

学习本章前，给大家打一下预防针， LCD 设备驱动本身属于字符设备驱动的范畴，难度属于一般难度。 在层次架构方面较输入子系统或平台总线驱动设备模型要简单，但是它有复杂的硬件操作。所以，建议大家在学习本章内容之前，要先学习第一部分的第 11 章---LCD 控制器原理与应用。 因为， Linux 下的 LCD 设备驱动引用了很大一部分裸机下的硬件相关的代码。

### 帧缓冲(framebuffer)

#### 帧缓冲(framebuffer)

帧缓冲（ framebuffer）是 Linux 为显示设备提供的一个接口，把显存抽象后的一种设备，他允许上层应用程序在图形模式下直接对显示缓冲区进行读写操作。这种操作是抽象的，统一的。用户不必关心物理显存的位置、换页机制等等具体细节。这些都是由 Framebuffer 设备驱动来完成的。

Linux FrameBuffer 本质上只是提供了对图形设备的硬件抽象，在开发者看来， FrameBuffer 是一块显示缓存，往显示缓存中写入特定格式的数据就意味着向屏幕输出内容。所以说 FrameBuffer 就是一块白板。例如对于初始化为 16 位色的 FrameBuffer 来说， FrameBuffer 中的两个字节代表屏幕上一个点，从上到下，从左至右，屏幕位置与内存地址是顺序的线性关系。 帧缓存有个地址，是在内存里。我们通过不停的向 frame buffer 中写入数据， 显示控制器就自动的从 frame buffer 中取数据并显示出来。

帧缓冲设备对应的设备文件为/dev/fb\*，如果系统有多个显示卡， Linux 下还可支持多个帧缓冲设备，最多可达 32 个，分别为/dev/fb0 到/dev/fb31 ，而/dev/fb则为当前缺省的帧缓冲设备，通常指向/dev/fb0。当然在嵌入式系统中支持一个显示设备就够了。帧缓冲设备为标准字符设备，主设备号为 29，次设备号则从0 到 31 。分别对应/dev/fb0-/dev/fb31 。

#### 帧缓冲与应用程序的交互

对于用户程序而言，它和其他的设备并没有什么区别，用户可以把 fb 看成是一块内存，既可以向内存中写数据，也可以读数据。 fb 的显示缓冲区位于内核空间，应用程序可以把此空间映射到自己的用户空间，在进行操作。 在应用程序中，操作/dev/fbn 的一般步骤如下：

(1) 打开/dev/fbn 设备文件。

(2) 用 ioctl()操作取得当前显示屏幕的参数，如屏幕分辨率、每个像素点的比特数。根据屏幕参数可计算屏幕缓冲区的大小。

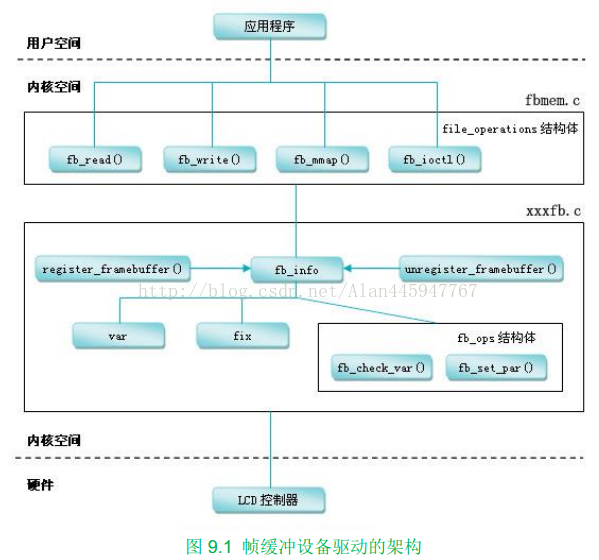
(3) 用 mmap()函数，将屏幕缓冲区映射到用户空间。

(4) 映射后就可以直接读/写屏幕缓冲区，进行绘图和图片显示了。

#### 帧缓冲显示原理

帧缓冲类似一个蓄水池，存放来自用户进程的数据，然后把这些数据再输入显示设备中。对于用户而言，帧缓冲就是内存中的一块区域，可以读、写、映射。只要在初始化阶段把显示设备映射到用户进程空间，可以理解为将屏幕中的每一点和帧缓冲的每一点一一对应起来。这样接下来就可以对这块内存区域填充任何已经定义的像素以及颜色，而屏幕也就可以根据刚才写入的像素及颜色呈现出五彩缤纷的画面。

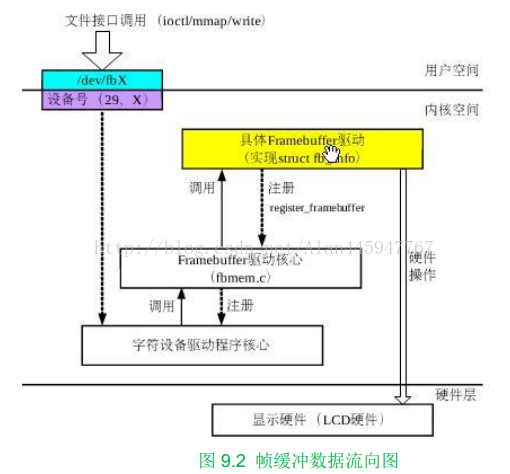
### Linux 帧缓冲设备驱动架构



一个帧缓冲区对应一个 struct fb\_info 结构，它包括了帧缓冲设备的属性和操作的完整集合。 具体的 lcd 设备驱动(xxxfb.c)通过 register\_framrbuffer()向内核注册帧缓冲设备。

帧缓冲设备提供给用户空间的 file\_operations 结构体由 fbmem.c 中的file\_operations 提供，而特定的帧缓冲设备 fb\_info 结构体的注册、注销以及其中的成员的实现，则由具体的 lcd 设备驱动(xxxfb.c)实现。

#### 应用系统调用到 LCD 寄存器的流程



如图 9.2 所示，当应用程序通过系统调用(ioctl/mmap/write)打开/dev/fbX时，就 会 调 用 到 主 设 备 号 为 29 ， 次 设 备 号 为 X 的 字 符 设 备 驱 动 里 的(fb\_ioctl/fb\_mmap/fb\_write) 函 数 ， 而 这 些 函 数 是 由 framebuffer 驱 动 核 心fbmem.c 文件实现的，并且它提供了一个 register\_framebuffer()注册接口函数。当有具体的 LCD 设备驱动时，就会使用 register\_framebuffer() 函数向内核注册一个帧缓冲设备。 这时候应用程序的系统调用会进一步调用到具体 LCD 设备驱动里的(xxx\_lcd\_ioctl/ xxx\_lcd\_mmap/ xxx\_lcd\_write)函数，最后操作到 LCD 硬件寄存器等。

#### fb\_info 结构体

无论是帧缓冲设备驱动(fbmem.c)还是具体的 LCD 设备驱动(xxxfb.c)都离不开 fb\_info 结构体，它包括了帧缓冲设备属性和操作的完整描述，包括设备的设置参数、状态以及操作函数指针。

/\* 参考 include\linux\fb.h \*/

struct fb\_info {

atomic\_t count; //原子变量

int node; //用作次设备号索引

int flags;

struct mutex lock; //用于 open/release/ioctl 函数的锁

struct fb\_var\_screeninfo var; //可变参数，重点关注

struct fb\_fix\_screeninfo fix; //固定参数，重点关注

struct fb\_monspecs monspecs; //显示器标准

struct work\_struct queue; //帧缓冲区队列

struct fb\_pixmap pixmap; //图像硬件映射

struct fb\_pixmap sprite; //光标硬件映射

struct fb\_cmap cmap; //当前颜色表

struct list\_head modelist; //模式链表

struct fb\_videomode \*mode; //当前 video 模式

struct fb\_ops \*fbops; //帧缓冲操作函数集

char \_\_iomem \*screen\_base; //显存基地址

unsigned long screen\_size; //显存大小

void \*pseudo\_palette; //伪 16 色颜色表

#define FBINFO\_STATE\_RUNNING 0

#define FBINFO\_STATE\_SUSPENDED 1

u32 state; //硬件状态，如挂起

void \*fbcon\_par; //用作私有数据区

void \*par; //info->par 指向了额外多申请内存空间的首地址

};

##### 可变参数(fb\_var\_screeninfo)

fb\_info结构体有个重要的成员，那就是 fb\_var\_screeninfo可变参数结构体，fb\_var\_screeninfo 结构体定义了一些在系统运行期间可以改变的信息。例如像素深度、灰度级、颜色格式、时序，屏幕边缘空白区等。

/\* 参考 include\uapi\linux\fb.h \*/

struct fb\_var\_screeninfo {

\_\_u32 xres; /\* 物理分辨率的 X 坐标 \*/

\_\_u32 yres; /\* 物理分辨率的 Y 坐标 \*/

\_\_u32 xres\_virtual; /\* 虚拟分辨率的 X 坐标 \*/

\_\_u32 yres\_virtual; /\* 虚拟分辨率的 Y 坐标 \*/

\_\_u32 xoffset; /\* 虚拟分辨率与物理分辨率的 X 坐标的偏移量\*/

\_\_u32 yoffset; /\* 虚拟分辨率与物理分辨率的 Y 坐标的偏移量\*/

\_\_u32 bits\_per\_pixel; /\* 每个像素占据多少位 \*/

struct fb\_bitfield red; /\* 红色位域 \*/

struct fb\_bitfield green; /\* 绿色位域 \*/

struct fb\_bitfield blue; /\* 蓝色位域 \*/

\_\_u32 activate; /\* 活动模式 \*/

\_\_u32 grayscale; /\* 灰度模式 \*/

struct fb\_bitfield transp; /\* 透明色 \*/

\_\_u32 nonstd; /\* 非标准像素格式 \*/

...

\_\_u32 pixclock; /\* 像素时钟，单位是皮秒 \*/

\_\_u32 left\_margin; /\* 左侧边缘区 \*/

\_\_u32 right\_margin; /\* 右侧边缘区 \*/

\_\_u32 upper\_margin; /\* 顶部边缘区 \*/

\_\_u32 lower\_margin;

\_\_u32 hsync\_len; /\* 水平扫描边缘区 \*/

\_\_u32 vsync\_len; /\* 垂直扫描边缘区 \*/

...

};

其中 fb\_bitfield 结构体表示位域，其定义如下：

/\* 参考 include\uapi\linux\fb.h \*/

struct fb\_bitfield {

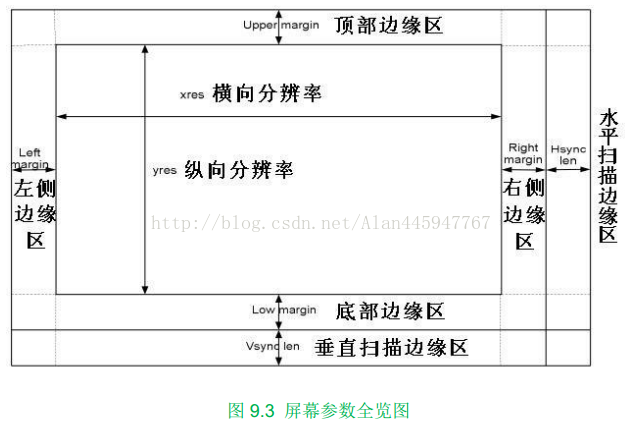
\_\_u32 offset; /\* 位域偏移值 \*/

\_\_u32 length; /\* 位域长度 \*/

\_\_u32 msb\_right; /\* 等于 1 ，表示最重要一位在右 \*/

};

图 9.3 标出了各种边缘区在整个屏幕上的位置:



##### 固定参数(fb\_fix\_screeninfo)

fb\_fix\_screeninfo，该数据结构定义了一些系统运行期间不能改变的信息，例如设备名，屏幕的像素数量，缓冲区的首址和长度等。这类信息一般通过 ioctl函数获得。

/\* 参考 include\uapi\linux\fb.h \*/

struct fb\_fix\_screeninfo {

char **id**[16]; **/\*** 设备名 **\*/**

unsigned long **smem\_start**; **/\*** 缓冲区物理地址起始地址 **\*/**

\_\_u32 **smem\_len**; **/\*** 缓冲区长度 **\*/**

\_\_u32 **type**; **/\*** 设备类型，例如 **TFT** 或 **STN \*/**  
...

\_\_u32 **visua**l; **/\*** 色彩类型，真彩色、假彩色或单色 **\*/**  
...

\_\_u32 **line\_length**; **/\*** 屏幕上每行的字节数 **\*/**

unsigned long mmio\_start; /\* IO 映射区起始地址（物理地址） \*/  
\_\_u32 mmio\_len; /\* IO 映射区长度 \*/

...

\_\_u16 reserved[3]; /\* 系统保留\*/

};

##### fb\_ops

结构体fb\_ino结构体里的成员变量 fb\_ops 为指向特定的 LCD 底层操作的函数指针，这些函数需要驱动工程师来完成。

/\* 参考 include\linux\fb.h \*/

struct fb\_ops {

struct module \*owner;

int (\*fb\_open)(struct fb\_info \*info, int user);

int (\*fb\_release)(struct fb\_info \*info, int user);

...

/\* 检测可变参数，并调整到支持的值 \*/

int (\*fb\_check\_var)(struct fb\_var\_screeninfo \*var,

struct fb\_info \*info);

/\* 根据 info->var 设置 video 模式 \*/

int (\*fb\_set\_par)(struct fb\_info \*info);

/\* 设置 color 寄存器，设置调色板 \*/

int (\*fb\_setcolreg)(unsigned regno, unsigned red, unsigned green,

unsigned blue, unsigned transp, struct fb\_info \*info);

...

/\* 画一个矩形 \*/

void (\*fb\_fillrect) (struct fb\_info \*info,

const struct fb\_fillrect \*rect);

/\* 数据拷贝 \*/

void (\*fb\_copyarea) (struct fb\_info \*info,

const struct fb\_copyarea \*region);

/\* 图像填充 \*/

void (\*fb\_imageblit) (struct fb\_info \*info,

const struct fb\_image \*image);

...

};

### 帧缓冲设备驱动浅析(fbmem.c)

看标题，注意关键词，是浅析而不是深入研究。 为什么是浅析呢？ 因为这个是 Linux 大牛已经帮我们写好的，通用的、核心的驱动。 它并不代表某一个具体的 LCD 设备驱动，但是它作为应用程序与底层 LCD 设备驱动的中转层，起到了很好的衔接作用。 大概知道一下它的工作原理，还是有必要滴。 钻的太深，我担心你爬不出来啊， 兄弟。

#### 入口函数分析一个函数，从它的入口函数开始分析。

/\* 参考 drivers\video\fbmem.c \*/

#define FB\_MAJOR 29 /\* /dev/fb\* framebuffers \*/

static int \_\_init fbmem\_init(void)

{

…

/\* 注册一个主设备号为 29，文件操作函数为 fb\_fops 的字符设备 \*/

if (register\_chrdev(FB\_MAJOR,"fb",&fb\_fops))

printk("unable to get major %d for fb devs\n", FB\_MAJOR);

/\* 创建一个 graphics 类 \*/

fb\_class = class\_create(THIS\_MODULE, "graphics");

…

return 0;

}

入口函数主要做了二件事，一是注册一个主设备号为 29，文件操作函数为fb\_fops 的字符设备；二是创建一个 graphics 类。这二个都是我们所非常熟悉的，但是，细心的读者可能会发现，以往还会多一件事，什么事呢？给你一分钟时间思考……那就是在类下面创建设备，这样 udev 机制就会自动创建设备节点，在这里是/dev/fb0--/dev/fb31 。 但是这里并没有这么做，是不是 Linux 大牛们不小心漏了呢？ 你再猜猜？ O(∩ \_∩ )O 哈哈~，好吧，这问题，留到后面再讲。

#### fb\_fops 实例

static const struct file\_operations fb\_fops = {

.owner = THIS\_MODULE,

.read = fb\_read,

.write = fb\_write,

.unlocked\_ioctl = fb\_ioctl,

……

.mmap = fb\_mmap,

.open = fb\_open,

.release = fb\_release,

……

.llseek = default\_llseek,

};

看到这里，这不就是我们常见的字符设备驱动里的文件操作集实例吗？对，没错，帧缓冲设备驱动本来就属于字符设备驱动的范畴。

#### fb\_open

假设应用程序 open("/dev/fb0", ...)， 主设备号: 29, 次设备号: 0。就会调用到fb\_open 函数。

/\* 参考 drivers\video\fbmem.c \*/

static int fb\_open(struct inode \*inode, struct file \*file)

\_\_acquires(&info->lock)

\_\_releases(&info->lock)

{

/\* 从设备节点中获取次设备号 \*/

int fbidx = iminor(inode);

struct fb\_info \*info;

int res = 0;

/\* 通过次设备号来获取一个 fb\_info 实例 \*/

info = get\_fb\_info(fbidx);

if (!info) {

request\_module("fb%d", fbidx);

info = get\_fb\_info(fbidx);

if (!info)

return -ENODEV;

}

if (IS\_ERR(info))

return PTR\_ERR(info);

mutex\_lock(&info->lock);

if (!try\_module\_get(info->fbops->owner)) {

res = -ENODEV;

goto out;

}

/\* 将 fb\_info 实例赋给文件私有数据，以供其他函数使用 \*/

file->private\_data = info;

/\* 如果新的 fb\_info 实例中的 fbops->fb\_open 存在的话就调用它 \*/

if (info->fbops->fb\_open) {

res = info->fbops->fb\_open(info,1);

if (res)

module\_put(info->fbops->owner);

}

#ifdef CONFIG\_FB\_DEFERRED\_IO

if (info->fbdefio)

fb\_deferred\_io\_open(info, inode, file);

#endif

out:

mutex\_unlock(&info->lock);

if (res)

put\_fb\_info(info);

return res;

}

#### get\_fb\_info

get\_fb\_info从函数名字上看，获得 fb\_info 结构体，简单明了。

/\* 参考 drivers\video\fbmem.c \*/

static struct fb\_info \*get\_fb\_info(unsigned int idx)

{

struct fb\_info \*fb\_info;

/\* 如果次设备号大于 32，出错 \*/

if (idx >= FB\_MAX)

return ERR\_PTR(-ENODEV);

mutex\_lock(?istration\_lock);

/\* 以次设备号为下标，在 registered\_fb 数组里找到一项 fb\_ino 实例 \*/

fb\_info = registered\_fb[idx];

if (fb\_info)

atomic\_inc(&fb\_info->count);

mutex\_unlock(?istration\_lock);

return fb\_info;

}

#### fb\_read

fb\_read假设打开/dev/fb0 后， 应用程序 read(),就会调用到 fb\_read 函数。

/\* 参考 drivers\video\fbmem.c \*/

static ssize\_t

fb\_read(struct file \*file, char \_\_user \*buf, size\_t count, loff\_t \*ppos)

{

unsigned long p = \*ppos;

/\* 以次设备号为下标，在 registered\_fb 数组里找到一项 fb\_ino 实例 \*/

struct fb\_info \*info = file\_fb\_info(file);

u8 \*buffer, \*dst;

u8 \_\_iomem \*src;

int c, cnt = 0, err = 0;

unsigned long total\_size;

if (!info || ! info->screen\_base)

return -ENODEV;

if (info->state != FBINFO\_STATE\_RUNNING)

return -EPERM;

/\* 如果新的 fb\_info 实例中的 fbops->fb\_read 存在的话就调用它 \*/

if (info->fbops->fb\_read)

return info->fbops->fb\_read(info, buf, count, ppos);

/\* 没有默认的读函数，就从显存的虚拟起始

\* 地址读数据，大小为虚拟显存大小

\*/

total\_size = info->screen\_size;

/\* 如果虚拟显存大小为了，则设置大小为 fb 缓冲区长度 \*/

if (total\_size == 0)

total\_size = info->fix.smem\_len;

/\* 如果读的偏移值大于 total\_size，则返回 0 个字节 \*/

if (p >= total\_size)

return 0;

/\* 如果读的总量大于 total\_size，则调整

\* 最多读 total\_size 个字节的数据

\*/

if (count >= total\_size)

count = total\_size;

/\* 调整读的位置及能读多少字节 \*/

if (count + p > total\_size)

count = total\_size - p;

/\* 分配内存，最大分配 4K 的大小 \*/

buffer = kmalloc((count > PAGE\_SIZE) ? PAGE\_SIZE : count,

GFP\_KERNEL);

if (!buffer)

return -ENOMEM;

/\* 源虚拟基地址 \*/

src = (u8 \_\_iomem \*) (info->screen\_base + p);

/\* 如果新的 fb\_info 实例中的 fbops->fb\_sync 存在的话就调用它 \*/

if (info->fbops->fb\_sync)

info->fbops->fb\_sync(info);

while (count) {

/\* 读多少计数变量，单位为 byte \*/

c = (count > PAGE\_SIZE) ? PAGE\_SIZE : count;

/\* dst 指向刚分配 buffer 内存的首地址 \*/

dst = buffer;

/\* 从源地址里拷贝数据到目的地址 \*/

fb\_memcpy\_fromfb(dst, src, c);

/\* 调整位置 \*/

dst += c;

src += c;

/\* 从内核刚申请内存的地址 buffer 拷贝

\* c 长度的数据到用户空间的 buf 里去

\*/

if (copy\_to\_user(buf, buffer, c)) {

err = -EFAULT;

break;

}

/\* 调整偏移位置 \*/

\*ppos += c;

buf += c;

cnt += c;

/\* count 变量减去已经读取的 c 数量，用于判断 while(count)是否为真\*/

count -= c;

}

/\* 释放内存 \*/

kfree(buffer);

/\* err = 0 时，返回被拷贝成功的数量 cnt \*/

return (err) ? err : cnt;

}

#### file\_fb\_info

file\_fb\_infofile\_fb\_info 函数的功能与 get\_fb\_info 函数的功能几乎是一样的。

static struct fb\_info \*file\_fb\_info(struct file \*file)

{

/\* 获取次设备号 \*/

struct inode \*inode = file->f\_path.dentry->d\_inode;

int fbidx = iminor(inode);

/\* 以次设备号为下标，在 registered\_fb 数组里找到一项 fb\_ino 实例 \*/

struct fb\_info \*info = registered\_fb[fbidx];

/\* 如果获取的 fb\_info 实例与 fb\_open 函数

\* 获取的 fb\_info 实例不一致，置为 NULL

\*/

if (info != file->private\_data)

info = NULL;

return info;

}

#### registered\_fb

通过分析 fb\_open 和 fb\_read 函数，都发现有个共同的特点，就是以次设备号为下标找到一项 fb\_info 结构体。 从哪里找？从 registered\_fb[]数组里找。 那这个 registered\_fb[]数组，在哪里被设置？ 还记得，在第八章里，内核如何注册平台总线了吗？我不是叫你回忆如何注册的，我是叫你回忆，怎么找到函数调用流程的。这就是使用 Source Insight 的好处啦，使用“ctrl+/” 搜索 registered\_fb会发现如下结果：

register\_framebuffer

-->do\_register\_framebuffer

-->registered\_fb[i] = fb\_info;

#### register\_framebuffer

/\* 参考 drivers\video\fbmem.c \*/  
int **register\_framebuffer**(struct **fb\_info** \*fb\_info)  
{  
int ret;  
mutex\_lock(®istration\_lock);  
ret = **do\_register\_framebuffer**(fb\_info);  
mutex\_unlock(®istration\_lock);  
return ret;  
}

register\_framebuffer 上锁后，调用 do\_register\_framebuffer 来向内核注册一个帧缓冲设备。

#### do\_register\_framebuffer

do\_register\_framebuffer 函数主要做了三件事：第一、 在 graphics 类下创建设备，设备节点为/dev/fb0--/dev/fb31 。第二、 简单初始化 fb\_info 结构体部分成员。第三、 将初始化好的 fb\_info 结构体赋给 registered\_fb[]。

/\* 参考 drivers\video\fbmem.c \*/

static int do\_register\_framebuffer(struct fb\_info \*fb\_info)

{

int i;

struct fb\_event event;

struct fb\_videomode mode;

......

if (num\_registered\_fb == FB\_MAX)

return -ENXIO;

num\_registered\_fb++;

for (i = 0 ; i < FB\_MAX; i++)

if (!registered\_fb[i])

break;

fb\_info->node = i;

atomic\_set(&fb\_info->count, 1);

mutex\_init(&fb\_info->lock);

mutex\_init(&fb\_info->mm\_lock);

/\* 在 graphics 类下创建设备，设备节点为/dev/fb0--/dev/fb31 \*/

fb\_info->dev = device\_create(fb\_class, fb\_info->device,

MKDEV(FB\_MAJOR, i), NULL, "fb%d", i);

...

/\* 简单初始化 fb\_info 结构体部分成员 \*/

fb\_init\_device(fb\_info);

if (fb\_info->pixmap.addr == NULL) {

fb\_info->pixmap.addr = kmalloc(FBPIXMAPSIZE, GFP\_KERNEL);

if (fb\_info->pixmap.addr) {

fb\_info->pixmap.size = FBPIXMAPSIZE;

fb\_info->pixmap.buf\_align = 1;

fb\_info->pixmap.scan\_align = 1;

fb\_info->pixmap.access\_align = 32;

fb\_info->pixmap.flags = FB\_PIXMAP\_DEFAULT;

}

}

fb\_info->pixmap.offset = 0;

if (!fb\_info->pixmap.blit\_x)

fb\_info->pixmap.blit\_x = ~(u32)0;

if (!fb\_info->pixmap.blit\_y)

fb\_info->pixmap.blit\_y = ~(u32)0;

if (!fb\_info->modelist.prev || !fb\_info->modelist.next)

INIT\_LIST\_HEAD(&fb\_info->modelist);

fb\_var\_to\_videomode(&mode, &fb\_info->var);

fb\_add\_videomode(&mode, &fb\_info->modelist);

/\* 将初始化好的 fb\_info 结构体赋给 registered\_fb[] \*/

registered\_fb[i] = fb\_info;

event.info = fb\_info;

if (!lock\_fb\_info(fb\_info))

return -ENODEV;

fb\_notifier\_call\_chain(FB\_EVENT\_FB\_REGISTERED, &event);

unlock\_fb\_info(fb\_info);

return 0;

}

好吧，看到这里， 9.2.1 小 节 留 下 的问 题 就 自然 而 然 的就 解 答 了 。registered\_fb[]数组是在 register\_framebuffer()函数里设置。

#### fb\_mmap

file\_operations 中的 mmap 成员函数非常重要， 正是它，将显示缓冲区映射到用户空间，从而使得用户空间可以直接操作显示缓冲区而省去一次从用户空间到内核空间的数据拷贝过程，明显提高了高效。而在 fbmem.c 里 mmap 对应的函数是 fb\_mmap。

static int fb\_mmap(struct file \*file, struct vm\_area\_struct \* vma)

{

/\* 以次设备号为下标找到一项 fb\_info 结构体 \*/

struct fb\_info \*info = file\_fb\_info(file);

struct fb\_ops \*fb;

unsigned long off;

unsigned long start;

u32 len;

if (!info)

return -ENODEV;

if (vma->vm\_pgoff > (~0UL >> PAGE\_SHIFT))

return -EINVAL;

off = vma->vm\_pgoff << PAGE\_SHIFT;

fb = info->fbops;

if (!fb)

return -ENODEV;

mutex\_lock(&info->mm\_lock);

/\* 如果新的 fb\_info 实例中的 fbops->fb\_mmap 存在的话就调用它 \*/

if (fb->fb\_mmap) {

int res;

res = fb->fb\_mmap(info, vma);

mutex\_unlock(&info->mm\_lock);

return res;

}

/\* 源:fb 缓冲内存的开始位置(物理地址) \*/

start = info->fix.smem\_start;

/\* 长度 \*/

len = PAGE\_ALIGN((start & ~PAGE\_MASK) + info->fix.smem\_len);

/\* 如果 off 大于 len，则要调整源起始地址、长度 \*/

if (off >= len) {

/\* memory mapped io \*/

off -= len;

if (info->var.accel\_flags) {

mutex\_unlock(&info->mm\_lock);

return -EINVAL;

}

start = info->fix.mmio\_start;

len = PAGE\_ALIGN((start & ~PAGE\_MASK) + info->fix.mmio\_len);

}

mutex\_unlock(&info->mm\_lock);

start &= PAGE\_MASK;

if ((vma->vm\_end - vma->vm\_start + off) > len)

return -EINVAL;

off += start;

vma->vm\_pgoff = off >> PAGE\_SHIFT;

vma->vm\_page\_prot = vm\_get\_page\_prot(vma->vm\_flags);

fb\_pgprotect(file, vma, off);

/\* 正式映射物理内存到用户空间虚拟地址 \*/

if (io\_remap\_pfn\_range(vma, vma->vm\_start, off >> PAGE\_SHIFT,

vma->vm\_end - vma->vm\_start, vma->vm\_page\_prot))

return -EAGAIN;

return 0;

}

#### fb\_ioctl

fb\_ioctl()函数最终实现对用户 I/O 控制命令的执行，比如获取可变的屏幕参数 命 令 (FBIOGET\_VSCREENINFO) ， 设 置 可 变 的 屏 幕 参 数 命 令(FBIOPUT\_VSCREENINFO) ， 获 取 固 定 的 屏 幕 参 数 命 令(FBIOGET\_FSCREENINFO)等等。

static long fb\_ioctl(struct file \*file, unsigned int cmd,unsigned long arg)

{

struct fb\_info \*info = file\_fb\_info(file);

if (!info)

return -ENODEV;

return do\_fb\_ioctl(info, cmd, arg);

}

fb\_ioctl()函数的实现，其实是通过调用 do\_fb\_ioctl()函数实现的，它才是资本执行者呀~~对于 do\_fb\_ioctl()函数的分析，这里就不再深入分析下去了，它主要是根据cmd 命令，来判断要执行哪部分功能，比如： cmd = FBIOGET\_VSCREENINFO,它就会转到 FBIOGET\_VSCREENINFO 标号去，将 fb\_info 成员设置好后，通过 copy\_to\_user()函数拷贝回用户空间；而用户空间要设置显示设备的话，则是通过 copy\_from\_user()函数到达内核空间的。

### 从零编写 LCD 设备驱动

/driver/video

一眼扫过去， s3c2410fb.c 貌似看起来相对比较熟悉。 进去咋一看，确实这是三星公司提供的基于 S3C2410/S3C2440 架构来编写的 LCD 设备驱动。 它属于我们第八章讲的平台总线驱动设备模型，入口函数注册了一个平台驱动，如果系统中存在有相同名字的平台设备，就会调用 platform\_driver 的 probe 函数。

由于 S5PV210 的 LCD 控制器的寄存器与 S3C2410/S3C2440 的 LCD 控制器的寄存器有太大的差别了，基于 s3c2410fb.c 来移植到 S5PV210 架构上实在是太麻烦了，要修改的地方太多。 所以，我们直接自己从零编写一个 LCD 设备驱动。

这里仅仅大概讲一下 s3c24xxfb\_probe 函数做了哪些事情，有兴趣的读者请自行分析 drivers/video/s3c2410fb.c

(1) 获取平台数据，这里平台数据应该存放板相关的 fb\_info 结构体实例。

(2) 通过平台设备 platform\_device 获得 IRQ。

(3) 使用 framebuffer\_alloc()函数分配一个 fb\_info 结构体。

(4) 通过平台设备 platform\_device 获得内存资源。

(5) 申请以 res->start 地址开始大小为 size 的 I/O 内存。

(6) 使用 ioremap()函数映射 I/O 地址。

(7) 禁止 Video output

(8) 设置 fb\_info 结构体通用的固定参数 fb\_fix\_screeninfo 结构体。

(9) 设置 fb\_info 结构体通用的可变参数 fb\_var\_screeninfo 结构体。

(10) 设置 fb\_ops 结构体

(11) 设置假调色板

(12) 申请中断

(13) 获取 lcd 时钟

(14) 分配显存

(15) 初始化 LCD 相关的寄存器

(16) 检查可变参数

(17) 注册 fb\_info 结构体

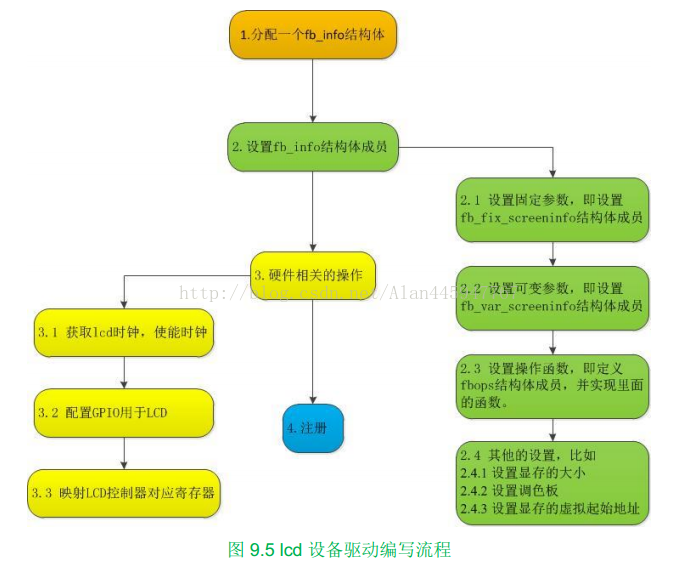
#### 图说 LCD 设备驱动编写流程

1. 分配一个 fb\_info 结构体

2. 设置 fb\_info 结构体成员

3. 硬件相关的操作

4. 注册



#### 从零编写 LCD 设备驱动分析

LCD 设备驱动源码路径为： webee210\_drivers\8th\_lcd\webee210\_lcd.c考虑到本驱动大量移植了裸机下的 LCD 源码，特意将裸机下的 LCD 源码也放在这里，方面大家参考。

LCD 裸机源码路径为： webee210\_drivers\8th\_lcd\210\_code\11.lcd

##### 入口函数

本驱动并没有使用平台总线驱动设备模型，直接在一个文件上使用了 LCD设备驱动， 入口函数完成了图 9.5 的大部分工作。

1.分配一个 fb\_info

/\* 第一个参数为 0，表示只需要分配结构体本身大小，无需分配额外空间

\* 第二个参数为 NULL，表示无 device 结构

\*/

webee\_fbinfo = framebuffer\_alloc(0 , NULL);

##### 设置固定的参数

/\* 关于这些参数的含义，请参考 9.1.4.2 小节 \*/

strcpy(webee\_fbinfo->fix.id, "webee210\_lcd");

webee\_fbinfo->fix.smem\_len = 800 \* 480 \* 32/8;

webee\_fbinfo->fix.type = FB\_TYPE\_PACKED\_PIXELS;

webee\_fbinfo->fix.visual = FB\_VISUAL\_TRUECOLOR;

webee\_fbinfo->fix.line\_length = 800 \* 32/8;

##### 设置可变的参数

/\* 关于这些参数的含义，请参考 9.1.4.1 小节 \*/

webee\_fbinfo->var.xres = 800;

webee\_fbinfo->var.yres = 480;

webee\_fbinfo->var.xres\_virtual = 800;

webee\_fbinfo->var.yres\_virtual = 480;

webee\_fbinfo->var.bits\_per\_pixel = 32;

/\*RGB:888\*/

webee\_fbinfo->var.red.offset = 16;

webee\_fbinfo->var.red.length = 8;

webee\_fbinfo->var.green.offset = 8;

webee\_fbinfo->var.green.length = 8;

webee\_fbinfo->var.blue.offset = 0;

webee\_fbinfo->var.blue.length = 8;

webee\_fbinfo->var.activate = FB\_ACTIVATE\_NOW

##### 设置操作函数

/\* 后面分析这个 fops \*/

webee\_fbinfo->fbops = &webee210\_lcdfb\_ops;

##### 设置显存的大小

webee\_fbinfo->screen\_size = 800 \* 480 \* 32/8;

##### 设置调色板

static u32 pseudo\_palette[16];

……

/\* pseudo\_palette 是一个 u32 型的数组,在 webee210\_lcdfb\_setcolreg

\* 函数中被设置， 如何设置看后面 webee210\_lcdfb\_setcolreg 函数分析。

\*/

webee\_fbinfo->pseudo\_palette = pseudo\_palette;

##### 设置显存的虚拟起始地址

webee\_fbinfo->screen\_base = dma\_alloc\_writecombine(NULL,webee\_fbinfo->fix.smem\_len,(u32\*)&(webee\_fbinfo->fix.smem\_start), GFP\_KERNEL);

/\* 函数原型如下：

\* 第一个参数为： device 结构体，设置为 NULL，表示无 device 结构

\* 第二个参数为： 需分配帧缓冲大小，这里为 800 \* 480 \* 32/8

\* 第三个参数为： 返回的帧缓冲的物理起始地址，表示 DMA 可用

\* 第四个参数为：分配标志，如 GFP\_KERNEL

\*/

static inline void \*dma\_alloc\_writecombine(struct device \*dev,size\_t size,dma\_addr\_t \*dma\_handle, gfp\_t flag)

#### 硬件相关的操作3.1 获取 lcd 时钟，使能时钟

lcd\_clk = clk\_get(NULL, "lcd");

if (!lcd\_clk || IS\_ERR(lcd\_clk)) {

printk(KERN\_INFO "failed to get lcd clock source\n");

}

clk\_enable(lcd\_clk);3.2 配置 GPIO 用于 LCD

gpf0con = ioremap(0xE0200120, 4);

gpf1con = ioremap(0xE0200140, 4);

gpf2con = ioremap(0xE0200160, 4);

gpf3con = ioremap(0xE0200180, 4);

gpd0con = ioremap(0xE02000A0, 4);

gpd0dat = ioremap(0xE02000A4, 4);

display\_control = ioremap(0xe0107008, 4);

/\* 设置相关 GPIO 引脚用于 LCD \*/

\*gpf0con = 0x22222222;

\*gpf1con = 0x22222222;

\*gpf2con = 0x22222222;

\*gpf3con = 0x22222222;

/\* 使能 LCD 本身 \*/

\*gpd0con |= 1<<4;

\*gpd0dat |= 1<<1;

/\* 显示路径的选择, 0b10: RGB=FIMD I80=FIMD ITU=FIMD \*/

\*display\_control = 2<<0;

关于这些引脚的含义，这里就不详细分析了， 我们要的是层次和架构，对于LCD 硬件的设置，大家可以参考第一部分的第 11 章《S5PV210 LCD 控制器原理与应用》，或者自己对照着手册来看看是什么含义。

#### 映射 LCD 控制器对应寄存器

lcd\_regs = ioremap(0xF8000000, sizeof(struct s5pv210\_lcd\_regs));

vidw00add0b0 = ioremap(0xF80000A0, 4);

vidw00add1b0 = ioremap(0xF80000D0, 4);

lcd\_regs->vidcon0 &= ~((3<<26) | (1<<18) | (0xff<<6) | (1<<2));

lcd\_regs->vidcon0 |= ((5<<6) | (1<<4) );

/\* 在 vclk 的下降沿获取数据 \*/

lcd\_regs->vidcon1 &= ~(1<<7);

/\* HSYNC 极性反转, VSYNC 极性反转 \*/

lcd\_regs->vidcon1 |= ((1<<6) | (1<<5));

lcd\_regs->vidtcon0 = (VBPD << 16) | (VFPD << 8) | (VSPW << 0);

lcd\_regs->vidtcon1 = (HBPD << 16) | (HFPD << 8) | (HSPW << 0);

lcd\_regs->vidtcon2 = (LINEVAL << 11) | (HOZVAL << 0);

lcd\_regs->wincon0 &= ~(0xf << 2);

lcd\_regs->wincon0 |= (0xB<<2)|(1<<15);

lcd\_regs->vidosd0a = (LeftTopX<<11) | (LeftTopY << 0);

lcd\_regs->vidosd0b = (RightBotX<<11) | (RightBotY << 0);

lcd\_regs->vidosd0c = (LINEVAL + 1) \* (HOZVAL + 1);

\*vidw00add0b0 = webee\_fbinfo->fix.smem\_start;

\*vidw00add1b0 =

webee\_fbinfo->fix.smem\_start + webee\_fbinfo->fix.smem\_len;

lcd\_regs->shadowcon = 0x1; /\* 使能通道 0 \*/

lcd\_regs->vidcon0 |= 0x3; /\* 开启总控制器 \*/

lcd\_regs->wincon0 |= 1; /\* 开启窗口 0 \*/

#### 注册

register\_framebuffer(webee\_fbinfo);

##### 出口函数

出口函数还是一如既往的， Webee 总喜欢把它形容为“聪明的跟屁虫”，为什么这么说呢？因为出口函数总喜欢做入口函数相反的事情，如果非要文艺点形容的话，还是那句话，做人要有始有终嘛~~

static void \_\_exit webee210\_lcd\_exit(void)

{

unregister\_framebuffer(webee\_fbinfo);

dma\_free\_writecombine(NULL,webee\_fbinfo->fix.smem\_len,webee\_fbinfo->screen\_base, webee\_fbinfo->fix.smem\_start);

iounmap(gpf0con);

iounmap(gpf1con);

iounmap(gpf2con);

iounmap(gpf3con);

iounmap(gpd0con);

iounmap(gpd0dat);

iounmap(display\_control);

iounmap(lcd\_regs);

iounmap(vidw00add0b0);

iounmap(vidw00add1b0);

framebuffer\_release(webee\_fbinfo);

}

#### webee210\_lcdfb\_ops

一看到这个，又回到了字符设备驱动的那套了，对吗？对不？细心的读者可能会发现，这个 fops 貌似与字符设备驱动的 fops 有点不一样哟~~

static struct fb\_ops webee210\_lcdfb\_ops =

{

.owner = THIS\_MODULE,

.fb\_setcolreg = webee210\_lcdfb\_setcolreg,

.fb\_fillrect = cfb\_fillrect,

.fb\_copyarea = cfb\_copyarea,

.fb\_imageblit = cfb\_imageblit,

};

看了源码后，发现有什么不大一样了吗？ 以前的 fops 类型是 file\_operations型的，而这里的 fops 是 fb\_ops 型的，与 9.1.4.3 小节讲的 fops 是一致的。 其中，webee210\_lcdfb\_ops 实例里有四个成员函数，但是只需要我们自己实现fb\_setcolreg 成员函数。而 fb\_fillrect、 fb\_copyarea、 fb\_imageblit 成员函数默认为由内核大牛们写好的。 注意了，我们并没有实现这三个成员，所以我们在加载

webee210\_lcd.ko 时，需要先加载这三个函数对应的模块。

##### webee210\_lcdfb\_setcolreg

static int webee210\_lcdfb\_setcolreg(unsigned regno,

unsigned red, unsigned green, unsigned blue,

unsigned transp, struct fb\_info \*info)

{

unsigned int val;

if (regno > 16)

return 1;

/\* 用 red,green,blue 三原色构造出 val \*/

val = chan\_to\_field(red, &info->var.red);

val |= chan\_to\_field(green, &info->var.green);

val |= chan\_to\_field(blue, &info->var.blue);

pseudo\_palette[regno] = val;

return 0;

}

webee210\_lcdfb\_setcolreg()函数使用应用程序提供的 red、 green、 bule 构成出颜色映射值 val，然后将这个 val 写入假调色板中（ pseudo\_palette[]）

##### chan\_to\_field

怎么构造出颜色映射值 val 呢？ 使用 chan\_to\_field()函数。

static unsigned int chan\_to\_field(unsigned int chan,struct fb\_bitfield \*bf)

{

chan &= 0xffff;

chan >>= 16 - bf->length;

return chan << bf->offset;

}

很简单，它根据 fb\_bitfield 位域的 length 和 offset 来构造。 对于 Webe210的 LCD，使用的 RGB 格式是 888 的。 把值代进去一算，你就知道如何构造啦。

#### 从零编写 LCD 设备驱动的测试

修改文件系统的 etc 配置文件，将新的 webee210\_lcd.ko 配置上。

/\* 下面的操作都在虚拟机下， Webee 的文件系统路径可能与你们的不一样 \*/

[root@localhost etc]# cd /home/webee210v2/rootfs/etc

[root@localhost etc]# vim profile

/\* etc/profile 文件部分内容 \*/

export PS1 HOSTNAME

insmod ko/cfbimgblt.ko

insmod ko/cfbfillrect.ko

insmod ko/cfbcopyarea.ko

#insmod ko/lcd.ko

insmod ko/webee210\_lcd.ko

insmod ko/bus.ko

insmod ko/dev.ko

insmod ko/drv.ko

……

将原来的 insmod ko/lcd.ko 屏蔽掉，替换成自己从零编写的 LCD 设备驱动，也就是增加这条命令： insmod ko/webee210\_lcd.ko 当然，前提是已经将编译好的 webee210\_lcd.ko 拷贝到文件系统的 ko 文件夹下了。

同样在 etc/profile 文件里，如果已经屏蔽了网蜂物联网 QT 界面的话，还需要将最后那行命令由：

#qt4 &改为：

qt4 &

改好后，无论你使用 NFS 启动还是以 yaffs2 启动，如果 LCD 上能够正常显示出网蜂物联网的那个 QT 界面，表示此 LCD 设备驱动成功了。 当然，如果你能够触摸的话，表示触摸屏驱动也成功了。但是，注意了，触摸屏和 LCD 屏是两种不同的设备了，不要混淆了哦。