一、什么是帧缓冲

在内存中挑选一段内存，然后通过配置将lcd控制器和这一段内存（显存）连接起来构成一个映射关系，一旦建立映射关系之后，lcd控制器就会自动从显存中读取像素的数据传输给lcd驱动器。所以我们只需要把要显示的图像的像素数据丢到显存中，在屏幕上就能自动看到显示的图像。上面获取的显存专业术语叫做帧缓存—framebuffer。

二、帧缓冲相关结构和接口

结构体：

描述帧缓冲设备，比较重要的成员有struct fb\_var\_screeninfo var、struct fb\_fix\_screeninfo fix和struct fb\_ops \*fbops。

struct fb\_info {

    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\_ops \*fbops; /\*对底层硬件操作的函数指针\*/

    char \_\_iomem \*screen\_base;   /\*显存起始虚拟地址\*/

    unsigned long screen\_size;   /\*映射后的显存大小\*/

    void \*pseudo\_palette;        /\*调色板\*/

};

struct fb\_var\_screeninfo {

    \_\_u32 xres;                /\*可见屏幕一行有多少个像素点\*/

    \_\_u32 yres;                /\*可见屏幕一列有多少个像素点\*/

    \_\_u32 xres\_virtual;        /\*虚拟屏幕一行有多少个像素点\*/

    \_\_u32 yres\_virtual;        /\*虚拟屏幕一列有多少个像素点\*/

    \_\_u32 xoffset;             /\*虚拟到可见屏幕之间的行偏移\*/

    \_\_u32 yoffset;             /\*虚拟到可见屏幕之间的列偏移\*/

    \_\_u32 bits\_per\_pixel;      /\*每个像素的位数，也叫做BPP\*/

    \_\_u32 grayscale;           /\*非0时，指的是灰度\*/

    struct fb\_bitfield red;    /\*fb缓存的R位域\*/

    struct fb\_bitfield green;  /\*fb缓存的G位域\*/

    struct fb\_bitfield blue;   /\*fb缓存的B位域\*/

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

    \_\_u32 activate;            /\* see FB\_ACTIVATE\_\* \*/    FB\_ACTIVATE\_NOW 当前立即设置值

    \_\_u32 height;              /\*高度\*/

    \_\_u32 width;               /\*宽度\*/

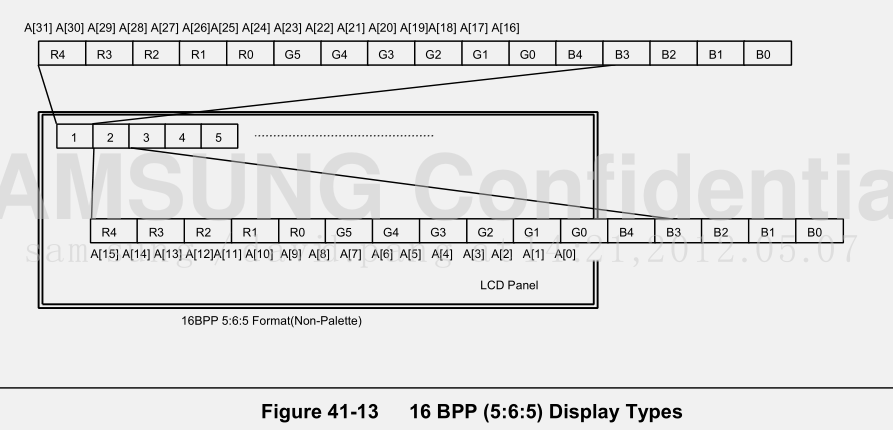
};

struct fb\_bitfield {

\_\_u32 offset;

\_\_u32 length;

};



根据上图可以知道：R的长度为5位，offset的值是11

G的长度为6位，offset的值是5

B的长度为5位，offset的值是0

——————————————————————————————————————————————————————————————————————————

struct fb\_fix\_screeninfo {

    char id[16];                /\*标记当前lcd驱动，用来避免和其他lcd驱动冲突 \*/

    unsigned long smem\_start;   /\*fb缓存的起始物理地址 \*/

    \_\_u32 smem\_len;             /\*fb缓存的长度 \*/

    \_\_u32 type;                 /\*看FB\_TYPE\_\* \*/

    \_\_u32 visual;               /\*看FB\_VISUAL\_\* \*/

    \_\_u32 line\_length;          /\*一行的字节数 \*/

};

成员type作用：代表像素值和显存的关系

#define FB\_TYPE\_PACKED\_PIXELS 0 像素值和显存有直接对应关系，比如写入5到显存，则显存中的具体像素值就是5。

成员visual作用：代表像素值与颜色的映射关系

#define FB\_VISUAL\_MONO01 0

#define FB\_VISUAL\_MONO10 1

#define FB\_VISUAL\_TRUECOLOR 2

#define FB\_VISUAL\_PSEUDOCOLOR 3

#define FB\_VISUAL\_DIRECTCOLOR 4

#define FB\_VISUAL\_STATIC\_PSEUDOCOLOR 5

#define FB\_VISUAL\_FOURCC 6

在了解像素值和颜色的映射关系前，我们需要先了解一下色彩深度。

色彩深度又叫[色彩位数](https://baike.baidu.com/item/%E8%89%B2%E5%BD%A9%E4%BD%8D%E6%95%B0)，即位图中要用多少个二进制位来表示每个点的颜色，是分辨率的一个重要指标。常用有1位(单色)，2位(4色，CGA)，4位(16色，VGA)，8位(256色)，16位(增强色)，24位(真彩色)和32位等。位数越多，由RGB三色组成的颜色种类越多,其中24位及以上称为真彩。

8位色(256色模式)：  R:3 bits;G: 3 bits;B: 2 bits;

16位色(High color)：   R: 5 bits;G: 6 bits;B: 5 bits;

24位色(True color)：  一般24bit就是8bit/通道，真彩色。

32位色中的24位用来保存颜色信息(R8G8B8)，另外的8位用来保存ALPHA信息，ALPHA属性就是透明度。(透明度实现的是3D效果,增加图像层次感效果)

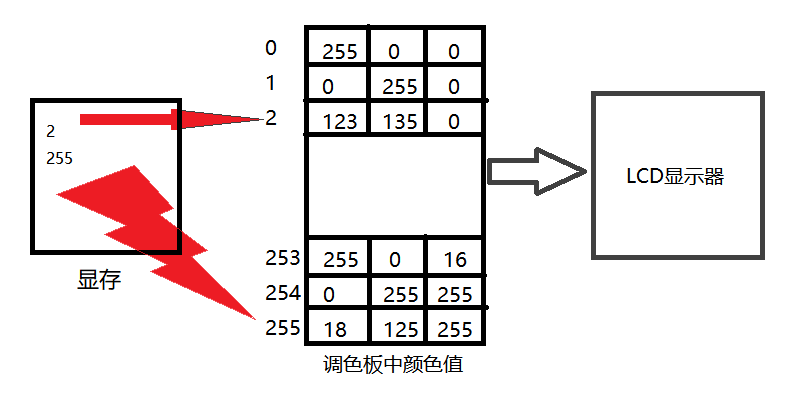
FB\_VISUAL\_MONO01和FB\_VISUAL\_MONO10描述的是单色显示器的像素值与颜色的映射关系，早期显示器只支持2种颜色：前景色和背景色。通常用黑白色来代称。FB\_VISUAL\_MONO01和FB\_VISUAL\_MONO10指刚好相反的两种前景色和背景色。

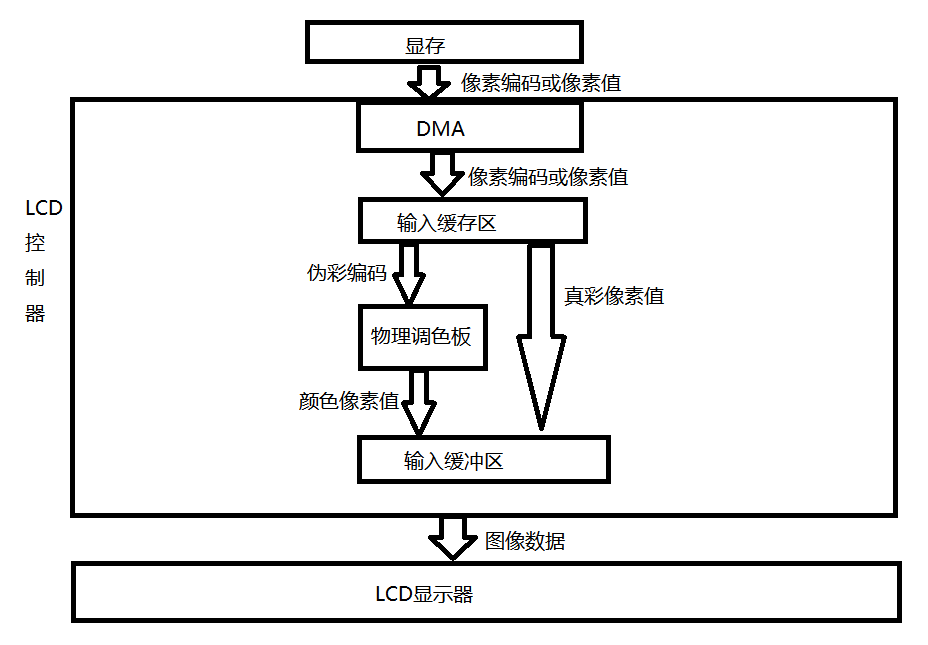
FB\_VISUAL\_STATIC\_PSEUDOCOLOR和FB\_VISUAL\_PSEUDOCOLOR支持小数量的彩色，但是早期受到内存大小的约束，为了显示丰富的彩色图像，使用了硬件调色板，在描述相同颜色数量时可以节省内存空间，这种方式被称为伪彩模式。FB\_VISUAL\_STATIC\_PSEUDOCOLOR代表静态伪彩，FB\_VISUAL\_PSEUDOCOLOR代表动态伪彩。静态伪彩的调色板内容不可修改，动态伪彩可以通过驱动修改调色板的值。

FB\_VISUAL\_DIRECTCOLOR和FB\_VISUAL\_TRUECOLOR，随着内存和显示技术的发展，已经不再限制一帧图像中可以显示的颜色数了，这两种情况都被称为真彩，比如我们可以直接分配1024\*600\*24/8的内存来存放像素值。嵌入式系统中RGB 5:6:5也可以被看作真彩。

调色板原理：

调色板是为了节约空间所用的，只有图片的颜色数小于等于256色的时候才使用,16位高彩和24位32位真彩是没有调色板的，但是在写16bpp或24bpp真彩的LCD驱动程序里，为了兼容以前的驱动程序，还需要提供一个假的调色板。驱动程序在使用调色板前，会在内存中分配256个单元用来存放256种颜色像素值。当显示的图像是伪彩时，驱动会先把调色板内存中的颜色值载入到物理调色板中。在DMA的作用下，显存中的像素值编码传递到显示控制器的输入缓存区，这些编码其实就是一些索引(每个索引值占1个字节，因为是256种)，用来调用调色板对应单元中的颜色像素值。





上图说明，如果在真彩模式下显存中直接存放的是实际像素值，而在伪彩模式下显存存放的是像素的索引值，例如BPP为16则每个像素占2个字节，而在256种调色板情况下，每个索引值只占1字节。当一幅颜色数为256色的图片处理完成后，下一帧图像的颜色与正在显示图像颜色范围不同时，我们只需要修改调色板中的某些选项。

DMA原理：

Direct Memory Access（存储器直接访问）。这是指一种高速的数据传输操作，允许在外部设备和存储器之间直接读写数据，不需要通过CPU控制。整个数据传输操作在一个称为"DMA控制器"的控制下进行的。CPU除了在数据传输开始和结束时做一点处理外，在传输过程中CPU可以进行其他的工作。这样，在大部分时间里，CPU和输入输出都处于并行操作。因此，使整个计算机系统的效率大大提高。

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

功能：通过一致性DMA映射的方式来申请空间

参数1：设备结构体指针，可以不使用

参数2：分配的空间(显存)大小，1024\*600\*2

参数3：用来存放获取的物理地址

参数4：GFP\_KERNEL

返回值：显存的虚拟地址，和参数3对应

什么是一致性DMA映射？

DMA(直接内存存取)传输是不通过CPU的，经过DMA操作，cache缓存对应的内存数据可能已经被修改了，而CPU本身不知道，它仍然认为cache中的数据就是内存中的数据，以后访问Cache映射的内存时，它仍然使用旧的Cache数据。这样就发生Cache与内存的数据“不一致性”错误。

DMA的映射分为：一致性DMA映射和流式DMA映射。

一致性DMA映射申请的缓存区能够使用cache，并且保持一致性。一致性映射具有很长的生命周期，在这段时间内占用的映射寄存器，即使不使用也不会释放，生命周期为该驱动的生命周期。

流式DMA映射生命周期比较短，而且禁用cache。一些硬件对流式映射有优化。建立流式DMA映射，需要告诉内核数据的流动方向。

因为LCD随时都在使用，因此在Framebuffer驱动中，使用一致性DMA映射来提高效率。

———————————————————————————————————————————————————————————————————————————

对底层硬件操作的函数指针。

struct fb\_ops {

    struct module \*owner;

    int (\*fb\_check\_var)(struct fb\_var\_screeninfo \*var, struct fb\_info \*info);//检查可变参数并进行设置

    int (\*fb\_set\_par)(struct fb\_info \*info);    //根据设置的值进行更新，使之有效

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

       unsigned blue, unsigned transp, struct fb\_info \*info);   //设置颜色寄存器

    int (\*fb\_blank)(int blank, struct fb\_info \*info);   //实现LCD背光开关控制

    void (\*fb\_fillrect) (struct fb\_info \*info, const struct fb\_fillrect \*rect);  //矩形填充

    void (\*fb\_copyarea) (struct fb\_info \*info, const struct fb\_copyarea \*region);    //复制数据

    void (\*fb\_imageblit) (struct fb\_info \*info, const struct fb\_image \*image);  //图形填充

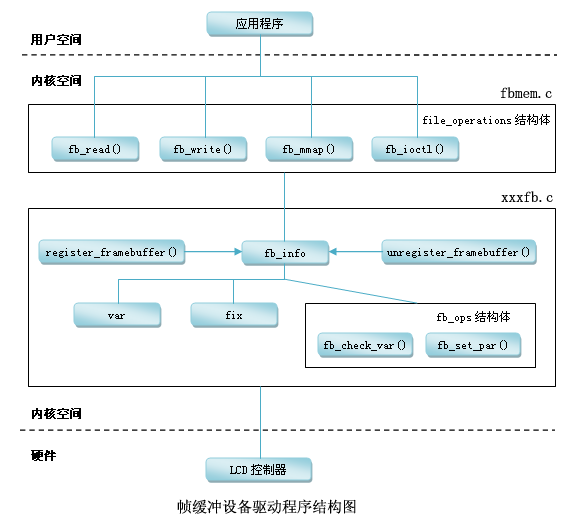
};

接口：

int register\_framebuffer(struct fb\_info \*fb\_info) 注册fb\_info结构体

struct fb\_info \*framebuffer\_alloc(size\_t size, struct device \*dev) 申请空间

三、帧缓冲框架结构



帧缓冲机制在Linux中也可以是一个完整的子系统，大体由fbmem.c和xxxfb.c组成。向上给应用程序提供完善的设备文件操作接口，接口在Linux提供的fbmem.c文件中实现；向下提供了硬件操作的接口，只是这些接口Linux并没有提供实现，要根据具体的LCD控制器硬件进行设置，所以这就是我们要做的事情了(即xxxfb.c部分的实现)。xxxfb.c可以参考s3c2410fb.c

四、帧缓冲源码分析

fbmem.c源码分析：

static int \_\_init fbmem\_init(void) { proc\_create("fb", 0, NULL, &fb\_proc\_fops);///向 proc文件系统报告驱动状态和参数 if (register\_chrdev(FB\_MAJOR,"fb",&fb\_fops))///注册字符设备驱动，主设备号是29 printk("unable to get major %d for fb devs\n", FB\_MAJOR); fb\_class = class\_create(THIS\_MODULE, "graphics");//创建 /sys/class/graphics设备类配合mdev生成设备文件 return 0; }

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, .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。也就是说应用层调用了read、write、ioctl等函数会调用fbmem.c中的fb\_ops结构中的read、write、ioctl等接口。

接下来我们分别跟进fbmem.c中的read、write、ioctl等接口发现，都是通过次设备号在已注册的fb\_info结构体指针数组(struct fb\_info \*registered\_fb[FB\_MAX])中找到匹配的那一个结构体指针之后，判断其中的fbops结构中的操作函数是否有定义，有的话就优先调用该函数，没有就使用往下执行。这样的好处就是多个相同的LCD设备可以使用同一套代码，减少代码的重复性，同时对于需要特殊定义的函数又可以方便实现重定义。

既然需要从registered\_fb[FB\_MAX]数组中获取struct fb\_info指针，所以我们必须要在自己写的驱动中申请struct fb\_info结构体空间，并且初始化这个结构体，最后注册到内核中。操作函数如下:

申请空间：struct fb\_info \*framebuffer\_alloc(size\_t size, struct device \*dev)

此函数在drivers/video/fbsysfs.c文件中

注册fb\_info结构体:int register\_framebuffer(struct fb\_info \*fb\_info)

1771 int register\_framebuffer(struct fb\_info \*fb\_info)

1773 {

1774 int ret;

1776 mutex\_lock(&registration\_lock);

1777 ret = do\_register\_framebuffer(fb\_info);

1778 mutex\_unlock(&registration\_lock);

1779

1780 return ret;

1781 }

1611 static int do\_register\_framebuffer(struct fb\_info \*fb\_info)

{

1629 num\_registered\_fb++;//用于统计已注册设备个数

1638 fb\_info->dev = device\_create(fb\_class, fb\_info->device,

1639 MKDEV(FB\_MAJOR, i), NULL, "fb%d", i); //在/dev下创建设备文件

...................................

中间省略的代码功能就是给fb\_info初始化

...................................

1675 registered\_fb[i] = fb\_info; 将配置好的fb\_info结构体首地址存放到registered\_fb数组中

1677 event.info = fb\_info;

.........................................

1684 fb\_notifier\_call\_chain(FB\_EVENT\_FB\_REGISTERED, &event);//通知帧缓冲区控制台，有一个新的帧缓冲区设备被注册到内核中来了

}

使用设备树实现lcd驱动程序如下：

设备树添加如下：

lcd{

compatible = "fs4412,lcd";

reg = <0x11400000 0x01e4 0x1003c234 0x4

0x1003c534 0x4 0x10010210 0x8 0x11c00000 0x184>;

};