

密级状态：绝密( ) 秘密( ) 内部( ) 公开( √ )

## RockChip\_DSS 开发文档

(技术部, MID 组)

文件状态：  [ ] 正在修改  [√] 正式发布	当前版本：	V1.2
	作 者：	yxj, xjh, hhh, zwl
	完成日期：	2014-06-17
	审 核：	XXX
	完成日期：	2014-XX-XX

说明：该文档针对 RK30XX 、RK2928、 RK31XX、RK32XX 以后的 FB、  
LCDC、LVDS、MIPI、HDMI 等显示接口组成的 DSS (Display  
Subsystem) 驱动框架

福州瑞芯微电子有限公司

Fuzhou Rockchips Semiconductor Co., Ltd

(版本所有, 翻版必究)

## 版 本 历 史

版本号	作者	修改日期	修改说明	备注
V1.0		2013-12-22	初始版本	
V1.1		2014-02-26	增加 Q&A	
V1.2		2014-06-18	linux 3.10 版本内核的支持	

# 目录

<b>1</b>	<b>LCD 硬件原理 .....</b>	<b>4</b>
<b>2</b>	<b>LCD 驱动框架及实现 .....</b>	<b>9</b>
2.1	FB 框架相关的代码 .....	9
2.2	LCDC 相关的代码 .....	9
2.3	LCD 屏幕配置相关代码 .....	10
2.4	LCD 电源相关操作的配置以及配置 .....	17
2.4.1	基于 Linux3.0.36 内核的板级配置 .....	17
2.4.2	基于 Linux 3.10 内核的 dts 配置 .....	27
2.4.3	make menuconfig 相关配置 .....	29
<b>3</b>	<b>HDMI .....</b>	<b>33</b>
3.1	HDMI 基础 .....	33
3.2	HDMI 驱动框架及实现 .....	35
3.2.1	HDMI 代码框架 .....	35
3.2.2	HDMI 软件流程 .....	36
3.3	HDMI 相关配置 .....	37
3.3.1	Linux3.0.36 内核配置 .....	37
3.3.2	Linux3.10 内核配置 .....	39
3.4	HDMI 调试节点说明 .....	40
<b>4</b>	<b>背光控制 .....</b>	<b>41</b>
	<b>附录: .....</b>	<b>42</b>
<b>A1</b>	<b>LVDS 、EDP、MIPI 显示接口转换芯片及其驱动 .....</b>	<b>42</b>
<b>A2</b>	<b>MIPI 屏幕的使用 .....</b>	<b>43</b>
<b>A3</b>	<b>RK32XX MIPI 屏幕的使用 .....</b>	<b>47</b>

---

<b>A3</b>	<b>快速判断 RGB 颜色是否有颠倒 .....</b>	<b>52</b>
<b>A4</b>	<b>利用 LUT 功能改善显示效果 .....</b>	<b>54</b>
<b>A5</b>	<b>支持 BMP 格式的开机 LOGO .....</b>	<b>55</b>
<b>A6</b>	<b>高清屏优化.....</b>	<b>57</b>
<b>A7</b>	<b>基于 RK FB 驱动的应用开发.....</b>	<b>58</b>
<b>A8</b>	<b>Q&amp;A .....</b>	<b>65</b>

## 1 LCD 硬件原理

LCD 是 Liquid Crystal Display 的简称，即液晶显示器，依据驱动方式可分为静态驱动、简单矩阵驱动以及主动矩阵驱动 3 种。其中，简单矩阵型又可再细分扭转向列型（TN）和超扭转式向列型（STN）两种，而主动矩阵型则以薄膜式晶体管型（TFT）为主流。表（1）列出了 TN、STN 和 TFT 显示器的区别。

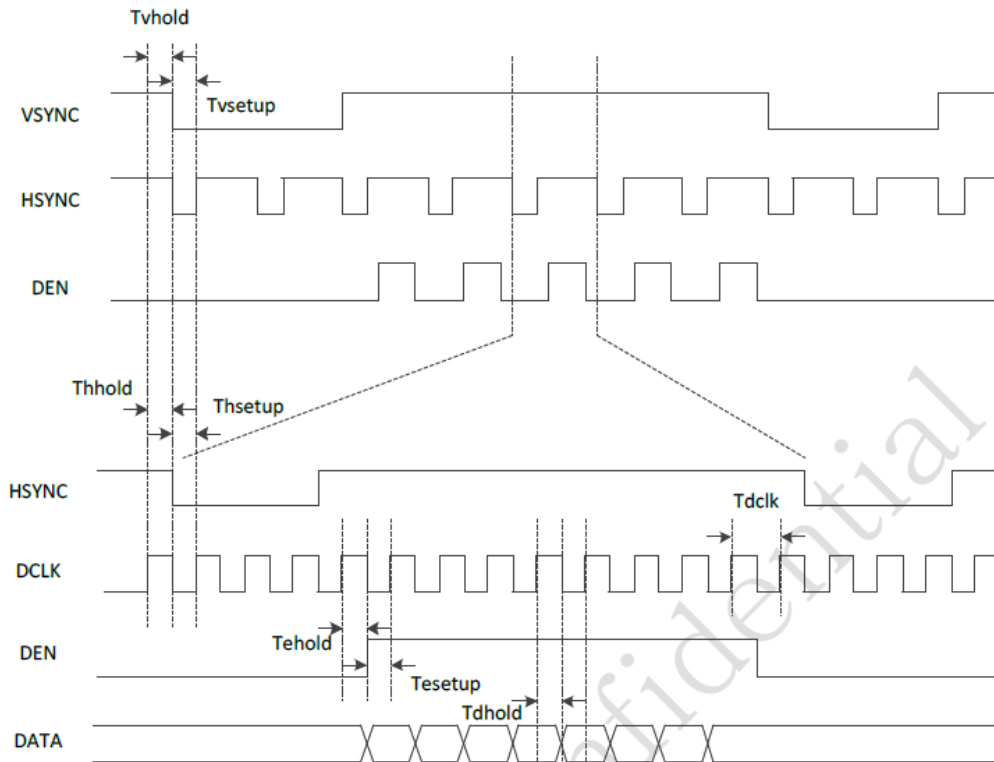
类别	TN	STN	TFT
原理	液晶分子，扭转 90 度	扭转 180~270 度	液晶分子，扭转 90 度
特性	黑白，单色，低对比	黑白，彩色，低对比	彩色，高对比
动画显示	否	否	是
视角	30° 以下	40° 以下	80° 以下
面板尺寸	1~3 英寸	1~12 英寸	37 英寸

表（1）TN、STN、TFT LCD 对比

由表（1）可以看出，TFT LCD 比 TN 和 STN LCD 的显示质量更佳，由于制造工艺的原因，他的对比度高，反应速度快，能显示丰富的色彩。因此在当前的嵌入式市场中，大量使用的就是 TFT LCD。

一块 LCD 屏显示图像不但需要 LCD 驱动器，还需要有相应的 LCD 控制器。通常 LCD 驱动器会以 COF/COG 的形式与 LCD 玻璃基板制作在一起，而 LCD 控制器（LCDC）则由外部电路来实现。许多 MCU 内部直接集成了 LCD 控制器，比如 RK 系列的 CPU，通过 LCD 控制器可以方便地控制 STN 和 TFT 屏。

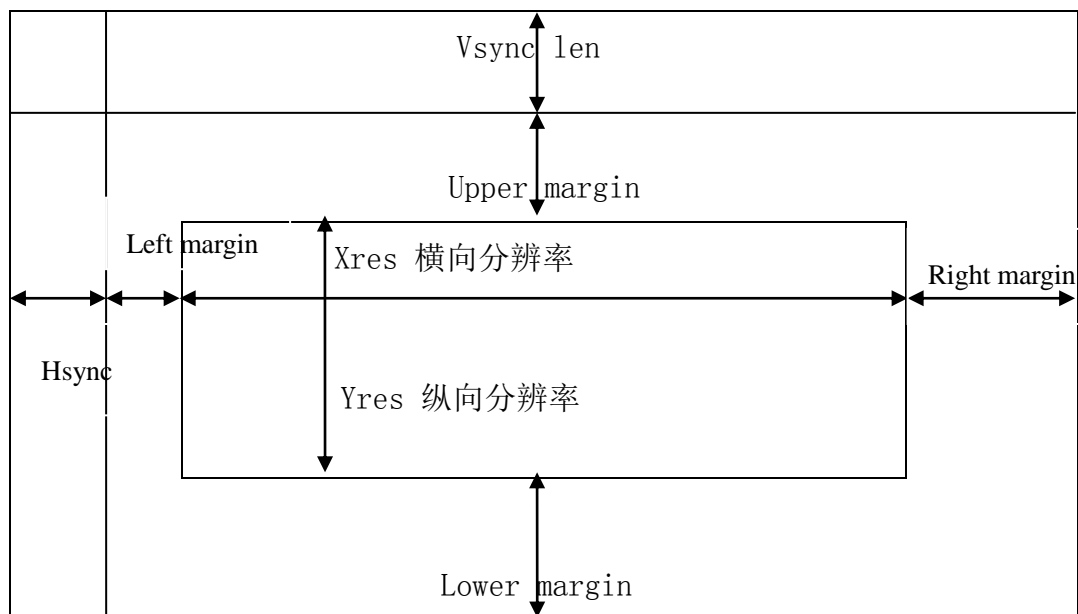
图（1）给出了 TFT 屏的典型时序。时序图中的 VCLK、HSYNC 和 VSYNC 分别为像素时钟信号（用于锁存图像数据的像素时钟）、行同步信号和帧同步信号，VDEN 为数据有效标志信号，VD 为图像的数据信号。



图（1）行场控制的 LCD 时序图

上图的 VSYNC 是帧同步信号，每发出一个脉冲，都意味着新的一屏图像数据开始发送。HSYNC 是行同步信号，每发出一个脉冲都表明新的一行图像资料开始发送。在帧同步以及行同步的头尾都必须留有回扫时间。这样的时序安排起源于 CRT 显示器电子枪偏转所需要的时间，但后来成为实际上的工业标准，因此 TFT 屏也包含了回扫时间，虽然对于 TFT 来说，这是不需要的。

图（2）给出了 LCD 控制器中应该设置的 TFT 屏的时序参数，其中的上边界(upper margin, vertical back porch)和下边界(low margin, vertical front porch)即为帧切换的回扫时间，左边界(left margin, horizontal back porch)和右边界(right margin, horizontal front porch)即为行切换的回扫时间，水平同步(hsync, horizontal pulse width)和垂直同步(vsync, vertical pulse width)分别是行和帧同步本身需要的时间。横向分辨率(xres, Horizontal valid data)和纵向分辨率(yres, vertical valid data)，在我们的 MID 系统上，常用的 LCD 分辨率主要为 800\*480、1024\*600、1024\*768、1280\*800、1920\*1200、2048\*1536 等。



图（2）LCD 中的时序参数

图（3）是一个典型的 LCD 的 timing characteristics:

DCLK	Dot Clock	$1/t_{CLK}$	29	33	38	MHz	
	DCLK pulse duty	$T_{cwh}$	40	50	60	%	
DE	Setup Time	$T_{esu}$	8	-	-	ns	
	Hold time	$T_{ehd}$	8	-	-	ns	
	Horizontal Period	$t_H$	1026	1056	1086	$t_{CLK}$	
	Horizontal Valid	$t_{HA}$		800		$t_{CLK}$	
	Horizontal Blank	$t_{HB}$	226	256	286	$t_{CLK}$	
	Vertical Period	$t_V$	515	525	535	$t_H$	
	Vertical Valid	$t_{VA}$		480		$t_H$	
SYNC	HSYNC Setup Time	$T_{hst}$	8	-	-	ns	
	HSYNC Hold Time	$T_{hhd}$	8	-	-	ns	
	VSYNC Setup Time	$T_{vst}$	8	-	-	ns	
	VSYNC Hold Time	$T_{vhd}$	8	-	-	ns	
	Horizontal Period	$t_H$	1026	1056	1086	$t_{CLK}$	
	Horizontal Pulse Width	$t_{HPW}$	-	30	-	$t_{CLK}$	$t_{HB} + t_{HPW} = 46DCLK$ is fixed
	Horizontal Back Porch	$t_{HB}$	-	16	-	$t_{CLK}$	
	Horizontal Front Porch	$t_{HFP}$	180	210	240	$t_{CLK}$	
	Horizontal Valid	$t_{HD}$		800		$t_{CLK}$	
	Vertical Period	$t_V$	515	525	535	$t_H$	
	Vertical Pulse Width	$t_{VPW}$	-	13	-	$t_H$	$t_{VPW} + t_{VB} = 23t_H$ is fixed
	Vertical Back Porch	$t_{VB}$	-	10	-	$t_H$	
	Vertical Front Porch	$t_{VFP}$	12	22	32	$t_H$	
	Vertical Valid	$t_{VD}$		480		$t_H$	

图（3） timing characteristics

可以看到，此款 LCD 可以用 DE 和 SYNC 两种模式去驱动，我们常用的是 SYNC 模式，从软件上来说，DE 模式和 SYNC 模式是一样的，软件上不做区分。从表中我们可以得到如下参数：

$Left\_margin = HBP$  (Horizontal Back Porch) = 16;

```
Right_margin = HFP (Horizontal Front Porch) = 210;
Hsync        = HPW (Horizontal Pulse Width ) = 30;
Xres         = HVD(Horizontal Valid)          = 800;
Upper_margin = VBP (Vertical Back Porch)      = 10;
low_margin   = VFP (Vertical Front Porch)     = 22;
Vsync        = VPW (Vertical Pulse Width)    = 13;
Yres         = VVD (Vertical Valid)          = 480;
```

而且这些参数满足如下公式:

```
Left_margin + right_margin + hsync + xres = horizontal period
Upper_margin + low_margin + vsync + yres = vertical period
```

这些参数都要写入相应的 LCDC 寄存器里面。

另外有些 DE 模式的屏幕, 并没有直接告诉 HFP、HSYNC、HBP、VFP、VSYNC、VBP 这些参数, 而给的是 Horizontal blank time 和 Vertical blank time, 这时我们只要保证如下关系即可:

```
HFB + HSYNC + HBP = Horizontal blank time
VFP + VSYNC + VBP = vertical blank time
```

在 LCD 驱动中, 还有一个重要的参数——点时钟, 即 dot clock, 在 LCD 的 data sheet 里面一般是 MHz, 名称为 PCLK 或者 DCLK。例如, 如果为 28.37516 MHz, 那么画 1 个像素需要 35242 ps (皮秒):

$$1/(28.37516 \times 10^6 \text{ Hz}) = 35.242 \times 10^{-9} \text{ s}$$

如果屏幕的分辨率是 640×480, 显示一行需要的时间是:

$$640 \times 35.242 \times 10^{-9} \text{ s} = 22.555 \times 10^{-6} \text{ s}$$

每条扫描线是 640, 但是水平回扫和水平同步也需要时间, 假设水平回扫同步需要 272 个像素时钟, 因此, 画一条扫描线完整的时间是:

$$(640+272) \times 35.242 \times 10^{-9} \text{ s} = 32.141 \times 10^{-6} \text{ s}$$

可以计算出水平扫描率大约是 31kHz:

$$1/(32.141 \times 10^{-6} \text{ s}) = 31.113 \times 10^3 \text{ Hz}$$

完整的屏幕有 480 线, 但是垂直回扫和垂直同步也需要时间, 假设垂直回扫和垂直同步需要 49 个像素时钟, 因此, 画一个完整的屏幕的时间是:

$$(480+49) \times 32.141 \times 10^{-6} \text{ s} = 17.002 \times 10^{-3} \text{ s}$$

可以计算出垂直扫描率大约是 59kHz:



$$1/(17.002\text{E-}3 \text{ s}) = 58.815 \text{ Hz}$$

这意味着屏幕数据每秒钟大约刷新 59 次。

由此可以得到如下公式：

$$\text{刷新率} = \text{dotclock} / ((\text{xres} + \text{left\_margin} + \text{right\_margin} + \text{hsync}) * (\text{yres} + \text{upper\_margin} + \text{low\_margin} + \text{vsync}))$$

在 linux frame buffer 子系统中，还有用到一个参数---像素时钟即 pixclock。

$$\text{Pixclock} = 1/\text{dotclock}$$

对于 LCDC 驱动来说，就是要根据屏幕的这些时序参数（DCLK、HSYNC、HBP、HVD、HFP、VSYNC、VBP、VVD、VFP）送出符合要求的信号。

这里需要说明的一点是，Android 的最高刷新频率为 60fps，所以我们最好保证 LCDC 的刷新频率也为 60fps，根据文档第一部分介绍的 LCDC 的刷新频率计算公式可以知道，LCDC 的刷新频率和 DCLK 成正比，和水平方向与垂直方向参数之和的乘积成反比。根据屏幕的 datasheet 我们可以看出，对于一款屏幕 H\_VD/V\_VD 对应屏幕的分辨率，这两个数值是固定的不能修改，BP、FP、PW 的值都有一个最大值和最小值的取值范围，所以当我们的 DCLK 分配不到想要的频率的时候，可以适当的调整 BP、FP、PW，使得 LCDC 的刷新率尽可能的接近 60FPS。

LCDC 的刷新率可以通过启动 log 查看：

```
0.571487] lcdc0:reg_phy_base = 0x1010c000,reg_vir_base:0xf70ac000
0.571562] fb0:win1
0.571567] fb1:win0
0.571572] fb2:win2
0.571673] rk30 lcdc0 clk enable...
0.615176] rk30 lcdc0 clk disable...
0.634054] fb0:phy:90c00000>>vir:f8000000>>len:0xc00000
0.634276] rk_fb_register>>>>fb0
0.642959] rk_fb_register>>>>fb1
0.643122] rk_fb_register>>>>fb2
0.643278] rk30 lcdc0 clk enable...
0.643318] lcdc0: dclk:70588235>>fps:61 rk30_load_screen for lcdc0 ok!
```

Linux3.0.36 内核

```
0.748764] rk-fb rockchip-fb: rk fb ion client create success!
0.748787] rk-fb rockchip-fb: rockchip framebuffer driver probe
0.749203] rk-screen rk_screen.9: rockchip screen probe success
0.749368] rk32-lvds ff96c000.lvds: screen is not lvds/rgb!
0.749391] rk32-lvds: probe of ff96c000.lvds failed with error -22
0.749861] rk32-edp rk32-edp: rk32 edp driver probe success
0.750574] graphics fb0: rockchip framebuffer registerd:fb0
0.750876] graphics fb1: rockchip framebuffer registerd:fb1
0.751027] graphics fb2: rockchip framebuffer registerd:fb2
0.751176] graphics fb3: rockchip framebuffer registerd:fb3
0.771400] rk3288-lcdc lcdc1: lcdc1: dclk:205000000>>fps:60
```

Linux3.10 版本内核

或者通过 fb 的 sys 节点查看：

```
shell@android:/ # cat sys/class/graphics/fb0/fps
fps:61
shell@android:/ #
```

## 2 LCD 驱动框架及实现

在 linux 内核中,显示相关的驱动称为 fb(framebuffer)驱动。在 RK30 以后(RK30XX、RK292X、RK31XX) 的平台上,为了尽可能的复用代码,fb 驱动被分为 fb 框架相关的部分、LCDC 控制器相关的部分、LCD 屏幕相关的部分、LCD 电源操作相关的板级配置部分。

### 2.1 fb 框架相关的代码

```
drivers/video/fbmem.c  
  
drivers/video/rockchip/rk_fb.c  
  
drivers/video/rockchip/rkfb_sysfs.c  
  
include/linux/rk_fb.h
```

这部分代码实现和 fb 相关的框架,不涉及具体的硬件操作,所有的 LCDC (RK3066、 RK302X、RK292X、RK3188、RK3288 等等) 驱动共用。其中 fbmem.c 为 linux 内核原生代码,他向上提供和用户空间交互的接口 (open、read、write、ioctl 等), 向下联系平台相关的 fb 驱动 rk\_fb.c。

注: 从 RK30 以后的 fb 驱动,我们使用新的框架,不再沿用 rk29 fb 的代码,因此相关的外围驱动,比如屏幕配置文件,如果要使用到 rk fb 中相关的数据结构,请直接 #include<linux/rk\_fb.h>,不要再使用 rk29\_fb.h、screen.h 等 rk2918 以前的头文件,否则有可能带来兼容性相关的问题。

### 2.2 LCDC 相关的代码

```
drivers/video/rockchip/lcdc/rkxxx_lcdc.c:  
  
drivers/video/rockchip/lcdc/rk30_lcdc.c  
  
drivers/video/rockchip/lcdc/rk3066b_lcdc.c  
  
drivers/video/rockchip/lcdc/rk2928_lcdc.c  
  
drivers/video/rockchip/lcdc/rk3188_lcdc.c  
  
drivers/video/rockchip/lcdc/rk3288_lcdc.c
```

这部分代码和具体的 LCDC 控制器相关，不同的 LCDC 控制器对应的代码不同。

## 2.3 LCD 屏幕配置相关代码

drivers/video/rockchip/screen/rk\_screen.c

include/linux/rk\_screen.h

drivers/video/rockchip/screen/lcd\_xxx.c

rk\_screen.c 是屏幕配置文件的共用代码，主要实现如下接口：

对于 3.0.36 的内核版本：

set\_scaler\_info: 仅使用主控的一个 LCDC+RK610 或者 RK616 的 scaler 实现双显的配置接口，只有这种双显模式下才会用到（即 CONFIG\_ONE\_LCDC\_DUAL\_OUTPUT\_INF）。

set\_lcd\_info: 屏幕参数配置接口，所有的屏幕都会用到。

get\_fb\_size: 根据屏幕的分辨率计算需要分配多大的 fb 空间，对于三 buffer，计算公式为  $X*Y*4*3$ ，双 buffer 计算公式为  $X*Y*4*2$ ，这里的 X 和 Y 分别代表屏幕的横向分辨率和纵向分辨率，目前我们都使用三 buffer。

建议每个项目相关的人员都要看下 rk\_screen.c 中相关接口的实现，以方便开发中遇到的 debug 问题。

对于 3.10 的内核版本：

通过 probe 函数获取屏幕参数（linux3.10 以后的内核版本，屏幕相关的参数通过 dts

(lcd\_XXX.dtsi) 传递)。

```
/*
 * RockChip. LCD_B101ew05
 *
 */
/ {
    disp_timings: display-timings {
        native-mode = <&timing0>;
        timing0: timing0 {
            screen-type = <SCREEN_LVDS>;
            lvds-format = <LVDS_8BIT_2>;
            out-face = <OUT_D888_P666>;
            clock-frequency = <71000000>;
            hactive = <1280>;
            vactive = <800>;
            hback-porch = <100>;
            hfront-porch = <18>;
            vback-porch = <8>;
            vfront-porch = <6>;
            hsync-len = <10>;
            vsync-len = <2>;
            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, 可取值为 SCREEN\_LVDS、SCREEN\_RGB、SCREEN\_EDP、SCREEN\_MIPI

lvds-format :LVDS 的数据格式, 如果是 LVDS 的屏幕需要设置该格式, 取值为 LVDS\_8BIT\_1、  
LVDS\_8BIT\_2、LVDS\_8BIT\_3.

out-face:屏幕的接线格式

这三个参数的取值都在 include/dt-bindings/rkfb/rk\_fb.h 中定义。

clock-frequency: dclk 的频率, 单位 hz

hactive: 屏幕水平分辨率

vactive: 屏幕纵向分辨率

hsync-len: 屏幕水平同步脉宽

vsync-len: 屏幕纵向同步脉宽

hsync-active、vsync-active、de-active 、pixelclock-active: 分别为 hsync、vsync、DEN、  
dclk 的极性控制。置 1 将对时钟极性进行翻转。

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

我们发布的 SDK 中已经包含了几个比较典型的屏幕配置 dtsi 文件：

```
Lcd-b1010ew05.dtsi      /*1280*800 的 LVDS 屏幕*/
Lcd-LP097QX1.dtsi      /*2048*1536 的 EDP 屏幕*/
Lcd-wqxga-mipi.dtsi     /*mipi 屏幕*/
```

在项目开发中，可以以这几个屏幕作为参考，在板级 dts 文件中 include 对应的屏幕 dtsi 文件。

提供 rk\_fb\_get\_prmry\_screen(struct rk\_screen \*screen)接口给内核其他模块获取屏幕参数。

在 linux3.0.36 版本的内核中

lcd\_XXX.c 是屏幕参数配置文件，而且必须以 lcd\_XXX.c（即以 lcd\_开头）的格式命名。在编译的时候经过 Makefile 的特殊处理，lcd\_XXX.c 中的内容会被拷贝到生成的 lcd.h 文件中，rk\_screen.c 会包含 lcd.h 这个头文件，然后引用对应屏幕配置文件中定义的相关参数去进行初始化。

这部分代码是根据 LCD datasheet 设置的和 LCD 屏幕相关的配置参数，这也是产品端在开发的过程中修改最多的代码，下面对里面涉及的内容详细讲解。以 lcd\_b101ew05.c 为例，该驱动为 RK3066X SDK 板所使用的屏幕所对应的驱动。

有如下宏比较重要：

```
#if defined(CONFIG_RK610_LVDS) || defined(CONFIG_RK616_LVDS)
#define SCREEN_TYPE          SCREEN_LVDS
#else
#define SCREEN_TYPE          SCREEN_RGB
#endif
#define LVDS_FORMAT          LVDS_8BIT_2
#define OUT_FACE              OUT_D888_P666

#define DCLK                  71000000
#define LCDC_ACLK              300000000          //29 lcdc axi DMA 频率

/* Timing */
#define H_PW                  10
```

```
#define H_BP          100
#define H_VD          1280
#define H_FP          18

#define V_PW          2
#define V_BP          8
#define V_VD          800
#define V_FP          6

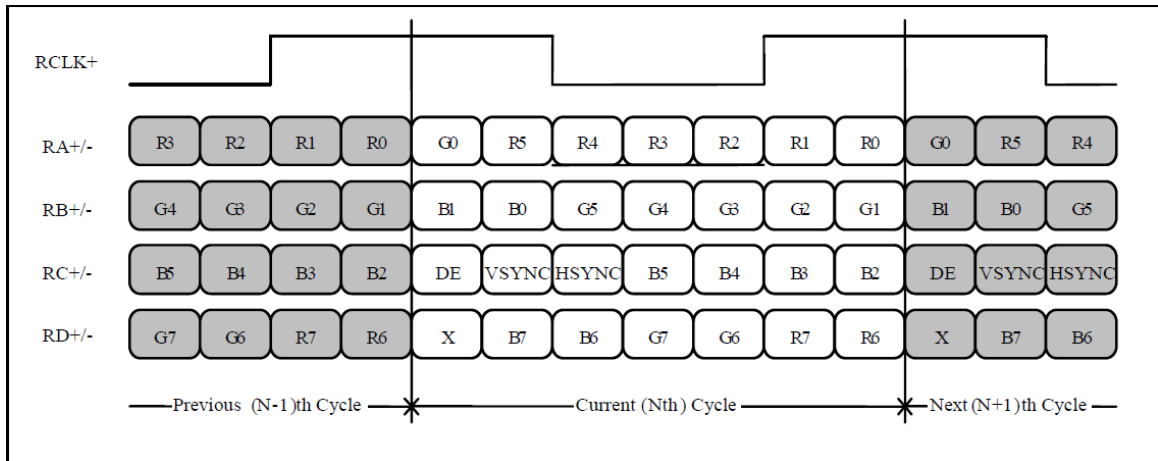
#define LCD_WIDTH      216
#define LCD_HEIGHT     135

#if defined(CONFIG_RK610_LVDS) || defined(CONFIG_RK616_LVDS)
#define DCLK_POL       1
#else
#define DCLK_POL       0
#endif
#define DEN_POL        0
#define VSYNC_POL      0
#define HSYNC_POL      0

#define SWAP_RB        0
#define SWAP_RG        0
#define SWAP_GB        0
```

SCREEN\_TYPE 表示该屏幕类型（RGB、LVDS、MIPI、HDMI、MCU），如果该屏幕为 LVDS 屏，且对应的 LVDS 必须用程序进行驱动（比如 RK610、RK616 或者 rk292x、RK302X 平台内部集成有 lvds 模块），则该宏应定义成 SCREEN\_LVDS。如果是 RGB 屏幕，该宏应定义成 SCREEN\_RGB, MIPI 屏幕则应该定义为 SCREEN\_MIPI。

LVDS\_FORMAT 该宏表示 LVDS 信号的分配方式，请结合屏幕和 LVDS 在硬件上的连接方式定义为 LVDS\_8BIT\_1、LVDS\_8BIT\_2、LVDS\_8BIT\_3、LVDS\_6BIT 中的一种，该宏定义在 rk\_screen.h 中，并在注释中表明了其对应的接线方式，具体 LVDS 屏幕的 datasheet 也会说明信号在 LVDS 各个通道上的分配方式。比如图（4）：



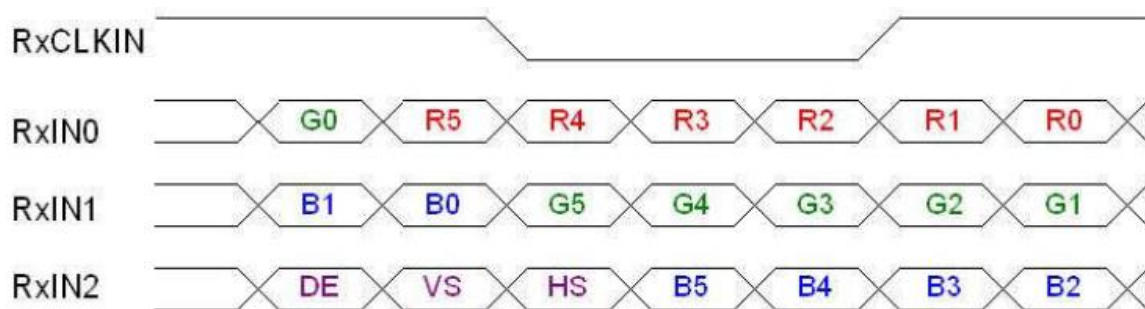
图（4）LVDS data format

上图说明在 Y0 通道上传输的是 R0~R5 和 G0，在 Y1 通道上传输的是 G1~G5 和 B0、B1，在 Y2 通道上传输的是 B2~B5 和 VSYNC、HSYNC 及 DEN，在 Y3 通道上传输的是各种颜色的高两位。参考 rk\_screen.h 中的定义，我们知道，这个格式对应 LVDS\_8BIT\_1。

OUT\_FACE 表示屏幕采用多少位的接线方式，如果是 24bit 的屏幕，则为 OUT\_P888；如果是 18bit 的屏幕，数据线分别接在每种单元色的高六位则定义为 OUT\_D888\_P666、如果数据线接在 LCDC 的低 16 位，则定义为 OUT\_P666。这个如果设置的不正确，比如 24bit 的接法，定义成 OUT\_D888\_P666 或者 OUT\_P666 就有可能出现色阶。屏幕接线的方法可参看《LVDS 应用详细说明.pdf》。

需要注意的是，很多屏幕，都有管脚接高电平还是低电平来控制该屏幕使用 18BIT 还是 24BIT，这个在实际开发的过程中要留意。

另外有一点需要说明，对于 18BIT 的屏幕，这种屏幕的 LVDS data format 一般如图（5）所示：



这种 18bit 的屏幕，一般只用三个 LVDS 差分通道，对应的 LVDS 格式为 LVDS\_6BIT。

但是，如果只使用主控的一个 LCDC+RK610/RK616 的 Scaler 实现双显，必须把 OUT\_FACE 定义为 OUT\_D888\_P666, LVDS\_FORMAT 定义为 LVDS\_8BIT\_2，否则在 HDMI 模式下，屏幕显示会异常。

DCLK 这个是屏幕的点时钟，根据 LCD data sheet 进行设置。另外由于时钟分频的局限性，并不一定能精确的分出所需要的 CLK，有可能你定义的是 65MHZ，实际分到的却是 63MHZ，因为时钟大部分都是整数分频，所以要想分到尽可能接近的频率，请根据实际得到值去调整 DCLK 设置的值。LCD 相关 clock 可以通过如下命令读取 `cat proc/clocks | busybox grep -r lcdc`

```
shell@android:/ # cat proc/clocks | busybox grep -r lcdc
pd_lcdcl 0 Hz usecount = 0 parent = pd_vio
pd_lcdco 0 Hz usecount = 1 parent = pd_vio
dclk_lcdcl off 74.250000 MHz usecount = 0 parent = general_pll
          hclk_lcdcl off 148.500000 MHz usecount = 0 parent = hclk_cpu
          hclk_lcdco on 148.500000 MHz usecount = 1 parent = hclk_cpu
dclk_lcdco on 70.588235 MHz usecount = 1 parent = codec_pll
acclk_lcdcl_pre on 300 MHz usecount = 1 parent = codec_pll
acclk_vio1 on 300 MHz usecount = 1 parent = acclk_lcdcl_pre
acclk_lcdcl off 300 MHz usecount = 0 parent = acclk_vio1
acclk_lcdco_pre on 300 MHz usecount = 1 parent = codec_pll
acclk_vio0 on 300 MHz usecount = 1 parent = acclk_lcdco_pre
acclk_lcdco on 300 MHz usecount = 1 parent = acclk_vio0
```

或者用示波器测量。

LCDC\_ACLK 为 LCDC 的 AXI 总线时钟，是 LCDC 通过 DMA 搬运 fb 中数据的时钟，这个在 RK30XX 以后的平台上，由 clock 管理代码统一锁定，外部驱动无法设置，所以这个宏定义多少都没关系。留在这里，只是为了和 RK29 以前的代码兼容。

H\_PW、H\_FP、H\_BP、H\_VD、V\_PW、V\_FP、V\_BP、V\_VD 请参考前面的描述，根据 lcd datasheet 设置。

LCD\_WIDTH、LCD\_HIGHT 为屏幕的长和宽，单位为 mm，根据 lcd datasheet 设置。

DCLK\_POL 是 dclk 时钟的极性，这个默认定义为 0，但是有些屏幕上显示像素点会有抖动的感觉，这时定义为 1，做一下翻转效果或许会好。

在 set\_lcd\_info 函数中，有如下代码：

```
screen->pin_vsync = VSYNC_POL;
screen->pin_den = DEN_POL;
screen->pin_dclk = DCLK_POL;
```

是设置 vsync、hsync、dclk、den 信号极性的，如果 lcd 的 datasheet 有要求，可以做对应的翻转即置 1；大部分的 lcd 都是不用翻转的。

SWAP\_RB、SWAP\_RG、SWAP\_GB 这个是对 RGB 信号进行交换，对于有些项目，可能由于接线的原因或者屏幕自身的特性，从 LCDC 输出的数据在屏幕上显示颜色是反的（R G B 三原色颠倒，比如红色变成了绿色 绿色变成了蓝色），然后把对应的宏置 1，可以对颜色进行纠正。至于如何快速判断是哪种颜色反了，请参考附录《快速判断 LCD 屏幕颜色显示是否正常》。



如果有使用 RK610、RK616，并且主控只用一个 LCDC+RK610 或者 RK616 的 scaler 实现双显，需要在屏幕的驱动里面加入 scaler 参数配置。具体请参考《RK610 配置\_V1.2.pdf》和《Rockchip\_RK61X 开发文档》。

如果在屏幕里面有用到 Board ID 的机制，需要在屏幕配置文件里面定义 RK\_USE\_SCREEN\_ID 的宏，并实现 set\_lcd\_info\_by\_id 接口。

如果有的屏幕需要初始化，需要在屏幕配置文件里面定义 RK\_SCREEN\_INIT 宏，然后实现 rk\_lcd\_init 和 rk\_lcd\_standby 接口。

如果有的屏幕要用 LUT 对显示效果进行调整，需要在屏幕文件中定义 USE\_RK\_DSP\_LUT 宏，并定义默认的 dsp\_lut 数组。

SDK 中有部分屏幕配置文件是我们 sdk 或者开发样机上经常使用的，可供参考：

- (1) drivers/video/rockchip/screen/lcd\_b101ew05.c
- (2) drivers/video/rockchip/screen/lcd\_LP097QX1.c
- (3) drivers/video/rockchip/screen/lcd\_hdmi\_800x480.c
- (4) drivers/video/rockchip/screen/lcd\_hdmi\_1024x600.c
- (5) drivers/video/rockchip/screen/lcd\_hdmi\_1024x768.c
- (6) drivers/video/rockchip/screen/lcd\_hdmi\_1280x800.c
- (7) drivers/video/rockchip/screen/lcd\_hsd100pxn.c

(1) 为 rk30xx、rk31xx sdk 板上使用的 LVDS 屏幕，分辨率为 1280\*800，支持单 LCDC + RK610 的双显方案。

(2) 为分辨率为 2048x1536 的高清屏幕，接口为 EDP。

(3)、(4)、(5)、(6) 是几种常用分辨率的屏幕在单 LCDC+RK610 双显方案中的配置文件。

(7)、为 rk29xx SDK 板上使用的 LVDS 接口的屏幕，支持 RK2918 和 RK2928 的 LVDS 接口。

对于 Linux3.10 以后版本的内核，屏幕参数通过 dts 文件(lcd\_XXX.dtsi)传递。screen 目录下只保留 lcd\_general.c 和 lcd\_mipi.c 这两个屏幕配置相关的文件。其中 lcd\_general.c 保留做后续扩展，如果某些屏幕需要特殊的控制，可以在这个文件

中实现相关控制接口，然后注册到 rk\_screen.c 中，类似 linux3.0.36 中的处理机制。lcd\_mipi.c 在使用 mipi 屏幕的时候必须选上。

## 2.4 LCD 电源相关操作的配置以及配置

### 2.4.1 基于 Linux3.0.36 内核的板级配置

由于不同的项目在 LCD 电源控制上有不同的设计，而且不同的 LCD 电源控制的时序不同，另外，为了硬件上更为灵活的设计，对于有两个 LCDC 的平台（比如 RK3066、RK3168、RK3188），LCD 屏幕和 HDMI 可以根据硬件设计的需要，灵活的选择接在 LCDC0 或者 LCDC1 上，因此这部分代码放在 board 文件里面实现，具体的接口为：

```
#ifdef CONFIG_FB_ROCKCHIP

#define LCD_CS_PIN        INVALID_GPIO
#define LCD_CS_VALUE      GPIO_HIGH

#define LCD_EN_PIN        RK30_PIN0_PB0
#define LCD_EN_VALUE      GPIO_HIGH

static int rk_fb_io_init(struct rk29_fb_setting_info *fb_setting)
{
    int ret = 0;

    if(LCD_CS_PIN !=INVALID_GPIO)
    {
        ret = gpio_request(LCD_CS_PIN, NULL);
        if (ret != 0)
        {
            gpio_free(LCD_CS_PIN);
            printk(KERN_ERR "request lcd cs pin fail!\n");
            return -1;
        }
    }
    else
    {
        gpio_direction_output(LCD_CS_PIN, LCD_CS_VALUE);
    }
}
```

```

    }

    if(LCD_EN_PIN !=INVALID_GPIO)
    {
        ret = gpio_request(LCD_EN_PIN, NULL);
        if (ret != 0)
        {
            gpio_free(LCD_EN_PIN);
            printk(KERN_ERR "request lcd en pin fail!\n");
            return -1;
        }
        else
        {
            gpio_direction_output(LCD_EN_PIN, LCD_EN_VALUE);
        }
    }
    return 0;
}

static int rk_fb_io_disable(void)
{
    if(LCD_CS_PIN !=INVALID_GPIO)
    {
        gpio_set_value(LCD_CS_PIN, !LCD_CS_VALUE);
    }
    if(LCD_EN_PIN !=INVALID_GPIO)
    {
        gpio_set_value(LCD_EN_PIN, !LCD_EN_VALUE);
    }
    return 0;
}

static int rk_fb_io_enable(void)
{
    if(LCD_CS_PIN !=INVALID_GPIO)
    {
        gpio_set_value(LCD_CS_PIN, LCD_CS_VALUE);
    }
    if(LCD_EN_PIN !=INVALID_GPIO)
    {
        gpio_set_value(LCD_EN_PIN, LCD_EN_VALUE);
    }
}

```

```

    return 0;
}

#if defined(CONFIG_LCDC0_RK3066B) || defined(CONFIG_LCDC0_RK3188)
struct rk29fb_info lcdc0_screen_info = {
#if defined(CONFIG_RK_HDMI) && defined(CONFIG_HDMI_SOURCE_LCDC0) &&
defined(CONFIG_DUAL_LCDC_DUAL_DISP_IN_KERNEL)
    .prop      = EXTEND,      //extend display device
    .io_init   = NULL,
    .io_disable = NULL,
    .io_enable = NULL,
    .set_screen_info = hdmi_init_lcdc,
#else
    .prop      = PRMRY,      //primary display device
    .io_init   = rk_fb_io_init,
    .io_disable = rk_fb_io_disable,
    .io_enable = rk_fb_io_enable,
    .set_screen_info = set_lcd_info,
#endif
};
#endif

#if defined(CONFIG_LCDC1_RK3066B) || defined(CONFIG_LCDC1_RK3188)
struct rk29fb_info lcdc1_screen_info = {
#if defined(CONFIG_RK_HDMI) && defined(CONFIG_HDMI_SOURCE_LCDC1) &&
defined(CONFIG_DUAL_LCDC_DUAL_DISP_IN_KERNEL)
    .prop      = EXTEND,      //extend display device
    .io_init   = NULL,
    .io_disable = NULL,
    .io_enable = NULL,
    .set_screen_info = hdmi_init_lcdc,
#else
    .prop      = PRMRY,      //primary display device
    .io_init   = rk_fb_io_init,
    .io_disable = rk_fb_io_disable,
    .io_enable = rk_fb_io_enable,
    .set_screen_info = set_lcd_info,
#endif
};

```

```
#endif

static struct resource resource_fb[] = {
    [0] = {
        .name = "fb0 buf",
        .start = 0,
        .end = 0, //RK30_FB0_MEM_SIZE - 1,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .name = "ipp buf", //for rotate
        .start = 0,
        .end = 0, //RK30_FB0_MEM_SIZE - 1,
        .flags = IORESOURCE_MEM,
    },
    [2] = {
        .name = "fb2 buf",
        .start = 0,
        .end = 0, //RK30_FB0_MEM_SIZE - 1,
        .flags = IORESOURCE_MEM,
    },
};

static struct platform_device device_fb = {
    .name = "rk-fb",
    .id = -1,
    .num_resources = ARRAY_SIZE(resource_fb),
    .resource = resource_fb,
};

#endif

#if defined(CONFIG_LCDC0_RK3066B) || defined(CONFIG_LCDC0_RK3188)
static struct resource resource_lcdc0[] = {
    [0] = {
        .name = "lcdc0 reg",
        .start = RK30_LCDC0_PHYS,
        .end = RK30_LCDC0_PHYS + RK30_LCDC0_SIZE - 1,
        .flags = IORESOURCE_MEM,
    },
};
#endif
```

```

    },

    [1] = {
        .name = "lcdc0 irq",
        .start = IRQ_LCDC0,
        .end = IRQ_LCDC0,
        .flags = IORESOURCE_IRQ,
    },
};

static struct platform_device device_lcdc0 = {
    .name = "rk30-lcdc",
    .id = 0,
    .num_resources = ARRAY_SIZE(resource_lcdc0),
    .resource = resource_lcdc0,
    .dev = {
        .platform_data = &lcdc0_screen_info,
    },
};

#endif

#if defined(CONFIG_LCDC1_RK3066B) || defined(CONFIG_LCDC1_RK3188)
static struct resource resource_lcdc1[] = {
    [0] = {
        .name = "lcdc1 reg",
        .start = RK30_LCDC1_PHYS,
        .end = RK30_LCDC1_PHYS + RK30_LCDC1_SIZE - 1,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .name = "lcdc1 irq",
        .start = IRQ_LCDC1,
        .end = IRQ_LCDC1,
        .flags = IORESOURCE_IRQ,
    },
};

static struct platform_device device_lcdc1 = {
    .name = "rk30-lcdc",
    .id = 1,
    .num_resources = ARRAY_SIZE(resource_lcdc1),

```

```
.resource      = resource_lcdcl,
.dev           = {
    .platform_data = &lcdcl_screen_info,
},
};
#endif
```

这里是针对 rk30xx sdk 板实现的电源控制接口。rk\_fb\_io\_init 为相关电源控制 IO 的初始化函数，在 fb lcdc 系统初始化的时候调用：

```
static int __devinit rk3188_lcdc_probe(struct platform_device *pdev)
{
    struct rk3188_lcdc_device *lcdc_dev = NULL;
    struct device *dev = &pdev->dev;
    rk_screen *screen;

    .....
    if(screen_ctr_info->set_screen_info)
    {
screen_ctr_info->set_screen_info(screen, screen_ctr_info->lcd_info)
        if(screen_ctr_info->io_init)
            screen_ctr_info->io_init(NULL); //rk_fb_io_init
    }

    rk_fb_register(&(lcdc_dev->driver), &lcdc_driver, lcdc_dev->id);

}

    printk("rk3188 lcdc%d probe ok!\n", lcdc_dev->id);
```

rk\_fb\_io\_disable 和 rk\_fb\_io\_enable 为电源的关闭和打开接口，在系统进入 suspend、resume 和 shutdown 的时候调用。

```
static int rk3188_lcdc_early_suspend(struct rk_lcdc_device_driver *dev_drv)
{
    struct rk3188_lcdc_device *lcdev =
        container_of(dev_drv, struct rk3188_lcdc_device, driver);

    if(dev_drv->screen0->standby)
        dev_drv->screen0->standby(1);
    if(dev_drv->screen_ctr_info->io_disable)
        dev_drv->screen_ctr_info->io_disable();

    spin_lock(&lcdev->reg_lock);
    if(likely(lcdev->clk_on))
    {
        lcdev_msk_reg(lcdev, INT_STATUS, m_FS_INT_CLEAR, v_FS_INT_CLEAR(1));
        lcdev_msk_reg(lcdev, DSP_CTRL1, m_DSP_OUT_ZERO, v_DSP_OUT_ZERO(1));
        lcdev_msk_reg(lcdev, SYS_CTRL, m_LCDC_STANDBY, v_LCDC_STANDBY(1));
        lcdev_cfg_done(lcdev);
        spin_unlock(&lcdev->reg_lock);
    }
    else //clk already disabled
    {
        spin_unlock(&lcdev->reg_lock);
        return 0;
    }
    rk3188_lcdc_clk_disable(lcdev);
}

#ifdef CONFIG_ARCH_RK3026
if(dev_drv->screen0->type != SCREEN_LVDS){
    int gpio_dclk = iomux_mode_to_gpio(LCDC0_DCLK);
    int ret = gpio_request(gpio_dclk, NULL);
    if(unlikely(ret < 0)){
        printk("Failed to request gpio:lcdev dclk\n");
        return ret;
    }
    gpio_direction_output(gpio_dclk, GPIO_LOW);
}
#endif
return 0;
}
```

上面是 RK3188 的 LCDC early suspend 代码，在系统进入一级休眠的时候会被调用。这段代码分位三部分：第一部分，调用 rk\_fb\_io\_disable 关闭显示相关的电源，第二部分关闭 LCDC 数据输出，第三部分，关闭 LCDC 相关时钟输出，比如 DCLK。

```
static int rk3188_lcdc_early_resume(struct rk_lcdc_device_driver *dev_drv)
{
    struct rk3188_lcdc_device *lcdev =
        container_of(dev_drv, struct rk3188_lcdc_device, driver);
    int i=0;
    int __iomem *c;
    int v;
#ifdef CONFIG_ARCH_RK3026
    if(dev_drv->screen0->type != SCREEN_LVDS){
        int gpio_dclk = iomux_mode_to_gpio(LCDC0_DCLK);
        gpio_free(gpio_dclk);
        iomux_set(LCDC0_DCLK);
    }
#endif
    if(dev_drv->screen_ctr_info->io_enable)
        dev_drv->screen_ctr_info->io_enable();

    if(!lcdev->clk_on)
    {
        rk3188_lcdc_clk_enable(lcdev);
    }
    rk3188_lcdc_reg_resume(lcdev); //resume reg
    spin_lock(&lcdev->reg_lock);
    if(lcdev->atv_layer_cnt)
    {
        lcdev_msk_reg(lcdev, DSP_CTRL1, m_DSP_OUT_ZERO, v_DSP_OUT_ZERO(0));
        lcdev_msk_reg(lcdev, SYS_CTRL, m_LCDC_STANDBY, v_LCDC_STANDBY(0));
        lcdev_cfg_done(lcdev);
    }
    spin_unlock(&lcdev->reg_lock);
    if(!lcdev->atv_layer_cnt)
        rk3188_lcdc_clk_disable(lcdev);
    if(dev_drv->screen0->standby)
        dev_drv->screen0->standby(0); //screen wake up

    return 0;
}
```

上面是 RK3188 LCDC 的 late resume 函数，在系统唤醒的时候被调用，这段代码的功能也大致分位三个部分：第一部分，调用 rk\_fb\_io\_enable 开启显示相关电源，第二部分，开启显示相关的时钟输出，比如 DCLK，第三部分，LCDC 开始输出数据。



```
static void rk3188_lcdc_shutdown(struct platform_device *pdev)
{
    struct rk3188_lcdc_device *lcdc_dev = platform_get_drvdata(pdev);
    if(lcdc_dev->driver.cur_screen->standby) //standby the screen if necessary
        lcdc_dev->driver.cur_screen->standby(1);
    if(lcdc_dev->driver.screen_ctr_info->io_disable) //power off the screen if necessary
        lcdc_dev->driver.screen_ctr_info->io_disable();
    rk3188_lcdc_deint(lcdc_dev);
    //rk_fb_unregister(&(lcdc_dev->driver));
}
```

上面是 RK3188 LCDC 的 shutdown 函数，该函数在系统关机的时候被调用，如果要在关机的时候对显示相关的电源进行处理，可以在这个函数里面进行。

这些接口在其他的平台上（RK30XX、RK292X 等）的实现方法类似。这些函数要针对具体的项目进行微调，包括这三部分的操作顺序，以及每部分之间是否要加延时，都和具体的硬件设计以及屏幕相关。

lcdc0\_screen\_info 和 lcdc1\_screen\_info 分别为 LCDC0 和 LCDC1 上连接的显示设备对应的控制接口。这个要根据 LCD、HDMI 和 LCDC 具体的连接情况去设置。其中 prop 为 LCDC 的属性，如果某个 LCDC 上接的是 LCD 屏幕，则应该将该 LCDC 对应的 prop 设置为 PRMRY，意为主显示设备，并且将 LCD 相关的电源控制接口挂到该 lcdc 的 lcdcx\_screen\_info 上（x =0,1）；如果某个 LCDC 上接的是 HDMI，则将该 LCDC 对应的 prop 设置为 EXTEND, 意为扩展显示设备，并将 HDMI 的初始化信息挂在该 LCDC 的 lcdcx\_screen\_info 上(x=0,1)主要是 hdmi\_init\_lcdc。如果仅使用一个 LCDC 和 RK61X 来实现双显，则该 LCDC 对应的 prop 应该设置为 PRMRY, 该 LCDC 对应的 lcdcx\_screen\_info 上（x=0,1）应该挂接 LCD 电源相关的控制接口。

需要注意的是 RK3028 上，RK3028 是基于 RK3168 和 RK610 实现的 MCP 封装，并且和 RK2928Pin to pin 兼容，在该平台上，LCDC 和 RK610 的连接关系如下：主控的 LCDC0 和 RK610 的 LCD0 连接，RK610 的 LVDS 输出接口和主控的 LCDC1 输出口并联，因而当我们接 LVDS 屏幕的时候，显示信号由主控的 LCDC0 输出，经过 RK610 的 LCD0 口到 LVDS，主控的 LCDC1 不工作，如果要实现双显，只能通过 RK610 的 scaler 模块实现，此时应设置 Lcdc0\_screen\_info 的 prop 为 PRMRY；如果接 RGB 屏幕，测 RGB LCD 屏幕上的显示信号由主控的 LCDC1 输出，直接送给 RGB 屏幕，可以用双 LCDC 实现双显，主控的 LCDC0 送出 HDMI 信号，经过 RK610 的 LCD0 到 HDMI。此时，lcdc0\_screen\_info 的 prop 应该设置为 EXTEND，lcdc1\_screen\_info 的 prop 应该设置为 PRMRY，作为主显示设备。

device\_fb、device\_lcdc0、device\_lcdc1 为 fb，lcdc0，lcdc1 对应的平台设备结构，这三个结构不可随意改动。

为了实现 LCD、HDMI、LCDC 灵活的连接配置，在 board 里面实现了专门的函数 rk\_platform\_add\_display\_devices() 去注册 fb、lcdc、backlight 这些设备。以保证这些设备按照 rk fb 驱动、主显示设备驱动（PRMRY）、扩展显示设备驱动（EXTEND）、背光驱动的顺序加载。

```
static int rk_platform_add_display_devices(void)
{
    struct platform_device *fb = NULL; //fb
    struct platform_device *lcdc0 = NULL; //lcdc0
    struct platform_device *lcdc1 = NULL; //lcdc1
    struct platform_device *bl = NULL; //backlight
#ifdef CONFIG_FB_ROCKCHIP
    fb = &device_fb;
#endif

    #if defined(CONFIG_LCDC0_RK3066B) || defined(CONFIG_LCDC0_RK3188)
        lcdc0 = &device_lcdc0,
    #endif

    #if defined(CONFIG_LCDC1_RK3066B) || defined(CONFIG_LCDC1_RK3188)
        lcdc1 = &device_lcdc1,
    #endif

    #ifdef CONFIG_BACKLIGHT_RK29_BL
        bl = &rk29_device_backlight,
    #endif

    __rk_platform_add_display_devices(fb, lcdc0, lcdc1, bl);

    return 0;
}
```

该函数在 .init\_machine 接口中调用：

```
static void __init machine_rk30_board_init(void)
{
    //avs_init();
    gpio_request(POWER_ON_PIN, "poweronpin");
    gpio_direction_output(POWER_ON_PIN, GPIO_HIGH);

    pm_power_off = rk30_pm_power_off;
```

```

gpio_direction_output(POWER_ON_PIN, GPIO_HIGH);

rk30_i2c_register_board_info();
spi_register_board_info(board_spi_devices, ARRAY_SIZE(board_spi_devices));
platform_add_devices(devices, ARRAY_SIZE(devices));
rk_platform_add_display_devices();
board_usb_detect_init(RK30_PIN0_PA7);

#ifdef CONFIG_WIFI_CONTROL_FUNC
    rk29sdk_wifi_bt_gpio_control_init();
#endif
}
fb 设备必须预先根据屏幕的分辨率为其分配在物理上连续的内存,该操作在.reserve 接口中实现:
static void __init rk30_reserve(void)
{
#ifdef CONFIG_ION
    rk30_ion_pdata.heaps[0].base = board_mem_reserve_add("ion", ION_RESERVE_SIZE);
#endif
#ifdef CONFIG_FB_ROCKCHIP
    resource_fb[0].start = board_mem_reserve_add("fb0 buf", get_fb_size());
    resource_fb[0].end = resource_fb[0].start + get_fb_size() - 1;
    #if defined(CONFIG_FB_ROTATE) || !defined(CONFIG_THREE_FB_BUFFER)
        resource_fb[2].start = board_mem_reserve_add("fb2 buf", get_fb_size());
        resource_fb[2].end = resource_fb[2].start + get_fb_size() - 1;
    #endif
#endif
#endif

#ifdef CONFIG_VIDEO_RK29
    rk30_camera_request_reserve_mem();
#endif

#ifdef CONFIG_GPS_RK
    //it must be more than 8MB
    rk_gps_info.u32MemoryPhyAddr = board_mem_reserve_add("gps", SZ_8M);
#endif
    board_mem_reserved();
}

```

分配内存的大小是根据 get\_fb\_size() 接口获取的,该接口在 rk\_screen.c 中实现:

```
size_t get_fb_size(void)
{
    size_t size = 0;
    #if defined(CONFIG_THREE_FB_BUFFER)
        size = ((H_VD)*(V_VD)<<2)* 3; //three buffer
    #else
        size = ((H_VD)*(V_VD)<<2)<<1; //two buffer
    #endif
    return ALIGN(size,SZ_1M);
}
```

## 2.4.2 基于 Linux 3.10 内核的 dts 配置

对于 Linux3.10 以后版本的内核，不再使用 board 文件，硬件相关的配置参数通过 dts 传递。

为了实现电源相关的控制，在 dts 的 lcdc 节点下，实现了 power\_ctr 控制节点：

```
/*lcd1 as PRMRY(LCD),lcd0 as EXTEND(HDMI)*/
&lcd1 {
    status = "okay";
    rockchip,mirror = <NO_MIRROR>;
    rockchip,cabc_mode = <0>;
    power_ctr: power_ctr {
        rockchip,debug = <0>;
        lcd_en: lcd_en {
            rockchip,power_type = <GPIO>;
            gpios = <&gpio7 GPIO_A3 GPIO_ACTIVE_HIGH>;
            rockchip,delay = <10>;
        };
        lcd_cs: lcd_cs {
            rockchip,power_type = <GPIO>;
            gpios = <&gpio7 GPIO_A4 GPIO_ACTIVE_HIGH>;
            rockchip,delay = <10>;
        };
        /*lcd_rst: lcd_rst {
            rockchip,power_type = <GPIO>;
            gpios = <&gpio3 GPIO_D6 GPIO_ACTIVE_HIGH>;
            rockchip,delay = <5>;
        };*/
    };
};
```

“power\_ctr”、“rockchip, debug”、“rockchip,power\_type”、“gpios”、“rockchip, delay”是系统使用的关键字，不能改动。

“rockchip, debug”节点，如果设置为 1，将打开 fb 中和电源解析相关的 debug 功能。

“rockchip, power\_type”节点，设置电源的控制方式，如果是用 GPIO 控制的，则设置为 GPIO，如果是 PMU 的 regulator 控制的，则设置为 REGULATOR。

“rockchip, delay”设置开启该电源后，需要延时的时间，单位为毫秒（ms）。

lcd\_en、lcd\_cs、lcd\_rst 这些节点的名字可以按照功能的需要去修改。内核在解析到这些电源控制节点后，会以他们在 dts 里面出现的顺序依次去操作他们。所以这些节点，请按照上电顺序要求编写。

Fb 代码中，电源控制相关的接口如下：

```
int rk_disp_pwr_ctr_parse_dt(struct rk_lcdc_driver *dev_drv) //解析 dts
```

```
int rk_disp_pwr_enable/disable(struct rk_lcdc_driver *dev_drv) //电源的开关接口，fb 代码直接
```

调用该接口去开关显示相关的电源。如果某个项目有特殊的电源控制要求，而 dts 又比较难以描述，可以直接修改这两个接口以达到项目要求。

### 2.4.3 make menuconfig 相关配置

```
Device Drivers --->
  Graphics support --->
    <*>Frame buffer support for Rockchip --->
      [ ] Mirroring Support
        Dual display poly select(implement dual display in kernel with dual lcdc)
      [ ] FB rotate support (NEW)
      [*] Three fb buffer support
      [*] rk3188 lcdc support
      [*] lcdc0 support
      [ ] lcdc0 1.8V io support
      [*] lcdc1 support
      [ ] lcdc1 1.8V io support
        LCD Panel Select (Display Port screen LP097QX1) --->
      [ ] RockChip display transmitter support --->
```

#### a. [ ] Mirroring support

该配置项和 wimmo 功能相关，一般项目不用，除非你知道它是干什么的以及你要达到什么目的，否则不用配置。

#### b. Dual display poly select ( )

该配置项为双显策略选择，RK30XX 以后的 SOC 方案，双显有多种实现方案。首先 RK30XX、RK3168、RK3188、RK302X 有两个 LCDC，RK292X 只有一个 LCDC。

(1) 双 lcdc 实现双显，例如 rk3066，用两个 LCDC 实现双显

(2) 一个 lcdc + rk610 实现双显，例如 rk3066 LCDC0 + rk610

(3) 单边显示，即切换到 HDMI 模式下，pad 端屏幕黑掉，例如 rk2926

至于使用哪种方案，根据硬件设计决定，

Dual display poly select (implement dual display in kernel with dual lcdc) --->

( ) implement dual display in kernel with dual lcdc

该配置使用两个 lcdc 实现双显

( ) one lcdc dual output display interface support

该配置使用一个 lcdc + rk61X 实现双显

( ) NO dual display needed

该配置只支持单边显示，例如 rk2926、RK3026

需要强调的一点是，使用一个 LCDC+RK61X 来实现双显，虽然能给硬件走线带来方便，但是这种双显存在很多局限性，一般不建议大家使用：

(1) 由于 RK61X 的 Scaler 不支持旋转功能，所以如果屏幕是竖屏，或者屏幕的模具和硬件做反了，需要进行 180 度的旋转，就不能使用一个 LCDC+RK61X 的 Scaler 实现双显，这种情况只能用双 LCDC 来实现双显。

(2) 如果项目上要用到 MIPI 屏幕，由于 MIPI 屏幕对时序非常敏感，在系统运行的过程中送给 MIPI 相关的时序不能变化，否则会显示异常，而我们在使用一个 LCDC+RK61X 的 scaler 功能实现双显的时候，接上 HDMI 的瞬间，要对 LCD 屏幕做时序切换，所以使用 MIPI 的屏幕，如果要接 HDMI 实现双显，必须使用双 LCDC 模式。

(3) 这种使用一个 LCDC+RK61X 的 scaler 实现的双显效果，在接上 HDMI 的时候，需要对 LCD 屏幕的时序做调整，有些对时序要求比较严格的 LCD 屏幕，可能显示效果无法保证。

另外，如果只使用到一个 lcdc，请在 make menuconfig 只选择对应的 lcdc 即可，不要配上另外一个没有使用的 lcdc，以节省功耗，例如使用 lcdc0 + rk610 进行双显，则应该如下配置

```
<*> rk3188 lcdc support
[*]   lcdc0 support
[ ]   lcdc0 1.8V io support
[ ]   lcdc1 support
[ ]   lcdc1 1.8V io support
```

RK3188、RK3168 的 LCDC 支持 1.8V 的 IO 输出，如果在硬件设计上使用的是 1.8V 的 IO，需要选上对应的 1.8V IO support。

如果是 linux3.10 以后的内核，该属性在 dts 中设置：rockchip,pwr18 = <1>;该选项被置 1 表示硬件采用的是 1.8V 电压设计。

c. [ ] Three fb buffer support

Android4.1 以后，我们使用 3 buffer 以提高显示流畅度，所以对应 Android4.1 以后的项目，这个必须选择。

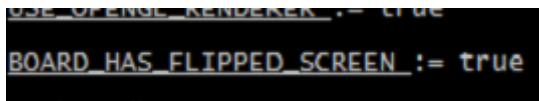
#### d. [ ] FB rotate support

这个是为了部分模具和硬件设计差错，导致 LCD 屏幕显示方向反掉的项目提供的接口，只需要对屏幕显示方向进行调整的时候才用到，否则不要打开。使用方法如下：

如果某个项目，因为模具和硬件差错导致屏幕方向不对（错了 180 度或者 90 度），而硬件又无法纠正（一般的 lcd 屏幕有两个 pin U/D L/R 可以调整方向），可以通过软件进行旋转，先修改 build.prop 的 ro.sf.hwrotation，把屏幕的显示方向纠正过来。这时候如果连接上 HDMI，HDMI 的方向也会由于 ro.sf.hwrotation 的改变而反掉，然后在 kernel 的 make menuconfig 里面选择 FB rotate support 选项，并且在 rotate orientation 里面填写 ro.sf.hwrotation 的值。注意，通过这样方法旋转会增加内存的开销，同时降低显示的流畅度，所以，尽可能的通过硬件调整屏幕的显示方向。同时，该方案只支持 1280\*800 及其以下分辨率的屏幕，而且该方法只适用于 rk30xx 等用两个 lcdc 进行双显的方案。

对于 linux3.10 版本的内核，该选项在 dts 中设置：rockchip,mirror = <X\_MIRROR>;根据实际需求，设置 mirror 属性为 X\_MIRROR/Y\_MIRROR/X\_Y\_MIRROR。

这样修改之后，在 recovery 界面，LCD 屏幕上还是反的，因为在 recovery 中 ro.sf.hwrotation 这个宏不起作用。为了将 recovery 界面选择 180 都，需要在 device/rockchip/rkxxsdk/BoardConfig.mk 中打开下面的宏。



```
USE_OPENGL_RENDERER := true
BOARD_HAS_FLIPPED_SCREEN := true
```

然后执行下面命令，重新生成 recovery.img：

- (1) touch bootable/recovery/minui/graphics.c
- (2) make out/target/product/rk30sdk/recovery.img
- (3) ./mkimage.sh

#### e. [ ] RockChip Display transmitter support

该选项为显示转换芯片相关的配置选项、比如 LVDS、EDP、MIPI 等转换芯片的驱动程序，如



果相关芯片不需要程序去驱动，就不需要配置该选项。

#### HDMI 相关配置

```
Device Drivers --->
Graphics support --->
[*] Rockchip HDMI support --->
    --- Rockchip HDMI support
    [*] CAT66121 HDMI support
    HDMI Source LCDC select (lcdc1) --->
```

选中对应的 HDMI 驱动后，要选中 HDMI 对应的 LCDC，即 HDMI source。给 HDMI 用的 LCDC 为 external LCDC，这个可以从启动的 log 分析，配置是否正确。

```
0.210883] lcdc0 is used as primary display device controller!
0.217389] lcdc1 is used as external display device controller!
```

该 log 说明 LCDC0 是用作主显示设备，LCDC1 用做扩展显示设备，即给 HDMI 输出显示信号

下面列举下我们 SDK 中相关参考项目，以供开发过程中根据自己的硬件设计选做参考：

- (1) 双 LCDC 实现双显，LCDC0 接 LCD 屏幕、LCDC1 接 HDMI：

配置文件：rk3066\_sdk\_defconfig

板级文件：board-rk30-sdk.c

- (2) 双 LCDC 实现双显，LCDC0 接 HDMI, LCDC1 接屏幕

配置文件：rk3188\_ds1006h\_defconfig

板级文件：board-rk3188-ds1006h.c

- (3) EDP 接口的高清屏项目，双 LCDC 实现双显，LCDC0 接 LCD 屏幕，LCDC1 接 HDMI

配置文件：rk3188\_LR097\_defconfig

板级文件：board-rk3168-LR097.c

- (4) 单 LCDC + RK610 实现双显，LCDC1 接屏幕

配置文件：rk3168m\_tb\_defconfig

板级文件：board-rk3168m-tb.c

## 3 HDMI

### 3.1 HDMI 基础

HDMI 是 High Definition Multimedia Interface 的简称，是一种全数字化图像和声音传送接口，可以传送未压缩的音频及视频信号。HDMI 可用于机顶盒、DVD 播放机、个人电脑、电视游乐器、综合扩大机、数字音响与电视机等设备。HDMI 主要由 3 个模块组成：HDMI 线、HDMI Source 和 HDMI Sink，如图 1-1 所示：

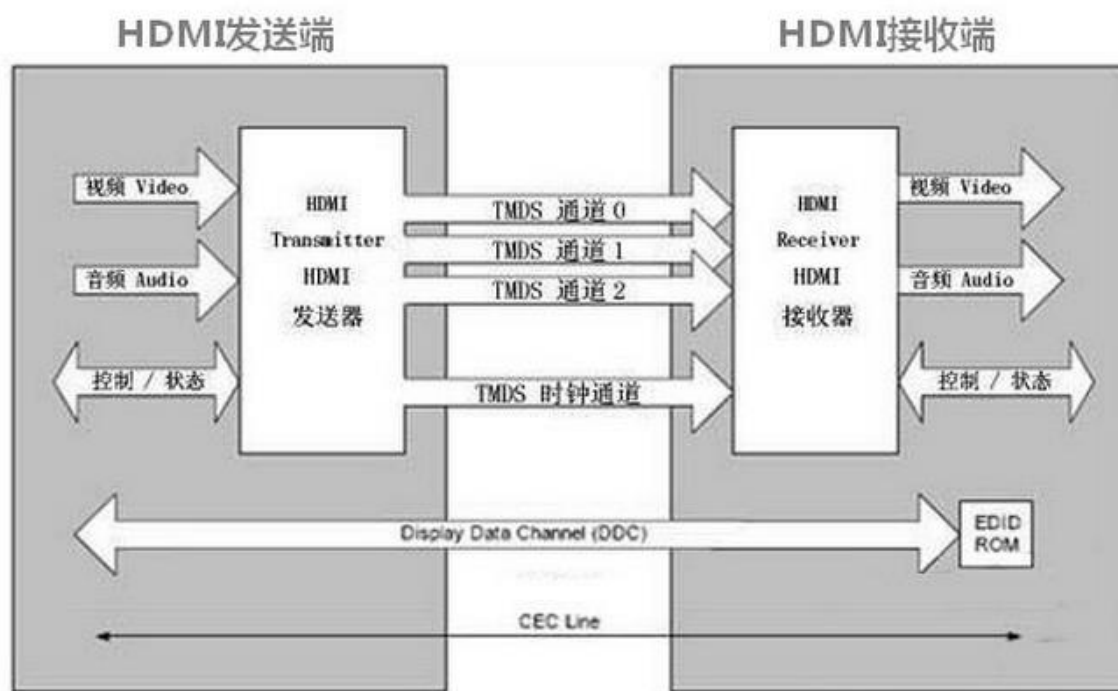


图 1-1

#### (1) PIN 脚定义

HDMI 通过 TMDS（最小化传输差分信号）技术传输音视频数据，标准的 Type A HDMI 接头有 19 个脚位，另有一种支持更高分辨率的 Type B (29 pin)，可传输 HDMI A type 两倍的 TMDS 数据量，还有 Type C 和 Type D 类型的接口用于不同的用途。常用的标准 Type A (19 pin) 接头的 pin 脚定义如图 2-1 所示：

Pin	Pin定义
1	TMDS Data2+
2	TMDS Data2 Shield
3	TMDS Data2 -
4	TMDS Data1+
5	TMDS Data1 Shield
6	TMDS Data1 -
7	TMDS Data0+
8	TMDS Data0 Shield
9	TMDS Data0 -
10	TMDS Clock+
11	TMDS Clock Shield
12	TMDS Clock -
13	CEC
14	Reserved (N.C. on device)
15	SCL
16	SDA
17	DDC/CEC Ground
18	+5V Power
19	Hot Plug Detect

图 2-1

## (2) TMDS

TMDS (Transition Minimized Differential Signaling) 也被称为最小化传输差分信号, HDMI 通过 TMDS 通道传输音频、视频以及各种辅助数据。支持的像素编码方式有: RGB 4:4:4, YCbCr 4:4:4 (8-16 bits per component); YCbCr 4:2:2 (12 bits per component)。不同分辨率对应的 TMDS 时钟频率如图 2-2 所示:

	1080p	720P	576P	480P
Clock rate(bps)	148.5MHz	74.25MHz	27MHz	27MHz

图 2-2

### (3) DDC

DDC 全称为 Display Data Channel，传送端与接收端可利用 DDC 通道得知彼此的传送与接收能力，但 HDMI 仅需单向获知接收端(显示屏)的能力，即从接收端读取 EDID，EDID 全称为 Extended Display Identification Data (扩展显示标识数据)是一种 VESA 标准数据格式，提供了几乎所有显示参数的通用描述。DDC 通道 (SCL 和 SDA) 一般使用 100kHz 时钟频率的 I<sup>2</sup>C 信号。

### (4) CEC

CEC 全称为 Consumer Electronics Control，用于电子消费产品的遥控控制。

### (5) HPD

Hot Plug 为 HDMI 中断检测脚，用于判断是否有 HDMI 的插入和拔插。

## 3.2 HDMI 驱动框架及实现

本部分描述系统的源代码分布框架，说明源代码存放的目录结构、各源文件的功能，以及软件处理流程以及配置说明。

### 3.2.1 HDMI 代码框架

#### (1) 目录结构

目录名称	目录说明	包含源文件列表
drivers/video/rockchip/hdmi/	实现 hdmi 框架 相关的代码	rk_hdmi.h rk_hdmi_task.c rk_hdmi_lcdc.c rk_hdmi_sysfs.c rk_hdmi_edid.c

drivers/video/rockchip/hdmi/chips/rk3288/ drivers/video/rockchip/hdmi/chips/rk616/ drivers/video/rockchip/hdmi/chips/cat66121/ drivers/video/rockchip/hdmi/chips/rk30/	各个 HDMI 芯片 的驱动代码	例如 rk3288 芯片包含以下 源文件：  Rk3288_hdmi.c  Rk3288_hdmi.h  rk3288_hdmi_hw.c  rk3288_hdmi_hw.h
---	---------------------	--

## (2) 源文件描述

源文件名称	文件描述
<b>HDMI 模块</b>	
rk_hdmi.h	相关结构体定义
rk_hdmi_task.c	HDMI 框架及驱动处理流程的实现
rk_hdmi_lcdc.c	与 lcdc 相关的接口函数实现
rk_hdmi_sys.c	HDMI sysfs 调试节点的代码实现
rk_hdmi_edid.c	解析 EDID 数据的相关函数实现
rkxxx_hdmi.c	实现 insert, remove, shutdown 接口，向核心层注册 HDMI 结构
rkxxx_hdmi.h	相关结构体定义.
rkxxx_hdmi_hw.c	HDMI 检测热插拔，读取 EDID, 配置音视频参数，系统输出和休眠等功能函数的实现
rkxxx_hdmi_hw.h	HDMI 的各个寄存器值和结构体定义，以及外部函数声明

### 3.2.2 HDMI 软件流程

HDMI 驱动框架主要实现 HDMI 热插拔检测、EDID 读取及解析、视频参数配置、音频参数配置、

系统输出和系统休眠等几个主要功能函数，主要的处理流程如图 3-1 所示：

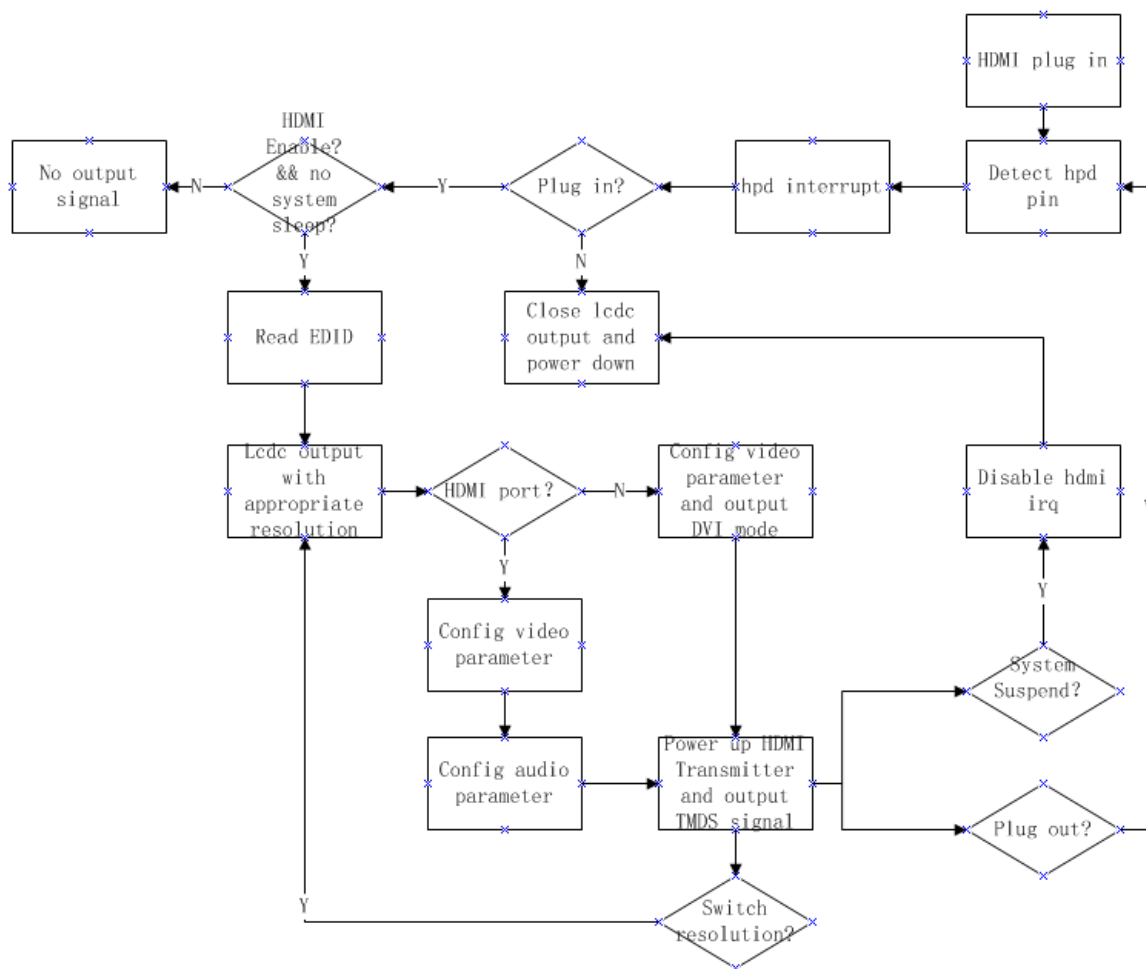


图 3-1

## 3.3 HDMI 相关配置

### 3.3.1 Linux3.0.36 内核配置

#### (1) MAKE MENUCONFIG 配置

```

Device Drivers --->
  Graphics support --->
    [*] Rockchip HDMI support --->
      --- Rockchip HDMI support
      [ ] CAT66121 HDMI support
      [*] RK616 HDMI support
  
```

```
[*] RK616 HDCP support
    HDMI Source LCDC select (lcdc1) --->
[ ] Rockchip HDMI Debugging
--*-- Mute Codec When HDMI Activated
```

(a) HDMI 芯片驱动选择,

若使用 ITE66121, 选择 CAT66121 HDMI support; 若使用 RK616, 选择 RK616 HDMI support;  
若使用 RK610, 选择 RK610 HDMI support。

(b) [ ] RK616 HDCP support

该选项用于配置 HDCP 功能是否可用, HDCP 功能依赖于 HDMI, 若要使 RK616 的 HDCP 功能可用, 请把 RK616 HDCP support 选项选上。

(c) HDMI Source LCDC select ( )

该选项用于配置与 HDMI 芯片连接的是主控的哪个 LCDC, 请根据实际电路连接进行配置。

(d) [ ] Rockchip HDMI Debugging

该选项为调试开关, 一般不需要打开。

## (2) board 文件配置

HDMI 芯片的 Reset 脚和 INT 脚均在 board 文件中配置, 下图是 RK610 的 RST 脚配置 (使用 GPIO3\_B2 作为 RK610 的 RESET 脚), 请根据硬件连接进行配置。

```
#if defined(CONFIG_MFD_RK610)
#define RK610_RST_PIN RK30_PIN3_PB2
static int rk610_power_on_init(void)
{
    int ret;
    if(RK610_RST_PIN != INVALID_GPIO)
    {
        ret = gpio_request(RK610_RST_PIN, "rk610 reset");
        if (ret)
        {
            printk(KERN_ERR "rk610_control_probe request gpio fail\n");
        }
        else
        {
            gpio_direction_output(RK610_RST_PIN, GPIO_HIGH);
            msleep(100);
            gpio_direction_output(RK610_RST_PIN, GPIO_LOW);
            msleep(100);
            gpio_set_value(RK610_RST_PIN, GPIO_HIGH);
        }
    }

    return 0;
}
```

下图是 RK610 的中断脚配置，若使用 GPIO 口作为中断脚，请把 irq 配置成对应的 GPIO 口，若硬件上 INT 脚没有连接，请将 irq 配置成 INVALID\_GPIO。

```
#ifdef CONFIG_HDMI_RK610
{
    .type           = "rk610_hdmi",
    .addr           = 0x46,
    .flags          = 0,
    .irq            = INVALID_GPIO,
},
#endif
```

若使用 RK616（或 RK618）芯片，还需要在 board-xxx.c 文件中配置 RK616（或 RK618）的 lcd0 接口和 lcd1 接口。若作为输入口，配置成 INPUT；若作为输出口，配置成 OUTPUT；若未使用，必须配置成 UNUSED，请根据具体的硬件连接进行配置。若 RK616 的 lcd0 接口作为输入，lcd1 接口作为输出，INT 中断脚连接 GPIO2\_D7，请配置 .lcd0\_func = INPUT, .lcd1\_func = OUTPUT, .hdmi\_irq = RK30\_PIN2\_PD6，如下图所示：

```
static struct rk616_platform_data rk616_pdata = {
    .power_init = rk616_power_on_init,
    .scl_rate   = RK616_SCL_RATE,
    .lcd0_func  = INPUT,           //port lcd0 as input
    .lcd1_func  = OUTPUT,         //port lcd1 as output
    .lvds_ch_nr = 1,              //the number of used lvds channel
    .hdmi_irq   = RK30_PIN2_PD6,
    .spk_ctl_gpio = RK30_PIN2_PD7,
};
```

### 3.3.2 Linux3.10 内核配置

对于 Linux3.10 以后版本的内核，不再使用 board 文件，硬件相关的配置参数通过 dts 传递。

#### (1) HDMI Video Source Select

对于 RK3288 芯片，HDMI 默认选择 LCDC0 作为输入源，如下图所示：

```
lcdc0: lcdc@ff930000 {
    compatible = "rockchip,rk3288-lcdc";
    rockchip,prop = <EXTEND>;
```

<EXTEND>属性的 LCDC 用于 HDMI 显示，若需要使用 LCDC1 作为 HDMI Source，请修改 dtsi 文件中的 lcdc1 节点的“rockchip,prop”属性为<EXTEND>，且修改 lcdc0 节点的“rockchip,prop”属性为<PRMRY>。



## (2) HDMI Audio Source Select

音频输入接口默认选择 I2S 接口

```
&hdmi {  
    status = "okay";  
    rockchip,hdmi_audio_source = <0>;  
};
```

选择 I2S 接口: dts 文件中的 hdmi 节点中的“rockchip,hdmi\_audio\_source”属性配置为<0>, 同时配置 CONFIG\_SND\_RK\_SOC\_I2S=y 和 CONFIG\_SND\_RK\_SOC\_HDMI\_I2S=y。

选择 SPDIF 接口: dts 文件中的 hdmi 节点中的“rockchip,hdmi\_audio\_source”属性配置为<1>, 同时配置 CONFIG\_SND\_RK\_SOC\_SPDIF=y 和 CONFIG\_SND\_RK\_SOC\_HDMI\_SPDIF=y。

## (3) 外部 HDMI 芯片的配置

若主控芯片中不带 HDMI, 即使用外部 HDMI 芯片 (如 RK616、CAT66121 等芯片), 还需配置中断的对应的 GPIO 口和 power ctrl 等信息, 如配置中断脚为 GPIO2\_D6, 配置芯片 Reset 脚为 GPIO3\_B2, 则配置如下图所示:

```
&cat66121 {  
    rockchip,hdmi_irq_gpio = <&gpio2 GPIO_D6 1>;  
  
    power_ctr: power_ctr {  
        rockchip,debug = <0>;  
        power_rst: power_rst {  
            rockchip,power_type = <GPIO>;  
            gpios = <&gpio3 GPIO_B2 GPIO_ACTIVE_HIGH>;  
            rockchip,delay = <100>;  
            rockchip,is_rst = <1>;  
        };  
    };  
};
```

## 3.4 HDMI 调试节点说明

在 sys/class/display/HDMI 节点下列出了 HDMI 的几个主要属性:

- (1) connect: HDMI 或 DVI 是否连接, 1 为连接, 0 为未连接。
- (2) enable: HDMI 是否可用, 1 为可用, 0 为不可用。
- (3) mode: 当前设置的分辨率, 例如通过 echo 1920x1080p-60 > mode 可修改分辨率为 1080p 且屏幕刷新率为 60Hz。
- (4) modes: 列出 Sink 端支持的且 Source 端也支持的所有分辨率。
- (5) scale: 水平缩放值和竖直缩放值, 默认 xscale 和 yscale 为 100

## 4 背光控制

在 linux3.10 版本的内核中，背光驱动为 drivers/video/backlight/pwm\_bl.c

相关的控制参数从 dts 里面传递：

```
backlight {
    compatible = "pwm-backlight";
    pwms = <#pwm0 0 25000>;
    brightness-levels = <0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101
102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185
186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227
228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255>;
    default-brightness-level = <128>;
    enable-gpios = <#gpio0 GPIO_A2 GPIO_ACTIVE_HIGH>;
};
```

Pwms = <#pwm0 0 25000>:为背光 pwm 控制端口定义， 第三项为 pwm 周期，单位纳秒

Brightness-levels:是具体的背光亮度列表。根据背光硬件设计的是正极性还是负极性，采用正序或者倒序去排列。

default-brighness-level: 为默认背光亮度

enable-gpios: 为背光电源控制使能 IO

## 附录:

### A1 LVDS 、EDP、MIPI 显示接口转换芯片及其驱动

从接口上区分，我们现在使用的屏幕大概有四种：RGB、LVDS、EDP、MIPI。一般小尺寸的屏幕（七寸以下）大部分使用 RGB 接口，大尺寸的高清屏幕都是用 LVDS 接口或者 EDP、MIPI 接口。

对于 RGB 接口的屏幕，LCDC 送出来的信号直接给屏幕，中间不需要其他的转换。

对于 LVDS 接口的屏幕，由于我们 RK30XX LCDC 输出的是 RGB 信号，所以中间要经过 RGB —— LVDS 信号的转换，这个转换由外接 LVDS 芯片实现；RK292X/RK302X 内部集成有 LVDS 模块，既可以输出 LVDS 信号，又可以通过 bypass 功能输出 RGB 信号。

LVDS 转换芯片，有的是不需要程序驱动，上电即可正常运行，比如 TI 的。

SN75LVDS83A 系列 LVDS 转换芯片，在 RK30XX 平台上，对于使用这种 LVDS 转换芯片的 LVDS 屏幕，在程序上不需要做特别设置，和 RGB 屏幕一样。有些 LVDS 转换芯片是需要程序进行驱动的，比如 RK61X 里面集成的 LVDS 转换模块。

在我们的 sdk 里面，目前需要进行驱动的 lvds 芯片有三颗，分别是 RK61X、rk292x 和 RK302X 里面自带的 lvds 模块，所以在使用的时候，要进行相应配置。

RK610 是多功能设备（Multifunction device），所以在使用的时候，先要做如下配置：

```
Device Driver
[*] Multifunction device drivers --->
    [*] RK610(Jetta) Multimedia support s --->

Device Driver
Graphics support --->
<*>Frame buffer support for Rockchip --->
[*] RockChip display transmitter support --->
    ---RockChip display transmitter support
    [*] RK610(Jetta) lvds transmitter support
    [ ] RGB to Display Port transmitter anx6345, anx9804, anx9805 support
    [ ] RockChip MIPI DSI Support
```

如果是 rk2928，则要做如下配置

```
Device Drivers --->
```

```
Graphics support --->
<*>Frame buffer support for Rockchip --->
[*] RK_LVDS support
```

EDP 转换接口的芯片，目前我们 sdk 中支持 ANX9804、ANX9805、ANX6345、DP501 四款芯片。

其中 ANX 系列的三款芯片在驱动上是兼容的，配置路径如下：

```
Device Driver
Graphics support --->
Display device support --->
[*] RockChip display transmitter support --->
[*] RGB to Display Port transmitter anx6345, anx9804, anx9805
[ ] RGB to Display Port transmitter dp501 support (NEW)
```

## A2 MIPI 屏幕的使用

RK 平台中目前支持三款 MIPI 驱动 IC，SSD2828、TC358768、RK618。

### 1 相关代码

相关的驱动代码如下：

```
drivers/video/rockchip/transmitter/mipi_dsi.c
drivers/video/rockchip/transmitter/mipi_dsi.h
drivers/video/rockchip/transmitter/ssd2828.c
drivers/video/rockchip/transmitter/tc358768.c
drivers/video/rockchip/transmitter/rk616_mipi_dsi.c
drivers/video/rockchip/transmitter/rk616_mipi_dsi.h
drivers/video/rockchip/screen/lcd_LD089WU1_mipi.c
```

### 2 内核配置

使用 mipi 接口的屏，kernel 配置如下：

```
Device Drivers --->
Graphics support --->
[*] RockChip display transmitter support --->
[*] RockChip display transmitter support --->
[*] Rockchip MIPI DSI support
```

选中你所使用的 mipi 芯片型号：

```
< >    toshiba TC358768 RGB to MIPI DSI
< >    solomon SSD2828 RGB to MIPI DSI
< >    RK616(JettaB) mipi dsi support
```

### 3 屏幕配置文件

请参考 drivers/video/rockchip/screen/lcd\_LD089WU1\_mipi.c

包含头文件：`#include "../transmitter/mipi_dsi.h"`

`#define SCREEN_TYPE SCREEN_MIPI` 定义屏幕类型为 MIPI，`SCREEN_TYPE` 如果不设为 `SCREEN_MIPI`，MIPI 的相关驱动不会执行。

`#define MIPI_DSI_LANE 4` //定义 LANE 个数

`MIPI_DSI_LANE` 是 LCD 的 DATA LANE 数，现在大多数分辨率稍高点的一般是 4 个 LANE 的，所以默认可以不用设置，必须和硬件相匹配。

`#define MIPI_DSI_HS_CLK 1000*1000000` //DSI\_HS\_CLK

`MIPI_DSI_HS_CLK` 是各个 LANE 传输时的速率(dds 双边沿)，最小值 80Mbps，最大 1Gbps，默认是 1Gbps，这个也基本上不用改，除非板子布线有问题，跑不了 1Gbps 的。

如果 MIPI 屏幕本身需要初始化，要定义 `RK_SCREEN_INIT`

`#define RK_SCREEN_INIT 1`

并实现下面两个函数：

```
int rk_lcd_init(void) {
    u8 dcs[16] = {0};
    if(dsi_is_active() != 1)
        return -1;
    /*below is changeable*/
    msleep(50);
    dsi_enable_hs_clk(1);

    dcs[0] = LPDT;
```

```

        dcs[1] = dcs_exit_sleep_mode;
        dsi_send_dcs_packet(dcs, 2);
        msleep(1);
        dcs[0] = LPDT;
        dcs[1] = dcs_set_display_on;
        dsi_send_dcs_packet(dcs, 2);
        msleep(10);
        dsi_enable_video_mode(1);
    }

int rk_lcd_standby(u8 enable) {

    u8 dcs[16] = {0};
    if(dsi_is_active() != 1)
        return -1;

    if(enable) {

        /*below is changeable*/
        dcs[0] = LPDT;
        dcs[1] = dcs_set_display_off;
        dsi_send_dcs_packet(dcs, 2);
        msleep(1);
        dcs[0] = LPDT;
        dcs[1] = dcs_enter_sleep_mode;
        dsi_send_dcs_packet(dcs, 2);
        msleep(1);

    } else {
        /*below is changeable*/
        rk_lcd_init();
    }
}

```

说明:

1、dsi\_is\_active()用来判断 DSI 是否已经处于工作状态了，如果是返回 1，如果不是返回 0，如果出错返回-1。

2、dcs[0] = LPDT;

```
dcsc[1] = dcs_enter_sleep_mode;
```

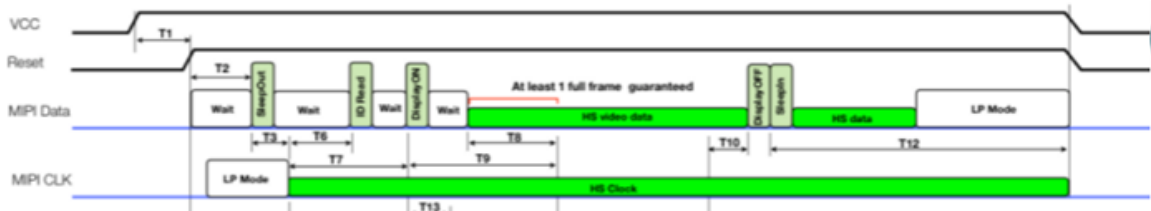
```
dsi_send_dcs_packet(dcs, 2);
```

屏的初始化是通过 DSI 来传送数据的，根据需要封装了 dsi\_send\_dcs\_packet 这个接口。对于 dcs[] 数组，第一个成员为写的模式，第二个成员为 DCS 命令，后面的成员为 DCS 命令的参数。

上面两句用来发送，dcs\_exit\_sleep\_mode 这个命令，这些 DCS 命令是在 drivers/video/rockchip/transmitter/mipi\_dsi.h 定义的，具体的含义请参考 MIPI DCS 的规范。当然屏自定义的命令也可以通过该接口发送。

### 3、dsi\_enable\_hs\_clk(1); 使能高速时钟

在写 LCD 初始化函数时，要根据 LCD 的初始化时序来操作，如下图：



```
dsi_enable_hs_clk(1);
dcs[0] = LPDT;
dcs[1] = dcs_exit_sleep_mode;
dsi_send_dcs_packet(dcs, 2);
msleep(1);
dcs[0] = LPDT;
dcs[1] = dcs_set_display_on;
dsi_send_dcs_packet(dcs, 2);
msleep(10);
dsi_enable_video_mode(1);
```

4、/\*below is changeable\*/ 有标这个注释的地方，可以根据自己的需要去初始化 LCD，包括 LCD 唤醒，复位等操作。

大部分的 MIPI 接口的屏都满足#define MIPI\_DSI\_LANE 4, #define MIPI\_DSI\_HS\_CLK 1000\*1000000, 并且不需要特殊初始化（只要向 LCD 发送 sleep out 和 display on 命令即可，或者都不需要），在编写 MIPI 接口 LCD 的驱动时，只要将 OUT\_TYPE 设为 SCREEN\_MIPI 即可。

5、板级上的配置与普通的 LCD 驱动一样，无需作特殊处理。

## A3 RK32XX MIPI 屏幕的使用

RK32XX 内嵌双通道 MIPI 控制 IC。

### 1 相关代码

相关的驱动代码如下：

```
drivers/video/rockchip/transmitter/mipi_dsi.c
drivers/video/rockchip/transmitter/mipi_dsi.h
drivers/video/rockchip/transmitter/rk32_mipi_dsi.c
drivers/video/rockchip/transmitter/rk32_mipi_dsi.h
drivers/video/rockchip/screen/ lcd_mipi.c
```

### 2 内核配置

使用 mipi 接口的屏，lcd 配置如下：

```
Device Drivers --->
Graphics support --->
[*] rk3288 lcdc support
    LCD Panel Select (General lcd panel) --->
        ( ) General lcd panel
        (*) rk mipi dsi lcd
```

Display transmitter 配置如下：

```
Device Drivers --->
Graphics support --->
[*] RockChip display transmitter support --->
    [*] Rockchip MIPI DSI support
        <*> rk32 mipi dsi support
```

### 3 屏幕配置文件

双 mipi 屏 dts 配置文件，请参考 arch/arm/boot/dts/lcd-wqxda-mipi.dtsi

单 mipi 屏 dts 配置文件，请参考 arch/arm/boot/dts/lcd-ld089wul-mipi.dtsi



以 arch/arm/boot/dts/lcd-wqxa-mipi.dtsi 为例，描述下 dts 的各个表项。

```
/* about mipi */

disp_mipi_init: mipi_dsi_init{

    compatible = "rockchip,mipi_dsi_init";

    rockchip,screen_init    = <1>;//mipi 屏需要初始化设置为 1，否则设置为 0

    rockchip,dsi_lane       = <4>;//定义 lane 的数目

    rockchip,dsi_hs_clk      = <940>;//设置 mipi 各个 lane 传输时的速率。

    rockchip,mipi_dsi_num   = <2>;//双 mipi 设置为 2，单 mipi 设置为 1

};

disp_mipi_power_ctr: mipi_power_ctr {

    compatible = "rockchip,mipi_power_ctr";

    //如果 mipi 屏有 rst 管脚，需要定义 mipi_lcd_rst

    mipi_lcd_rst:mipi_lcd_rst{

        compatible = "rockchip,lcd_rst";

        rockchip,gpios = <&gpio7 GPIO_A4 GPIO_ACTIVE_HIGH>;//设置 rst 管脚

        rockchip,delay = <10>;//设置 rst 延时时间，单位 ms

    };

    //如果 mipi 屏有 en 管脚，需要定义 mipi_lcd_en

    mipi_lcd_en:mipi_lcd_en {

        compatible = "rockchip,lcd_en";

        rockchip,gpios = <&gpio7 GPIO_A3 GPIO_ACTIVE_HIGH>;//设置 mipi 屏的

                                                //enable 管脚

        rockchip,delay = <10>;//设置 enable 延时时间，单位 ms

    };

};

//如果 mipi 屏需要初始化，需要定义 mipi_lcd_en
```

```
disp_mipi_init_cmds: screen-on-cmds {

    rockchip, cmd_debug = <0>;

    compatible = "rockchip, screen-on-cmds";

    rockchip, on-cmds1 {

        compatible = "rockchip, on-cmds";

        rockchip, cmd_type = <LPDT>; //cmd 类型, LPDT 或者 HSDT

        rockchip, dsi_id = <2>; //0: 只在 mipi0 发送此命令
                                //1: 只在 mipi1 发送此命令
                                //2: 在 mipi0 和 mipi1 发送此命令

        rockchip, cmd = <0x05 0x01>; //发送的具体命令

        rockchip, cmd_delay = <10>; //发送命令后的延时

    };

    .....

    rockchip, on-cmds19 {

        compatible = "rockchip, on-cmds";

        rockchip, cmd_type = <LPDT>;

        rockchip, dsi_id = <2>;

        rockchip, cmd = <0x05 dcs_set_display_on>;

        rockchip, cmd_delay = <10>;

    };

};

//定义 MIPI 屏的 timing

disp_timings: display-timings {

    native-mode = <&timing0>;

    compatible = "rockchip, display-timings";

    timing0: timing0 {
```

```

screen-type = <SCREEN_DUAL_MIPI>;//双 mipi 屏为 SCREEN_DUAL_MIPI
                                //单 mipi 屏为 SCREEN_MIPI

lvds-format = <LVDS_8BIT_2>;//这里不管

out-face    = <OUT_P888>;//设置 pixel 格式

clock-frequency = <265000000>;//设置 dclk, 单位为 HZ

hactive = <2560>;

vactive = <1600>;

hsync-len = <38>;

hback-porch = <40>;

hfront-porch = <108>;

vsync-len = <4>;

vback-porch = <4>;

vfront-porch = <12>;

hsync-active = <0>;

vsync-active = <0>;

de-active = <0>;

pixelclk-active = <0>;

swap-rb = <0>;

swap-rg = <0>;

swap-gb = <0>;

};

};

```

## 4 板级配置文件

MIPI 控制 IC 的板级配置文件在 arch/arm/boot/dts/rk3288.dtsi。

单 mipi 屏只需将 dsi host0 打开，而双 mipi 屏还需要将 dsi host1 打开。

```
dsihost0: mipi@ff960000{

    compatible = "rockchip,rk32-dsi";

    rockchip,prop = <0>;

    reg = <0xff960000 0x4000>;

    interrupts = <GIC_SPI 19 IRQ_TYPE_LEVEL_HIGH>;

    clocks = <&clk_gates5 15>, <&clk_gates16 4>, <&pd_mipidsi>;

    clock-names = "clk_mipi_24m", "pclk_mipi_dsi", "pd_mipi_dsi";

    status = "okay";

};

dsihost1: mipi@ff964000{

    compatible = "rockchip,rk32-dsi";

    rockchip,prop = <1>;

    reg = <0xff964000 0x4000>;

    interrupts = <GIC_SPI 20 IRQ_TYPE_LEVEL_HIGH>;

    clocks = <&clk_gates5 15>, <&clk_gates16 5>, <&pd_mipidsi>;

    clock-names = "clk_mipi_24m", "pclk_mipi_dsi", "pd_mipi_dsi";

    status = "okay";//双 mipi 屏设置为 okay，单 mipi 屏设置为 disabled

};
```

## 5 MIPI 屏的初始化代码

如果 MIPI 屏本身需要初始化，需要将屏 dtsi 文件中的“rockchip,screen\_init”设置为 1。

初始化的代码在 lcd\_mipi.c 中，各函数的功能描述如下。

```
void rk_mipi_screen_pwr_disable(struct mipi_screen *screen)
```

功能：通过控制 rst，en 管脚，关闭 MIPI 屏。

```
void rk_mipi_screen_pwr_enable(struct mipi_screen *screen)
```

功能：通过控制 rst，en 管脚，开启 MIPI 屏。

```
void rk_mipi_screen_cmd_init(struct mipi_screen *screen)
```

功能：如果屏需要初始化，将通过这个函数，解析 dtsti 中的初始化命令序列，对 mipi 屏进行初始化。

```
int rk_mipi_screen_init_dt(struct mipi_screen *screen)
```

功能：实现对 mipi 屏 dtsti 文件的解析。

```
int rk_mipi_screen_standby(u8 enable)
```

功能：实现对 mipi 屏的打开及关闭，enable：0-打开，1-关闭。

## A3 快速判断 RGB 颜色是否有颠倒

在有些项目上，由于 LCD 屏幕自身原因或者硬件连接原因，会导致 RGB 颜色出现颠倒，比如 R 和 G 反了、或者 R 和 B 反了、或者 B 和 G 反了，导致显示颜色出现异常。这种情况可以通过软件配置进行纠正。

首先，要判断出哪两种颜色反掉了，方法如下：通过 RK 系统中自带的 IO 命令向 FB 里面写单元色红色（0x00ff0000）、绿色（0x0000ff00）、蓝色（0x000000ff）看对应的显示效果，就可以判断是哪两种颜色反掉了。具体方法如下：

（1）在串口里面执行 stop 命令（需要 root 权限），这样 Android 不再进行屏幕更新，以免影响测试。

（2）然后查询 LCDC 寄存器，获取当前 FB 地址：

我们系统启动的时候，会打印 LCDC 寄存器的基地址，比如 RK3188 系统启动 log 有如下打印：

```
0.625959] lcdc0:reg_phy_base = 0x1010c000,reg_vir_base:0xf709c000
0.626044] fb0:win0
0.626049] fb1:win1
0.626054] fb2:win2
0.626194] rk3188 lcdc0 clk enable...
0.694206] rk3188 lcdc0 clk disable...
0.723944] fb0:phy:90c00000>>vir:f8000000>>len:0xc00000
0.724187] rk_fb_register>>>>fb0
0.845645] rk_fb_register>>>>fb1
0.845748] rk3188 lcdc0 clk enable...
0.845788] lcdc0: dclk:64000000>>fps:56
```

这个说明 LCDC0 寄存器的物理地址是 0x1010c000，虚拟地址是 0xf709c000，fb0 对应 lcdc0 的 win0，其物理地址基地址为 0x90c00000，大小为 0xc00000。由于我们的系统使用的是双 buffer

或者 3 buffer 机制，FB 的当前显示地址会在 0x90c00000 到 0x90c00000 + 0xc00000 动态切换即 (0x90c000/0x90fe8000/0x913d0000)。

然后执行在串口里面执行 `io -r -4 -l 200 0x1010c000` 读取 LCDC 寄存器的配置信息

```
shell@android:/ # io -r -4 -l 200 0x1010c000
1010c000: 90000001 08000c00 00000000 00000000
1010c010: 00000020 00000000 00000000 00000000
1010c020: 90fe8000 90c00000 00000000 00000000
1010c030: 00000500 031f04ff 031f04ff 000a006e
1010c040: 10001000 10001000 00000000 00000000
1010c050: 07ff07ff 00000000 00000000 00000000
1010c060: 00000000 00000000 00000000 0580000a
1010c070: 006e056e 03300002 000a032a 00000000
1010c080: 00000000 90000001 08000c00 00000000
1010c090: 00000000 00000000 00000000 00000000
1010c0a0: 00000000 00000000 00000000 00000000
1010c0b0: 00000000 00000000 00000000 00000000
1010c0c0: 00000000 00000000
```

由于上面 log 显示，FB0 对应于 LCDC 的 WIN0，查看 RK3188 LCDC 的 datasheet 可知，WIN0 RGB 数据的基地址寄存器为 0x1010c020，我们看到该寄存器里面存储的值为 0x90fe8000，这个即为当前 FB 的地址。

或者读取 sys 节点，也可以打印当前 fb 的地址

```
shell@android:/ # cat sys/class/graphics/fb0/disp_info
win0:enabled
xvir:1280
xact:1280
yact:800
xdsp:1280
ydsp:800
x_st:110
y_st:10
x_scale:1.0
y_scale:1.0
format:ARGB888
YRGB buffer addr:0x90fe8000
CBR buffer addr:0x90c00000

win1:disabled
xvir:0
xdsp:2048
ydsp:2048
x_st:0
y_st:0
format:ARGB888
YRGB buffer addr:0x00000000
overlay:win1 on the top of win0
```

上面打印信息说明 win0 RGB 格式数据当前的 FB 地址为：0x90fe8000

(3) 用 `io` 命令依次向该地址写入 RGB 单元色数据，看屏幕对应的显示情况：

`io -w -4 -l 0x3e8000 0x90fe8000 0x00ff0000`      红色

`io -w -4 -l 0x3e8000 0x90fe8000 0x0000ff00`      绿色

`io -w -4 -l 0x3e8000 0x90fe8000 0x000000ff`      蓝色

注意 -1 参数后面跟的是写入数据的长度，这里都是全屏写入，我使用的设备的屏幕分辨率为 1280\*800，因此一帧数据的长度为  $1280*800*4 = 0x3e8000$ 。

正常情况下，依次在串口中输入上面三个命令，屏幕上依次显示红、绿、蓝三种颜色，如果显示的某种颜色不对，比如写入红色，屏幕上显示的却是蓝色，如果 R 和 B 反了，则应该在屏幕驱动中#define SWAP\_RB 1，对 RB 进行交换。如果向 FB 里面写入 RGB 中任意一种的单元色屏而屏幕显示的图像不是这三个单元色中的任意一种，那就有可能是屏幕有问题，或者硬件设计有问题，或者中间的转换芯片比如 LVDS 有问题，或者 LVDS\_FORMAT 设置的不对。

RK2928 判断颜色方法类似，不过 rk2928 LCDC 的基址为 0x1010e000

```
lcdc0:reg_phy_base = 0x1010e000,reg_vir_base:0xf700c000
fb0:win0
fb1:win1
fb2:win2
rk_output_lvt1>>connect to lcdc output interface0
RGB screen connect to rk2928 lcdc interface0
lcdc0: dclk:33000000>>fps:59 rk2928_load_screen for lcdc0 ok!
fb0:phy:90c00000>>vir:f8000000>>len:0xc00000
```

而且 FB0 对应 LCDC0 的 WIN0，WIN0 RGB 数据的基址寄存器为 0x1010e01c

## A4 利用 LUT 功能改善显示效果

由于屏幕自身的 gamma 特性，同一 RGB 值的数据，输入到不同的屏幕，显示效果可能不同，为了弥补屏幕 gamma 的不一致性，RK30XX, RK31XX (RK292x 平台无此功能) 平台提供了 LUT 功能，可以对显示效果进行调整，类似 gamma 校正。使驱动用方法如下：

(1) 在屏幕的驱动配置文件 (lcd\_XXX.c) 中加入默认的 LUT 表，该表是理想的线性 LUT 表，可以从内核 sdk 中拷贝. 并且定义 USE\_RK\_DSP\_LUT 宏

kernel/drivers/video/display/screen/lcd\_b101ew05.c

```
#define USE_RK_DSP_LUT
int dsp_lut[256] = {
    0x00000000, 0x00010101, 0x00020202, 0x00030303, 0x00040404, 0x00050505, 0x00060606, 0x00070707,
    0x00080808, 0x00090909, 0x000a0a0a, 0x000b0b0b, 0x000c0c0c, 0x000d0d0d, 0x000e0e0e, 0x000f0f0f,
    0x00101010, 0x00111111, 0x00121212, 0x00131313, 0x00141414, 0x00151515, 0x00161616, 0x00171717,
    0x00181818, 0x00191919, 0x001a1a1a, 0x001b1b1b, 0x001c1c1c, 0x001d1d1d, 0x001e1e1e, 0x001f1f1f,
    0x00202020, 0x00212121, 0x00222222, 0x00232323, 0x00242424, 0x00252525, 0x00262626, 0x00272727,
    0x00282828, 0x00292929, 0x002a2a2a, 0x002b2b2b, 0x002c2c2c, 0x002d2d2d, 0x002e2e2e, 0x002f2f2f,
    0x00303030, 0x00313131, 0x00323232, 0x00333333, 0x00343434, 0x00353535, 0x00363636, 0x00373737,
    0x00383838, 0x00393939, 0x003a3a3a, 0x003b3b3b, 0x003c3c3c, 0x003d3d3d, 0x003e3e3e, 0x003f3f3f,
    0x00404040, 0x00414141, 0x00424242, 0x00434343, 0x00444444, 0x00454545, 0x00464646, 0x00474747,
    0x00484848, 0x00494949, 0x004a4a4a, 0x004b4b4b, 0x004c4c4c, 0x004d4d4d, 0x004e4e4e, 0x004f4f4f,
    0x00505050, 0x00515151, 0x00525252, 0x00535353, 0x00545454, 0x00555555, 0x00565656, 0x00575757,
    0x00585858, 0x00595959, 0x005a5a5a, 0x005b5b5b, 0x005c5c5c, 0x005d5d5d, 0x005e5e5e, 0x005f5f5f,
    0x00606060, 0x00616161, 0x00626262, 0x00636363, 0x00646464, 0x00656565, 0x00666666, 0x00676767,
    0x00686868, 0x00696969, 0x006a6a6a, 0x006b6b6b, 0x006c6c6c, 0x006d6d6d, 0x006e6e6e, 0x006f6f6f,
    0x00707070, 0x00717171, 0x00727272, 0x00737373, 0x00747474, 0x00757575, 0x00767676, 0x00777777,
    0x00787878, 0x00797979, 0x007a7a7a, 0x007b7b7b, 0x007c7c7c, 0x007d7d7d, 0x007e7e7e, 0x007f7f7f,
    0x00808080, 0x00818181, 0x00828282, 0x00838383, 0x00848484, 0x00858585, 0x00868686, 0x00878787,
    0x00888888, 0x00898989, 0x008a8a8a, 0x008b8b8b, 0x008c8c8c, 0x008d8d8d, 0x008e8e8e, 0x008f8f8f,
    0x00909090, 0x00919191, 0x00929292, 0x00939393, 0x00949494, 0x00959595, 0x00969696, 0x00979797,
    0x00989898, 0x00999999, 0x009a9a9a, 0x009b9b9b, 0x009c9c9c, 0x009d9d9d, 0x009e9e9e, 0x009f9f9f,
    0x00a0a0a0, 0x00a1a1a1, 0x00a2a2a2, 0x00a3a3a3, 0x00a4a4a4, 0x00a5a5a5, 0x00a6a6a6, 0x00a7a7a7,
    0x00a8a8a8, 0x00a9a9a9, 0x00aaaaaa, 0x00ababab, 0x00acacac, 0x00adadad, 0x00aeaeae, 0x00afafaf,
    0x00b0b0b0, 0x00b1b1b1, 0x00b2b2b2, 0x00b3b3b3, 0x00b4b4b4, 0x00b5b5b5, 0x00b6b6b6, 0x00b7b7b7,
    0x00b8b8b8, 0x00b9b9b9, 0x00bababa, 0x00bbbbbb, 0x00bcbcbc, 0x00bdbdbd, 0x00bebebe, 0x00bfbfbf,
    0x00c0c0c0, 0x00c1c1c1, 0x00c2c2c2, 0x00c3c3c3, 0x00c4c4c4, 0x00c5c5c5, 0x00c6c6c6, 0x00c7c7c7,
    0x00c8c8c8, 0x00c9c9c9, 0x00cacaca, 0x00cbcbcb, 0x00cccccc, 0x00cdcdcd, 0x00cecece, 0x00cfcfcf,
    0x00d0d0d0, 0x00d1d1d1, 0x00d2d2d2, 0x00d3d3d3, 0x00d4d4d4, 0x00d5d5d5, 0x00d6d6d6, 0x00d7d7d7,
    0x00d8d8d8, 0x00d9d9d9, 0x00dadada, 0x00dbdbdb, 0x00dcdcdc, 0x00dedede, 0x00dfdfdf,
    0x00e0e0e0, 0x00e1e1e1, 0x00e2e2e2, 0x00e3e3e3, 0x00e4e4e4, 0x00e5e5e5, 0x00e6e6e6, 0x00e7e7e7,
    0x00e8e8e8, 0x00e9e9e9, 0x00eaeaea, 0x00ebebeb, 0x00eccece, 0x00ededed, 0x00eeeeee, 0x00efefef,
    0x00f0f0f0, 0x00f1f1f1, 0x00f2f2f2, 0x00f3f3f3, 0x00f4f4f4, 0x00f5f5f5, 0x00f6f6f6, 0x00f7f7f7,
    0x00f8f8f8, 0x00f9f9f9, 0x00fafafa, 0x00fbfbfb, 0x00fcfcfc, 0x00fdfdfd, 0x00fefefe, 0x00ffffff
};
```

如果屏幕配置文件中没有这个表，调节 LUT 的时候，屏幕显示会花掉。

系统起来后会有如下节点：

sys/class/graphics/fb0/dsp\_lut

(2) 然后在 init.rc 里面给该节点加上相应的权限，使得 BizlineAdjust\_3XXX 能够对这个节点进行写入，如果权限不对，该 APK 无法对该节点进行读写操作，调节不会有效果。

(3) 安装 BizlineAdjust\_30XX，调整亮度和对比度状态条，得到自己认为满意的显示效果。该 apk 会把对应的 lut 数据保存到如下文件中：

/data/data/com.rockchip.graphics/files/dsp\_lut\_bkp

(4) 从 dsp\_lut\_bkp 文件中导出数据，复制到对应屏幕驱动的 dsp\_lut 【256】数组中。系统再次启动会从 dsp\_lut 中读取数据，写入 lcdc 的 lut 表中。

## A5 支持 BMP 格式的开机 logo

Linux 系统启动的时候，默认支持的开机 LOGO 是 RGB565（16）格式的（ppm），为了提高开机 LOGO 的显示质量，我们加入了对 BMP 格式（24bit）的支持。BMP 格式 LOGO 的配置方法如下：

- 1) 准备 24BIT 的 BMP 格式 LOGO 图片，不支持 32 bit，并且图片的分辨率不能超过屏幕分辨率。
- 2) 将图片命名为 xxx\_bmp.bmp（必须以\_bmp.bmp 结尾，比如 logo\_android\_bmp.bmp），放在



drivers/video/logo 目录下。

### 3) 修改 Makefile、Kconfig、logo.c linux\_logo.h:

```

--- a/drivers/video/logo/Kconfig
+++ b/drivers/video/logo/Kconfig
@@ -98,16 +98,21 @@ config LOGO_LINUX_800x480_CLUT224
    menuconfig LOGO_LINUX_BMP
        bool "Bmp logo support"
        default n
+config LOGO_LINUX_BMP_ANDROID
+    bool "Bmp logo android"
+    depends on LOGO_LINUX_BMP
+    default n

index c7396a4..ccae7cc 100644
--- a/drivers/video/logo/Makefile
+++ b/drivers/video/logo/Makefile
obj-$(CONFIG_LOGO_G3_CLUT224)      += logo_g3_clut224.o
obj-$(CONFIG_LOGO_LINUX_BMP_SUNSET) += logo_sunset_bmp.o
+obj-$(CONFIG_LOGO_LINUX_BMP_ANDROID) += logo_android_bmp.o

--- a/drivers/video/logo/logo.c
+++ b/drivers/video/logo/logo.c
@@ -141,9 +141,14 @@ const struct linux_logo * __init_refok fb_find_logo(int depth)
    if (depth >= 24)
    {
        #ifdef CONFIG_LOGO_LINUX_BMP
        #ifdef CONFIG_LOGO_LINUX_BMP_SUNSET
            logo = &logo_sunset_bmp;
        #endif
+
        #ifdef CONFIG_LOGO_LINUX_BMP_ANDROID
            logo = &logo_android_bmp;
        #endif
+
        #endif
    }

--- a/include/linux/linux_logo.h
+++ b/include/linux/linux_logo.h
@@ -54,6 +54,7 @@ extern const struct linux_logo logo_m32r_clut224;
```

```
extern const struct linux_logo logo_spe_clut224;
extern const struct linux_logo logo_g3_clut224;
extern const struct linux_logo logo_sunset_bmp;
+extern const struct linux_logo logo_android_bmp;
```

(4) 在 make menuconfig 里面去掉对普通 logo 的支持, 并打开 bmp logo support 和对应的 logo

```
Graphics support --->
[*] Bootup logo --->
[*] Bmp logo support --->
[*] Bmp logo android
```

需要注意的是, 由于 BMP 图像是 24bit 的, 所以如果使用 BMP 格式的 LOGO 会使生成的内核镜像变大, 可能超出分区或者 loader 的支持范围, 在编译的内核的时候应该使用 make zkernel.img 命令生成压缩的内核镜像。

## A6 高清屏优化

目前在 RK3188 平台上, 我们最高支持的屏幕分辨率可达 2048\*1536, 对于这种超大分辨率的屏幕, 对系统带宽要求极高 (每帧画面数据量为 12M), 为了让这种分辨率的屏幕, 获得更加流畅的用户体验, 我们采取了专门的优化策略。开启该优化测试需要两方面的支持:

(1) 内核为 GPU 开启 512M 内存, 该步骤在对应项目的 board 代码中实现:

```
#define HD_SCREEN_SIZE 1920UL*1200UL*4*3
static void __init rk30_reserve(void)
{
    int size, ion_reserve_size;
#if defined(CONFIG_ARCH_RK3188)
    /*if lcd resolution great than or equal to 1920*1200, reserve the ump memory */
    if(!(get_fb_size() < ALIGN(HD_SCREEN_SIZE, SZ_1M)))
    {
        int ump_mem_phy_size=512UL*1024UL*1024UL;
        resource_mali[0].start = board_mem_reserve_add("ump buf",
ump_mem_phy_size);
        resource_mali[0].end = resource_mali[0].start + ump_mem_phy_size -1;
    }
#endif
}
```

HD\_SCREEN\_SIZE 为我们定义的高清屏下限，即我们默认，如果某款屏幕的分辨率高于 1920\*1200，即开启高清优化，为 GPU 分配 512M 空间。如果你的屏幕的分辨率小于 1920\*1200，你也需要开启高清优化，那么需要修改 HD\_SCREEN\_SIZE 的大小，比如某款屏幕的分辨率为 1920\*1080，如果需要开启优化，需要把 HD\_SCREEN\_SIZE 改小一点：

```
#define HD_SCREEN_SIZE 1920UL*1080UL*4*3
```

(2) 开启 Android 层的高清优化宏开关：

在 device/rockchip/rk30sdk/BoardConfig.mk 中把 BOARD\_USE\_LCDC\_COMPOSER 定义为 true

```
BOARD_USE_LCDC_COMPOSER ?= true
```

然后 make clean，再重新编译 Android 固件。

## A7 基于 rk fb 驱动的应用开发

### 1 概述

首先 rk fb 是遵循 linux frame buffer 驱动扩展的 fb 驱动，因此基于 rk fb 的应用开发基本和标准的 linux frame buffer 应用开发一致。

RK 系列主控的 LCDC 在内部是分层的，每一次叫做 win，每一层可以在屏幕上任意位置显示支持范围内任意大小的图像，并且各层可以通过 alpha blending 或者 color key 实现 overlay 合成输出。其中 RK3066 的 LCDC 有三层，win0、win1、win2，win0、win1 支持 RGB 和 YUV 格式、win2 只支持 YUV 格式。RK292X、

RK3168、RK3188 的 LCDC 有两层，win0 支持 RGB 和 YUV 格式，win1 只支持 RGB 格式。

在 rk fb 中，每一层 win 对应一个 fb 设备，它们在 linux 系统中对应的设备节点为 /dev/graphics/fbx，其中 win 和 fb 的对应关系是可以通程序设置的，以 RK3066 为例：

```
/dev/graphics/fb0-----win1
```

```
/dev/graphics/fb1-----win0
```

```
/dev/graphics/fb2-----win2
```

具体的对应关系可以通过 cat sys/class/graphics/fb0/map 节点查询：

```
shell@android:/ # cat sys/class/graphics/fb0/map
fb0:win1
fb1:win0
fb2:win2
```

## 2 相关系统调用：

Fb 设备作为一个标准的字符设备，需要通过相关的系统调用对其进行操作，主要是 open、close、mmap、以及相关的 ioctl 调用，可以以 android sdk 中的 hal 层代码作为参考：

androidsrc/hardware/rk29/libgralloc\_ump/framebuffer\_device.cpp

### (1) open

在使用某个 fb 设备之前必须先 open：

```
Fd = open(/dev/graphics/fbx, O_RDWR, 0);
```

### (2) mmap

该调用用于将在内核中分配的 fb 地址映射到用户空间，以方便应用进行读写。

```
void* vaddr = mmap(0, fbSize, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

### (3) ioctl: FBIOGET\_FSCREENINFO

该 ioctl 为 linux fb 驱动原生的，用于获取 fb\_fix\_screeninfo

### (4) ioctl:FBIOGET\_VSCREENINFO

该 ioctl 为 linux fb 驱动原生的，用于获取 fb\_var\_screeninfo

### (5) ioctl: FBIOPUT\_VSCREENINFO

该 ioctl 为 linux fb 驱动原生的，用于修改设置 fb\_var\_screeninfo

应用显示相关的信息都通过修改 fb\_var\_screen\_info,然后通过该 ioctl 设置下去。

struct fb\_var\_screeninfo 是 fb 中十分重要的一个结构，大量和显示相关的参数都是通过该结构进行配置。

```
struct fb_var_screeninfo {
    __u32 xres;           /* visible resolution */
    __u32 yres;
    __u32 xres_virtual;   /* virtual resolution */
    __u32 yres_virtual;
    __u32 xoffset;        /* offset from virtual to visible */
```

```

__u32 yoffset;          /* resolution          */

__u32 bits_per_pixel;    /* guess what          */
__u32 grayscale;        /* != 0 Graylevels instead of colors */

struct fb_bitfield red;   /* bitfield in fb mem if true color, */
struct fb_bitfield green; /* else only length is significant */
struct fb_bitfield blue;
struct fb_bitfield transp; /* transparency          */

__u32 nonstd;            /* != 0 Non standard pixel format */

__u32 activate;          /* see FB_ACTIVATE_*     */

__u32 height;            /* height of picture in mm */
__u32 width;             /* width of picture in mm  */

__u32 accel_flags;       /* (OBSOLETE) see fb_info.flags */

/* Timing: All values in pixclocks, except pixclock (of course) */
__u32 pixclock;          /* pixel clock in ps (pico seconds) */
__u32 left_margin;       /* time from sync to picture */
__u32 right_margin;      /* time from picture to sync  */
__u32 upper_margin;      /* time from sync to picture  */
__u32 lower_margin;
__u32 hsync_len;          /* length of horizontal sync */
__u32 vsync_len;          /* length of vertical sync    */
__u32 sync;               /* see FB_SYNC_*             */
__u32 vmode;              /* see FB_VMODE_*            */
__u32 rotate;             /* angle we rotate counter clockwise */
__u32 reserved[5];        /* Reserved for future compatibility */
};

```

xres:内核启动的时候将其初始化为对应屏幕的横向分辨率，实际使用的过程中，他表示将要用于显示的数据的实际宽度，即实宽（xact）。

yres: 内核启动的时候将其初始化为对应屏幕的纵向分辨率，实际使用的过程中，它代表将要用于显示的数据的实际高度，即实高（yact）。

xres\_virtual:将要显示的数据的虚宽，步长（stride）。

yres\_virtual:设置为屏幕的纵向分辨率乘以 buffer 的个数。

xoffset: 内存中将用于显示的数据的横向偏移。

yoffset:内存中将用于显示的数据的纵向偏移。

fb 驱动将利用 xoffset 和 yoffset 去确定从什么地址去取数据进行显示。

bits\_per\_pixel:每个像素占多少 bit，RGB565 为 16bits，ARGB888 为 32bit。

还有其他的显示相关的信息需要设置，但是内核中原生的 fb 驱动并没有提供相关接口，我们利用 fb\_var\_screen\_info 中系统默认没有用到的成员变量进行设置。他们是对应内容在屏幕上的显示位置（xpos、ypos）、对应内容在屏幕上的显示大小（xsize，ysize），以及数据的显示格式（data\_format）

xsize:对应内容在屏幕上的显示长度，用 grayscale 的 bit8~bit19 表示

ysize: 对应内容在屏幕上的显示宽度，用 grayscale 的 bit20~bit31 表示。

因此：

$\text{grayscale} = (\text{xsize} \ll 8) \mid (\text{ysize} \ll 20)$

xpos:对应内容在屏幕上显示的 x 坐标，用 nonstd 的 bit8~bit19 表示。

ypos: 对应内容在屏幕上显示的 y 坐标，用 nonstd 的 bit20~bit31 表示。

data\_format:应用写到 fb 中的数据格式，用 nonstd 的 bit0~bit7 表示。

因此：

$\text{nonstd} = (\text{xpos} \ll 8) \mid (\text{ypos} \ll 20) \mid (\text{data\_format})$

数据格式用 rk\_fb.h 中的如下宏表示：

```
enum {
    HAL_PIXEL_FORMAT_RGBA_8888      = 1,
    HAL_PIXEL_FORMAT_RGBX_8888      = 2,
    HAL_PIXEL_FORMAT_RGB_888        = 3,
    HAL_PIXEL_FORMAT_RGB_565        = 4,
    HAL_PIXEL_FORMAT_BGRA_8888      = 5,
    HAL_PIXEL_FORMAT_RGBA_5551      = 6,
    HAL_PIXEL_FORMAT_RGBA_4444      = 7,
```

```
.....
HAL_PIXEL_FORMAT_YCrCb_NV12      = 0x20,
}
```

这些宏是 Android 系统定义的，需要说明的是，Android 宏定义中 ARGB 排列的顺序和数据实际在内存中排列的顺序是相反的，比如 RGBA\_8888 数据在内存中的排列顺序是 ABGR, RGBX\_8888 格式的数据在内存中的排列顺序是 XBGR8888, X 表明该位对应的 8bit 数据是无效的, BGRA\_8888 格式的数据在内存中的排列顺序是 ARGB8888, 以此类推。

#### (6) ioctl: FBIOPAN\_DISPLAY

该 ioctl 为 linux fb 原生的系统调用，应用调用该 ioctl 的时候，将调用到平台相关的 fb 驱动(rk\_fb.c)中注册的 pan\_display 函数，驱动将根据 fb\_var\_screen\_info 中设置的 xoffset、yoffset 设置新一帧数据的显示地址。另外需要说明的是，在执行 ioctl FBIOPUT\_VSCREENINFO 的时候，linux 系统会自动调用平台相关的 fb 驱动中注册的 pan\_display 函数。

以上为 linux framebuffer 中原生的系统调用，为了适应 rockchip 自动平台的相关特性，我们扩展了相关的 ioctl 系统调用，这些系统调用在头文件 rk\_fb.h 中定义。

#### (7) ioctl: RK\_FBIOGET\_PANEL\_SIZE

该系统调用用于获取屏幕的分辨率，它返回一个指向 u32 panel\_size[2] 的指针，其中 panel\_size[0] 中保存的为屏幕的 x 分辨率，panel\_size[1] 中保存的为屏幕的 y 分辨率。

#### (8) ioctl: RK\_FBIOSET\_YUV\_ADDR

该系统调用用于设置 fb 的基地址，rk fb 驱动默认只为 fb0 分配 buffer，如果应用要使用到 fb1、fb2、需要应用分配物理上连续的地址，并将地址通过该系统调用传递给 fb。该系统调用传递一个指向 u32 addr[2] 的指针，如果应用绘制的是 RGB 数据，只需要将 RGB 数据对应的物理地址保存在 addr[0] 中，addr[1] 中的数据无意义，如果应用绘制的是 YUV 数据，则 addr[0] 保存的是 Y 数据对应的物理地址，addr[1] 中保存的是 UV 数据对应的物理地址。

#### (9) ioctl: RK\_FBIOSET\_OVERLAY\_STATE

该系统调用用于交换 win0、win1 对应的 zorder，在 RK LCDC 中，win0、win1 的位置是可以交换的，默认 win0 在 win1 的下面，如果需要，可以通过该系统调用交换 win0，win1 的位置。该系统调用传递一个指向 int ovl 的指针，如果 ovl = 1，则讲 win0 翻转到 win1 的上面，如果 ovl =

0, 则讲 win0 置于 win1 的下面。

(10) ioctl: RK\_FBIOGET\_OVERLAY\_STATE

该系统调用用于后去 win0、win1 的位置信息, 它返回一个指向 int ovl 的指针, 如果 ovl = 1, 说明 win0 在 win1 的上面, 如果 ovl = 0 说明 win0 在 win1 的下面。

(11) ioctl: RK\_FBIOSET\_ENABLE

该系统调用用于 enable、disable fb 对应的 win, 现在的 fb 驱动在 open fb 设备的时候, 会自动 enable 对应的 win, 但是在 close fb 设备的时候, 并没有去 disable 对应的 win, 所以如果要 disable fb 对应的 win, 需要通过该系统调用进行。该系统调用传递一个指向 int enable 的指针, 如果 enable = 0 关闭对应的 fb 设备, 如果 enable = 1, 打开对应的 fb 设备。

(12) ioctl: RK\_FBIOGET\_ENABLE:

该系统调用用于返回 fb 对应的 win 的 enable/disable 状态, 该系统调用返回一个指向 int enable 的指针, 如果 enable = 0 说明该 win 是关闭的, 如果 enable = 1 说明该 win 是打开的。

(13) ioctl: RK\_FBIOSET\_VSYNC\_ENABLE

Android4.1 以后, 默认需要 fb 驱动向上报 vsync 信息, 才会进行渲染, 该 ioctl 用于 enable 或者 disable 内核的 vsync 的上报功能。该系统调用传递一个指向 int enable 的指针, 如果 enable = 1 打开 vsync 上报功能, 如果 enable = 0, 关闭 vsync 上报功能。

(14) ioctl: RK\_FBIOPUT\_COLOR\_KEY\_CFG

该系统调用用于配置 lcdc 的 color key 功能, 该系统调用传递一个指向 struct color\_key\_cfg clr\_key\_cfg 的指针, 该结构中每一个变量对应一个 win 的 color key 配置字。其中 bit24 = 1 打开对应的 color key 功能, bit0~bit23 设置 color 的值。

(15) ioctl: RK\_FBIOSET\_CONFIG\_DONE

对于使用多个 win 进行 overlay 合成显示的应用, 为了保证各个 layer 在配置上的同步, 我们加了该 ioctl, 在相关的配置完成后, 只有调用该 ioctl, 相关配置才会生效。同时该 ioctl 还专递一个指向 int wait\_fs 的指针, 如果 wait\_fs = 1, 则该系统调用会等待 lcdc 开始以本次配置进行刷新后返回, 如果 wait\_fs = 0, 则直接返回。Android 4.1 以后, 推荐的相关同步都在 android 中实现, 因而默认 wait\_fs = 0。在内核中等待会降低显示的流畅度。



### 3 相关系统调用的流程

在用户空间的相关系统调用先通过 linux 系统的处理再传递给 rk fb 驱动，再传递给对应的 lcdc 驱动，流程图如下：

```
User system call ---->fbmem.c ----->rk_fb.c ---->rkxxx_lcdc.c
```

在开发应用程序的时候，如果遇到了异常，可以根据该调用流程进行相关的 debug 跟踪。最常用的 open、ioctl 调用流程如下：

```
Open---->fb_open(fbmem.c)--->rk_fb_open(rk_fb.c)--->
rkxxx_lcdc_open(rkxxx_lcdc.c)

Ioctl--->fb_ioctl(fbmem.c)--->do_fb_ioctl(fbmem.c)-->
rk_fb_ioctl(rk_fb.c)--->rkxxx_lcdc_ioctl(rkxxx_lcdc.c)
```

Lcdc 驱动中做了相关的开关，可以动态的打开关闭对应的调试信息用来查看自己的设置是否正常。

```
echo x > sys/module/rk30_lcdc/parameters/dbg_thresd
```

该打印是分级别的，x=0 不打印，x=1 打印部分关键信息，x=2 打印更多信息。同时可以 cat 如下节点获取更加详细的配置显示信息

```
cat sys/class/graphics/fb0/disp_info
```

```
root@android:/ # cat sys/class/graphics/fb0/disp_info
cat sys/class/graphics/fb0/disp_info
win0:enabled
xvir:2048
xact:2048
yact:1440
xdsp:2048
ydsp:1440
x_st:505
y_st:10
x_scale:1.0
y_scale:1.0
format:ARGB8888
YRGB buffer addr:0x8f7c0000
GBR buffer addr:0x8fa90000

win1:enabled
xvir:1024
xdsp:2048
ydsp:96
x_st:505
y_st:1450
format:RGB565
YRGB buffer addr:0x96380000
overlay:win1 on the top of win0
```

## A8 Q&A

(1)、新设计的一块板子，系统起来后，屏幕没有显示，该怎么排查？

背光有没有亮？如果背光没亮，检查背光电源是否正常，PWM 通道配置是否正确。

从 LCDC 出来能否测到相关信号（DCLK、数据信号）？

LCD 屏幕相关的供电是否正常？

LCD 屏幕相关的参数是否严格按照屏幕的 datasheet 配置？

(2)、系统起来后，某些界面显示有斑点、噪点？

尝试反转 DCLK 时钟，如果没有效果，请硬件协助检查相关信号质量。

(3)、系统起来后，在主界面滑动不流畅，低于 45fps 以下？

在串口终端下执行 `cat sys/class/graphics/fb0/fps` 查询 LCDC 的刷新频率，看是否接近 60fps，如果相差很远，请核对设置的 DCLK 是否和屏幕的 datasheet 一致。

是否是高清屏幕（屏幕分辨率高于 1920\*1080），如果是，请确保 DDR 频率在 533MHZ，另外打开 Android 层的高清优化开关。

（4）、用 BizlineAdjust\_30XX apk 做 gamma 调整，发现屏幕没有任何变化。

是否有修改 sys/class/graphics/fb0/dsp\_lut 节点的权限，使得上层 apk 可对改节点进行读写操作。具体请参考第五章节《利用 LUT 功能改善显示效果》

（5）、用 BizlineAdjust\_30XX apk 做 gamma 调整，发现一调整屏幕就花掉了。

查看内核 log，是否有打印 “no buffer to backup lut data”，如果是，请检查屏幕配置文件，确认是否有定义 USE\_RK\_DSP\_LUT 宏，和 dsp\_lut 表。具体请参考第五章节《利用 LUT 功能改善显示效果》。