密级状态：绝密(　　)　　秘密(　　)　　内部(　)　　公开(√)

RockChip_LCD_开发文档

（MID）

文件状态： [　] 正在修改 [√] 正式发布	当前版本：	V1.6
	作　　者：	Yxj、lb，hhb
	完成日期：	2013-07-22
	审　　核：	XXX
	完成日期：	201X-XX-XX

说明：该文档针对 RK30XX 、RK2928、 RK31XX 以后的 FB 以及 LCDC 驱动框架

福州瑞芯微电子有限公司

Fuzhou　Rockchips　Semiconductor　Co . , Ltd

版 本 历 史

版本号	作者	修改日期	修改说明	备注

目录

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。

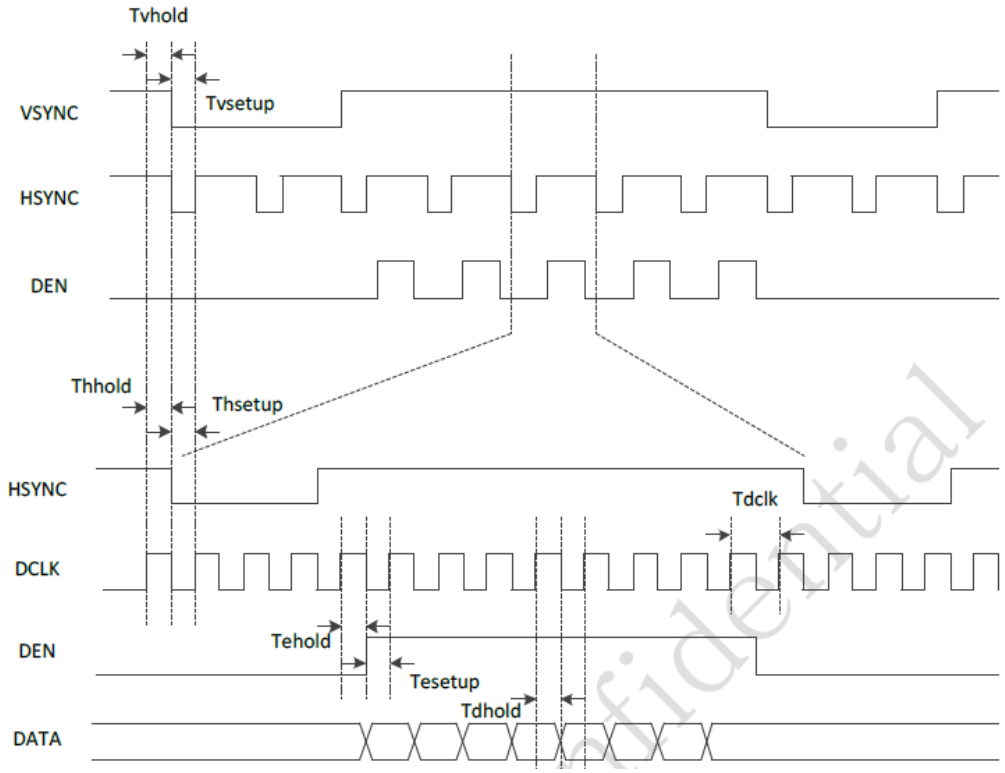
类别	TN	STN	TFT
原理	液晶分子，扭转 90 度	扭转 180~270 度	液晶分子，扭转 90 度
特性	黑白，单色，低对比	黑白，彩色，低对比	彩色，高对比
动画显示	否	否	是
视角	30° 以下	40° 以下	80° 以下
面板尺寸	1~3 英寸	1~12 英寸	37 英寸

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。

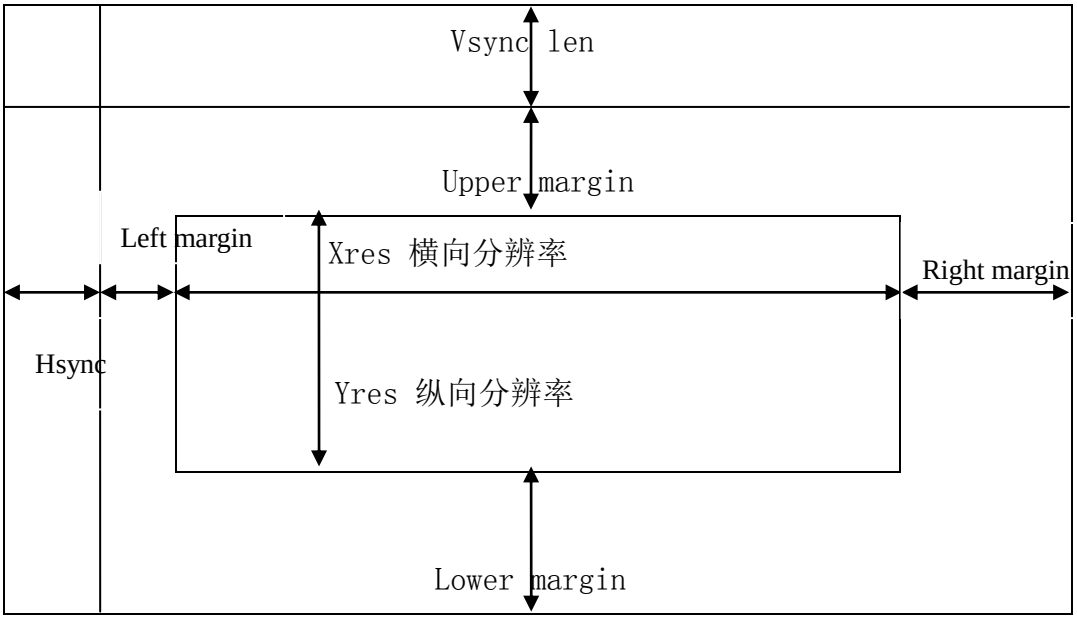


图（1）行场控制的 LCD 时序图

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	Tcwh	40	50	60	%	
DE	Setup Time	Tesu	8	-	-	ns	
	Hold time	Tehd	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	Thst	8	-	-	ns	
	HSYNC Hold Time	Thhd	8	-	-	ns	
	VSYNC Setup Time	Tvst	8	-	-	ns	
	VSYNC Hold Time	Tvhd	8	-	-	ns	
	Horizontal Period	th	1026	1056	1086	t_{CLK}	
	Horizontal Pulse Width	thpw	-	30	-	t_{CLK}	thb + thpw=46DCLK is fixed
	Horizontal Back Porch	thb	-	16	-	t_{CLK}	
	Horizontal Front Porch	thfp	180	210	240	t_{CLK}	
	Horizontal Valid	thd		800		t_{CLK}	
	Vertical Period	tv	515	525	535	th	
	Vertical Pulse Width	tvpw	-	13	-	th	tvpw + tvb = 23th is fixed
	Vertical Back Porch	tvb	-	10	-	th	
	Vertical Front Porch	tvfp	12	22	32	th	
	Vertical Valid	tvd		480		th	

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/(28.37516E6 Hz) = 35.242E-9 s

如果屏幕的分辨率是 640×480，显示一行需要的时间是：

640*35.242E-9 s = 22.555E-6 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)*35.242E-9 s = 32.141E-6 s

可以计算出水平扫描率大约是 31kHz：

1/(32.141E-6 s) = 31.113E3 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 个象素时钟，因此，画一个完整的屏幕的时间是：

(480+49)*32.141E-6 s = 17.002E-3 s

可以计算出垂直扫描率大约是 59kHz：

1/(17.002E-3 s) = 58.815 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：

刷新率 =dotclock/((xres+left_margin+right_margin+hsync) *(yres+upper_margin+low_margin+vsync))

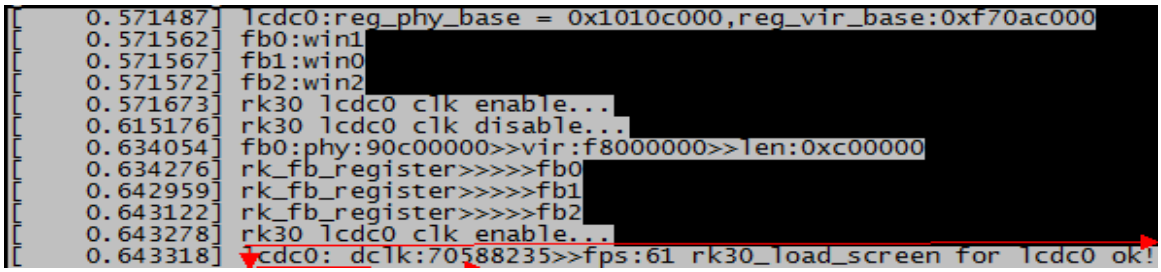
在 linux frame buffer 子系统中，还有用到一个参数---像素时钟即 pixclock。

Pixclock = 1/dotclock

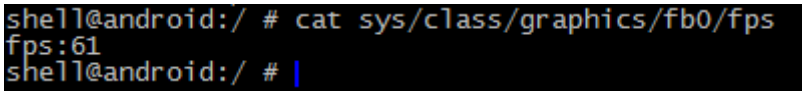
对于 LCDC 驱动来说，就是要根据屏幕的这些时序参数（DCLK、HSYNC、HBP、HVD、HFP、VSYNC、VBP、VVD、VFP）送出符合要求的信号。

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，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，是的 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：



或者通过 fb 的 sys 节点查看：



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
include/linux/rk_screen.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、RK292X 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

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

drivers/video/rockchip/screeen/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：

set_scaler_info: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。

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，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。

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。

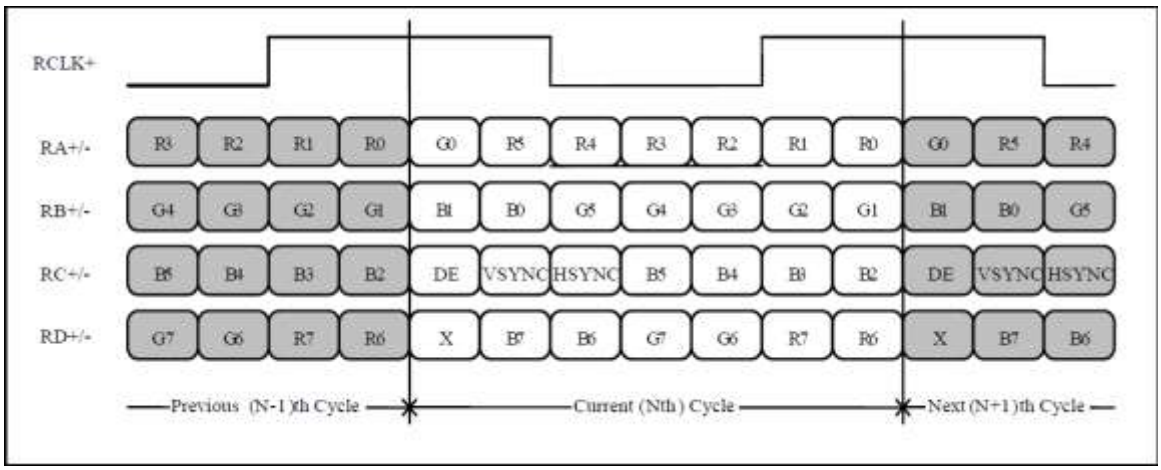
这部分代码是根据 LCD data sheet 设置的和 LCD 屏幕相关的配置参数，这也是产品端在开发的过程中修改最多的代码，下面对里面涉及的内容详细讲解。以 lcd_b101ew05.c 为例，该驱动为 RK3066X SDK 板所使用的屏幕所对应的驱动。
有如下宏比较重要：

```
#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 rk2928，rk292x 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，具体请参考<<RK610 配置_V1.2.pdf>>。LVDS 的屏幕的 datasheet 也会说明信号在 LVDS 各个通道上的分配方式。比如图（4）：

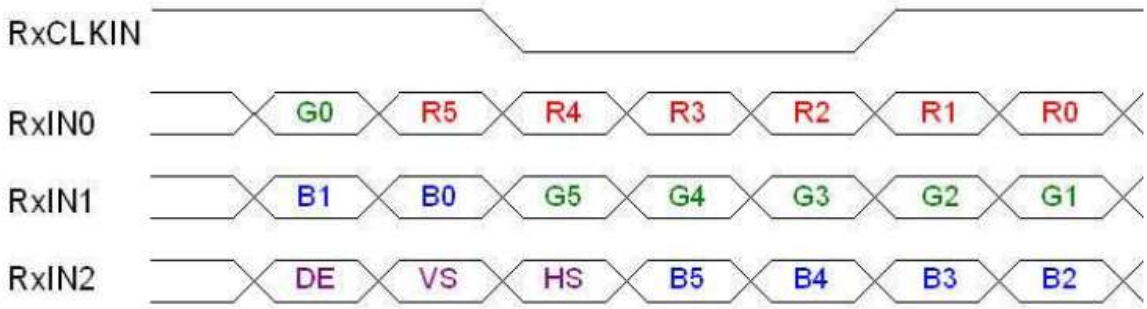


图（4）LVDS data format

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》。

这里面有一点需要说明，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 查分通道,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_P888_D666,LVDS_FORMAT 定义为 LVDS_8BIT_2，否则在 HDMI 模式下，屏幕显示会异常。

OUT_CLK 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 OUT_CLK 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_lcdc1     0 Hz usecount = 0 parent = pd_vio
    pd_lcdc0     0 Hz usecount = 1 parent = pd_vio
        dclk_lcdc1  off 74.250000 MHz usecount = 0 parent = general_pll
                    hclk_lcdc1   off 148.500000 MHz usecount = 0 parent = hclk_cpu
                    hclk_lcdc0   on  148.500000 MHz usecount = 1 parent = hclk_cpu
        dclk_lcdc0  on  70.588235 MHz usecount = 1 parent = codec_pll
        aclk_lcdc1_pre on  300 MHz usecount = 1 parent = codec_pll
            aclk_vio1   on  300 MHz usecount = 1 parent = aclk_lcdc1_pre
                aclk_lcdc1  off 300 MHz usecount = 0 parent = aclk_vio1
        aclk_lcdc0_pre on  300 MHz usecount = 1 parent = codec_pll
            aclk_vio0   on  300 MHz usecount = 1 parent = aclk_lcdc0_pre
                aclk_lcdc0  on  300 MHz usecount = 1 parent = aclk_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。为了兼容 rk29 的 fb 驱动，这边不去掉。

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 信号进行调换，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那种颜色反了，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 scale 参数配置。具体请参考<<RK610 配置_V1.2.pdf>>和《Rockchip RK616（JettaB）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 板子上使用的 LVDS 接口的屏幕，支持 RK2918 和 RK2928 的 LVDS 接口。

2.4 LCD 电源相关的操作的配置以及板级配置

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,
    },

    [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_lcdc1,
    .dev        = {
        .platform_data = &lcdc1_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 系统初始化的时候调用。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。这些函数要针对具体的项目进行微调，有的板子，由于硬件的设计，可能在开关相关电源的前后要进行延时，否则会有白屏或者闪烁的可能。

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。如果只使用 RK610 和一个 LCDC 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 封装和 RK2928 Pin 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

#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);
    }
```

2.5 make menuconfig 相关配置

```
          Device Drivers  --->
            Graphics support  --->
            <*>Frame buffer support for Rockchip --->
            [  ]Mirroring Support
               Dual display ploy 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 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 ploy 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 + rk610

() NO dual display needed

该配置只支持单边显示，例如 rk2926

需要强调的一点是，由于 Jetta 的 Scaler 不支持旋转功能，所以如果屏幕是竖屏，或者屏幕的磨具和硬件做反了，需要进行 180 度的旋转，就不能使用一个 LCDC+jetta 的 Scaler 实现双显，这种情况只能用双 LCDC 来实现双显。

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。

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 屏幕显示方向反调的项目提供的接口，只有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 meuconfig 里面选择 FB rotatesupport 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。

需要注意的是，该方法只适用于 rk30xx 等用两个 lcdc 进行双显的方案。

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。

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

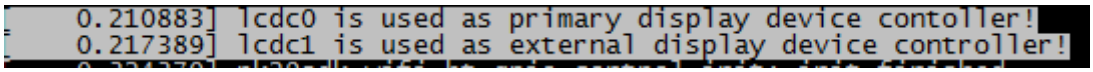
e.[ ]RockChip Display transmitter support

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。



该 log 说明 LCDC0 是用作主显示设备，LCDC1 用做扩展显示设备，给你 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

附录：

1 LVDS 、EDP、MIPI 显示接口转换芯片及其驱动

从接口上面区分，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）有使用 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 内核集成有 LVDS 模块，它既可以输出 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 RK610 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 芯片有两颗，一颗是 RK610，一颗是 rk292x 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 support
```

2 MIPI 屏幕的使用

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

2.1 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

2.2 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
```

2.3 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(ddr 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;
dcs[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。

3 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 -l 参数后面跟的是写入数据的长度，这里都是满屏写入，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 的基地址为 0x1010e000

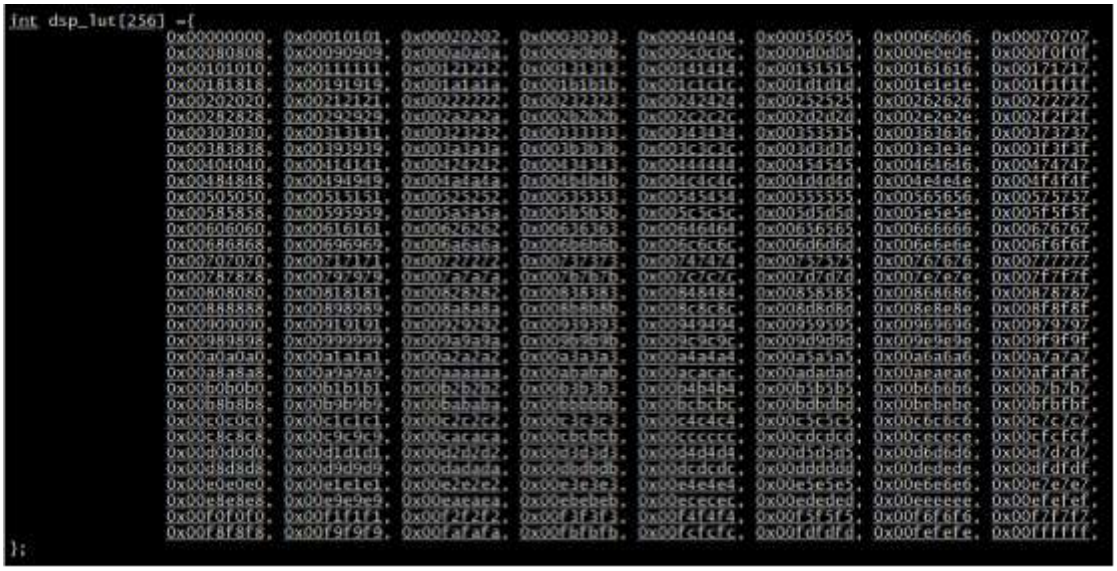
```
lcdc0:reg_phy_base = 0x1010e000,reg_vir_base:0xf700c000
fb0:win0
fb1:win1
fb2:win2
rk_output_lvtt1>>connect to lcdc output interface0
RGB screen connect to rk2928 lcdc interface0
lcdc0: dclk:33000000>>fps:59 rk2928_load_screen for lcdc0 ok!
fb0:phy:90c00000>>vir:f8000000>>len:0xc00000
```

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

4 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
kernel/drivers/video/display/screen/lcd_b101ew05.c



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 DisplayAdjust.apk 能够对这个节点进行写入。
（3）安装 DisplayAdjust.apk ，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。

5 基于 rk fb 驱动的应用开发

5.1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。

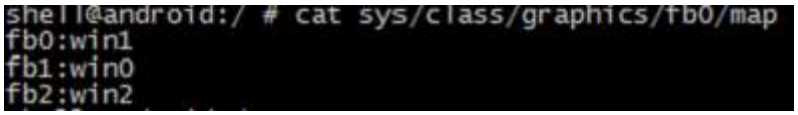
在 rk fb 中，每一层 win 对应一个 fb 设备，它们在 linux 系统中对应的设备节点为/dev/graphics/fbx,其中 win 和 fb 的对应关系是可以通过程序设置的，以 RK3066 为例：

/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：



5.2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_screeen_info 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:每个像素站多少 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，yszie），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：

grayscale = (xsize<<8) | (ysize<<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：

nonstd = （xpos<<8）| (ypos<<20) | (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次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_diaplay 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 至于 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 enbale 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。

5.3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:ARGB888
YRGB buffer addr:0x8f7c0000
CBR 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
```

6 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。