基于 framebuffer 框架的 VOP 驱动开发

黄家钗

（福州瑞芯微电子股份有限公司，福建省福州市铜盘路软件大道 89 号软件园 A 区 21 号楼，350003）

（Fuzhou Rockchip Electronics Co.Ltd - Graphics & Display Dept., No. 21 Building, A District, No.89,software Boulevard Fuzhou,Fujian,PRC, 350003）

摘要

显示设备作为人机交互类系统中不可或缺的重要组成部分，随着人们消费水平的提高，对显示效果的要求也越来越高，这也推动着显示技术的不断创新，GPU 的图像处理能力，VOP 控制器(也称 LCDC 控制器)合成和输出分辨率的能力，液晶面板工艺的提升对显示分辨率提高和屏响应时间改善，高速信号传输稳定性等等，这些技术的改进都促进了显示效果的提升。

关键词： GPU; VOP(Video Out Process)控制器;分辨率;高速信号;

Abstract

Display device as the important part in human-computer interaction, as people consumption levelrising,more and more highly requirement to display device, and this promotes innovation of the displaytechnology, the ability of GPU process and VOP control(or called LCDC control) output resolution, the LCD panel reponse time, high speed signal transmission ability and so on. All of these technical improvements are promoted the display effect.

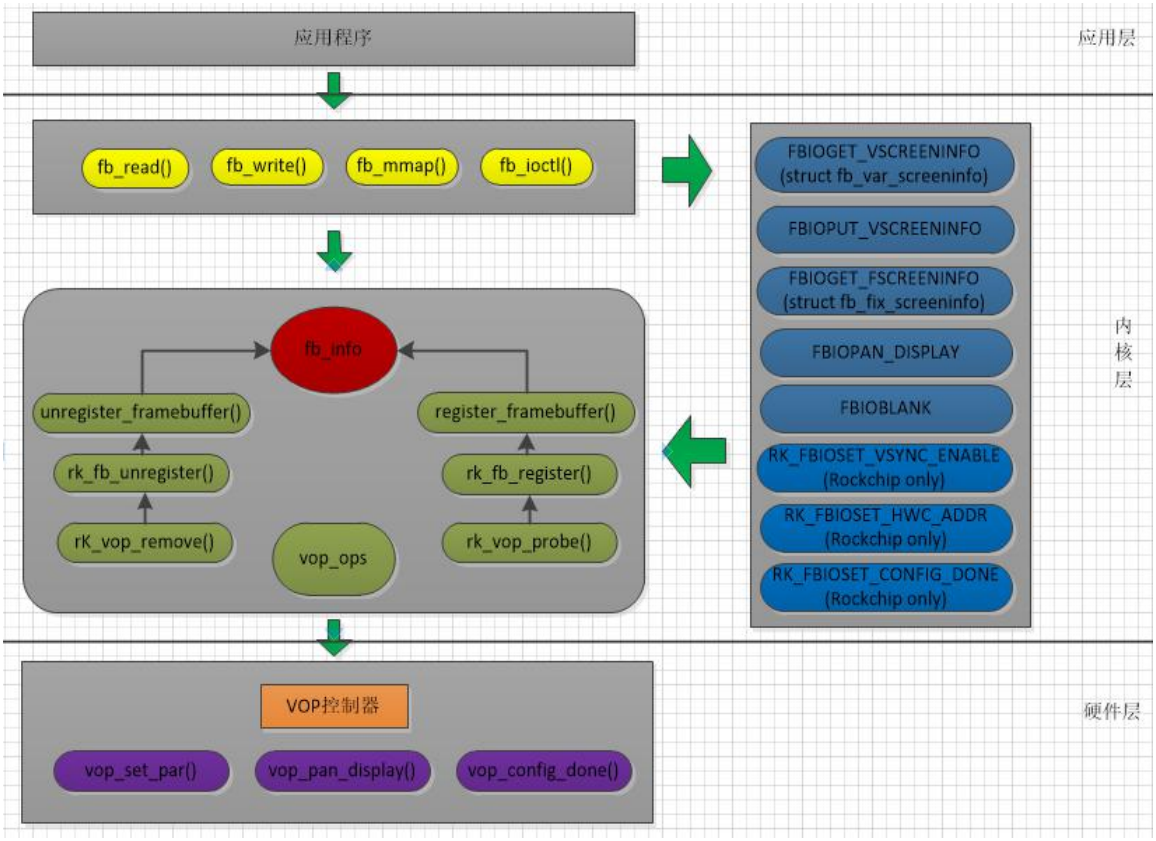
Keywords: GPU;VOP control;resolution; high speed signal;

引言

Framebuffer 是 Linux 下 LCD 对应的硬件抽象层，应用程序通过 framebuffer 提供的统一接口实现对图像地址和图像显示信息的设置，进而实现对 LCD 显示内容的配置；VOP 作为 Framebuffer 显示框架中承上启下的硬件设备，实现对图像数据的合成、缩放、色域转换、时序输出等功能。

一、Linux Framebuffer 显示框架

1.1 显示框架



1.2 驱动注册与实现

Framebuffer 框架的驱动文件主要有：

linux/include/linux/fb.h

linux/drivers/video/fbmem.c

Rockchip 抽象出来的驱动文件主要有：

include/linux/rk_fb.h

drivers/video/rockchip/rk_fb.c

下面主要通过上面几个文件的函数接口和结构体分析实现 framebuffer 驱动要做的工作:

1. struct fb_var_screeninfo

这个结构体是应用程序和内核交互的主要接口，它定义了在系统运行期间可以改变的一些信息。例如像素深度、灰度级、虚宽、颜色格式、时序、屏幕边缘空白区、分辨率等,应用程序通过 IOCTL 发送 FBIOGET_VSCREENINFO 获得内核的信息，也可以通过 IOCTL 发送 FBIOPUT_VSCREENINFO 更新内核的配置信息。

2. struct fb_fix_screeninfo

该数据结构定义了一些系统运行期间不能被修改的信息，例如设备、屏幕的像素数量、缓冲区的首址和长度等，应用程序通过 IOCTL 发送 FBIOGET_FSCREENINF 获得底层硬件的一些信息。

3. struct fb_ops

这里面的函数指针在 fbmem.c 里用来支持应用层各种复杂的 IOCTL，这些函数的具体实现都在文件 rk_fb.c 里。

4. struct fb_info

这个是只有在内核中可见的结构体，一个 fb_info 就代表一个显示图层，他包括了图层所有显示相关的信息以及提供给应用层操作的接口。

5. register_framebuffer()/unregister_framebuffer()

这两个是 framebuffer 框架层 fbmem.c 提供给设备层向系统注册和注销直接的函数接口。在 rk_fb.c 的驱动，在填充好 fb_info 结构体后调用两个接口来注册和注销设备。

6. struct file_operations

提供了给应用层读写，内存映射，ioctl 等系统调用的接口。

1.3 应用使用参考

从应用层的角度看，framebuffer 其实就是一个普通的文件，应用层可以像对待普通文件那样对 framebuffer 进行读写操作；通过打开/dev/graphics/fb0 获得设备节点，调用 mmap()接口将 framebuffer 预留的内存映射到用户空间的虚拟地址，还可以通过 ioctl()读取或者设置参数等等。

下面是应用处理 framebuffer 的处理流程：

1. 打开设备节点/dev/graphics/fb0：

   fd = open("/dev/graphics/fb0", O_RDWR, 0)。

2. 获得硬件的相关信息：

   ioctl(fd,FBIOGET_VSCREENINFO,&fb_var)。

   ioctl(fd,FBIOGET_FSCREENINFO,&fb_fix)。

3. 将 framebuffer 预留的内存映射到内存空间：

   mmap(0, fi.smem_len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)。

4. 配置显示相关信息：

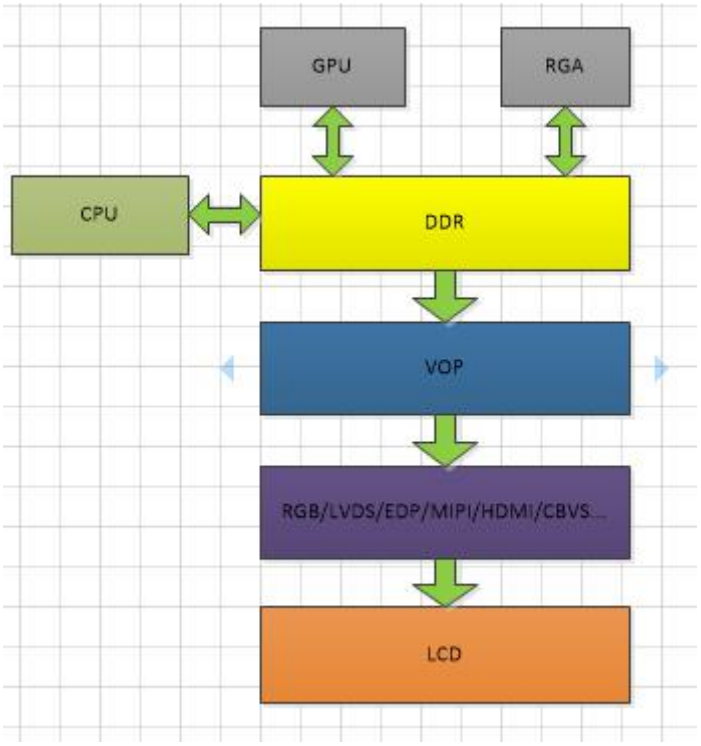
ioctl(fd,FBIOPUT_VSCREENINFO,&fb_var)。

5. overlay 配置扩展，可以同时配置多个图层的显示相关信息：

ioctl(fd,RK_FBIOSET_CONFIG_DONE,&rk_fb_info)。

二、VOP 显示模块

2.1 VOP 在 SOC 中位置



2.2 硬件处理流程和模块说明:

VOP 通过 AXI 总线从 DDR 实时取数据，VOP 把每个图层取到的数据做缩放，叠加，alpha 后通过好 RGB 数据接口和行场同步信号送个转换 IC 或者屏，下面是 VOP 内部各个小模块的功能说明：

1. AXI:用于从 DDR 搬运数据的总线。

2. AHB:用于配置 VOP 寄存器的总线。

3. IEP DPI:用于将 VPU 解码出来的数据直接传输到 VOP 的总线。

4. VOP_IFBDC:用于对 GPU 压缩好的数据做解压缩的模块。

5. BG/win0/win1/win2/win3/hwc: 处理各个显示模块的硬件图层。

6. BCSH:亮度，对比度，饱和度的调节。

7. gamma: 可以针对不同 gamma 值的屏做校正。

8. CABC:自动背光控制模块。

2.3 驱动实现功能

VOP 驱动的实现主要是对 framebuffer 驱动框架下对一些接口的具体实现，主要有：

1. vop_probe()和 vop_remove()

用于对硬件的资源信息的处理和向系统注册或者注销 vop 设备；

2. vop_set_par()

控制图层的源数据的大小、显示目标的宽高、CSC 通路的配置、缩放算法的选择和倍数的计算等；

3. vop_pan_display()

控制每一个显示图层的地址信息；

4. vop_early_resume()和 vop_early_suspend()

开关硬件模块改善设备休眠状态下的功耗；

5. vop_ovl_mgr()

用于控制多个图层的叠加顺序和 alpha 处理；

2.4 高速信号转换模块的特殊需求

随着分辨率的越来越高，高速信号传输需求的时钟频率也越来越高，在 HDMI 2.0 出来后的 4k@60hz 分辨率要求每条 lane 的带宽为：4096*2160*3Byte*60fps*8bit/4lane = 3Gbps/lane 这么高的传输频率对硬件的要求也越来越高，为了满足这种高分辨率的需求同时又可以降低对硬件布板的要求，VOP 需要支持输出 YUV420 的数据，在这种场景下，VOP 需要将 RGB 的数据通过 CSC 模块转成 YUV420，在这样对于 4k@60hz 的分辨率在每条 lane 上的带宽需要为：4096*2160*1.5Byte*60fps*8bit/4lane = 1.5Gbps/lane,VOP 驱动的具体实现为：vop_win_csc_mode();

三、驱动优化

1. 引入 fence 同步机制；

在传统的同步机制里，由于 cpu 不知道 GPU 什么时候渲染好一块 bufefr，cpu 也不了解送给 VOP 的 buffer 是否使用完，所以使用同步的机制等 gpu 画好后在送给 VOP，发送给 VOP 显示的 buffer 也是等当前 buffer 生效后才去释放前一帧的 buffer，这种同步机制虽然可靠，但是代价太大，在 gpu 遇到大数据处理的时候往往由于 cpu 的等待导致帧率降低。fence 同步机制里所有的等待都是异步的，CPU 在向 GPU 发送万命令后直接将 buffer 丢给 VOP 去显示，VOP 返回来的 buffer 也可以直接发送给需要 buffer 的模块，也就是说 CPU 不会去等待 GPU 画完或者 VOP

使用完，这样异步的处理，可以让 GPU 和 VOP 的性能得到最大的使用。

2. config done；

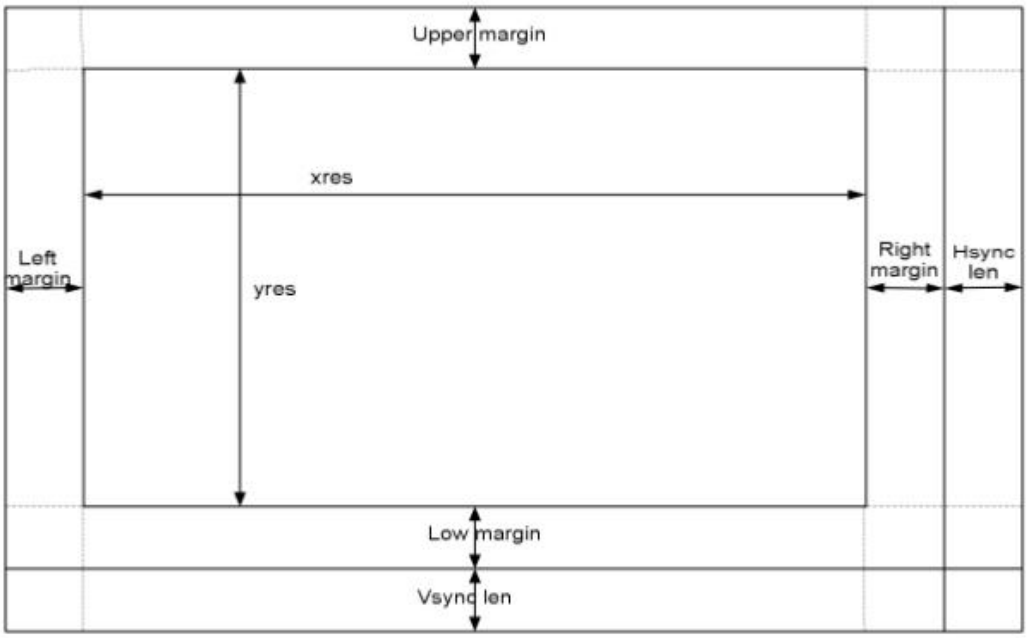
framebuffer 传统的接口使用 IOCTL 发送 FBIOPUT_VSCREENINFO 设置图层的配置已经无法满足多个图层的配置需求，为此，我们在 IOCTL 增加了的一个扩展接口 RK_FBIOSET_CONFIG_DONE 将所有图层的配置以及同步信息一次性配置给内核，这样可以有效的保证相关图层的配置会在同一帧生效；

3. DPI for hwc

对于 android 系统大多数的场景 60fps 的帧率已经可以满足人们的肉眼的需求，但是对于使用鼠标做快速滑动的情况下如果只有 60fps 会让人感觉鼠标移动的速度跟不上手势移动的速度， 所以我们开通了一个鼠标层的快速通道(DPI, Direct Path Interface)，可以将鼠标位置信息的变换实时得反馈到屏上。

4. 动态帧率控制

对于固定分辨率的屏来说，帧率越高，VOP 对总线的数据访问量就越大，GPU 工作负荷也越大，这样功耗也越多。所以为了配合消费类电子设备对低功耗的需求，当检测到应用刷新率变低的时候，通过降低 VOP 工作的时钟频率，来降低帧率，进而降低系统的功耗。为了不影响显示效果，必须要求做到无缝切换，所以我们选择在 VFP+VSYNC+VBP 阶段做时钟的和帧率的切换,下面是 VOP 控制输出的时序图：



四、总结和展望

从进入公司以来，前后负责了 RK3026/RK3288/RK3368/RK3366/RK3399 等公司这几年主要芯片 VOP 模块的验证和驱动维护工作，自己从中学到了很多，同时也发现了自己很多的不足，随着人们需求的提升，显示效果的改

善是一个永远做不完的工作。感谢公司提供的平台和机会，感谢同事、家人和朋友的支持。