不同的固件。

AMP功能需要trust配合, 目前仅部分平台完整支持。AMP固件采用FIT格式, 默认支持sha256固件完整性校验。

配置:

```
CONFIG_AMP  
CONFIG_ROCKCHIP_AMP
```

框架代码:

```
./drivers/cpu/amp-uclass.c  
./drivers/cpu/rockchip_amp.c
```

its 模版:

```
./drivers/cpu/amp.its
```

打包工具:

```
./tools/mkimage // 完整编译一次U-Boot后会自动生成
```

代码提交点至少:

```
commit 31f8f6ebae796097f7f10c67dff58fd907371c63  
Author: Joseph Chen <chenjh@rock-chips.com>  
Date:   Wed Feb 24 14:34:25 2021 +0800  
  
cpu: amp: support brought up rockchip fit image  
  
. ./tools/mkimage -f amp.its -E -p 0xe00 amp.img  
  
Signed-off-by: Joseph Chen <chenjh@rock-chips.com>  
Change-Id: Icdaf370900472622cf31df402aff84ecc6821fe4
```

## 功能启用

1. 制作amp.img需要一份its文件, 请基于 `drivers/cpu/amp.its` 修改:

```
/dts-v1/;  
/  
  description = "FIT source file for rockchip AMP";  
  #address-cells = <1>;  
  
  images {  
    amp0 {  
      description = "bare-metal-core0";           // 必选项: 描述信息  
      data       = /incbin/("../hal0.bin");        // 必选项: amp0固件  
      type       = "firmware";  
      compression = "none";  
      arch       = "arm";             // 必选项: "arm64": 64位, "arm": 32位  
      cpu        = <0x000>;          // 必选项: cpu硬件id(mpdr)  
      thumb     = <0>;            // 必选项: 0: arm or thumb2; 1: 纯  
    thumb  
      hyp       = <0>;            // 必选项: 0: el1/svc; 1: el2/hyp  
      load     = <0xa00000>;        // 必选项: 固件加载和运行地址  
      udelay   = <500000>;        // 可选项: 启动完当前CPU后做延时后再启动下一个CPU  
    linux-os;                  // 特殊可选项!  
    hash {
```

```

        algo = "sha256";
    };
};

amp1 {
    .....
};

amp2 {
    .....
};

amp3 {
    .....
};

configurations {
    default = "conf";
    conf {
        description = "Rockchip AMP images";
        rollback-index = <0x0>;
        // 指定需要被加载的固件和顺序。
        loadables = "amp0", "amp1", "amp2", "amp3";
        signature {
            algo = "sha256,rsa2048";
            padding = "pss";
            key-name-hint = "dev";
            sign-images = "loadables";
        };
    };
};

```

说明:

- 主CPU: 我们称当前跑U-Boot、负责启动其它核的CPU为“主CPU”，主CPU的状态是最后切换的。
- data: 固件路径。该路径是基于amp.its的相对路径。
- arch: CPU 32/64模式。ARMv7只能是"arm"; ARMv8可指定"arm64"或"arm"分别表示AArch64位或AArch32。
- cpu: cpu硬件ID，即mpidr(Multiprocessor Affinity Register)，取(低)32位即可。例如：

```

cpus {
    #address-cells = <2>;
    #size-cells = <0>;

    cpu0: cpu@0 {
        device_type = "cpu";
        compatible = "arm,cortex-a55";
        reg = <0x0 0x0>;      // mpidr
        .....
    };

    cpu1: cpu@100 {
        device_type = "cpu";
        compatible = "arm,cortex-a55";
    };
}

```

```
    reg = <0x0 0x100>; // mpidr
    .....
};

.....
};
```

- thumb: CPU指令模式。如果是纯thumb则指定为1，否则为0。
- hyp: CPU虚拟机模式。
- load: 固件加载和运行地址。
- udelay: 加载完成后的延时，单位us。启动完当前CPU后做相应延时后再启动下一个CPU。
- loadables: AMP固件加载顺序。主CPU是最后被加载的，不受此处指定的顺序影响。
- linux-os: 主CPU跑Linux。可选，只能在主CPU的节点下添加。若该属性存在：
  - 主CPU启动完其它核之后继续跑Linux(即按照原本的流程、固件格式被U-Boot加载启动)。
  - 主CPU节点依然要指定在"loadables"里。
  - 主CPU节点内的"data", "load"字段没有实际作用，但依然要保留，否则mkimage打包时会报错。

建议创建一个空文件指定data字段，例如：touch hal0.bin。

- 特别说明：AMP OS需要各自处理好内存管理，避免内存冲突。

## 2. 固件打包：

```
// 0xe00为固件头大小，不建议改变
./tools/mkimage -f ./drivers/cpu/amp.its -E -p 0xe00 amp.img
```

需要完整编译一次U-Boot才会自动生成mkimage工具。

## 3. 分区表增加amp分区

在 parameter.txt 分区表文件中增加 "amp" 分区，然后烧写amp.img。

U-Boot是直接加载整个amp分区的内容到内存，建议amp分区大小按实际需要配置。

## 4. Bring up

U-Boot 框架会在合适的时机自动发起所有 AMP 的 bring up。例如：

```
.....
## Loading loadables from FIT Image at 3b3f7c80 ...
Trying 'amp0' loadables subimage
  Description: bare-mental-core0
  Type:        Firmware
  Compression: uncompressed
  Data Start:  0x3b3f8a80
  Data Size:   1016690 Bytes = 992.9 kib
  Architecture: ARM
  Load Address: 0x00a00000
  Hash algo:   sha256
  Hash value:
3376b6ee20dd52a06652b2e15c688206fbec021140d37c15f1020f9ddb20b76d
  Verifying Hash Integrity ... sha256+ OK
  Loading loadables from 0x3b3f8a80 to 0x00a00000
## Loading loadables from FIT Image at 3b3f7c80 ...
Trying 'amp1' loadables subimage
```

```

Description: bare-mental-core1
Type: Firmware
Compression: uncompressed
Data Start: 0x3b4f0e80
Data Size: 1016690 Bytes = 992.9 KiB
Architecture: ARM
Load Address: 0x00b00000
Hash algo: sha256
Hash value:
f3a08027d58113e9611434ac4f12dd68a40c80df2a6603c10138a2519ccc5f5d
    Verifying Hash Integrity ... sha256+ OK
    Loading loadables from 0x3b4f0e80 to 0x00b00000
## Loading loadables from FIT Image at 3b3f7c80 ...
    Trying 'amp2' loadables subimage
        Description: bare-mental-core2
        Type: Firmware
        Compression: uncompressed
        Data Start: 0x3b5e9280
        Data Size: 1016690 Bytes = 992.9 KiB
        Architecture: ARM
        Load Address: 0x00c00000
        Hash algo: sha256
        Hash value:
602eda361c86fb8ad19cd7db813c67bb962d12d7d7355cac015bfff018fef788
    Verifying Hash Integrity ... sha256+ OK
    Loading loadables from 0x3b5e9280 to 0x00c00000
## Loading loadables from FIT Image at 3b3f7c80 ...
    Trying 'amp3' loadables subimage
        Description: bare-mental-core3
        Type: Firmware
        Compression: uncompressed
        Data Start: 0x3b6e1680
        Data Size: 1016690 Bytes = 992.9 KiB
        Architecture: AArch64
        Load Address: 0x00d00000
        Hash algo: sha256
        Hash value:
1491d53be18e58165f750813291f2ce26e736bda0c3a9edb257be4614b1e00f2
    Verifying Hash Integrity ... sha256+ OK
    Loading loadables from 0x3b6e1680 to 0x00d00000
// 成功打印(如果失败, 会有相应错误打印)
AMP: Brought up cpu[100] with state 0x10, entry 0x00b00000 ...OK
AMP: Brought up cpu[200] with state 0x10, entry 0x00c00000 ...OK
AMP: Brought up cpu[300] with state 0x10, entry 0x00d00000 ...OK
AMP: Brought up cpu[0, self] with state 0x10, entry 0x00a00000 ...OK
.....

```

## IO-Domain

### 框架支持

U-Boot 框架默认没有对 io-domain 的支持, RK 自己实现了一套。

配置:

```

CONFIG_IO_DOMAIN
CONFIG_ROCKCHIP_IO_DOMAIN

```

框架代码：

```
./drivers/power/io-domain/io-domain-uclass.c
```

驱动代码：

```
./drivers/power/io-domain/rockchip-io-domain.c
```

## 相关接口

```
void io_domain_init(void)
```

用户不需要主动调用 `io_domain_init()`，只需要开启上述配置即可，U-Boot框架会自动初始化。

## Watchdog

### 框架支持

watchdog 驱动使用 `wdt-uclass.c` 框架和标准接口。

配置：

```
CONFIG_WDT
CONFIG_ROCKCHIP_WATCHDOG
```

框架代码：

```
./drivers/watchdog/wdt-uclass.c
```

驱动代码：

```
./drivers/watchdog/rockchip_wdt.c
```

## 相关接口

```
// 设置喂狗超时时间且启动wdt (@flags默认填0)
int wdt_start(struct udevice *dev, u64 timeout_ms, ulong flags);
// 关闭wdt
int wdt_stop(struct udevice *dev);
// 喂狗
int wdt_reset(struct udevice *dev);
// 忽略，目前未做底层驱动实现
int wdt_expire_now(struct udevice *dev, ulong flags)
```

目前 U-Boot 的默认流程里不启用、也不使用 wdt 功能，用户可根据自己的产品需求进行启用。

## Crypto

Crypto 模块主要用于实现硬件级别的加密和哈希算法，目前有两个IP版本：

- Crypto V1：rk3399/rk3368/rk3328/rk3229/rk3288/rk3128；
- Crypto V2：rk3326/px30/rk3308/rk1808；

## 框架支持

U-Boot 默认没有crypto框架支持， RK 自己实现了一套。

配置：

```
CONFIG_DM_CRYPTO  
CONFIG_ROCKCHIP_CRYPTO_V1  
CONFIG_ROCKCHIP_CRYPTO_V2
```

框架代码：

```
./drivers/crypto/crypto-uclass.c  
.cmd/crypto.c
```

驱动代码：

```
// crypto v1:  
./drivers/crypto/rockchip/crypto_v1.c  
  
// crypto v2:  
./drivers/crypto/rockchip/crypto_v2.c  
./drivers/crypto/rockchip/crypto_v2_pka.c  
./drivers/crypto/rockchip/crypto_v2_util.c
```

## 相关接口

```
// 获取crypto:  
struct udevice *crypto_get_device(u32 capability);  
// SHA接口:  
int crypto_sha_init(struct udevice *dev, sha_context *ctx);  
int crypto_sha_update(struct udevice *dev, u32 *input, u32 len);  
int crypto_sha_final(struct udevice *dev, sha_context *ctx, u8 *output);  
int crypto_sha_csum(struct udevice *dev, sha_context *ctx,  
                    char *input, u32 input_len, u8 *output);  
// RSA接口:  
int crypto_rsa_verify(struct udevice *dev, rsa_key *ctx, u8 *sign, u8 *output);
```

- 接口使用请参考： `./cmd/crypto.c`；
- v1 和 v2 的 SHA 使用不同： v1 要求 `crypto_sha_init()` 时必须先把要计算的数据总长度赋给 `ctx->length`， v2 不需要；

## DTS 配置

crypto 节点必须定义在 U-Boot dts，主要原因：

- 各平台旧 SDK 的内核 dts 没有 crypto 节点，因此需要考虑对旧 SDK 的兼容；
- U-Boot 的 secure boot 会用到 crypto，因此由 U-Boot 自己控制 crypto 更为安全合理；

1. crypto v1 配置 (RK3399 为例)：

```
crypto: crypto@ff8b0000 {
    u-boot,dm-pre-reloc;

    compatible = "rockchip,rk3399-crypto";
    reg = <0x0 0xff8b0000 0x0 0x10000>;
    clock-names = "sclk_crypto0", "sclk_crypto1";
    clocks = <&cru SCLK_CRYPTO0>, <&cru SCLK_CRYPTO1>; // 不需要指定频率, 默认100M
    status = "disabled";
};
```

2. crypto v2 配置 (px30 为例) :

```
crypto: crypto@ff0b0000 {
    u-boot,dm-pre-reloc;

    compatible = "rockchip,px30-crypto";
    reg = <0x0 0xff0b0000 0x0 0x4000>;
    clock-names = "sclk_crypto", "apkcclk_crypto";
    clocks = <&cru SCLK_CRYPTO>, <&cru SCLK_CRYPTO_APK>;
    clock-frequency = <200000000>, <300000000>; // 一般需要指定频率
    status = "disabled";
};
```

- crypto v1 和 v2 的 dts 配置差异在于 clk 频率指定。

## Reset

### 框架支持

reset 驱动使用 reset-uclass.c 框架和标准接口。RK 平台上reset 的实质是进行 CRU 软复位。

配置:

```
CONFIG_DM_RESET
CONFIG_RESET_ROCKCHIP
```

框架代码:

```
./drivers/reset/reset-uclass.c
```

驱动代码:

```
./drivers/reset/reset-rockchip.c
```

## 相关接口

```

// 获取reset句柄
int reset_get_by_index(struct udevice *dev, int index, struct reset_ctl
*reset_ctl);
int reset_get_by_name(struct udevice *dev, const char *name,
                      struct reset_ctl *reset_ctl);
// 释放reset
int reset_free(struct reset_ctl *reset_ctl);
// 请求reset
int reset_request(struct reset_ctl *reset_ctl);
// 触发reset、释放reset
int reset_assert(struct reset_ctl *reset_ctl);
int reset_deassert(struct reset_ctl *reset_ctl);

```

范例：

```

struct reset_ctl reset_ctl;

ret = reset_get_by_name(dev, "mac-phy", &reset_ctl);
if (ret) {
    debug("reset_get_by_name() failed: %d\n", ret);
    return ret;
}

ret = reset_request(&reset_ctl);
if (ret)
    return ret;

ret = reset_assert(&reset_ctl);
if (ret)
    return ret;

.....
ret = reset_deassert(&reset_ctl);
if (ret)
    return ret;

.....
ret = reset_free(&reset_ctl);
if (ret)
    return ret;

```

## DTS 配置

U-Boot 默认启用reset功能，用户只需在外设节点里指定要操作的 reset 对象即可：

```

// 格式:
reset-names = <name-string-list>
resets = <cru-phandle-list>

```

例如 gmac2phy:

```
gmac2phy: ethernet@ff550000 {
    compatible = "rockchip,rk3328-gmac";
    .....
    // 指定reset属性
    reset-names = "stmmaceth", "mac-phy";
    resets = <&cru SRST_GMAC2PHY_A>, <&cru SRST_MACPHY>;
};
```

## Led

### 框架支持

Led 驱动使用 led-uclass.c 框架和标准接口。

配置:

```
CONFIG_LED_GPIO
```

框架代码:

```
drivers/led/led-uclass // 默认编译
```

驱动代码:

```
drivers/led/led_gpio.c // 支持 compatible = "gpio-leds"
```

### 相关接口

```
// 获取led device
int led_get_by_label(const char *label, struct udevice **devp);
// 设置/获取led状态
int led_set_state(struct udevice *dev, enum led_state_t state);
enum led_state_t led_get_state(struct udevice *dev);
// 忽略, 目前未做底层驱动实现
int led_set_period(struct udevice *dev, int period_ms);
```

### DTS 节点

U-Boot 的 led\_gpio.c 功能相对简单, 只解析 led 节点下的 3 个属性:

- gpios: led 控制引脚和有效状态;
- label: led 名字;
- default-state: 默认状态, 驱动 probe 时会被设置;

```
leds {
    compatible = "gpio-leds";
    status = "okay";

    blue-led {
        gpios = <&gpio2 RK_PA1 GPIO_ACTIVE_LOW>;
        label = "battery_full";
        default-state = "off";
    };
};
```

```
green-led {
    gpios = <&gpio2 RK_PA0 GPIO_ACTIVE_LOW>;
    label = "greenled";
    default-state = "off";
};

.....
};
```

## Efuse/Otp

### 框架支持

efuse/otp 驱动使用 misc-uclass.c 框架和标准接口。通常情况，efuse/otp 一般会有 secure 和 non-secure 之分。U-Boot 提供 non-secure 的访问，U-Boot SPL 提供 secure otp 某些区域的访问。

non-secure 配置：

```
CONFIG_MISC
CONFIG_ROCKCHIP_EFUSE
CONFIG_ROCKCHIP OTP
```

secure 配置：

```
CONFIG_SPL_MISC=y
CONFIG_SPL_ROCKCHIP_SECURE OTP=y
```

框架代码：

```
./drivers/misc/misc-uclass.c
```

驱动代码：

```
// non-secure:
./drivers/misc/rockchip-efuse.c
./drivers/misc/rockchip-otp.c
// secure:
./drivers/misc/rockchip-secure-otp.S
```

## 相关接口

```
// non-secure:
int misc_read(struct udevice *dev, int offset, void *buf, int size)
// secure:
int misc_read(struct udevice *dev, int offset, void *buf, int size)
int misc_write(struct udevice *dev, int offset, void *buf, int size)
```

## DTS 配置

以 rk3308 为例：

non-secure:

```
otp: otp@ff210000 {
    compatible = "rockchip,rk3308-otp";
    reg = <0x0 0xff210000 0x0 0x4000>;
};
```

secure:

```
secure_otp: secure_otp@0xff2a8000 {
    compatible = "rockchip,rk3308-secure-otp";
    reg = <0x0 0xff2a8000 0x0 0x4000>;
    secure_conf = <0xff2b0004>;
    mask_addr = <0xff540000>;
};
```

## 调用示例

non-secure 示例:

```
char data[10] = {0};
struct udevice *dev;

/* retrieve the device */
ret = uclass_get_device_by_driver(UCLASS_MISC,
                                  DM_GET_DRIVER(rockchip_otp), &dev);
if (ret) {
    printf("no misc-device found\n");
    return 0;
}

misc_read(dev, 0x10, &data, 10);
```

secure 示例:

```
char data[10] = {0};
struct udevice *dev;
int i;

/* retrieve the device */
ret = uclass_get_device_by_driver(UCLASS_MISC,
                                  DM_GET_DRIVER(rockchip_secure_otp), &dev);
if (ret) {
    printf("no misc-device found\n");
    return 0;
}

for (i = 0; i < 10; i++)
    data[i] = i;

misc_write(dev, 0x10, &data, 10);
memset(data, 0, 10);
misc_read(dev, 0x10, &data, 10);
```

## Secure-Otp

RK 对 secure otp 只开放部分区域读写，区域如下：

```
0x0;           // Rockchip 定义为 secure boot enable flag  
0x10-0x2f;   // Rockchip 定义为 RSA Public key hash  
0x80-0x187;  // Rockchip 定义为 reserved for OEM
```

## MTD

MTD (Memory Technology Device) 即内存技术设备，支持 nand、spi nand、spi nor。RK 设计了 MTD block 层用于支持 MTD 设备的读写。

### 框架支持

U-Boot 配置：

```
// MTD 驱动  
CONFIG_MTD=y  
CONFIG_CMD_MTDPARTS=y  
CONFIG_MTD_DEVICE=y  
  
// MTD block 设备驱动  
CONFIG_CMD_MTD_BLK=y  
CONFIG_MTD_BLK=y  
  
// 其他 nand 设备驱动 config  
.....
```

SPL 配置：

```
CONFIG_MTD=y  
CONFIG_CMD_MTDPARTS=y  
CONFIG_MTD_DEVICE=y  
CONFIG_SPL_MTD_SUPPORT=y  
  
// 其他 nand 设备驱动 config  
.....
```

框架代码：

```
drivers/mtd/mtd-uclass.c  
drivers/mtd/mtdcore.c  
drivers/mtd/mtd_uboot.c  
drivers/mtd/mtd_blk.c
```

驱动为各个控制器驱动，把读写等接口挂接到 MTD 层。

### 相关接口

```
unsigned long blk_dread(struct blk_desc *block_dev, lbaint_t start,  
                         lbaint_t blkcnt, void *buffer)
```

## 以太网

### 框架支持

框架代码：

```
./net/*
./drivers/net/*
./drivers/net/phy/*
```

驱动代码：

```
./drivers/net/designware.c
./drivers/net/dwc_eth_qos.c
./drivers/net/gmac_rockchip.c
```

menuconfig 配置：

- 驱动配置

Rockchip 以太网驱动有两套驱动，如果对驱动的选择有疑问，请参考我们对应的 sdk config 配置。

```
// designware:
CONFIG_DM_ETH=y
CONFIG_ETH_DESIGNWARE=y
CONFIG_GMAC_ROCKCHIP=y
```

```
// dwc_eth_qos:
CONFIG_DM_ETH=y
CONFIG_DM_ETH_PHY=y
CONFIG_DWC_ETH_QOS=y
CONFIG_GMAC_ROCKCHIP=y
```

另外 dwc\_eth\_qos 驱动需要配置 nocache memory，参考 RV1126：

```
diff --git a/include/configs/rv1126_common.h b/include/configs/rv1126_common.h
index 933917f3f0..9d70795fb8 100644
--- a/include/configs/rv1126_common.h
+++ b/include/configs/rv1126_common.h
@@ -50,6 +50,7 @@
#define CONFIG_SYS_SDRAM_BASE          0
#define SDRAM_MAX_SIZE                0xfd000000

+#define CONFIG_SYS_NOCACHED_MEMORY    (1 << 20)      /* 1 MiB */
#ifndef CONFIG_SPL_BUILD
```

- cmd 配置

需要的功能手动配置上。

```
Command line interface ---> Network commands --->

[*] bootp, tftpboot
[ ] tftp put
[ ] tftp download and bootm
[ ] tftp download and flash
[ ] tftpsrv
[ ] rarpboot
-*-
-dhcp
-*-
pxe
```

```
[ ] nfs
-*= mii
-*= ping
[ ] cdp
[ ] sntp
[ ] dns
[ ] linklocal
[ ] ethsw
```

## 相关接口

- 数据结构初始化接口

```
void net_init(void);
int eth_register(struct eth_device *dev);
int phy_init(void);
```

- 设备注册接口

```
int eth_register(struct eth_device *dev);
int phy_register(struct phy_driver *drv);
```

- 网络数据读写 和 phy 读写

U-Boot的数据收发需要主动调用，没有采用中断或轮询方式，具体实现可参照 NetLoop().

```
int eth_send(void *packet, int length);
int eth_rx(void);

int phy_read(struct phy_device *phydev, int devad, int regnum);
int phy_write(struct phy_device *phydev, int devad, int regnum, u16 val);
```

## DTS 配置

DTS 节点与 kernel 一样，需要关注的是以下板级相关的属性配置：

- phy 接口配置(phy-mode)
- phy 复位脚与复位时间(snps,reset-gpio) (snps,reset-delays-us)
- 针对主控的时钟输出方向(clock\_in\_out)
- 时钟源选择与频率设定(assigned-clock-parents) (assigned-clock-rates)
- RGMII Delayline， RGMII 接口需要(tx\_delay) (rx\_delay)

```
&gmac {
    phy-mode = "rgmii";
    clock_in_out = "input";

    snps,reset-gpio = <&gpio3 RK_PA0 GPIO_ACTIVE_LOW>;
    snps,reset-active-low;
    /* Reset time is 20ms, 100ms for rtl8211f */
    snps,reset-delays-us = <0 20000 100000>

    assigned-clocks = <&cru CLK_GMAC_SRC>, <&cru CLK_GMAC_TX_RX>, <&cru
    CLK_GMAC_ETHERNET_OUT>;
    assigned-clock-parents = <&cru CLK_GMAC_SRC_M1>, <&cru RGMIIMODE_CLK>;
    assigned-clock-rates = <125000000>, <0>, <25000000>;
```

```
    pinctrl-names = "default";
    pinctrl-0 = <&rgmiiim1_pins &clk_out_ethernetm1_pins>;
    tx_delay = <0x2a>;
    rx_delay = <0x1a>;
    phy-handle = <&phy>;
    status = "okay";
};
```

使用示例

常用的网络命令：

- DHCP

Usage:  
dhcpc [loadAddress] [[hostIPAddr:]bootfilename]

使用这条命令，就不需要设置 serverip，ipaddr，以及 gateway 了。

当 dhcp 成功从 dhcp 服务器上面拿到 ip 地址后，其就会从 hostIPaddr 地址，以 tftp 的方式获取文件。

100M 环境：

- PING

```
=> ping 192.168.0.1
ethernet@ffc40000 Waiting for PHY auto negotiation to complete. done
using ethernet@ffc40000 device
host 192.168.0.1 is alive
```

- TFTP

## 1000M 环境：

**Usage:**  
tftp [loadAddress] [[hostIPAddr:]bootfilename]

也可以自己设置 IP 地址：

## 网络故障排查

## 1. 网络环境，常见的有下面几个方向

- 电脑端防火墙是否没关；

- 如果是跨网段的，确认网关是否设置；
- TFTP 服务器配置是否正确；
- 某些路由器的 TFTP 功能是否被关闭。

## 2. 代码问题，一般来说，主要确认以下3个地方：

- pinctrl 配置是否正确。检查相关 pin 的 iomux 和驱动强度是否正确，也可以 dump 相关寄存器与内核比较是否一致，大部分情况下，我们是先调通了内核的网络再调 U-Boot 的。
- clock 配置是否正确。时钟配置的检查相对麻烦一些，主要检查分频比，时钟源，以及时钟方向，大部分寄存器在 CRU 里面，也存在有的芯片部分寄存器在 GRF，同样可以 dump 相关寄存器与内核的比较是否一致。
- PHY 复位脚。主要检测复位脚配置是否正确，以及复位波形是否符合 PHY 的要求。

## PCIe

### 框架支持

框架代码：

```
./drivers/pci/*
./drivers/phy/*
```

驱动代码：

```
drivers/pci/pcie_dw_rockchip.c
drivers/phy/phy-rockchip-snps-pcie3.c
```

menuconfig 配置：

- 驱动配置

Rockchip PCIe驱动目前支持的平台请查看pcie\_dw\_rockchip.c文件中的compatible属性，如果对驱动的选择有疑问，请参考我们对应的 sdk config 配置。

```
CONFIG_DM_REGULATOR_GPIO=y
CONFIG_PCI=y
CONFIG_DM_PCI=y
CONFIG_DM_PCI_COMPAT=y
CONFIG_PCI_PNP=y
CONFIG_PCIE_DW_ROCKCHIP=y
CONFIG_PHY_ROCKCHIP_SNPS_PCIE3=y
CONFIG_PHY=y
CONFIG_CMD_PCI=y
//添加NVMe支持
CONFIG_NVME=y
CONFIG_CMD_NVME=y
//添加PCIe转USB支持
CONFIG_USB_XHCI_PCI=y
```

## DTS 配置

直接复用内核DTB节点，相关文档请参考内核PCIe配置说明。

## 使用示例

常用的命令：

- pci

```
// 启动PCIe扫描， 总线枚举识别到一个Gen3的2个Lane的设备
=> pci enum
PCIe Linking... LTSSM is 0x1
PCIe Link up, LTSSM is 0x230011
PCIE-0: Link up (Gen3-x2, Bus0)

// 扫描bus0， 默认是bridge设备
=> pci scan
Scanning PCI devices on bus 0
BusDevFun VendorId DeviceId Device Class Sub-Class
-----
00.00.00 0x1d87 0x3566 Bridge device 0x04

// 扫描bus1， 目前看到接的是一个NVMe
=> pci 1
Scanning PCI devices on bus 1
BusDevFun VendorId DeviceId Device Class Sub-Class
-----
01.00.00 0x144d 0xa808 Mass storage controller 0x08

// 发起nvme 扫描
=> nvme scan

// 罗列nvme设备详细信息
=> nvme details
    Blk device 0: Optional Admin Command Support:
        Namespace Management/Attachment: no
        Firmware Commit/Image download: yes
        Format NVM: yes
        Security Send/Receive: no
    Blk device 0: Optional NVM Command Support:
        Reservation: yes
        Save>Select field in the Set/Get features: yes
        Write Zeroes: yes
        Dataset Management: yes
        Write Uncorrectable: yes
    Blk device 0: Format NVM Attributes:
        Support Cryptographic Erase: No
        Support erase a particular namespace: Yes
        Support format a particular namespace: Yes
    Blk device 0: LBA Format Support:
    Blk device 0: End-to-End DataProtect Capabilities:
        As last eight bytes: No
        As first eight bytes: No
        Support Type3: No
        Support Type2: No
        Support Type1: No
    Blk device 0: Metadata capabilities:
        As part of a separate buffer: No
        As part of an extended data LBA: No

// 看到一个256GB的NVMe， 如果看不到容量，需要拔出设备确保完全掉电，重来。
=> nvme info
Device 0: Vendor: 0x144d Rev: EXD7201Q Prod: S444NA0M384608
```

```

Type: Hard Disk
Capacity: 244198.3 MB = 238.4 GB (500118192 x 512)

// 选择ID为0的nvme设备
=> nvme device 0

Device 0: Vendor: 0x144d Rev: EXD7201Q Prod: S444NA0M384608
    Type: Hard Disk
    Capacity: 244198.3 MB = 238.4 GB (500118192 x 512)
... is now current device

// 将0x40000000内存设置位0x55aa55aa
=> md.1 0x40000000 1
    40000000: d08ec033           3...
=> mw.l 0x40000000 0x55aa55aa
=> md.1 0x40000000 1
    40000000: 55aa55aa          .U.U

// 从0x40000000内存开始取1个block数据，写入NVME的LBA 0地址
=> nvme write 0x40000000 0x0 0x1

    nvme write: device 0 block # 0, count 1 ... 1 blocks written: OK

// 检查下0x44000000内存，确认原始数据
=> md.1 0x44000000 1
    44000000: ffffffff         .....

// 从NVMe的LBA 0地址，读取1个block数据，写入内存0x44000000
=> nvme read 0x44000000 0x0 0x1

    nvme read: device 0 block # 0, count 1 ... 1 blocks read: OK

// 确认0x44000000内存数据是从NVMe读回来的
=> md.1 0x44000000 1
    44000000: 55aa55aa          .U.U

```

## Chapter-6 进阶原理

### kernel-DTB

#### 设计背景

U-Boot 的原生架构要求一块板子必须对应一份 U-Boot dts，并且U-Boot dts生成的dtb是打包到U-Boot自己的镜像中的。这样就会出现各SoC平台上，N块板子需要N份U-Boot镜像。

我们不难发现，其实一个SoC平台不同的板子之间主要是外设的差异，SoC核心部分是一致的。RK平台为了实现一个SoC平台仅需要一份U-Boot镜像，因此增加了 kernel DTB 机制。本质就是在较早的阶段切到kernel DTB，用它的配置信息初始化外设。

所以 RK 平台通过支持 kernel DTB 可以达到兼容板子差异，如：display、pmic/regulator、pinctrl、clk 等。

kernel DTB 的启用需要依赖 OF\_LIVE (live device tree，简称：live-dt) 。

```

config USING_KERNEL_DTB
    bool "Using dtb from Kernel/resource for U-Boot"
    depends on RKIMG_BOOTLOADER && OF_LIVE
    default y
    help
        This enable support to read dtb from resource and use it for U-Boot,
        the uart and emmc will still using U-Boot dtb, but other devices like
        regulator/pmic, display, usb will use dts node from kernel.

```

## Live device tree

### 背景和原理:

引入kernel DTB后U-Boot阶段存在两份DTB，其中存储、串口、Crypto等模块与U-Boot DTB相关，其余模块与kernel DTB相关。那么在U-Boot阶段可能在不同时刻需要交叉访问到这两类的模块，而模块又可能需要访问自己的DTB节点信息。

那么，这两类模块隶属于不同的DTB，而 `gd->fdt_blob` 又只能指向其中一份且不方便随意切换，这要怎么办？因为kernel 的 dts 最终要传递给 kernel 使用，所以也不能把 U-Boot dts 中的某些节点直接 overlay 到 kernel dts 上合成一份。

Live dt 可以解决这个问题。live dt 的原理是：初始化阶段U-Boot直接扫描整个DTB，把所有DTB节点转换成 `struct device_node` 节点链表，并且和具体的device-driver绑定。以后device-driver要访问DTB节点时直接访问自己的 `device_node` 即可，不需要再访问原有 DTB。

所以，相当于U-Boot和kernel 的DTB都绑定了各自的device-driver群组，且不再需要直接访问DTB文件。

这就解决了访问两份DTB引起的冲突。

更多参考:

```
./doc/driver-model/livetree.txt
```

### fdt 和 live dt 转换:

`ofnode` 类型 (`include/dm/ofnode.h`) 是两种 dt 都支持的一种封装格式，使用 live dt 时用 `device_node` 来访问 dt 结点，使用 fdt 时用 `offset` 访问 dt 节点。当需要同时支持两种类型的驱动时请使用 `ofnode` 类型。

`ofnode` 结构:

```

/*
 * @np: Pointer to device node, used for live tree
 * @of_offset: Pointer into flat device tree, used for flat tree. Note that
this
 *      is not a really a pointer to a node: it is an offset value. See above.
 */
typedef union ofnode_union {
    const struct device_node *np; /* will be used for future live tree */
    long of_offset;
} ofnode;

```

- "dev\_"、"ofnode\_"开头的函数为支持两种 dt 访问方式；
- "of\_"开头的函数是只支持 live dt 的接口；
- "fdtdec\_"、"fdt\_"开头的函数是只支持 fdt 的接口；

## 机制实现

kernel dtb 切换是在 `./arch/arm/mach-rockchip/board.c` 的 `init_kernel_dtb()` 里实现的。此时 U-Boot 的 dts 已经扫描完成，mmc/nand/nor 等存储驱动可正常工作。

此时从固件中读取 kernel dtb，然后进行 live dt 建表并 bind 所有 device-driver，最后更新 `gd->fdt_blob` 指针指向 kernel dtb 即可。

## U-Boot

- U-Boot 编译完成后会在 `./dts/` 目录下生成两个 dtb：
  - `dt.dtb`: 由 `defconfig` 里 `CONFIG_DEFAULT_DEVICE_TREE` 指定的 dts 编译得到的；
  - `dt-spl.dtb`: 把 `dt.dtb` 中带 `u-boot`, `dm-pre-reloc` 属性的节点全部抽取出来，再去掉 `defconfig` 里 `CONFIG_OF_SPL_REMOVE_PROPS` 指定的 property 得到。一般仅包含串口、DDR、存储等驱动必须依赖的节点：DMC、UART、MMC、NAND、GRF、CRU 等。
- 不启用 `CONFIG_USING_KERNEL_DTB` 时系统使用 `dt.dtb`；启用 `CONFIG_USING_KERNEL_DTB` 时系统使用 `dt-spl.dtb`。
- `dt.dtb` 或者 `dt-spl.dtb` 在 U-Boot 编译结束后都被命名为 `u-boot.dtb`，然后追加到 `u-boot.bin` 的末尾。用户可以通过 `fdtdump` 命令检查 `u-boot.dtb` 的内容。

## 内核传参

本章节介绍 U-Boot 如何向 kernel 传递参数。

### cmdline

U-Boot 把 kernel DTB 里的 `/chosen/bootargs` 读取出来，修改/追加上新内容后重新写回 `/chosen/bootargs` 节点，达到传递 cmdline 的目的。

### 内存容量

U-Boot 修改 kernel DTB 里的 `/memory` 节点，把可用的内存容量信息填写进去。开机信息有相关打印：

```
.....
## Chapter-6 Booting Android Image at 0x0027f800 ...
Kernel load addr 0x00280000 size 23387 KiB
RAM disk load addr 0x0a200000 size 782 KiB
## Chapter-6 Flattened Device Tree blob at 08300000
Booting using the fdt blob at 0x8300000
XIP Kernel Image ... OK
'reserved-memory' ramoops@110000: addr=110000 size=f0000
Using Device Tree in place at 000000008300000, end 000000008314648

// kernel 可用的内存空间
Adding bank: 0x00200000 - 0x08400000 (size: 0x08200000)
Adding bank: 0x0a200000 - 0x80000000 (size: 0x75e00000)
Total: 473.217 ms

Starting kernel ...
```

## 其它方式

其它的传参方式本质也都是修改 kernel DTB。如下：

节点/属性	操作	作用
/serial-number	创建	序列号
/memory	修改	kernel 可见内存
/display-subsystem/route/route-edp/	追加	显示相关参数(edp 为例)
/chosen/linux,initrd-start	创建	ramdisk 起始地址
/chosen/linux,initrd-end	创建	ramdisk 结束地址
/bootargs	修改	kernel 可见 cmdline
GMAC 节点内的 mac-address 或 local-mac-address	修改	mac 地址
arch/arm/mach-rockchip/board.c: board_fdt_fixup()	修改	板级的 fdt fixup

## AB系统

### AB 数据格式

A/B的数据结构位于 misc 分区偏移 2KB 位置。

```

/* Magic for the A/B struct when serialized. */
#define AVB_AB_MAGIC "\0AB0"
#define AVB_AB_MAGIC_LEN 4

/* Versioning for the on-disk A/B metadata - keep in sync with avbtool. */
#define AVB_AB_MAJOR_VERSION 1
#define AVB_AB_MINOR_VERSION 0

/* Size of AvbABData struct. */
#define AVB_AB_DATA_SIZE 32

/* Maximum values for slot data */
#define AVB_AB_MAX_PRIORITY 15
#define AVB_AB_MAX_TRIES_REMAINING 7

typedef struct AvbABSlotData {
    /* Slot priority. Valid values range from 0 to AVB_AB_MAX_PRIORITY,
     * both inclusive with 1 being the lowest and AVB_AB_MAX_PRIORITY
     * being the highest. The special value 0 is used to indicate the
     * slot is unbootable.
     */
    uint8_t priority;

    /* Number of times left attempting to boot this slot ranging from 0
     * to AVB_AB_MAX_TRIES_REMAINING.
     */
    uint8_t tries_remaining;

    /* Non-zero if this slot has booted successfully, 0 otherwise. */
    uint8_t successful_boot;

    /* Reserved for future use. */
    uint8_t reserved[1];
} AVB_ATTR_PACKED AvbABSlotData;

```

```

/* Struct used for recording A/B metadata.
 */
/* when serialized, data is stored in network byte-order.
 */
typedef struct AvbABData {
    /* Magic number used for identification - see AVB_AB_MAGIC. */
    uint8_t magic[AVB_AB_MAGIC_LEN];

    /* Version of on-disk struct - see AVB_AB_{MAJOR,MINOR}_VERSION. */
    uint8_t version_major;
    uint8_t version_minor;

    /* Padding to ensure |slots| field start eight bytes in. */
    uint8_t reserved1[2];

    /* Per-slot metadata. */
    AvbABSlotData slots[2];

    /* Reserved for future use. */
    uint8_t reserved2[12];

    /* CRC32 of all 28 bytes preceding this field. */
    uint32_t crc32;
} AVB_ATTR_PACKED AvbABData;

```

对于小容量存储，没有 misc 分区但有 vendor 分区，可以考虑存储到 vendor。

在此基础上增加 lastboot，标记最后一个可启动固件。主要应用于低电情况或工厂生产测试时 retry 次数用完，而还没有进入系统调用 boot\_ctrl 服务。参考如下：

```

typedef struct AvbABData {
    /* Magic number used for identification - see AVB_AB_MAGIC. */
    uint8_t magic[AVB_AB_MAGIC_LEN];

    /* Version of on-disk struct - see AVB_AB_{MAJOR,MINOR}_VERSION. */
    uint8_t version_major;
    uint8_t version_minor;

    /* Padding to ensure |slots| field start eight bytes in. */
    uint8_t reserved1[2];

    /* Per-slot metadata. */
    AvbABSlotData slots[2];

    /* mark last boot slot */
    uint8_t last_boot;
    /* Reserved for future use. */
    uint8_t reserved2[11];

    /* CRC32 of all 28 bytes preceding this field. */
    uint32_t crc32;
} AVB_ATTR_PACKED AvbABData;

```

同时在 AvbABSlotData 中增加 is\_update 标志位，标志系统升级的状态，更改如下：

```

typedef struct AvbABSlotData {
    /* Slot priority. Valid values range from 0 to AVB_AB_MAX_PRIORITY,
     * both inclusive with 1 being the lowest and AVB_AB_MAX_PRIORITY
     * being the highest. The special value 0 is used to indicate the
     * slot is unbootable.
     */
    uint8_t priority;

    /* Number of times left attempting to boot this slot ranging from 0
     * to AVB_AB_MAXTRIES_REMAINING.
     */
    uint8_t tries_remaining;

    /* Non-zero if this slot has booted successfully, 0 otherwise. */
    uint8_t successful_boot;

    /* Mark update state, mark 1 if the slot is in update state, 0 otherwise. */
    uint8_t is_update : 1;
    /* Reserved for future use. */
    uint8_t reserved : 7;
} AVB_ATTR_PACKED AvbABSlotData;

```

最后表格来说明各个参数的含义：

AvbABData：

参数	含义
priority	标志 slot 优先级，0 为不可启动，15 为最高优先级
tries_remaining	尝试启动次数，设置为 7 次
successful_boot	系统启动成功后会配置该参数，1：该 slot 成功启动，0：该 slot 未成功启动
is_update	标记该 slot 的升级状态，1：该 slot 正在升级，0：该 slot 未升级或升级成功

AvbABSlotData：

参数	含义
magic	结构体头部信息：\0AB0
version_major	主版本信息
version_minor	次版本信息
slots	slot 引导信息，参见 AvbABData
last_boot	上一次成功启动的 slot，0：slot A 上次成功启动，1：slot B 上次成功启动
crc32	数据校验

## AB 启动模式

目前 system bootctrl 设计两套控制模式，bootloader 支持这两种模式启动。

### successful-boot

正常进入系统后，boot\_ctrl 依据 androidboot.slot\_suffix，设置当前 slot 的变量：

```
successful_boot = 1;
priority = 15;
tries_remaining = 0;
is_update = 0;
last_boot = 0 or 1;      :refer to androidboot.slot_suffix
```

升级系统中，boot\_ctrl 设置：

```
升级的slot设置:
successful_boot = 0;
priority = 14;
tries_remaining = 7;
is_update = 1;
lastboot = 0 or 1;      :refer to androidboot.slot_suffix

当前slot设置:
successful_boot = 1;
priority = 15;
tries_remaining = 0;
is_update = 0;
last_boot = 0 or 1;      :refer to androidboot.slot_suffix
```

升级系统完成，boot\_ctrl 设置：

```
升级的slot设置:
successful_boot = 0;
priority = 15;
tries_remaining = 7;
is_update = 0;
lastboot = 0 or 1;      :refer to androidboot.slot_suffix

当前slot设置:
successful_boot = 1;
priority = 14;
tries_remaining = 0;
is_update = 0;
last_boot = 0 or 1;      :refer to androidboot.slot_suffix
```

### **reset-retry**

正常进入系统后，boot\_ctrl 依据 androidboot.slot\_suffix，设置当前 slot 的变量：

```
successful_boot = 0;
priority = 15;
tries_remaining = 7;
is_update = 0;
last_boot = 0 or 1;      :refer to androidboot.slot_suffix
```

升级系统中，boot\_ctrl 设置：

```
升级的slot设置:  
successful_boot = 0;  
priority = 14;  
tries_remaining = 7;  
is_update = 1;  
lastboot = 0 or 1;      :refer to androidboot.slot_suffix
```

```
当前slot设置:  
successful_boot = 0;  
priority = 15;  
tries_remaining = 7;  
is_update = 0;  
last_boot = 0 or 1;      :refer to androidboot.slot_suffix
```

升级系统完成，boot\_ctrl 设置：

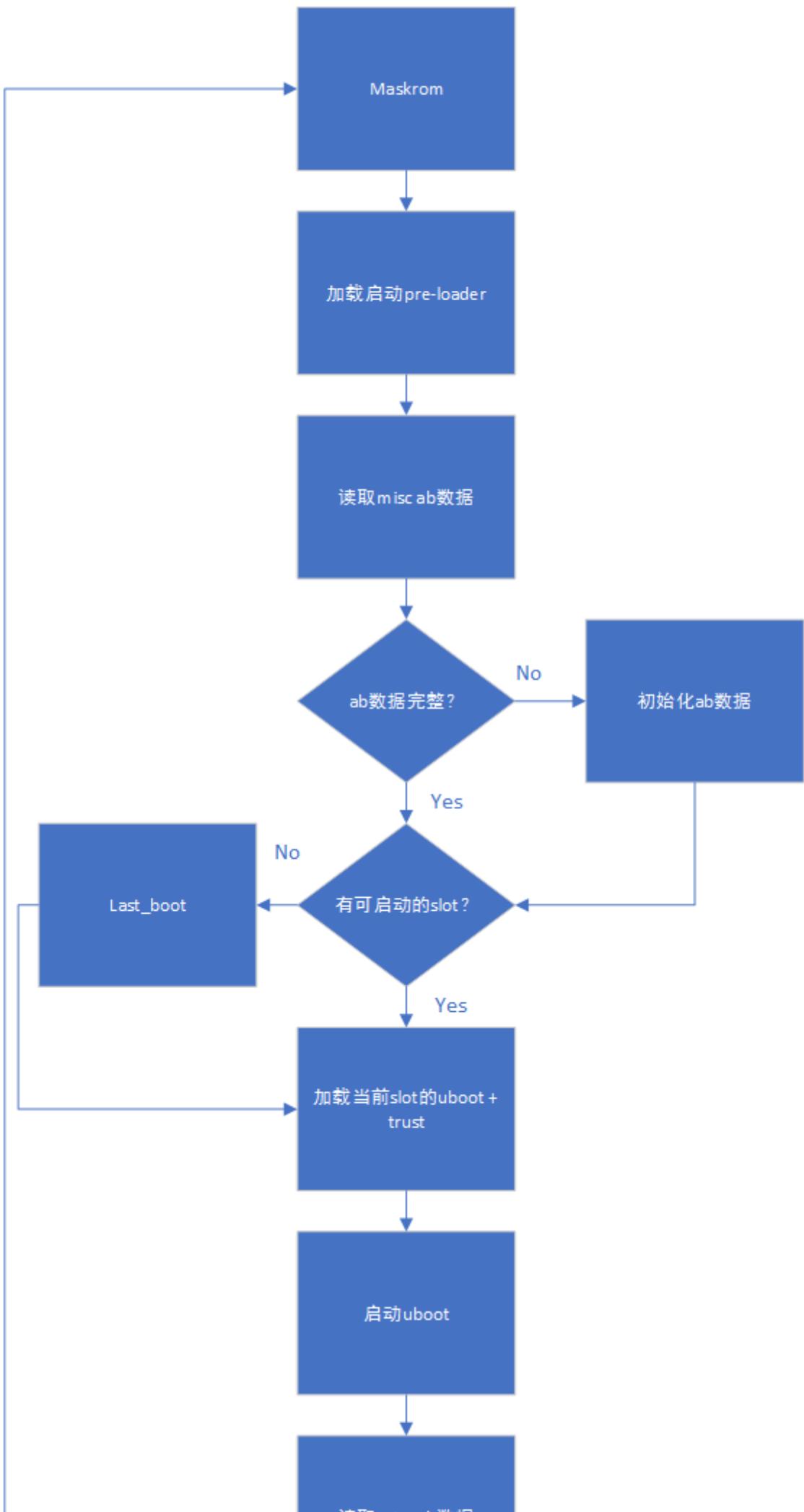
```
升级的slot设置:  
successful_boot = 0;  
priority = 15;  
tries_remaining = 7;  
is_update = 0;  
lastboot = 0 or 1;      :refer to androidboot.slot_suffix
```

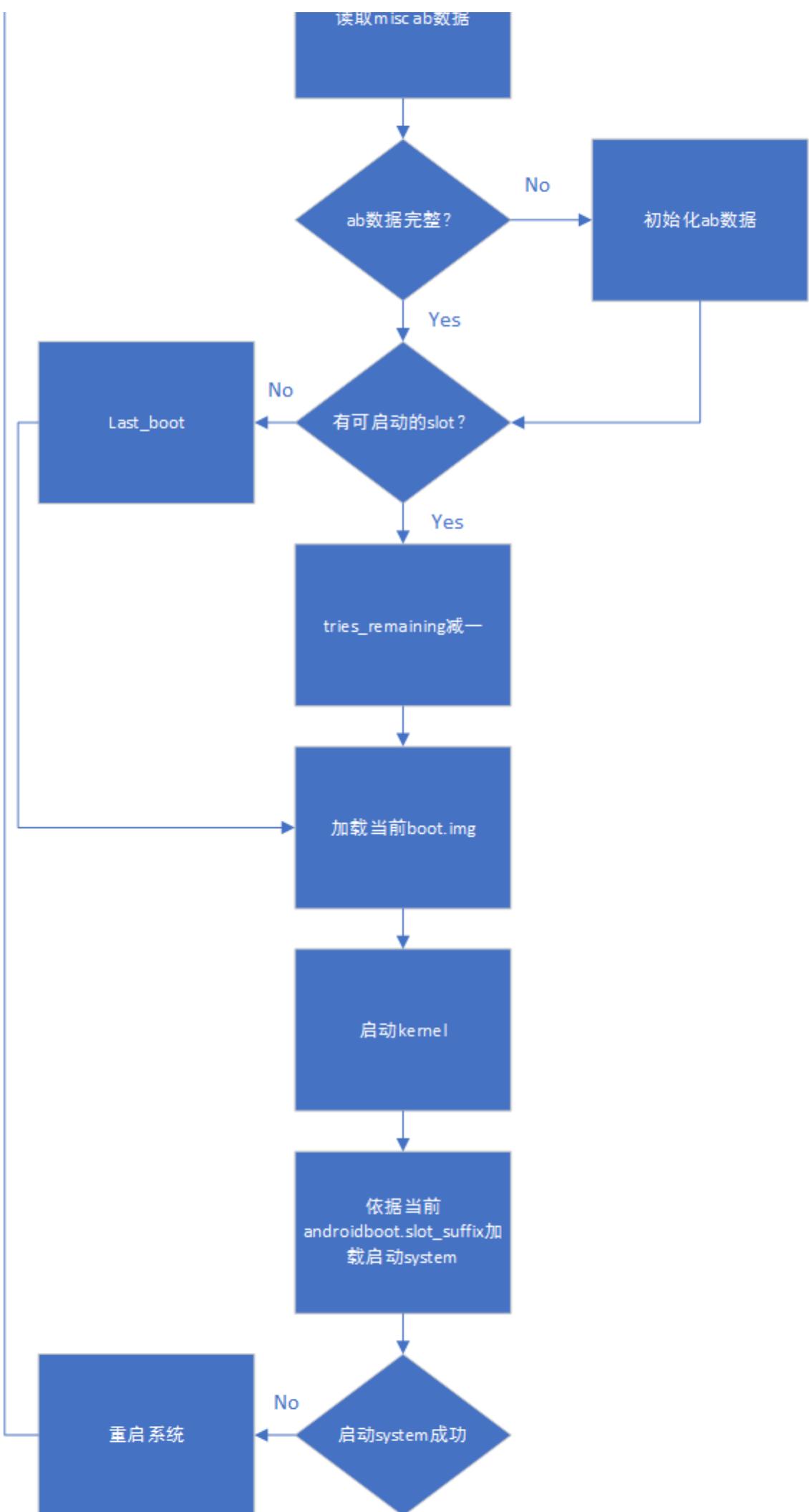
```
当前slot设置:  
successful_boot = 0;  
priority = 14;  
tries_remaining = 7;  
is_update = 0;  
last_boot = 0 or 1;      :refer to androidboot.slot_suffix
```

## 模式对比

- successful\_boot 模式
  - 优点：只要正常启动系统，不会回退到旧版本固件，除非 system bootctrl 配置
  - 缺点：设备长时间工作后，如果存储某些颗粒异常，会导致系统一直重启
- reset retry 模式
  - 优点：始终保持 retry 机制，可以应对存储异常问题
  - 缺点：会回退到旧版本固件

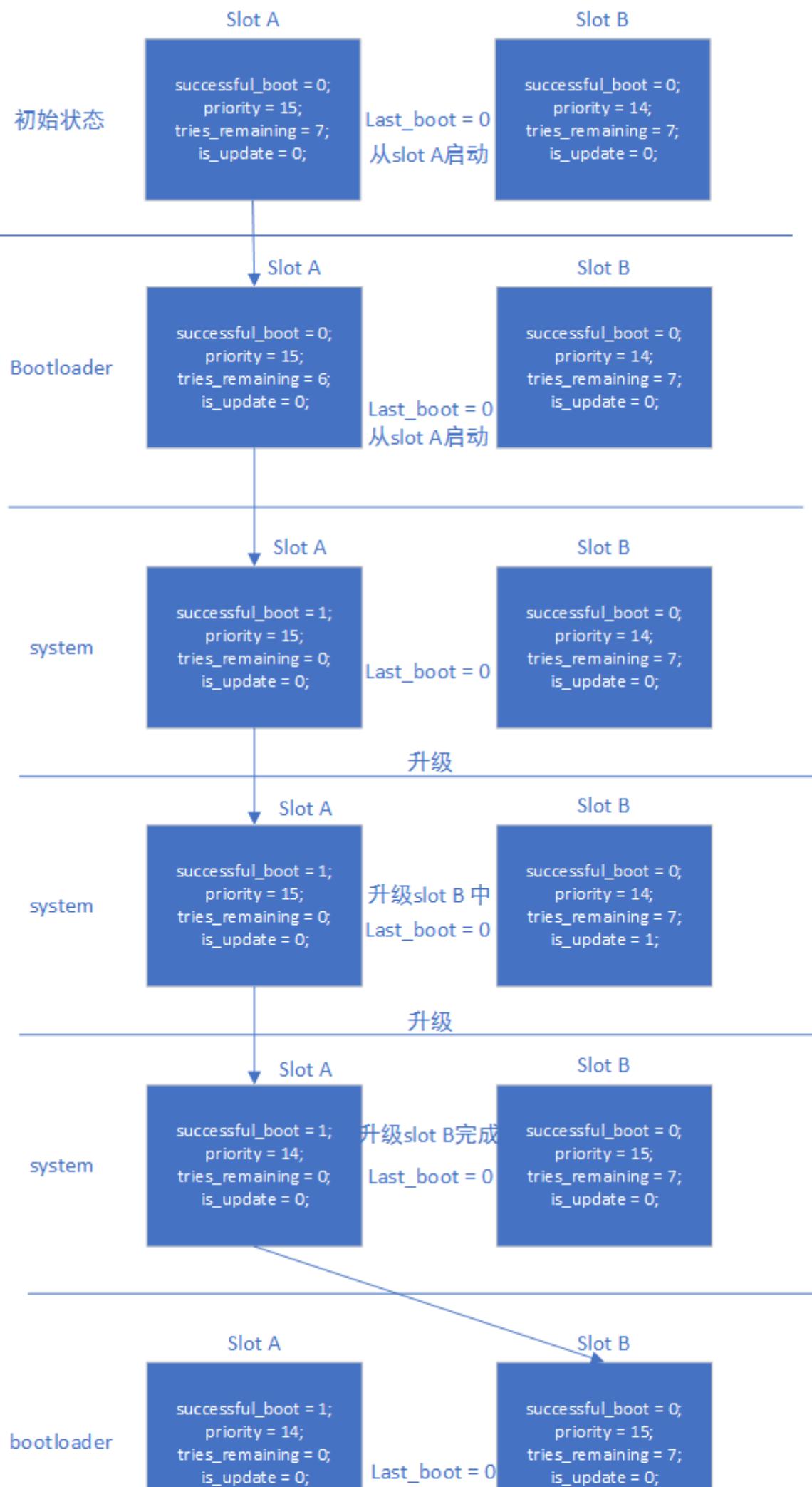
## 启动流程





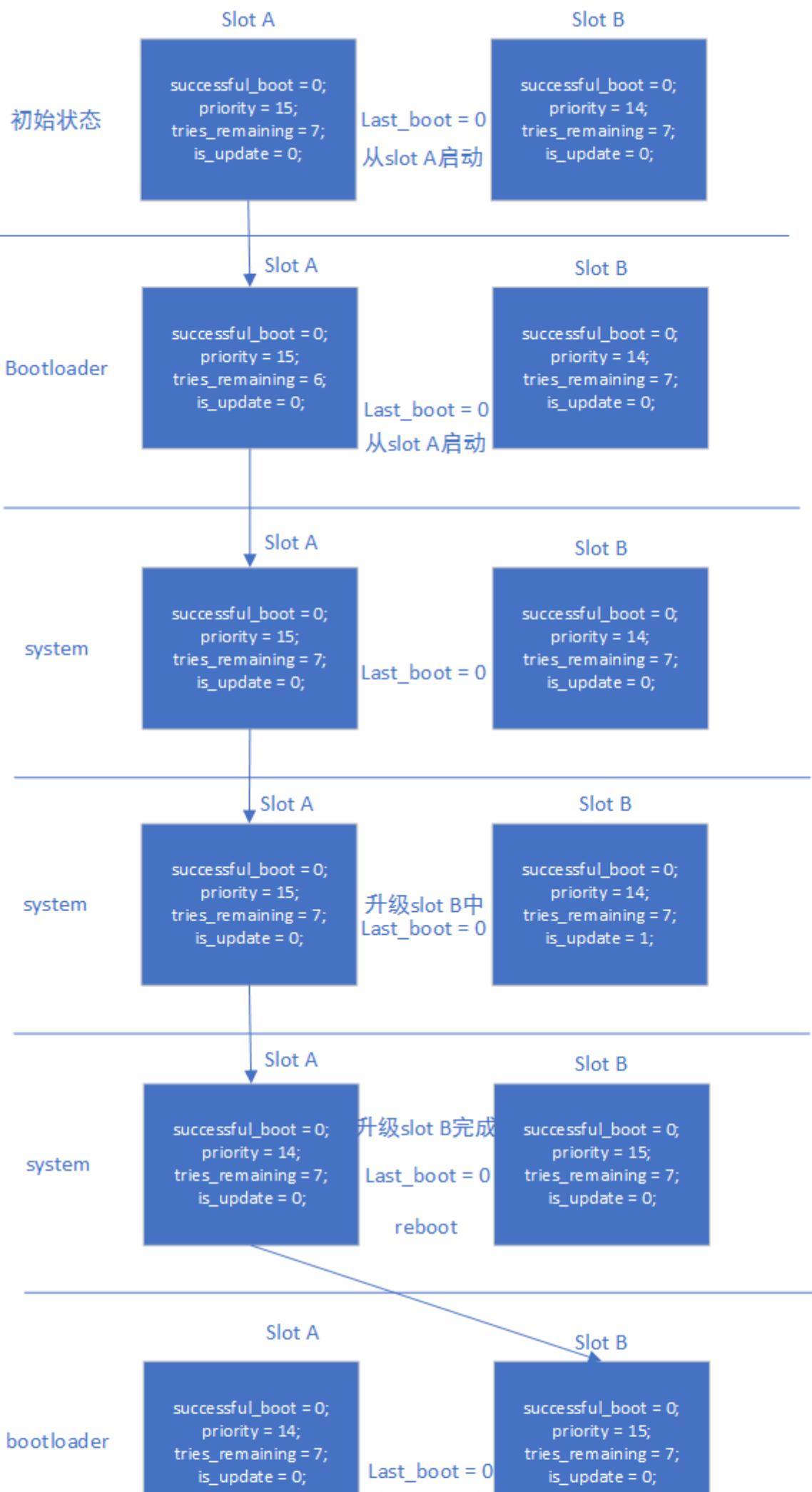


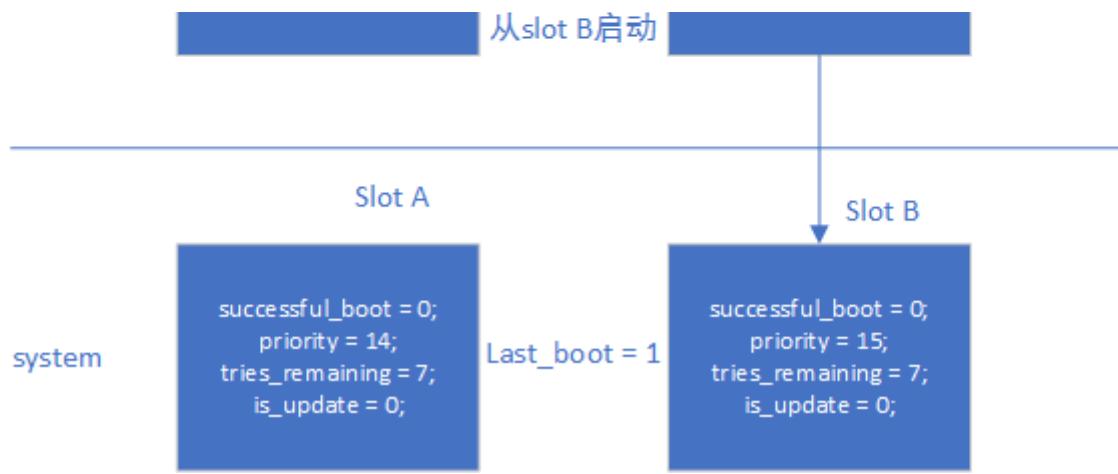
**AB successful\_boot 模式数据流程:**





AB reset retry 模式数据流程:





## 升级和异常

- 系统升级：参考《Rockchip Linux 升级方案开发指南》
- recovery 升级：AB system 不考虑支持 recovery 升级

## 验证方法

### successful-boot

1. 只烧写 slot A，系统从 slot A 启动。设置从 slot B 启动，系统从 slot A 启动。测试完成，清空 misc 分区
2. 烧写 slot A 与 slot B，启动系统，当前系统为 slot A。设置系统从 slot B 启动，reboot 系统，当前系统为 slot B。测试完成，清空 misc 分区
3. 烧写 slot A 与 slot B，迅速 reset 系统 14 次后，retry counter 用完，还能从 last\_boot 指定的系统启动，即能正常从 slot A 启动。测试完成，清空 misc 分区
4. 烧写 slot A 与 slot B，启动系统，当前系统为 slot A。设置系统从 slot B 启动，reboot 系统，当前系统为 slot B。设置系统从 slot A 启动，reboot 系统，当前系统为 slot A。测试完成，清空 misc 分区

### reset-retry

1. 只烧写 slot A，系统从 slot A 启动。设置从 slot B 启动，系统从 slot A 启动。测试完成，清空 misc 分区
2. 烧写 slot A 与 slot B，启动系统，当前系统为 slot A。设置系统从 slot B 启动，reboot 系统，当前系统为 slot B。测试完成，清空 misc 分区
3. 烧写 slot A 与 slot B，迅速 reset 系统 14 次后，retry counter 用完，还能从 last\_boot 指定的系统启动，即能正常从 slot A 启动。测试完成，清空 misc 分区
4. 烧写 slot A 与 slot B，其中 slot B 的 boot.img 损坏，启动系统，当前系统为 slot A。设置系统从 slot B 启动，reboot 系统，系统会重启 7 次后，从 slot A 正常启动系统。测试完成，清空 misc 分区
5. 烧写 slot A 与 slot B，启动系统，当前系统为 slot A。设置系统从 slot B 启动，reboot 系统，当前系统为 slot B。设置系统从 slot A 启动，reboot 系统，当前系统为 slot A。测试完成，清空 misc 分区

## 引用参考

- 《Rockchip-Secure-Boot2.0.md》
- 《Rockchip-Secure-Boot-Application-Note.md》
- 《Android Verified Boot 2.0》

## AVB安全启动

## 引用参考

《Rockchip-Secure-Boot-Application-Note.md》

《Android Verified Boot 2.0》

《Rockchip\_Developer\_Guide\_Linux4.4\_SecureBoot\_CN.pdf》

## 术语

AVB : Android Verified Boot

OTP & efuse : One Time Programmable

Product RootKey (PRK): AVB 的 root key 由签名 loader, uboot & trust 的 root key 校验

ProductIntermediate Key (PIK): 中间 key, 中介作用

ProductSigning Key (PSK): 用于签固件的 key

ProductUnlock Key (PUK): 用于解锁设备

**各种key分离，职责明确，可以降低key被泄露的风险。**

## 简介

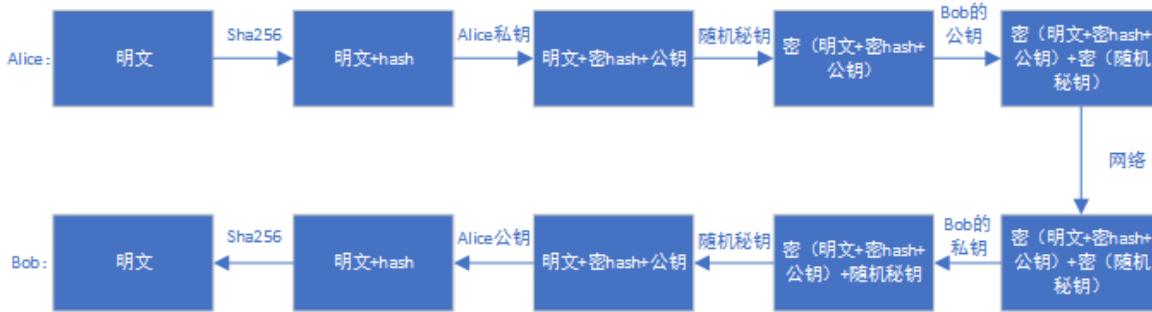
本文介绍 Rockchip 安全验证引导流程。所谓的安全验证引导流程分为安全性校验与完整性校验。安全性校验是加密公钥的校验，流程为从安全存储（OTP & efuse）中读取公钥 hash，与计算的公钥 hash 对比，是否一致，然后公钥用于解密固件 hash。完整性校验为校验固件的完整性，流程为从存储里加载固件，计算固件的 hash 与解密出来的 hash 对比是否一致。

## 加密示例

设备的安全验证启动流程与通信中的数据加密校验流程类似，通过该例子可以加速对 avb 校验流程的理解。假如现在 Alice 向 Bob 传送数字信息，为了保证信息传送的保密性、真实性、完整性和不可否认性，需要对传送的信息进行数字加密和签名，其传送过程为：

- 1.Alice 准备好要传送的数字信息（明文）；
- 2.Alice 对数字信息进行哈希运算，得到一个信息摘要；
- 3.Alice 用自己的私钥对信息摘要进行加密得到 Alice 的数字签名，并将其附在数字信息上；
- 4.Alice 随机产生一个加密密钥，并用此密码对要发送的信息进行加密，形成密文；
- 5.Alice 用 Bob 的公钥对刚才随机产生的加密密钥进行加密，将加密后的 DES 密钥连同密文一起传送给 Bob；
- 6.Bob 收到 Alice 传送来的密文和加密过的 DES 密钥，先用自己的私钥对加密的 DES 密钥进行解密，得到 Alice 随机产生的加密密钥；
- 7.Bob 然后用随机密钥对收到的密文进行解密，得到明文的数字信息，然后将随机密钥抛弃；
- 8.Bob 用 Alice 的公钥对 Alice 的数字签名进行解密，得到信息摘要；
- 9.Bob 用相同的哈希算法对收到的明文再进行一次哈希运算，得到一个新的信息摘要；
- 10.Bob 将收到的信息摘要和新产生的信息摘要进行比较，如果一致，说明收到的信息没有被修改过。

上面提及的 DES 算法可以更换其他算法，如 AES 加密算法，公私钥算法可以采用 RSA 算法，流程如下：



## AVB

AVB 为 Android Verified Boot，谷歌设计的一套固件校验流程，主要用于校验 boot system 等固件。Rockchip Secure Boot 参考通信中的校验方式及 AVB，实现一套完整的 Secure Boot 校验方案。

### AVB 特性

- 安全校验
- 完整性校验
- 防回滚保护
- persistent partition 支持
- chained partitions 支持，可以与 boot, system 签名私钥一致，也可以由 oem 自己保存私钥，但必须由 PRK 签名

### key+签名+证书

```

#!/bin/sh
touch temp.bin
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -outform PEM -out testkey_prk.pem
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -outform PEM -out testkey_psk.pem
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -outform PEM -out testkey_pik.pem
python avbtool make_atx_certificate --output=pik_certificate.bin --
subject=temp.bin --subject_key=testkey_pik.pem --
subject_is_intermediate_authority --subject_key_version 42 --
authority_key=testkey_prk.pem
python avbtool make_atx_certificate --output=psk_certificate.bin --
subject=product_id.bin --subject_key=testkey_psk.pem --subject_key_version 42 --
authority_key=testkey_pik.pem
python avbtool make_atx_metadata --output=metadata.bin --
intermediate_key_certificate=pik_certificate.bin --
product_key_certificate=psk_certificate.bin

```

permanent\_attributes.bin 生成：

```

python avbtool make_atx_permanent_attributes --output=permanent_attributes.bin --
-product_id=product_id.bin --root_authority_key=testkey_prk.pem

```

其中 product\_id.bin 需要自己定义，占 16 字节，可作为产品 ID 定义。

boot.img 签名示例：

```

avbtool add_hash_footer --image boot.img --partition_size 33554432 --
partition_name boot --key testkey_psk.pem --algorithm SHA256_RSA4096

```

**注意：partition size 要至少比原固件大 64K，大小还要 4K 对齐，且不大于 parameter.txt 定义的分区大小。**

system.img 签名示例：

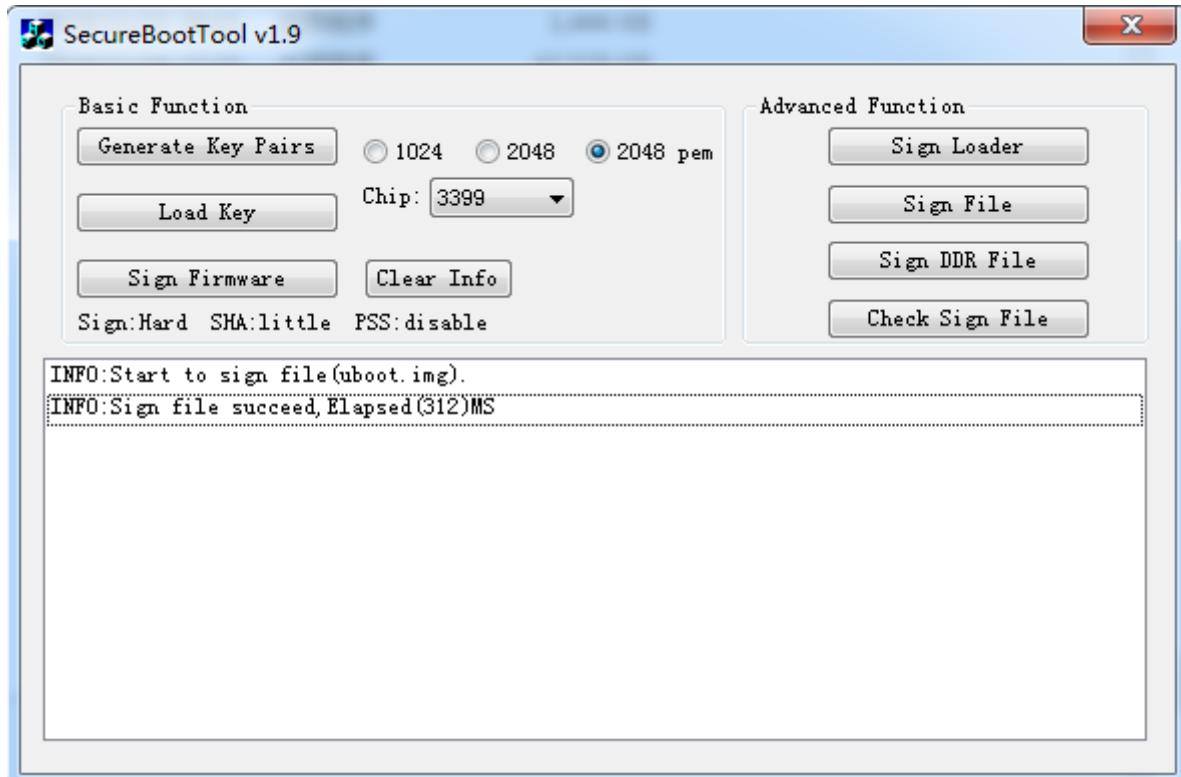
```
avbtool add_hashtree_footer --partition_size 536870912 --partition_name system -  
-image system.img --algorithm SHA256_RSA4096 --key testkey_psk.pem
```

生成 vbmeta 包含 metadata.bin，命令示例如下：

```
python avbtool make_vbmeta_image --public_key_metadata metadata.bin --  
include_descriptors_from_image boot.img --include_descriptors_from_image  
system.img --generate_dm_verity_cmdline_from_hashtree system.img --algorithm  
SHA256_RSA4096 --key testkey_psk.pem --output vbmeta.img
```

最终把生成的 vbmeta.img 烧写到对应的分区，如 vbmeta 分区。

通过 SecureBootTool 生成 PrivateKey.pem 和 PublicKey.pem。



对 permanent\_attributes.bin 进行签名：

```
openssl dgst -sha256 -out permanent_attributes_cer.bin -sign PrivateKey.pem  
permanent_attributes.bin
```

permanent\_attributes.bin 是整个系统的安全认证数据，它需要烧写它的 hash 到 efuse 或 OTP，或它的数据由前级的安全认证（pre-load）。由于 rockchip 平台规划的 efuse 不足，所以 permanent\_attributes.bin 的验证由前级的公钥加 permanent\_attributes.bin 的证书进行认证。而对于有OTP的平台，安全数据空间足够，会直接烧写 permanent\_attributes.bin 的 hash 到 OTP。

各个平台efuse与OTP支持情况：

平台	efuse	OTP
rk3399	✓	
rk3368	✓	
rk3328		✓
rk3326		✓
rk3308		✓
rk3288	✓	
rk3229	✓	
rk3126	✓	
rk3128	✓	

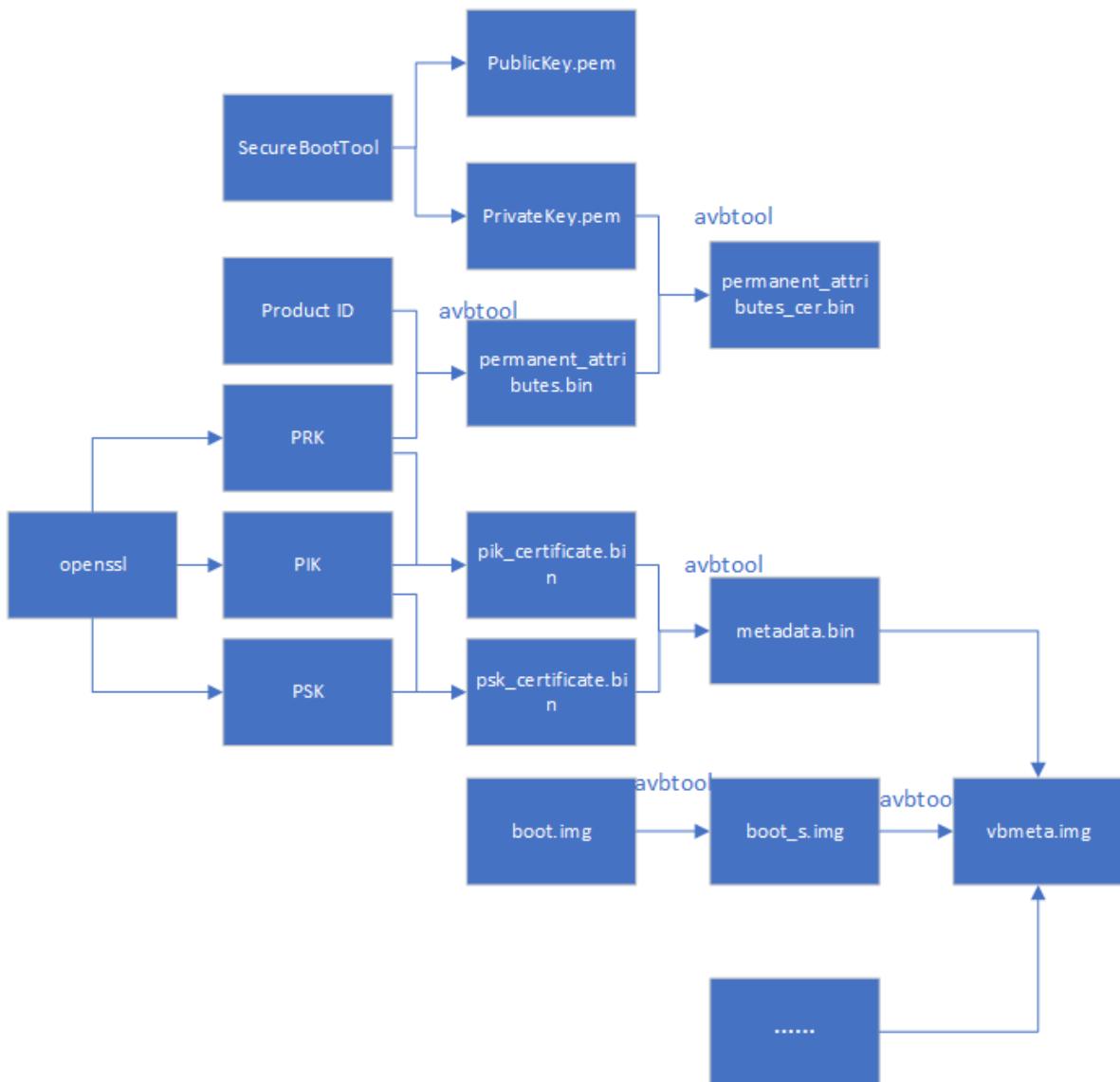
efuse 平台 pub\_key 烧写:

```
fastboot stage permanent_attributes.bin
fastboot oem fuse at-perm-attr
fastboot stage permanent_attributes_cer.bin
fastboot oem fuse at-rsa-perm-attr
```

OTP 平台 pub\_key 烧写:

```
fastboot stage permanent_attributes.bin
fastboot oem fuse at-perm-attr
```

整个签名流程:



## AVB lock

```
fastboot oem at-lock-vboot
```

如何进入 fastboot 见 fastboot 命令支持章节。

## AVB unlock

目前 Rockchip 采用严格安全校验，需要在对应的defconfig内添加

```
CONFIG_RK_AVB_LIBAVB_ENABLE_ATH_UNLOCK=y
```

否则输入 fastboot oem at-unlock-vboot 就可以解锁设备，启动校验 vbmeta.img, boot.img 失败也会成功启动设备。

首先，需要生成 PUK:

```
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -outform PEM -out testkey_puk.pem
```

unlock\_credential.bin 为需要下载到设备解锁的证书，其生成过程如下：

```

python avbtool make_atx_certificate --output=puk_certificate.bin --
subject=product_id.bin --subject_key=testkey_puk.pem --
usage=com.google.android.things.vboot.unlock --subject_key_version 42 --
authority_key=testkey_pik.pem

```

从设备获取 unlock\_credential.bin，使用 avb-challenge-verify.py 脚本获取 unlock\_credential.bin，执行下列命令获取 unlock\_credential.bin：

```

python avbtool make_atx_unlock_credential --output=unlock_credential.bin --
intermediate_key_certificate=pik_certificate.bin --
unlock_key_certificate=puk_certificate.bin --challenge=unlock_challenge.bin --
unlock_key=testkey_puk.pem

```

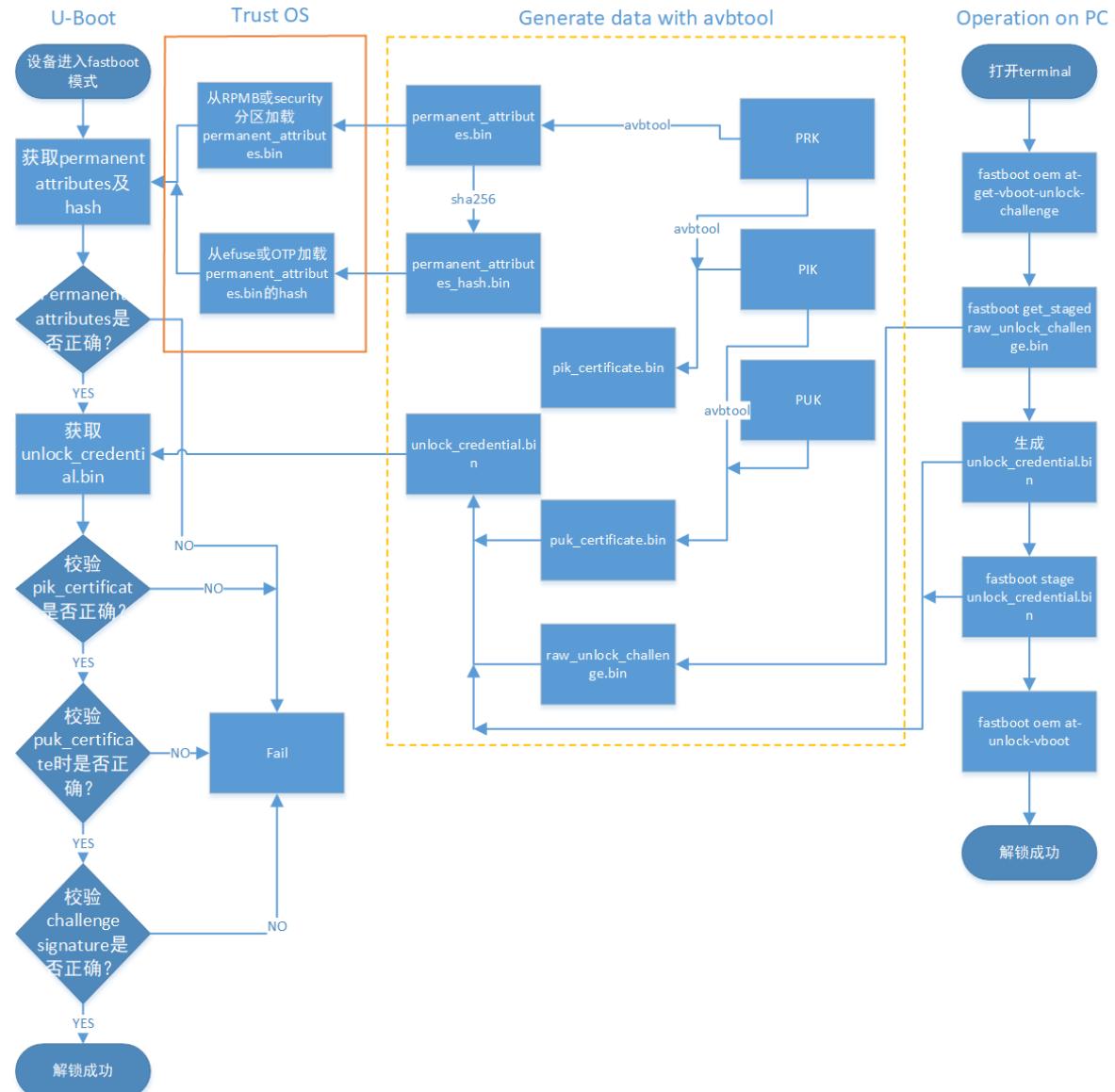
最终可以把证书通过 fastboot 命令下载到设备，并解锁设备，fastboot 命令如下：

```

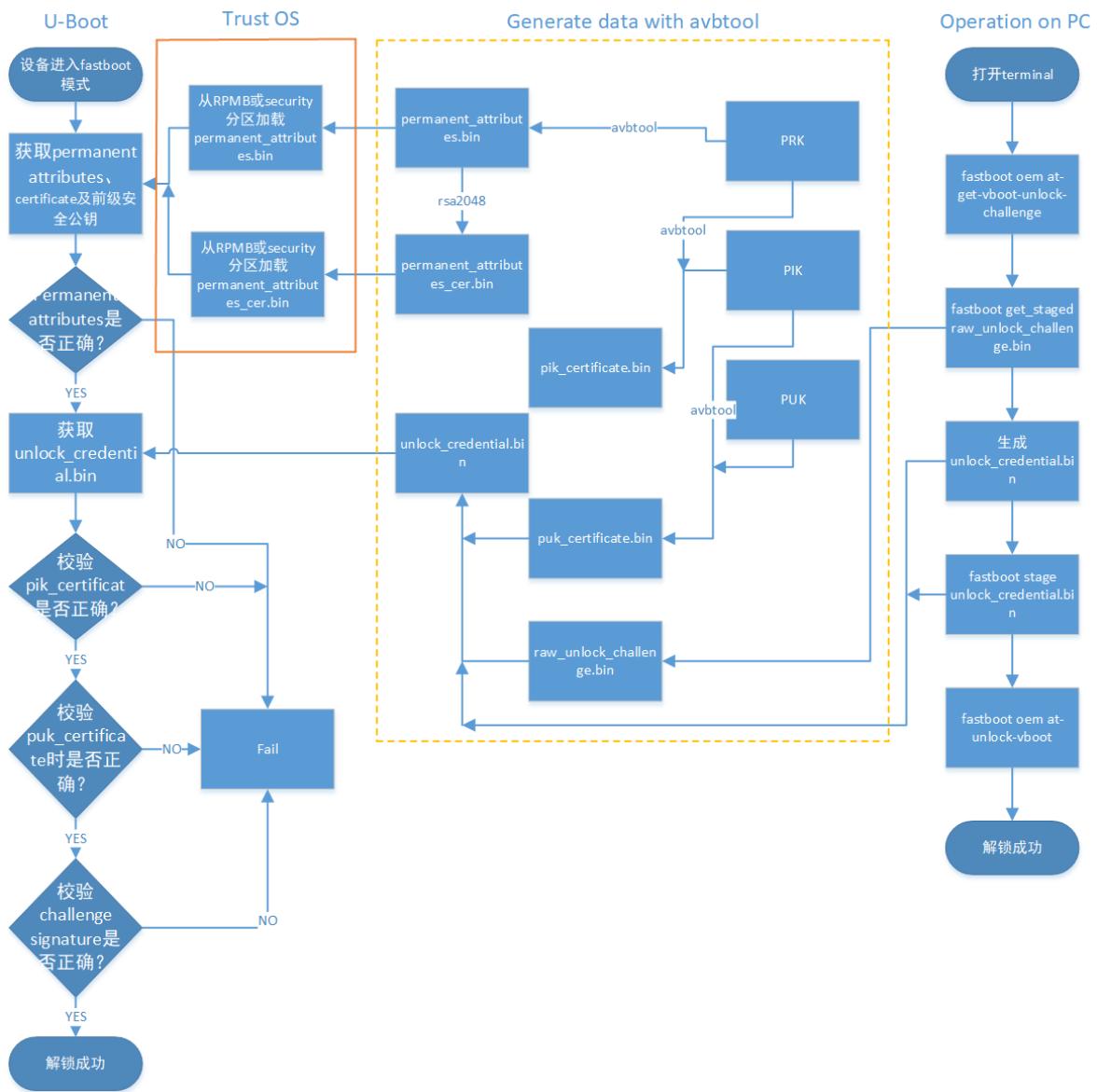
fastboot stage unlock_credential.bin
fastboot oem at-unlock-vboot

```

最后 OTP 设备解锁流程：



最后 efuse 设备解锁流程：



最后操作流程如下：

1. 设备进入 fastboot 模式，电脑端输入

```
fastboot oem at-get-vboot-unlock-challenge
fastboot get_staged raw_unlock_challenge.bin
```

获得带版本、Product Id 与 16 字节的随机数的数据，取出随机数作为 unlock\_challenge.bin。

1. 使用 avbtool 生成 unlock\_credential.bin，参考make\_unlock.sh。
2. 电脑端输入

```
fastboot stage unlock_credential.bin
fastboot oem at-unlock-vboot
```

**注意：**此时设备状态一直处于第一次进入 fastboot 模式状态，在此期间不能断电、关机、重启。因为步骤 1.做完后，设备存储着生成的随机数，如果断电、关机、重启，会导致随机数丢失，后续校验 challenge signature 会因为随机数不匹配失败。

如果开启：

```
CONFIG_MISC=y
CONFIG_ROCKCHIP_EFUSE=y
CONFIG_ROCKCHIP OTP=y
```

就会使用 CPUID 作为 challenge number，而 CPUID 是与机器匹配的，数据不会因为关机而丢失，生成的unlock\_credential.bin 可以重复使用。省去重复生成unlock\_challenge.bin，制作 unlock\_credential.bin 的步骤。再次解锁步骤变为：

```
fastboot oem at-get-vboot-unlock-challenge
fastboot stage unlock_credential.bin
fastboot oem at-unlock-vboot
```

1. 设备进入解锁状态，开始解锁。

make\_unlock.sh 参考

```
#!/bin/sh
python avb-challenge-verify.py raw_unlock_challenge.bin product_id.bin
python avbtool make_unlock_credential --output=unlock_credential.bin --
intermediate_key_certificate=pik_certificate.bin --
unlock_key_certificate=puk_certificate.bin --challenge=unlock_challenge.bin --
unlock_key=testkey_puk.pem
```

avb-challenge-verify.py 源码

```
#!/usr/bin/env python
"This is a test module for getting unlock_challenge.bin"
import sys
import os
from hashlib import sha256

def challenge_verify():
    if (len(sys.argv) != 3) :
        print "Usage: rkpublickey.py [challenge_file] [product_id_file]"
        return
    if ((sys.argv[1] == "-h") or (sys.argv[1] == "--h")):
        print "Usage: rkpublickey.py [challenge_file] [product_id_file]"
        return
    try:
        challenge_file = open(sys.argv[1], 'rb')
        product_id_file = open(sys.argv[2], 'rb')
        challenge_random_file = open('unlock_challenge.bin', 'wb')
        challenge_data = challenge_file.read(52)
        product_id_data = product_id_file.read(16)
        product_id_hash = sha256(product_id_data).digest()
        print("The challenge version is %d" %ord(challenge_data[0]))
        if (product_id_hash != challenge_data[4:36]) :
            print("Product id verify error!")
            return
        challenge_random_file.write(challenge_data[36:52])
        print("Success!")

    finally:
        if challenge_file:
            challenge_file.close()
```

```

if product_id_file:
    product_id_file.close()
if challenge_random_file:
    challenge_random_file.close()

if __name__ == '__main__':
    challenge_verify()

```

#### 4.5.5 U-boot 使能

开启 avb 需要 trust 支持，需要 U-Boot 在 defconfig 文件中配置：

```

CONFIG_OPTEE_CLIENT=y
CONFIG_OPTEE_V1=y
CONFIG_OPTEE_ALWAYS_USE_SECURITY_PARTITION=y // 安全数据存储到security分区

```

CONFIG\_OPTEE\_V1：适用平台有 312x,322x,3288,3228H,3368,3399。

CONFIG\_OPTEE\_V2：适用平台有 3326,3308。

CONFIG\_OPTEE\_ALWAYS\_USE\_SECURITY\_PARTITION：当 emmc 的 rpmb 不能用，才开这个宏，默认不开。

avb 开启需要在 defconfig 文件中配置：

```

CONFIG_AVB_LIBAVB=y
CONFIG_AVB_LIBAVB_AB=y
CONFIG_AVB_LIBAVB_ATX=y
CONFIG_AVB_LIBAVB_USER=y
CONFIG_RK_AVB_LIBAVB_USER=y
// 上面几个为必选，下面选择为支持 AVB 与 A/B 特性，两个特性可以分开使用。
CONFIG_ANDROID_AB=y //这个支持 A/B
CONFIG_ANDROID_AVB=y //这个支持 AVB
// 下面宏为仅有 efuse 的平台使用
CONFIG_ROCKCHIP_PRELOADER_PUB_KEY=y
// 下面宏需要严格unlock校验时候打开
CONFIG_RK_AVB_LIBAVB_ENABLE_ATH_UNLOCK=y
// 安全校验开启
CONFIG_AVB_VBMETA_PUBLIC_KEY_VALIDATE=y
// 如果需要cpuid作为challenge number，开启以下宏
CONFIG_MISC=y
CONFIG_ROCKCHIP_EFUSE=y
CONFIG_ROCKCHIP OTP=y

```

### kernel 配置

system, vendor, oem 等分区的校验由 kernel 的 dm-verify 模块加载校验，所以需要使能该模块。

使能 AVB 需要在 kernel dts 上配置参数 avb，参考如下：

```

&firmware_android {
    compatible = "android,firmware";
    boot_devices = "fe330000.sdhci";
    vbmota {
        compatible = "android,vbmota";
        parts = "vbmota,boot,system,vendor,dtbo";
    };
    fstab {

```

```

    compatible = "android,fstab";
    vendor {
        compatible = "android,vendor";
        dev = "/dev/block/by-name/vendor";
        type = "ext4";
        mnt_flags = "ro,barrier=1,inode_readahead_blnks=8";
        fsmgr_flags = "wait,avb";
    };
};

};


```

使能 A/B system 需要配置 slotselect 参数，参考如下：

```

firmware {
    android {
        compatible = "android,firmware";
        fstab {
            compatible = "android,fstab";
            system {
                compatible = "android,system";
                dev = "/dev/block/by-name/system";
                type = "ext4";
                mnt_flags = "ro,barrier=1,inode_readahead_blnks=8";
                fsmgr_flags = "wait,verify,slotselect";
            };
            vendor {
                compatible = "android,vendor";
                dev = "/dev/block/by-name/vendor";
                type = "ext4";
                mnt_flags = "ro,barrier=1,inode_readahead_blnks=8";
                fsmgr_flags = "wait,verify,slotselect";
            };
        };
    };
};

};


```

## Android SDK

如下介绍 Android SDK 上的一些配置说明。

### AVB Enable

使能 BOARD\_AVB\_ENABLE

### A/B system

这些变量主要有三类：

- A/B 系统必须定义的变量
  - AB\_OTA\_UPDATER := true
  - AB\_OTA\_PARTITIONS := boot system vendor
  - BOARD\_BUILD\_SYSTEM\_ROOT\_IMAGE := true
  - TARGET\_NO\_RECOVERY := true
  - BOARDUSESRECOVERYASBOOT := true
  - PRODUCT\_PACKAGES += update\_engine update\_verifier
- A/B 系统可选定义的变量

- PRODUCT\_PACKAGES\_DEBUG += update\_engine\_client
- A/B 系统不能定义的变量
  - BOARD\_RECOVERYIMAGE\_PARTITION\_SIZE
  - BOARD\_CACHEIMAGE\_PARTITION\_SIZE
  - BOARD\_CACHEIMAGE\_FILE\_SYSTEM\_TYPE

## Cmdline 新内容

```
Kernel command line: androidboot.verifiedbootstate=green
androidboot.slot_suffix=_a dm=="1 vroot none ro 1,0 1031864 verity 1
PARTUUID=b2110000-0000-455a-8000-44780000706f PARTUUID=b2110000-0000-455a-8000-
44780000706f 4096 4096 128983 128983 sha1
90d1d406caac04b7e3fbf48b9a4dc6992cc628e
4172683f0d6b6085c09f6ce165cf152fe3523c89 10 restart_on_corruption
ignore_zero_blocks use_fec_from_device PARTUUID=b2110000-0000-455a-8000-
44780000706f fec_roots 2 fec_blocks 130000 fec_start 130000" root=/dev/dm-0
androidboot.vbmeta.device=PARTUUID=f24f0000-0000-4e1b-8000-791700006a98
androidboot.vbmeta.avb_version=1.1 androidboot.vbmeta.device_state=unlocked
androidboot.vbmeta.hash_alg=sha512 androidboot.vbmeta.size=6528
androidboot.vbmeta.digest=41991c02c82ea1191545c645e2ac9cc7ca08b3da0a2e3115aff479
d2df61feaccdd35b6360cfa936f6f4381e4557ef18e381f4b236000e6ecc9ada401eda4cae
androidboot.vbmeta.invalidate_on_error=yes androidboot.veritymode=enforcing
```

这里说明几个参数：

1. 为什么传递 vbmeta 的 PARTUUID? 因为确保后续使用 vbmeta hash-tree 的合法性, 需要 kernel 再校验一遍 vbmeta, digest 为 androidboot.vbmeta.digest.
2. skip\_initramfs: boot ramdisk 有无打包到 boot.img 问题, 在 A/B system 中, ramdisk 是没有打包到 boot.img, cmdline需要传递这个参数。
3. root=/dev/dm-0 开启 dm-verify, 指定system。
4. androidboot.vbmeta.device\_state: android verify 状态
5. androidboot.verifiedbootstate: 校验结果。

green: If in LOCKED state and an the key used for verification was not set by the end user。

yellow: If in LOCKED state and an the key used for verification was set by the end user。

orange: If in the UNLOCKED state。

## 这里特别说明一下 dm=="1 vroot none ro....."参数生成:

```
avbtool make_vbmeta_image --include_descriptors_from_image boot.img --
include_descriptors_from_image system.img --
generate_dm_verity_cmdline_from_hashtree system.img --
include_descriptors_from_image vendor.img --algorithm SHA512_RSA4096 --key
testkey_psk.pem --public_key_metadata metadata.bin --output vbmeta.img
```

avbtool 生成 vbmeta 时, 对 system 固件加--generate\_dm\_verity\_cmdline\_from\_hashtree 即可。dm=="1 vroot none ro....."这些信息会保存到 vbmeta。这部分安卓专用, 如果分区只校验到 boot.img, 无需增加该参数。

Android SDK 开启 BOARD\_AVB\_ENABLE 会把这些信息加到 vbmeta 内。

## 分区参考

新增加 vbmeta 分区与 security 分区，vbmeta 分区存储固件校验信息，security 分区存储加密过的安全数据。

```
FIRMWARE_VER:8.0
MACHINE_MODEL:RK3326
MACHINE_ID:007
MANUFACTURER: RK3326
MAGIC: 0x5041524B
ATAG: 0x00200800
MACHINE: 3326
CHECK_MASK: 0x80
PWR_HLD: 0,0,A,0,1
TYPE: GPT
CMDLINE:mtdparts=rk29xxnand:0x00002000@0x00004000(uboot),0x00002000@0x00006000(trust),0x00002000@0x00008000(misc),0x00008000@0x0000a000(resource),0x00010000@0x0012000(kernel),0x00002000@0x00022000(dtb),0x00002000@0x00024000(dtbo),0x00000080@0x00026000(vbmeta),0x00010000@0x00026800(boot),0x00020000@0x00036800(recovery),0x00038000@0x00056800(backup),0x00002000@0x0008e800(security),0x000c0000@0x00090800(cache),0x00514000@0x00150800(system),0x00008000@0x00664800(metadata),0x000c0000@0x0066c800(vendor),0x00040000@0x0072c800(oem),0x00000400@0x0076c800(frp),-@0x0076cc00(userdata:grow)
uuid:system=af01642c-9b84-11e8-9b2a-234eb5e198a0
```

A/B System 分区定义参考：

```
FIRMWARE_VER:8.1
MACHINE_MODEL:RK3326
MACHINE_ID:007
MANUFACTURER: RK3326
MAGIC: 0x5041524B
ATAG: 0x00200800
MACHINE: 3326
CHECK_MASK: 0x80
PWR_HLD: 0,0,A,0,1
TYPE: GPT
CMDLINE:
mtdparts=rk29xxnand:0x00002000@0x00004000(uboot_a),0x00002000@0x00006000(uboot_b),0x00002000@0x00008000(trust_a),0x00002000@0x0000a000(trust_b),0x00001000@0x0000c000(misc),0x00001000@0x0000d000(vbmeta_a),0x00001000@0x0000e000(vbmeta_b),0x00020000@0x0000e000(boot_a),0x00020000@0x0002e000(boot_b),0x00100000@0x0004e000(system_a),0x00300000@0x0032e000(system_b),0x00100000@0x0062e000(vendor_a),0x00100000@0x0072e000(vendor_b),0x00002000@0x0082e000(oem_a),0x00002000@0x00830000(oem_b),0x00100000@0x00832000(factory),0x00008000@0x842000(factory_bootloader),0x000800@0x008ca000(oem),-@0x0094a000(userdata)
```

## fastboot 命令

U-Boot 下可以通过输入命令进入 fastboot：

```
fastboot usb 0
```

## 命令速览

```
fastboot flash < partition > [ < filename > ]
fastboot erase < partition >
```

```
fastboot getvar < variable > | all
fastboot set_active < slot >
fastboot reboot
fastboot reboot-bootloader
fastboot flashing unlock
fastboot flashing lock
fastboot stage [ < filename > ]
fastboot get_staged [ < filename > ]
fastboot oem fuse at-perm-attr-data
fastboot oem fuse at-perm-attr
fastboot oem fuse at-rsa-perm-attr
fastboot oem at-get-ca-request
fastboot oem at-set-ca-response
fastboot oem at-lock-vboot
fastboot oem at-unlock-vboot
fastboot oem at-disable-unlock-vboot
fastboot oem fuse at-bootloader-vboot-key
fastboot oem format
fastboot oem at-get-vboot-unlock-challenge
fastboot oem at-reset-rollback-index
```

## 命令使用

1. fastboot flash < partition > [ < filename > ]

功能：分区烧写。

例： fastboot flash boot boot.img

1. fastboot erase < partition >

功能：擦除分区。

举例： fastboot erase boot

1. fastboot getvar < variable > | all

功能：获取设备信息

举例： fastboot getvar all (获取设备所有信息)

variable 还可以带的参数：

version	/* fastboot 版本 */
version-bootloader	/* U-Boot 版本 */
version-baseband	
product	/* 产品信息 */
serialno	/* 序列号 */
secure	/* 是否开启安全校验 */
max-download-size	/* fastboot 支持单次传输最大字节数 */
logical-block-size	/* 逻辑块数 */
erase-block-size	/* 擦除块数 */
partition-type : < partition >	/* 分区类型 */
partition-size : < partition >	/* 分区大小 */
unlocked	/* 设备lock状态 */
off-mode-charge	
battery-voltage	
variant	
battery-soc-ok	
slot-count	/* slot 数目 */

```
has-slot: < partition >          /* 查看slot内是否有该分区名 */
current-slot                      /* 当前启动的slot */
slot-suffixes                      /* 当前设备具有的slot, 打印出其name */
slot-successful: < _a | _b >      /* 查看分区是否正确校验启动过 */
slot-unbootable: < _a | _b >      /* 查看分区是否被设置为unbootable */
slot-retry-count: < _a | _b >     /* 查看分区的retry-count次数 */
at-attest-dh
at-attest-uuid
at-vboot-state
```

fastboot getvar all 举例：

```
PS E:\U-Boot-AVB\adb> .\fastboot.exe getvar all
(bootloader) version:0.4
(bootloader) version-bootloader:U-Boot 2017.09-gc277677
(bootloader) version-baseband:N/A
(bootloader) product:rk3229
(bootloader) serialno:7b2239270042f8b8
(bootloader) secure:yes
(bootloader) max-download-size:0x04000000
(bootloader) logical-block-size:0x512
(bootloader) erase-block-size:0x80000
(bootloader) partition-type:bootloader_a:U-Boot
(bootloader) partition-type:bootloader_b:U-Boot
(bootloader) partition-type:tos_a:U-Boot
(bootloader) partition-type:tos_b:U-Boot
(bootloader) partition-type:boot_a:U-Boot
(bootloader) partition-type:boot_b:U-Boot
(bootloader) partition-type:system_a:ext4
(bootloader) partition-type:system_b:ext4
(bootloader) partition-type:vbmeta_a:U-Boot
(bootloader) partition-type:vbmeta_b:U-Boot
(bootloader) partition-type:misc:U-Boot
(bootloader) partition-type:vendor_a:ext4
(bootloader) partition-type:vendor_b:ext4
(bootloader) partition-type:oem_bootloader_a:U-Boot
(bootloader) partition-type:oem_bootloader_b:U-Boot
(bootloader) partition-type:factory:U-Boot
(bootloader) partition-type:factory_bootloader:U-Boot
(bootloader) partition-type:oem_a:ext4
(bootloader) partition-type:oem_b:ext4
(bootloader) partition-type:userdata:ext4
(bootloader) partition-size:bootloader_a:0x400000
(bootloader) partition-size:bootloader_b:0x400000
(bootloader) partition-size:tos_a:0x400000
(bootloader) partition-size:tos_b:0x400000
(bootloader) partition-size:boot_a:0x2000000
(bootloader) partition-size:boot_b:0x2000000
(bootloader) partition-size:system_a:0x20000000
(bootloader) partition-size:system_b:0x20000000
(bootloader) partition-size:vbmeta_a:0x10000
(bootloader) partition-size:vbmeta_b:0x10000
(bootloader) partition-size:misc:0x10000
(bootloader) partition-size:vendor_a:0x4000000
(bootloader) partition-size:vendor_b:0x4000000
```

```
(bootloader) partition-size:oem_bootloader_a:0x400000
(bootloader) partition-size:oem_bootloader_b:0x400000
(bootloader) partition-size:factory:0x2000000
(bootloader) partition-size:factory_bootloader:0x1000000
(bootloader) partition-size:oem_a:0x10000000
(bootloader) partition-size:oem_b:0x10000000
(bootloader) partition-size:userdata:0x7ad80000
(bootloader) unlocked:no
(bootloader) off-mode-charge:0
(bootloader) battery-voltage:0mv
(bootloader) variant:rk3229_evb
(bootloader) battery-soc-ok:no
(bootloader) slot-count:2
(bootloader) has-slot:bootloader:yes
(bootloader) has-slot:tos:yes
(bootloader) has-slot:boot:yes
(bootloader) has-slot:system:yes
(bootloader) has-slot:vbmeta:yes
(bootloader) has-slot:misc:no
(bootloader) has-slot:vendor:yes
(bootloader) has-slot:oem_bootloader:yes
(bootloader) has-slot:factory:no
(bootloader) has-slot:factory_bootloader:no
(bootloader) has-slot:oem:yes
(bootloader) has-slot:userdata:no
(bootloader) current-slot:a
(bootloader) slot-suffixes:a,b
(bootloader) slot-successful:a:yes
(bootloader) slot-successful:b:no
(bootloader) slot-unbootable:a:no
(bootloader) slot-unbootable:b:yes
(bootloader) slot-retry-count:a:0
(bootloader) slot-retry-count:b:0
(bootloader) at-attest-dh:1:P256
(bootloader) at-attest-uuid:
all: Done!
finished. total time: 0.636s
```

### 1. fastboot set\_active < slot >

功能：设置重启的 slot。

举例：fastboot set\_active \_a

### 1. fastboot reboot

功能：重启设备，正常启动

举例：fastboot reboot

### 1. fastboot reboot-bootloader

功能：重启设备，进入 fastboot 模式

举例：fastboot reboot-bootloader

### 1. fastboot flashing unlock

功能：解锁设备，允许烧写固件

举例：fastboot flashing unlock

1. fastboot flashing lock

功能：锁定设备，禁止烧写

举例：fastboot flashing lock

1. fastboot stage [<filename>]

功能：下载数据到设备端内存，内存起始地址为 CONFIG\_FASTBOOT\_BUF\_ADDR

举例：fastboot stage permanent\_attributes.bin

1. fastboot get\_staged [<filename>]

功能：从设备端获取数据

举例：fastboot get\_staged raw\_unlock\_challenge.bin

1. fastboot oem fuse at-perm-attr

功能：烧写 permanent\_attributes.bin 及 hash

举例：fastboot stage permanent\_attributes.bin

fastboot oem fuse at-perm-attr

1. fastboot oem fuse at-perm-attr-data

功能：只烧写 permanent\_attributes.bin 到安全存储区域 (RPMB)

举例：fastboot stage permanent\_attributes.bin

fastboot oem fuse at-perm-attr-data

1. fastboot oem at-get-ca-request

2. fastboot oem at-set-ca-response

3. fastboot oem at-lock-vboot

功能：锁定设备

举例：fastboot oem at-lock-vboot

1. fastboot oem at-unlock-vboot

功能：解锁设备，现支持 authenticated unlock

举例：fastboot oem at-get-vboot-unlock-challenge

fastboot get\_staged raw\_unlock\_challenge.bin

./make\_unlock.sh (见 make\_unlock.sh 参考)

fastboot stage unlock\_credential.bin

fastboot oem at-unlock-vboot

1. fastboot oem fuse at-bootloader-vboot-key

功能：烧写 bootloader key hash

举例：fastboot stage bootloader-pub-key.bin

fastboot oem fuse at-bootloader-vboot-key

1. fastboot oem format

功能：重新格式化分区，分区信息依赖于\$partitions

举例：fastboot oem format

1. fastboot oem at-get-vboot-unlock-challenge

功能：authenticated unlock，需要获得 unlock challenge 数据

举例：参见 16. fastboot oem at-unlock-vboot

1. fastboot oem at-reset-rollback-index

功能：复位设备的 rollback 数据

举例：fastboot oem at-reset-rollback-index

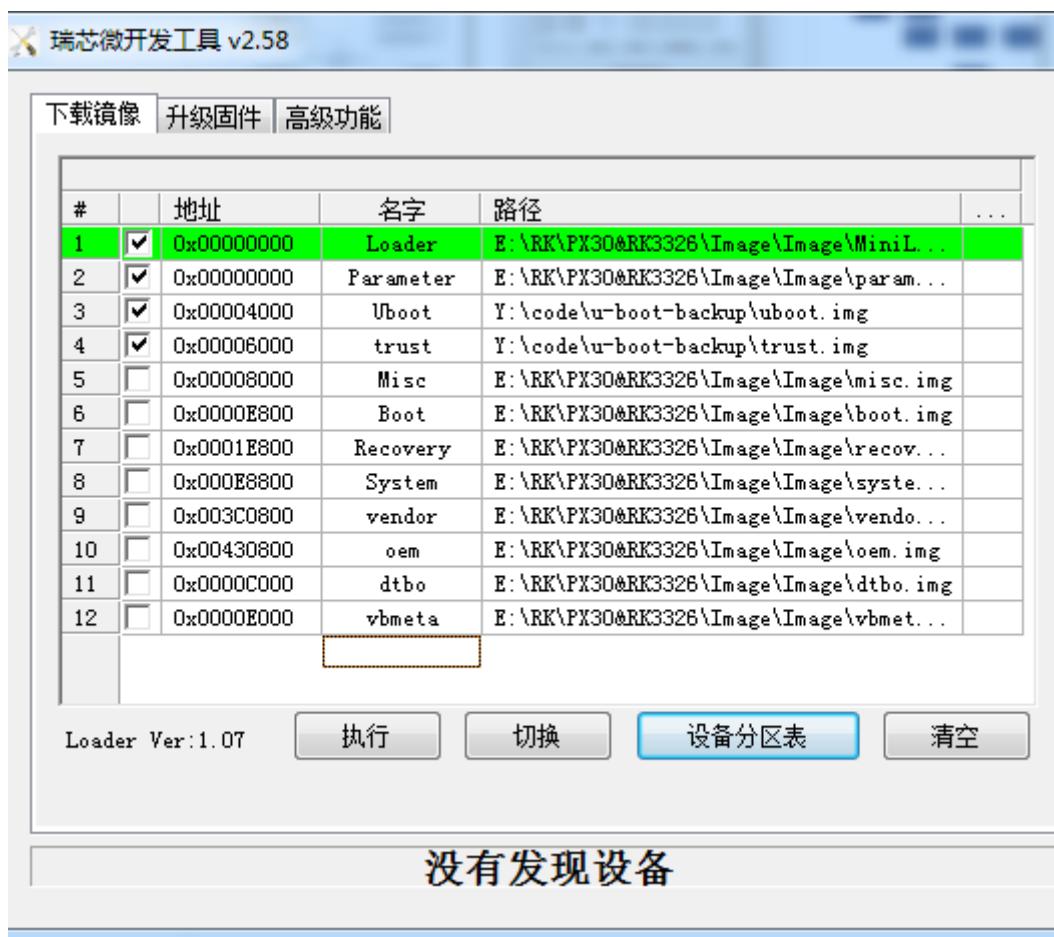
1. fastboot oem at-disable-unlock-vboot

功能：使 fastboot oem at-unlock-vboot 命令失效

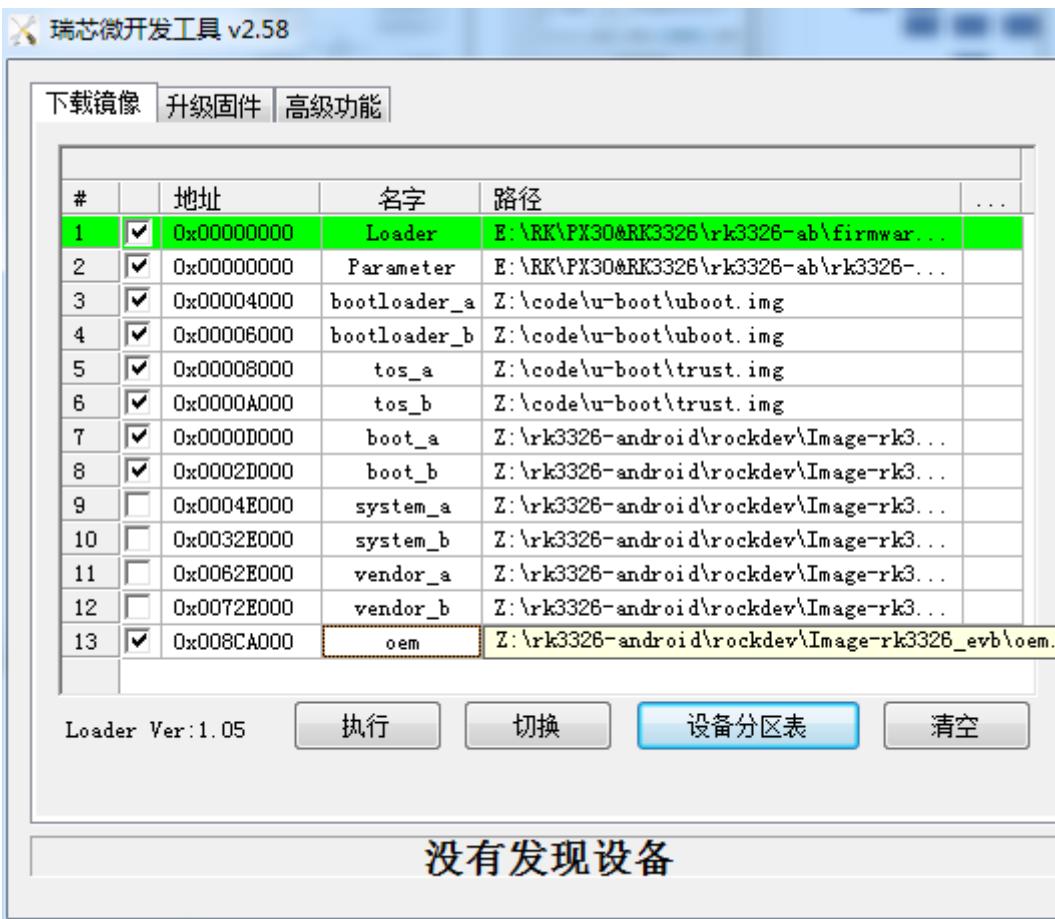
举例：fastboot oem at-disable-unlock-vboot

## 固件烧写

如下是windows固件烧写工具：

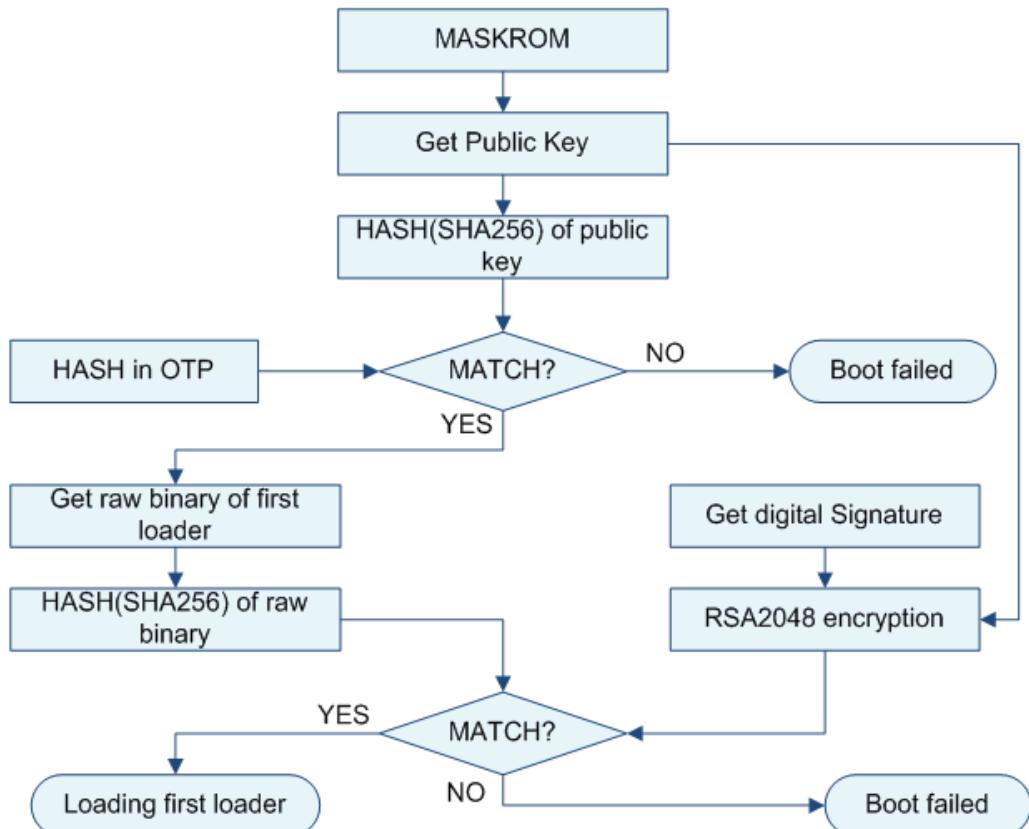


A/B System 烧写



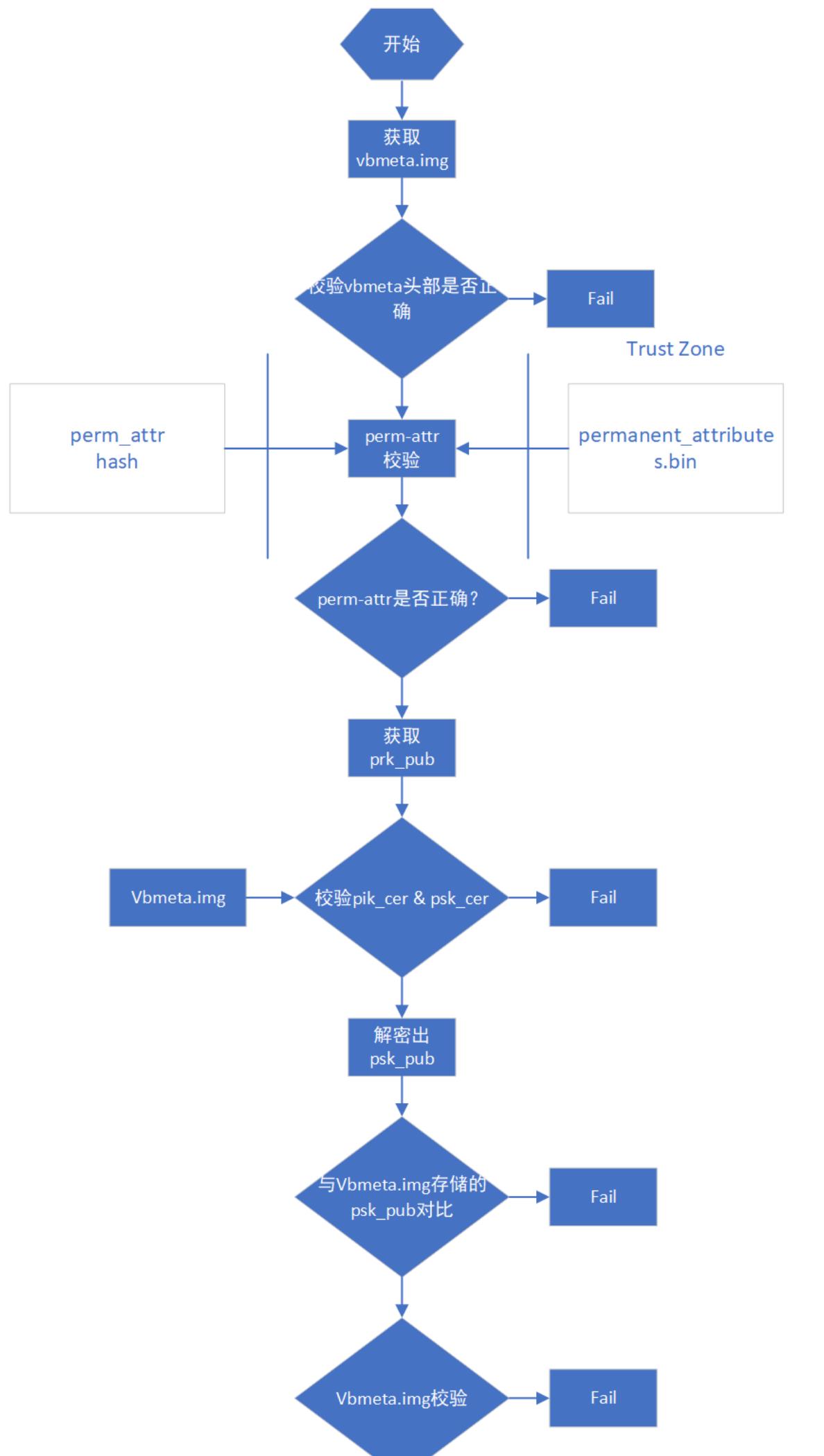
## Pre-loader verified

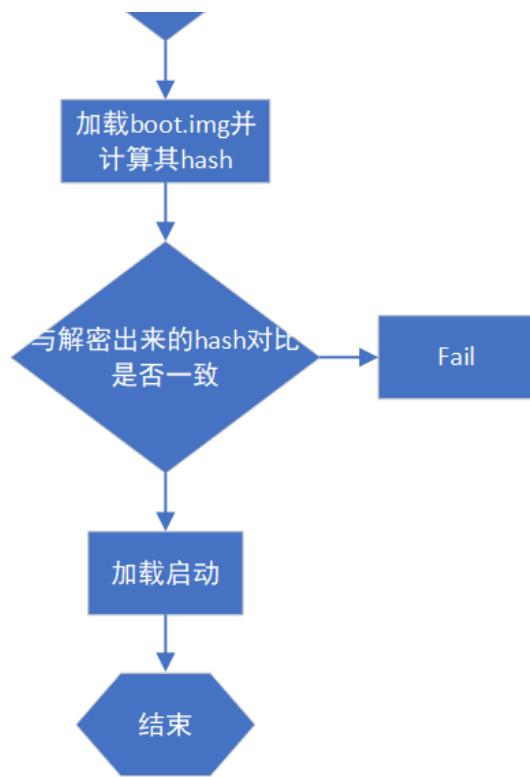
参见《Rockchip-Secure-Boot-Application-Note.md》



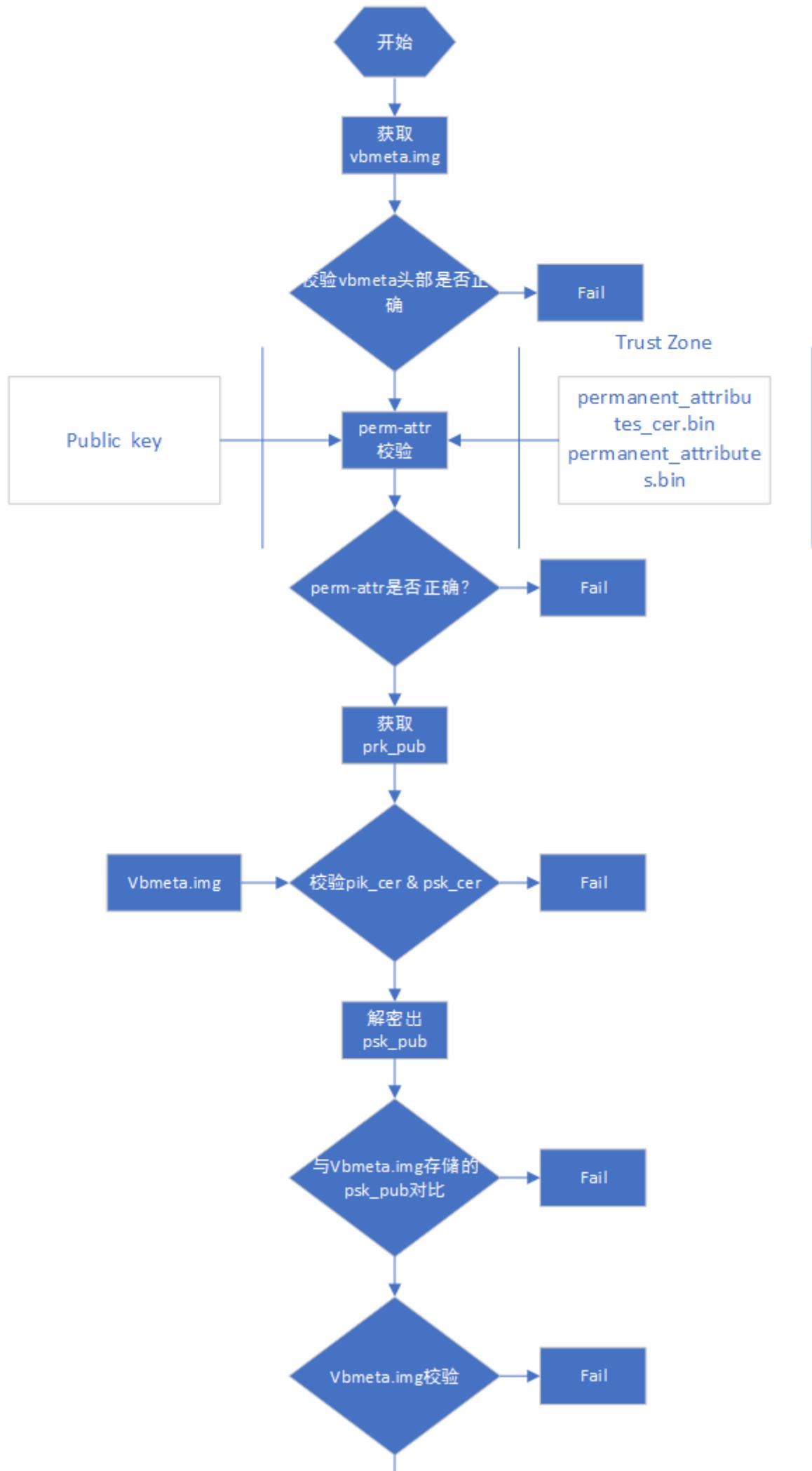
## U-boot verified

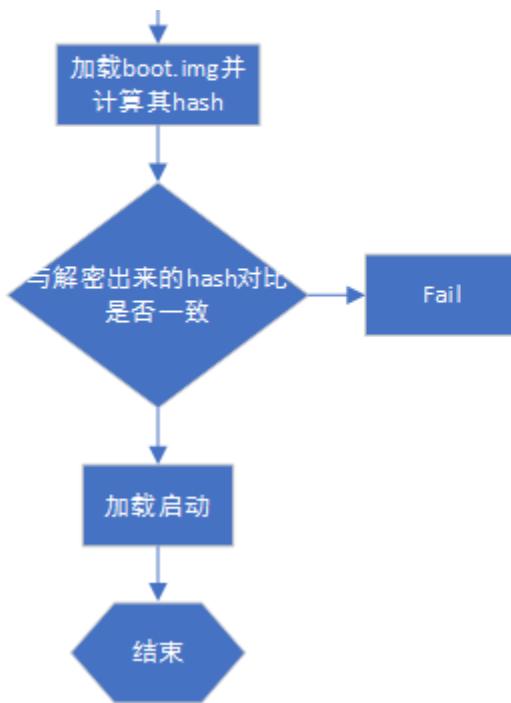
OTP 设备校验流程:



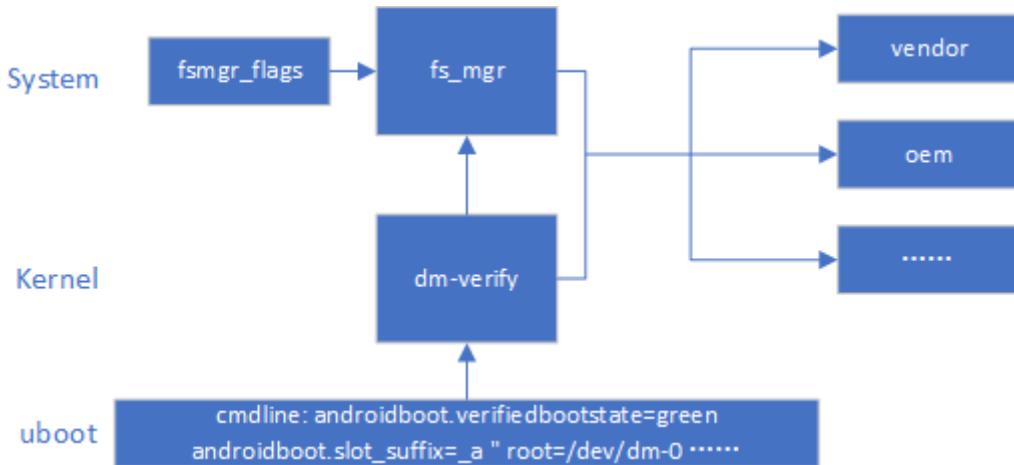


efuse设备校验流程：





## 系统校验启动



系统启动到 kernel, kernel 首先解析 U-Boot 传递的 cmdline 参数, 确认系统启动是否使用 dm-verify。然后加载启用 system 的 fs\_mgr 服务。fs\_mgr 依据 fsmgr\_flags 的参数来校验加载固件, 固件 hash & hash tree 存放于 vbmeta.img。主要有如下参数:

avb: 使用 avb 的方式加载校验分区

slotselect: 该分区分 A/B, 加载时会使用到 cmdline 的"androidboot.slot\_suffix=\_a"这个参数。

## Linux AVB

如下介绍基于linux 环境的 AVB 操作及验证流程。

### 操作流程

- 生成整套固件
- 使用 SecureBootConsole 生成 PrivateKey.pem 与 PublicKey.pem, 工具为 rk\_sign\_tool, 命令如下:

```

rk_sign_tool cc --chip 3399
rk_sign_tool kk --out .
  
```

- load key

```
rk_sign_tool lk --key privateKey.pem --pubkey publicKey.pem
```

#### 4. 签名 loader

```
rk_sign_tool sl --loader loader.bin
```

#### 5. 签名 uboot.img & trust.img

```
rk_sign_tool si --img uboot.img  
rk_sign_tool si --img trust.img
```

#### 6. avb 签名固件准备：准备空的 temp.bin，16 字节的 product\_id.bin，待签名的 boot.img，运行下列代码

```
#!/bin/bash  
touch temp.bin  
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -outform PEM -out testkey_prk.pem  
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -outform PEM -out testkey_psk.pem  
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:4096 -outform PEM -out testkey_pik.pem  
python avbtool make_atx_certificate --output=pik_certificate.bin --  
subject=temp.bin --subject_key=testkey_pik.pem --  
subject_is_intermediate_authority --subject_key_version 42 --  
authority_key=testkey_prk.pem  
python avbtool make_atx_certificate --output=psk_certificate.bin --  
subject=product_id.bin --subject_key=testkey_psk.pem --subject_key_version 42 --  
authority_key=testkey_pik.pem  
python avbtool make_atx_metadata --output=metadata.bin --  
intermediate_key_certificate=pik_certificate.bin --  
product_key_certificate=psk_certificate.bin  
python avbtool make_atx_permanent_attributes --output=permanent_attributes.bin --  
-product_id=product_id.bin --root_authority_key=testkey_prk.pem  
python avbtool add_hash_footer --image boot.img --partition_size 33554432 --  
partition_name boot --key testkey_psk.pem --algorithm SHA256_RSA4096  
python avbtool make_vbmeta_image --public_key_metadata metadata.bin --  
include_descriptors_from_image boot.img --algorithm SHA256_RSA4096 --key  
testkey_psk.pem --output vbmeta.img  
openssl dgst -sha256 -out permanent_attributes_cer.bin -sign PrivateKey.pem  
permanent_attributes.bin
```

生成 vbmeta.img, permanent\_attributes\_cer.bin, permanent\_attributes.bin。

该步骤就签名了 boot.img.....

#### 7. 固件烧写

```
rkdevelopool db loader.bin  
rkdevelopool ul loader.bin  
rkdevelopool gpt parameter.txt  
rkdevelopool wlx uboot uboot.img  
rkdevelopool wlx trust trust.img  
rkdevelopool wlx boot boot.img  
rkdevelopool wlx system system.img
```

rkdevelop tool 可以参考<https://github.com/rockchip-linux/rkdevelop>

1. 烧写 permanent\_attributes\_cer.bin, permanent\_attributes.bin

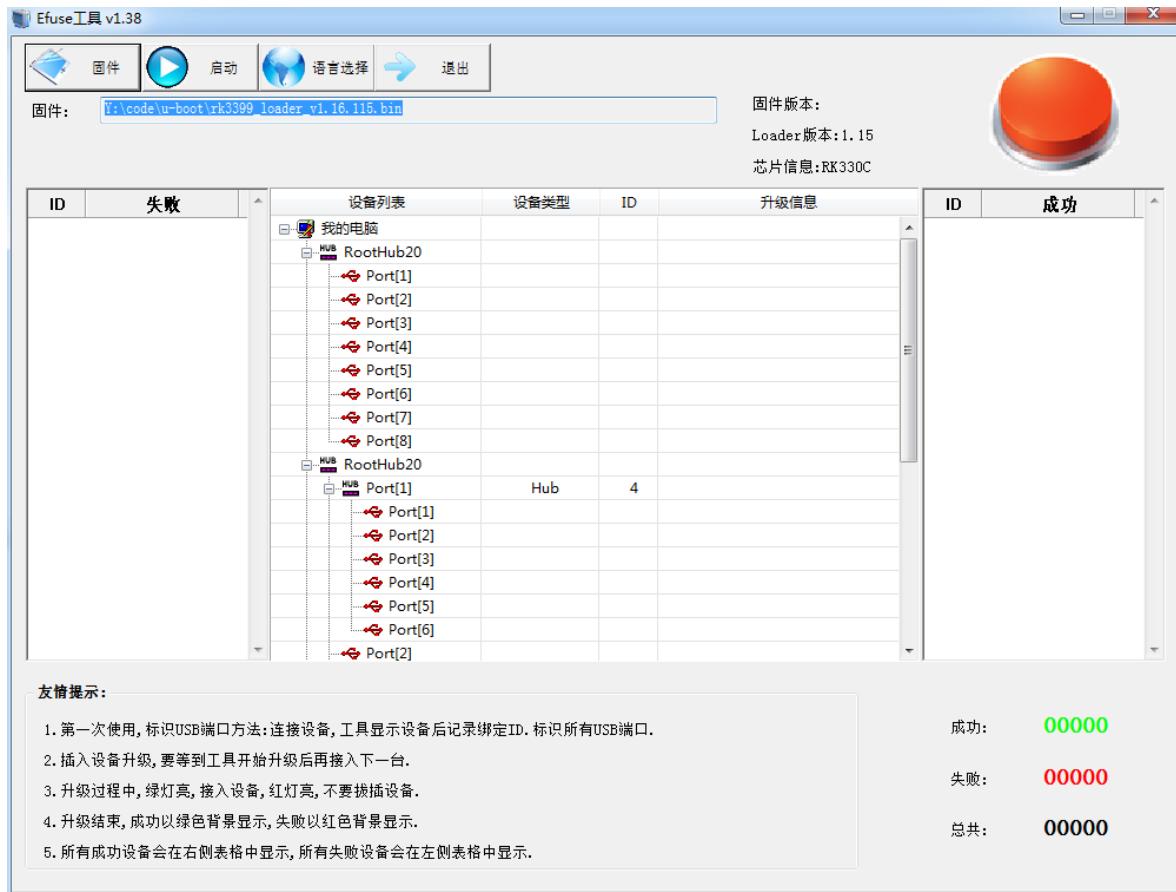
有OTP平台:

```
fastboot stage permanent_attributes.bin  
fastboot oem fuse at-perm-attr
```

有 efuse 平台:

```
fastboot stage permanent_attributes.bin  
fastboot oem fuse at-perm-attr  
fastboot stage permanent_attributes_cer.bin  
fastboot oem fuse at-rsa-perm-attr
```

1. efuse 烧写 (efuse 工具目前只有 windows 版本) , 选择特定的 loader, 选择对应的设备, 点击启动烧写。



1. OTP 平台 loader public key烧写

参考《Rockchip-Secure-Boot-Application-Note.md》

**验证流程**

[TODO]

## SD启动和升级

### 简介

Rockchip 现将 SD 卡划分为常规 SD 卡, SD 升级卡, SD 启动卡, SD 修复卡。可以通过瑞芯微创建升级磁盘工具将 update.img 下载到 SD 卡内, 制作不同的卡类型。

卡类型	功能
常规 SD 卡	普通的存储设备
SD 升级卡	设备从 SD 卡内启动到 recovery, 由 recovery 负责把 sd 内固件更新到设备存储
SD 启动卡	设备直接从 SD 卡启动
SD 修复卡	从 pre-loader 开始拷贝 SD 卡内的固件到设备存储

## 分类

### 常规卡

普通 SD 卡与 PC 使用完全一样, 可以在 U-Boot 和 Kernel 系统中作为普通的存储空间使用, 无需工具对 SD 卡做任何操作。

### 升级卡

SD 升级卡是通过 RK 的工具制作, 实现 SD 卡对本地存储(如 eMMC, nand flash)固件的升级。SD 卡升级是可以脱离 PC 机或网络的一种固件升级方法。具体是将 SD 卡启动代码写到 SD 卡的保留区, 然后将固件拷贝到 SD 卡可见分区上, 主控从 SD 卡启动时, SD 卡启动代码和升级代码将固件烧写到本地主存储中。同时 SD 升级卡支持 PCBA 测试和 Demo 文件的拷贝。SD 升级卡的这些功能可以使固件升级做到脱离 PC 机进行, 提高生产效率。

已经制作好的升级用 SD 卡, 如果只需要更新固件和 demo 文件时, 可以按下面步骤来完成:

1. 拷贝固件到 SD 卡根目录, 并重命名为 sdupdate.img
2. 拷贝 demo 文件到 SD 卡根目录下的 Demo 目录中

### SD 引导升级卡格式(非 GPT)

偏移	数据段
扇区 0	MBR
扇区 64-4M	IDBLOCK(启动标志置 0)
4M-8M	Parameter
12M-16M	uboot
16M-20M	trust
.....	misc
.....	resource
.....	kernel
.....	recovery
剩下空间	Fat32 存放 update.img

### SD 引导升级卡格式(GPT)

偏移	数据段
扇区 0	MBR
扇区 1-34	GPT 分区表
扇区 64-4M	IDBLOCK(启动标志置 0)
4M-8M	Parameter
.....	uboot
.....	trust
.....	misc
.....	resource
.....	kernel
.....	recovery
剩下空间	Fat32 存放 update.img

## 启动卡

SD 启动卡是通过 RK 的工具制作，实现直接从 SD 卡启动，极大的方便用户更新启动新固件而不用重新烧写固件到设备存储内。具体实现是将固件烧写到 SD 卡中，把 SD 卡当作主存储使用。主控从 SD 卡启动时，固件以及临时文件都存放在 SD 卡上，有没有本地主存储都可以正常工作。目前主要用于设备系统从 SD 卡启动，或用于 PCBA 测试。**注意：**PCBA 测试只是 recovery 下面的一个功能项，可用于升级卡与启动卡。

SD 引导启动卡格式(非 GPT)

偏移	数据段
扇区 0	MBR
扇区 64-4M	IDBLOCK(启动标志置 1)
4M-8M	Parameter
8M-12M	uboot
12M-16M	trust
.....	misc
.....	resource
.....	boot
.....	kernel
.....	recovery
.....	system
.....	user

## SD 引导启动卡格式(GPT)

偏移	数据段
扇区 0	MBR
扇区 1-34	GPT 分区表
扇区 64-4M	IDBLOCK(启动标志置 1)
.....	uboot
.....	Boot
.....	trust
.....	resource
.....	kernel
.....	recovery
.....	system
.....	vendor
.....	oem
.....	user
最后 33 扇区	备份 GPT

## 修复卡

SD 修复卡类似于 SD 卡升级功能，但固件升级工作由 miniloader 完成。首先工具会将启动代码写到 SD 卡的保留区，然后将固件拷贝到 SD 卡可见分区上，主控从 SD 卡启动时，SD 卡升级代码将固件升级到本地主存储中。主要用于设备固件损坏，SD 卡可以修复设备。

## SD 修复卡格式(非 GPT)

<b>偏移</b>	<b>数据段</b>
扇区 0	MBR
扇区 64-4M	IDBLOCK(启动标志置 2)
4M-8M	Parameter
8M-12M	uboot
12M-16M	trust
.....	misc
.....	resource
.....	boot
.....	kernel
.....	recovery
.....	system
.....	user

### SD 修复卡格式(GPT)

<b>偏移</b>	<b>数据段</b>
扇区 0	MBR
扇区 1-34	GPT 分区表
扇区 64-4M	IDBLOCK(启动标志置 2)
.....	uboot
.....	Boot
.....	trust
.....	resource
.....	kernel
.....	recovery
.....	system
.....	vendor
.....	oem
.....	user
最后 33 扇区	备份 GPT

### 固件标志

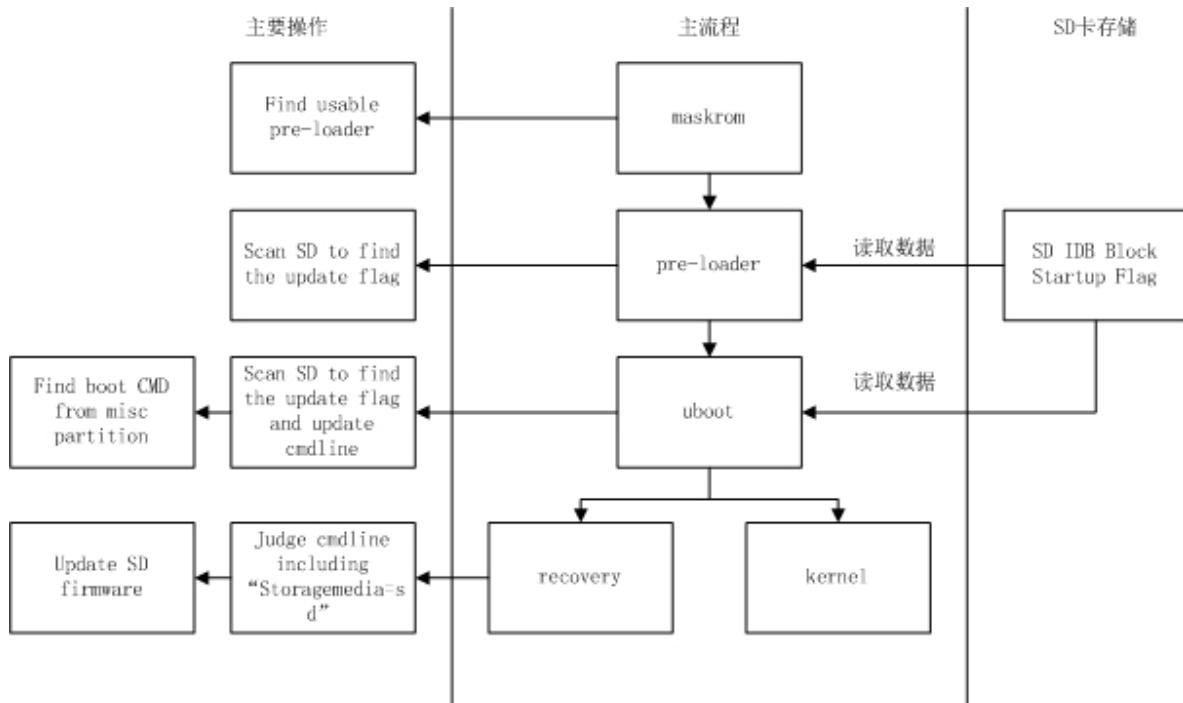
SD 卡作为各种不同功能的卡，会在 sd 卡内做一些标志。

在 SD 卡的第 64 扇区处，起始标志若为 (magic number) 为 0xFCDC8C3B，则为一些特殊卡，会从 SD 卡内读取固件，启动设备。如果不是，则作为普通 SD 卡看待。在第 (64 扇区 + 616bytes) 地方，存放各种卡的标志。目前有三种类型：

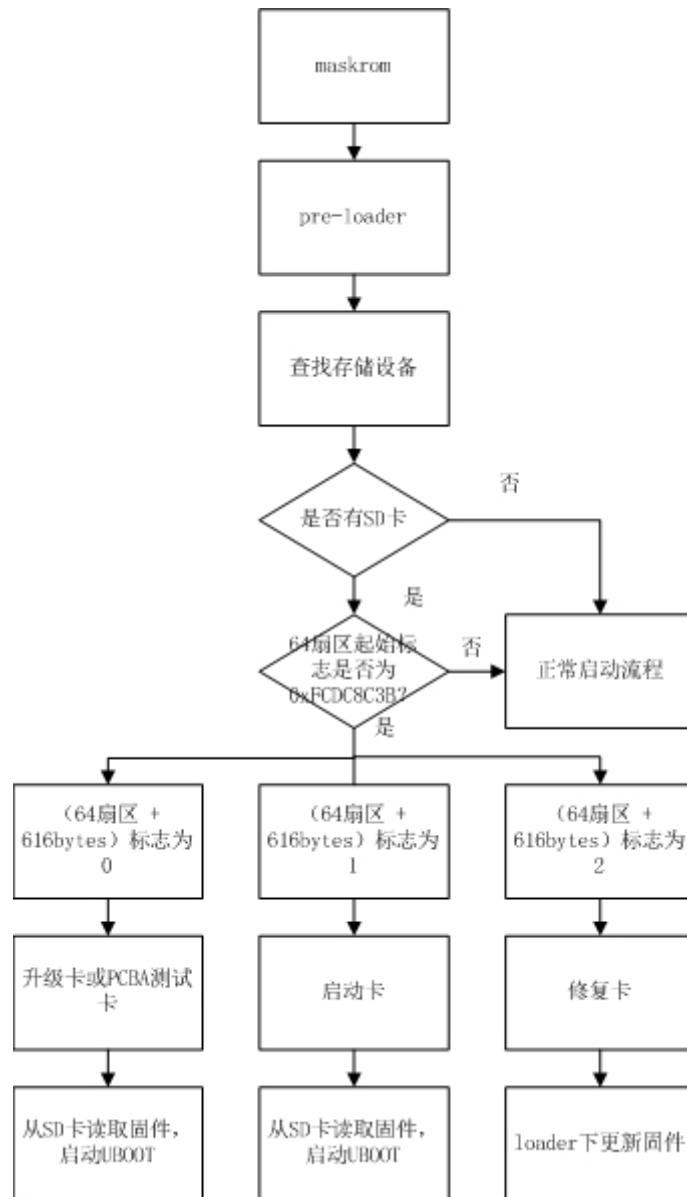
标志	卡类型
0	升级卡或 PCBA 测试卡
1	启动卡
2	修复卡

## 启动流程

SD 卡的 boot 流程可分为 pre-loader 启动流程与 uboot 启动流程，这两个流程都需要加载检测 SD 卡及 SD 卡内 IDB Block 内 Startup Flag 标志，并且会依据这些标志执行不同的功能。流程如下：

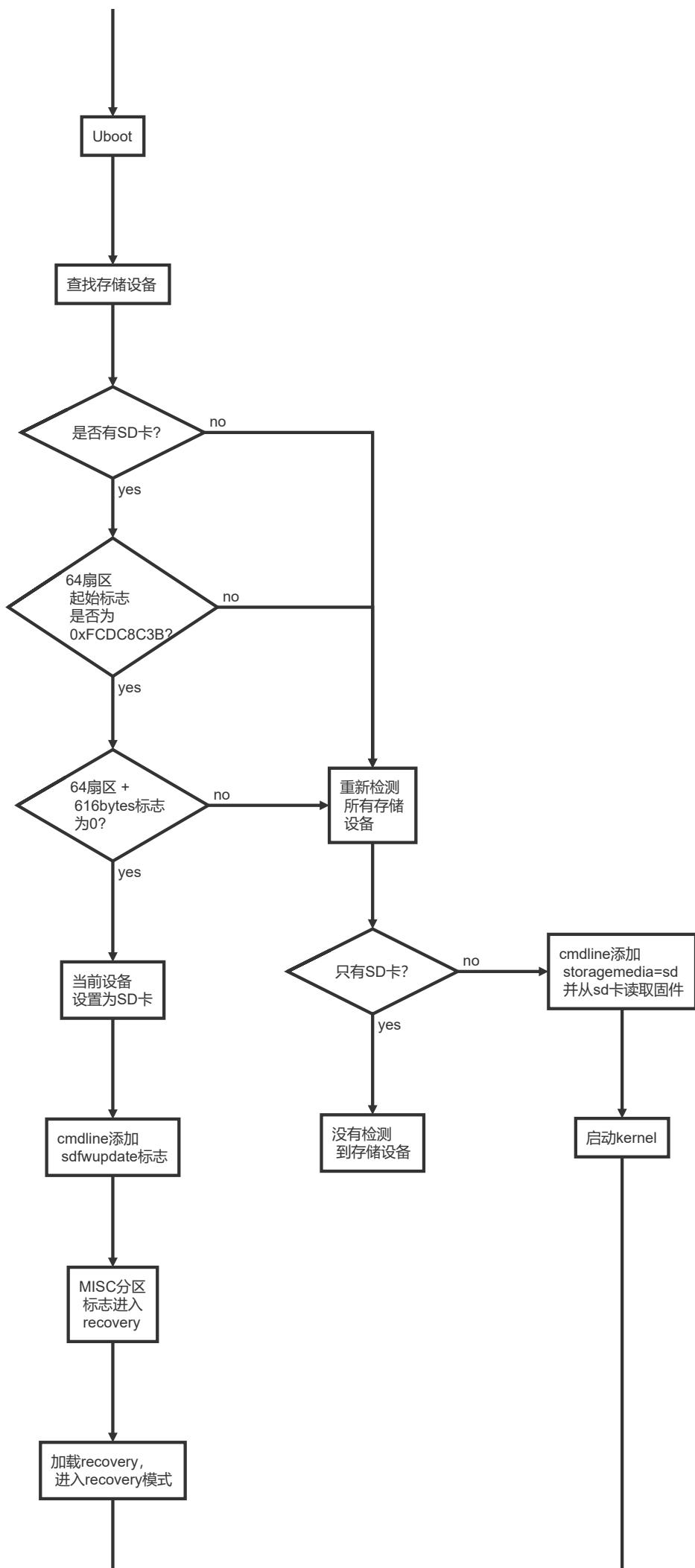


### pre-loader 启动



maskrom 首先先找到一份可用的 miniloader 固件（可以从 TRM 确定 Maskrom 支持的启动存储介质和优先顺序，maskrom 会依次扫描可用存储里的固件），然后跳转到 miniloader。miniloader 重新查找存储设备，如果检测到 SD 卡，检测 SD 卡是否包含 IDB 格式固件。如果是，再判断卡标志。如果 SD 卡可用且标志位为 '0' 或 '1'，则从 SD 卡内读取 U-Boot 固件，加载启动 U-Boot。如果标志为 '2'，则进入修复卡流程，在 loader 下更新固件。正常启动流程为扫描其他存储，加载启动下级 loader。

## U-Boot 启动





升级卡：U-Boot 重新查找存储设备，如果检测到 SD 卡，检测 SD 卡是否包含 IDB 格式固件。如果是，再判断偏卡标志是否为 0，传递给 kernel 的 cmdline 添加'sdfwupdate'。最后读取 SD 卡的 misc 分区，读取卡启动模式，若为 recovery 模式，加载启动 recovery。

启动卡：U-Boot 重新查找存储设备，如果检测到 SD 卡，检测 SD 卡是否包含 IDB 格式固件。如果是，再判断卡标志是否为 1。最后读取 SD 卡的 misc 分区，读取卡启动模式，如果为 recovery，加载启动 recovery。如果是 normal 模式，则加载启动 kernel。

### Recovery和PCBA

具体可参考《Rockchip Recovery 用户操作指南 V1.03.pdf》。

### 注意事项

- 制作非GPT 格式固件时，U-Boot 需要配置 CONFIG\_RKPARM\_PARTITION。
- 制作 SD 升级卡时，update.img 必须包含 MiniloaderAll.bin, parameter.txt, uboot.img, trust.img, misc.img, resource.img, recovery.img 这些固件，否则烧写 update.img 会出现写入 MBR 失败的提示。

---

## Chapter-7 配置裁剪

---

TODO

---

## Chapter-8 调试手段

---

本章节主要介绍一些U-Boot阶段常用的调试手段，包括使用命令、脚本、配置选项、开机打印等等。

### DEBUG

功能：让全局 `debug()` 打印生效。

可在各平台的 `rkxxx_common.h` 中增加宏定义进行使能：

```
#define DEBUG
```

### Initcall

功能：打印启动流程。

U-Boot 的启动实质是一系列initcall调用，把 `initcall_run_list()` 函数内的 `debug()` 改成 `printf()`。例：

```
U-Boot 2017.09-01725-g03b8d3b-dirty (Jul 06 2018 - 10:08:27 +0800)

initcall: 0000000000214388
initcall: 0000000000214724
Model: Rockchip RK3399 Evaluation Board
initcall: 0000000000214300
DRAM: initcall: 0000000000203f68
initcall: 0000000000214410 // 结合反汇编找出地址对应的函数
initcall: 00000000002140dc
....
3.8 GiB
initcall: 00000000002143b8
....
Relocation Offset is: f5c03000
initcall: 00000000f5e176bc
initcall: 00000000002146a4 (relocated to 00000000f5e176a4)
initcall: 0000000000214668 (relocated to 00000000f5e17668)

...
```

## io命令

功能：读写内存。

```
// 读操作
md - memory display
Usage: md [.b, .w, .l, .q] address [# of objects]

// 写操作
mw - memory write (fill)
Usage: mw [.b, .w, .l, .q] address value [count]
```

读操作。范例：显示 0x76000000 地址开始的连续 0x10 个数据。

```
=> md.l 0x76000000 0x10
76000000: ffffffe ffffffff ffffffff ffffffff ..... .
76000010: ffffffdf ffffffff feffffff ffffffff ..... .
76000020: ffffffff ffffffff ffffffff ffffffff ..... .
76000030: ffffffff ffffffff ffffffff ffffffff ..... .
```

写操作。范例：对 0x76000000 地址赋值为 0x1234；

```
=> mw.l 0x76000000 0xffff1234 // 高16位有mask
=> md.l 0x76000000 0x10 // 回读
76000000: fffff1234 ffffffff ffffffff ffffffff ..... .
76000010: ffffffdf ffffffff feffffff ffffffff ..... .
76000020: ffffffff ffffffff ffffffff ffffffff ..... .
76000030: ffffffff ffffffff ffffffff ffffffff ..... .
```

## iomem命令

功能：读内存。比md命令更灵活，通过自动解析DTS节点获取基址信息。

```
=> iomem
iomem - Show iomem data by device compatible

Usage:
// @<compatible>: 节点的compatible部分关键词匹配
iomem <compatible> <start offset> <end offset>
eg: iomem -grf 0x0 0x200
```

范例：RK3228 读取 GRF 里 0x00 ~ 0x20 的数据：

```
// 这里为了和"rockchip, rk3228-pmugrf" 区分开，所以用"-grf"作为关键词
=> iomem -grf 0x0 0x20
rockchip,rk3228-grf:
11000000: 00000000 00000000 00004000 00002000
11000010: 00000000 00005028 0000a5a5 0000aaaa
11000020: 00009955
```

## i2c命令

功能：i2c设备读写。

```
=> i2c
i2c - I2C sub-system

Usage:
i2c dev [dev] - show or set current I2C bus
i2c md chip address[.0, .1, .2] [# of objects] - read from I2C device
i2c mw chip address[.0, .1, .2] value [count] - write to I2C device (fill)
.....
```

读操作。范例：

```
=> i2c dev 0 // 切到i2c0 (指定一次即可)
Setting bus to 0

=> i2c md 0x1b 0x2e 0x20 // i2c设备地址为1b(7位地址)，读取0x2e开始的连续0x20个
寄存器值
002e: 11 0f 00 00 11 0f 00 00 01 00 00 00 09 00 00 0c ..... .
003e: 00 0a 0a 0c 0c 0c 00 07 07 0a 00 0c 0c 00 00 00 .....
```

写操作。范例：

```
=> i2c dev 0 // 切到i2c0 (指定一次即可)
Setting bus to 0

=> i2c mw 0x1b 0x2e 0x10 // i2c设备地址为1b(7位地址)，对0x2e寄存器赋值为0x10
=> i2c md 0x1b 0x2e 0x20 // 回读
002e: 10 0f 00 00 11 0f 00 00 01 00 00 00 09 00 00 0c ..... .
003e: 00 0a 0a 0c 0c 0c 00 07 07 0a 00 0c 0c 00 00 00 .....
```

## gpio命令

功能：pin脚输入输出读写

```
=> gpio
gpio - query and control gpio pins

Usage:
gpio <input|set|clear|toggle> <pin>
    - input/set/clear/toggle the specified pin
gpio status [-a] [<bank> | <pin>] - show [all/claimed] GPIOs
```

查看pin脚状态：如RV1126

```
=> gpio status -a
Bank A:
A0: input: 0 []
A1: output: 1 []
A2: input: 1 []
...
A29: unused: 1 []
A30: unknown
A31: unused: 0 []
...
D6: input: 0 []
D7: output: 1 [x] vcc18-lcd-n.gpio
...
D31: input: 0 []

Bank E:
E0: input: 0 []
E1: input: 0 []
```

pin输入：

```
=> gpio input A7
```

pin输出INACTIVE：

```
=> gpio clear A7
```

pin输出ACTIVE：

```
=> gpio set A7
```

pin状态切换：如A7: input: 0 改为 A7: output: 1

```
=> gpio toggle A7
```

## fdt命令

功能：打印DTB内容。

```
=> fdt
fdt - flattened device tree utility commands

Usage:
fdt addr [-c] <addr> [<length>]      - Set the [control] fdt location to <addr>
fdt print <path> [<prop>]            - Recursive print starting at <path>
fdt list   <path> [<prop>]            - Print one level starting at <path>
.....
NOTE: Dereference aliases by omitting the leading '/', e.g. fdt print ethernet0.
```

其中如下两条组合命令一起使用，可以把 device-tree 完整 dump 出来：

```
=> fdt addr $fdt_addr_r // 指定fdt地址
=> fdt print           // 把fdt内容全部打印出来
```

## mmc命令

功能：MMC 设备读写、切换。

MMC 设备查看：

```
=> mmc info
Device: dwmmc@ff0f0000          // 设备节点
Manufacturer ID: 15
OEM: 100
Name: 8GME4
Timing Interface: High Speed    // 速度模式
Tran Speed: 52000000            // 当前速度
Rd Block Len: 512
MMC version 5.1
High Capacity: Yes
Capacity: 7.3 GiB                // 存储容量
Bus Width: 8-bit                 // 总线宽度
Erase Group Size: 512 KiB
HC WP Group Size: 8 MiB
User Capacity: 7.3 GiB WRREL
Boot Capacity: 4 MiB ENH
RPMB Capacity: 512 KiB ENH
```

MMC 设备切换：

```
=> mmc dev 0                   // 切换到eMMC
=> mmc dev 1                   // 切换到sd卡
```

MMC 设备读写：

```

mmc read addr blk# cnt
mmc write addr blk# cnt
mmc erase blk# cnt
例:
=> mmc read 0x70000000 0 1      // 读取MMC设备第一个block, 大小为1 sector的数据到内存
0x70000000
=> mmc write 0x70000000 0 1    // 把内存0x70000000起1 sector的数据写到存储第一个block
起位置
=> mmc erase 0 1                // 擦除存储第一个block起1 sector数据

```

如果 MMC 设备读写异常，可以通过以下简单步骤快速定位：

把 drivers/mmc/dw\_mmc.c 内的 `debug()` 改为 `printf()` 后重新编译烧写。查看 MMC 设备的打印信息：

- 如果最后的打印为 Sending CMD0, 请检查硬件供电、管脚连接；检查软件 IOMUX 是否被其他 IP 切走；
- 如果最后打印为 Sending CMD8, 安全软件部分请配置 MMC 设备允许访问安全存储；
- 如果初始化命令都已通过，最后打印为 Sending CMD18, 请检查 MMC 硬件供电、靠近 MMC 供电端的电容是否足够（可以更换大电容）、软件可以降低时钟频率、切换 MMC 设备的速度模式。

## 时间戳

功能：给U-Boot 打印信息增加时间戳（相对时间）。

CONFIG\_BOOTSTAGE\_PRINTF\_TIMESTAMP

范例：

```

[ 0.259266] U-Boot 2017.09-01739-g856f373-dirty (Jul 10 2018 - 20:26:05
+0800)
[ 0.260596] Model: Rockchip RK3399 Evaluation Board
[ 0.261332] DRAM: 3.8 GiB
Relocation offset is: f5bfd000
Using default environment

[ 0.354038] dwmmc@fe320000: 1, sdhci@fe330000: 0
[ 0.521125] Card did not respond to voltage select!
[ 0.521188] mmc_init: -95, time 9
[ 0.671451] switch to partitions #0, OK
[ 0.671500] mmc0(part 0) is current device
[ 0.675507] boot mode: None
[ 0.683738] DTB: rk-kernel.dtb
[ 0.706940] Using kernel dtb
.....

```

时间戳仅是把当前系统timer的时间打印出来，而不是从0开始计时。所以时间戳打印的仅仅是相对时间，而不是绝对时间。

## dm tree

功能：查看所有 device-driver 之间的绑定、probe状态。

=> dm tree

Class	Probed	Driver	Name
<hr/>			
root	[ + ]	root_driver	root_driver
syscon	[ ]	rk322x_syscon	-- syscon@11000000
serial	[ + ]	ns16550_serial	-- serial@11030000 *
clk	[ + ]	clk_rk322x	-- clock-controller@110e0000
sysreset	[ ]	rockchip_sysreset	-- sysreset
reset	[ ]	rockchip_reset	'-- reset
mmc	[ + ]	rockchip_rk3288_dw_mshc	-- dwmmc@30020000 *
blk	[ + ]	mmc_blk	'-- dwmmc@30020000.blk *
ram	[ ]	rockchip_rk322x_dmc	-- dmc@11200000
serial	[ + ]	ns16550_serial	-- serial@11020000
i2c	[ + ]	i2c_rockchip	-- i2c@11050000
<hr/>			

打印含义：

- 列出所有已经完成 bind 的 device-driver
- 列出所有 uclass-device-driver 之间的隶属关系
- [+] 表示当前 driver 已经完成 probe
- \* 表示当前device-driver来自于U-Boot的DTB，否则来在kernel DTB

## dm uclass

功能：查看某类uclass下的所有设备。

```
=> dm uclass

uclass 0: root
- * root_driver @ 7be54c88, seq 0, (req -1)

uclass 11: adc
- * saradc@ff100000 @ 7be56220, seq 0, (req -1)
.....
uclass 40: backlight
- * backlight @ 7be81178, seq 0, (req -1)

uclass 77: key
- rockchip-key @ 7be811f0
.....
```

## stacktrace.sh

利用调用栈回溯机制分析abort、dump\_stack()的现场。请参考RK架构章节。

## 系统卡死

功能：打印当前CPU的现场和调用栈，适用于系统卡死时使用。串口会每隔 5s dump 出和abort时类似的信息。

CONFIG\_ROCKCHIP\_DEBUGGER

获取到调用栈信息后再使用stacktrace脚本转换。请参考RK架构章节。

## CRC 校验

功能：校验RK格式的固件完整性。

RK 格式的镜像头包含了整个镜像的CRC32，打开如下宏可以用CRC32验证固件的完整性。

```
CONFIG_ROCKCHIP_CRC
```

范例：

```
=Booting Rockchip format image=
kernel image CRC32 verify... okay.          // kernel 校验成功（如果失败则打印“fail! ”）
boot image CRC32 verify... okay.            // boot 校验成功（如果失败则打印“fail! ”）
kernel @ 0x02080000 (0x01249808)
ramdisk @ 0x0a200000 (0x001e6650)
## Chapter-8 Flattened Device Tree blob at 01f00000
    Booting using the fdt blob at 0x1f00000
'reserved-memory' secure-memory@20000000: addr=20000000 size=10000000
    Loading Ramdisk to 08019000, end 081ff650 ... OK
    Loading Device Tree to 0000000008003000, end 0000000008018c97 ... OK
Adding bank: start=0x00200000, size=0x08200000
Adding bank: start=0x0a200000, size=0xede00000

Starting kernel ...
```

## HASH校验

功能：校验Android格式的固件完整性。

```
ANDROID_BOOT_IMAGE_HASH
```

启用该配置后，加载Android格式的固件时会校验固件的完整性。

因为一些历史原因，如果上述配置无法正确校验固件，请同时打开如下配置试试：

```
HASH_ROCKCHIP_LEGACY
```

## 修改DDR容量

开机时DDR初始化代码会把DDR容量传递给U-Boot，U-Boot会去除一些安全内存后再传递给内核。用户可以在U-Boot阶段修改传递给内核的DDR容量。

传递范例：

```
.....
// 传递给内核的可用内存块（已去除安全内存块）。
Adding bank: 0x00200000 - 0x08400000 (size: 0x08200000)
Adding bank: 0x0a200000 - 0x40000000 (size: 0x35e00000)
Total: 895.411 ms

Starting kernel ...
[    0.000000] Booting Linux on physical CPU 0x0
```

代码位置：

```
./arch/arm/mach-rockchip/param.c
```

修改位置：

```
struct memblock *param_parse_ddr_mem(int *out_count)
{
    .....

    // 这里就是ddr传递给U-Boot的容量信息。
    // 因为可能出现不连续的地址，所以会分块传递，分别指明各个内存块的起始地址和大小。
    // PS：一般情况下都是连续内存，不会需要分块。
    for (i = 0, n = 0; i < count; i++, n++) {
        // 比如2GB容量（连续地址），则：count=1, base = 0, size = 0x80000000。
        // 用户调试时可以在这里按需修改。
        base = t->u.ddr_mem.bank[i];
        size = t->u.ddr_mem.bank[i + count];

        /* 0~4GB */
        if (base < SZ_4GB) {
            mem[n].base = base;
            mem[n].size = ddr_mem_get_usable_size(base, size);
            if (base + size > SZ_4GB) {
                n++;
                mem[n].base_u64 = SZ_4GB;
                mem[n].size_u64 = base + size - SZ_4GB;
            }
        } else {
            /* 4GB+ */
            mem[n].base_u64 = base;
            mem[n].size_u64 = size;
        }

        assert(n < count + MEM_RESV_COUNT);
    }
    .....
}
```

## 跳转信息

功能：确认固件版本和流程。某些情况下，开机信息也可以帮助用户定位一些死机问题。

1. trust 跑完后就卡死

trust 跑完后就卡死的可能性：固件打包或者烧写有问题，导致 trust 跳转到错误的 U-Boot 启动地址。此时用户可以通过 trust 开机打印的 U-Boot 启动地址来确认。

64 位平台 U-Boot 启动地址一般是偏移 0x200000 (DRAM 起始地址是 0x0)：

```
NOTICE: BL31: v1.3(debug):d98d16e
NOTICE: BL31: Built : 15:03:07, May 10 2018
NOTICE: BL31: Rockchip release version: v1.1
INFO: GICv3 with legacy support detected. ARM GICV3 driver initialized in EL3
INFO: Using opteed sec cpu_context!
INFO: boot cpu mask: 0
INFO: plat_rockchip_pmu_init(1151): pd status 3e
INFO: BL31: Initializing runtime services
INFO: BL31: Initializing BL32
INFO: BL31: Preparing for EL3 exit to normal world
INFO: Entry point address = 0x200000 // U-Boot地址
INFO: SPSR = 0x3c9
```

32位平台U-Boot启动地址一般是偏移0x0 (DRAM起始地址是0x60000000) :

```
INF [0x0] TEE-CORE:init_primary_helper:378: Release version: 1.9
INF [0x0] TEE-CORE:init_primary_helper:379: Next entry point address: 0x60000000
// U-Boot地址
INF [0x0] TEE-CORE:init_teecore:83: teecore inits done
```

2. U-Boot版本回溯:

通过U-Boot开机信息可回溯编译版本。如下对应提交点是commit: b34f08b。

```
U-Boot 2017.09-01730-gb34f08b (Jul 06 2018 - 17:47:52 +0800)
```

开机信息中出现"dirty"，说明编译时有本地改动没有提交进仓库，编译点不干净。

```
U-Boot 2017.09-01730-gb34f08b-dirty (Jul 06 2018 - 17:35:04 +0800)
```

## 启动信息

用户通过U-Boot开机信息可获知当前U-Boot的流程和各外设的状态，方便快速定位异常。

目前U-Boot支持三种固件类型引导：Android格式 > RK格式 > DISTRO格式。RK发布的SDK主要是前两种固件格式，DISTRO一般是开源用户会使用。

说明：如果用户的代码不够新，有些打印可能看不到，这不影响用户对于U-Boot开机信息的整体了解。

### 17.1 Android固件

```
// U-Boot第一行打印，包含了commit版本、编译时间等信息
// 注意：这里只是U-Boot"相对早"的第一行正规打印，而不是U-Boot能做出的最早打印
// 打开debug信息，可以看到更早的调试打印
U-Boot 2017.09-03033-g81b79f7-dirty (Jul 04 2019 - 15:04:00 +0800)

// U-Boot dts的"model"字段内容，通过这个信息可以知道我们使用了U-Boot哪份dts
Model: Rockchip RK3399 Evaluation Board
// 启用了preloader-serial功能，即沿用前级loader的串口配置，当前使用UART2作为打印口
PreSerial: 2
// 板子的总内存容量是2GB
DRAM: 2 GiB
// 当前版本支持了sysmem内存卡管理机制
Sysmem: init
```

```
// U-Boot会把自己的代码进行自搬移，从当前ddr靠前的位置搬到靠后的位置（详见U-Boot开发文档启动流程）
// 自搬移后的代码起始地址是0x7dbe2000，这个信息在反汇编调试时可能有用到
Relocation Offset is: 7dbe2000
// ENV默认保存在ddr里。如果是选择保存在eMMC、Nand等存储介质里，则不会有这个打印
Using default environment

// 当前的存储介质是mmc0，即eMMC(如果是sd卡，则为mmc1)
dwmmc@fe320000: 1, sdhci@fe330000: 0
// 存储介质类型是通过atags，由前级miniloader传参告知U-Boot
Bootdev(atags): mmc 0
// 当前的eMMC的工作在HS400模式，时钟频率是150M
MMC0: HS400, 150Mhz
// 当前采用GPT分区表（如果是RK parameter分区表，则打印：RKPARM）
PartType: EFI
// 当前是recovery模式
// kernel里执行的"reboot xxx"命令，最终也是由这个打印来体现
boot mode: recovery
// Kernel DTB来自于recovery.img，其正常被加载
Load FDT from recovery part
DTB: rk-kernel.dtb
HASH: OK(c)

// ==> 注意：自此之后，U-Boot已经切到kernel dtb，后续所有外设驱动都使用kernel dtb的信息！！！

// DTBO执行成功
ANDROID: fdt overlay OK
// I2C的速度，这个是U-Boot开机速度的影响因素之一，尤其对于DCDC和LDO非常多的PMIC，如果I2C速度慢，
// 那么对开机速度有一定影响。如果用户关心开机速度，可以关注这个信息
I2c speed: 400000Hz
// 当前PMIC是RK818
// on值对应ON_SOURCE寄存器，表明了当前这次PMIC上电的原因
// off值对应OFF_SOURCE寄存器，表明前一次关机或掉电的原因
// on和off信息，对于系统出现异常重启或关机等情况时，这是个有价值的信息
PMIC: RK818 (on=0x20 off=0x40)
// 各路可调压regulator的当前电压值，一般是DCDC且对应的是RK平台的arm、logic、center等电压
// 在出现系统启动异常、开机不稳定等问题时，这是个有价值的信息vdd_center 900000 uV
vdd_cpu_1 900000 uV
vdd_log 900000 uV
// Kernel dts的“model”字段内容，通过这个信息可以知道我们使用了Kernel的哪份dts
Model: Rockchip RK3399 Excavator Board edp avb (Android)
enter Recovery mode!
// 显示驱动的相关信息
Rockchip UBOOT DRM driver version: v1.0.1
Using display timing dts
Detailed mode clock 200000 kHz, flags[a]
    H: 1536 1548 1564 1612
    V: 2048 2056 2060 2068
bus_format: 100e
// clk-tree信息，具体含义请参考U-Boot开发文档的CLK章节
CLK: (uboot. arm1: enter 816000 KHz, init 816000 KHz, kernel ON/A)
CLK: (uboot. arm2: enter 24000 KHz, init 24000 KHz, kernel ON/A)
apll1 816000 KHz
apll2 24000 KHz
dpll 800000 KHz
cpll 200000 KHz
```

```
gp11 800000 KHz
np11 600000 KHz
vp11 24000 KHz
aclk_perihip 133333 KHz
hclk_perihip 66666 KHz
pclk_perihip 33333 KHz
aclk_perilp0 266666 KHz
hclk_perilp0 88888 KHz
pclk_perilp0 44444 KHz
hclk_perilp1 100000 KHz
pclk_perilp1 50000 KHz
// GMAC驱动使能
Net: eth0: ethernet@fe300000
// 开机长按ctrl+c，可在如下打印之后进入U-Boot命令行模式
Hit key to stop autoboot('CTRL+C'): 0
// 再一次知道当前是recovery模式
ANDROID: reboot reason: "recovery"
// vboot=0表示没有启用secureboot; 当前是AVB固件，所以会走AVB的常规校验流程
Vboot=0, AVB images, AVB verify
// 设备是否unlock
read_is_device_unlocked() ops returned that device is UNLOCKED
// 原生的U-Boot默认是把整个boot.img/recovery.img加载起来，然后再把ramdisk、fdt、kernel再进行
// 一次搬移（称为relocation），搬到用户预定的地址上，这样是比较耗时的，尤其当ramdisk非常大的时候。
// RK平台做了修改，一次性直接从存储上把ramdisk、fdt、kernel搬到预定的内存地址。
// 有如下打印则说明启用了这种一次性搬移的操作，更省时间
Fdt Ramdisk skip relocation

// 加载Android格式的固件，把kernel加载到0x00280000，fdt加载到0x8300000
// 假如是LZ4压缩内核，这里可能打印：
// Booting LZ4 kernel at 0x00680000(uncompress to 0x00280000) with fdt at
0x8300000...
Booting IMAGE kernel at 0x00280000 with fdt at 0x8300000...

// 忽略，可不关心
### Booting Android Image at 0x0027f800 ...
// kernel和ramdisk的加载地址以及大小
Kernel load addr 0x00280000 size 19081 KiB
RAM disk load addr 0x0a200000 size 9627 KiB
// fdt的加载地址
### Flattened Device Tree blob at 08300000
    Booting using the fdt blob at 0x8300000
// 忽略，可不关心
    XIP Kernel Image ... OK
// 这里仅仅是打印kernel dts指定的reserved-memory，可作为出内核启动出问题时的一个分析信息
    'reserved-memory' secure-memory@20000000: addr=20000000 size=10000000
// fdt的起始和结束地址
    Using Device Tree in place at 000000008300000, end 00000000831c6f7
// 传递给内核，告知内核可使用的内存空间范围（ATF、optee等空间已经被除去）
Adding bank: 0x00200000 - 0x08400000 (size: 0x08200000)
Adding bank: 0x0a200000 - 0x80000000 (size: 0x75e00000)
// U-Boot阶段开机耗时
Total: 367.128 ms

// 由U-Boot打印，这个打印之后，U-Boot会完成一些ARM架构相关（比如：清cache、关中断、
// cpu状态切换等）和U-Boot的dm设备注销等清零工作，出问题的概率极低。
// 完成上述工作后就跳到kernel，因此也可以理解为，出现这个打印就是到了内核阶段。
```

```
Starting kernel ...
```

```
// kernel阶段的打印信息
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Initializing cgroup subsys cpufreq
[    0.000000] Initializing cgroup subsys schedtune
[    0.000000] Linux version 4.4.167 (hgc@ubuntu) (gcc version 6.3.1 20170404
(Linaro
GCC 6.3-2017.05) ) #83 SMP PREEMPT Thu Mar 21 09:31:08 CST 2019
[    0.000000] Boot CPU: AArch64 Processor [410fd034]
[    0.000000] earlycon: Early serial console at MMIO32 0xff1a0000 (options '')
[    0.000000] bootconsole [uart0] enabled
[    0.000000] Reserved memory: failed to reserve memory for node 'stb-
devinfo@00000000': base 0x0000000000000000, size 0 MiB
[    0.000000] cma: Reserved 16 MiB at 0x000000007f000000
....
```

## RK固件

```
U-Boot 2017.09-03352-gb1265b5 (Jul 12 2019 - 09:57:24 +0800)
```

```
Model: Rockchip RK3399 Evaluation Board
PreSerial: 2
DRAM: 2 GiB
Sysmem: init
Relocation Offset is: 7dbe2000
Using default environment
.....
Hit key to stop autoboot('CTRL+C'): 0
ANDROID: reboot reason: "recovery"
// 因为是RK格式的固件，所以不可能是AVB格式
Not AVB images, AVB skip
// 因为是RK格式的固件，所以这里会提示加载android格式固件失败
// 因为目前启动优先级是： android格式 > RK格式 > distro格式
** Invalid Android Image header **
Android image load failed
Android boot failed, error -1.
// 当前是recovery模式
boot mode: recovery
// 启动RK格式的固件，加载ramdis、kernel、fdt
=Booting Rockchip format image=
fdt      @ 0x08300000 (0x00012dd0)
kernel   @ 0x00280000 (0x0119e008)
ramdisk  @ 0x0a200000 (0x00754540)

// 下面基本类同android格式固件的启动信息
Fdt Ramdisk skip relocation
### Flattened Device Tree blob at 08300000
    Booting using the fdt blob at 0x8300000
    Using Device Tree in place at 000000008300000, end 000000008315dcf
Adding bank: 0x00200000 - 0x08400000 (size: 0x08200000)
Adding bank: 0x0a200000 - 0x80000000 (size: 0x75e00000)
Total: 508.11 ms
```

```
Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
.....
```

## Didstro固件

```
U-Boot 2017.09-03352-gb1265b5 (Jul 12 2019 - 09:57:24 +0800)

Model: Rockchip RK3399 Evaluation Board
PreSerial: 2
DRAM: 2 GiB
Sysmem: init
Relocation Offset is: 7dbe2000
Using default environment

.....
// 找到mmc0, 即eMMCswitch to partitions #0, OK
mmc0(part 0) is current device
// 查找eMMC存储上第6个分区的固件（GPT分区表里，6对应的是boot.img分区，GPT里用"-bootable"属性指明）
Scanning mmc 0:6...
// 找到了配置文件extlinux.conf
Found /extlinux/extlinux.conf
Retrieving file: /extlinux/extlinux.conf

// 加载kernel
205 bytes read in 82 ms (2 KiB/s)
1:      rockchip-kernel-4.4
Retrieving file: /Image
13484040 bytes read in 1833 ms (7 MiB/s)

// 打包时指定的cmdline信息
append: earlycon=uart8250,mmio32,0xff1a0000 console=ttyS2,1500000n8 rw
root=/dev/mmcblk0p7 rootwait rootfstype=ext4 init=/sbin/init

// 加载fdt
Retrieving file: /rk3399.dtb
61714 bytes read in 54 ms (1.1 MiB/s)

// ==> 如果打包时没有ramdisk, 就不会有ramdisk信息打印; 否则这里也会有打印

### Flattened Device Tree blob at 01f00000
Booting using the fdt blob at 0x1f00000
Loading Device Tree to 000000007df14000, end 000000007df26111 ... OK

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
.....
```

## 无有效固件

```
U-Boot 2017.09-03352-gb1265b5 (Jul 12 2019 - 09:57:24 +0800)

Model: Rockchip RK3399 Evaluation Board
PreSerial: 2
DRAM: 2 GiB
Sysmem: init
Relocation Offset is: 7dbe2000
Using default environment

.....
// 找到mmc0, 即eMMC
找不到固件的开机信息
switch to partitions #0, OK
mmc0(part 0) is current device
// 查找eMMC存储上第6个分区的固件 (GPT分区表里, 6对应的是boot.img分区, GPT里用"-bootable"属性指明)
Scanning mmc 0:6...
// 找到了配置文件extlinux.conf
Found /extlinux/extlinux.conf
Retrieving file: /extlinux/extlinux.conf

// 加载kernel
205 bytes read in 82 ms (2 KiB/s)
1:      rockchip-kernel-4.4
Retrieving file: /Image
13484040 bytes read in 1833 ms (7 MiB/s)

// 打包时指定的cmdline信息
append: earlycon=uart8250,mmio32,0xff1a0000 console=ttyS2,1500000n8 rw
root=/dev/mmcblk0p7 rootwait rootfstype=ext4 init=/sbin/init

// 加载fdt
Retrieving file: /rk3399.dtb
61714 bytes read in 54 ms (1.1 MiB/s)

// ==> 如果打包时没有ramdisk, 就不会有ramdisk信息打印; 否则这里也会有打印

### Flattened Device Tree blob at 01f00000
Booting using the fdt blob at 0x1f00000
Loading Device Tree to 000000007df14000, end 000000007df26111 ... OK

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
.....
U-Boot 2017.09-03352-gb1265b5 (Jul 12 2019 - 09:57:24 +0800)

Model: Rockchip RK3399 Evaluation Board
PreSerial: 2
DRAM: 2 GiB
Sysmem: init
Relocation Offset is: 7dbe2000
```

```
Using default environment

.....



Net: eth0: ethernet@fe300000
Hit key to stop autoboot('CTRL+C'): 0 ANDROID: reboot reason: "recovery"
// 不是Android格式固件
Not AVB images, AVB skip
** Invalid Android Image header **
Android image load failed
Android boot failed, error -1.
boot mode: recovery

// 不是RK格式固件
=Booting Rockchip format image=
kernel: invalid image tag(0x45435352)
boot_rockchip_image kernel part read error

// 不是DISTRO格式固件。后面所有的打印都来在distro加载命令，因为distro命令会试图从mmc、
nand、net、
// usb等所有我们预先定义的设备（详见rockchip-common.h中的宏定义：BOOT_TARGET_DEVICES）中
去寻
// 找distro固件，即逐一扫描进行查找
switch to partitions #0, OK
mmc0(part 0) is current device
Failed to mount ext2 filesystem...
** Unrecognized filesystem type **
starting USB...
USB0: Register 2000140 NbrPorts 2
Starting the controller
USB XHCI 1.10
USB1: Register 2000140 NbrPorts 2
Starting the controller
USB XHCI 1.10
USB2: USB EHCI 1.00
USB3: USB OHCI 1.0
USB4: USB EHCI 1.00
USB5: USB OHCI 1.0
scanning bus 0 for devices... 1 USB Device(s) found
scanning bus 1 for devices... 1 USB Device(s) found
scanning bus 2 for devices... 1 USB Device(s) found
scanning bus 3 for devices... 1 USB Device(s) found
scanning bus 4 for devices... 1 USB Device(s) found
scanning bus 5 for devices... 1 USB Device(s) found
    scanning usb for storage devices... 0 Storage Device(s) found

Device 0: unknown device
ethernet@fe300000 Waiting for PHY auto negotiation to complete..... TIMEOUT !
Could not initialize PHY ethernet@fe300000
missing environment variable: pxeuuid
missing environment variable: bootfile
Retrieving file: pxelinux.cfg/01-7a-1d-33-50-3d-a1
ethernet@fe300000 Waiting for PHY auto negotiation to complete..
.....



// 最终distro命令对所有可能的存储介质都扫描后也找不到固件，就停在U-Boot命令行模式
=>
```

## Chapter-9 测试用例

## Chapter-10 SPL

### 固件引导

SPL 的作用是代替miniloader完成 trust.img 和 uboot.img的加载和引导工作。SPL 目前支持引导两种固件：

- FIT 固件：默认使能；
- RKFW 固件：默认关闭，需要用户单独配置和使能；

#### FIT 固件

FIT (flattened image tree) 格式是 SPL 支持的一种比较新颖的固件格式，支持多个 image 打包和校验。FIT 使用 DTS 的语法对打包的 image 进行描述，描述文件为 u-boot.its，最终生成的 FIT 固件为 u-boot.itb。

FIT的优点：复用 dts 的语法和编译规则，比较灵活，固件解析可以直接使用 libfdt 库。

#### u-boot.its 文件：

- `/images`：静态定义了所有可获取的资源配置（最后可用、可不用），类似 dtsi 的角色；
- `/configurations`：每个 config 节点描述了一套可启动的配置，类似一个板级 dts。
- 使用 `default =` 指定当前选用的默认配置；

范例：

```
/dts-v1/;

/ {
    description = "Configuration to load ATF before U-Boot";
    #address-cells = <1>

    images {
        uboot@1 {
            description = "U-Boot (64-bit)";
            data = /incbin/("u-boot-nodtb.bin");
            type = "standalone";
            os = "U-Boot";
            arch = "arm64";
            compression = "none";
            load = <0x00200000>;
        };
        atf@1 {
            description = "ARM Trusted Firmware";
            data = /incbin/("bl31_0x00010000.bin");
            type = "firmware";
            arch = "arm64";
            os = "arm-trusted-firmware";
            compression = "none";
            load = <0x00010000>;
        };
    };
}
```

```

        entry = <0x00010000>;
};

atf@2 {
    description = "ARM Trusted Firmware";
    data = /incbin/("bl31_0xff091000.bin");
    type = "firmware";
    arch = "arm64";
    os = "arm-trusted-firmware";
    compression = "none";
    load = <0xff091000>;
};

optee@1 {
    description = "OP-TEE";
    data = /incbin/("bl32.bin");
    type = "firmware";
    arch = "arm64";
    os = "op-tee";
    compression = "none";
    load = <0x08400000>;
};

fdt@1 {
    description = "rk3328-evb.dtb";
    data = /incbin/("arch/arm/dts/rk3328-evb.dtb");
    type = "flat_dt";
    compression = "none";
};
};

configurations {
    default = "config@1";
    config@1 {
        description = "rk3328-evb.dtb";
        firmware = "atf@1";
        loadables = "uboot@1", "atf@2", "optee@1" ;
        fdt = "fdt@1";
    };
};
};

```

### u-boot.itb 文件:

mkimage + dtc	
[u-boot.its] + [images]	==>
	[u-boot.itb]

上述是itb文件的生成过程。FIT 固件可以理解为一种特殊的 DTB 文件，只是它的内容是 image。用户可以用 fdtdump 命令查看 itb文件：

```

cjh@ubuntu:~/uboot-nextdev/u-boot$ fdtdump u-boot.itb | less

/dts-v1/;
// magic:          0xd00dfeed
// totalsize:      0x497 (1175)
// off_dt_struct:  0x38

```

```

// off_dt_strings:      0x414
// off_mem_rsvmap:     0x28
// version:             17
// last_comp_version:  16
// boot_cpuid_phys:    0x0
// size_dt_strings:    0x83
// size_dt_struct:     0x3dc

{
    timestamp = <0x5d099c85>;
    description = "Configuration to load ATF before U-Boot";
    #address-cells = <0x00000001>;
    images {
        uboot@1 {
            data-size = <0x0009f8a8>;
            data-offset = <0x00000000>;
            description = "U-Boot (64-bit)";
            type = "standalone";
            os = "U-Boot";
            arch = "arm64";
            compression = "none";
            load = <0x00600000>;
        };
        atf@1 {
            data-size = <0x0000c048>; // 编译过程自动增加了该字段，描述atf@1固件大小
            data-offset = <0x0009f8a8>; // 编译过程自动增加了该字段，描述atf@1固件偏移
            description = "ARM Trusted Firmware";
            type = "firmware";
            arch = "arm64";
            os = "arm-trusted-firmware";
            compression = "none";
            load = <0x00010000>;
            entry = <0x00010000>;
        };
        atf@2 {
            data-size = <0x00002000>;
            data-offset = <0x000ab8f0>;
            description = "ARM Trusted Firmware";
            type = "firmware";
            arch = "arm64";
            os = "arm-trusted-firmware";
            compression = "none";
            load = <0xffff82000>;
        };
        fdt@1 {
            data-size = <0x00005793>;
            data-offset = <0x000ad8f0>;
            description = "rk3308-evb.dtb";
            type = "flat_dt";
            .....
        };
        .....
    };
}

```

更多 FIT 信息请参考：

```
./doc/uImage.FIT/
```

## RKFW 固件

为了能更直接替换掉 miniloader 且不用修改后级固件的分区、打包格式。因此RK平台增加了RKFW 格式（即独立分区的固件：trust.img 和 uboot.img）的引导。

配置：

```
CONFIG_SPL_LOAD_RKFW          // 使能开关
CONFIG_RKFW_TRUST_SECTOR      // trust.img 分区地址，需要和分区表的定义保持一致
CONFIG_RKFW_U_BOOT_SECTOR      // uboot.img 分区地址，需要和分区表的定义保持一致
```

代码：

```
./include/spl_rkfw.h
./common/spl/spl_rkfw.c
```

## 存储优先级

U-Boot dts 中通过 `u-boot,spl-boot-order` 指定存储设备的启动优先级。

```
/ {
    aliases {
        mmc0 = &emmc;
        mmc1 = &sddmmc;
    };

    chosen {
        u-boot,spl-boot-order = &sddmmc, &nandc, &emmc;
        stdout-path = &uart2;
    };
    .....
};
```

## 编译打包

### 代码编译

U-Boot 根据**不同的编译路径** 对同一份U-Boot代码编译获得SPL固件，当编译 SPL 时会自动生成 `CONFIG_SPL_BUILD` 宏。U-Boot会在编译完 `u-boot.bin` 之后继续编译 SPL，并创建独立的输出目录 `./spl/`。

```
// 编译u-boot
.....
DTC      arch/arm/dts/rk3399-puma-ddr1866.dtb
DTC      arch/arm/dts/rv1108-evb.dtb
make[2]: `arch/arm/dts/rk3328-evb.dtb' is up to date.
SHIPPED dts/dt.dtb
FDTGREP dts/dt-spl.dtb
CAT      u-boot-dtb.bin
MKIMAGE u-boot.img
COPY    u-boot.dtb
MKIMAGE u-boot-dtb.img
```

```
COPY      u-boot.bin

// 编译spl，有独立的spl/目录
LD       spl/arch/arm/cpu/built-in.o
CC       spl/board/rockchip/evb_rk3328/evb_rk3328.o
LD       spl/dts/built-in.o
CC       spl/common/init/board_init.o
COPY    tpl/u-boot-tpl.dtb
CC       spl/cmd/nvedit.o
CC       spl/env/common.o
CC       spl/env/env.o
.....
LD       spl/drivers/block/built-in.o
.....
```

编译结束后得到：

```
./spl/u-boot-spl.bin
```

## 固件打包

## 系统模块

### GPT

SPL 使用GPT分区表。

配置：

```
CONFIG_SPL_LIBDISK_SUPPORT=y
CONFIG_SPL_EFI_PARTITION=y
CONFIG_PARTITION_TYPE_GUID=y
```

驱动：

```
./disk/part.c
./disk/part_efi.c
```

接口：

```
int part_get_info(struct blk_desc *dev_desc, int part, disk_partition_t *info);
int part_get_info_by_name(struct blk_desc *dev_desc,
                         const char *name, disk_partition_t *info);
```

## A/B system

SPL 支持A/B 系统启动。

配置：

```
CONFIG_SPL_AB=y
```

驱动：

```
./common/spl/spl_ab.c
```

接口：

```
int spl_get_current_slot(struct blk_desc *dev_desc, char *partition, char *slot);
int spl_get_partitions_sector(struct blk_desc *dev_desc, char *partition, u32 *sectors);
```

## 启动优先级

- SPL 使用 `u-boot,spl-boot-order` 定义的启动顺序，位于 `rkxxxx-u-boot.dtsi`：

```
chosen {
    stdout-path = &uart2;
    u-boot,spl-boot-order = &sddmmc, &sfc, &nandc, &emmc;
};
```

- Maskrom 的启动优先级：

```
spi nor > spi nand > emmc > sd
```

- Pre-loader(SPL) 的启动优先级：

```
sd > spi nor > spi nand > emmc
```

把 sd 卡的优先级提到最高可以方便系统从 sd 卡启动。

## ATAGS

SPL 与 U-Boo 通过 ATAGS 机制实现传参。传递的信息有：启动的存储设备、打印串口等。

配置：

```
CONFIG_ROCKCHIP_PRELOADER_ATAGS=y
```

驱动：

```
./arch/arm/include/asm/arch-rockchip/rk_atags.h
./arch/arm/mach-rockchip/rk_atags.c
```

接口：

```
int atags_set_tag(u32 magic, void *tagdata);
struct tag *atags_get_tag(u32 magic);
```

## kernel boot

通常 kernel 是由 U-Boot 加载和引导，SPL 也可以支持加载 kernel。目前支持加载 android head version 2 的 boot.img，支持 RK 格式固件。

启动顺序：

## pinctrl

配置:

```
CONFIG_SPL_PINCTRL_GENERIC=y  
CONFIG_SPL_PINCTRL=y
```

驱动:

```
./drivers/pinctrl/pinctrl-uclass.c  
./drivers/pinctrl/pinctrl-generic.c  
./drivers/pinctrl/pinctrl-rockchip.c
```

DTS 配置:

以 sdmmc 为例:

```
&pinctrl {  
    u-boot,dm-spl;  
};  
  
&pcfg_pull_none_4ma {  
    u-boot,dm-spl;  
};  
  
&pcfg_pull_up_4ma {  
    u-boot,dm-spl;  
};  
  
&sdmmc {  
    u-boot,dm-spl;  
};  
  
&sdmmc_pin {  
    u-boot,dm-spl;  
};  
  
&sdmmc_clk {  
    u-boot,dm-spl;  
};  
  
&sdmmc_cmd {  
    u-boot,dm-spl;  
};  
  
&sdmmc_bus4 {  
    u-boot,dm-spl;  
};  
  
&sdmmc_pwren {  
    u-boot,dm-spl;  
};
```

## 注意事项：

SPL 启用pinctrl时要修改 defconfig 里的 `CONFIG_OF_SPL_REMOVE_PROPS` 定义，删除其中的 `pinctrl-0 pinctrl-names` 字段。

## secure boot

[TODO]

## 驱动模块

### MMC

配置：

```
CONFIG_SPL_MMC_SUPPORT=y // 默认已使能
```

驱动：

```
./common/spl/spl_mmc.c
```

接口：

```
int spl_mmc_load_image(struct spl_image_info *spl_image,
                        struct spl_boot_device *bootdev);
```

## MTD block

SPL 统一 nand、spi nand、spi nor 接口到 block 层。

配置：

```
// MTD 驱动支持
CONFIG_MTD=y
CONFIG_CMD_MTD_BLK=y
CONFIG_SPL_MTD_SUPPORT=y
CONFIG_MTD_BLK=y
CONFIG_MTD_DEVICE=y

// spi nand 驱动支持
CONFIG_MTD_SPI_NAND=y
CONFIG_ROCKCHIP_SFC=y
CONFIG_SPL_SPI_FLASH_SUPPORT=y
CONFIG_SPL_SPI_SUPPORT=y

// nand 驱动支持
CONFIG_NAND=y
CONFIG_CMD_NAND=y
CONFIG_NAND_ROCKCHIP=y
CONFIG_SPL_NAND_SUPPORT=y
CONFIG_SYS_NAND_U_BOOT_LOCATIONS=y
CONFIG_SYS_NAND_U_BOOT_OFFS=0x8000
CONFIG_SYS_NAND_U_BOOT_OFFS_REDUND=0x10000
// nand page size需要按真实大小定义，如果使用容量大于等于512MB的NAND，一般需要配置为4096
#define CONFIG_SYS_NAND_PAGE_SIZE 2048
```

```
// spi nor 驱动支持
CONFIG_CMD_SF=y
CONFIG_CMD_SPI=y
CONFIG_SPI_FLASH=y
CONFIG_SF_DEFAULT_MODE=0x1
CONFIG_SF_DEFAULT_SPEED=50000000
CONFIG_SPI_FLASH_GIGADEVICE=y
CONFIG_SPI_FLASH_MACRONIX=y
CONFIG_SPI_FLASH_WINBOND=y
CONFIG_SPI_FLASH_MTD=y
CONFIG_ROCKCHIP_SFC=y
CONFIG_SPL_SPI_SUPPORT=y
CONFIG_SPL_MTD_SUPPORT=y
CONFIG_SPL_SPI_FLASH_SUPPORT=y
```

驱动：

```
./common/spl/spl_mtd_blk.c
```

接口：

```
int spl_mtd_load_image(struct spl_image_info *spl_image,
                      struct spl_boot_device *bootdev);
```

## OTP

用于存储不可更改数据，secure boot 中用到。

配置：

```
CONFIG_SPL_MISC=y
CONFIG_SPL_ROCKCHIP_SECURE OTP=y
```

驱动：

```
./drivers/misc/misc-uclass.c
./drivers/misc/rockchip-secure-otp.S
```

接口：

```
int misc_read(struct udevice *dev, int offset, void *buf, int size);
int misc_write(struct udevice *dev, int offset, void *buf, int size);
```

## Crypto

Secure-boot 会使用crypto完成hash、ras的计算。

配置：

```
CONFIG_SPL_DM_CRYPTO=y

// crypto v1 支持平台: rk3399/rk3368/rk3328/rk3229/rk3288/rk3128
CONFIG_SPL_ROCKCHIP_CRYPTO_V1=y

// crypto v2 支持平台: px30/rk3326/rk1808/rk3308
CONFIG_SPL_ROCKCHIP_CRYPTO_V2=y
```

驱动:

```
./drivers/crypto/crypto-uclass.c
./drivers/crypto/rockchip/crypto_v1.c
./drivers/crypto/rockchip/crypto_v2.c
./drivers/crypto/rockchip/crypto_v2_pka.c
./drivers/crypto/rockchip/crypto_v2_util.c
```

接口:

```
u32 crypto_algo_nbits(u32 algo);
struct udevice *crypto_get_device(u32 capability);
int crypto_sha_init(struct udevice *dev, sha_context *ctx);
int crypto_sha_update(struct udevice *dev, u32 *input, u32 len);
int crypto_sha_final(struct udevice *dev, sha_context *ctx, u8 *output);
int crypto_sha_csum(struct udevice *dev, sha_context *ctx,
                     char *input, u32 input_len, u8 *output);
int crypto_rsa_verify(struct udevice *dev, rsa_key *ctx, u8 *sign, u8 *output);
```

## Uart

SPL 串口通过 `rkxxxx-u-boot.dtsi` 的 chosen 节点指定。以 rk3308 为例:

```
chosen {
    stdout-path = &uart2;
};

&uart2 {
    u-boot,dm-pre-reloc;
    clock-frequency = <24000000>;
    status = "okay";
};
```

## Chapter-11 TPL

TPL是比U-Boot更早阶段的Loader，TPL运行在SRAM中，其作用是代替ddr bin负责完成DRAM的初始化工作。TPL是代码开源的版本，ddr bin是代码闭源的版本。

## 编译打包

### 配置

- UART配置

CONFIG\_DEBUG\_UART\_BASE: UART基地址。

`CONFIG_ROCKCHIP_UART_MUX_SEL_M`: UART IOMUX GROUP。

Example:

RV1126配置UART2 M2用于打印DEBUG LOG。

方式1) 通过修改rv1126\_defconfig文件

```
CONFIG_DEBUG_UART_BASE=0xff570000
CONFIG_ROCKCHIP_UART_MUX_SEL_M=2
```

方式2) 通过make menuconfig

```
Device Drivers ---> Serial drivers ---> (0xff570000) Base address of UART
ARM architecture ---> (2) UART mux select
```

- DRAM TYPE配置

通过`CONFIG_ROCKCHIP_TPL_INIT_DRAM_TYPE`配置TPL支持的DRAM TYPE。

DDR TYPE	配置值
DDR2	2
DDR3	3
DDR4	0
LPDDR2	5
LPDDR3	6
LPDDR4	7

Example:

RV1126配置TPL DRAM TYPE为支持DDR3。

方式1) 通过修改rv1126\_defconfig文件

```
CONFIG_ROCKCHIP_TPL_INIT_DRAM_TYPE=3
```

方式2) 通过make menuconfig, 需要注意的是编译时如果make.sh后面有带上芯片型号的话, make时会有一个make xxxdefconfig的动作, 会覆盖menuconfig的改动。可不带参数的执行make.sh编译, 来防止menuconfig的改动被覆盖。

```
Device Drivers ---> (3) TPL select DRAM type
```

Example:

make rv1126\_defconfig或者./make.sh rv1126 -> make menuconfig修改相关配置 -> ./make.sh。

- 快速开机配置

如果需要编译生成支持快速开机的tpl.bin, 可以通过打开`CONFIG_SPL_KERNEL_BOOT`来编译生成。

当前仅支持RV1126/RV1109平台。

- 宽温的支持

如果需要编译生成支持宽温的tpl.bin，可以通过打开  
CONFIG\_ROCKCHIP\_DRAM\_EXTENDED\_TEMP\_SUPPORT来编译生成。

当前仅支持RV1126/RV1109平台。

- 其他参数修改

ddr初始化源码位于drivers/ram/rockchip目录下，其他ddr相关参数如频率，驱动强度，ODT强度等均需要在源码中修改。对于RV1126/RV1109来说有将ddr相关参数集中到该目录下的“sdram\_inc/rv1126/sdram-rv1126-loader\_params.inc”中，可以直接在该文件中修改对应的参数。其他平台参数修改需要在对应sdram\_xxx.c中修改。

## 编译

U-Boot根据不同的编译路径对同一份U-Boot代码编译获得TPL固件，当编译TPL时会自动生成  
CONFIG\_TPL\_BUILD宏。U-Boot会在编译完u-boot.bin之后继续编译TPL，并创建独立的输出目录  
./tpl/。

```
// 编译u-boot
.....
DTC      arch/arm/dts/rv1108-evb.dtb
DTC      arch/arm/dts/rk3399-puma-ddr1866.dtb
DTC      arch/arm/dts/rv1126-evb.dtb
FDTGREP dts/dt.dtb
FDTGREP dts/dt-spl.dtb
FDTGREP dts/dt-tpl.dtb
CAT      u-boot-dtb.bin
MKIMAGE u-boot.img
COPY    u-boot.dtb
MKIMAGE u-boot-dtb.img
COPY    u-boot.bin
ALIGN   u-boot.bin

// 编译tpl，有独立的tpl/目录
.....
CC      tpl/common/init/board_init.o
CC      tpl/disk/part.o
LD      tpl/common/init/built-in.o
.....
LD      tpl/u-boot-tpl
.....
OBJCOPY tpl/u-boot-tpl-nodtb.bin
COPY    tpl/u-boot-tpl.bin
```

编译结束后得到：

```
./tpl/u-boot-tpl.bin
```

Example:

编译 RV1126 uboot。

```
./make.sh rv1126
```

## 打包

1. 编译生成的u-boot-tpl.bin需要将头4个byte替换成相应平台的tag后才是一个合法的ddr bin。如 RV1126/RV1109平台该tag是“110B”。如果只需要ddr bin的话需要自己手动完成该步骤tag的替换动作，该动作可参考scripts/spl.sh脚本。

Example：替换RV1126 u-boot-tpl.bin的tag。

```
dd bs=4 skip=1 if=tpl/u-boot-tpl.bin of=tpl/u-boot-tpl-tag.bin && sed -i  
'1s/^/110B/' tpl/u-boot-tpl-tag.bin
```

2. 如果需要生成完整的可烧写入板子的Loader文件的话，可通过下面命令自动完成u-boot-tpl.bin tag的替换动作以及和spl.bin打包成一个完整的Loader文件动作。

```
./make.sh tpl
```

## Chapter-12 FIT

### 前言

本章节将介绍FIT格式和基于FIT格式的安全/非安全启动方案细节。本章节为了便于介绍，全文主要以boot.img为说明和操作对象，但是同样适用于recovery.img。

### 简介

#### 基础介绍

FIT (flattened image tree) 是U-Boot支持的一种新固件类型引导方案，支持任意多个image打包和校验。FIT 使用 its (image source file) 文件描述image信息，最后通过mkimage工具生成 itb (flattened image tree blob) 镜像。its文件使用 DTS 的语法规则，非常灵活，可以直接使用libfdt 库和相关工具。

FIT 是U-Boot默认支持且主推的固件格式，SPL和U-Boot阶段都支持对FIT格式的固件引导。更多信息请参考：

```
./doc/uImage.FIT/
```

因为官方的FIT功能无法满足实际产品需求，所以RK平台对FIT进行了适配和优化。所以FIT方案中必须使用RK U-Boot编译生的mkimage工具，不能使用PC自带的mkimage。

#### 范例介绍

如下以u-boot.its和u-boot.itb作为范例进行介绍。

- `/images`：静态定义了所有的资源，相当于一个dtsi文件；
- `/configurations`：每个 config 节点都描述了一套可启动的配置，相当于一个板级dts文件。
- `default =`：指明默认启用的config；

```
/dts-v1/;  
  
/  
 {  
     description = "Simple image with OP-TEE support";  
     #address-cells = <1>;  
  
     images {
```

```

uboot {
    description = "U-Boot";
    data = /incbin/("./u-boot-nodtb.bin");
    type = "standalone";
    os = "U-Boot";
    arch = "arm";
    compression = "none";
    load = <0x00400000>;
    hash {
        algo = "sha256";
    };
};

optee {
    description = "OP-TEE";
    data = /incbin/("./tee.bin");
    type = "firmware";
    arch = "arm";
    os = "op-tee";
    compression = "none";
    load = <0x8400000>;
    entry = <0x8400000>;
    hash {
        algo = "sha256";
    };
};

fdt {
    description = "U-Boot dtb";
    data = /incbin/("./u-boot.dtb");
    type = "flat_dt";
    compression = "none";
    hash {
        algo = "sha256";
    };
};

};

// configurations 节点下可以定义任意多个不同的conf节点，但实际产品方案上我们只需要一个
// conf即可。
configurations {
    default = "conf";
    conf {
        description = "Rockchip armv7 with OP-TEE";
        rollback-index = <0x0>;
        firmware = "optee";
        loadables = "uboot";
        fdt = "fdt";
        signature {
            algo = "sha256,rsa2048";
            padding = "pss";
            key-name-hint = "dev";
            sign-images = "fdt", "firmware", "loadables";
        };
    };
};

```

使用mkimage工具和its文件可以生成itb文件：

```
mkimage + dtc  
[u-boot.its] + [images] =====> [u-boot.itb]
```

fddt dump 命令可以查看 itb 文件内容：

```
cjh@ubuntu:~/uboot-nextdev/u-boot$ fddt dump fit/u-boot.itb | less

/dts-v1/;
// magic:      0xd00dfeed
// totalsize:   0x600 (1536)
// off_dt_struct: 0x48
// off_dt_strings: 0x48c
// off_mem_rsvmap: 0x28
// version:    17
// last_comp_version: 16
// boot_cpuid_phys: 0x0
// size_dt_strings: 0xc3
// size_dt_struct: 0x444

/memreserve/ 7f34d3411000 600;
{
    version = <0x00000001>;                      // 新增固件版本号
    totalsize = <0x000bb600>;                     // 新增字段描述整个itb文件的大小
    timestamp = <0x5ecb3553>;                     // 新增当前固件生成时刻的时间戳
    description = "Simple image with OP-TEE support";
    #address-cells = <0x00000001>;
    images {
        uboot {
            data-size = <0x0007ed54>;           // 新增字段描述固件大小
            data-position = <0x00000a00>; // 新增字段描述固件偏移
            description = "U-Boot";
            type = "standalone";
            os = "U-Boot";
            arch = "arm";
            compression = "none";
            load = <0x00400000>;
            hash {
                // 新增固件的sha256校验和
                value = <0xedaa8cd52 0x8f058118 0x00000003 0x35360000 0x6f707465
0x0000009f 0x00000091 0x00000000>;
                algo = "sha256";
            };
        };
        optee {
            data-size = <0x0003a058>;
            data-position = <0x0007f800>;
            description = "OP-TEE";
            type = "firmware";
            arch = "arm";
            os = "op-tee";
            compression = "none";
            load = <0x08400000>;
            entry = <0x08400000>;
            hash {
                value = <0xa569b7fc 0x2450ed39 0x00000003 0x35360000 0x66647400
0x00001686 0x000b9a00 0x552d426f>;
            };
        };
    };
}
```

```

        algo = "sha256";
    };
};

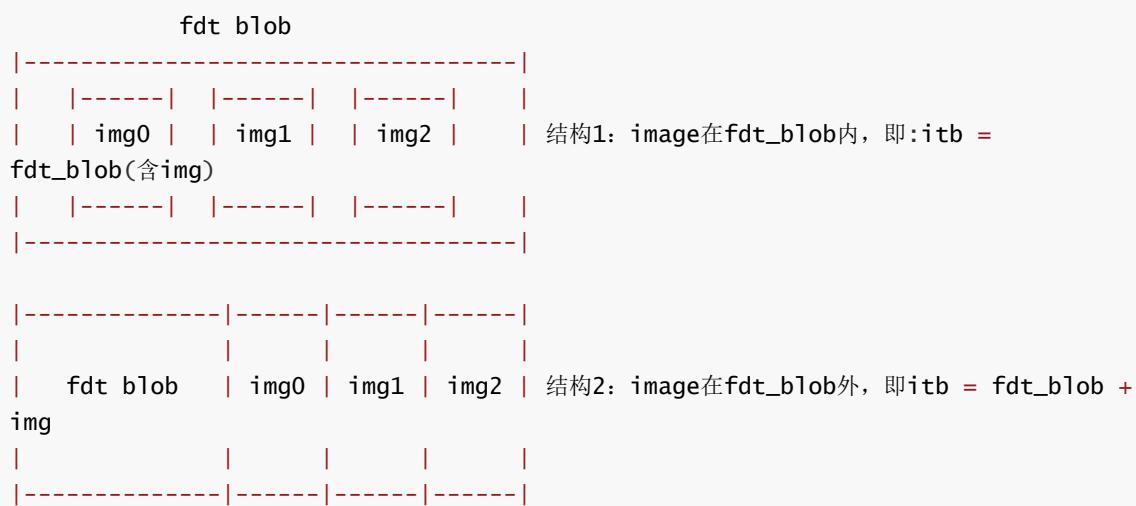
fdt {
    data-size = <0x00001686>;
    data-position = <0x000b9a00>;
    description = "U-Boot dtb";
    type = "flat_dt";
    compression = "none";
    hash {
        value = <0x0f718794 0x78ece7b2 0x00000003 0x35360000 0x00000001
0x6e730000 0x636f6e66 0x00000000>;
        algo = "sha256";
    };
};
};

configurations {
    default = "conf";
    conf {
        description = "Rockchip armv7 with OP-TEE";
        rollback-index = <0x00000001>; // 固件防回滚版本号，没有手动指定时默认为0
        firmware = "optee";
        loadables = "uboot";
        fdt = "fdt";
        signature {
            algo = "sha256,rsa2048";
            padding = "pss";
            key-name-hint = "dev";
            sign-images = "fdt", "firmware", "loadables";
        };
    };
};
};

```

## itb结构

itb本质是fdt\_blob + images的文件集合，有如下两种打包方式，RK平台方案采用结构2方式。



## 平台配置

### 支持列表

目前已经作为正式Feature发布在SDK的平台有：

支持平台
RV1126/RV1109
RK3566/RK3568

## 代码配置

代码：

```
// 框架代码  
.common/image.c  
.common/image-fit.c  
.common/spl/spl_fit.c  
  
// 平台代码：  
.arch/arm/mach-rockchip/fit.c  
.cmd/bootfit.c  
  
// 工具代码  
.tools/mkimage.c  
.tools/fit_image.c
```

配置：

```
// U-Boot阶段支持FIT  
CONFIG_ROCKCHIP_FIT_IMAGE=y  
  
// U-Boot阶段：安全启动、防回滚、硬件crypto  
CONFIG_FIT_SIGNATURE=y  
CONFIG_FIT_ROLLBACK_PROTECT=y  

```

如果FIT方案是作为SDK正式发布的feature，那么大部分基础配置已使能，用户需要自己配置的选项有：

```
// U-Boot 安全启动和防回滚机制
CONFIG_FIT_SIGNATURE=y
CONFIG_FIT_ROLLBACK_PROTECT=y

// SPL 安全启动和防回滚机制
CONFIG_SPL_FIT_SIGNATURE=y
CONFIG_SPL_FIT_ROLLBACK_PROTECT=y
```

- CONFIG\_FIT\_SIGNATURE没有使能：uboot可以同时支持引导三种格式的固件：android、uimage、fit（发布的SDK会根据平台需求选择开启哪几种支持）。
- CONFIG\_FIT\_SIGNATURE使能：uboot只支持引导fit固件。

## 镜像文件

FIT方案上最终输出两个FIT格式的固件用于烧写，分别是uboot.img（没有trust.img）和boot.img，还有一个SPL文件用于打包成loader。

- uboot.img 文件  
uboot.itb = trust + u-boot.bin + mcu.bin(option)  
uboot.img = uboot.itb \* N份（N一般是2）  
trust 和 mcu 文件来自rkbin工程，编译脚本会自动从rkbin工程索引并获取它们。
- boot.img 文件  
boot.itb = kernel + fdt + resource + ramdisk(optional)  
boot.img = boot.itb \* M份（M一般是1）
- MCU 配置

目前某些平台可能带有MCU固件，不同产品可以根据相应的 TRUST ini 配置来决定是否启用。例如：

```
// 文件: RKTRUST/RV1126TOS_TB.ini, 用于快速开机产品, 启用了MCU。
[TOS]
TOSTA=bin/rv11/rv1126_tee_ta_tb_v1.04.bin
ADDR=0x00040000

// MCU配置格式: 固件路径, 启动地址, 状态(okay或disabled)。
// 如果为disabled, 则mcu不会被打包进uboot.img中。
[MCU]
MCU=bin/rv11/rv1126_mcu_v1.02.bin,0x108000,okay
```

- 固件压缩

目前某些平台可以支持uboot.img内部子固件的压缩，支持如下：

平台	压缩格式	固件
RV1126	gzip、none	u-boot.bin, trust, mcu(optional)

用户可以在 rkbin 工程中对应的 TRUST ini 增加属性来启用。例如：

```

// 文件: RKTRUST/RV1126TOS_SPI_NOR_TINY.ini, 用于小容量SPI Nor产品。
[TOS]
TOS=bin/rv11/rv1126_tee_v1.02.bin
ADDR=0x08400000
[MCU]
MCU=bin/rv11/rv1126_mcu_v1.00.bin,0x208000,disabled

// 压缩格式: gzip或none, 不存在如下配置字段则默认非压缩。
[COMPRESSION]
COMPRESSION=gzip

```

- SPL 文件

SPL文件指的是编译完成后生成的 `sp1/u-boot-sp1.bin`, 负责引导FIT格式的uboot.img。用户需要用它替换RK平台上不开源的miniloader, 最终打包出loader。

- `./fit` 目录

U-Boot编译完成后会在目录下生成 `./fit` 文件夹, 包含了一些中间文件, 后续章节会介绍。

`boot.img`和`uboot.img`分别在sdk工程和uboot工程下被编译生成。但是支持安全启动的`boot.img`必须放在 U-Boot工程下重新打包签名, 后续章节会介绍。

## its 文件

- uboot的its文件为`./fit/u-boot.its`, 由defconfig中 `CONFIG_SPL_FIT_GENERATOR` 指定的脚本动态创建, 固件编译成功后可见。
- boot的its文件位于SDK工程下:

```
device/rockchip/[platform]/xxx.its // [platform]是平台目录
```

## 相关工具

```

// 核心打包工具, 编译完成后会自动生成, U-Boot和rkbin仓库下都有 (U-Boot仓库下是实时编译生成)。
./tools/mkimage
// 固件打包脚本
./make.sh
// 固件重签名脚本
scripts/fit-resign.sh
// 固件解包脚本
scripts/fit-unpack.sh
// 固件替换脚本
./scripts/fit-repack.sh

```

脚本工具的使用在后续章节会介绍, 此处先重点介绍`make.sh`的参数:

### 可选项(用户根据实际情况决定是否传递):

- `--sp1-new`: 传递此参数, 表示使用当前编译的sp1文件打包loader; 否则使用rkbin工程里的sp1文件。
- `--version-uboot [n]`: 指定uboot.img的固件版本号, n必须是十进制正整数。
- `--version-boot [n]`: 指定boot.img的固件版本号, n必须是十进制正整数;
- `--version-recovery [n]`: 指定recovery.img的固件版本号, n必须是十进制正整数;

### 必选项(启用安全启动的情况):

- `--rollback-index-uboot [n]`：指定uboot.img 固件防回滚版本号，n必须是十进制正整数；
- `--rollback-index-boot [n]`：指定boot.img 固件防回滚版本号，n必须是十进制正整数；
- `--rollback-index-recovery [n]`：指定recovery.img 固件防回滚版本号，n必须是十进制正整数；
- `--no-check`：打包安全固件时被使用，用于跳过安全固件打包脚本的自校验。

说明：

1. 固件防回滚版本号：只有在启用了安全启动的前提下才允许被激活使用，该版本号保存在OTP或者其它安全存储中。主要作用：为了防止固件版本被回退后进行漏洞攻击。
2. 固件版本号：可选，不指定的情况下默认为0。主要作用：只是作为固件版本标识，方便用户对固件进行版本管理。

## 非安全启动

### uboot.img

编译命令：

```
./make.sh rv1126 --spl-new --uboot-version 10 // 可不指定 --spl-new和--uboot-version
```

编译结果：

```
.....
CC      spl/common/spl/spl.o
CC      spl/lib/display_options.o
LD      spl/common/spl/built-in.o
LD      spl/lib/built-in.o
LD      spl/u-boot-spl
OBJCOPY spl/u-boot-spl-nodtb.bin
CAT     spl/u-boot-spl-dtb.bin
COPY    spl/u-boot-spl.bin
CFGCHK  u-boot.cfg

out:rv1126_spl_loader_v1.00.100.bin
fix opt:rv1126_spl_loader_v1.00.100.bin
merge success(rv1126_spl_loader_v1.00.100.bin)
/home4/cjh/uboot-nextdev

// 生成 rv1126_spl_loader_v1.00.100.bin (用spl替代了RK平台传统的miniloader
// loader ini 文件来源
pack loader(SPL) okay! Input: /home4/cjh/rkbin/RKBOOT/RV1126MINIALL.ini
// 来自 --spl-new 参数的提示；用户可以选择不加这个参数。
pack loader with new: spl/u-boot-spl.bin

// 生成 uboot.img (包含trust和uboot)，版本号为10
Image(no-signed, version=10): uboot.img (FIT with uboot, trust...) is ready
// trust ini文件来源
pack uboot.img okay! Input: /home4/cjh/rkbin/RKTRUST/RV1126TOS.ini

Platform RV1126 is build OK, with exist .config
```

打包备份：通过defconfig配置指定uboot.img的多备份：

```
CONFIG_SPL_FIT_IMAGE_KB=2048 // 单份itb大小  
CONFIG_SPL_FIT_IMAGE_MULTIPLE=2 // 打包的份数
```

SPL根据这个配置去探测和引导U-Boot和trust，主要是应对OTA升级过程中异常掉电引起的固件损坏，而无法启动的问题。

## boot.img

FIT方案如果作为SDK正式发布的feature，SDK编译完成后会生成FIT格式的boot.img。

如果要生成安全启动用的boot.img，必须把SDK生成的boot.img放到U-Boot工程下重新打包并签名，因为安全固件打包的签名工具、配置、参数等都来源于U-Boot工程。

## 安全启动

FIT方案支持安全启动，相关的feature：

- sha256 + rsa2048 + pkcs-v2.1(pss) padding
- 固件防回滚
- 固件重签名(远程签名)
- Crypto硬件加速

## 原理

### 校验流程

- Maskrom 校验 loader (包含SPL, ddr, usbplug)
- SPL 校验uboot.img (包含trust、U-Boot...)
- U-Boot校验boot.img (包含kernel, fdt, ramdisk...)

目前默认只支持 sha256+rsa2048+pkcs-v2.1(pss) padding 的安全校验模式。

### key存放

RSA key被mkimage打包在u-boot.dtb和u-boot-spl.dtb中，然后它们再被打包进u-boot.bin和u-boot-spl.bin。

u-boot.dtb里RSA key的格式如下（同理u-boot-spl.dtb）：

```
cjh@ubuntu:~/uboot-nextdev$ fdtdump u-boot.dtb | less  
/dts-v1/;  
....  
/  
 {  
 #address-cells = <0x00000001>;  
 #size-cells = <0x00000001>;  
 compatible = "rockchip,rv1126-evb", "rockchip,rv1126";  
 model = "Rockchip RV1126 Evaluation Board";  
  
 // signature节点由mkimage工具自动插入生成，节点里保存了RSA-SHA算法类型、RSA核心因子参数等信息。  
 signature {  
 key-dev {  
 required = "conf";  
 algo = "sha256,rsa2048";
```

```

        rsa,np = <0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x1327f633 0x00000003 0x00000003 0x00000003
0xc7aead6a 0xb4c79f40 0xa82bdf76 0xfb2f8387 0xa1e06dce 0xd451a706 0xc7f865e3
0x3e2d7ca8 0x6a71762e 0x125f1828 0x36ab1a41 0xb7e9e852 0x7bd0011a 0x7279e0b8
0xf37e189c 0x8cf00963 0x00000100 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000377 0x00000004
0x00000004 0x00000004 0x00000002 0x00000003 0x69616c40 0x00000003 0x6d634066
0x00000010 0x66633630 0x73797363>;
    rsa,c = <0x00000000>;
    rsa,r-squared = <0x00000000>;
    rsa,modulus = <0xc25ae693 0xc359f2a4 0xa866c89d 0xb7b1994f
0xf9f9f690 0x518d54a7 0xda0b83e8 0x06606e12 0x6ad1cbf9 0x92438edd 0x81e039c0
0x5d7322cc 0x124cdc80 0xa0c3288a 0x9265c3ae 0x6ac47a4b 0x00000003 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000008 0x00000003 0x00000003 0x00000003 0x00000002 0x73657300
0x2f736572 0x00000000 0x2f64776d 0x00000003 0x6d634066 0x00000001 0x30303000
0x726f636b 0x67726600 0x00000008 0x00000003 0x00000004 0x00000001 0x30303000
0x726f636b 0x706d7567 0x00000003 0x00001000 0x00000003 0x00000002 0x6e616765
0x30000000 0x726f636b 0x706d7500 0x00000008>;
    rsa,exponent-BN = <0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000003 0x00010001
0xe95771c5 0x00000800 0x64657600 0x616c6961 0x0000002c 0x30303030 0x00000034
0x30303000 0x2f64776d 0x00000002 0x65303030 0x0000001b 0x3132362d 0x00000003
0x00020000 0x00000003 0x00000002 0x65303230 0x0000001b 0x3132362d 0x6e000000
0xfe020000 0x00000042 0x0000006d 0x722d6d61 0x65303030 0x0000001b 0x3132362d
0x00000003 0x00001000 0x00000002 0x6e74726f 0x30000000 0x726f636b 0x706d7563
0x0000003e 0x00000004 0x00000004 0x00000004 0x00000000 0x00000050 0x636c6f63
0x40666634 0x00000014 0x2c727631 0x00000008>;
    rsa,exponent = <0x00000000 0x00000368>;
    rsa,n0-inverse = <0xe95771c5>;
    rsa,num-bits = <0x00000800>;
    key-name-hint = "dev";
};

};

}

```

SPL 支持烧写 key hash 的功能，u-boot-spl.dtb 的 key-dev 会多出 `burn-key-hash = <0x00000001>`。

## key使用

从 Maskrom 到 kernel 为止的安全启动，统一使用一把 RSA 公钥完成安全校验：

- Maskrom 校验 loader。

RSA 公钥需要使用 PC 工具 `rk_sign_tool` 写入 loader 的文件头中。安全启动时，Maskrom 首先从 loader 固件头中获取 RSA 公钥并校验合法性；然后再使用该公钥校验 loader 的固件签名。

`rk_sign_tool` 可从 rkbin 仓库中获取，U-Boot 会自动完成对 loader 的签名。

- SPL 校验 U-Boot 和 trust。

SPL 把 RSA 公钥保存在 u-boot-spl.dtb 中，u-boot-spl.dtb 会被打包进 u-boot-spl.bin 文件（最后打包进 loader）；安全启动时 SPL 从自己的 dtb 文件中拿出 RSA 公钥对 uboot.img 进行安全校验。

- U-Boot 校验 boot。

U-Boot把RSA公钥保存在u-boot.dtb中，u-boot.dtb会被打包进u-boot.bin文件（最后打包为uboot.img）；安全启动时U-Boot从自己的dtb文件中拿RSA公钥对boot.img进行校验。

所以当前这级的RSA Key已经作为自身固件的一部分，由前一级loader完成了安全校验，从而保证了Key的安全。

## 签名存放

RSA的签名结果被保存在itb文件中；被签名内容由 `hashed-nodes` 指定：包括了整个 `conf` 节点的属性、被打包固件的节点等。

如下是u-boot.itb的签名信息，同理boot.itb：

```
cjh@ubuntu:~/uboot-nextdev$ fdtdump uboot.img | less
/dts-v1/;

.....


configurations {
    default = "conf";
    conf {
        description = "Rockchip armv7 with OP-TEE";
        // 当前的固件版本号
        rollback-index = <0x0000001c>;
        firmware = "optee";
        loadables = "uboot";
        fdt = "fdt";

        // 被签名内容和签名结果，由mkimage自动插入
        signature {
            hashed-strings = <0x00000000 0x000000da>;
            // 指定被签名内容
            hashed-nodes = "/", "/configurations", "/configurations/conf",
                "/images/fdt", "/images/fdt/hash", "/images/optee", "/images/optee/hash",
                "/images/uboot", "/images/uboot/hash";
            // 进行签名的时间、签名者、版本
            timestamp = <0x5e9427b4>;
            signer-version = "2017.09-g8bb63db-200413-dirty #cjh";
            signer-name = "mkimage";
            // 签名结果！！(采用sha256+rsa2048)
            value = <0x78397d5d 0xb9219a0b 0xa7cb91a7 0xe1f32867 0x62719d9b
0x8901200c 0xfcba03a 0x1295ccc8 0x1cff9608 0xdf5f69d2 0x21391225 0x7af10ca7
0x5527864f 0xb13f527e 0xddf9ee62 0xea50199d 0x00000003 0x35362c72 0x00000004
0x00000017 0x77617265 0x00000002 0x00000009 0x23616464 0x6d616765 0x73006172
0x6f6e006c 0x72790064 0x61636b2d 0x7265006c 0x006b6579 0x69676e2d 0x706f7369
0x7a650074 0x75650073 0x69676e65 0x73686564 0x642d7374 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000>;
            algo = "sha256,rsa2048";
            key-name-hint = "dev";
            sign-images = "fdt", "firmware", "loadables";
        };
    };
};
```

- 安全启动支持对boot.img和uboot.img分别指定当前固件版本号，如果当前固件版本号小于机器上的最小版本号，则不允许启动。
- 最小版本号的更新：完成安全校验且确认系统可以正常启动后，被更新到OTP或安全存储中。

## 前期准备

### Key

U-Boot工程下执行如下三条命令可以生成签名用的RSA密钥对。通常情况下只需要生成一次，此后都用这对密钥签名和验证固件，请妥善保管。

```
// 1. 放key的目录: keys
mkdir -p keys

// 2. 使用RK的"rk_sign_tool"工具生成RSA2048的私钥privateKey.pem和publicKey.pem，分别
更名存放到: keys/dev.key和keys/dev.pubkey。命令为:
./rkbin/tools/rk_sign_tool kk --bits 2048 --out .

// 3. 使用-x509和私钥生成一个自签名证书: keys/dev.crt (效果本质等同于公钥)
openssl req -batch -new -x509 -key keys/dev.key -out keys/dev.crt
```

如果报错用户目录下没有.rnd文件：

```
Can't load /home4/cjh//.rnd into RNG
140522933268928:error:2406F079:random number generator:RAND_load_file:Cannot open
file:../crypto/rand/randfile.c:88:Filename=/home4/cjh//.rnd
```

请先手动创建：touch ~/.rnd

ls keys/ 查看结果：

```
dev.crt  dev.key  dev.pubkey
```

注意：上述的"keys"、"dev.key"、"dev.crt"、"dev.pubkey"名字都不可变。因为这些名字已经在its文件中静态定义，如果改变则会打包失败。

### 配置

U-Boot的defconfig打开如下配置：

```
// 必选。
CONFIG_FIT_SIGNATURE=y
CONFIG_SPL_FIT_SIGNATURE=y

// 可选。
CONFIG_FIT_ROLLBACK_PROTECT=y      // boot.img防回滚
CONFIG_SPL_FIT_ROLLBACK_PROTECT=y // uboot.img防回滚
```

建议通过make menuconfig的方式选中配置后，再通过make savedefconfig更新原本的defconfig文件。这样可以避免因为强加defconfig配置而导致依赖关系不对，进而导致编译失败的情况。

### 固件

把SDK工程下生成的boot.img复制一份到U-Boot根目录下。

## 编译打包

## (1) 基础命令 (不防回滚) :

```
./make.sh rv1126 --spl-new --boot_img boot.img --recovery_img recovery.img
```

编译结果:

```
.....  
// 编译完成后，生成已签名的uboot.img和boot.img。  
start to sign rv1126_spl_loader_v1.00.100.bin  
.....  
sign loader ok.  
.....  
Image(signed, version=0): uboot.img (FIT with uboot, trust...) is ready  
Image(signed, version=0): recovery.img (FIT with kernel, fdt, resource...) is ready  
Image(signed, version=0): boot.img (FIT with kernel, fdt, resource...) is ready  
Image(signed): rv1126_spl_loader_v1.05.106.bin (with spl, ddr, usbplug) is ready  
pack uboot.img okay! Input: /home4/cjh/rkbin/RKTRUST/RV1126TOS.ini  
Platform RV1126 is build OK, with new .config(make rv1126-secure_defconfig)
```

## (2) 扩展命令1:

如果开启防回滚，必须对上述 (1) 追加rollback参数。例如:

```
// 指定 uboot.img和boot.img的最小版本号分别为10、12。  
./make.sh rv1126 --spl-new --boot_img boot.img --recovery_img recovery.img --  
rollback-index-uboot 10 --rollback-index-boot 12 --rollback-index-recovery 12
```

编译结果:

```
.....  
// 编译完成后，生成已签名的uboot.img和boot.img，且包含防回滚版本号。  
start to sign rv1126_spl_loader_v1.00.100.bin  
.....  
sign loader ok.  
.....  
Image(signed, version=0, rollback-index=10): uboot.img (FIT with uboot, trust)  
is ready  
Image(signed, version=0, rollback-index=12): recovery.img (FIT with kernel,  
fdt, resource...) is ready  
Image(signed, version=0, rollback-index=12): boot.img (FIT with kernel, fdt,  
resource...) is ready  
Image(signed): rv1126_spl_loader_v1.00.100.bin (with spl, ddr, usbplug) is ready
```

## (3) 扩展命令2:

如果要把公钥hash烧写到OTP/eFUSE，必须对上述 (1) 或 (2) 追加参数 `--burn-key-hash`。例如:

```
// 指定uboot.img和boot.img的最小版本号分别为10、12。  
// 要求SPL阶段把公钥hash烧写到OTP/eFUSE中。  
.make.sh rv1126 --spl-new --boot_img boot.img --recovery_img recovery.img --  
rollback-index-uboot 10 --rollback-index-boot 12 --rollback-index-recovery 12 --  
burn-key-hash
```

编译结果：

```
.....  
  
// 使能 burn-key-hash  
### spl/u-boot-spl.dtb: burn-key-hash=1  
  
// 编译完成后，生成已签名的uboot.img和boot.img，且包含防回滚版本号。  
start to sign rv1126_spl_loader_v1.00.100.bin  
.....  
sign loader ok.  
.....  
Image(signed, version=0, rollback-index=10): uboot.img (FIT with uboot, trust)  
is ready  
Image(signed, version=0, rollback-index=12): recovery.img (FIT with kernel,  
fdt, resource...) is ready  
Image(signed, version=0, rollback-index=12): boot.img (FIT with kernel, fdt,  
resource...) is ready  
Image(signed): rv1126_spl_loader_v1.00.100.bin (with spl, ddr, usbplug) is  
ready
```

上电开机时能看到SPL打印：RSA: Write key hash successfully。

#### (4) 注意事项：

- `--boot_img`：可选。指定待签名的boot.img。
- `--recovery_img`：可选。指定待签名的recovery.img。
- `--rollback-index-uboot`、`--rollback-index-boot`、`--rollback-index-recovery`：可选。指定防回滚版本号。
- `--spl-new`：如果编译命令不带此参数，则默认使用rkbin中的spl文件打包生成loader；否则使用当前编译的spl文件打包loader。

因为u-boot-spl.dtb中需要被打包进RSA公钥（来自于用户），所以RK发布的SDK不会在rkbin仓库提交支持安全启动的spl文件。因此，用户编译时要指定该参数。但是用户也可以把自己的spl版本提交到rkbin工程，此后编译固件时就可以不再指定此参数，每次都使用这个稳定版的spl文件。

- 编译后会生成三个固件：loader、uboot.img、boot.img，只要RSA key 没有更换，就允许单独更新其中的任意固件。

## 启动信息

如下是安全启动的信息：

```
BW=32 Col=10 Bk=8 CS0 Row=15 CS=1 Die BW=16 Size=1024MB  
out  
U-Boot SPL board init  
U-Boot SPL 2017.09-gacb99c5-200408-dirty #cjh (Apr 09 2020 - 20:51:21)  
unrecognized JEDEC id bytes: 00, 00, 00
```

```
Trying to boot from MMC1
// SPL完成签名校验
sha256,rsa2048:dev+
// 防回滚检测: 当前uboot.img固件版本号是10, 本机的最小版本号是9
rollback index: 10 >= 9, OK
// SPL完成各子镜像的hash校验
### Checking optee ... sha256+ OK
### Checking uboot ... sha256+ OK
### Checking fdt ... sha256+ OK

Jumping to U-Boot via OP-TEE
I/TC:
E/TC:0 0 plat_rockchip_pmu_init:2003 0
E/TC:0 0 plat_rockchip_pmu_init:2006 cpu off
E/TC:0 0 plat_rockchip_pmusram_prepare:1945 pmu sram prepare 0x14b10000
0x8400880 0x1c
E/TC:0 0 plat_rockchip_pmu_init:2020 pmu sram prepare
E/TC:0 0 plat_rockchip_pmu_init:2056 remap
I/TC: OP-TEE version: 3.6.0-233-g35ecf936 #1 Tue Mar 31 08:46:13 UTC 2020 arm
I/TC: Next entry point address: 0x00400000
I/TC: Initialized

U-Boot 2017.09-gacb99c5-200408-dirty #cjh (Apr 09 2020 - 20:51:21 +0800)

Model: Rockchip RV1126 Evaluation Board
PreSerial: 2
DRAM: 1023.5 MiB
Sysmem: init
Relocation Offset: 00000000, fdt: 3df404e0
Using default environment

dwmmc@ffc50000: 0
Bootdev(atags): mmc 0
MMC0: HS200, 200Mhz
PartType: EFI
boot mode: normal
conf: sha256,rsa2048:dev+
resource: sha256+
DTB: rk-kernel.dtb
FIT: signed, conf required
HASH(c): OK

I2c0 speed: 400000Hz
PMIC: RK8090 (on=0x10, off=0x00)
vdd_logic 800000 uV
vdd_arm 800000 uV
vdd_npu init 800000 uV
vdd_vepu init 800000 uV
.....
Hit key to stop autoboot('CTRL+C'): 0
### Booting FIT Image at 0x3d8122c0 with size 0x0052b200
Fdt Ramdisk skip relocation
### Loading kernel from FIT Image at 3d8122c0 ...
    Using 'conf' configuration
    // uboot完成签名校验
    Verifying Hash Integrity ... sha256,rsa2048:dev+ OK
```

```
// 防回滚检测: 当前boot.img固件版本号是22, 本机的最小版本号是21
Verifying Rollback-index ... 22 >= 21, OK
Trying 'kernel' kernel subimage
  Description: Kernel for arm
  Type: Kernel Image
  Compression: uncompressed
  Data Start: 0x3d8234c0
  Data Size: 5349248 Bytes = 5.1 MiB
  Architecture: ARM
  OS: Linux
  Load Address: 0x02008000
  Entry Point: 0x02008000
  Hash algo: sha256
  Hash value:
64b4a0333f7862967be052a67ee3858884fcfefebf4565db5c3828a941a15f34a
  Verifying Hash Integrity ... sha256+ OK // 完成kernel的hash校验
### Loading ramdisk from FIT Image at 3d8122c0 ...
Using 'conf' configuration
Trying 'ramdisk' ramdisk subimage
  Description: Ramdisk for arm
  Type: RAMDisk Image
  Compression: uncompressed
  Data Start: 0x3dd3d4c0
  Data Size: 0 Bytes = 0 Bytes
  Architecture: ARM
  OS: Linux
  Load Address: 0x0a200000
  Entry Point: unavailable
  Hash algo: sha256
  Hash value:
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
  Verifying Hash Integrity ... sha256+ OK // 完成ramdisk的hash校验
  Loading ramdisk from 0x3dd3d4c0 to 0x0a200000
### Loading fdt from FIT Image at 3d8122c0 ...
Using 'conf' configuration
Trying 'fdt' fdt subimage
  Description: Device tree blob for arm
  Type: Flat Device Tree
  Compression: uncompressed
  Data Start: 0x3d812ec0
  Data Size: 66974 Bytes = 65.4 KiB
  Architecture: ARM
  Load Address: 0x08300000
  Hash algo: sha256
  Hash value:
8fb1f170766270ed4f37cce4b082a51614cb346c223f96ddfe3526fafc5729d7
  Verifying Hash Integrity ... sha256+ OK // 完成fdt的hash校验
  Loading fdt from 0x3d812ec0 to 0x08300000
  Booting using the fdt blob at 0x8300000
  Loading Kernel Image from 0x3d8234c0 to 0x02008000 ... OK
  Using Device Tree in place at 08300000, end 0831359d
  Adding bank: 0x00000000 - 0x08400000 (size: 0x08400000)
  Adding bank: 0x0848a000 - 0x40000000 (size: 0x37b76000)
  Total: 236.327 ms

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0xf00
```

```
[    0.000000] Linux version 4.19.111 (cjh@ubuntu) (gcc version 6.3.1 20170404  
(Linaro GCC 6.3-2017.05)) #28 SMP PREEMPT Wed Mar 25 16:03:27 CST 2020  
[    0.000000] CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=10c5387d
```

## 远程签名

从上述的章节可以看出，制作安全固件时要求用户在本地PC上完成，即用户必须持有：RSA密钥对和固件。但在实际场景中，用户可能需要把固件上传到远程服务器，由服务器持有RSA私钥完成签名，然后把签名过的固件返回给本地用户。对于这种情况，RK的FIT方案上需要通过“重签名”实现。

### 实现思路

- 因为只能拿到服务器的公钥，所以用户先用临时私钥+服务器公钥在本地PC上对固件进行一次打包签名，会生成带有临时签名的安全固件和被签名数据；
  - 公钥的作用是为了把公钥打包进dtb文件，在安全启动流程时使用；私钥的作用是做临时签名。
- 用户把被签名数据发送给服务器即可（不需整个固件，更节省时间），服务器使用私钥对被签名数据进行签名，然后把签名返回给用户；
- 用户使用这份签名替换安全固件中的临时签名即可获得最后用于烧写的安全固件。

### 被签名数据

上述章节提到的被签名数据包含：fdt blob配置 + 子镜像hash集合。

- fdt blob 节点配置
  - hashed-nodes 指定了一系列节点，这些节点的内容都会纳入被签名数据。

```
cjh@ubuntu:~/uboot-nextdev$ fdtdump uboot.img | less  
/dts-v1/;  
.....  
  
configurations {  
    default = "conf";  
    conf {  
        description = "Rockchip armv7 with OP-TEE";  
        rollback-index = <0x00000001c>;  
        firmware = "optee";  
        loadables = "uboot";  
        fdt = "fdt";  
  
        signature {  
            hashed-strings = <0x00000000 0x000000da>;  
            // 这些节点的内容都会纳入被签名数据  
            hashed-nodes = "/", "/configurations/conf", "/images/fdt",  
                "/images/fdt/hash", "/images/optee", "/images/optee/hash", "/images/uboot",  
                "/images/uboot/hash";  
            .....  
        }  
    }  
};
```

- 子镜像hash的集合。

mkimage会为各个子镜像自动生成hash值，并追加进hash节点。`sign-images`指定的所有子镜像hash值都会纳入被签名数据（本质是通过 `hashed-nodes` 进行指定了hash节点）。例如：

```
cjh@ubuntu:~/uboot-nextdev/u-boot$ fdtdump fit/u-boot.itb | less
```

```

/dts-v1/;

.....


/ {
    totalsize = <0x000bb600>;
    timestamp = <0x5ecb3553>;
    description = "Simple image with OP-TEE support";
    #address-cells = <0x00000001>;
    images {
        uboot {
            data-size = <0x0007ed54>;
            data-position = <0x00000a00>;
            description = "U-Boot";
            type = "standalone";
            os = "U-Boot";
            arch = "arm";
            compression = "none";
            load = <0x00400000>;
            hash {
                // uboot镜像的hash, 由mkimage工具自动计算生成
                value = <0xedaa8cd52 0x8f058118 0x00000003 0x35360000
0x6f707465 0x0000009f 0x00000091 0x00000000>;
                algo = "sha256";
            };
        };
    .....
}

```

## 具体步骤

用于签名固件的RSA密钥对是：dev.key、dev.pubkey和dev.crt。dev.key作为私钥由远程服务器持有，用户只有dev.pubkey和dev.crt。

### 步骤1：

在本地U-Boot工程环境下：用户把dev.crt放到keys目录下，然后用RK的“rk\_sign\_tool”工具随机生成一把临时私钥，命名为dev.key放到keys目录下。参考上面的章节（但是编译参数要追加--no-check）生成签名固件uboot.img和boot.img（实际最后不会被使用，用户需要的是中间文件）。

注意：**编译命令要指定参数 --no-check**，否则会因为dev.key和dev.crt不匹配导致打包脚本自校验失败。比如：

```

./make.sh rv1126 --spl-new --boot_img boot.img --rollback-index-uboot 10 --
rollback-index-boot 12 --no-check

```

除了生成签名固件uboot.img和boot.img，用户还可以在fit/目录下得到中间文件：

```

// 被签名内容(data2sign意为: data to sign)
fit/uboot.data2sign
fit/boot.data2sign

// 已签名itb文件(使用临时私钥)，我们的img文件由它们进行多备份后获得。
fit/uboot.itb
fit/boot.itb

```

### 步骤2：

用户把uboot.data2sign发送给远程服务器。假设远程服务器持有的私钥为dev.key，使用如下命令签名并输出签名结果：uboot.sig

```
openssl dgst -sha256 -sign dev.key -sigopt rsa_padding_mode:pss -out uboot.sig  
uboot.data2sign
```

服务器把签名结果文件uboot.sig返回给用户，用户使用uboot.sig替换uboot.itb中的临时签名：

```
./scripts/fit-resign.sh -f fit/uboot.itb -s uboot.sig // 会生成新的uboot.img，用于  
烧写
```

同理boot.itb文件。由此用户获得了最终有效的签名固件uboot.img和boot.img。

#### 注意事项：

- fit-resign.sh时-f指定的itb文件，不是img文件。脚本会对itb重签名后生成img文件。
- 执行fit-resign.sh时用的itb文件必须是步骤1编译生成的，即itb文件和data2sign文件是一对一对应的，因为data2sign信息中包含了生成itb文件的时间戳，即`/timestamp = <...>`。所以即使当前没有任何代码改动，重新编译获得一个新的uboot.itb，把uboot.sig替换进新的uboot.itb中也会引起安全启动失败！
- 由于没有私钥，loader需要单独发送到服务器端进行签名。

## 其它方案

除了“重签名”方式，是否可以直接上传整个固件（boot.img, uboot.img）或分立镜像（u-boot.bin, fdt, ramdisk, kernel ...）给服务器进行签名？

基于FIT的设计原理和实现，其它方案的实现比较困难。如下进行说明：

- 方案一：上传非安全的boot.img, uboot.img给服务器重新打包+签名  
问题点：还需要上传本地U-Boot编译环境下的配置信息、u-boot-spl.bin文件等。
- 方案二：上传安全的boot.img, uboot.img给服务器重新打包+签名  
问题点：本地编译固件时已经打包了RSA公钥，服务器会进行RSA公钥二次打包。
- 方案三：上传所有分立镜像（kernel, dtb, ramdisk, resource...）进行打包+签名  
问题点：上传文件太多，比较繁琐，而且同样存在方案一的问题。

以上方案的共同问题点：服务器端必须使用RK的mkimage工具，而这个工具有可能被RK更新。

所以目前的“重签名”是操作最简便、没有依赖、最不容易出错的方案：用户只需上传被签名数据，服务器使用openssl命令签名即可。

## 固件解包

用户可以借助脚本对固件解包，例如boot.img：

```
cjh@ubuntu:~/uboot-nextdev$ ./scripts/fit-unpack.sh -f boot.img -o out  
Unpack to directory out:  
fdt : 82813 bytes... sha256+  
kernel : 5844640 bytes... sha256+  
ramdisk : 0 bytes... sha256+  
resource : 120832 bytes... sha256+
```

如果img包含多备份，脚本只解包第一份itb；sha256+表示固件没有损坏，否则显示sha256-。

## 固件替换

用户可以借助脚本批量替换子固件。例如：用out目录里存在的子固件去替换uboot.img里同名的子固件。

```
cjh@ubuntu:~/uboot-nextdev$ ./scripts/fit-repack.sh -f uboot.img -d out/
Unpack to directory out/repack/:
uboot           : 6 bytes... sha256+
optee           : 6 bytes... sha256+
fdt             : 4 bytes... sha256+
...
Image(repack): uboot.img is ready
```

## 安全校验Step-by-Step

1. 进入u-boot目录，打开对应平台的configs/rxxxxx\_defconfig，选择如下配置：

```
// 必选。
CONFIG_FIT_SIGNATURE=y
CONFIG_SPL_FIT_SIGNATURE=y

// 可选。
CONFIG_FIT_ROLLBACK_PROTECT=y      // boot.img防回滚
CONFIG_SPL_FIT_ROLLBACK_PROTECT=y  // uboot.img防回滚
```

2. 执行如下操作生成keys：

```
mkdir -p keys
./rkbin/tools/rk_sign_tool kk --bits 2048 --out .
cp privateKey.pem keys/dev.key && cp publicKey.pem keys/dev.pubkey
openssl req -batch -new -x509 -key keys/dev.key -out keys/dev.crt
```

注意：该步骤执行一次即可，然后妥善保存这些keys。

3. 编译签名，以rv1126为例（如果编译签名其他芯片固件，如rk3566，将下列命令内的rv1126改为rk3566即可）：

```
// Linux: 拷贝boot.img, recovery.img到u-boot文件下，执行下列脚本签名
loader,uboot,boot,recovery，设置uboot,boot,recovery的防版本回滚号，注意防版本回滚号依据
需要配置
./make.sh rv1126 --spl-new --boot_img boot.img --recovery_img recovery.img --
rollback-index-uboot 1 --rollback-index-boot 2

// Android: 签名loader,uboot，设置uboot的防版本回滚号，注意防版本回滚号依据需要配置
./make.sh rv1126 --spl-new --rollback-index-uboot 1
```

如果编译出现：

```
Can't load xxxxxx//.rnd into RNG
```

执行：

```
touch ~/.rnd
```

#### 4. 公钥hash烧写:

```
// Linux: 拷贝boot.img, recovery.img到u-boot文件下, 执行下列脚本签名  
loader, uboot, boot, recovery, 设置uboot, boot, recovery的防版本回滚号, 注意防版本回滚号依据  
需要配置, 使能烧写key hash  
.make.sh rv1126 --spl-new --boot_img boot.img --recovery_img recovery.img --  
rollback-index-uboot 1 --rollback-index-boot 2 --burn-key-hash  
  
// Android: 签名loader, uboot, 设置uboot的防版本回滚号, 注意防版本回滚号依据需要配置, 使能烧  
写key hash  
.make.sh rv1126 --spl-new --rollback-index-uboot 1 --burn-key-hash
```

注意：该步骤在整个产品开发验证完后再配置`--burn-key-hash`，否则安全开启，产品开发过程中每次只能更新签名过的固件。

#### 5. Android其他固件签名：

参考《Rockchip\_Developer\_Guide\_Secure\_Boot\_for\_UBoot\_Next\_Dev\_CN.md》

## Chapter-13 快速开机

### 芯片支持

- rv1126

### 存储支持

- eMMC
- spi nor

### bootrom 支持

目前 bootrom 的 spi nor 驱动支持四线DMA模式加载下级固件，这项支持已直接在 usbplug 烧写固件时做了配置，客户无需再配置。

eMMC目前无此优化。

### U-Boot SPL 支持

U-Boot SPL 下支持 fit 格式的快速开机，同时支持按键进入loader模式和低电检测。

配置：

```
CONFIG_SPL_KERNEL_BOOT=y          // 开启快速开机功能  
CONFIG_SPL_BLK_READ_PREPARE=y    // 开启预加载功能  
CONFIG_SPL_MISC_DECOMPRESS=y     // 开启解压功能  
CONFIG_SPL_ROCKCHIP_HW_DECOMPRESS=y
```

U-Boot SPL 支持预加载功能，使能预加载功能后，可以在执行其他程序的同时加载固件。目前主要用  
来预加载ramdisk。

例如预加载经过 gzip 压缩过的 ramdisk，压缩命令：

```
cat ramdisk | gzip -n -f -9 > ramdisk.gz
```

its文件的配置如下：

```
ramdisk {  
    data = /incbin/("./images-tb/ramdisk.gz");  
    compression = "gzip"; // 压缩格式  
    type = "ramdisk";  
    arch = "arm";  
    os = "linux";  
    preload = <1>; // 预加载标志  
    comp = <0x5800000>; // 加载地址  
    load = <0x2800000>; // 解压地址  
    decomp-async; // 异步解压  
    hash {  
        algo = "sha256";  
        uboot-ignore = <1>; // 不做hash校验  
    };  
};
```

编译固件，比如编译rv1126 eMMC固件：

```
./make.sh rv1126-emmc-tb && ./make.sh --spl
```

## mcu配置

目前mcu的主要作用是辅助系统启动，对ISP等模块提前做初始化。kernel启动后，会接回这些硬件模块的控制权。

在 rkbin/RKTRUST 对应的芯片文件内配置，以 rv1126 为例：

```
[MCU]  
MCU=bin/rv11/rv1126_mcu_v1.02.bin,0x108000,okay // 配置对应固件位置，启动地址和使能标志
```

mcu程序地址：

```
https://10.10.10.29/admin/repos/rtos/rt-thread/rt-thread-amp  
https://10.10.10.29/admin/repos/rk/mcu/hal
```

U-Boot编译后，会将mcu固件打包到uboot.img内。系统启动时，SPL会从uboot.img中解析加载mcu固件。

## kernel 支持

配置：

```
CONFIG_ROCKCHIP_THUNDER_BOOT=y // 开启快速开机功能  
CONFIG_ROCKCHIP_THUNDER_BOOT_MMC=y // 开启支持emmc快速开机优化功能  
CONFIG_ROCKCHIP_THUNDER_BOOT_SFC=y // 开启支持spi nor快速开机优化功能  
CONFIG_VIDEO_ROCKCHIP_THUNDER_BOOT_ISP=y // 开启支持ISP快速开机优化功能
```

为了快速开机，SPL不会依据实际的硬件参数修改kernel dtb的参数，所以有些参数需要用户自己配置，具体有：

- memory

- ramdisk解压前后大小

详见: kernel/arch/arm/boot/dts/rv1126-thunder-boot.dtsi

```

memory: memory {
    device_type = "memory";
    reg = <0x00000000 0x20000000>; // 需要依据真实DDR容量预先定义, SPL不修正
};

reserved-memory {
    trust@0 {
        reg = <0x00000000 0x00200000>; // trust 空间
        no-map;
    };

    trust@200000 {
        reg = <0x00200000 0x00008000>;
    };

    ramoops@210000 {
        compatible = "ramoops";
        reg = <0x00210000 0x000f0000>;
        record-size = <0x20000>;
        console-size = <0x20000>;
        ftrace-size = <0x00000>;
        pmsg-size = <0x50000>;
    };

    rtos@300000 {
        reg = <0x00300000 0x00100000>; // 预留给用户端使用, 没有使用可以删掉
        no-map;
    };
};

ramdisk_r: ramdisk@2800000 {
    reg = <0x02800000 (48 * 0x00100000)>; // 解压源地址, 可以依据实际大小进行更改
};

ramdisk_c: ramdisk@5800000 {
    reg = <0x05800000 (20 * 0x00100000)>; // 压缩源地址, 可以依据实际大小进行更改
};
};

```

针对emmc的配置:

```

/ {
    reserved-memory {
        mmc_ecsd: mmc@20f000 {
            reg = <0x0020f000 0x00001000>; // SPL 给kernel上传ecsd区域
        };

        mmc_idmac: mmc@500000 {
            reg = <0x00500000 0x00100000>; // 预加载ramdisk时, 预留的
        idmac的内存区域, 预加载完成, 该区域内存释放掉
        };
    };

    thunder_boot_mmc: thunder-boot-mmc {

```

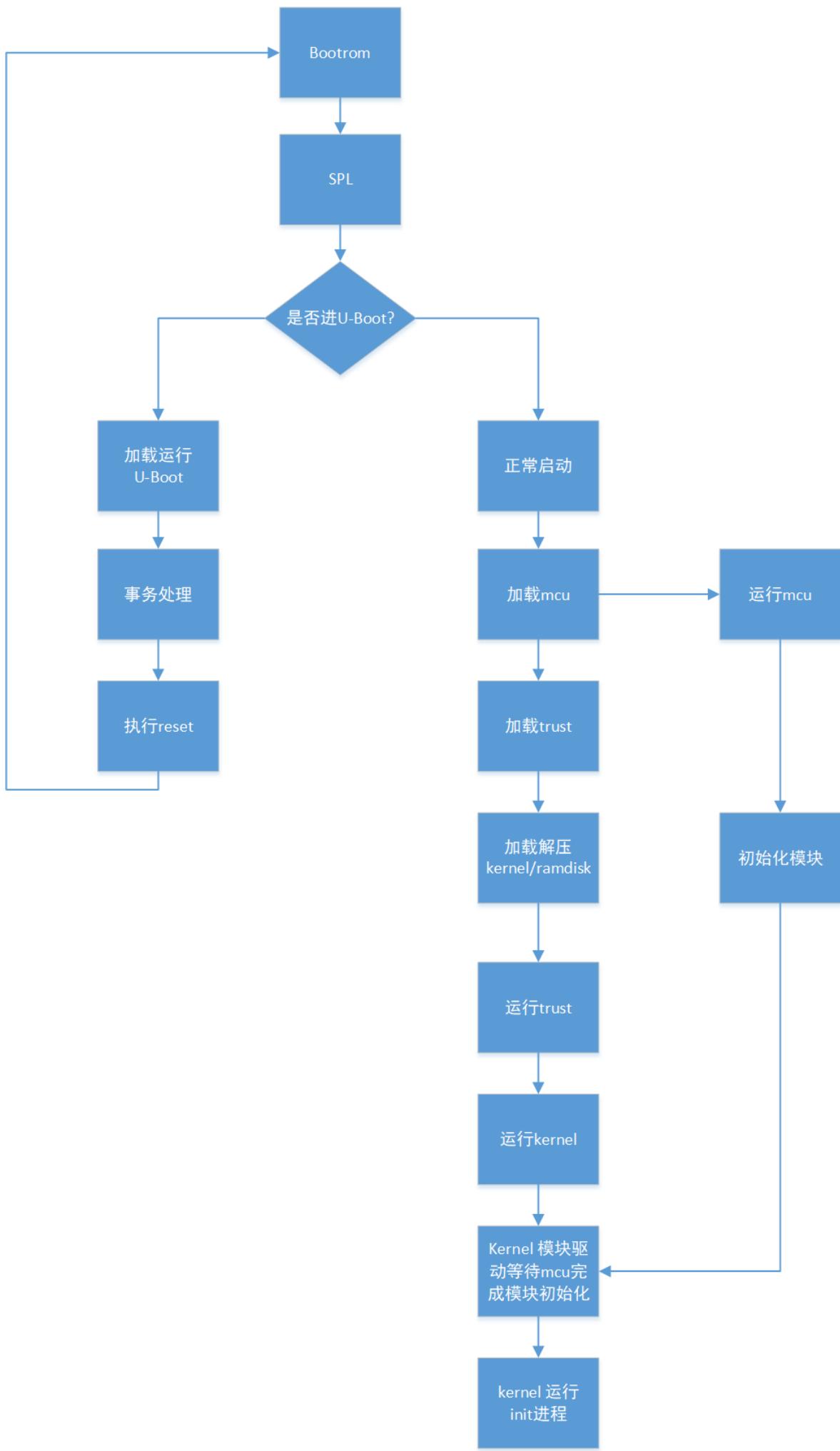
```
    compatible = "rockchip,thunder-boot-mmc";
    reg = <0xffc50000 0x4000>;
    memory-region-src = <&ramdisk_c>;
    memory-region-dst = <&ramdisk_r>;
    memory-region-idmac = <&mmc_idmac>;
};

};
```

针对spi nor的配置：

```
/ {
    thunder_boot_spi_nor: thunder-boot-spi-nor {
        compatible = "rockchip,thunder-boot-sfc";
        reg = <0xffc90000 0x4000>;
        memory-region-src = <&ramdisk_c>;
        memory-region-dst = <&ramdisk_r>;
    };
};
```

## 快速开机流程



# Chapter-14 注意事项

## SDK 兼容

### androidboot.mode 兼容

低于 Android 8.1 的 SDK 版本，U-Boot 必须开启如下配置才能正常启动 Android：

```
CONFIG_RKIMG_ANDROID_BOOTMODE_LEGACY
```

原因请参考提交：

```
commit a7774f5911624928ed1d9cfed5453aab206c512e
Author: Zhangbin Tong <zebulun.tong@rock-chips.com>
Date:   Thu Sep 6 17:35:16 2018 +0800

common: boot_rkimg: set "androidboot.mode=" as "normal" or "charger"

- The legacy setting rule is deprecated(Android SDK < 8.1).
- Provide CONFIG_RKIMG_ANDROID_BOOTMODE_LEGACY to enable legacy setting.

Change-Id: i5c8b442b02df068a0ab98ccc81a4f008ebe540c1
Signed-off-by: Zhangbin Tong <zebulun.tong@rock-chips.com>
Signed-off-by: Joseph Chen <chenjh@rock-chips.com>
```

## misc 兼容

misc.img的用途是作为U-Boot和Android之间的启动交互，主要内容为BCB(Bootloader Control Block)。

由于RK平台的历史原因，大于等于Android-10.0的SDK版本，misc.img里的BCB必须存放在misc分区偏移0位置；低于Android-10.0的版本，BCB必须存放在misc分区偏移16KB位置。

用户拿到发布的SDK后不需要额外处理，U-Boot会自适应兼容。但是用户如果拿不同SDK的misc.img混用，则可能出现问题。现象一般是Android会一直进入recovery模式。

# Chapter-15 工具

本章节相关的开发工具路径（以U-Boot根目录为参考点）：

```
./scripts/mkbootimg  
./scripts/unpack_bootimg  
./scripts/repack-bootimg  
./scripts/unpack_resource.sh  
./scripts/stacktrace.sh  
.tools/patman/patman  
.tools/buildman/buildman  
  
../rbin/tools/resource_tool  
../rbin/tools/loaderimage  
../rbin/tools/trust_merger  
../rbin/tools/boot_merger
```

## trust\_merger

功能：根据ini配置文件打包64位平台的 bl30、bl31、bl32 bin文件，生成 trust.img。

### ini 文件：

以 RK3368TRUST.ini 为例：

```
[VERSION]  
MAJOR=0          -----主版本号  
MINOR=1          -----次版本号  
[BL30_OPTION]    -----bl30，目前设置为mcu bin  
SEC=1            -----存在BL30 bin  
PATH=tools/rk_tools/bin/rk33/rk3368bl30_v2.00.bin  -----指定bin路径  
ADDR=0xff8c0000 -----固件DDR中的加载和运行地址  
[BL31_OPTION]    -----bl31，目前设置为多核和电源管理相关的bin  
SEC=1            -----存在BL31 bin  
PATH=tools/rk_tools/bin/rk33/rk3368bl31-20150401-v0.1.bin-----指定bin路径  
ADDR=0x00008000 -----固件DDR中的加载和运行地址  
[BL32_OPTION]  
SEC=0            -----不存在BL32 bin  
[BL33_OPTION]  
SEC=0            -----不存在BL33 bin  
[OUTPUT]  
PATH=trust.img [OUTPUT]      -----输出固件名字
```

### 打包命令：

```
/*  
 * @<sha>: 可选。sha相关，参考make.sh  
 * @<rsa>: 可选。rsa相关，参考make.sh  
 * @<size>: 可选，格式：--size [KB] [count]。输出文件大小，省略时默认单份2M，打包2份  
 * @[ini file]: 必选。ini文件  
 */  
  
.tools/trust_merger <sha> <rsa> <size> [ini file]
```

范例：

```
./tools/trust_merger --rsa 3 --sha 2 ./ RKTRUST/RK3399TRUST.ini  
out:trust.img  
merge success(trust.img)
```

解包命令：

```
// @[input image]: 必选。用于解包的固件，一般是trust.img
```

```
./tools/trust_merger --unpack [input image]
```

范例：

```
./tools/trust_merger --unpack trust.img
```

```
File Size = 4194304
Header Tag:BL3X
Header version:256
Header flag:35
SrcFileNum:4
SignOffset:992
Component 0:
ComponentID:BL31
StorageAddr:0x4
ImageSize:0x1c0
LoadAddr:0x10000
Component 1:
ComponentID:BL31
StorageAddr:0x1c4
ImageSize:0x10
LoadAddr:0xff8c0000
Component 2:
ComponentID:BL31
StorageAddr:0xd4
ImageSize:0x48
LoadAddr:0xff8c2000
Component 3:
ComponentID:BL32
StorageAddr:0x21c
ImageSize:0x2e0
LoadAddr:0x8400000
unpack success
```

## boot\_merger

功能：根据ini配置文件打包 miniloader + ddr + usb plug，生成 loader固件。

ini 文件：

以 RK3288MINIALL.ini 文件为例：

```
[CHIP_NAME]
NAME=RK320A          ----芯片名称：“RK”加上与maskrom约定的4B芯片型号
[VERSION]
MAJOR=2              ----主版本号
MINOR=36             ----次版本号
[CODE471_OPTION]
NUM=1                ----code471，目前设置为ddr bin
Path1=tools/rk_tools/bin/rk32/rk3288_ddr_400MHz_v1.06.bin
[CODE472_OPTION]
NUM=1                ----code472，目前设置为usbplug bin
```

```
Path1=tools/rk_tools/bin/rk32/rk3288_usbplug_v2.36.bin  
[LOADER_OPTION]  
NUM=2  
LOADER1=FlashData          ----flash data, 目前设置为ddr bin  
LOADER2=FlashBoot          ----flash boot, 目前设置为miniloader bin  
FlashData=tools/rk_tools/bin/rk32/rk3288_ddr_400MHz_v1.06.bin  
FlashBoot=tools/rk_tools/bin/rk32/rk3288_miniloader_v2.36.bin  
[OUTPUT]                   ----输出文件名  
PATH=rk3288_loader_v1.06.236.bin
```

### 打包命令:

```
// @[ini file]: 必选。ini文件  
./tools/boot_merger [ini file]
```

### 范例:

```
./tools/boot_merger ./RKBOOT/RK3399MINIALL.ini  
out:rk3399_loader_v1.17.115.bin  
fix opt:rk3399_loader_v1.17.115.bin  
merge success(rk3399_loader_v1.17.115.bin)
```

### 解包命令:

```
// @[input image]: 必选。用于解包的固件，一般是loader文件  
./tools/boot_merger --unpack [input image]
```

### 范例:

```
./tools/boot_merger --unpack rk3399_loader_v1.17.115.bin  
unpack entry(rk3399_ddr_800MHz_v1.17)  
unpack entry(rk3399_usbplug_v1.15)  
unpack entry(FlashData)  
unpack entry(FlashBoot)  
unpack success
```

## loaderimage

### 功能:

- 打包u-boot.bin生成uboot.img
- 打包32位平台的tee bin生成trust.img

### 打包u-boot:

```
/*
 * @[input bin]: 必选。bin源文件
 * @[output image]: 必选。输出文件
 * @load_addr: 必选。加载地址
 * <size>: 可选，格式: --size [KB] [count]。输出文件大小，省略时默认单份1M，打包4份
 */
```

```
./tools/loaderimage --pack --uboot [input bin] [output image] [load_addr] <size>
```

范例：

```
./tools/loaderimage --pack --uboot ./u-boot.bin uboot.img 0x60000000 --size 1024
2

load addr is 0x60000000!
pack input u-boot.bin
pack file size: 701981
crc = 0xc595eb85
uboot version: U-Boot 2017.09-02593-gb6e59d9 (Feb 18 2019 - 13:58:53)
pack uboot.img success!
```

**解包u-boot：**

```
/*
 * @[input image]: 必选。解包源文件
 * @[output bin]: 必选。解包输出文件，任意名字均可
 */
```

```
./tools/loaderimage --unpack --uboot [input image] [output bin]
```

范例：

```
./tools/loaderimage --unpack --uboot uboot.img uboot.bin
unpack input uboot.img
unpack uboot.bin success!
```

**打包trust：**

```
/*
 * @[input bin]: 必选。bin文件
 * @[output image]: 必选。输出文件
 * @load_addr: 必选。加载地址
 * <size>: 可选。格式: --size [KB] [count]，输出文件大小，省略时默认单份1M，打包4份
 */
```

```
./tools/loaderimage --pack --trustos [input bin] [output image] [load_addr]
<size>
```

范例：

```
./tools/loaderimage --pack --trustos ./bin/rk32/rk322x_tee_v2.00.bin trust.img \
    0x80000000 --size 1024 2

load addr is 0x80000000!
pack input bin/rk32/rk322x_tee_v2.00.bin
pack file size: 333896
crc = 0x2de93b46
pack trust.img success!
```

#### 解包trust:

```
/*
 * @[input image]: 必选。解包源文件
 * @[output bin]: 必选。解包输出文件，任意名均可
 */

./tools/loaderimage --unpack --trustos [input image] [output bin]
```

范例:

```
./tools/loaderimage --unpack --trustos trust.img tee.bin
unpack input trust.img
unpack tee.bin success!
```

## resource\_tool

功能：用于打包任意资源文件，生成resource.img。

#### 打包命令:

```
./tools/resource_tool [--pack] [--image=<resource.img>] <file list>
```

范例:

```
./scripts/resource_tool ./arch/arm/boot/dts/rk3126-evb.dtb logo.bmp
logo_kernel.bmp
Pack to resource.img successed!
```

#### 解包命令:

```
./tools/resource_tool --unpack --image=<resource.img> [output dir]
```

范例:

```
./tools/resource_tool --unpack --image=resource.img ./out/

Dump header:
partition version:0.0
header size:1
index tbl:
    offset:1      entry size:1      entry num:3
Dump Index table:
entry(0):
```

```
path:rk-kernel.dtb
offset:4          size:33728
entry(1):
    path:logo.bmp
    offset:70        size:170326
entry(2):
    path:logo_kernel.bmp
    offset:403       size:19160
Unack resource.img to ./out successed!
```

## mkimage

功能：生成 SPL 模式下的Loader固件。

例如：通过下面的命令生成 Rockchip 的 bootrom 所需 IDBLOCK 格式，这个命令会同时修改 u-boot-tpl.bin 的头 4 个 byte 为 Bootrom 所需校验的 ID：

```
./tools/mkimage -n rk3328 -T rksd -d tpl/u-boot-tpl.bin idbloader.img
```

详细参考：

```
./doc/mkimage.1
```

## stacktrace.sh

功能：解析调用栈信息，请参考RK架构章节。

## mkbootimg

功能：打包固件生成boot和recovery.img，源文件来在android工程。

范例：

```
./scripts/mkbootimg --kernel zImage --second resource.img --ramdisk ramdisk.img
--out boot.img
```

## unpack\_bootimg

功能：用于boot和recovery.img解包，源文件来在android工程。

范例：

```
./scripts/unpack_bootimg --boot_img boot.img --out out/
```

## repack-bootimg

功能：替换boot和recovery.img中的固件。

范例：

```
// 例如：只替换kernel  
./scripts/repack-bootimg --boot_img boot.img --kernel zImage -o boot_repack.img  
  
// 例如：只替换resource  
./scripts/repack-bootimg --boot_img boot.img --second resource.img -o  
boot_repack.img
```

## pack\_resource.sh

功能：打包 ./tools/images/ 目录下的充电图片进resource.img。

范例：

```
./scripts/pack_resource.sh resource.img  
  
Pack ./tools/images/ & resource.img to resource.img ...  
Unpacking old image(resource.img):  
rk-kernel.dtb 1  
Pack to resource.img successed!  
Packed resources:  
rk-kernel.dtb battery_1.bmp battery_2.bmp battery_3.bmp battery_4.bmp  
battery_5.bmp battery_fail.bmp battery_0.bmp 8  
  
resource.img is packed ready
```

## buildman

功能：批量编译代码，非常适合用于验证当前平台的提交是否影响到其他平台。详细参考：

```
./tools/buildman/README
```

使用 buildman 需要提前设置好 toolchain 路径，编辑'~/.buildman'文件：

```
[toolchain]  
arm: ~/prebuilt/gcc/linux-x86/arm/gcc-linaro-6.3.1-2017.05-x86_64_arm-linux-  
gnueabihf/  
aarch64: ~/prebuilt/gcc/linux-x86/aarch64/gcc-linaro-6.3.1-2017.05-  
x86_64_aarch64-linux-gnu/
```

典型用例，如编译所有 Rockchip 平台的 U-Boot 代码：

```
./tools/buildman/buildman rockchip
```

理想结果如下：

```
$ ./tools/buildman/buildman rockchip  
boards.cfg is up to date. Nothing to do.  
Building current source for 34 boards (4 threads, 1 job per thread)  
34    0    0 /34      evb-rk3326
```

显示的结果中，第一个是完全 pass 的平台数量（绿色），第二个是含 warning 输出的平台数量（黄色），第三个是有 error 无法编译通过的平台数量（红色）。如果编译过程中有 warning 或者 error 会在终端上显示出来。

## patman

功能：python 写的工具，通过调用其他工具完成 patch 的检查提交，是做 patch Upstream (U-Boot、Kernel) 非常好用的必备工具。主要功能：

- 根据参数自动 format 补丁；
- 调用 checkpatch 进行检查；
- 从 commit 信息提取并转换成 upstream mailing list 所需的 Cover-letter、patch version、version changes 等信息；
- 自动去掉 commit 中的 change-id；
- 自动根据 Maintainer 和文件提交信息提取每个 patch 所需的收件人；
- 根据'~/.gitconfig'或者'./gitconfig'配置把所有 patch 发送出去。

详细参考：

```
./tools/patman/README
```

使用'-h'选项查看所有命令选项：

```
$ patman -h
Usage: patman [options]

Create patches from commits in a branch, check them and email them as
specified by tags you place in the commits. Use -n to do a dry run first.

Options:
-h, --help                  show this help message and exit
-H, --full-help              Display the README file
-c COUNT, --count=COUNT      Automatically create patches from top n commits
-i, --ignore-errors          Send patches email even if patch errors are found
-m, --no-maintainers         Don't cc the file maintainers automatically
-n, --dry-run                 Do a dry run (create but don't email patches)
-p PROJECT, --project=PROJECT
                           Project name; affects default option values and
                           aliases [default: u-boot]
-r IN_REPLY_TO, --in-reply-to=IN_REPLY_TO
                           Message ID that this series is in reply to
-s START, --start=START      Commit to start creating patches from (0 = HEAD)
-t, --ignore-bad-tags        Ignore bad tags / aliases
--test                      run tests
-v, --verbose                Verbose output of errors and warnings
--cc-cmd=CC_CMD              Output cc list for patch file (used by git)
--no-check                  Don't check for patch compliance
--no-tags                   Don't process subject tags as aliases
-T, --thread                 Create patches as a single thread
```

典型用例：提交最新的 3 个 patch

```
patman -t -c3
```

命令运行后 checkpatch 如果有 error 或者 warning 会自动 abort，需要修改解决 patch 解决问题后重新运行。

## 其他常用选项

- '-t' 标题中":"前面的都当成 TAG，大部分无法被 patman 识别，需要使用'-t'选项；
- '-i' 如果有些 warning (如超过 80 个字符) 我们认为无需解决，可以直接加'-i'选项提交补丁；
- '-s' 如果要提交的补丁并不是在当前 tree 的 top，可以通过'-s'跳过 top 的 N 个补丁；
- '-n' 如果并不是想提交补丁，只是想校验最新补丁是否可以通过 checkpatch，可以使用'-n'选项；

patchman 配合 commit message 中的关键字，生成 upstream mailing list 所需的信息。

典型的 commit：

```
commit 72aa9e3085e64e785680c3fa50a28651a8961feb
Author: Kever Yang <kever.yang@rock-chips.com>
Date:   wed Sep 6 09:22:42 2017 +0800

spl: add support to booting with OP-TEE

OP-TEE is an open source trusted OS, in armv7, its loading and
running are like this:
loading:
- SPL load both OP-TEE and U-Boot
running:
- SPL run into OP-TEE in secure mode;
- OP-TEE run into U-Boot in non-secure mode;

More detail:
<https://github.com/OP-TEE/optee\_os>
and search for 'boot arguments' for detail entry parameter in:
core/arch/arm/kernel/generic_entry_a32.s

Cover-letter:
rockchip: add tpl and OPTEE support for rk3229

Add some generic options for TPL support for arm 32bit, and then
and TPL support for rk3229(cortex-A7), and then add OPTEE support
in SPL.

Tested on latest u-boot-rockchip master.

END

Series-version: 4
Series-changes: 4
- use NULL instead of '0'
- add fdt_addr as arg2 of entry

Series-changes: 2
- Using new image type for op-tee

Change-Id: I3fd2b8305ba8fa9ea687ab7f3fd1ffd2fac9ece6
Signed-off-by: Kever Yang <kever.yang@rock-chips.com>
```

这个 patch 通过 patman 命令发送的时候，会生成一份 Cover-letter：

```
[PATCH v4 00/11] rockchip: add tpl and OPTEE support for rk3229
```

对应 patch 的标题如下，包含 version 信息和当前 patch 是整个 series 的第几封：

[PATCH v4,07/11] spl: add support to booting with OP-TEE

Patch 的 commit message 已经被处理过了，change-id 被去掉、Cover-letter 被去掉、version-changes 信息被转换成非正文信息：

```
OP-TEE is an open source trusted os, in armv7, its loading and
running are like this:
loading:
- SPL load both OP-TEE and U-Boot
running:
- SPL run into OP-TEE in secure mode;
- OP-TEE run into U-Boot in non-secure mode;

More detail:
<https://github.com/OP-TEE/optee\_os>
and search for 'boot arguments' for detail entry parameter in:
core/arch/arm/kernel/generic_entry_a32.S

Signed-off-by: Kever Yang <kever.yang@rock-chips.com>
---
Changes in v4:
- use NULL instead of '0'
- add fdt_addr as arg2 of entry

Changes in v3: None
Changes in v2:
- Using new image type for op-tee

common/spl/Kconfig      |  7 ++++++++
common/spl/Makefile      |  1 +
common/spl/spl.c         |  9 ++++++++
common/spl/spl_optee.s  | 13 ++++++ ++
include/spl.h            | 13 ++++++ ++
5 files changed, 43 insertions(+)
create mode 100644 common/spl/spl_optee.s
```

更多关键字使用，如"Series-prefix"、"Series-cc"等请参考 README。

## Chapter-16 FAQ

TODO

## Chapter-17 附录

### 下载地址

#### rkbin

- RK 内部工程师：登录 gerrit 搜索仓库：“rk/rkbin”
- 外部工程师（2选1）：
  - 下载 RK 发布的完整 SDK

- Github 下载: <https://github.com/rockchip-linux/rkbin>

## GCC

- RK 内部工程师: gerrit 搜索仓库: “gcc-linaro-6.3.1”
- 外部工程师: 下载 RK 发布的完整 SDK 或 Linaro官网下载

## 术语

- U-Boot: [Universal Boot Loader](#)
- AOSP: [Android Open-Source Project](#)
- AVB: Android Verified Boot
- DTB: [Device Tree Binary](#)
- DTS: [Device Tree Source](#)
- Fastboot: [原为 Android 的一种更新固件方式，现在已被广泛应用于嵌入式领域](#)
- GPT: [GUID Partition Table](#)
- MMC: [Multi Media Card](#), 包括: eMMC, SD 卡等
- SPL: Secondary Program Loader
- TPL: Tertiary Program Loader
- DTB: 名词, 设备树 Blob
- DTB: 名词, 用于叠加的设备树 Blob
- DTC: 名词, 设备树编译器
- DTO: 动词, 设备树叠加操作
- DTS: 名词, 设备树源文件
- FDT: 名词, 扁平化设备树