密级状态：绝密(　　)　　秘密(　　)　　内部(　　)　　公开(√)

PX3SE_Linux 常见问题解决方法

（技术部，产品二部）

文件状态： [√] 正在修改 [　] 正式发布	当前版本：	V1.0
	作　　者：	黄国椿
	完成日期：	2017-09-13
	审　　核：	王剑辉 邓训金
	完成日期：	2017-09-13

福州瑞芯微电子股份有限公司

Fuzhou Rockchips Electronics Co . , Ltd

版 本 历 史

版本号	作者	修改日期	修改说明	审核	备注
V1.0	黄国椿	2017-09-13	发布初版	王剑辉 邓训金	

目 录

1. 如何创建 GPIO 设备节点及 Debug 方法

1.1 DTS 中创建描述 GPIO 设备信息的节点

首先在 DTS 文件中增加驱动的资源描述：

```
kernel/arch/arm/boot/dts/px3se-sdk-dts

data-pm25 {
        compatible = "data-pm25";
        PM25_power_gpio=<&gpio1 GPIO_C5 GPIO_ACTIVE_HIGH>;
        pinctrl-names = "default";
        pinctrl-0 = <&pm25_gpio>;
        status = "okay";
    };
```

这里定义了一个脚作为一般的输出输入口：

```
PM25_power_gpio GPIO1_C5
```

GPIO_ACTIVE_HIGH 表示高电平有效，如果想要低电平有效，可以改为：GPIO_ACTIVE_LOW，这个属性将被驱动所读取。

1.2 GPIO 的驱动

在 probe 函数中对 DTS 所添加的资源进行解析，代码如下：

```
static int data_pm25_probe(struct platform_device *pdev)
{
    ... ...
    gpio = of_get_named_gpio_flags(node, "PM25_power_gpio", 0, &flags);
    if (!gpio_is_valid(gpio)) {
    LOG("%s: get property: PM25,power_gpio = %d. is invalid\n", __func__, gpio);
```

```
        return -ENODEV;
    }else{
        err = gpio_request(&pdev->dev, gpio, "PM25_power_gpio");
        if(err){
        dev_err(&pdev->dev,"failed to request GPIO%d for PM25_power\n", gpio);
        ret = err;
        gpio_free(gpio);
        return -ENODEV;
        }
    }
    g_pm25->io = gpio;
    g_pm25->enable = (flags == GPIO_ACTIVE_HIGH)? 1:0;
    gpio_direction_output(g_pm25->io, g_pm25->enable);
    ret = device_create_file(&pdev->dev, &dev_attr_enable);
    if(ret) {
        LOG("Failed to create PM25,power_gpio file.\n");
        return -ENODEV;
    }
    return 0;
}
```

of_get_named_gpio_flags 从设备树中读取 PM25_power_gpio 的 GPIO 配置编号和标志，gpio_is_valid 判断该 GPIO 编号是否有效，gpio_request 则申请占用该 GPIO。如果初始化过程出错，需要调用 gpio_free 来释放之前申请过且成功的 GPIO。在驱动中调用 gpio_direction_output 就可以设置输出高还是低电平，这里默认输出从 DTS 获取得到的有效电平 GPIO_ACTIVE_HIGH，即为高电平，如果驱动正常工作，可以用万用表测得对应的引脚应该为高电平。 实际中如果要读出 GPIO，需要先设置成输入模式，然后再读取值：

```
int val;
gpio_direction_input(your_gpio);
val=gpio_get_value(your_gpio);
```

当使用 gpio_request 时候，会将该 PIN 的 MUX 值强制切换为 GPIO，所以使用该 pin 脚为 GPIO 功能的时候确保该 pin 脚没有被其他模块所使用。

下面是常用的 GPIO API 定义：

```
int of_get_named_gpio_flags(struct device_node *np, const char *propname,
                            int index, enum of_gpio_flags *flags);
 int gpio_is_valid(int gpio);
 int gpio_request(unsigned gpio, const char *label);
 void gpio_free(unsigned gpio);
 int gpio_direction_input(int gpio);
 int gpio_direction_output(int gpio, int v)
int gpio_set_value(int gpio,int v)
```

gpio_set_value()与 gpio_direction_output()有什么区别？

如果使用该 GPIO 时，不会动态的切换输入输出，建议在开始时就设置好 GPIO 输出方向，后面拉高拉低时使用 gpio_set_value()接口，而不建议使用 gpio_direction_output()，因为 gpio_direction_output 接口里面有 mutex 锁，对中断上下文调用会有错误异常，且相比 gpio_set_value，gpio_direction_output 所做事情更多。

1.3 创建设备节点

```
static ssize_t px3se_dev_enable(struct device *dev, struct device_attribute *attr,
const char *buf, size_t count)
{
    struct hidraw *devraw = dev_get_drvdata(dev);
    if (0 == strncmp(buf, "1", 1)) {
```

```
        gpio_set_value(g_pm25->io, 1);
    } else {
        gpio_set_value(g_pm25->io, 0);
    }
    return count;
}
static DEVICE_ATTR(enable, S_IWUSR, NULL, px3se_dev_enable);
```

在 data_pm25_probe 中通过调用 device_create_file(&pdev->dev, &dev_attr_enable);

即可创建一个 enable 的设备节点。

2 如何查看配置 GPIO 的状态

```
[root@rockchip:/]# cat /sys/kernel/debug/gpio
GPIOs 0-31, platform/pinctrl, gpio0:
 gpio-4   (                    |bt_default_wake_host) in  hi
 gpio-5   (                    |GPIO Key Power      ) in  hi
 gpio-9   (                    |bt_default_reset    ) out hi
 gpio-10  (                    |reset               ) out hi
GPIOs 32-63, platform/pinctrl, gpio1:
 gpio-34  (                    |int-n               ) in  hi
 gpio-45  (                    |enable              ) out hi
 gpio-46  (                    |vsel                ) out lo
 gpio-49  (                    |vsel                ) out lo
GPIOs 64-95, platform/pinctrl, gpio2:
 gpio-64  (                    |vbus-5v             ) out lo
 gpio-83  (                    |bt_default_rts      ) out lo
 gpio-90  (                    |bt_default_wake     ) out hi
```

```
GPIOs 96-127, platform/pinctrl, gpio3:
 gpio-111 (                    |mdio-reset           ) out hi
GPIOs 128-159, platform/pinctrl, gpio4:
 gpio-150 (                    |?                    ) out hi
 gpio-153 (                    |vcc5v0_host          ) out hi
 gpio-158 (                    |enable               ) out hi
```

从上面可以比较清楚看到系统中将 GPIO 配置输入输出以及输出电平的情况，如果在/sys/kernel/debug/未能发现相应的文件，则需要检查内核是否使能 debugfs：

```
Symbol: DEBUG_FS [=y]
  Type    : boolean
  Prompt: Debug Filesystem
    Defined at lib/Kconfig.debug:77
    Location:
    -> Kernel hacking
```

启动目标硬件并挂载：

```
mount -t debugfs none /sys/kernel/debug
```

3 如何实时读写每个 IO 口的状态

GPIO 调试有一个很好用的工具，那就是 IO 指令，px3se 的 linux 系统已经默认内置了 IO 命令，使用 IO 指令可以实时读取或写入每个 IO 口的状态，这里简单介绍 IO 指令的使用，首先查看 io 指令帮助：

```
[root@rockchip:/]# io --help
Unknown option: ?
Raw memory i/o utility - $Revision: 1.5 $

io -v -1|2|4 -r|w [-l <len>] [-f <file>] <addr> [<value>]

   -v         Verbose, asks for confirmation
```

```
    -1|2|4      Sets memory access size in bytes (default byte)
    -l <len>    Length in bytes of area to access (defaults to
                one access, or whole file length)
    -r|w        Read from or Write to memory (default read)
    -f <file>   File to write on memory read, or
                to read on memory write
    <addr>      The memory address to access
    <val>       The value to write (implies -w)


Examples:
    io 0x1000                   Reads one byte from 0x1000
    io 0x1000 0x12              Writes 0x12 to location 0x1000
    io -2 -l 8 0x1000           Reads 8 words from 0x1000
    io -r -f dmp -l 100 200     Reads 100 bytes from addr 200 to file
    io -w -f img 0x10000        Writes the whole of file to memory
Note access size (-1|2|4) does not apply to file based accesses.
```

从以上信息可以看出，如果要读或写一个寄存器，可以用：

```
io -4 -r 0x1000 //读从 0x1000 起的 4 位寄存器的值
io -4 -w 0x1000 //写从 0x1000 起的 4 位寄存器的值
```

使用事例：

查看 GPIO2_B5 引脚的复用情况

1. 从主控的 datasheet 查到 GPIO2 对应寄存器基地址为：0x0x20084000
2. 从主控的 datasheet 查到 GPIO2B_IOMUX 的偏移量为：0x000cc
3. GPIO2_B5 的 iomux 寄存器地址为：基址(Operational Base) + 偏移量(offset)=0x20080000+0x000bc=0x200840cc
4. 用以下指令查看 GPIO2_B5 的复用情况：

```
[root@rockchip:/]# io -4 -r 0x200840cc
200840cc:  00008000
```

5.从 datasheet 查到[11:10]

```
gpio2b5_sel
GPIO2B[5] iomux select
01:lcdc0_d11
10: ebc_sdce3
11: gmac_txen
00: gpio
```

因此可以确定该 GPIO 就是复用为普通 GPIO。

6.如果想要复用为 lcdc0_dll 功能，可以使用以下指令设置：

```
#io -4 -w 0x200840cc 0x1000a000
```

如果用 IO 命令读某个 GPIO 的寄存器，读出来的值异常,如 0x00000000 或 0xffffffff 等，请确认该 GPIO 的 CLK 是不是被关了，GPIO 的 CLK 是由 CRU 控制，可以通过读取 datasheet 下面 CRU_CLKGATE_CON* 寄存器来查到 CLK 是否开启，如果没有开启可以用 io 命令设置对应的寄存器，从而打开对应的 CLK，打开 CLK 之后应该就可以读到正确的寄存器值了。

4 如何实现 UBOOT 开机充电动画

4.1 dts 中创建充电模式的开关配置

```
uboot-charge {
compatible = "rockchip,uboot-charge";
rockchip,uboot-charge-on = <1>;
};
```

这样就可以灵活配置使用 uboot 的关机充电模式。

4.2 配置 u-boot

为了实现充电动画，需要在 uboot/include/config/rk30plat.h 中定义如下宏开关：

```
#define CONFIG_UBOOT_CHARGE
#define CONFIG_SCREEN_ON_VOL_THRESD 3400//3.4v
#define CONFIG_SYSTEM_ON_VOL_THRESD 3500//3.5v
```

其中 CONFIG_SCREEN_ON_VOL_THRESD 是系统点亮屏幕的电压门限，低于这个电压，禁止系统亮屏。CONFIG_SYSTEM_ON_VOL_THRESD 是系统正常启动的电压门限，低于这个电压，禁止 uboot 启动内核，这两个电压可以根据具体的产品设计灵活调整。

4.3 打包动画资源

选择使用 uboot 阶段充电动画，需要将动画图片资源文件打包在 resource.img，方法如下：

在 uboot 根目录下执行：

```
sudo ./tools/resource_tool/pack_resource.sh tools/resource_tool/resources/ ../kernel
/resource.img resource.img tools/resource_tool/
```

将 tools/resource_tool/resources/目录下的动画图片资源打包，并在 uboot 根目录生成新的 resource.img。

5 px3se 小容量系统分区配置

px3se 根据存储方式有四种编译方法，有时候因为 rootfs.img 定制容量超过预定义的容量大小，因此编译后烧写固件发现系统不断重启。

log 如下：

```
... ...
[     4.467210] List of all partitions:
[     4.467261] 1f00              448 idblock   (driver?)
```

[4.467293] 1f01 8448 kernel (driver?)

[4.467319] 1f02 65536 rootfs (driver?)

[4.467346] 1f03 10240 userdata (driver?)

[4.467372] 1f04 2 fw_header_p (driver?)

[4.467392] No filesystem could mount root, tried: squashfs

[4.467423] Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(31,2)

... ...

这个时候需要重新调整 rootfs.img 的容量大小区间：

在工程根目录相应的文件比如：../device/rockchip/px3-se/mini_fs/setting_emmc.ini 中

```
[IDBlock]
Flag=1
DDR_Bin=./px3seddr.bin
Loader_Bin=./px3seloader.bin
PartOffset=0x40
PartSize=0x380
[UserPart1]
Name=kernel
Type=0x4
Flag=1
File=./kernel.img
PartOffset=0x2000
PartSize=0x4200
[UserPart2]
Name=rootfs
Type=0x8
```

```
Flag=1
File=./rootfs.img
PartOffset=0x6200
PartSize=0x40000
[UserPart3]
Name=userdata
Type=0x80000000
```

可以根据你所编译的固件大小适当调整对应的 PartSize，比如这个文件中默认 kernel.img 的大小是 8M，rootfs.img 的大小是 128M。如果你所定制的 rootfs.img 超过该大小，尝试增大这个容量大小，并相应修改下面存储地址的起始地址，再重新打包你的固件并下载。

6 ADB 使用

6.1 Windows 下的 ADB 安装

首先安装 RK USB 驱动。

然后到 http://adbshell.com/download/download-adb-for-windows.html 下载 adb.zip，解压到 C:\adb 以方便调用。

打开命令行窗口，输入：

```
cd C:\adb
adb shell
```

如果一切正常，就可以进入 adb shell ，在设备上面运行命令。

6.2 Ubuntu 下的 ADB 安装

安装 adb 工具：

```
sudo apt-get -y install android-tools-adb
```

加入设备标识：

```
mkdir -p ~/.android

vi ~/.android/adb_usb.ini

#添加以下一行

0x2207
```

加入 udev 规则：

```
sudo vi /etc/udev/rules.d/51-android.rules

#添加如下：

SUBSYSTEM==”usb”,ATTR{idvendor}==”2207”,MODE=”0666”
```

重新插拔数据线 type-C，或运行如下命令，让 udev 规则生效：

```
sudo udevadm control --reload-rules

sudo udevadm trigger
```

重新启动 adb 服务器

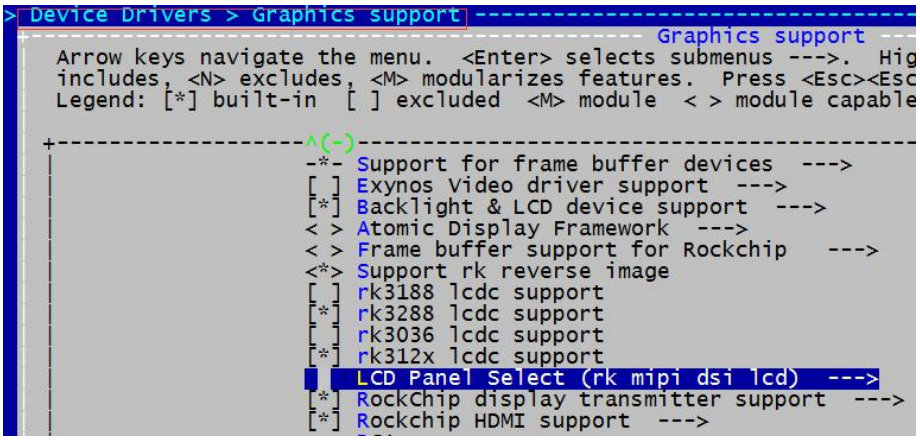
```
sudo adb kill-server

adb start-server
```

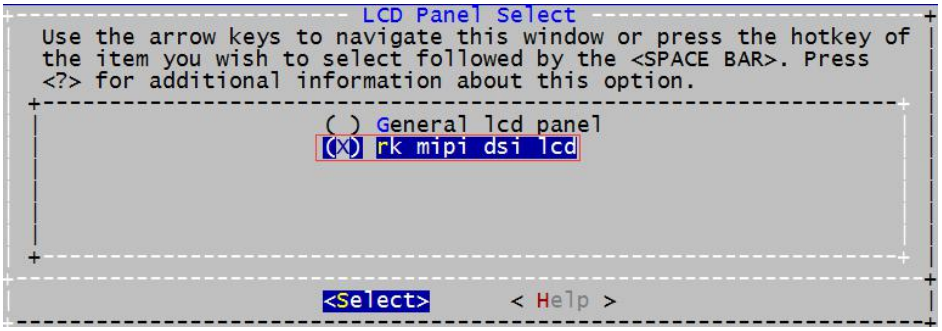
每次插上 usb 后 adb shell 即可。

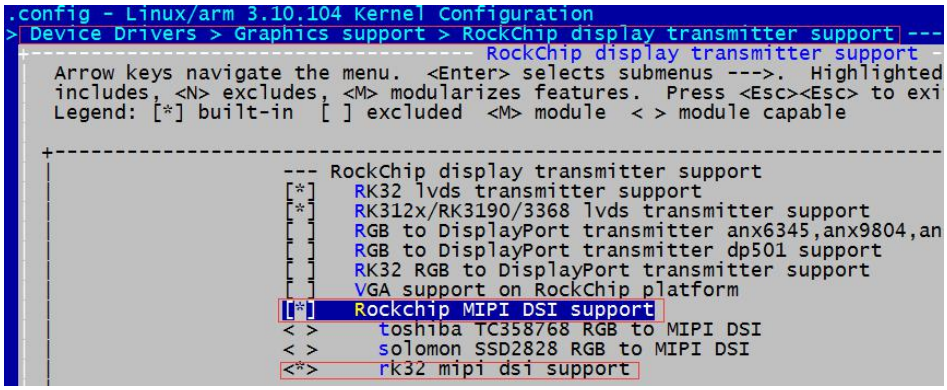
7 MIPI LCD 调试

7.1 make menuconfig

要调用到 MIPI 驱动需要在 kernel/下 make menuconfig 来进行配置，目前在开发分支里面已经默认配置好，开启和关闭由板级 dts 文件来控制。配置如下列所示（红色标记表示配置路径和需要配置的选项）：







7.2 屏参文件配置与说明

屏参文件包含四个部分： mipi host 配置、屏电源控制配置、屏初始化序列和屏参。

屏参文件的解析： mipi host 配置、屏电源控制配置、屏初始化序列三部分是在 drivers/video/rockchip/screen/lcd_mipi.c 中解析的，最后的屏参是在 drivers/video/of_display_timing.c 中解析。因为该部分信息 mipi/edp/lvds/hdmi 之类显示设备都存在，所以在统一的地方进行解析。

屏参文件所在目录为 arch/arm/boot/dts/中。

mipi host 配置结构如下所示：

```
disp_mipi_init: mipi_dsi_init {
    compatible = "rockchip,mipi_dsi_init";
    rockchip,screen_init = <1>;
    rockchip,dsi_lane = <2>;
    rockchip,dsi_hs_clk = <800>;
    rockchip,mipi_dsi_num = <1>;
};
```

screem_init:表示屏是否需要初始化，如果需要则置 1.

dsi_lane：mipi 数据传输需要几条数据 lane，这个一般根据原理图和 mipi 屏的规格书来配置。这个指的是每个 mipi 的数据 lane 数。譬如如果是单 mipi，每个 mipi 为 2 lane。那么此处仍然设置为 2。

dsi_hs_clk：屏 ddr clk，表示一条数据 lane 的传输速率，单位为 Mbits/s。有个大概的计算公式： 100+H_total*V_total*fps*3*8/lanes。

H_total， V_total 包括 active， bp， fp 和 sync-len 的和；

fps 为帧率， 刚调试一款屏时， fps 为 50 多帧就好，然后慢慢抬高；

3 表示一个像素点，代表 rgb 的 3 个字节；

8 为 8 bits；

lanes 为(dsi_lane*mipi_dsi_num) ；

100 为实际的结果要比理论值大 100M 左右。

上面计算得到的值只是大概值并非精确的值，但是对于一般的屏都适用，对于部分屏需要微调该

值。

mipi_dsi_num ： 单 mipi 还是双 mipi，也是根据原理图和屏幕规格书来配置的。如果是双 mipi 则置为 1。 单 MIPI 的屏 SDK 代码默认设置的是 MIPI_TX，所以单 MIPI 是接 MIPI_TX 这一路；双 MIPI 接法： MIPI_TX 这路接左屏， MIPI_TX/RX 这一路接右屏。

7.3 屏电源控制配置

屏的电源控制配置可以放在 dtsi 中的 lcdc 模块中去配置，而且默认放在 lcdc 中去配置。原因有二：一、我们 mipi/edp/lvds 电源控制部分是一致的，放在 lcdc 中便于兼容；二、客户一般都会基于我们的硬件参考资料去布板，所以屏的电源控制部分基本都和我们 SDK 一致。但是如果客户的电源控制部分和我们 SDK 有差异，那么可以在屏参文件中单独配置。屏电源控制配置的结构如下：

```
disp_mipi_power_ctr: mipi_power_ctr {
compatible="rockchip,mipi_power_ctr";
mipi_lcd_rst:mipi_lcd_rst{
compatible = "rockchip,lcd_rst";
rockchip,gpios = <&gpio3 GPIO_D6 GPIO_ACTIVE_HIGH>;
rockchip,delay = <10>;
};
mipi_lcd_en:mipi_lcd_en {
compatible = "rockchip,lcd_en";
rockchip,gpios = <&gpio7 GPIO_A3 GPIO_ACTIVE_HIGH>;
rockchip,delay = <10>;
};
mipi_lcd_cs:mipi_lcd_cs {
compatible = "rockchip,lcd_cs";
rockchip,gpios = <&gpio7 GPIO_A4 GPIO_ACTIVE_HIGH>;
```

```
rockchip,delay = <10>;
    };
};
```

电源配置的 gpios 需要根据原理图来配置 lcd_en 等各对应哪路 gpio，另外还可能存在不需要三路电源控制的情况。

delay 部分是操作完后的延迟时间，这个关系到 mipi 的上电时序。在需要的时候要对延时时间进行调整。

对应的操作函数： driver/video/rockchip/screen/lcd_mipi.c 的 rk_mipi_screen_pwr_enable()， rk_mipi_screen_pwr_disable()。uboot 中对应的操作也是这两个函数。

7.4 屏初始化序列

屏是否需要初始化要根据屏的规格书来确定，如果需要初始化，可以向屏厂要初始化序列 。

屏初始化命令发送在 driver/video/rockchip/screen/lcd_mipi.c 的 rk_mipi_screen_cmd_init()中完成。

屏初始化序列的结构如下所示：

```
disp_mipi_init_cmds: screen-on-cmds {
compatible = "rockchip,screen-on-cmds";
rockchip,cmd_debug = <1>;
rockchip,on-cmds1 {
compatible = "rockchip,on-cmds";
rockchip,cmd_type = <HSDT>;
rockchip,dsi_id = <2>;
rockchip,cmd = <0xb0 0x02>;
rockchip,cmd_delay = <0>;
    };
};
```

rockchip,on-cmds1 结构：便是一条初始化命令的结构，如果有多条初始化命令，那么需要多个命令结构。

cmd_type：命令是在 low power(LPDT)还是 high speed(HSDT)下发送。

dsi_id：命令通过哪个 mipi 发送。 0 表示在 mipi0 发送， 1 表示在 mipi1 发送， 2 表示双 mipi 同时发送。因为很少出现单独使用 mipi1 的情况，所以对于单 mipi，这个值默认是 0；对于双 mipi，这个值是 2。

cmd：初始化命令。格式：命令类型（如 0x05/0x15/0x39）+命令+参数。具体细节见 mipi 协议文档。

cmd_delay：命令发完后延迟时间，这个根据屏厂给的初始化序列来配置。

如果屏幕不需要初始化，也就是 rockchip,screen_init 为 0 的情况。不需要初始化并不是表示没有发送命令。在不需要初始化的情况下，我们等 mipi host, phy 供电初始化完以及屏的供电结束以后会按照 mipi 协议发送 exit_sleep_mode 和 set_display_on 命令。

Exit_sleep_mode：表示退出 sleep 模式。

Set_display_on：表示告诉显示设备（屏）可以开始显示图像数据了。

7.5 屏参

屏参文件包括屏幕的格式， dclk，时序等信息。如下所示，其中红色标记的是需要重点关注的参数：

```
disp_timings: display-timings {
native-mode = <&timing0>;
compatible = "rockchip,display-timings";
timing0: timing0 {
screen-type = <SCREEN_MIPI>;
lvds-format = <LVDS_8BIT_2>;
out-face = <OUT_P888>;
clock-frequency = <145000000>;
```

```
hactive = <1920>;
vactive = <1200>;
hback-porch = <16>;
hfront-porch = <24>;
vback-porch = <10>;
vfront-porch = <16>;
hsync-len = <10>;
vsync-len = <3>;
hsync-active = <0>;
vsync-active = <0>;
de-active = <0>;
pixelclk-active = <0>;
swap-rb = <0>;
swap-rg = <0>;
swap-gb = <0>;
    }
}
```

screen-type：屏幕类型，对于 mipi 屏来说有两种选择：单 mipi（SCREEN_MIPI）；双 mipi（SCREEN_DUAL_MIPI）。

lvds-format：lvds 数据格式。对于 mipi 来说是无效参数，不用配置。

out-face：屏幕接线格式

上述三个参数的取值在 include/dt-bindings/rkfb/rk_fb.h 中定义。

clock-frequency：dclk 频率，单位为 Hz，一般屏的规格书中有，如果没有可以通过公式计算：H*V(包括同步信号)*fps

Hactive：水平有效像素

Vactive：垂直有效像素

hback-porch/hfront-porch/hsync-len：水平同步信号

vback-porch/vfront-porch/vsync-len：垂直同步信号

hsync-active 、vsync-active 、de-active 、pixelclk-active：分 别 为 hync,vsync, DEN,dclk 的极性控制。置 1 将对极性进行翻转。

swap-rb、swap-rg、swap-gb：设 1 将对对应的颜色进行翻转。

屏参文件配置好以后，需要在板级文件中包含这个屏参文件。