U-Boot next-dev开发指南

发布版本：1.0

作者邮箱：chenjh@rock-chips.com

```
Kever Yang <kever.yang@rock-chips.com>
```

日期：2018.02

文件密级：公开资料

前言

概述

本文主要指导读者如何在U-Boot next-dev分支进行项目开发。

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

各芯片feature支持状态

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

修订记录

日期	版本	作者	修改说明
2018-02-28	V1.0	陈健洪	初始版本

---

1. U-Boot next-dev简介

next-dev是Rockchip从U-Boot官方的v2017.09正式版本中切出来进行开发的版本。目前在该平台上已经支持RK所有主流在售芯片。

目前支持的功能主要有：

- 支持RK Android平台的固件启动；
- 支持最新Android AOSP(如GVA)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等显示设备；
- 支持Emmc、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分区格式；
- 支持开机logo显示、充电动画显示，低电管理、电源管理；
- 支持I2C、PMIC、CHARGE、GUAGE、USB、GPIO、PWM、GMAC、EMMC、NAND、中断等驱动；
- 支持RockUSB 和 Google Fastboot两种USB gadget烧写EMMC；
- 支持Mass storage, ethernet, HID等USB设备；
- 支持使用kernel的dtb；
- 支持dtbo功能；

U-Boot的doc目录下提供了很丰富的README文档，它们向开发者介绍了U-Boot里各个功能模块的概念、设计理念、实现方法等，建议读者好好利用这些文档提高开发效率。

2. 平台架构文件

2.1 SoC架构文件

各SoC的架构级文件在如下各自的芯片目录里，主要都是芯片级别的初始化代码。一般情况下普通用户不需要、也不要轻易修改它们。

头文件：

```
./arch/arm/include/asm/arch-rockchip/qos_rk3288.h
./arch/arm/include/asm/arch-rockchip/grf_rk3188.h
./arch/arm/include/asm/arch-rockchip/pmu_rk3288.h
./arch/arm/include/asm/arch-rockchip/grf_rk3368.h
./arch/arm/include/asm/arch-rockchip/grf_rk322x.h
......
```

驱动文件：

```
./arch/arm/mach-rockchip/rk3036/rk3036.c
./arch/arm/mach-rockchip/rk3066/sdram_rk3036.c
./arch/arm/mach-rockchip/rk3128/rk3128.c
./arch/arm/mach-rockchip/rk3188/rk3188.c
./arch/arm/mach-rockchip/rk322x/rk322x.c
......
```

2.2 board架构文件

由于每个项目硬件上的设计不同，Upstream U-Boot的设计是每块板子一份board实体,所以会存在不同的board驱动文件, 参考RK3288的板子可以明显看出这个结构, Rockchip为了简化板级支持, 引入支持kernel dtb的feature, 在U-Boot阶段共用eMMC dts和驱动, 而在PMIC/regulator, Display, IOMUX等存在板级差异的模块直接使用kernel dtb,使U-Boot可以一颗芯片共用一个evb配置.

头文件：

```
./include/configs/rk3368_common.h
./include/configs/evb_rk3288_rk1608.h
./include/configs/tinker_rk3288.h
./include/configs/evb_rk3288.h
./include/configs/vyasa-rk3288.h
......
```

驱动文件：

```
./board/rockchip/evb_px5/evb-px5.c
./board/rockchip/evb_rk3036/evb_rk3036.c
./board/rockchip/evb_rk3128/evb_rk3128.c
./board/rockchip/evb_rk3229/evb_rk3229.c
./board/rockchip/sheep_rk3368/sheep_rk3368.c
......
```

板级指导文档：

```
./board/rockchip/evb_rv1108/README
./board/rockchip/sheep_rk3368/README
./board/rockchip/gva_rk3229/README
./board/rockchip/evb_rk3399/README
./board/rockchip/evb_rk3328/README
......
```

这些文档可以有效指导开发者如何让自己的机器正常运行起来。

2.3 defconfig文件

每一款board都有相对应的defconfig文件：

```
./configs/evb-rk3328_defconfig
./configs/evb-rk3036_defconfig
./configs/evb-rk3229_defconfig
./configs/firefly-rk3288_defconfig
./configs/evb-rk3399_defconfig
......
```

如果新增一个defconfig文件，请遵循文件命名格式：**[board]-[chip]_defconfig**。

2.4 dts 文件

U-Boot使用的是kernel的dts文件。

2.5 宏配置介绍

[TODO]

3. 平台编译

3.1 准备

3.1.1 rkbin

rkbin工程主要存放了Rockchip不开源的bin文件（trust、loader等）、脚本、打包工具等，所以rkbin只是一个“工具包”工程 。

rkbin工程需要和U-Boot工程保持同级目录关系，否则编译时会报找不到rkbin仓库。当在U-Boot工程执行编译的时候，编译脚本会从rkbin仓库里索引相关的bin文件和打包工具，最后在U-Boot根目录下生成trust.img、uboot.img、loader等相关固件。

3.1.2 gcc工具链

默认使用的编译器是gcc-linaro-6.3.1版本：

```
32位编译器：gcc-linaro-6.3.1-2017.05-x86_64_arm-linux-gnueabihf
64位编译器：gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu
```

3.1.3 U-Boot分支

确认U-Boot工程里的代码使用的是next-dev分支：

```
remotes/origin/next-dev
```

开发者可以从U-Boot根目录下的./Makefile里获知版本（v2017-09）：

```
SPDX-License-Identifier:      GPL-2.0+

VERSION = 2017
PATCHLEVEL = 09
SUBLEVEL =
EXTRAVERSION =
NAME =
```

3.2 编译配置

3.2.1 gcc工具链路径指定

默认使用Rockchip提供的工具包：prebuilts，请保证它和U-Boot工程保持同级目录关系，确保gcc-linaro-6.3.1版本的编译器放到如下路径：

```
../prebuilts/gcc/linux-x86/arm/gcc-linaro-6.3.1-2017.05-x86_64_arm-linux-gnueabihf/bin
../prebuilts/gcc/linux-x86/aarch64/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu/bin
```

如果需要更改编译器路径，可以修改编译脚本./make.sh里的内容：

```
GCC_ARM32=arm-linux-gnueabihf-
GCC_ARM64=aarch64-linux-gnu-
TOOLCHAIN_ARM32=../prebuilts/gcc/linux-x86/arm/gcc-linaro-6.3.1-2017.05-x86_64_arm-linux-gnueabihf/bin
TOOLCHAIN_ARM64=../prebuilts/gcc/linux-x86/aarch64/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu/bin
```

3.2.2 menuconfig支持

U-Boot和Linux kernel一样，已经支持Kbuild编译机制，开发者可以使用 make menuconfig对某块进行开启或者关闭；使用make savedefconfig来保存配置修改。

3.2.3 编译

编译命令：

```
./make.sh [board]        ---- [board]的名字来源是：configs/[board]_defconfig文件。
```

无论32位或64位平台，只要确认好defconfig文件，直接执行上述的编译命令即可（编译脚本里执行make [board]_defconfig）。

命令范例：

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

3.2.4 固件生成

1. 编译最终打包生成的固件：trust、uboot、loader等，都在U-Boot根目录下：

```
./uboot.img
./trust.img
./rk3126_loader_v2.09.247.bin
```

2. 上述固件打包过程的提示信息如下，从打印可以知道打包用的原始二进制可执行文件的路径或者INI文件。

uboot.img打包提示：

```
 load addr is 0x60000000!
pack input rockdev/rk3126/out/u-boot.bin
pack file size: 478737
crc = 0x840f163c
uboot version: v2017.12 Dec 11 2017
pack uboot.img success!
pack uboot okay! Input: rockdev/rk3126/out/u-boot.bin
```

loader打包提示：

```
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/guest/project/rkbin/RKBOOT/RK3126MINIALL.ini
```

trust.img打包提示：

```
 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/guest/project/rkbin/RKTRUST/RK3126TOS.ini
```

3.2.5 辅助命令

为了调试方便，./make.sh会支持一些常用的命令，目前支持：”elf"：

```
./make.sh evb-px30 elf                  ----- 反汇编（默认使用objdmp -D参数）
```

其中反汇编命令的第三个参数，它的格式可以是elf[option]。例如：“elf-d”、“elf-D”、“elf-S”等，[option]会被用来做为objdump的参数，如果省略[option]，即“elf”，则会默认使用“-D”作为参数。

如果不清楚[option]有哪些参数可选，可以执行如下命令进行帮忙：

```
./make.sh evb-px30 elf-H                ----- 反汇编参数的help指导信息
```

3.2.6 烧写要求

Windows烧写工具版本必须是**v2.5**版本或以上；

3.2.7 分区表

1. 目前U-Boot支持parameter分区表和GPT分区表；
2. 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；
3. GPT和parameter分区表的具体格式请参考文档：《Rockchip-Parameter-File-Format-Version1.4.md》。

4. cache机制

目前所有芯片的cache配置都采用U-Boot提供的标准接口。

4.1 dcache和icache的开关

- CONFIG_SYS_ICACHE_OFF：如果定义，则关闭icache功能；否则打开。
- CONFIG_SYS_DCACHE_OFF：如果定义，则关闭dcache功能；否则打开。

目前Rockchip都默认使能了icache和dcache功能。

4.2 dcache 模式

- CONFIG_SYS_ARM_CACHE_WRITETHROUGH: 如果定义，则配置为 dcache writethrouch模式；
- CONFIG_SYS_ARM_CACHE_WRITEALLOC: 如果定义，则配置为 dcache writealloc模式；
- 如果上述两个宏都没有配置，则默认为dcache writeback 模式；

目前Rockchip都默认选择dcache writeback模式。

4.3 icache/dcache操作的常用接口

icache 的常用接口：

```
void icache_enable (void);
void icache_disable (void);
void invalidate_icache_all(void);
```

dcache 的常用接口：

```
void dcache_disable (void);
void enable_caches(void);
void flush_cache(unsigned long, unsigned long);
void flush_dcache_all(void);
void flush_dcache_range(unsigned long start, unsigned long stop);
void invalidate_dcache_range(unsigned long start, unsigned long stop);
void invalidate_dcache_all(void);
```

5. 驱动支持

5.1 中断驱动

5.1.1 框架支持

中断功能方面，U-Boot框架默认没有给与足够的支持，因此我们自己实现了一套中断框架机制来支持中断管理功能（支持GICv2/v3）。

驱动代码：

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

5.1.2 相关接口

1. 开关**CPU**本地中断

```
void enable_interrupts(void);
int disable_interrupts(void);
```

2. 申请**IRQ**

目前支持3种方式把gpio转换成对应的irq。

法1：传入标准gpio框架的struct gpio_desc结构体：

```
int gpio_to_irq(struct gpio_desc *gpio);

*此方法可以动态解析dts里的配置信息，比较灵活，常用。
```

节点范例：

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

代码范例：

```
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);
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);
}
```

法2：传入gpio phandle和pin

```
int phandle_gpio_to_irq(u32 gpio_phandle, u32 pin);     --参考：./drivers/input/rk8xx_pwrkey.c

*此方法可以动态解析dts里的配置信息，比较灵活，常用。
```

节点范例：如下是rk817的中断引脚GPIO0_A7的信息：

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

代码范例：

```
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;
}

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);
```

法**3**：强制指定**GPIO**引脚

```
int hard_gpio_to_irq(unsigned gpio);

*此方法直接强制指定gpio的方法，传入的gpio必须通过Rockchip特殊的宏来声明才行。不够灵活，比较少用。
```

代码范例：如下是对GPIO0_A0申请中断：

```
int gpio0_a0, irq;

gpio = RK_IRQ_GPIO(RK_GPIO0, RK_PA0);
irq = hard_gpio_to_irq(gpio0_a0);
irq_install_handler(irq, ...);
irq_handler_enable(irq);
```

3. 使能**/**注册**/**注销**handler**

```
void irq_install_handler(int irq, interrupt_handler_t *handler, void *data);
void irq_free_handler(int irq);
int irq_handler_enable(int irq);
int irq_handler_disable(int irq);
int irq_set_irq_type(int irq, unsigned int type);
```

5.2 clock支持

驱动代码位于drivers/clk/rockchip目录, 每颗芯片有一份独立的驱动. 驱动probe时会调用rkclk_init()函数对CPU和通用BUS进行初始化, 其他模块的clock如eMMC, I2C等在各自的驱动初始化时调用clk_get_by_indel()或者clk_get_by_name()获取clk句柄, 然后调用clk_set_rate()进行设置.

U-Boot只提供了已使用设备的clock驱动, 没有提供整个SoC完整的clock驱动, 所以如果新增驱动需要先确认clock驱动中是否有相应接口.

[TODO] assigned-clocks CPU clock init

5.3 GPIO驱动

5.3.1 框架支持

GPIO走的是gpio-uclass的通用框架，相关接口由uclass框架提供。框架里管理GPIO的核心结构体是

struct gpio_desc，这个结构体必须依赖device而存在。所以如果想要操作某个gpio，必须要有对应的device设备存在。

框架代码：

```
./include/asm-generic/gpio.h
./drivers/gpio/gpio-uclass.c
```

驱动代码：

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

5.3.2 相关接口

1. **gpio申请（初始化struct gpio_desc）**

```
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)
```

上述的申请接口：目的都是为了从传入的device里获取对应的gpio（即初始化struct gpio_desc结构体）。

2. **gpio input/out**

```
int dm_gpio_set_dir_flags(struct gpio_desc *desc, ulong flags);
```

其中flags：GPIOD_IS_OUT：输出模式； GPIOD_IS_IN：输入模式；

3. **gpio set/get**

```
int dm_gpio_get_value(const struct gpio_desc *desc)
int dm_gpio_set_value(const struct gpio_desc *desc, int value)
```

4. 代码范例

```
struct gpio_desc *gpio;
int value;

gpio_request_by_name(dev, "gpios", 0, gpio, GPIOD_IS_OUT);  // 申请gpio
dm_gpio_set_value(gpio, enable);                            // 设置gpio输出电平
dm_gpio_set_dir_flags(gpio, GPIOD_IS_IN);                   // 设置gpio为输入
value = dm_gpio_get_value(gpio);                            // 读取gpio电平
```

5.4 Pinctrl

[TODO]

5.5. I2C驱动

5.5.1 框架支持

I2C走的是i2c-uclass的通用框架，相关接口由uclass框架提供。i2c的相关接口都必须依赖device，因此类同内核的处理一样，需要在dts里把 子设备挂接到i2c bus节点之下。i2c框架在初始化的时候会把这些device作为i2c slave纳入自己的管理中。

框架代码：

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

驱动代码：

```
./drivers/i2c/rk_i2c.c
```

5.5.2 相关接口

```
int dm_i2c_reg_read(struct udevice *dev, uint offset)
int dm_i2c_reg_write(struct udevice *dev, uint offset, unsigned int val);
```

5.6 显示驱动

[TODO]

5.7 PMIC/Regulator驱动

5.7.1 框架支持

PMIC/regulator驱动走的是标准pmic-uclass、regulator-uclass的通用框架。目前支持的PMIC：RK805/RK808/RK816/RK818。

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
```

5.7.2 相关接口：

1. 获取**regulator**

```
int regulator_get_by_platname(const char *platname, struct udevice **devp);
```

platname：dts中regulator节点里“regulator-name”指定的名字，例如：vdd_arm、vdd_logic等；

devp：指向vdd_arm、vdd_logic对应的regulator device；

2. 开**/**关**regulator**

```
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);
```

3. 设置**regulator**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);
```

5.8 充电驱动

5.8.1 框架支持

充电功能方面，U-Boot里默认没有给与足够支持，因此我们自己增加了一套处理的框架代码，包括电量计部分和充电动画部分。目前支持的电量计：RK818/RK816。

电量计框架代码：

```
./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
......
```

充电框架代码：

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

充电动画驱动：

```

./drivers/power/charge_animation.c
```

charge_animation.c是真正具体实现充电流程的驱动，驱动里面会调用电量计上报的电量、适配器状态、检测按键、进入低功耗休眠等。逻辑流程：

```
charge-display-uclass.c
    -> charge_animation.c
        -> fuel_gauge_uclass.c
            ->fg_rk818.c/fg_rk817.c
```

5.8.2 充电图片打包

充电图片需要打包进resource.img才能被充电驱动读取并且显示。编译内核时默认不会打包充电图片，所以需要另外单独把这些图片打包进resource.img。

打包命令：

```
./pack_resource <input resource.img>
```

这个命令默认会把./tools/images/目录里的图片作为充电图片打包进resource.img，新的resource.img会生成在U-Boot根目录下，烧写的时候请烧写这个新的resource.img。

如下是打包时的提示信息：

```
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_5.bmp
battery_fail.bmp logo.bmp logo_kernel.bmp battery_0.bmp

resource.img is packed ready
```

5.8.3 DTS使能充电

默认代码已经使能了该驱动，通过在dts里增加并且使能charge-animation节点即可使能充电动画的功能。

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

    rockchip,uboot-charge-on = <0>;                 // 是否在U-Boot进行充电
    rockchip,android-charge-on = <1>;               // 是否在Android进行充电

    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>;                  // 灭屏时进入trust进行低功耗待机
```

```
    rockchip,auto-off-screen-interval = <20>;          // 亮屏超时后自动灭屏，单位秒。(如果没有这个属性，则默认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)
    rockchip,auto-wakeup-screen-invert = <1>;          // 休眠自动唤醒的时候，是否让屏幕产生亮/灭效果
};
```

- 自动休眠唤醒功能的作用：1. 考虑到有些电量计（比如adc）需要定时更新软件算法，否则会造成电量统计不准，因此不能让cpu一直处于休眠状态；2. 方便进行休眠唤醒的压力测试；

5.8.4 低功耗休眠

进入充电流程后可通过短按power实现系统亮灭屏，灭屏时进入低功耗待机状态，再次按下按键可唤醒。非低电状态下，长按power可退出充电流程进行开机。

5.8.5 更换充电图片

1. 更换./tools/images/目录下的图片，图片采用8bit或24bit bmp格式。使用命令“ls |sort”确认图片排列顺序是低电量到高电量，所有图片按照这个顺序打包进resource；
2. 修改./drivers/power/charge_animation.c里的图片和电量关系信息：

```
/*
 * 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_5.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：最后一张图片一定要是failed的图片，且“soc=-1”不可改变。

3. 执行pack_resource.sh打包命令获取新的resource.img；

5.9 存储驱动

U-Boot的存储驱动走的是标准的存储通用框架，所有接口都对接到block层支持文件系统。目前支持的存储设备有：emmc、nandflash。

5.9.1 相关接口

获取**blk**描述符：

```
struct blk_desc *rockchip_get_bootdev(void)
```

读写接口：

```
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)
```

代码范例：

```
struct rockchip_image *img;

dev_desc = rockchip_get_bootdev();              // 获取blk描述符

img = memalign(ARCH_DMA_MINALIGN, RK_BLK_SIZE);
if (!img) {
    printf("out of memory\n");
    return -ENOMEM;
}
...
ret = blk_dread(dev_desc, 0x2000, 1, img);      //  读操作
if (ret != 1) {
    ret = -EIO;
    goto err;
}
...
ret = blk_write(dev_desc, 0x2000, 1, img);      // 写操作
if (ret != 1) {
    ret = -EIO;
    goto err;
}
```

5.10 串口支持

U-Boot主要通过串口来打印启动过程中的log信息。

在U-Boot中串口驱动有两种（目前Rockchip平台的串口对应的驱动为`drivers/serial/ns16550.c`）。

U-Boot正常启动的时候，在relocation之前，会在board_f.c--->board_init_f函数中通过serial_init加载serial驱动。这是U-Boot中正式的debug console驱动，如果该驱动加载失败，U-Boot将停止启动。该驱动依赖dts中的chosen节点的stdout-path配置：

假如某块板子使用UART2作为debug console，波特率为1500000，则DTS需做如下配置：

“”

```
chosen {
    stdout-path = "serial2:1500000n8";
};
```

需要注意的是，serial驱动在加载的时候，需要依赖clk驱动，如果这时候clk驱动还没有正常加载，需要在对应uart的dts节点中加入clock-frequency属性：

```
&uart2 {
        clock-frequency = <24000000>;
};
```

这种debug console驱动在U-Boot启动的过程中加载的相对比较晚，如果在这之前就出现了异常，那依赖debug console就看不到具体的异常信息，针对这种情况，U-Boot提供了另外一种能更早进行debug打印的机制，Early Debug UART，使能Early Debug UART的方法如下：

在defconfig文件中打开DEBUG_UART, 指定该UART寄存器的基地址，时钟：

```
CONFIG_DEBUG_UART=y

CONFIG_DEBUG_UART_BASE=0x10210000

CONFIG_DEBUG_UART_CLOCK=24000000

CONFIG_DEBUG_UART_SHIFT=2

CONFIG_DEBUG_UART_BOARD_INIT=y

```

在board文件中实现`board_debug_uart_init`, 该函数一般负责设置iomux：

```
void board_debug_uart_init(void)
{
        static struct rk3308_grf * const grf = (void *)GRF_BASE;

        /* Enable early UART2 channel m1 on the rk3308 */
        rk_clrsetreg(&grf->gpio4d_iomux,
                     GPIO4D3_MASK | GPIO4D2_MASK,
                     GPIO4D2_UART2_RX_M1 << GPIO4D2_SHIFT |
                     GPIO4D3_UART2_TX_M1 << GPIO4D3_SHIFT);
}
```

在尽可能早的地方调用`debug_uart_init`:

```
#define EARLY_UART
#if defined(EARLY_UART) && defined(CONFIG_DEBUG_UART)
        /*
         * Debug UART can be used from here if required:
         *
         * debug_uart_init();
         * printch('a');
         * printhex8(0x1234);
         * printascii("string");
         */
        debug_uart_init();
        printascii("U-Boot SPL board init");
#endif
```

在U-Boot/arch目录下搜索debug_uart_init可以看到很多使用范例。

5.11 按键支持

5.11.1 框架支持

按键功能方面，U-Boot框架默认没有给与足够的支持，因此我们自己实现了一套按键框架机制来支持按键管理。

按键框架代码：

```
drivers/input/key-uclass.c
include/key.h
```

按键驱动：

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

- 上面4个驱动包含了Rockchip平台上所有已在使用的key节点；
- 考虑到U-Boot有充电休眠的功能，为了支持按键唤醒cpu，因此所有gpio类型的按键，目前全部都以中断的形式进行触发（不是轮询）。

5.11.2 相关接口

接口：

```
int platform_key_read(int code)
```

code头文件：

```
/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）或长按（按下时间不够长），都属于none事件；
    KEY_PRESS_DOWN,         // 一次完整的短按（按下->释放）才算是一个press down事件；
    KEY_PRESS_LONG_DOWN,    // 一次完整的长按（可以不释放）才算是一个press long down事件；
    KEY_NOT_EXIST,          // 找不到code对应的按键
};
```

KEY_PRESS_LONG_DOWN 事件的默认时长为2000ms，长按事件目前只在U-Boot充电时长按开机的时候会使用到。

```
#define KEY_LONG_DOWN_MS    2000
```

范例：

```
platform_key_read(KEY_VOLUMEUP);
platform_key_read(KEY_VOLUMEDOWN);
platform_key_read(KEY_POWER);
platform_key_read(KEY_HOME);
platform_key_read(KEY_MENU);
platform_key_read(KEY_ESC);
...
```

6. USB download

6.1 rockusb

命令行手动启用rockusb, 进入Windows烧写工具对应的Loader模式, eMMC:

```
rockusb 0 mmc 0
```

RKNAND:

```
rockusb 0 rknand 0
```

6.2 Fastboot

Fastboot 默认使用Google adb的VID/PID, 命令行手动启动fastboot:

```
fastboot usb 0
```

7. 固件加载

固件加载涉及parameter/gpt分区表、boot、recovery、kernel、resource分区以及dtb文件。

7.1 分区表

U-Boot支持两种分区表：parameter格式和GPT格式。启动的时候优先使用GPT分区表，如果不存在就尝试使用parameter分区表。

7.2 dtb文件

dtb文件是新版本kernel的dts配置文件的二进制化文件。目前dtb文件可以存放于AOSP的boot/recovery分区中，也可以存放于RK格式的resource分区。

7.3 boot/recovery分区

boot.img和recovery.img的固件分为两种打包格式：AOSP格式（android标准格式）和RK格式。

7.3.1 AOSP格式（android标准格式）

镜像文件的魔数为"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 dtb + android parameter；

recovery.img = kernel + ramdisk(for recovery) + dtb；

分区表 = parameter和GPT都支持（2选1）；

7.3.2 RK格式

RK格式的镜像单独打包kernel、dtb（从boot、recovery中剥离），镜像文件的魔数为"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 + kernel logo + uboot logo；

boot.img = ramdisk；

recovery.img = kernel + ramdisk(for recovery) + dtb；

分区表 = parameter和GPT都支持（2选1）；

7.3.3 优先级

U-Boot启动的时候默认优先使用"boot_android"命令加载AOSP格式的固件。如果加载失败则继续使用"bootrkp"命令加载RK格式的固件。

7.4 Kernel分区

Kernel分区包含kernel信息，即打包过的zImage或者Image。

7.5 resource分区

Resource镜像格式是为了能够同时存储多个资源文件（dtb、图片等）而设计的镜像格式，其魔数为"RSCE"：

```
00000000   52 53 43 45  00 00 00 00  01 01 01 00  01 00 00 00  RSCE............
00000010   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
```

目前这个分区主要用来打包dtb、开机logo、充电图片等。

7.6 U-Boot负责加载的固件

U-Boot负责加载ramdisk、dtb、kernel到内存中，具体的加载地址可以通过串口信息知道。

7.7 进入烧写模式

开机阶段，在插着USB的情况下长按 "音量+/recovery" 即可进入loader烧写模式;

8. SPL和TPL

SPL和TPL的介绍可以参考下面两份文档. doc/README.TPL doc/README.SPL

在Rockchip的方案中, TPL和SPL都是由Bootrom加载和引导的,具体引导流程, 相关固件的生成方法和存放位置可参考如下链接内容: http://opensource.rock-chips.com/wiki_Boot_option

TPL功能是DDR初始化, 代码运行在IRAM中,完成后返回Bootrom； SPL在没有TPL的情况下需要初始化DDR, 然后加载Trust(可选)和U-Boot, 并引导进入下一级.

SPL+TPL的组合实现rockchip ddr.bin+miniloader完全一致的功能, 可相互替换.

9. U-Boot和kernel DTB支持

9.1 设计出发点:

按照U-Boot的最新架构, 每个驱动代码本身需要依赖dts, 因此每一块板子都有一份对应的dts.

为了降低U-Boot在不同项目的维护量, 实现一颗芯片在同一类系统中能共用一份U-Boot, 而不是每一块板子都需要独立的dts编译成不同的U-Boot固件, 在U-Boot中增加支持使用kernel dtb, 复用其中的display, pmic/regulator, pinctrl等硬件相关信息,

因为u-boot本身有一份dts, 再加上kernel的dts, 原有的fdt用法会有冲突. 同时由于kernel的dts还需要提供给kernel使用, 所以不能把u-boot dts中部分dts节点overlay到kernel dts上传给kernel, 综合u-boot后续发展方向是使用live dt, 决定启动Live dt.

9.2 关于live dt:

live dt功能是在v2017.07版本合并的, 提交记录如下:

https://lists.denx.de/pipermail/u-boot/2017-January/278610.html

live dt的原理,是在初始化阶段直接扫描整个dtb, 把所有设备节点转换成struct device_node节点链表, 后续的bind和驱动访问dts都通过这个device_node或ofnode(device_node的封装)进行, 而不再访问原有dtb.

更多详细信息请参考: doc/driver-model/livetree.txt

9.3 fdt代码转换为支持live dt的代码

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时使用device_node来访问dt结点, 使用fdt时使用offset访问dt节点. 需要同时支持两种类型的驱动,请使用ofnode类型.

```
47  * @np: Pointer to device node, used for live tree
48  * @of_offset: Pointer into flat device tree, used for flat tree. Note that this
49  *      is not a really a pointer to a node: it is an offset value. See above.
50  */
51 typedef union ofnode_union {
52         const struct device_node *np;   /* will be used for future live tree */
53         long of_offset;
54 } ofnode;
```

"dev", *"ofnode*"开头的函数为支持两种dt访问方式, 根据程序当前使用dt类型来调用对应接口；

"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；

驱动程序做转换的时候可以参考标题包含"live dt"的提交.

9.4 支持kernel dtb的实现:

kernel的dtb支持, 是加在board_init的开头, 此时u-boot的dts已经扫描完成, 可以通过增加代码实现mmc/nand的读操作来读取kernel dtb, kernel的dtb读进来后, 进行live dt建表, 并bind所有设备, 最后更新gd->fdt_blob指针指向kernel dtb.

请注意该功能启用后, 大部分设备修改U-Boot的dts是无效的, 需要修改kernel的dts.

通过查找.config是否包含CONFIG_USING_KERNEL_DTB确认是否已启用kernel dtb.

该功能依赖live dt, 读dtb依赖rk格式固件或rk android固件, 所以Android以外的平台未启用.

9.5 关于U-Boot dts

U-Boot的根目录有个dts/文件夹, 编译完成后会生产dt.dtb和dt-spl.dtb两个DTB, dt.dtb是config的CONFIG_DEFAULT_DEVICE_TREE指定的dts编译得到的dtb拷贝过来的, 而dt-spl.dtb是把dt.dtb中带"u-boot,dm-pre-reloc"节点的设备的设备过滤出来, 并且去掉CONFIG_OF_SPL_REMOVE_PROPS选项中所有的property, 这样可以得到一个用于SPL的最简dtb.

dt-spl.dtb一般仅包含dmc, uart, mmc, nand, grf, cru等节点, 即串口, DDR和存储设备控制器及其依赖的CRU/GRF.

u-boot.bin默认打包的是dt.dtb, 在CONFIG_USING_KERNEL_DTB使能后, 默认打包的是dt-spl.dtb, 因为其他设备驱动将使用kernel中的dts.

U-Boot中所有芯片级dtsi请和kernel保持完全一致, 板级dts视情况简化得到一个evb的即可, 因为kernel的dts全套下来可能有几十个, 没必要全部引进到u-boot.

U-Boot 特有的节点, 如uart, emmc的alias等,请全部加到独立的rkxx-u-boot.dtsi里面, 不要破坏原有dtsi.

10. U-Boot相关工具

10.1 trust_merger工具

trust_merger用于64bit SoC打包bl30、bl31 bin、bl32 bin等文件，生成烧写工具需要的TrustImage格式固件。

10.1.1 trust的打包和解包

打包命令：

```
./tools/trust_merger [--pack] <config.ini>
```

打包需要传递描述打包参数的ini配置文件路径。

解包命令：

```
./tools/trust_merger --unpack <trust.img>
```

10.1.2 工具参数

以3368的配置文件为例：

```
[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BL31 bin
[BL33_OPTION]
SEC=0                       ----不存在BL31 bin
[OUTPUT]
PATH=trust.img [OUTPUT]     ----输出固件名字
```

10.2 boot_merger工具

boot_merger用于打包loader、ddr bin、usb plug bin等文件，生成烧写工具需要的loader格式的固件。

10.2.1 Loader的打包和解包

打包命令：

```
./tools/boot_merger [--pack] <config.ini>
```

打包需要传递描述打包参数的ini配置文件路径。

解包命令：

```

./tools/boot_merger --unpack <loader.bin>
```

10.2.2 工具参数

以3288的配置文件为例：

```
[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=15                    ----次版本号
[CODE471_OPTION]            ----code471，目前设置为ddr bin
NUM=1
Path1=tools/rk_tools/32_LPDDR2_300MHz_LPDDR3_300MHz_DDR3_300MHz_20140404.bin
[CODE472_OPTION]            ----code472，目前设置为usbplug bin
NUM=1
Path1=tools/rk_tools/rk32xxusbplug.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UBOOT bin
FlashData=tools/rk_tools/32_LPDDR2_300MHz_LPDDR3_300MHz_DDR3_300MHz_20140404.bin
FlashBoot=u-boot.bin
[OUTPUT]                    ----输出路径，目前文件名会自动添加版本号
PATH=RK3288Loader_UBOOT.bin
```

10.3 resource_tool工具

resource_tool用于打包任意资源文件，最终生成resource.img镜像。

打包命令：

```
./tools/resource_tool [--pack] [--image=<resource.img>] <file list>
```

解包命令：

```
./tools/resource_tool --unpack --image=<resource.img>
```

10.4 loaderimage

loaderimage工具用于打包rockchip miniloader所需固件, 含uboot.img和32bit的trust.img 用法:

```
loaderimage [--pack|--unpack] [--uboot|--trustos] file_in file_out [load_addr]
loaderimage --pack --trustos ${RKBIN}/${TOS} ./trust.img
loaderimage --pack --uboot u-boot.bin uboot.img 0x60000000
```

需要注意不同平台的'load_addr'不一样.

10.5 patman

详细信息参考tools/patman/README 这是一个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.

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 aliaes
  -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.

10.6 buildman工具

详细信息请参考tools/buildman/README

这个工具最主要的用处在于批量编译代码, 非常适合用于验证当前平台的提交是否影响到其他平台.

使用buildman需要提前设置好toolchain路径, 编辑'~/.buildman'文件:

```
[toolchain]
arm: ~/prebuilts/gcc/linux-x86/arm/gcc-linaro-6.3.1-2017.05-x86_64_arm-linux-gnueabihf/
aarch64: ~/prebuilts/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.

10.7 mkimage工具

详细信息参考doc/mkimage.1 这个工具可用于生成所有U-Boot/SPL支持的固件, 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
```

附录

[TODO]