Linux Framebuffer驱动剖析之二—驱动框架、接口实现和使用

yueqian_scut
博客专家

原创 2015年12月29日 22:16:48

标签：framebuffer / linux设备驱动 / linux驱动框架 / 字符设备驱动 / LCD驱动 / 3874

本文继上一篇文章《Linux Framebuffer驱动剖析之一—软件需求》，深入分析LinuxFramebuffer子系统的驱动框架、接口实现和使用。

一、LinuxFramebuffer的软件需求

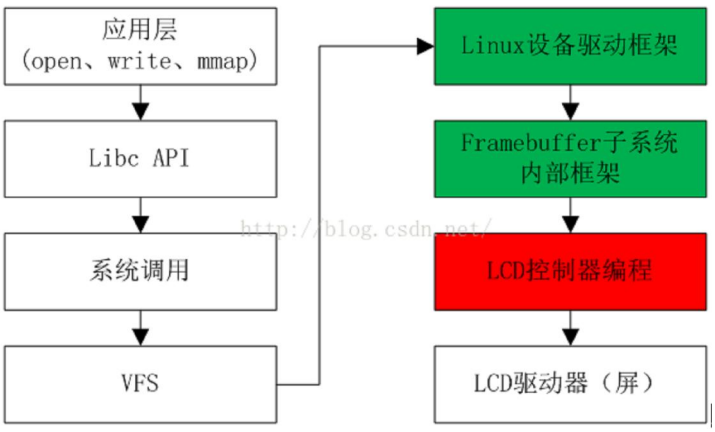
上一篇文章详细阐述了LinuxFramebuffer的软件需求（请先理解第一篇文章再来阅读本篇文章），总结如下：

1. 针对SOC的LCD控制寄存器进行编程，以支持不同的LCD屏，以使该SOC的应用场景最大化。这是硬件平台相关的需求。其对应Linux源码路径arch\arm\mach-s5pv210\XXX210-lcds.c中的实现内容。

2. 给用户提供一个进程空间映射到实际的显示物理内存的接口(mmap)，以使应用在一次拷贝的情况下即可以将图像资源显示到LCD屏幕上。这需要SOC的MMU（内存管理单元）、Linux操作系统的内存管理、SOC的SDRAM（内存）三者支持。由于内存管理作为操作系统的公共组成部分，给其他子系统提供了独立的接口。所以我们可以认为framebuffer通过调用内存管理接口来实现映射的接口属于非平台相关的需求。

3.Framebuffer支持32个显示缓存。其在内部进行了抽象，即其向上层应用统一抽象为一个字符主设备，而不同的显示缓存即视为不同的字符从设备。对显示从设备的管理属于Framebuffer内部框架的功能。

4. Linux设备驱动支持多种类型的设备驱动，除了Framebuffer，还包括input、USB、watchdog、MTD等等，这种不同类型的设备一般都表述为不同的主设备。管理不同的主设备是由Linux设备驱动框架来完成的。另外，由于Linux把设备也认为是一个文件，因此设备驱动之上还架设了一层虚拟文件系统（VFS），因此，实际上，应用层是跟framebuffer对应的VFS接口进行交互的。如下图:



对于驱动开发人员来说，其实只需要针对具体的硬件平台SOC和具体的LCD（焊接连接到该SOC的引脚上）来进行第一部分的寄存器编程(红色部分)。而第二、三、四部分内容（绿色部分）已经被抽象并实现在Linux driver发布源码中了，LCD驱动开发人员只需要理解framebuffer内部的框架和接口使用即可。其实，对于其他的设备驱动，包括按键、触摸屏、SD卡、usb等等，Linuxdriver都已经实现了大部分非平台相关的需求任务了，开发人员同样是只需要理解所属驱动的内部框架和接口使用即可。

接下来，我们就详细分析LinuxFramebuffer如何实现支持第二和第三点需求的。其对应driver\video\fbmem.c等实现内容。先分析第三点驱动框架，再分析映射接口。

二、LinuxFramebuffer的内部驱动框架

1. 初始化

--driver\video\fbmem.c

```
static int __init
fbmem_init(void)
{
        proc_create("fb", 0, NULL, &fb_proc_fops);
        register_chrdev(FB_MAJOR,"fb",&fb_fops); //FB_MAJOR=29
        fb_class = class_create(THIS_MODULE, "graphics");
```

向proc文件系统报告驱动状态和参数

注册字符设备驱动，主设备号是29

创建/sys/class/graphics设备类，配合mdev生成设备文件

Framebuffer作为一个子系统，在fbmem_init中通过register_chrdev接口向系统注册一个主设备号位29的字符设备驱动。通过class_create创建graphics设备类，配合mdev机制生成供用户访问的设备文件（位于/dev目录）。设备文件和mdev机制详见《 Linux设备文件的创建和mdev》。

Framebuffer设备驱动的接口集fb_fops的定义为：

```
static const struct file_operations fb_fops = {
        .owner =        THIS_MODULE,
        .read =         fb_read,
        .write =        fb_write, 二次拷贝
        .unlocked_ioctl = fb_ioctl,
        .mmap =         fb_mmap, 映射，一次拷贝
        .open =         fb_open,
        .release =      fb_release,
};
```

在linux设备驱动中，所有的显示缓存设备均由framebuffer子系统内部管理，即linux设备驱动框架只认识一个主设备号为29的framebuffer设备。应用层所有针对显示缓存（最多32个）的访问均会推送给fb_fops进行进一步分发操作。

2. register_framebuffer

单个显示缓存视为一个framebuffer从设备，其在驱动加载初始化时需要通过register_framebuffer接口向framebuffer子系统注册自己。这样，当应用层要访问该从设备时，才能通过framebuffer子系统进行操作管理分发。我们跟踪一下：

```
register_framebuffer(struct fb_info *fb_info)
{
        int i;
        struct fb_event event;
        struct fb_videomode mode;

        num_registered_fb++;
        for (i = 0 ; i < FB_MAX; i++)
                if (!registered_fb[i])
                        break;
        fb_info->node = i;
        mutex_init(&fb_info->lock);
        mutex_init(&fb_info->mm_lock);
//在 sys/class/grapics/下 创 建 fbx设 备 ， 用 于 设 备 文 件 的 创 建
        fb_info->dev = device_create(fb_class, fb_info->device,
                                     MKDEV(FB_MAJOR, i), NULL, "fb%d", i);

        registered_fb[i] = fb_info;
```

每个从设备都需要传递一个fb_info的数据结构指针，其即代表单个显示缓存设备。从中，可以看到fb_info最终会存储到全局数组struct fb_info*registered_fb[FB_MAX]中，FB_MAX是32，从这里我们也可以看出，framebuffer最多支持32个从设备。另外，每个从设备注册还会在/sys/class/graphics/设备类中创建一个设备，最终由mdev在/dev/目录中生成对应的设备文件。假设M个从设备调用register_framebuffer接口，即会在/dev中生成M个设备文件，如/dev/fb0、/dev/fb1、/dev/fb2等等。这M个设备的主设备号都是29，从设备则是0、1、2等等。

3. fb_info结构体

fb_info结构体代表单个显示缓存从设备，在调用register_framebuffer接口之前，必须要初始化其中的重要数据成员。其定义如下：

```
struct fb_info {
    int node;//次 设 备 号
    int flags;
    struct mutex lock;          /* Lock for open/release/ioctl funcs */
    struct mutex mm_lock;       /* Lock for fb_mmap and smem_* fields */
    struct fb_var_screeninfo var; //LCD可 变 参 数 结 构 体
    struct fb_fix_screeninfo fix;  //LCD固 定 参 数 结 构 体
    struct fb_monspecs monspecs;    /* Current Monitor specs */
    struct work_struct queue;     //帧 缓 存 事 件 队 列
    struct fb_pixmap pixmap;      /* Image hardware mapper */
    struct fb_pixmap sprite;  /* Cursor hardware mapper */
    struct fb_cmap cmap;          /* Current cmap */
    struct list_head modelist;    /* mode list */
    struct fb_videomode *mode;  //显 示 模 式

    struct fb_ops *fbops; //LCD底 层 硬 件 操 作 接 口 集
    struct device *device;//设 备 驱 动 模 型
    struct device *dev;
    char __iomem *screen_base; //显 示 内 存 虚 拟 基 地 址 (内 核 态 )
    unsigned long screen_size;    //显 示 内 存 大 小
```

其中,fb_var_screeninfo和fb_fix_screeninfo两个结构体跟LCD硬件属性相关，fb_var_screeninfo代表可修改的LCD显示参数，如分辨率和像素比特数；fb_fix_screeninfo代表不可修改的LCD属性参数，如显示内存的物理地址和长度等。另外一个非常重要的成员是fb_ops，其是LCD底层硬件操作接口集。

fb_ops硬件操作接口集包含很多接口，如设置可变参数fb_set_par、设置颜色寄存器fb_setcolreg、清屏接口fb_blank、画位图接口fb_imageblit、内存映射fb_mmap等等。

fb_info结构体在调用register_framebuffer之前完成初始化。一般来说，LCD设备属于平台设备，其初始化是在平台设备驱动的probe接口完成。而LCD设备所涉及的硬件初始化（第一部分需求）则在平台设备初始化中完成。有关平台设备驱动的分析，笔者会另写一篇文章阐述。

4. fb_open接口

当一个framebuffer设备完成初始化时，其对应的fb_info结构体会在全局数组registered_fb中记录，并且位于跟从设备号相等的位置上；而且，在/dev目录下也会生成一个/dev/fbx的设备文件。以下分析假设是访问第一个framebuffer设备：

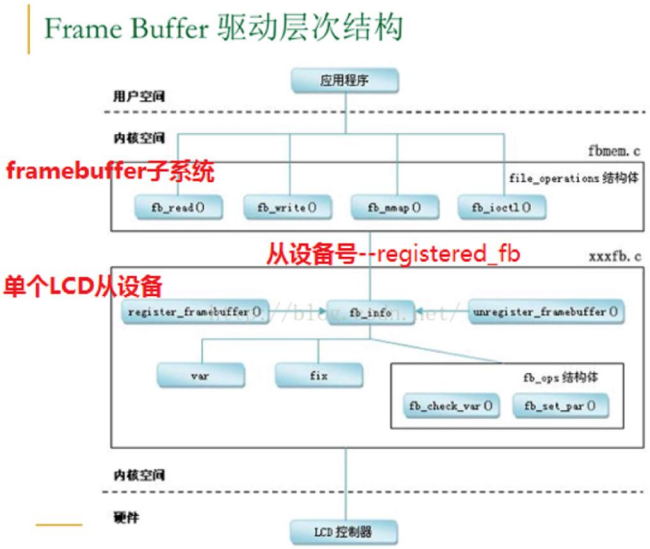
对于应用层open(“/dev/fb0”,...)访问该framebuffer设备来说，vfs先通过设备名（/dev/fb0）获得该设备的主设备（29）和从设备号（0）。而linux设备驱动框架则通过主设备29找到该设备对应的设备驱动接口集fb_fops。Linux驱动框架的分析过程请看《Linux字符设备驱动剖析》。接着linux设备驱动框架会调用fb_fops的fb_open，我们来跟踪一下：

```
fb_open(struct inode *inode, struct file *file)
__acquires(&info->lock)
__releases(&info->lock)
{
    int fbidx = iminor(inode);//获 得 次 设 备 号
    struct fb_info *info;//代 表 一 个 framebuffer的 全 局 数 据 结 构
    int res = 0;

    if (fbidx >= FB_MAX)
        return -ENODEV;
    info = registered_fb[fbidx];
    if (!info)
        request_module("fb%d", fbidx);
    info = registered_fb[fbidx];//找 到 对 应 的 fb_info

    file->private_data = info;
    if (info->fbops->fb_open) {//调 用 info->fbops->fb_open
        res = info->fbops->fb_open(info,1);
        if (res)
            module_put(info->fbops->owner);
    }
```

这个接口很简单，即是次设备号在全局数组registered_fb中找出对应的fb_info数据结构，将其设置到file指针的私有数据中，便于之后用户访问调用其他接口如mmap、ioctl等能够直接找到对应的fb_info。最后也会调用fb_info->fb_ops->fb_open，不过这个接口一般没干啥，赋值为NULL即可。

所以，framebuffer子系统内部驱动框架即负责通过从设备号找到对应的从设备（具体LCD操作接口所在的fb_info）。

5. 驱动框架总结

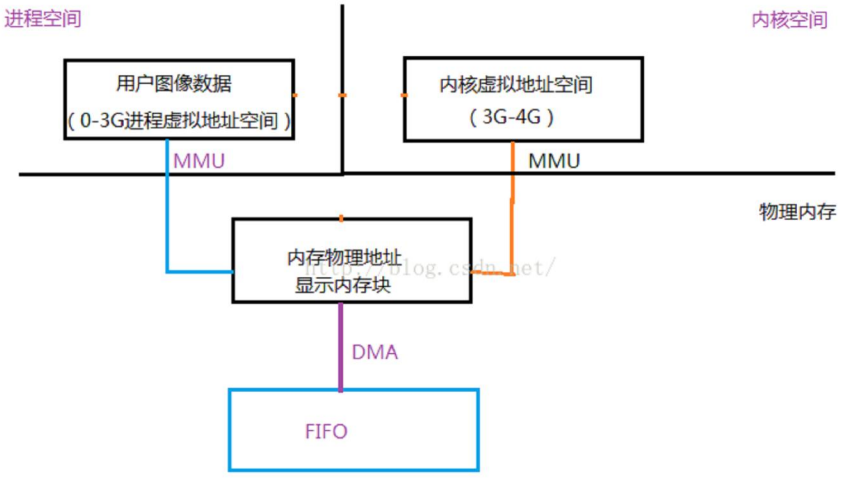


帧缓冲设备驱动程序结构图

经过上面的分析，这张图应该很容易理解了。

三、mmap映射

framebuffer驱动最重要的功能就是给用户提供一个进程空间映射到实际的显示物理内存的接口(mmap)。当用户图像数据buffer和内核虚拟地址空间buffer对应的都是同一块物理内存时，资源数据拷贝到用户图像数据buffer时，即是直接拷贝到显示物理内存了。



应用层mmap映射接口会经历以下层次调用：

1. sys_mmap

sys_mmap是虚拟文件系统层的映射实现，其会在用户进程虚拟空间（0到3G）申请一块虚拟内存，最终会调用到framebuffer子系统的fb_mmap，并向其传递vm_area_struct结构体指针，该结构体已经包括进程用户空间的内存地址信息。

Vfs虚拟文件系统是linux系统的重要组成部分，这里不再展开，以后再分析。

2. fb_mmap

来跟踪一下：

```
fb_mmap(struct file *file, struct vm_area_struct * vma)
{
        int fbidx = iminor(file->f_path.dentry->d_inode);
        struct fb_info *info = registered_fb[fbidx];
        struct fb_ops *fb = info->fbops;
        unsigned long off;
        unsigned long start;
        u32 len;

        if (vma->vm_pgoff > (~0UL >> PAGE_SHIFT))
                return -EINVAL;
        off = vma->vm_pgoff << PAGE_SHIFT;
        if (!fb)
                return -ENODEV;
        mutex_lock(&info->mm_lock);
        if (fb->fb_mmap) {
                int res;
                res = fb->fb_mmap(info, vma);
                mutex_unlock(&info->mm_lock);
                return res;
        }
```

该接口也是找到具体的从设备对应的LCD操作接口集，并调用fb_mmap接口。

我们选一个具体的LCD的接口实现看看：

```
static int lcd_drv_mmap(struct fb_info *info, struct vm_area_struct *vma)
{
    int ret;
    ret =  remap_pfn_range(vma,
        vma->vm_start,
        virt_to_phys((void*)((unsigned long)lcd_mem)) >> PAGE_SHIFT, //PAGE_SHIFT=12
        vma->vm_end-vma->vm_start,
        vma->vm_page_prot);
    if(ret) {
        return -EAGAIN;
    }

    return 0;
}
```

remap_pfn_range接口即是建立进程地址空间（0到3G）到实际的显示物理内存的映射。其中,lcd_mem是fb_info结构体初始化使用kzmalloc申请的，其代表内核态（3G到4G）的虚拟内存首地址。而virt_to_phys即是将虚拟地址转化为物理地址，更新vm_area_struct的数据成员，而其最终会影响进程的页表设置，进而影响MMU的设置。

当该接口完成之后，最终向应用层返回进程空间的内存首地址（0到3G）。这样，应用层即可以直接访问这块内存，进行读写操作。其直接通过MMU来访问实际的物理地址，而不需要再经过驱动的管理。

四、接口使用

如下，操作是不是很简单？不过，大家要记得，framebuffer只负责位图显示，而很多图像都是有编码格式的，如JPG等等，需要先解码，再送给framebuffer。

```
int fb;

unsigned char  *fb_mem;
fb = open ("/dev/fb0", O_RDWR);  1.open 设备文件
fb_mem = mmap (0,     2..mmap
1024*768*4,PROT_READ|PROT_WRITE,MAP_SHARED,fb,0);
fb_mem指向的是进程虚拟地址空间
memset (fb_mem, 0x00, 1024*768*4); 3.操作
```

mmap将虚拟连续的地址空间映射到实际的物理连续空间

写了这么多，我发现，再写一篇如何进行LCD驱动开发的文章，实现framebuffer的第一部分需求，以串起以前对驱动框架的分析，会更加清晰地掌握LCD驱动和framebuffer子系统的来龙去脉。这个可以安排在平台设备驱动的分析之后进行，因为大部分的驱动都是平台设备驱动，应该先讲平台设备驱动的框架，再回到LCD驱动开发，包括LCD平台设备相关的部分（GPIO资源、中断资源配置等）和LCD驱动部分。

这真的是嵌入式企鹅圈2015年最后一篇原创了:-)，在此，祝福大家在新的一年工作学习顺利、身体健康、万事如意！

更多嵌入式Linux和物联网IOT原创技术分享敬请关注微信公众号：嵌入式企鹅圈

阅读全文

本文已收录于以下专栏：嵌入式Linux内核驱动情景分析

关于framebuffer这几天学习的一点心得体会和疑惑

这几天一直在做 imx6q 的视频和

zhongzai2010 2014年09月19日 17:59 1164

一、 FrameBuffer 原理、实现与应用 写屏（转）

一、FrameBuffer 原理、实现与应用 一、FrameBuffer的原理 FrameBuffer 是出现在 2.2.xx 内核当中的一种驱动程序接口。 Linux是工作在保护...

yu704645129 2016年05月31日 17:22 4301

framebuffer的入门介绍-实现程序分析

如想想对lcd屏进行操作（例如在lcd屏幕上画线，或者显示视频数据），我们就必须得了framebuffer（帧缓冲），网上各种百度，大多都说的很官方，至少很难找到那些让人觉得很生动的描述，让我们这些出...

liuzijiang1123 2015年07月20日 19:32 3354

基于framebuffer的驱动分析

framebuffer帧缓冲（简称fb）是linux内核中虚拟出的一个设备，是一个platform类型设备，设备文件位于/dev/fb*...

qq_28992301 2016年10月03日 10:47 2213

一个简单的framebuffer的显示使用例子

本例子中，显示设备是一个oled的显示屏； 没有过多的关于分辨率，刷新频率的设置； 只是演示一个framebuffer的例子。 一，kernel层的驱动代码如下： 1. 注册，这是一个使...

u011006622 2017年06月16日 10:39 1237

2016/1/9：深度剖析安卓Framebuffer设备驱动

忙了几天，今天在公司居然没什么活干 ，所以早上就用公司的电脑写写之前在公司编写framebuffer的使用心得体会总结，这也算是一点开发经验，不过我还没写全，精华部分还是自己藏着吧。直到下午才开始有点...

morixinguan 2016年01月08日 21:26 1787

Android Framebuffer介绍及使用

作者： Aaron 主页： http://www.wxtlife.com/2017/06/07/Android-framebuffer/ 欢迎订阅我的公众号 FrameBuffe...

wx_962464 2017年09月12日 12:45 919

FrameBuffer

FrameBuffer相关概念 FrameBuffer中文译名为帧缓冲驱动，它是出现在2.2.xx内核中的一种驱动程序接口。 Linux是工作在保护模式下，所以用户态进程是无法象DOS那样使用显卡B...

farmwang 2017年04月11日 15:42 346