U-Boot next-dev开发指南

发布版本：1.21

作者邮箱： Joseph Chen chenjh@rock-chips.com Kever Yang kever.yang@rock-chips.com Jon Lin jon.lin@rock-chips.com Chen Liang cl@rock-chips.com

日期：2019.01

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	V1.00	陈健洪	初始版本
2018-06-22	V1.01	朱志展	fastboot说明，OPTEE Client说明
2018-07-23	V1.10	陈健洪	完善文档，更新和调整大部分章节
2018-07-26	V1.11	林鼎强	完善Nand、SFC SPI Flash存储驱动部分
2018-08-08	V1.12	陈亮	增加HW-ID使用说明
2018-09-20	V1.13	张晴	增加CLK使用说明
2018-11-06	V1.20	陈健洪	增加/更新defconfig/rktest/probe/interrupt/kernel dtb/uart/atags
2019-01-21	V1.21	陈健洪	增加dtbo/amp/dvfs宽温/fdt命令说明

---

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 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

2.1 DM(Driver Model)

这是目前U-Boot的一套driver-device的标准开发模型，它和kernel的driver-device模式是非常类似的。U-Boot使用这套DM模型对各类设备进行规范化管理：驱动框架对应uclass，设备驱动对应driver，设备对应device。Rockchip提供的这套U-Boot也都遵循现有的标准驱动框架进行开发。

如下是README文档中的片段：

```
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.
```

建议读者先阅读U-Boot自带的相关文档，对DM模型有一定了解后方便对本文档后续的理解和U-Boot开发。

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

2.2 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
......
```

2.3 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
......
```

统一后的**board**文件：

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

有了这个统一的board.c文件后，目前大部分平台都可以走通用的板级初始化流程，我们在这个流程里使能了kenrel dtb方便兼容板级差异。

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。

2.4 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
......
```

如果要新增一个defconfig文件，命名方面并没有特殊的格式要求，建议遵循现有大多数defconfig的命名方式：[board]-[chip]_defconfig。

2.5 dtb的使用

2.5.1 启用kernel dtb

U-Boot的启动从时间先后来划分，可以分为两级启动阶段。

1. 第一级（relocate之前）：使用的是U-Boot自己的dtb。

一般第一阶段**只需要加载emmc、nand、cru、grf、uart等模块**，为了加快设备树的解析过程，dts里一般只去使能会用到的节点（板级差异的信息，如：电源、显示、clk等都会从第二阶段dtb中获取）。

需要特别注意：

- 第一阶段为了速度和效率，会对dtb做特殊处理，删除一些属性，例如：pinctrl-0 pinctrl-names clock-names interrupt-parent等，可以通过defconfig里的CONFIG_OF_SPL_REMOVE_PROPS指定。
- 第一阶段要使能的节点除了指明 "status=okay" 之外，还必须增加"u-boot,dm-pre-reloc;"属性，否则解析设备树时该节点会被忽略。这部分一般都在平台相关的[chip]-u-boot.dtsi里定义，例如：

```
./arch/arm/dts/px30-u-boot.dtsi
./arch/arm/dts/rk3399-u-boot.dtsi
./arch/arm/dts/rk3128-u-boot.dtsi
......
```

./arch/arm/dts/px30-u-boot.dtsi如下：

```
......
&nandc0 {
    u-boot,dm-pre-reloc;
};

&emmc {
    u-boot,dm-pre-reloc;
};

&cru {
    u-boot,dm-pre-reloc;
};
......
```

2. 第二级启动（relocate之后）：使用的是kernel的dtb。

一旦进入第二级阶段后，启动流程里会迅速切到kernel的dtb(取决于CONFIG_USING_KERNEL_DTB是否使能），后续更多的驱动初始化都是使用kernel的dtb信息。

一般而言，用户可能会涉及第二阶段的修改，第一阶比较少需要改动。关于kernel dtb的更详细内容，可以参考本文的 9. U-Boot和kernel DTB支持 。

2.5.2 关闭kernel dtb

如果出于某些特殊原因想要关闭kernel dtb的功能，即让U-Boot始终都使用U-Boot自身的dtb，则有如下操作点和注意事项。以rk3399为例，使用的是rk3399-evb.dts和rk3399_defconfig：

- rk3399_defconfig：关闭CONFIG_USING_KERNEL_DTB；
- rk3399-evb.dts：保留#include "rk3399-u-boot.dtsi"、chosen节点内容、各节点中的“u-boot,dm-pre-reloc;”属性（这些部分都是U-Boot特殊自用的内容，需要保留）；
- 在第二点保留项的基础上，再追加kernel dts的内容（**注意：是追加，不是覆盖！**）。

2.6 宏配置

目前的宏配置选项一般出现在如下几个地方（以rk3399为例，其余平台类同）

```
./include/configs/rockchip-common.h
./include/configs/evb_rk3399.h
./include/configs/rk3399_common.h
configs/rk3399_defconfig
arch/arm/mach-rockchip/Kconfig
```

如下对用户可能改动的重要宏配置做说明：

./include/configs/rockchip-common.h

```
......
#define RKIMG_DET_BOOTDEV \                             // 动态探测当前设备的存储类型
    "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 \                             // 启动kernel的命令
    "boot_android ${devtype} ${devnum};" \              // 启动AOSP标准格式的固件
    "bootrkp;" \                                        // 启动rockchip格式的固件
    "run distro_bootcmd;"                               // 启动linux固件
......
```

./include/configs/evb_rk3399.h：

```
......
#ifndef CONFIG_SPL_BUILD
#undef CONFIG_BOOTCOMMAND
#define CONFIG_BOOTCOMMAND RKIMG_BOOTCOMMAND       // 设置U-Boot的自启动命令为
RKIMG_BOOTCOMMAND
#endif
......
#define ROCKCHIP_DEVICE_SETTINGS \                 // 使能显示模块
        "stdout=serial,vidconsole\0" \
        "stderr=serial,vidconsole\0"
......
```

./include/configs/rk3399_common.h：

```
......
#ifndef CONFIG_SPL_BUILD
#define ENV_MEM_LAYOUT_SETTINGS \          // 固件的加载地址
    "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 \        // 把上述所有相关的环境变量在此汇合
    ENV_MEM_LAYOUT_SETTINGS \
    "partitions=" PARTS_DEFAULT \          // 默认的GPT分区表内容
    ROCKCHIP_DEVICE_SETTINGS \
    RKIMG_DET_BOOTDEV \
    BOOTENV                                // 启动linux的设备探测顺序
#endif

#define CONFIG_PREBOOT                     // 在CONFIG_BOOTCOMMAND之前被执行的预处理命令
......
```

2.7 debug手段

目前U-Boot里debug的手段相比kernel是比较有限的，例如：不支持dump_stack()等。这里介绍几个比较常用、重要的debug手段，方便用户在开发过程中对遇到的问题进行调试。

2.7.1 流程类

2.7.1.1 debug函数

debug()函数默认定义为空函数，通过增加DEBUG宏定义就可以让debug()函数生效。打开这个调试信息之后用户可以很方便追踪整个U-Boot的启动过程。

一般各个平台有对应的common文件： ./include/configs/rkxxx_common.h文件，可以在里面增加定义：

```
#define DEBUG
```

2.7.1.2 Early Debug UART

参考本文档5.10.2 Early Debug UART配置。

2.7.1.3 initcall

U-Boot分成board_f.c和board_r.c两个阶段启动，分别对应init_sequence_f[]和init_sequence_r[]两个系统函数列表，如果想定位是在哪个系统函数里被调用、出现问题、死机等，可以把initcall_run_list()函数里的debug改为printf打印出调用顺序。上述涉及相关文件：

```
./common/board_f.c
./common/board_r.c
./lib/initcall.c
```

修改initcall_run_list()后的启动打印如下：

```
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
initcall: 00000000002140dc
....
initcall: 00000000002143a8
initcall: 00000000002143cc
3.8 GiB
initcall: 00000000002143b8
initcall: 00000000002141f8
initcall: 00000000002143c0
initcall: 000000000021423c
Relocation Offset is: f5c03000
initcall: 00000000f5e176bc
initcall: 00000000f5e174a8
initcall: 00000000002146a4 (relocated to 00000000f5e176a4)
initcall: 0000000000214668 (relocated to 00000000f5e17668)
initcall: 00000000002146c4 (relocated to 00000000f5e176c4)
```

```
initcall: 0000000000202900 (relocated to 00000000f5e05900)
....
```

有了如上信息之后，此时我们只需要进行反汇编或者打开符号表即可知道每个initcall的地址对应哪个函数，具体请参考本文的3.2.6 debug辅助命令。

2.7.2 读写类

2.7.2.1 进入U-Boot命令行

U-Boot的命令行模式提供了很多命令供用户调试问题使用。命令行下输入"?"即可列出所有支持的命令：

```
=> ?
?       - alias for 'help'
base    - print or set address offset
bdinfo  - print Board Info structure
boot    - boot default, i.e., run 'bootcmd'
boot_android- Execute the Android Bootloader flow.
bootd   - boot default, i.e., run 'bootcmd'
bootefi - Boots an EFI payload from memory
bootelf - Boot from an ELF image in memory
......
```

通常在默认情况下，U-Boot启动时不会自动进入串口的命令行模式，用户有2种方式进入（任选其一）：

1. 在对应的defconfig配置CONFIG_BOOTDELAY=<seconds>，就可以让U-Boot进入命令行倒计时模式；
2. U-Boot开机阶段，长按ctrl + c 组合键直到强制进入命令行模式；

2.7.2.2 md/mw：内存/寄存器读写

U-Boot提供的"md"、"mw"命令可以实现内存或寄存器的读写。如下：

```
// 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]
```

其中：

```
 .b 表示的数据长度是： 1 byte;
 .w 表示的数据长度是： 2 byte;
 .l 表示的数据长度是： 4 byte; (推荐)
 .q 表示的数据长度是： 8 byte;
```

使用范例：

1. 读操作：显示0x76000000地址开始的连续0x10个数据单元，每个数据单元的长度是4byte。

```
=> md.l 0x76000000 0x10
76000000: fffffffe ffffffff ffffffff ffffffff    ................
76000010: ffffffdf ffffffff feffffff ffffffff    ................
76000020: ffffffff ffffffff ffffffff ffffffff    ................
76000030: ffffffff ffffffff ffffffff ffffffff    ................
```

2. 写操作：对0x76000000地址的数据单元赋值为0xffff0000；

```
=> mw.l 0x76000000 0xffff0000

=> md.l 0x76000000 0x10 // 回读
76000000: ffff0000 ffffffff ffffffff ffffffff    ................
76000010: ffffffdf ffffffff feffffff ffffffff    ................
76000020: ffffffff ffffffff ffffffff ffffffff    ................
76000030: ffffffff ffffffff ffffffff ffffffff    ................
```

3. 写操作（连续）：对0x76000000地址开始的连续0x10个数据单元都赋值为0xffff0000，每个数据单元的长度是4byte。

```
=> mw.l 0x76000000 0xffff0000   0x10

=> md.l 0x76000000 0x10     // 回读
76000000: ffff0000 ffff0000 ffff0000 ffff0000    ................
76000010: ffff0000 ffff0000 ffff0000 ffff0000    ................
76000020: ffff0000 ffff0000 ffff0000 ffff0000    ................
76000030: ffff0000 ffff0000 ffff0000 ffff0000    ................
```

2.7.2.3 iomem：读寄存器

命令行方式：

注意，这里介绍的方式只支持读取寄存器，不支持写操作。相比md命令需要手动指定寄存器地址，我们提供了一个iomem命令直接解析dts的device节点，获取基地址信息，用起来应该是更加省时方便的。iomem命令如下：

```
=> iomem
iomem - Show iomem data by device compatible

Usage:
iomem iomem <compatible> <start offset>  <end offset>
  eg: iomem -grf 0x0 0x200
```

这里的<compatible>内容支持子字符串进行匹配。以grf为例，不同平台的grf节点的compatible字段名字不同，例如：“rockchip,px30-grf”、"rockchip,rk3368-grf"等，为了通用性强，这个接口可以支持关键字匹配。但是仅匹配最先查找到的device节点。

使用范例：

```
=> iomem -grf 0x0 0x50
rockchip,rk3228-grf:
11000000:  00000000 00000000 00004000 00002000
11000010:  00000000 00005028 0000a5a5 0000aaaa
11000020:  00009955 00000000 00000000 00000000
11000030:  00000000 00000000 00000000 00000000
11000040:  00000000 00000000 00000000 00000000
11000050:  0000090f
```

函数接口方式：

上述是以命令的形式提供了读寄存器的接口。目前也提供了函数接口方便用户调试：

```
./arch/arm/mach-rockchip/iomem.c
./include/iomem.h
```

接口：

```
void iomem_show(const char *label, unsigned long base, size_t start, size_t end);
void iomem_show_by_compatible(const char *compat, size_t start, size_t end);
```

2.7.2.4 i2c读写

U-Boot提供的"i2c"命令可以实现i2c设备的寄存器读写。如下：

```
=> i2c
i2c - I2C sub-system

Usage:
i2c bus [muxtype:muxaddr:muxchannel] - show I2C bus info
crc32 chip address[.0, .1, .2] count - compute CRC32 checksum
i2c dev [dev] - show or set current I2C bus
i2c edid chip - print EDID configuration information
i2c loop chip address[.0, .1, .2] [# of objects] - looping read of device
i2c md chip address[.0, .1, .2] [# of objects] - read from I2C device
i2c mm chip address[.0, .1, .2] - write to I2C device (auto-incrementing)
i2c mw chip address[.0, .1, .2] value [count] - write to I2C device (fill)
i2c nm chip address[.0, .1, .2] - write to I2C device (constant address)
i2c probe [address] - test for and show device(s) on the I2C bus
i2c read chip address[.0, .1, .2] length memaddress - read to memory
i2c write memaddress chip address[.0, .1, .2] length [-s] - write memory
          to I2C; the -s option selects bulk write in a single transaction
i2c flags chip [flags] - set or get chip flags
i2c olen chip [offset_length] - set or get chip offset length
i2c reset - re-init the I2C Controller
i2c speed [speed] - show or set I2C bus speed
```

使用范例：

1. 读操作：

```
=> 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    ................
```

2. 写操作：

```
=> 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（对比上述"1.读操作"的内容）
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    ................
```

2.7.2.5 fdt读写

U-Boot提供的fdt命令可以实现对当前dtb的读写操作：

```
=> 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可以把fdt完整dump出来，比较常用：

```
=> fdt addr $fdt_addr_r   // 指定fdt地址
=> fdt print              // 把fdt内容全部打印出来
```

2.7.3 状态类

2.7.3.1 printf 时间戳

目前U-Boot也可以支持让printf打印的信息带有时间戳，这样便于开发者快速确认各个阶段的启动流程耗时（注意：本身串口打印也是需要耗时的）。启动该项功能，只需要打开宏：

```
CONFIG_BOOTSTAGE_PRINTF_TIMESTAMP
```

注意：这里的时间戳并不是从0开始，仅仅是把当前系统的timer时间读出来而已，所以仅适合计算时间差。

RK3399开机信息范例：

```
```

```
[    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
......
```

因为U-Boot阶段是单核，串口打印过多本身就会影响启动速度，加入时间戳之后更加会消耗时间。因此一般情况下，建议关闭该功能，仅在调试阶段打开。

2.7.3.2 dm框架统计信息

U-Boot提供的"dm"命令可以查看dm框架的统计信息。通过这些信息我们可以知道U-Boot里所有设备的管理情况（拓扑图），能让我们从更高的视角去审视当前的系统状态。这个功能一般对于搭建、调试、维护整个U-Boot基础平台的用户会比较有帮助。

dm信息主要是把"status=okay"的device-driver进行展示，从这个信息中我们可以知道：

- 某个device是否已经被dm框架解析且和对应的driver进行绑定；
- 某个驱动是否已经被probe过；
- 某个uclass到底挂载了多少个device；
- 各个device之间的parent-child关系；

```
=> dm
dm - Driver model low level access

Usage:
dm tree         Dump driver model tree ('*' = activated)
dm uclass        Dump list of instances for each uclass
dm devres        Dump list of device resources for each device
```

使用范例1：

```
=> dm tree

 Class      Probed        Driver               Name
-----------------------------------------------------------
 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
 syscon     [   ]   rk322x_syscon             |-- syscon@31090000
 clk        [ + ]   fixed_rate_clock          |-- oscillator
 syscon     [ + ]   rk322x_syscon             |-- syscon@11000000
 phy        [   ]   rockchip_usb2phy          |   |-- usb2-phy@760
 phy        [   ]   rockchip_usb2phy_port     |   |   |-- otg-port
 phy        [   ]   rockchip_usb2phy_port     |   |   `-- host-port
 phy        [   ]   rockchip_usb2phy          |   `-- usb2-phy@800
 phy        [   ]   rockchip_usb2phy_port     |       |-- otg-port
 phy        [   ]   rockchip_usb2phy_port     |       `-- host-port
 serial     [ + ]   ns16550_serial            |-- serial@11020000
 i2c        [ + ]   i2c_rockchip              |-- i2c@11050000
 ......
```

使用范例2：

```
=> dm uclass

uclass 0: root
- * root_driver @ 7be54c88, seq 0, (req -1)

uclass 10: simple_bus
uclass 11: adc
- * saradc@ff100000 @ 7be56220, seq 0, (req -1)

uclass 13: blk
-   dwmmc@ff0c0000.blk @ 7be54ea0
- * dwmmc@ff0f0000.blk @ 7be550e8, seq 0, (req -1)
-   dwmmc@ff0d0000.blk @ 7be55da0

uclass 14: clk
- * oscillator @ 7be55b50, seq 0, (req -1)
- * clock-controller@ff760000 @ 7be7d058, seq 1, (req -1)
- * external-gmac-clock @ 7be80c58, seq 2, (req -1)
- * xin32k @ 7be814c8, seq 3, (req -1)

uclass 17: display
- * dp@ff970000 @ 7be7d2c8, seq 0, (req -1)

uclass 21: firmware
-   psci @ 7be810a8

uclass 22: i2c
- * i2c@ff650000 @ 7be562c8, seq 0, (req 0)
-   i2c@ff140000 @ 7be7c838, seq -1, (req 1)
-   i2c@ff150000 @ 7be7c890, seq -1, (req 3)
-   i2c@ff160000 @ 7be7c8e8, seq -1, (req 4)
-   i2c@ff660000 @ 7be7c9b0, seq -1, (req 2)

uclass 24: i2c_generic
uclass 34: mmc
- * dwmmc@ff0c0000 @ 7be54d10, seq 1, (req 1)
```

```
- * dwmmc@ff0f0000 @ 7be54f78, seq 0, (req 0)
-   dwmmc@ff0d0000 @ 7be55c30

uclass 39: panel
- * edp-panel @ 7be80bd0, seq 0, (req -1)

uclass 40: backlight
- * backlight @ 7be81178, seq 0, (req -1)

uclass 77: key
-   rockchip-key @ 7be811f0
......
```

2.7.3.3 panic cpu信息

当U-Boot发生异常产生panic的时候，系统会打印出panic时刻的CPU状态信息。通过这些信息我们可以知道当前CPU状态和异常原因。如下：

```
* Relocate offset = 000000003db55000
* ELR(PC)    =   000000000025bd78
* LR         =   000000000025def4
* SP         =   0000000039d4a6b0

* ESR_EL2    =   0000000040732550
        EC[31:26] == 001100, Exception from an MCRR or MRRC access
        IL[25] == 0, 16-bit instruction trapped

* DAIF       =   00000000000003c0
        D[9] == 1, DBG masked
        A[8] == 1, ABORT masked
        I[7] == 1, IRQ masked
        F[6] == 1, FIQ masked

* SPSR_EL2   =   0000000080000349
        D[9] == 1, DBG masked
        A[8] == 1, ABORT masked
        I[7] == 0, IRQ not masked
        F[6] == 1, FIQ masked
        M[4] == 0, Exception taken from AArch64
        M[3:0] == 1001, EL2h

* SCTLR_EL2  =   0000000030c51835
        I[12] == 1, Icaches enabled
        C[2] == 1, Dcache enabled
        M[0] == 1, MMU enabled

* VBAR_EL2   =   000000003dd55800
* HCR_EL2    =   000000000800003a
* TTBR0_EL2  =   000000003fff0000

x0 : 00000000ff300000 x1 : 0000000054808028
x2 : 000000000000002f x3 : 00000000ff160000
x4 : 0000000039d7fe80 x5 : 000000003de24ab0
```

```
36 ......
37 x28: 0000000039d81ef0 x29: 0000000039d4a910
```

其中EC[31:26]说明了当前这次panic的原因，此外还提供了各种寄存器状态信息。其中比较关注的有：pc、lr、sp等。我们结合反汇编就可以快速定位错误的点，关于反汇编的方式请参考本文的3.2.6 debug辅助命令。

2.7.3.4 panic 寄存器信息

当U-Boot发生panic的时候，我们还可以让寄存器信息一起dump出来：目前默认提供cru，pmucru, grf，pmugrf。要使能这个功能，需要打开宏：

```
1 CONFIG_ROCKCHIP_CRASH_DUMP
```

打印信息是追加在cpu的panic信息之中，如下：

```
1  ......
2  * VBAR_EL2    =   000000003dd55800
3  * HCR_EL2     =   000000000800003a
4  * TTBR0_EL2   =   000000003fff0000
5
6  x0 : 00000000ff300000 x1 : 0000000054808028
7  x2 : 000000000000002f x3 : 00000000ff160000
8  ......
9
10 // 平台相关的寄存器dump:
11
12 rockchip,px30-cru:
13 ff2b0000:  0000304b 00001441 00000001 00000007
14 ff2b0010:  00007f00 00000000 00000000 00000000
15 ff2b0020:  00003053 00001441 00000001 00000007
16 ......
17
18 rockchip,px30-grf:
19 ff140000:  00002222 00002222 00002222 00001111
20 ff140010:  00000000 00000000 00002200 00000033
21 ff140020:  00000000 00000000 00000000 00000202
22 ......
```

如果想增加更多的打印，则需要修改代码。位置如下：

```
vim ./arch/arm/lib/interrupts_64.c

void show_regs(struct pt_regs *regs)
{
......
#ifdef CONFIG_ROCKCHIP_CRASH_DUMP
    iomem_show_by_compatible("-cru", 0, 0x400);
    iomem_show_by_compatible("-pmucru", 0, 0x400);
    iomem_show_by_compatible("-grf", 0, 0x400);
    iomem_show_by_compatible("-pmugrf", 0, 0x400);
    /* tobe add here ... */
#endif
}
```

2.7.3.5 hang信息（relocate之后）

有时候我们会碰到U-Boot启动时突然hang住不动，串口也毫无响应，并且没有任何有效打印输出的情况。以往在这种情况下，我们只能增加大量的log来追踪启动流程或者直接连JTAG进行定位。

现在如果遇到这种情况，用户可以打开CONFIG_ROCKCHIP_DEBUGGER。如果U-Boot启动后5s内还没有进入kernel，则串口每隔5s就会自动dump当前的CPU现场状态。这部分内容同上面提到的PANIC信息是一样的格式：

```
>>> Rockchip Debugger:
* Relocate offset = 000000003db55000
* ELR(PC)    =   0000000000025bd78
* LR         =   0000000000025def4
* SP         =   0000000039d4a6b0

* ESR_EL2    =   0000000040732550
        <NULL>      // 因为只是hang住，CPU本身可能状态正常，所以EC[31:26]没有显示异常原因。
        IL[25] == 0, 16-bit instruction trapped

* DAIF       =   00000000000003c0
        D[9] == 1, DBG masked
        A[8] == 1, ABORT masked
        I[7] == 1, IRQ masked
        F[6] == 1, FIQ masked

* SPSR_EL2   =   0000000080000349
        D[9] == 1, DBG masked
        A[8] == 1, ABORT masked
        I[7] == 0, IRQ not masked
```

一般情况下，建议默认把这个功能关闭，仅当出问题时再打开即可。

2.7.3.6 固件crc校验

固件在打包的时候在img头里有打包工具计算的固件CRC值，如果遇到问题怀疑是U-Boot加载到内存的固件有完整性问题，则可以打开CRC校验功能进行确认：

```
CONFIG_ROCKCHIP_CRC
```

打开后的U-Boot提示信息：

```
=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)
## 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 ...
```

打开CRC校验后U-Boot的启动时间会变长，所以一般仅在调试问题时才打开，默认配置不要打开。

2.7.3.7 开机log

目前各个平台的固件启动流程如下：

```
pre-loader => trust => U-Boot => kernel
```

情况1：

有时候我们会遇到跑完trust后没有任何U-Boot打印输出就卡死的情况，比较大的可能是打包固件或者烧写固件有问题。

此时可以注意trust打印信息中的"INFO: Entry point address = 0x200000"和"INF [0x0] TEE-CORE:init_primary_helper:379: Next entry point address: 0x60000000" 指明了U-Boot的运行地址，这个地址来自于固件的打包头信息，参考本文档3.2.4 固件生成。

一般情况下，U-Boot的启动地址：64位平台上是从SDRAM偏移2M地址处，32位平台上是从SDRAM偏移0地址处。

64位平台trust：

```
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trust：

```
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
```

情况2：

通过U-Boot开机第一行打印信息回溯固件对应的代码仓库的提交点。如下可以看出这份固件对应的代码commit-id是b34f08b（前面的'g'忽略），可以达到精确回溯。

```
U-Boot 2017.09-01730-gb34f08b (Jul 06 2018 - 17:47:52 +0800)
```

相比上面的情况，如下的信息中出现了"dirty"，说明当时编译固件的时候本地还存在临时改动，而且没有通过git commit提交进仓库。这个固件编译点是不干净的，虽然同样可以确认是b34f08b提交点，但是因为当时还有本地临时代码，所以无法达到精确回溯。

```
U-Boot 2017.09-01730-gb34f08b-dirty (Jul 06 2018 - 17:35:04 +0800)
```

2.7.3.8 分区表信息

有时候可能会遇到开机时pre-loader（一级loader）加载固件报异常的情况，比较大的可能性是固件的地址烧写存在问题。例如：

```
SdmmcInit=0 1
StorageInit ok = 30370
tag:LOADER error,addr:0x2000
hdr 032c77e4 + 0x0:0x20534f54,0x20202020,0x00000000,0x00000000,
tag:LOADER error,addr:0x4000
hdr 032c77e4 +
0x0:0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

tag:LOADER error,addr:0x2800
hdr 032c77e4 + 0x0:0x20534f54,0x20202020,0x00000000,0x00000000,
tag:LOADER error,addr:0x4800
......
```

此时我们可能会想知道当前机器的分区表信息，包括各个分区的大小、地址等，我们可以通过命令行提供的"part"命令查看，具体请参考7.1 分区表 。

2.7.4 烧写类

2.7.4.1 maskrom/loader烧写模式

在U-Boot开发调试阶段如果出现在U-Boot阶段就启动失败，进入命令行的情况。这时候的情况是：

1. 可能无法识别recovery按键进入loader烧写模式，这时可以通过命令行的方式进入loader烧写模式；
2. 可能无法识别recovery按键，也无法使用loader烧写模式。这时可以通过命令行的方式进入maskrom模式，（否则要硬件上短接相关引脚才行，比较麻烦）。

上述两种情况，如果通过命令进入烧写模式，请参考本文的3.2.8 烧写和工具。

2.8 atags传参机制

运行在kernel之前的固件有：一级loader、trust（bl31和trust os）、U-Boot。这些前级固件之间有时候需要共享一些信息，因此需要一个统一的传参机制。由于atags实现起来比较精简，因此目前使用atags进行传参（注意：只传递到U-Boot为止，不会传递给kernel）。目前传递的信息包括：串口的配置、启动设备的类型、bl31和trust os的内存布局、ddr的容量信息等，具体参考代码：

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

2.9 驱动的probe

这章节的内容非常重要，所以在此优先提出。具体请参考本文档5. 驱动支持 的前言。

3. 平台编译

3.1 前期准备

3.1.1 rkbin 仓库

rkbin仓库主要存放了Rockchip不开源的bin文件（trust、loader等）、脚本、打包工具等，它只是一个“工具包”仓库。**bin文件会一直在不断更新，用户最好能及时同步相关内容，避免因为版本过旧引起问题。**

rkbin仓库需要和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。

下载方式见附录rkbin仓库下载 。

3.1.2 gcc工具链

默认使用的编译器是gcc-linaro-6.3.1版本，下载方式见附录gcc编译器下载 。

```
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 =
```

3.1.4 各平台defconfig

目前主要在使用的各个芯片平台的defconfig对应情况如下（包含但不限于，基于commit:58d85a1）。大部分平台都开启了kernel dtb的支持，这意味着这个平台在board_r[]阶段使用的是kernel dtb，因此能兼容大多数的板级差异（如：外设、电源、clk、显示等）。对于不支持kernel dtb的defconfig，则无法兼容板级差异，但是有更优的启动速度和uboot.bin的size。

通常情况下，如果没有对速度和固件大小有特别严苛的要求，建议采用开启了kernel dtb的defconfig。关于kernel dtb，可以参考本文的 9. U-Boot和kernel DTB支持 。

芯片	**defconfig**	**kernel dtb** 支持
rv1108	evb-rv1108_defconfig	N
rk1808	rk1808_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-npu	rk3399pro-npu_defconfig	Y
rk3308(aarch32)	rk3308-aarch32_defconfig	Y
rk3308(aarch32)	evb-aarch32-rk3308_defconfig	N
rk3308(aarch64)	evb-rk3308_defconfig	Y
px30	evb-px30_defconfig	Y
rk3326	evb-rk3326_defconfig	Y

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：

```
# debug使用
ADDR2LINE_ARM32=arm-linux-gnueabihf-addr2line
ADDR2LINE_ARM64=aarch64-linux-gnu-addr2line

# debug使用
OBJ_ARM32=arm-linux-gnueabihf-objdump
OBJ_ARM64=aarch64-linux-gnu-objdump

# 编译使用
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 固件编译

帮助信息：

```
./make.sh --help
```

编译命令：

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

1. 首次编译

无论32位或64位平台，如果是第一次或者想重新指定defconfig进行编译，则必须指定[board]，这样才会生成新的.config。如下：

```
./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
```

编译完成后的提示：

```
......
Platform RK3399 is build OK, with new .config(make evb-rk3399_defconfig)
```

2. 二次编译

无论32位或64位平台，如果想使用已有的.config进行二次编译，则不需要指定[board]字段。如下：

```
./make.sh
```

编译完成后的提示：

```
......
Platform RK3399 is build OK, with exist .config
```

3.2.4 固件生成

1. 编译完成后，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!               // U-Boot的运行地址会被追加在打包头信息里
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!          // trust的运行地址会被追加在打包头信息里
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
```

注意：当执行make clean/mrproper/distclean的时候，Makefile会默认把编译阶段生成的中间文件都删除，其中包括bin文件。因为loader固件的格式是.bin，所以也会被同时删除。**用户需要注意：不要把重要的、不想被删除的.bin文件放在U-Boot的根目录下。**

3.2.5 pack辅助命令

命令格式：

```
./make.sh   [loader|loader-all|uboot|trust]
```

如果用户不想每次生成固件的时候都编译整个U-Boot工程，则可以通过辅助命令对某个固件进行单独打包（用.config里获取芯片信息）。如下：

```
./make.sh trust        --- 只打包trust.img
./make.sh loader       --- 只打包loader bin
./make.sh loader-all   --- 打包所有支持的loader bin
./make.sh uboot        --- 只打包uboot.img
```

loader-all：打包当前平台所有的loader

有些平台上会支持多种存储启动引导，因此会提供特殊的loader进行支持（比如支持spi nor flash...）。默认编译U-Boot时只会生成一个默认的loader（适用于大部分产品形态），不会打包生成这些特殊loader。如果需要的话，请使用"loader-all"命令：

例如rk3399平台执行完"loader-all"后生成：

```
./rk3399_loader_v1.12.112.bin          // 支持emmc、nand的默认loader，可满足大部分产品形态需求
./rk3399_loader_spinor_v1.12.114.bin   // 支持spi nor flash的loader
```

3.2.6 debug辅助命令

命令格式：

```
./make.sh       [elf|map|sym|addr]
```

为了开发时候调试方便，我们支持一些辅助命令快速打开一些调试文件（用.config里获取芯片信息）。如下：

```
./make.sh elf       --- 反汇编，默认使用-D参数
./make.sh elf-S     --- 反汇编，使用-S参数
./make.sh elf-d     --- 反汇编，使用-d参数
./make.sh map       --- 打开u-boot.map
./make.sh sym       --- 打开u-boot.sym
./make.sh <addr>    --- 需要addr对应的函数名和代码位置
```

addr命令：

通过add命令可以打印出地址对应的函数名和具体的代码位置：

```
guest@ubuntu:~/u-boot$ ./make.sh 000000000024fb1c

000000000024fb1c l     F .text  000000000000004c spi_child_pre_probe
/home/guest/u-boot/drivers/spi/spi-uclass.c:153
```

如果是无效地址，则不会有解析结果：

```
guest@ubuntu:~/u-boot$ ./make.sh 000000000024fb1c

??:0
```

elf命令：

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，可以执行如下命令获取帮信息：

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

3.2.7 编译报错处理

关于**make clean/mrproper/distclean**：

```
1. make clean:
    Delete most generated files Leave enough to build external modules
2. make mrproper:
    Delete the current configuration, and all generated files
3. make distclean:
    Remove editor backup files, patch leftover files and the like Directories & files
removed with 'make clean
```

清除强度：distclean > mrproper > clean。

报错1：

```
  UPD     include/config/uboot.release
  Using .. as source for U-Boot
  .. is not clean, please run 'make mrproper'
  in the '..' directory.
  CHK     include/generated/version_autogenerated.h
  UPD     include/generated/version_autogenerated.h
make[1]: *** [prepare3] Error 1
make[1]: *** Waiting for unfinished jobs....
  HOSTLD  scripts/dtc/dtc
make[1]: Leaving directory `/home/guest/uboot-nextdev/u-boot/rockdev'
make: *** [sub-make] Error 2
```

如上的编译报错信息，一般是因为改变了编译输出的目录，导致新旧目录之间的中间文件让Makefile对编译依赖产生了不清晰的判断。只需要按照提示信息，执行make mrproper 即可。

报错2：

```
make[2]: *** [silentoldconfig] Error 1
make[1]: *** [silentoldconfig] Error 2
make: *** No rule to make target `include/config/auto.conf', needed by `include/config/kernel.release'.  Stop.
```

如上的编译报错信息，一般也是因为编译的工程环境不干净导致的。通过make mrproper或distclean可以解决。

3.2.8 烧写和工具

3.2.8.1 工具

Windows烧写工具版本必须是**V2.5版本或以上**(推荐使用最新的版本)；

3.2.8.2 loader烧写模式

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

3.2.8.3 命令行进入烧写模式

在U-Boot命令行下：

1. 输入"rbrom"可以进入maskrom烧写模式；
2. 输入“rockusb 0 mmc 0”可以进入loader烧写模式（也可能是"rknand"，取决于当前存储类型）；

3.2.9 分区表

1. 目前U-Boot支持RK 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RK parameter分区表的具体格式请参考文档：《Rockchip-Parameter-File-Format-Version1.4.md》和本文的7.1 分区表。

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

前言

U-Boot使用DM框架去管理所有的设备和驱动，它和kernel的device-driver模型非常类似。但是有一点需要注意的是，kernel在初始化时会使用initcall的机制自动把所有有效的driver进行probe，但是U-Boot里并没有这样的机制进行probe。U-Boot里想要probe某个驱动的话，必须由用户主动调用相应的框架接口进行发起，相关的接口如下：

```
./include/dm/uclass.h

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,
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);
......
```

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）。目前而言，会使用到中断情况主要有：

- U-Boot充电休眠时cpu进入低功耗休眠模式，需要中断按键进行唤醒；
- CONFIG_ROCKCHIP_DEBUGGER对应的驱动会使用到中断；

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驱动

5.2.1 框架支持

CLK使用的是clk-uclass的通用框架，相关接口由uclass框架提供。

1. probe中会rkclk_init（）可以设置部分PLL、CPU、总线等频率，可用于SPL阶段提高频率加速开机。

```
./drivers/clk/rockchip/clk_rk3399.c

static void rkclk_init(struct rk3399_cru *cru)
```

```
{
    rk3399_configure_cpu(cru, APLL_600_MHZ, CPU_CLUSTER_LITTLE);

    /* configure perihp aclk, hclk, pclk */
    aclk_div = DIV_ROUND_UP(GPLL_HZ, PERIHP_ACLK_HZ) - 1;

    hclk_div = PERIHP_ACLK_HZ / PERIHP_HCLK_HZ - 1;
    assert((hclk_div + 1) * PERIHP_HCLK_HZ ==
           PERIHP_ACLK_HZ && (hclk_div <= 0x3));

    pclk_div = PERIHP_ACLK_HZ / PERIHP_PCLK_HZ - 1;
    assert((pclk_div + 1) * PERIHP_PCLK_HZ ==
           PERIHP_ACLK_HZ && (pclk_div <= 0x7));

    rk_clrsetreg(&cru->clksel_con[14],
             PCLK_PERIHP_DIV_CON_MASK | HCLK_PERIHP_DIV_CON_MASK |
             ACLK_PERIHP_PLL_SEL_MASK | ACLK_PERIHP_DIV_CON_MASK,
             pclk_div << PCLK_PERIHP_DIV_CON_SHIFT |
             hclk_div << HCLK_PERIHP_DIV_CON_SHIFT |
             ACLK_PERIHP_PLL_SEL_GPLL << ACLK_PERIHP_PLL_SEL_SHIFT |
             aclk_div << ACLK_PERIHP_DIV_CON_SHIFT);

    rkclk_set_pll(&cru->gpll_con[0], &gpll_init_cfg);
}
```

2. probe中增加clk_set_defaults解析cru节点设置assigned-clock的频率。

```
./drivers/clk/rockchip/clk_px30.c

ret = clk_set_defaults(dev);
    if (ret)
        debug("%s clk_set_defaults failed %d\n", __func__, ret);
```

```
./arch/arm64/boot/dts/rockchip/px30.dtsi

        ......
        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 =
            <1200000000>, <100000000>,
            <26000000>, <600000000>,
            <200000000>, <200000000>,
            <150000000>, <150000000>,
            <100000000>, <200000000>;
        ......
```

3. CPU提频开机加速

CPU频率设置可以使用上述中的2设置，但是需要注意电压是否足够，如果不够，在设置频率之前要设置电压，电压可以通过在对应的regulator节点下追加regulator-init-microvolt=<...>指定初始化电压。

```
./arch/arm64/boot/dts/rockchip/px30-evb-ddr4-v10.dts

    ......
    vdd_arm: DCDC_REG2 {
        ......
        regulator-init-microvolt = <1100000>;
        ......
    };
```

框架代码：

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

驱动代码：

```
./drivers/clk/rockchip/clk_rkxxx.c
./drivers/clk/rockchip/clk_pll.c
```

驱动代码位于drivers/clk/rockchip目录, 每颗芯片有一份独立的驱动。clk_pll.c是公用代码。

5.2.2 相关接口

1. **clk** 接口

使用clk-ops结构注册clk的接口，设备最常使用的接口：

```
ulong (*get_rate)(struct clk *clk);
ulong (*set_rate)(struct clk *clk, ulong rate);
int (*get_phase)(struct clk *clk);
int (*set_phase)(struct clk *clk, int degrees);
int (*set_parent)(struct clk *clk, struct clk *parent);
```

2. 代码范例

```
ret = clk_get_by_name(crtc_state->dev, "dclk_vop", &dclk);
或者
/* clocks = <&cru ACLK_VOPB>, <&cru DCLK_VOPB>, <&cru HCLK_VOPB>; */
ret = clk_get_by_index(rtc_state->dev, 1, &dclk)

if (!ret)
    ret = clk_set_rate(&dclk, mode->clock * 1000);
if (IS_ERR_VALUE(ret)) {
    printf("%s: Failed to set dclk: ret=%d\n", __func__, ret);
    return ret;
}
```

3. **clk init**

有三种方式实现部分时钟的init:

- 驱动probe时会调用rkclk_init()函数对PLL、CPU和通用BUS进行初始化， 详细上文有描述；
- 使用clk_set_defaults（dev），解析内核cru节点中的assigned-clocks 设置初始频率， 详细上文有描述；目前除了cru节点会解析assigned-clocks 设置初始频率，又在VOP和GMAC中增加此功能用于频率设置及PARENT设置。后续其他设备驱动里如果需要此功能请自行增加。
- 其他设备的时钟设置，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。

4. **clk dump**

在clks_dump结构中增加想打印出的时钟的ID，然后使用soc_clk_dump()函数打印。目前默认会打印PLL、CPU、总线频率，如果需要其他时钟频率自行增加。

备注：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。

5.3 GPIO驱动

5.3.1 框架支持

GPIO使用的是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，则必须要有对应的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)
```

关于返回值：

dm_gpio_get_value()的返回值和dts里指定的电平属性（GPIO_ACTIVE_LOW/HIGH）有关系，并不表示当前的引脚电平值，而是表示是否触发了，其中1表示是触发了，0表示没有触发。例如：gpios = <&gpio 0 GPIO_ACTIVE_LOW>，如果此时引脚电平为低，则函数返回1；如果电平引脚为高，函数返回值为0。

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

5.4.1 框架支持

pinctrl走的是pinctrl-class的通用框架，相关接口由uclass框架提供。使用方法同kernel类似，通过dts里的pinctrl节点指定。

框架代码：

```
./drivers/pinctrl/pinctrl-uclass.c
./include/dm/pinctrl.h
```

驱动代码：

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

5.4.2 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
```

一般情况下，用户很少会需要调用上述接口进行引脚功能的切换，通常都是在device进行probe时设置"default"状态即可满足使用，而这部分已经由系统框架在驱动probe时自动完成，用户不用关心。

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

5.6.1 框架支持

Rockchip 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等显示接口的支持。U-Boot显示的图片为保存在kernel根目录下的logo.bmp，和logo_kernel.bmp一起打包进resource.img中，图片要求:

1. 8bit或者24bit BMP格式的图片；
2. logo.bmp和logo_kernel.bmp的图片分辨率大小一致；
3. 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。

   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
   ```

```
12
13 lvds:
14       drivers/video/drm/rockchip_lvds.c
15       drivers/video/drm/rockchip_lvds.h
16
17 mipi:
18       drivers/video/drm/rockchip_mipi_dsi.c
19       drivers/video/drm/rockchip_mipi_dsi.h
20       drivers/video/drm/rockchip-inno-mipi-dphy.c
21
22 edp:
23       drivers/video/drm/rockchip_analogix_dp.c
24       drivers/video/drm/rockchip_analogix_dp.h
25       drivers/video/drm/rockchip_analogix_dp_reg.c
26       drivers/video/drm/rockchip_analogix_dp_reg.h
27
28 hdmi:
29       drivers/video/drm/dw_hdmi.c
30       drivers/video/drm/dw_hdmi.h
31       drivers/video/drm/rockchip_dw_hdmi.c
32       drivers/video/drm/rockchip_dw_hdmi.h
33
34 panel:
35       drivers/video/drm/rockchip_panel.c
36       drivers/video/drm/rockchip_panel.h
```

5.6.2 相关接口

1. 显示U-Boot logo和kernel logo：

```
1 void rockchip_show_logo(void);
```

2. 显示指定的bmp图片，目前主要用于充电logo的显示：

```
1 void rockchip_show_bmp(const char *bmp);
```

3. 将U-Boot中确定的一些变量通过dtb文件传递给内核，包括kernel logo的大小、地址和格式，crtc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。

```
1 rockchip_display_fixup(void *blob);
```

5.6.3 DTS配置

```
1 reserved-memory {
2     #address-cells = <2>;
3     #size-cells = <2>;
4     ranges;
5
6     drm_logo: drm-logo@00000000 {
7         compatible = "rockchip,drm-logo";
```

```
        reg = <0x0 0x0 0x0 0x0>; //预留buffer用于kernel logo的存放，具体地址和大小在U-Boot中会修改
    };
};

&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 {
            ...
        };
    };
};
```

5.6.4 defconfig配置

目前除了RK3308之外的其他平台U-Boot中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，驱动代码在：

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

如果要使用这套驱动可以打开配置CONFIG_VIDEO_ROCKCHIP同时关闭CONFIG_DRM_ROCKCHIP，和我们目前SDK使用的显示驱动对比，后者的优势有：

1. 支持的平台和显示接口更全面；

2. 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。

3. U-Boot logo可以平滑过渡到kernel logo直到系统起来；

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/RK809/RK816/RK817/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);
```

5.7.3 初始化电压

Buck1/2在使用时通常需要进行调压，因此regulator节点里设置的"regulator-min-microvolt"和"regulator-min-microvolt"一般不会相等，这样在初始化regulator的时候就只会用PMIC的默认上电电压，软件不会去设置电压。当需要制定初始化电压的时候，可以通过"regulator-init-microvolt"指定，一般在U-Boot阶段进行CPU提频时会用到。如下：

```
regulator-min-microvolt = <900000>
regulator-max-microvolt = <1500000>
regulator-init-microvolt = <1100000>// 初始化电压设置为1.1v
```

5.7.4 debug方法

方法**1**：**regulator**初始化阶段

系统各路regulator的初始化位置。如下：

```
./arch/arm/mach-rockchip/board.c
    --> board_init
        --> regulators_enable_boot_on(false);
```

把上述的"false"修改"true"即可打印出各路regulator的配置。如下：

```
           DCDC_REG1@     vdd_center:  750000uV <-> 1350000uV, set  900000uV, enabling | supsend     -61uV, disabled
           DCDC_REG2@      vdd_cpu_l:  750000uV <-> 1350000uV, set  900000uV, enabling | supsend     -61uV, disabled
           DCDC_REG3@        vcc_ddr:     -61uV <->     -61uV, set  712500uV, enabling | supsend     -61uV, enabling
           DCDC_REG4@        vcc_1v8: 1800000uV <-> 1800000uV, set 1800000uV, enabling | supsend 1800000uV, enabling
            LDO_REG1@     vcc1v8_dvp: 1800000uV <-> 1800000uV, set 1800000uV, enabling | supsend     -61uV, disabled
            LDO_REG2@      vcc3v0_tp: 3000000uV <-> 3000000uV, set 3000000uV, enabling | supsend     -61uV, disabled
            LDO_REG3@     vcc1v8_pmu: 1800000uV <-> 1800000uV, set 1800000uV, enabling | supsend 1800000uV, enabling
            LDO_REG4@       vccio_sd: 1800000uV <-> 3000000uV, set 3000000uV, enabling | supsend 3000000uV, enabling
            LDO_REG5@  vcca3v0_codec: 3000000uV <-> 3000000uV, set 3000000uV, enabling | supsend     -61uV, disabled
            LDO_REG6@        vcc_1v5: 1500000uV <-> 1500000uV, set 1500000uV, enabling | supsend 1500000uV, enabling
            LDO_REG7@  vcca1v8_codec: 1800000uV <-> 1800000uV, set 1800000uV, enabling | supsend     -61uV, disabled
            LDO_REG8@        vcc_3v0: 3000000uV <-> 3000000uV, set 3000000uV, enabling | supsend 3000000uV, enabling
         SWITCH_REG1@      vcc3v3_s3:     -61uV <->     -61uV, set     -38uV, enabling | supsend     -61uV, disabled
         SWITCH_REG2@      vcc3v3_s0:     -61uV <->     -61uV, set     -38uV, enabling | supsend     -61uV, disabled
          vcc3v3-sys@    vcc3v3_sys: 3300000uV <-> 3300000uV, set 3300000uV, enabling | supsend     -61uV, enabling (ret: -38)
vcc5v0-host-regulator@   vcc5v0_host:     -61uV <->     -61uV, set     -61uV, enabling | supsend     -61uV, enabling
          vcc5v0-sys@    vcc5v0_sys: 5000000uV <-> 5000000uV, set 5000000uV, enabling | supsend     -61uV, enabling (ret: -38)
              vcc-sd@        vcc_sd: 3300000uV <-> 3300000uV, set 3300000uV, disabled | supsend     -61uV, enabling
   vcc-phy-regulator@       vcc_phy:     -61uV <->     -61uV, set     -61uV, enabling | supsend     -61uV, enabling
            vdd-log@       vdd_log:  800000uV <-> 1400000uV, set      -1uV, enabling | supsend     -61uV, enabling
            vcc-lcd@       vcc_lcd: 3300000uV <-> 3300000uV, set 3300000uV, enabling | supsend     -61uV, enabling (ret: -38)
```

内容说明：

1. “-61”对应的是错误码，表示没有找到dts里对应的属性；

```
#define ENODATA     61  /* No data available */
```

2. "（ret: -38）"对应的错误码是，表示没有实现对应的回调接口；

```
#define ENOSYS      38  /* Invalid system call number */,
```

3. 如果对上述各参数的内部含义有疑问，可直接阅读对应的源代码。

```
static void regulator_show(struct udevice *dev, int ret)
```

方法2：regulator初始化完成后

U-Boot串口命令行下，使用"regulator"命令。驱动如下：

```
cmd/regulator.c
```

命令格式：

```
=> regulator
regulator - uclass operations

Usage:
regulator list                 - list UCLASS regulator devices
regulator dev [regulator-name] - show/[set] operating regulator device
regulator info                 - print constraints info
regulator status [-a]          - print operating status [for all]
regulator value [val] [-f]     - print/[set] voltage value [uV] (force)
regulator current [val]        - print/[set] current value [uA]
regulator mode [id]            - print/[set] operating mode id
regulator enable               - enable the regulator output
regulator disable              - disable the regulator output
```

法3：regulator初始化完成后（类同法2）

U-Boot串口命令行下使用"rktest regulator"命令，具体参考11. rktest测试程序。

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：RK809/RK816/RK817/RK818。

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   // rk809复用
./drivers/power/fuel_gauge/fg_rk816.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.sh <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_resource.sh  /home/guest/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_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，在使用pack_resource.sh脚本打包时，所有图片会按照这个顺序被打包进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即可；

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、Nand flash、SPI Nand flash、SPI Nor flash。

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;
```

```
}
```

5.9.2 DTS配置

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

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

注：nandc节点是与nand flash设备通信的控制器节点，sfc节点是与spi flash设备通信的控制器节点，如果只用nand flash设备或只用spi flash设备，可以只使能对应节点，而两个节点都使能也是兼容的。

5.9.3 defconfig配置

rknand

rknand通常是指drivers/rknand/目录下的存储驱动，其是针对大容量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》，适用以下存储：

- SLC、MLC、TLC Nand flash

```
CONFIG_RKNAND=y
```

rkflash

rkflash则是drivers/rkflash/目录下的存储驱动，其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设备通过Nandc host与Nand flash device通信，SPI flash通过sfc host与SPI flash devices通信，适用的存储设备主要包括：

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

具体适用颗粒选型参考《RK SpiNor and SLC Nand SupportList》。

```
CONFIG_RKFLASH=y
CONFIG_RKNANDC_NAND=y
CONFIG_RKSFC_NOR=y
CONFIG_RKSFC_NAND=y
```

注意：rknand/驱动与rkflash/驱动的ftl框架不兼容，所以两个框架无法同时配置使能Nand设备。

5.10 串口支持

5.10.1 串口配置

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
./drivers/serial/serial-uclass.c
./include/debug_uart.h
```

U-Boot正常启动的时候，在relocation之前，会在board_init_f[]函数列表中通过serial_init()加载驱动。这是U-Boot中正式的debug console驱动，如果该驱动加载失败则U-Boot将停止启动。具体的配置流程如下（以uart2为例）：

1. iomux配置：每个平台都有`board_debug_uart_init()`函数，一般位于rkxxx.c里（例如：rk3399.c/rk3368.c/px30.c等），需要在这个函数里完成uart iomux的配置。
2. clock配置：每个平台默认都是把uart的时钟源配置为24Mhz，一般pre-loader里会帮忙配置好，在U-Boot阶段可以不用配置。但是如果是修改串口号，且pre-loader没有进行对应频率初始化，则U-Boot阶段要确认当前所用串口的时钟是24Mhz。代码修改一般加在`board_debug_uart_init()`函数里。
3. uart节点配置：uart2节点里需要指定如下2个属性：

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

4. chosen节点配置：必须以stdout-path的形式指定串口（这是U-Boot比较特殊的地方）

   ```
   chosen {
       stdout-path = "serial2:1500000n8";  // 这里的波特率值实际是无效的
   };
   或者：  // 推荐采用下面这种方式
   chosen {
       stdout-path = &uart2;
   };
   ```

5. baudrate配置：通过宏`CONFIG_BAUDRATE`指定串口波特率，一般在对应的defconfig或者rkxxx_common.h里进行指定。

5.10.2 Early Debug UART配置

上述这种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，本质上是绕过console框架，直接往uart寄存器写数据。目前各个平台默认都有启用这个功能，配置方法如下：

1. 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    // 修改串口号时，只需要修改基地址即可
   CONFIG_DEBUG_UART_CLOCK=24000000
   CONFIG_DEBUG_UART_SHIFT=2
   CONFIG_DEBUG_UART_BOARD_INIT=y
   ```

2. 在board文件中实现`board_debug_uart_init()`，该函数一般负责设置iomux。请在尽可能早的地方调用它，目前默认一般都是放在board文件里调用，即rkxxx.c中。

```
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);
}
```

5.10.3 更改串口

如果仅仅是 U-Boot需要更改串口号，而前级的loader、trust等固件不做改变，那么请执行5.10.1 串口配置和5.10.2 Early Debug UART配置里的修改步骤。

5.10.4 Pre-loader serial

同样是更改串口号的方式，但采取的是沿用前级的loader的配置。即loader改完串口后，后面的trust和U-Boot都继续沿用，这样就不必每一级固件都做修改。这个功能需要依赖：

1. 一级loader和U-Boot都要支持atags传参，这样才能把前级loader的serial配置传递到U-Boot使用；
2. 一级loader要更改好串口并且进行atags传参；
3. U-Boot自身需要在rkxx-u-boot.dtsi里把需要的uart增加上属性“u-boot,dm-pre-reloc;”和在aliases里建立serial别名，例如./arch/arm/dts/rk1808-u-boot.dtsi里为了方便，把所有uart都配置上：

```
aliases {
    mmc0 = &emmc;
    mmc1 = &sdmmc;

// 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;
};
```

5.10.5 关闭串口打印

使能CONFIG_SILENT_CONSOLE即可关闭console打印（UART驱动还是会正常加载），仅仅保留一条提示信息。

```
......
INFO:    Entry point address = 0x200000
INFO:    SPSR = 0x3c9

U-Boot: enable slient console           // 只有一条U-Boot提示信息，没有其余打印信息

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
......
```

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 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：

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

5.12 Vendor Storage

Vendor Storage 是设计用来存放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》。

5.12.1 原理概述

我们一共把vendor的存储块分成4个分区，vendor0、vendor1、vendor2、vendor3。每个vendorX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移到下一个vendor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。

5.12.2 框架支持

Vendor Storage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机制。

驱动文件：

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

5.12.3 相关接口

读写接口：

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

5.12.4 自测程序

U-Boot串口命令行下，使用"rktest vendor"命令可以进行Vendor Storage功能的测试，具体参考11. rktest测试程序。这个测试命令可以测试当前Vendor Storage驱动的基本读写和逻辑等功能是否正常，如果内部的所有测试项都pass，则说明一切正常。

5.13 OPTEE Client支持

目前一些安全的操作需要在U-Boot这级操作或读取一些数据必须需要OPTEE帮忙获取。U-Boot里面实现了OPTEE Client代码，可以通过该接口与OPTEE通信。配置及说明如下：

CONFIG_OPTEE_CLIENT，U-Boot调用trust总开关。 CONFIG_OPTEE_V1，旧平台使用，如312x,322x,3288,3228H,3368,3399。 CONFIG_OPTEE_V2，新平台使用，如3326,3308。 CONFIG_OPTEE_ALWAYS_USE_SECURITY_PARTITION， 当emmc的rpmb不能用，才开这个宏，默认不开。

5.14 DVFS宽温

5.14.1 宽温策略

U-Boot框架没有支持DVFS，但是为了支持某些芯片的宽温功能，我们实现了一套DVFS宽温驱动去根据芯片温度调整cpu/dmc频率-电压。但有别于内核DVFS驱动，这套宽温驱动仅仅对触发了最高/低温度阈值的时刻进行控制。

具体的宽温策略：

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（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（即准备加载固件之前）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），那么再执行完本次宽温策略后如果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。

5.14.2 框架支持

框架代码：

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

宽温驱动：

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

5.14.3 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);
```

5.14.4 启用宽温

1. defconfig里使能配置：

   ```
   CONFIG_DM_DVFS=y
   CONFIG_ROCKCHIP_WTEMP_DVFS=y
   ```

   依赖于：

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

2. 对应平台的rkxxx_common.h指定CONFIG_PREBOOT：

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

3. 内核dts的宽温节点配置：

```
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。

5.14.5 宽温结果

当cpu温控启用的时候，正确解析完参数后会有如下打印，主要是关键信息的内容：

```
// <NULL>表明没有指定低温阈值
DVFS: cpu: low=<NULL>'c, high=95.5'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): ......
```

5.15 AMP(Asymmetric Multi-Processing)

目前的U-Boot架构只支持单核启动和运行，并不支持AMP运行。如果用户有AMP的需求（这里指的是：不同的core运行在不同的firmwarm上），那么U-Boot阶段可以通过psci_cpu_on()指定core从不同的firmware入口地址启动。

```
/*
 * psci_cpu_on() - Standard ARM PSCI cpu on call.
 *
 * @cpuid:          cpu id
 * @entry_point:    boot entry point
 *
 * @return 0 on success, otherwise failed.
 */
int psci_cpu_on(unsigned long cpuid, unsigned long entry_point);
```

特别注意：如果上述firmware是在U-Boot阶段加载到DRAM上，那么当加载完成后需要把cache数据全都刷到DRAM上，否则可能引起core启动失败等问题。

cache接口：

```
void flush_cache(unsigned long start, unsigned long size) // 需要指定刷cache的空间范围
void flush_dcache_all(void)  // 默认全部空间刷cache，推荐用这种方式。
```

5.16 DTBO/DTO(Devcie Tree Overlay)

为了便于用户对本章节内容的理解，这里先明确相关的专业术语，本章节更多相关知识可参考：https://source.android.google.cn/devices/architecture/dto。

名词	解释
DTB	名词。设备树 Blob
DTBO	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

它们之间的关系，可以描述为：

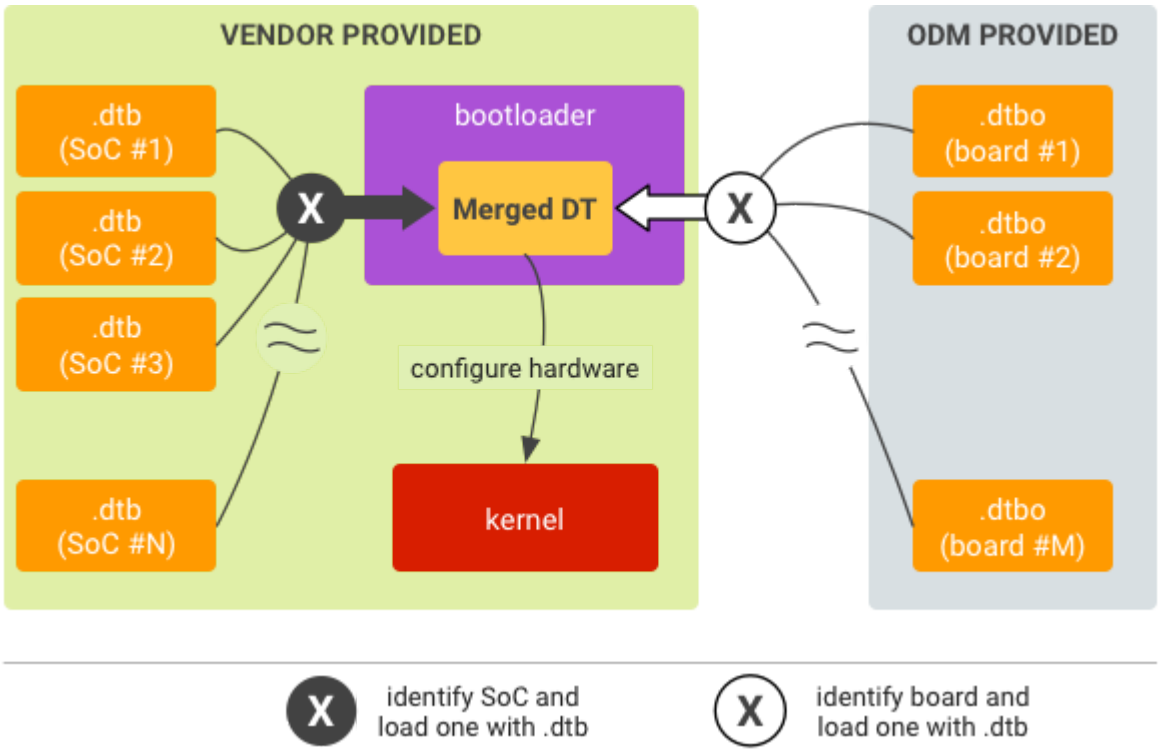
- DTS是用于描述FDT的文件；
- DTS经过DTC编译后可生成DTB/DTBO；
- DTB和DTBO通过DTO操作可合并成一个新的DTB；

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。

5.16.1 原理介绍

DTO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。如图：



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
```

5.16.2 DTO启用

1. defconfig里使能配置：

```
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.
 *
 * Example:
 *  struct dt_table_entry *entry;
 *  int index;
 *
 *  dt_for_each_entry(entry, dt_table_hdr, index) {
 *
 *      .... (use entry)
 *  }
 *
 *  return index;
 */
    return 0;
}
```

5.16.3 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. 当DTO执行成功后，会在传递给kernel的cmdline里追加如下内容，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执行成功后如果想进一步确认新生成的dtb内容，用户可通过"fdt"命令把新生成的dtb内容打印出来确认，具体参考2.7.2.5 fdt读取。

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
```

6.2.1 fastboot支持命令速览

```
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 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
```

6.2.2 fastboot具体使用

1. fastboot flash < partition > [< filename >]

功能：分区烧写。

例： fastboot flash boot boot.img

2. fastboot erase < partition >

功能：擦除分区。

举例：fastboot erase boot

3. 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                  /* 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:0x100000
(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
```

4. fastboot set_active < slot >

功能：设置重启的slot。

举例：fastboot set_active _a

5. fastboot reboot

功能：重启设备，正常启动

举例：fastboot reboot

6. fastboot reboot-bootloader

功能：重启设备，进入fastboot模式

举例：fastboot reboot-bootloader

7. fastboot flashing unlock

功能：解锁设备，允许烧写固件

举例：fastboot flashing unlock

8. fastboot flashing lock

功能：锁定设备，禁止烧写

举例：fastboot flashing lock

9. 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 atx_permanent_attributes.bin

10. fastboot get_staged [< filename >]

功能：从设备端获取数据

举例：fastboot get_staged raw_atx_unlock_challenge.bin

11. fastboot oem fuse at-perm-attr

功能：烧写ATX及hash

举例：fastboot stage atx_permanent_attributes.bin

fastboot oem fuse at-perm-attr

12. fastboot oem fuse at-perm-attr-data

功能：只烧写ATX到安全存储区域（RPMB）

举例：fastboot stage atx_permanent_attributes.bin

fastboot oem fuse at-perm-attr-data

13. fastboot oem at-get-ca-request
14. fastboot oem at-set-ca-response

15. fastboot oem at-lock-vboot

功能：锁定设备

举例：fastboot oem at-lock-vboot

16. 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_atx_unlock_challenge.bin

./make_unlock.sh（见make_unlock.sh参考）

fastboot stage atx_unlock_credential.bin fastboot oem at-unlock-vboot

可以参考《how-to-generate-keys-about-avb.md》

17. 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

18. 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

19. 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

20. fastboot oem at-reset-rollback-index

功能：复位设备的rollback数据

举例：fastboot oem at-reset-rollback-index

21. 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

7. 固件加载

固件加载涉及RK 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：RK parameter格式和GPT格式。启动的时候优先寻找GPT分区表，如果不存在就尝试使用RK parameter分区表。

7.1.1 分区表文件

如下是GPT分区的parameter.txt文件内容。可以通过"TYPE: GPT"属性可以确认当前是GPT分区表还是RK parameter分区表（没有这个属性）。

```
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
CMDLINE:mtdparts=rk29xxnand:0x00002000@0x00004000(uboot),0x00002000@0x00006000(trust),0x00002000@0x00008000(misc),0x00008000@0x0000a000(resource),0x00010000@0x00012000(kernel),0x00010000@0x00022000(boot),0x00020000@0x00032000(recovery),0x00038000@0x00052000(backup),0x00002000@0x0008a000(security),0x00100000@0x0008c000(cache),0x00500000@0x0018c000(system),0x00008000@0x0068c000(metadata),0x00100000@0x00694000(vendor),0x00100000@0x00796000(oem),0x00000400@0x00896000(frp),-@0x00896400(userdata:grow)
```

GPT和RK parameter分区表的具体格式请参考文档：《Rockchip-Parameter-File-Format-Version1.4.md》。

7.1.2 分区表查看

在U-Boot串口命令行下，可以通过如下命令进行分区表信息查看：

```
part list <interface> <dev>

<interface>: 存储设备类型，可以是：mmc、rknand、rksfc；
<dev>: 设备号，可以是：0、1、2....。
```

1. GPT分区表（Partition Type: EFI）：

```
=> part list mmc 0

Partition Map for MMC device 0  --   Partition Type: EFI

Part    Start LBA       End LBA         Name
        Attributes
        Type GUID
        Partition GUID
  1     0x00004000      0x00005fff      "uboot"
        attrs:  0x0000000000000000
        type:   3b600000-0000-423e-8000-128b000058ca
        guid:   727b0000-0000-4069-8000-68d500005dea
  2     0x00006000      0x00007fff      "trust"
        attrs:  0x0000000000000000
        type:   bf570000-0000-440f-8000-42dc000079ef
        guid:   ff3c0000-0000-4d3a-8000-5e9c00006be6
  3     0x00008000      0x00009fff      "misc"
        attrs:  0x0000000000000000
        type:   4f030000-0000-4744-8000-545300000e1e
```

```
          guid:   0c240000-0000-4f6a-8000-207e00006722
  4       0x0000a000      0x00011fff      "resource"
          attrs:  0x0000000000000000
          type:   d3460000-0000-4360-8000-37d9000037c0
          guid:   81500000-0000-4f59-8000-166100000c05
  5       0x00012000      0x00021fff      "kernel"
          attrs:  0x0000000000000000
          type:   33770000-0000-401d-8000-505400004c3e
          guid:   464f0000-0000-4317-8000-1f2f00004af7
  6       0x00022000      0x00031fff      "boot"
          attrs:  0x0000000000000000
          type:   575e0000-0000-4666-8000-74ae000055fe
          guid:   43270000-0000-456c-8000-0ace00004560
  7       0x00032000      0x00051fff      "recovery"
          attrs:  0x0000000000000000
          type:   273b0000-0000-4d5e-8000-6fcd0000106a
          guid:   614e0000-0000-4b53-8000-1d28000054a9
  8       0x00052000      0x00089fff      "backup"
          attrs:  0x0000000000000000
          type:   8c3f0000-0000-4d58-8000-009b00006ee9
          guid:   86300000-0000-4f7a-8000-102300000338
  9       0x0008a000      0x0008bfff      "security"
          attrs:  0x0000000000000000
          type:   6c100000-0000-4e5c-8000-5afe000015e2
          guid:   9b2f0000-0000-4843-8000-12a900001176
  10      0x0008c000      0x0018bfff      "cache"
          attrs:  0x0000000000000000
          type:   b1490000-0000-4927-8000-24e000005fbf
          guid:   891d0000-0000-4e45-8000-43a1000072cb
  11      0x0018c000      0x0068bfff      "system"
          attrs:  0x0000000000000000
          type:   41770000-0000-442b-8000-7928000058e7
          guid:   36430000-0000-484a-8000-37f200004ca0
  12      0x0068c000      0x00693fff      "metadata"
          attrs:  0x0000000000000000
          type:   061c0000-0000-480a-8000-67be000043c2
          guid:   8c5d0000-0000-4052-8000-798600007d5b
  13      0x00694000      0x00793fff      "vendor"
          attrs:  0x0000000000000000
          type:   e62f0000-0000-4e1e-8000-738a000015b8
          guid:   721a0000-0000-4d0e-8000-044400001366
  14      0x00796000      0x00895fff      "oem"
          attrs:  0x0000000000000000
          type:   cb190000-0000-4c74-8000-137300007831
          guid:   cf200000-0000-4765-8000-4b1400005227
  15      0x00896000      0x008963ff      "frp"
          attrs:  0x0000000000000000
          type:   9c380000-0000-4c4b-8000-326400004995
          guid:   8d060000-0000-4772-8000-32de00003108
  16      0x00896400      0x00e8ffde      "userdata"
          attrs:  0x0000000000000000
          type:   415f0000-0000-4419-8000-2f420000194c
          guid:   93580000-0000-4303-8000-128a00005c6f
```

2. RK parameter分区表（Partition Type: RKPARM）：

```
=> part list mmc 0

Partition Map for MMC device 0  --   Partition Type: RKPARM

Part    Start LBA       Size            Name
  1     0x00004000      0x00002000      uboot
  2     0x00006000      0x00002000      trust
  3     0x00008000      0x00002000      misc
  4     0x0000a000      0x00008000      resource
  5     0x00012000      0x00010000      kernel
  6     0x00022000      0x00010000      boot
  7     0x00032000      0x00020000      recovery
  8     0x00052000      0x00038000      backup
  9     0x0008a000      0x00002000      security
 10     0x0008c000      0x00100000      cache
 11     0x0018c000      0x00500000      system
 12     0x0068c000      0x00008000      metadata
 13     0x00694000      0x00100000      vendor
 14     0x00796000      0x00100000      oem
 15     0x00896000      0x00000400      frp
 16     0x00896400      0x005f9c00      userdata
```

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。

对于U-Boot阶段的dtb使用，可以参考本文的 9. U-Boot和kernel DTB支持 。

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 = RK 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 = RK 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（Android标准格式）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，如果加载失败则继续使用"run distro"命令加载Linux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 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。例如：

```
......
=Booting Rockchip format image=
kernel   @ 0x02080000 (0x0124e008)
ramdisk  @ 0x0a200000 (0x0017871c)
## Flattened Device Tree blob at 01f00000
   Booting using the fdt blob at 0x1f00000
   Loading Ramdisk to 08087000, end 081ff71c ... OK
   Loading Device Tree to 0000000008070000, end 00000000080860b7 ... OK
Adding bank: start=0x00200000, size=0x08200000
Adding bank: start=0x0a200000, size=0xede00000

Starting kernel ...
```

7.7 固件启动流程

```
pre-loader => trust => U-Boot => kernel
```

7.8 HW-ID适配硬件版本

7.8.1 HW-ID设计目的

硬件会经常更新版本，更换一些元器件，比如，屏幕，wifi模组等，如果每一个版本硬件，都要一套软件，就会比较麻烦，通过HW_ID，可以保证一套软件适配不同版本的硬件。

7.8.2 HW-ID设计原理

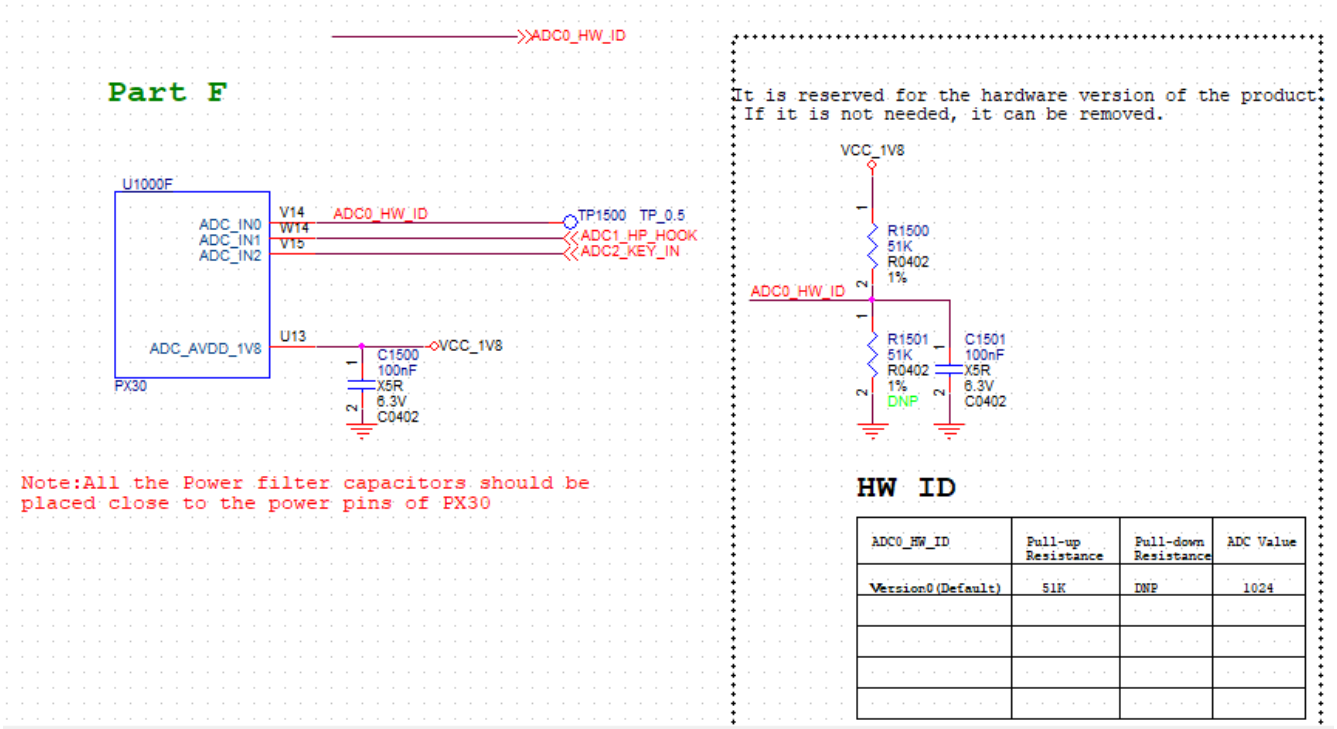
把多份dtb文件打包到同一个resource.img里面，U-Boot引导kernel的时候，从resource.img里面找到一份和当前硬件版本匹配的dtb，并传递给kernel，加载不同的软件配置。通过硬件配置ADC/GPIO的唯一值，可以确定当前的硬件版本，U-Boot就可以找到对应的dtb文件。

7.8.3 硬件参考设计

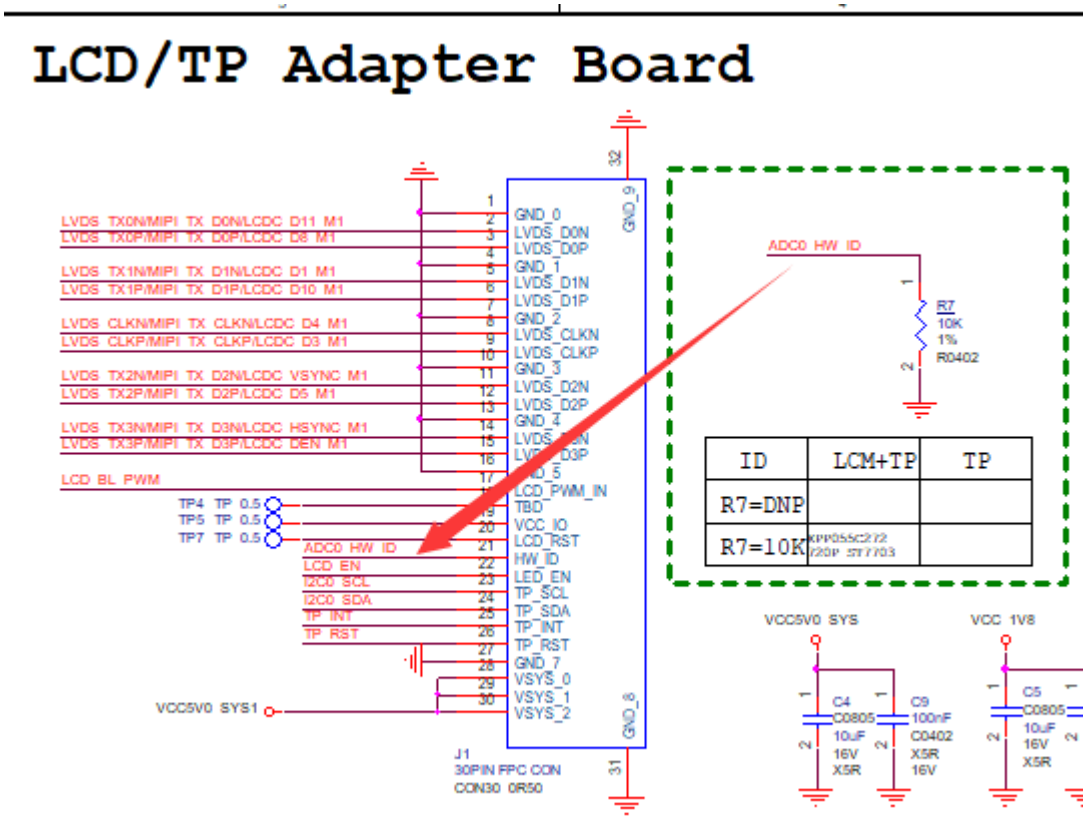
目前支持ADC和GPIO两种方式确定硬件版本。

ADC参考设计

RK3326-EVB\PX30-EVB主板上有预留分压电阻，不同的电阻分压可以确定不同的硬件版本号:



配套使用的MIPI屏小板预留有另外一颗下拉电阻:



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）

7.8.4 软件配置

把ADC和GPIO的信息放在dtb的文件名里面，U-Boot解析dtb的时候，从文件名中获取到当前dtb文件支持的硬件版本，并和实际的硬件版本做匹配。

ADC作为HW_ID的dtb文件命名规则

1. 文件名以“.dtb”结尾；
2. HW_ID格式： *#[controller]*ch[channel]=[adcval]

   [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。
3. 上述（2）表示一个完整含义，必须使用小写字母，一个完整含义内不能有空格之类的字符；
4. 多个含义之间通过#进行分隔，最多支持10个完整含义；

合法范例：

rk3326-evb-lp3-v10#*saradc_ch2=111*#saradc_ch1=810.dtb

rk3326-evb-lp3-v10#_saradc_ch2=569.dtb

GPIO作为HW_ID的dtb命令规则

1. 文件名以".dtb"结尾；
2. HW_ID格式：#gpio[pin]=[levle]

   [pin]: GPIO脚，如0a2表示gpio0a2

   [levle]: GPIO引脚电平。
3. 上述（2）表示一个完整含义，必须使用小写字母，一个完整含义内不能有空格之类的字符；
4. 多个含义之间通过#进行分隔，最多支持10个完整含义；

合法范例：rk3326-evb-lp3-v10#gpio0a2=0#gpio0c3=1.dtb

7.8.5 代码位置

代码实现位置在uboot工程：arch/arm/mach-rockchip/resource_img.c，主要包含下面两个函数：

```
static int rockchip_read_dtb_by_gpio(const char *file_name);
static int rockchip_read_dtb_by_adc(const char *file_name);
```

7.8.6 打包脚本

通过脚本，可以很方便地把多个dtb打包到一个resource.img里面，脚本位置在kernel工程：scripts/mkmultidtb.py，打开脚本文件，把需要打包的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')])
...
```

以上例子，执行scripts/mkmultidtb.py PX30-EVB，就会生成包含多份dtb的resource.img，包含以下三个dtb:

rk-kernel.dtb：rk默认的dtb，所有dtb都没匹配到时，会使用这个dtb，打包脚本使用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；

7.8.7 确认当前的dtb

下面是开机U-Boot的log：

mmc0(part 0) is current device boot mode: None **DTB: rk3326-evb-lp3-v10#_saradc_ch0=166.dtb** Using kernel dtb

从这个log可以看出来，当前硬件版本匹配到了resource.img里面的rk3326-evb-lp3-v10#_saradc_ch0=166.dtb，如果匹配失败，则会使用rk-kernel.dtb。

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的组合实现了跟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。因此在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，如果再加上kernel的dts，那么原有的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。当需要同时支持两种类型的驱动，请使用ofnode类型。

```
 * @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，
- 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。

用户可以通过查找.config是否包含CONFIG_USING_KERNEL_DTB确认是否已启用kernel dtb，该功能需要依赖live dt。因为读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是由defconfig里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。也就是串口、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U-Boot 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
```

11. rktest测试程序

rktest集成了对某些模块的测试命令，可以快速确认哪些模块是否正常。

命令格式：

```
=> rktest
Command: rktest [module] [args...]
  - module: timer|key|emmc|rknand|regulator|eth|ir|brom|rockusb|fastboot|vendor
  - args: depends on module, try 'rktest [module]' for test or more help

  - Enabled modules:
     -      timer: test timer and interrupt
     -       brom: enter bootrom download mode
     -    rockusb: enter rockusb download mode
     -   fastboot: enter fastboot download mode
     -        key: test board keys
     -  regulator: test regulator volatge set and show regulator status
     -     vendor: test vendor storage partition read/write
```

1. timer测试：用于确认当前环境下系统timer是否正常工作（延时是否准确）、系统中断是否正常。

```
=> rktest timer

sys timer delay test, round-1
        desire delay 100us, actually delay 100us
        desire delay 100ms, actually delay: 100ms
        desire delay 1000ms, actually delay: 1000ms
sys timer delay test, round-2
        desire delay 100us, actually delay 100us
        desire delay 100ms, actually delay: 100ms
        desire delay 1000ms, actually delay: 1000ms
sys timer delay test, round-3
```

```
        desire delay 100us, actually delay 100us
        desire delay 100ms, actually delay: 100ms
        desire delay 1000ms, actually delay: 1000ms
sys timer delay test, round-4
        desire delay 100us, actually delay 100us
        desire delay 100ms, actually delay: 100ms
        desire delay 1000ms, actually delay: 1000ms
timer_irq_handler: round-0, irq=114, period=1000ms
timer_irq_handler: round-1, irq=114, period=1000ms
timer_irq_handler: round-2, irq=114, period=1000ms
timer_irq_handler: round-3, irq=114, period=1000ms
timer_irq_handler: round-4, irq=114, period=1000ms
timer_irq_handler: irq test finish.
```

2. key测试：用于确认当前环境下系统的按键是否能正常响应。输入命令后，可以按下各个按键进行确认；按下ctrl+c组合键可以退出测试。

```
=> rktest key

volume up key pressed..
volume up key pressed..
volume down key pressed..
volume down key pressed..
volume up key pressed..
power key short pressed..
power key short pressed..
power key long pressed..
```

3. emmc测试：用于确认当前环境下系统的emmc读写速度。

   命令格式：rktest emmc<start_lba> <blocks>

```
=> rktest emmc 0x2000 2000

Round up to 8192 blocks compulsively

MMC write: dev # 0, block # 8192, count 8192 ... 8192 blocks written: OK
eMMC write: size 4MB, used 187ms, speed 21MB/s

MMC read: dev # 0, block # 8192, count 8192 ... 8192 blocks read: OK
eMMC read: size 4MB, used 95ms, speed 43MB/s
```

注意：测试后对应的被写存储区域的数据已经变化了。如果这个区域对应的是固件分区，则固件可能已经被破坏，请重新烧写固件。

4. rknand测试：用于确认当前环境下系统的rknand读写速度。

   命令格式：rktest rknand <start_lba> <blocks>

```
=> rktest rknand 0x2000 2000

Round up to 8192 blocks compulsively

rknand write: dev # 0, block # 8192, count 8192 ... 8192 blocks written: OK
rknand write: size 4MB, used 187ms, speed 21MB/s

rknand read: dev # 0, block # 8192, count 8192 ... 8192 blocks read: OK
rknand read: size 4MB, used 95ms, speed 43MB/s
```

5. vendor storage测试：用于确认当前环境下系统的vendor storage功能是否正常。

```
=> rktest vendor

[Vendor Test]:Test Start...
[Vendor Test]:Before Test, Vendor Resetting.
[Vendor Test]:<All Items Used> Test Start...
[Vendor Test]:item_num=126, size=448.
[Vendor Test]:<All Items Used> Test End,States:OK
[Vendor Test]:<Overflow Items Cnt> Test Start...
[Vendor Test]:id=126, size=448.
[Vendor Test]:<Overflow Items Cnt> Test End,States:OK
[Vendor Test]:<Single Item Memory Overflow> Test Start...
[Vendor Test]:id=0, size=6464.
[Vendor Test]:<Single Item Memory Overflow> Test End, States:OK
[Vendor Test]:<Total memory overflow> Test Start...
[Vendor Test]:item_num=9, size=6464.
[Vendor Test]:<Total memory overflow> Test End, States:OK
[Vendor Test]:After Test, Vendor Resetting...
[Vendor Test]:Test End.
```

6. maskrom下载模式识别测试：用于确认当前环境下，能否退回到maskrom模式进行烧写。

```
=> rktest brom

敲完命令可以看下烧写工具是否显示当前处于maskrom烧写模式，且能正常进行固件下载。
```

7. regulator测试：用于显示各路regulator的dts配置状态、当前的实际状态；BUCK调压是否正常。

```
=> rktest regulator
```

打印dts配置和当前实际各路电压情况：

```
<Board dts config>:
               DCDC_REG1@     vdd_center:  750000uV <-> 1350000uV, set  750000uV, enabling | supsend     -61uV, disabled
               DCDC_REG2@      vdd_cpu_l:  750000uV <-> 1350000uV, set  750000uV, enabling | supsend     -61uV, disabled
               DCDC_REG3@        vcc_ddr:     -61uV <->     -61uV, set     -61uV, enabling | supsend     -61uV, enabling
               DCDC_REG4@        vcc_1v8: 1800000uV <-> 1800000uV, set 1800000uV, enabling | supsend 1800000uV, enabling
                LDO_REG1@     vcc1v8_dvp: 1800000uV <-> 1800000uV, set 1800000uV, enabling | supsend     -61uV, disabled
                LDO_REG2@      vcc3v0_tp: 3000000uV <-> 3000000uV, set 3000000uV, enabling | supsend     -61uV, disabled
                LDO_REG3@     vcc1v8_pmu: 1800000uV <-> 1800000uV, set 1800000uV, enabling | supsend 1800000uV, enabling
                LDO_REG4@       vccio_sd: 1800000uV <-> 3000000uV, set 1800000uV, enabling | supsend 3000000uV, enabling
                LDO_REG5@  vcca3v0_codec: 3000000uV <-> 3000000uV, set 3000000uV, enabling | supsend     -61uV, disabled
                LDO_REG6@        vcc_1v5: 1500000uV <-> 1500000uV, set 1500000uV, enabling | supsend 1500000uV, enabling
                LDO_REG7@  vcca1v8_codec: 1800000uV <-> 1800000uV, set 1800000uV, enabling | supsend     -61uV, disabled
                LDO_REG8@        vcc_3v0: 3000000uV <-> 3000000uV, set 3000000uV, enabling | supsend 3000000uV, enabling
             SWITCH_REG1@      vcc3v3_s3:     -61uV <->     -61uV, set     -61uV, enabling | supsend     -61uV, disabled
             SWITCH_REG2@      vcc3v3_s0:     -61uV <->     -61uV, set     -61uV, enabling | supsend     -61uV, disabled
              vcc3v3-sys@     vcc3v3_sys: 3300000uV <-> 3300000uV, set 3300000uV, enabling | supsend     -61uV, enabling
   vcc5v0-host-regulator@    vcc5v0_host:     -61uV <->     -61uV, set     -61uV, enabling | supsend     -61uV, enabling
             vcc5v0-sys@     vcc5v0_sys: 5000000uV <-> 5000000uV, set 5000000uV, enabling | supsend     -61uV, enabling
                 vcc-sd@        vcc_sd: 3300000uV <-> 3300000uV, set 3300000uV, disabled | supsend     -61uV, enabling
      vcc-phy-regulator@        vcc_phy:     -61uV <->     -61uV, set     -61uV, enabling | supsend     -61uV, enabling
                vdd-log@        vdd_log:  800000uV <-> 1400000uV, set  800000uV, enabling | supsend     -61uV, enabling
                vcc-lcd@        vcc_lcd: 3300000uV <-> 3300000uV, set 3300000uV, enabling | supsend     -61uV, enabling

<Board current status>:
               DCDC_REG1@     vdd_center: set  900000uV, enabling | suspend  712500uV, disabled
               DCDC_REG2@      vdd_cpu_l: set  900000uV, enabling | suspend  712500uV, disabled
               DCDC_REG3@        vcc_ddr: set  712500uV, enabling | suspend  712500uV, enabling
               DCDC_REG4@        vcc_1v8: set 1800000uV, enabling | suspend 1800000uV, enabling
                LDO_REG1@     vcc1v8_dvp: set 1800000uV, enabling | suspend 1800000uV, disabled
                LDO_REG2@      vcc3v0_tp: set 3000000uV, enabling | suspend 1800000uV, disabled
                LDO_REG3@     vcc1v8_pmu: set 1800000uV, enabling | suspend 1800000uV, enabling
                LDO_REG4@       vccio_sd: set 3000000uV, enabling | suspend 3000000uV, enabling
                LDO_REG5@  vcca3v0_codec: set 3000000uV, enabling | suspend 1800000uV, disabled
                LDO_REG6@        vcc_1v5: set 1500000uV, enabling | suspend 1500000uV, enabling
                LDO_REG7@  vcca1v8_codec: set 1800000uV, enabling | suspend  800000uV, disabled
                LDO_REG8@        vcc_3v0: set 3000000uV, enabling | suspend 3000000uV, enabling
             SWITCH_REG1@      vcc3v3_s3: set     -38uV, enabling | suspend       0uV, disabled
             SWITCH_REG2@      vcc3v3_s0: set     -38uV, enabling | suspend       0uV, disabled
              vcc3v3-sys@     vcc3v3_sys: set 3300000uV, enabling | suspend     -38uV, enabling
   vcc5v0-host-regulator@    vcc5v0_host: set     -61uV, enabling | suspend     -38uV, enabling
             vcc5v0-sys@     vcc5v0_sys: set 5000000uV, enabling | suspend     -38uV, enabling
                 vcc-sd@        vcc_sd: set 3300000uV, enabling | suspend     -38uV, enabling
      vcc-phy-regulator@        vcc_phy: set     -61uV, enabling | suspend     -38uV, enabling
                vdd-log@        vdd_log: set      -1uV, enabling | suspend     -38uV, enabling
                vcc-lcd@        vcc_lcd: set 3300000uV, enabling | suspend     -38uV, enabling
```

调压精度测试：

```
[DCDC_REG1@vdd_center] set: 900000 uV -> 912500 uV;  ReadBack: 912500 uV

Confirm 'vdd_center' voltage, then hit any key to continue...

[DCDC_REG1@vdd_center] set: 912500 uV -> 937500 uV;  ReadBack: 937500 uV

Confirm 'vdd_center' voltage, then hit any key to continue...

[DCDC_REG1@vdd_center] set: 937500 uV -> 975000 uV;  ReadBack: 975000 uV

Confirm 'vdd_center' voltage, then hit any key to continue...

[DCDC_REG2@vdd_cpu_l] set: 900000 uV -> 912500 uV;  ReadBack: 912500 uV

Confirm 'vdd_cpu_l' voltage, then hit any key to continue...

[DCDC_REG2@vdd_cpu_l] set: 912500 uV -> 937500 uV;  ReadBack: 937500 uV

Confirm 'vdd_cpu_l' voltage, then hit any key to continue...

[DCDC_REG2@vdd_cpu_l] set: 937500 uV -> 975000 uV;  ReadBack: 975000 uV

Confirm 'vdd_cpu_l' voltage, then hit any key to continue..
```

8. ethernet测试

   [TODO]

9. ir测试

[TODO]

附录

IRAM程序内存分布(SPL/TPL)

bootRom出来后的第一段代码在Intermal SRAM(U-Boot叫IRAM), 可能是TPL或者SPL, 同时存在TPL和SPL时描述的是TPL的map, SPL的map类似.

Name	start addr	size	Desc
Bootrom	IRAM_START	TPL_TEXT_BASE-IRAM_START	data and stack
TAG	TPL_TEXT_BASE	4	RKXX
text	TEXT_BASE	sizeof(text)	
bss	text_end	sizeof(bss)	append to text
dtb	bss_end	sizeof(dtb)	append to bss
SP	gd start		stack
gd	malloc_start - sizeof(gd)	sizeof(gd)	
malloc	IRAM_END-MALLOC_F_LEN	*PL_SYS_MALLOC_F_LEN	malloc_simple

text, bss, dtb的空间是编译时根据实际内容大小决定的； malloc, gd, SP是运行时根据配置来确定的位置； 一般要求dtb尽量精简,把空间留给代码空间, text如果过大, 运行时比较容易碰到的问题是Stack把dtb冲了, 导致找不到dtb.

U-Boot内存分布(relocate后)

U-Boot代码一开始由前级Loader搬到TEXT_BASE的位置,U-Boot在探明实际可用DRAM空间后,把自己relocate到ram_top位置, 其中Relocation Offset = 'U-Boot start - TEXT_BASE'.

Name	start addr	size	Desc
ATF	RAM_START	0x200000	Reserved for bl31
OP-TEE	0x8400000	2M~16M	参考TEE开发手册
kernel fdt	fdt_addr_r		
kernel	kernel_addr_r		
ramdisk	ramdisk_addr_r		
fastboot buffer	CONFIG_FASTBOOT_BUF_ADDR	CONFIG_FASTBOOT_BUF_SIZE	
SP			stack
FDT		sizeof(dtb)	U-Boot自带dtb
GD		sizeof(gd)	
Board		sizeof(bd_t)	board info, eg. dram size
malloc		TOTAL_MALLOC_LEN	约64M
U-Boot		sizeof(mon)	含text, bss
Video FB		fb size	约32M
TLB table	RAM_TOP-64K	32K	

Video FB/U-Boot/malloc/Board/GD/FDT/SP是由顶向下根据实际需求大小来分配的, 起始地址对齐到4K大小； ATF在armv8是必需的, 属于TE, armv7没有； OP-TEE在armv7属于TE+TOS, 可选, 根据是否需要TA来确定大小；在armv8属于bl32(TOS), 可选, 依据内含TA数量来确定大小；U-Boot在dram_init_banksize()函数解析实际占用空间；kernel fdt/kernel/ramdisk几个起始位置在includ/config/rkxx_common.h中的ENV_MEM_LAYOUT_SETTINGS定义, 注意不能和已定义位置重合； FASTBOOT/ROCKUSB等下载功能的BUFFER地址,在config/evb-rkxx_defconfig中定义, FASTBOOT_BUF_ADDR注意不能和已定义位置重合, 可以跟上一条内容重合；

fastboot一些参考

make_unlock.sh参考

```
#!/bin/sh
python avb-challenge-verify.py raw_atx_unlock_challenge.bin atx_product_id.bin
python avbtool make_atx_unlock_credential --output=atx_unlock_credential.bin --
intermediate_key_certificate=atx_pik_certificate.bin --
unlock_key_certificate=atx_puk_certificate.bin --challenge=atx_unlock_challenge.bin --
unlock_key=testkey_atx_puk.pem
```

avb-challenge-verify.py源码

```
#/user/bin/env python
"this is a test module for getting unlock challenge"
```

```
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('atx_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 challege 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()
```

rkbin仓库下载

1. Rockchip内部工程师：

   登录gerrit -> project -> list -> Filter搜索框输入：“rk/rkbin” -> 下载；

2. 外部工程师：

   （1）下载产品部门发布的完整SDK工程；

   （2）从Github下载：https://github.com/rockchip-linux/rkbin。

gcc编译器下载

1. Rockchip内部工程师：

   登录gerrit -> project -> list -> Filter搜索框输入：“gcc-linaro-6.3.1” -> 下载；

2. 外部工程师：

   下载产品部门发布的完整SDK工程；