网名“鱼树”的学员聂龙浩，

学习“韦东山 Linux 视频第 2 期”时所写的笔记很详细，供大家参考。

也许有错漏，请自行分辨。

目录

LCD 驱动框架：

fbmem.c <--- 底层硬件驱动提供

fb_info 结构

字符驱动程序编写：
APP: open,　　read,　　write.　----> 对应提供驱动程序的读写等函数。

驱动：drv_open, drv_read,　drv_write

硬件

代码步骤：

1,**确定主设备号：**

可以自己确定，也可让内核分配。

2，**要构造驱动中的**“open,read,write **等**”

是将它们放在一个“file_operations”结构体中.
File_operations==> .open, .read, .write, .poll 等。
这里"open"函数会去配置硬件的相关引脚等，还有注册中断。

3，register_chrdev **注册字符设备**:

使用这个 file_operations 结构体。
Register_chrdev(主设备号，设备号，file_operations 结构）.

4，**入口函数：**

调用这个“register_chrdev()”函数。

5，**出口函数：**

看内核中的 LCD 如何写，融入自己的代码：fbmem.c 是内核自带的 LCD 驱动程序。基于分层的思想也是抽出了共性的内容。

Fbmem.c 分析：

1，先看“入口函数”：

int __init fbmem_init(void)
  -->register_chrdev(FB_MAJOR,"fb",&fb_fops) 注册"file_operations”结构体“fb_fops”。

主设备号：

```
#define FB_MAJOR        29
```

File_operations 结构体：

```
static const struct file_operations fb_fops = {
    .owner =    THIS_MODULE,
    .read =     fb_read,
    .write =    fb_write,
    .ioctl =    fb_ioctl,
#ifdef CONFIG_COMPAT
    .compat_ioctl = fb_compat_ioctl,
#endif
    .mmap =     fb_mmap,
    .open =     fb_open,
    .release =  fb_release,
#ifdef HAVE_ARCH_FB_UNMAPPED_AREA
    .get_unmapped_area = get_fb_unmapped_area,
#endif
#ifdef CONFIG_FB_DEFERRED_IO
    .fsync =    fb_deferred_io_fsync,
#endif
};
```

因为“fbmem.c”是通用的文件，故并不能直接使用这个 file_operations 结构中的.read 等函数。

int __init fbmem_init(void)
  -->register_chrdev(FB_MAJOR,"fb",&fb_fops)
  -->fb_class = class_create(THIS_MODULE, "graphics");//创建类（是额外代码自动创建设备节点。）
  这里 fbmem.c 没有在设备类下创建设备，只有真正有硬件设备时才有必要在这个类下去创建设备。，在“register_framebuffer()”中可以看到创建设备“fb_info->dev = device_create(fb_class, fb_info->device,MKDEV(FB_MAJOR, i), "fb%d", i);”

假设

app: open("/dev/fb0", ...) 假设 APP 打开一个主设备号: 29, 次设备号: 0 的设备时，最终会找到 file_operations fb_fops 结构中的“.open = fb_open,”函数（其中用到：registered_fb[iminor(inode)]数组）。

Int fb_open(struct inode *inode, struct file *file)

  -->int fbidx = iminor(inode); //iminor(inode)得到这个设备节点的“次设备号”。

  -->struct fb_info *info; //帧缓冲区结构体。

  -->info = registered_fb[fbidx];//即 struct fb_info *info = registered_fb[fbidx]; 假设“次设备号”为 0.即：

     struct fb_info *info = registered_fb[iminor(inode)] = registered_fb[0]。从这个 registered_fb[]数组里得到“以次设备号为 0 为下标”的一项。

  -->info->fbops->fb_open //若这个 info = registered_fd[0] 的“fbops”有“fb_open”函数时就：

     -->res = info->fbops->fb_open(info,1);

简明过程：

kernel:

        fb_open

                        int fbidx = iminor(inode);

                        struct fb_info *info = = registered_fb[0];

//fb 是 frame buffer 帧缓冲区

App:read() ：想知道内存上有什么内容。看“file_operations fb_fops”结构中的“.read = fb_read”

ssize_t fb_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)

-->struct fb_info *info = registered_fb[iminor(inode)];以次设备号为下标从 registered_fd 数组中得一项赋给"info"结构。

-->info->fbops->fb_read //若这个 info 数组项中提供了“fbops”结构的“fb_read”读函数时：就调用此读函数。

  -->return info->fbops->fb_read(info, buf, count, ppos);

-->若没有提供"info"项，接着就：total_size = info->screen_size;（屏幕大小）等。

-->buffer = kmalloc((count > PAGE_SIZE) ? PAGE_SIZE : count, GFP_KERNEL);//分配一个缓冲区。

-->src = (u32 __iomem *) (info->screen_base + p); //screen_base 是指显存的基地址。这里是读源 src 等于显存的
基地址加上某个偏移值。

-->*dst++ = fb_readl(src++);//读源（从显存基地址+P 偏移）那里读到一个数据放到目标“*dst++”里。dst 是 buffer，buffer 是 kmalloc()上面分配的空间。
-->copy_to_user(buf, buffer, c); //把数据拷贝到用户空间。

总结：

就是说有"if (info->fbops->fb_read)"函数时就从读函数中读“info->fbops->fb_read(info, buf, count, ppos);”；
若没有则从“src = (u32 __iomem *) (info->screen_base + p);”里读（从 screen_base 显存基地址加一个 P 偏移处）。
从这个“src”源处读到用 kmalloc（）分配的一个目标地址“dst”中（*dst++ = fb_readl(src++);）。最后“copy_to_user(buf, buffer, c)”把读到的数据拷贝到用户空间。

简明过程：

kernel:

```
fb_read
int fbidx = iminor(inode);
struct fb_info *info = registered_fb[fbidx];
if (info->fbops->fb_read)
return info->fbops->fb_read(info, buf, count, ppos);

src = (u32 __iomem *) (info->screen_base + p);
dst = buffer;
*dst++ = fb_readl(src++);
copy_to_user(buf, buffer, c)
```

从上面知道“.open”和“.read”都依赖一个结构体 fb_info 结构体。这个结构体是从数组“registered_fd[]”经“次设备号”为下标得到一项为 fd_info 结构体。

```
“struct fb_info *info = registered_fb[fbidx]”。
```

registered_fb 这个 fb_info 结构数组的定义：

Fbmem.c 提供的都是抽象出来的东西。最终都得依赖这个“registered_fb”数组里的“fb_info”结构体。

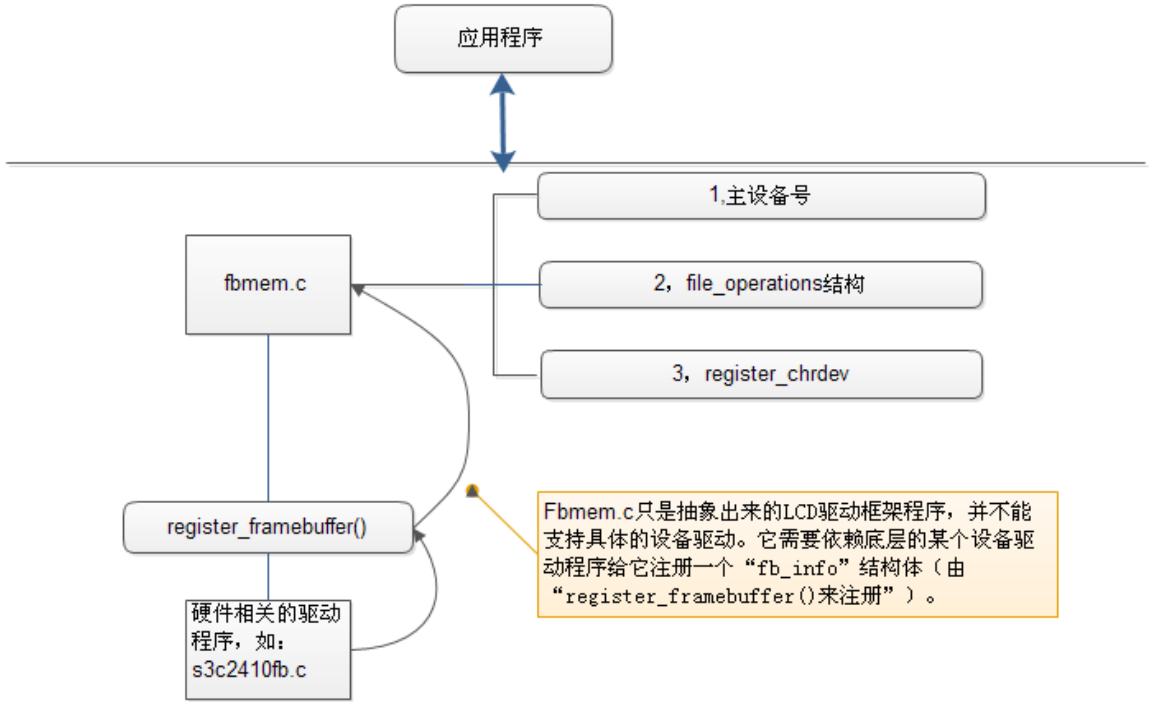
搜索源代码找“registered_fb”这个 fb_info 结构数组的出处。

1,“registered_fd”bf_inof 结构数组的定义：

```
struct fb_info *registered_fb[FB_MAX] __read_mostly;
```

2,“registered_fd”bf_inof 结构数组的设置：

```
Int register_framebuffer(struct fb_info *fb_info); //注册framebuffer。
    -->registered_fb[i] = fb_info;
```



这样分层之后，APP 就知道读写函数里面的形参是什么含义了。Ioctl()要传什么参数就固定下来了。

分析“register_framebuffer ()”:

```
Int register_framebuffer(struct fb_info *fb_info)
  -->if (!registered_fb[i]) //先找出一个空项。
  -->fb_info->dev = device_create(fb_class, fb_info->device,
MKDEV(FB_MAJOR, i), "fb%d", i);
```

在“fd_class”类下面创建设备。只有真正有硬件设备时才有必要在这个类下去创建设备。这样 mdev 或 udev 才能去自动创建设备节点。Fbmem.c 只是抽象出来的 LCD 驱动框架程序，并不能支持具体的驱动。它需要依赖底层的某个驱 动程序给它注册一个“fb_info”结构体（由“register_framebuffer()来注册”）。

搜索内核会发现有各种 LCD 驱动程序调用这个“register_framebuffer()”（如 6832fb.c、amifb.c、atmel_lcdfb.c、还有 2410 的如 s3fb.c、s3c2410fb.c 等）。所以想要用"fbmem.c"这一套代码时，就要按上图的框架来写代码，要自己定义底层的硬件驱动程序（如上面内核中有 s3c2410fb.c 这个 LCD 底层驱动程序。）

问 1. registered_fb 在哪里被设置？

答 1. register_framebuffer

分析层底硬件驱动程序

如：inux/drivers/video/s3c2410fb.c（从入口函数看起）
__devinit s3c2410fb_init(void)
  -->platform_driver_register(&s3c2410fb_driver);注册一个平台驱动（总线设备驱动模型时关心.probe 函数）

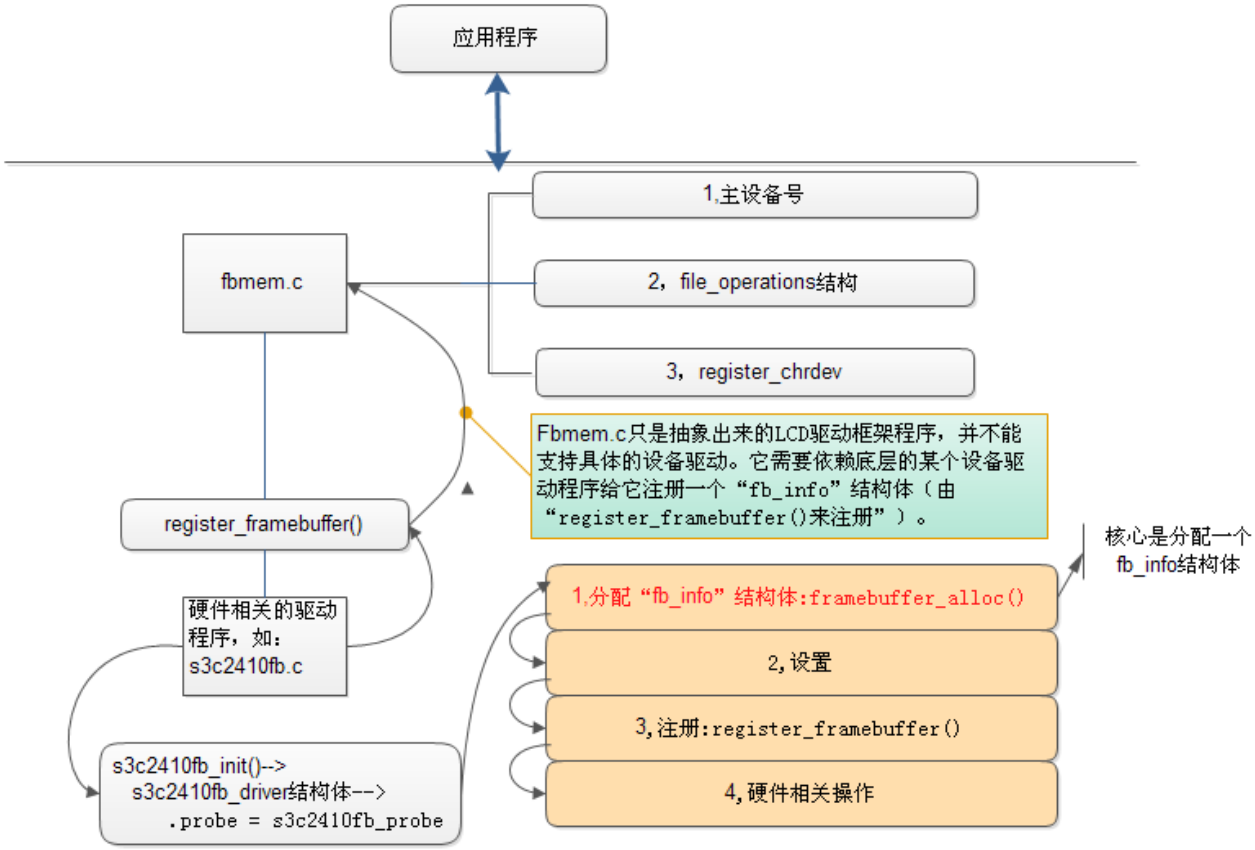
```
struct platform_driver s3c2410fb_driver
   -->.probe                = s3c2410fb_probe, //若内核中有同名"s3c2410-lcd"的平台设备就调用
"s3c2410fb_driver"函数。
   -->.driver               = {.name = "s3c2410-lcd",.owner = THIS_MODULE,},

      int __init s3c2410fb_probe(struct platform_device *pdev)  根据平台设备pdev来获得相关硬件信息。
         -->mach_info = pdev->dev.platform_data;  根据pdev获得"mach_info"信息.
         -->irq = platform_get_irq(pdev, 0);       根据pdev获得"irq"中断信息.
         -->struct fb_info *fbinfo = framebuffer_alloc(sizeof(struct s3c2410fb_info), &pdev->dev);
```

分配一个"bf_info"结构。接着就开始设置这个 bf_info 结构体.
      -->

```
info = fbinfo->par;
info->fb = fbinfo;
info->dev = &pdev->dev;
       -->硬件相关设置：
info->regs.lcdcon1 &= ~S3C2410_LCDCON1_ENVID;
lcdcon1 = readl(S3C2410_LCDCON1);
writel(lcdcon1 & ~S3C2410_LCDCON1_ENVID, S3C2410_LCDCON1);
  -->硬件相关的操作：
// add by thisway.diy@163.com, for eBlocks
s3c2410_gpio_setpin(S3C2410_GPB0, 0);                // back light control

info->mach_info                                 = pdev->dev.platform_data;

fbinfo->fix.type                    = FB_TYPE_PACKED_PIXELS;
```

```
fbinfo->fix.type_aux          = 0;
fbinfo->fix.xpanstep          = 0;
fbinfo->fix.ypanstep          = 0;
fbinfo->fix.ywrapstep         = 0;
fbinfo->fix.accel             = FB_ACCEL_NONE;

fbinfo->var.nonstd            = 0;
fbinfo->var.activate          = FB_ACTIVATE_NOW;
fbinfo->var.height            = mach_info->height;
fbinfo->var.width             = mach_info->width;
fbinfo->var.accel_flags       = 0;
fbinfo->var.vmode             = FB_VMODE_NONINTERLACED;

fbinfo->fbops                 = &s3c2410fb_ops;
fbinfo->flags                 = FBINFO_FLAG_DEFAULT;
fbinfo->pseudo_palette        = &info->pseudo_pal;

fbinfo->var.xres              = mach_info->xres.defval;
fbinfo->var.xres_virtual      = mach_info->xres.defval;
fbinfo->var.yres              = mach_info->yres.defval;
fbinfo->var.yres_virtual      = mach_info->yres.defval;
fbinfo->var.bits_per_pixel    = mach_info->bpp.defval;

fbinfo->var.upper_margin      = S3C2410_LCDCON2_GET_VBPD(mregs->lcdcon2) + 1;
fbinfo->var.lower_margin      = S3C2410_LCDCON2_GET_VFPD(mregs->lcdcon2) + 1;
fbinfo->var.vsync_len         = S3C2410_LCDCON2_GET_VSPW(mregs->lcdcon2) + 1;

fbinfo->var.left_margin       = S3C2410_LCDCON3_GET_HFPD(mregs->lcdcon3) + 1;
fbinfo->var.right_margin      = S3C2410_LCDCON3_GET_HBPD(mregs->lcdcon3) + 1;
fbinfo->var.hsync_len         = S3C2410_LCDCON4_GET_HSPW(mregs->lcdcon4) + 1;

fbinfo->var.red.offset        = 11;
fbinfo->var.green.offset      = 5;
fbinfo->var.blue.offset       = 0;
fbinfo->var.transp.offset     = 0;
fbinfo->var.red.length        = 5;
fbinfo->var.green.length      = 6;
fbinfo->var.blue.length       = 5;
fbinfo->var.transp.length     = 0;
fbinfo->fix.smem_len          = mach_info->xres.max *
mach_info->yres.max *
mach_info->bpp.max / 8;
```

```
。。。。。。。
/* Initialize video memory */
ret = s3c2410fb_map_video_memory(info);
if (ret) {
printk( KERN_ERR "Failed to allocate video RAM: %d\n", ret);
ret = -ENOMEM;
goto release_clock;
}
-->ret = register_framebuffer(fbinfo); //注册fb_info结构体
```



总结上面的过程：抽象出驱动程序：怎么写 LCD 驱动程序？

1. 分配一个 fb_info 结构体：framebuffer_alloc
2. 设置
3. 注册：register_framebuffer
4. 硬件相关的操作

要得到 LCD 的分辨率等信息：从上往下分析（fbmem.c 开始）

Fbmem.c: 看“.ioctl”。

```
int __init fbmem_init(void)
  -->register_chrdev(FB_MAJOR,"fb",&fb_fops) //看 fb_ops 结构
（file_operations）。
  -->
struct file_operations fb_fops = {
.. .. ..
.ioctl =                fb_ioctl, //看 fb_ioctrl()中获得什么内容.
.. .. ..
}
   -->
int fb_ioctl(struct inode *inode, struct file *file, unsigned int
cmd,unsigned long arg) //有各各 cmd
  -->以次设备号为下标,在 registered_fb[]数组中得到 fb_info 结构体变量
"info"。
int fbidx = iminor(inode);
struct fb_info *info = registered_fb[fbidx];
  --> 将这个 info 结构中的 var 成员拷贝回用户空间。
switch (cmd) {
case FBIOGET_VSCREENINFO: //GET 获得.V(var)可变的.SCREEN 屏幕.INFO 信
息.
return copy_to_user(argp, &info->var,
    sizeof(var)) ? -EFAULT : 0;
```

看 info->var 这个成员中有什么内容：

```
struct fb_info {
    int node;
    int flags;
    struct fb_var_screeninfo var;    /* Current var */
    struct fb_fix_screeninfo fix;    /* Current fix */
    struct fb_monspecs monspecs;     /* Current Monitor specs */
    struct work_struct queue;    /* Framebuffer event queue */
    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;   /* current mode */
```

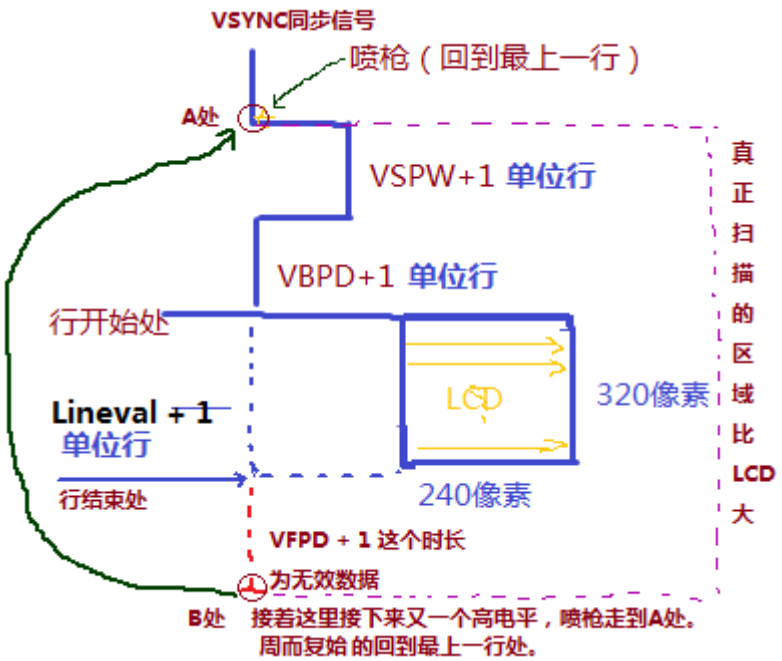
```
struct fb_var_screeninfo {
    __u32 xres;           /* visible resolution:x,y方向的分辨率.   */
    __u32 yres;
    __u32 xres_virtual;       /* virtual resolution          */
    __u32 yres_virtual;
    __u32 xoffset;            /* offset from virtual to visible */
    __u32 yoffset;            /* resolution                  */

    __u32 bits_per_pixel;         /* guess what              */
    __u32 grayscale;          /* != 0 Graylevels instead of colors */
```

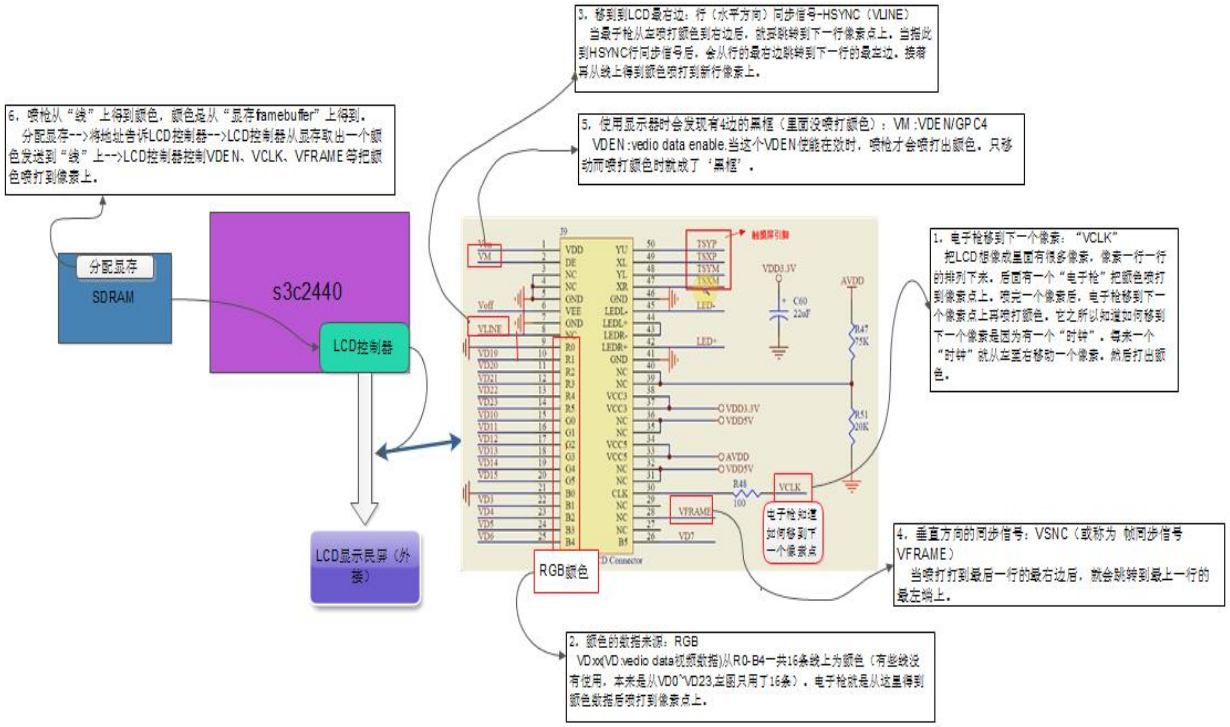
分析 fbmem.c 中的 LCD 分辨率也证实了 fbmem.c 是抽象出来的内容，最终它得依赖具体的底层设备驱动提供的“fb_info”结构体。

LCD 硬件操作步骤：

在写驱动之前，先看硬件的具体操作：

参考以前的 LCD 裸机程序笔记。





设置 LCD 控制器时，不同的 LCD 的"VCLK"(频率）可能不会一样，所以要调整 LCD 控制器来设置频率的快和慢，使其发出合适的时钟（若 1 秒一次，则喷打颜色到像素点时都看的清晰太慢了。若 VCLK 为 1GHz，可能又反应不过来。所以要调 LCD 控制器使其发出合适的时钟）。

发出使命的“时钟”要看 LCD 屏的硬件手册。

硬件操作过程：

1，根据 LCD 手册设置 LCD 控制器。（时钟参数等）

2，分配显存，并将显存的地址告诉 LCD 控制器。

还要说明颜色格式。一个像素用多少个字节来表示等。

3，之后就是 2440 相关的操作:配置引脚用于 LCD。芯片的管脚可以配置成输入输出，还可以配置为 LCD 引脚。

LCD 硬件参数设置：

1，头文件：

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/string.h>
#include <linux/mm.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/fb.h>
#include <linux/init.h>
#include <linux/dma-mapping.h>
#include <linux/interrupt.h>
#include <linux/workqueue.h>
#include <linux/wait.h>
#include <linux/platform_device.h>
#include <linux/clk.h>

#include <asm/io.h>
#include <asm/uaccess.h>
#include <asm/div64.h>

#include <asm/mach/map.h>
#include <asm/arch/regs-lcd.h>
#include <asm/arch/regs-gpio.h>
#include <asm/arch/fb.h>
```


2，入口、出口函数以及修饰它们的框架：

1.1,先写入口函数：

```
static int lcd_init(void)
{
return 0;
}
```

1.2,出口函数;

```
static void lcd_exit()
{
}
```

1.3，修饰入口中与出口函数.

```
module_init(lcd_init);
module_exit(lcd_exit);

MODULE_LICENSE("GPL");

static int lcd_init(void)
{
```

2.,分配一个 fb_info 结构体.

3.,设置 fb_info 结构体。

4.，硬件相关的设置。

因为一注册就要用到硬件相关的设置信息,所以这一步放在前面.

5.,注册

```
return 0;
}
```

3.1，分配一个 fb_info 结构体：

```
static struct fb_info *s3c_lcd; //定义一个 fb_info 结构体变量 s3c_lcd.
s3c_led = framebuffer_alloc(0, NULL);//可能内存不足要判断返回值.
```

分析：struct fb_info *framebuffer_alloc(size_t size, struct device *dev)原型：

参 1 为大小。在定义 fb_info 结构时，结构体的大小是固定了的。内核中经常有这个取巧的方法，本来一个结构体原本分配的大小，而内核紧接着再分配了一段大小给这个结构体。定义时的结构体里面有一个指针，指向这个内核又分配的一段空间，这个空间里放“私有数据”。

struct fb_info *framebuffer_alloc(size_t size, struct device *dev)
-->传入了一个“size”大小，本来 fb_info 结构体的大小如下：

int fb_info_size = sizeof(struct fb_info);
-->然后传入的 size 大小与原本定义的“fb_info”结构体大小相加：分配这一段 p 空间。
p = kzalloc(fb_info_size + size, GFP_KERNEL);
-->然后 par 指向额外分配的那段 size 空间。
info->par = p + fb_info_size;我们不需要这段额外空间就不设置。就把 size=0.

3.2,**注册这个** fb_info **结构变量** s3c_led **结构。**

```
//5.,注册
    //2.3,注册这个 fb_info 结构.
    register_framebuffer(s3c_lcd);
```

3.3,**设置“**fb_info**”结构体：**

①，先看 fb_info 结构的具体成员：fix,var,fbops 等。
②，设置固定的参数：

fb_info 结构定义中的“struct fb_fix_screeninfo fix;”

固定参数的内容：

```
struct fb_fix_screeninfo {
char id[16];                                    //名字，可随便取。
(s3c_lcd->fix.id, "mylcd");
unsigned long smem_start; //显存的起始地址.没分配显存前不知道.
/* (physical address) */
__u32 smem_len;    //显存的长度(看 LCD 手册可设)：
s3c_lcd->fix.smem_len = 240*320*16;
__u32 type;                    //s3c_lcd->fix.type     =
FB_TYPE_INTERLEAVED_PLANES;
__u32 type_aux;    //附加的类型,若是平板要用到这个附加的类型。
__u32 visual;                                                   /* see
FB_VISUAL_*                                 */
__u16 xpanstep;                                 /* zero if no hardware
panning  */
__u16 ypanstep;                                 /* zero if no hardware
panning  */
```

```
__u16 ywrapstep;                                /* zero if no
hardware ywrap    */
__u32 line_length;                               /* length of a line
in bytes    */
unsigned long mmio_start;                  /* Start of Memory Mapped
I/O   */
/* (physical address) */
__u32 mmio_len;                                                /*
Length of Memory Mapped I/O  */
__u32 accel;                                                  /*
Indicate to driver which                */
/*  specific chip/card we have                  */
__u16 reserved[3];                               /* Reserved for
future compatibility */
};
```

a,__u32 smem_len; 设置显存长度。

显存长度设置查看 LCD 手册“F:\embedded\第二期：深入驱动\源码_文档_图片_原理图_芯片手册\原理图及芯片手册\JZ2440v2\芯片手册/液晶屏.pdf”

分辨率：

(5) Resolution	240 x 3(R,G,B)(W) x 320 (H) dots

颜色：

(9) Number of Colors	262^K Colors (R,G,B 6Bit Digital each)

RGB 红绿蓝每个像素占 6bit，实际上在 2410 里面，RGB 只能是 R-5 位，G-6 位，B-5 位，不能是 666，一色里丢弃 1 位没关系。这是因为 2440 不支持 18 位，只支持 16 位或其他如 24 位等。但这里浪费了 2 位也没关系。

则显存的长度：分辨率*颜色位数---240*320*16(位)

```
s3c_lcd->fix.smem_len = 240*320*16;    //显存的长度,查LCD手册.
```

b,__u32 type:看 FB_TYPE_*宏。

```
#define FB_TYPE_PACKED_PIXELS         0    /* Packed Pixels     */
#define FB_TYPE_PLANES                1    /* Non interleaved planes */
#define FB_TYPE_INTERLEAVED_PLANES    2    /* Interleaved planes     */
#define FB_TYPE_TEXT                  3    /* Text/attributes   */
#define FB_TYPE_VGA_PLANES            4    /* EGA/VGA planes    */
```

```
#define
FB_TYPE_PACKED_PIXELS                                  0
  //默认值--0
```

```
#define
FB_TYPE_PLANES                                          1
          //planes--平台
#define FB_TYPE_INTERLEAVED_PLANES              2               /*
Interleaved planes              */
#define
FB_TYPE_TEXT                                            3
        //文本
#define
FB_TYPE_VGA_PLANES                              4               //
VGA 平台
```

此实例中是用“FB_TYPE_PACKED_PIXELS”。是个默认值。（一般看不懂时就使用默认值）。默认值就是可以支持大部分类的 LCD。

```
s3c_lcd->fix.type       = FB_TYPE_INTERLEAVED_PLANES; //默认值,此宏为0。默认值即支持大多类的LCD
   /*因为是0则此项可以不设置。p = kzalloc(fb_info_size + size, GFP_KERNEL);分配空间默认也填0.*/
```

c,__u32 type_aux:附加的类型
若是平板要用到这个附加的类型。但此实例中不用设置。

d,__u32 visual:查看宏“FB_VISUAL_*”

```
#define FB_VISUAL_MONO01                                0 /* Monochr.
1=Black 0=White MONO 单色，1 为黑，0 为白
#define FB_VISUAL_MONO10                                1 /* Monochr.
1=White 0=Black 或者单色，1 表示白，0 为黑。
#define FB_VISUAL_TRUECOLOR                             2 /* True
color：真彩色。
#define FB_VISUAL_PSEUDOCOLOR                           3 /*
Pseudo color (like atari) */
#define FB_VISUAL_DIRECTCOLOR                           4 /*
Direct color */
#define FB_VISUAL_STATIC_PSEUDOCOLOR            5 /* Pseudo color
readonly */
```

我们是 TFT 真彩色屏：

```
s3c_lcd->fix.visual     = FB_VISUAL_TRUECOLOR; //TFT屏是真彩色
```

```
e,__u16 xpanstep;               /* zero if no hardware panning 若没
有硬件的 panning(平移?)此值写为 0 */
  __u16 ypanstep;                       /* zero if no
hardware panning  */
```

__u16 ywrapstep; /* zero if no hardware ywrap */

上面三项都设置为 0.

f,__u32 line_length; /* length of a line in bytes */

固定信息里面，一行的长度。单位是 byte。此 LCD 是一行 240 个像素，一个像素 RGB 三色是 16 位(2 字节)。

g,unsigned long mmio_start; /* Start of Memory Mapped I/O */

__u32 mmio_len; /* Length of Memory Mapped I/O */

内存映射的端口，寄存器的地址等。先不用配置。若让应用程序直接访问寄存器时，可以设置这些内容。

h，__u32 accel; /* Indicate to driver which */

/* specific chip/card we have */

__u16 reserved[3]; /* Reserved for future compatibility */

上面的两项也不用配置。

最终固定信息设置结果为：

```
//3.,设置 fb_info  结构体。
    //3.1,设置fb_info结构中的固定参数信息:struct fb_fix_screeninfo fix.
    strcpy(s3c_lcd->fix.id, "mylcd"); //fix.id:名字，随便取.
    s3c_lcd->fix.smem_len = 240*320*16/8;    //显存的长度,查LCD手册.每像素占16位即2字节.
    s3c_lcd->fix.type     = FB_TYPE_INTERLEAVED_PLANES; //默认值,此宏为0。默认值即支持大多类的LCD
        /*因为是0则此项可以不设置。p = kzalloc(fb_info_size + size, GFP_KERNEL);分配空间默认也填0.*/
    s3c_lcd->fix.visual   = FB_VISUAL_TRUECOLOR; //TFT屏是真彩色.
    s3c_lcd->fix.line_length = 240*2;     //固定信息里面，一行的长度。单位是byte。
                                          //此LCD是一行240个像素，一个像素RGB三色是16位(2字节)。
```

只有一个“显存”的起始地址没有设置，在之后分配显存的时候再去设置它。

③，设置可变的参数：

fb_info 结构定义中的“struct fb_var_screeninfo var;”

可变信息的结构定义：

struct fb_var_screeninfo {

__u32 xres; /、visible resolution:x 方向的分辨率。

__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 */
};
```

A, __u32 xres:X 方向的分辨率。

__u32 yres:Y 方向的分辨率。

```
s3c_lcd->var.xres = 240; // x方向的分辨率为240
s3c_lcd->var.yres = 320; // y方向的分辨率为320
```

B，__u32 xres_virtual：虚拟分辨率。

买回的 LCD 分辨率固定死了，但还能在 PC 桌面上右键设置虚拟分辨率。

__u32 yres_virtual：这里 x,y 方向的虚拟分辨率都设置成和实际的分辨率一样。

```
s3c_lcd->var.xres_virtual = 240; // x方向的虚拟分辨率，设置成和实际的x方向一样。
s3c_lcd->var.yres_virtual = 320; // y方向的虚拟分辨率，设置成和实际的y方向一样。
```

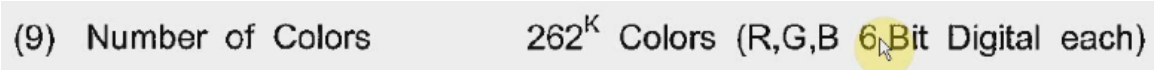
C, __u32 xoffset:虚拟和实际分辨率之间的偏移值（差值）。

__u32 yoffset：

上面设置了 x，y 方向上的实际和虚拟的分辨率是相同的，所以这两个项设置为 0 即可。因为是 0 则此项可以不设置。p = kzalloc(fb_info_size + size, GFP_KERNEL);分配空间默认也填 0

D,__u32 bits_per_pixel:每个像素用多少位。

(此项重要)。这里从 LCD 手册知道是 16 位。



RGB 红绿蓝每个像素占 6bit，实际上在 2410 里面，RGB 只能是 R-5 位，G-6 位，B-5 位，不能是 666，丢弃 1 位没关系。2440 里面不支持 18，只支持 16 位或其他的 24 位等。

```
s3c_lcd->fix.smem_len = 240*320*16/8;    //显存的长度,查LCD手册.每像素占16位即2字节.
```

E,__u32 grayscale：灰度值，这里不用设置。

F,struct fb_bitfield red:

红 bitfield：位区域。（bit--位，field--区域）
struct fb_bitfield green:　绿
struct fb_bitfield blue:　蓝
struct fb_bitfield transp:　透明度（这里没有）
红绿蓝分别在每像素 16 位里面从哪一位开始，三者分别多长。一般来说是：
RGB--565　分别这样对位。
所以 R 是从 11bit 开始。

```
struct fb_bitfield {
     __u32 offset;              /* beginning of bitfield  */
     __u32 length;              /* length of bitfield          */
     __u32 msb_right;           /* != 0 : Most significant bit is */
                    /* right */
};
```

从“fb_bitfield”结构中，offset 指颜色的偏移；length 是指颜色占的位数；最重要的一位在右边（msb_right 不等于 0，则最重要的位在右边），但我们这里最重要的一位在左边：
RGB:565 (左<--千<--百<--十<--个<--右边)
所以此项这里“msb_right”不用设置。

```
s3c_lcd->var.red.offset   = 11; //RGB--565:红色的偏移值从bit11开始.
s3c_lcd->var.red.length   = 5;  //RGB--565,则红色占5位.

s3c_lcd->var.green.offset = 5; //RGB--565:绿色的偏移值从bit5开始.
s3c_lcd->var.green.length = 6;  //RGB--565,则绿色占6位.

s3c_lcd->var.blue.offset  = 0; //RGB--565:蓝色的偏移值从bit0开始.
s3c_lcd->var.blue.length  = 5;  //RGB--565,则蓝色占5位.
```

G,__u32 nonstd：non-非，std-标准。我们是标准的，这里不用设置。

H,__u32 activate：查看 FB_ACTIVATE_* 宏有如下设置。

当不明白时就直接用默认值。

```
#define FB_ACTIVATE_NOW     0    /* set values immediately (or vbl)*/
#define FB_ACTIVATE_NXTOPEN 1    /* activate on next open  */
#define FB_ACTIVATE_TEST    2    /* don't set, round up impossible */
#define FB_ACTIVATE_MASK       15      不明白就用默认值 0 。
                    /* values          */
#define FB_ACTIVATE_VBL        16    /* activate values on next vbl  */
#define FB_CHANGE_CMAP_VBL     32    /* change colormap on vbl */
#define FB_ACTIVATE_ALL        64    /* change all VCs on this fb    */
#define FB_ACTIVATE_FORCE     128    /* force apply even when no change*/
#define FB_ACTIVATE_INV_MODE  256         /* invalidate videomode */
```

```
s3c_lcd->var.activate = FB_ACTIVATE_NOW; //直接用默认值:#define FB_ACTIVATE_NOW      0
```


I,__u32 height : height of picture in mm

  __u32 width : width of picture in mm
高度 和 宽度，以毫秒为单位。应该是 LCD 的物理尺寸。不关心可以不写，只是写了这里没什么用处。

J,__u32 accel_flags :

(OBSOLETE--过时) see fb_info.flags
过时了这里也不配置了。

K,下面都不用配置：

只是给应用程序去设置的。有真实的数据可以写进去，但没什么用处。
/* Timing: All values in pixclocks, except pixclock (of course) */
__u32 pixclock;   /* pixel clock in ps (pico seconds) */ 像素时钟频率。

```
__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;
```

左右上下的那个黑框，也不需要配置。

```
__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 */
```


最后可变参数的设置如下：

```
//3.2,设置可变的参数：fb_info 结构定义中的“struct fb_var_screeninfo var;”
s3c_lcd->var.xres = 240; // x方向的分辨率为240.
s3c_lcd->var.yres = 320; // y方向的分辨率为320
s3c_lcd->var.xres_virtual    = 240; // x方向的虚拟分辨率，设置成和实际的x方向一样。
s3c_lcd->var.yres_virtual    = 320; // y方向的虚拟分辨率，设置成和实际的y方向一样.
s3c_lcd->var.bits_per_pixel = 16;//LCD手册是RGB分别6位共18位，但2440不支持18位故丢弃2位.

s3c_lcd->var.red.offset    = 11; //RGB--565:红色的偏移值从bit11开始.
s3c_lcd->var.red.length    = 5;   //RGB--565,则红色占5位.

s3c_lcd->var.green.offset = 5; //RGB--565:绿色的偏移值从bit5开始.
s3c_lcd->var.green.length = 6;   //RGB--565,则绿色占6位.

s3c_lcd->var.blue.offset   = 0; //RGB--565:蓝色的偏移值从bit0开始.
s3c_lcd->var.blue.length   = 5;   //RGB--565,则蓝色占5位.

s3c_lcd->var.activate = FB_ACTIVATE_NOW; //直接用默认值:#define FB_ACTIVATE_NOW      0
```

④，设置操作函数：

fb_info 结构定义中的"struct fb_ops *fbops;".
找一个其他 LCD 的驱动代码，从入口函数看起，如：drivers/video/68328fb.c 中，fbops 的使用如下：

```
Fb_info.fbops = &mc68x32fb_ops;
  -->static struct fb_ops mc68x328fb_ops = {
.fb_check_var                = mc68x328fb_check_var,
.fb_set_par                 = mc68x328fb_set_par,
.fb_setcolreg                = mc68x328fb_setcolreg,
.fb_pan_display                = mc68x328fb_pan_display,
.fb_fillrect                 = cfb_fillrect,
.fb_copyarea                 = cfb_copyarea,
.fb_imageblit                 = cfb_imageblit,//留心这个三个函数.好像
每个 fb_ops 里都有它们。
.fb_mmap                = mc68x328fb_mmap,
};
```

在“e:\sdk_code\kernel\linux-2.6.22.6\drivers\video\Atmel_lcdfb.c”中也有这三个函数。

```
static struct fb_ops atmel_lcdfb_ops = {
.owner                                = THIS_MODULE,
.fb_check_var                = atmel_lcdfb_check_var,
.fb_set_par                 = atmel_lcdfb_set_par,
.fb_setcolreg                = atmel_lcdfb_setcolreg,
.fb_pan_display                = atmel_lcdfb_pan_display,
.fb_fillrect                 = cfb_fillrect,
.fb_copyarea                 = cfb_copyarea,
.fb_imageblit                 = cfb_imageblit,
};
```

这三个函数的意思：

.fb_fillrect   = cfb_fillrect,（fill-填充，rect-矩形：填充一个矩形.）

.fb_copyarea   = cfb_copyarea, (拷贝一个区域)

.fb_imageblit   = cfb_imageblit,(图像？)

这三个函数是其他人提供的，从名字上看可能是对显存的一些操作。

fb_imageblit 也就是这个函数完成的 cfb_imageblit,在此函数中将索引检测的颜色数值，送给

info->screen_base，用图象去填充这个缓冲区。开机的 LOGO 是这样显示的。

a:

```
//3.3.1.为此LCD驱动定义一个 fb_ops 结构变量 s3c_lcdfb_ops。成员定义如下:
static struct fb_ops s3c_lcdfb_ops = {
    .owner      = THIS_MODULE,
    //.fb_setcolreg   = s3c_lcdfb_setcolreg, 以后会用到。
    .fb_fillrect    = cfb_fillrect,
    .fb_copyarea    = cfb_copyarea,
    .fb_imageblit   = cfb_imageblit, //上面三个一般都有，是别人提供的.
};
```

b:

```
//3.3.设置操作函数:fb_info 结构定义中的"struct fb_ops *fbops;".
s3c_lcd->fbops = &s3c_lcdfb_ops;
```

⑤，其他设置：

看结构体 fb_info 的定义中还有什么,参考别人的代码--Atmel_lcdfb.c 中看不懂的就试着不去设置：可以看到其中设置了“flags”，跟踪后面的宏。

```
info->flags = ATMEL_LCDFB_FBINFO_DEFAULT;


#if defined(CONFIG_ARCH_AT91)
#define ATMEL_LCDFB_FBINFO_DEFAULT  FBINFO_DEFAULT


#define FBINFO_DEFAULT  0
```

上面的这个"flags"最终也是 0。所以这个不懂时就不配置即可。

下面几个要设置：在 fb_info 结构体中定义了。

```
char __iomem *screen_base;   /* Virtual address */
unsigned long screen_size;   /* Amount of ioremapped VRAM or 0 */
void *pseudo_palette;        /* Fake palette of 16 colors */
```

这个“void *pseudo_palette; /* Fake palette of 16 colors */ ”要设置，这个也是后来经过实验知道要设置它。

“screen_base”：显存的虚拟地址。

“screen_size”：显存的物理地址。

在"fbmem.c"中看 fb_write()函数：

```
int fbidx = iminor(inode);
struct fb_info *info = registered_fb[fbidx];
```

以“次设备号”为下标，从 registered_fd[]数组中得到构造的 fb_info 结构体。若没提供写函数时：

```
if (info->fbops->fb_write)
return info->fbops->fb_write(info, buf, count, ppos);

total_size = info->screen_size;（screen_size 显存大小）

dst = (u32 __iomem *) (info->screen_base + p);（screen_base 显存虚拟地址）
```

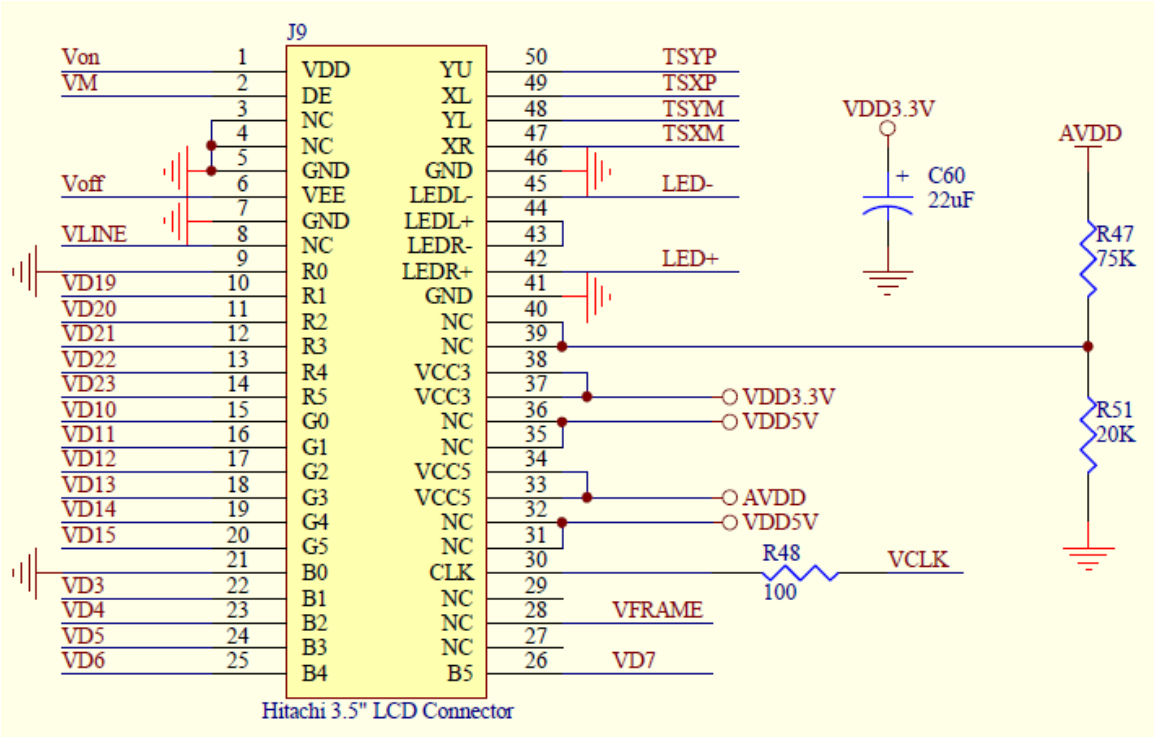
在这里面有用到“screen_base”显存虚拟地址，所以这要设置它。等分配了显存之后来设置虚拟显存地址。

```
//3.4，其他设置(看结构体fb_info的定义中还有什么,参考别人的代码):
//s3c_lcd->pseudo_palette = ?;  //假的调色码
//s3c_lcd->screen_base = ?;     //显存的虚拟地址,在fbmem.c的fb_write()有用到,在分配了显存后再
s3c_lcd->screen_size = 240*320*16/8;      //显存的大小
```

3.4，硬件相关的操作。

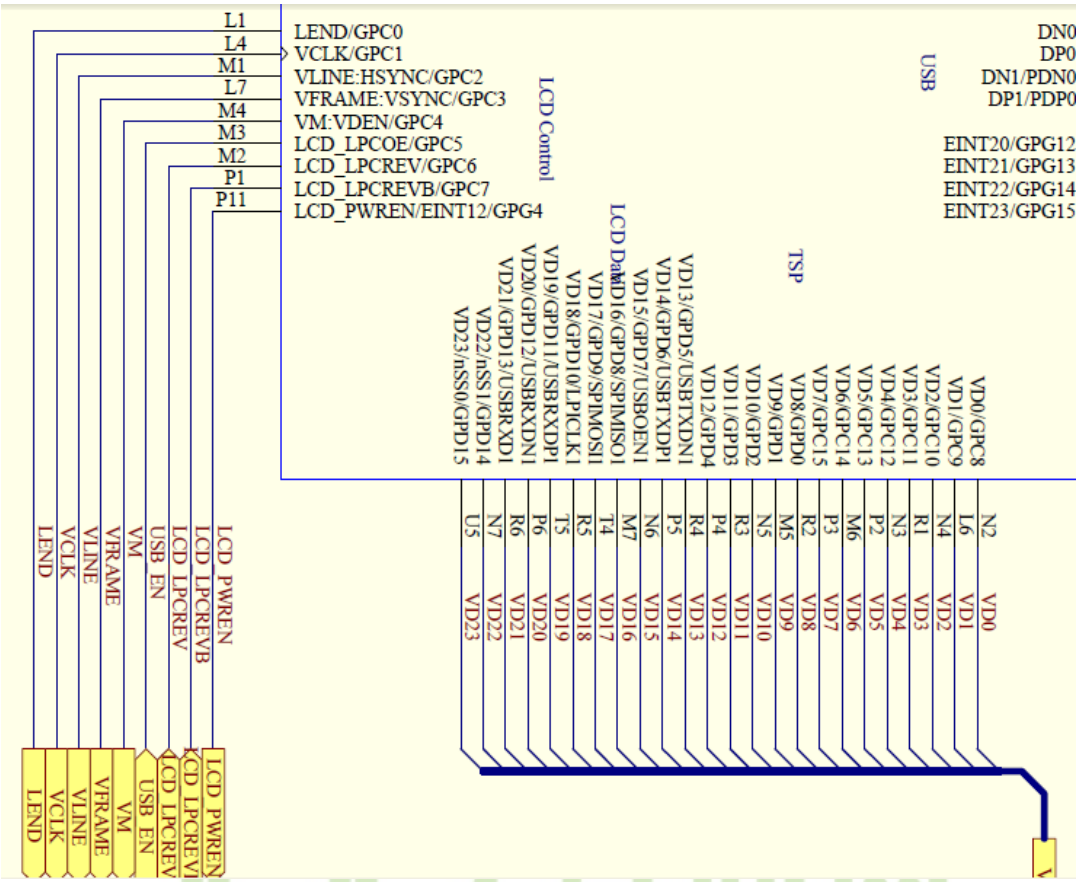
因为一注册就要用到硬件相关的设置信息,所以这一步放在前面

①，配置 2410 相关的，配置 GPIO 用于 LCD.



搜索原理图上的这些引脚与 2440 相连的 GPIO 管脚。如：

VD19 -- VD19/GPD11 ： 则要把 GPDCON 寄存器相应的位设置成用于 LCD。



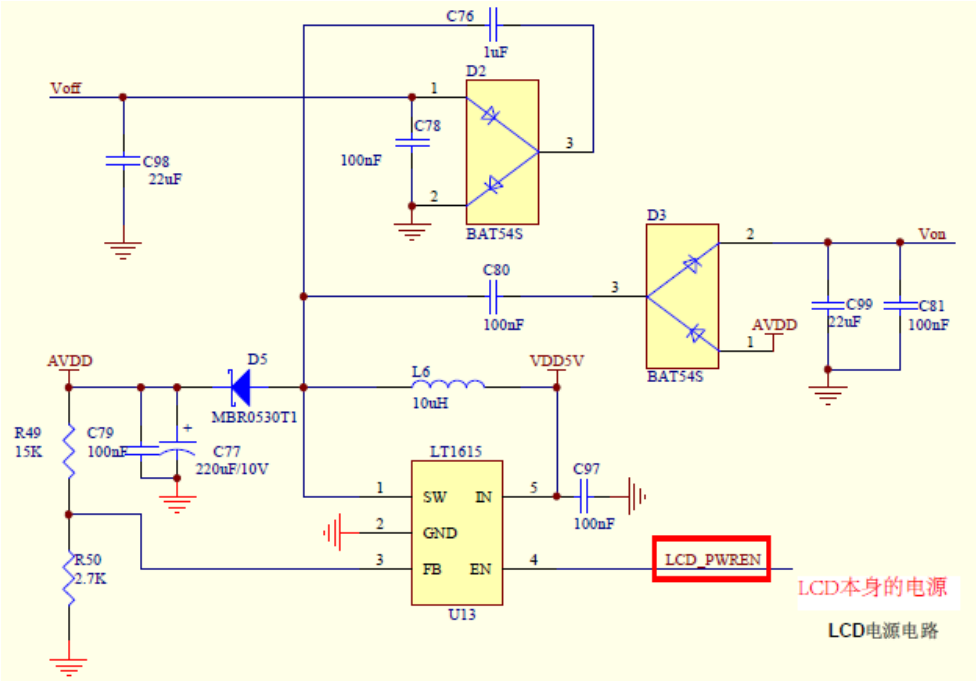
从上面的原理图可以知道用于 LCD 的寄存器有：GPC? GPD? GPG?
查看之前的裸板程序的 GPOP 设置：直接拿过来 ioremap()即可。

```
/*
 * 初始化用于LCD的引脚
 */
void Lcd_Port_Init(void)
{
    GPCUP    = 0xffffffff;    // 禁止内部上拉
    GPCCON   = 0xaaaaaaaa;    // GPIO管脚用于VD[7:0],LCDVF[2:0],VM,VFRAME,VLINE,VCLK,LEND
    GPDUP    = 0xffffffff;    // 禁止内部上拉
    GPDCON   = 0xaaaaaaaa;    // GPIO管脚用于VD[23:8]
    GPBCON &= ~(GPB0_MSK);    // Power enable pin
    GPBCON |= GPB0_out;
    GPBDAT &= ~(1<<0);            // Power off
    printf("Initializing GPIO ports.....\n");
}
```

F:\embedded\第二期：深入驱动\源码_文档_图片_原理图_芯片手册\硬件部件实验代码\lcd\lcddrv.c

上面显示用到了“GPCUP”,"GPCCON","GPDUP","GPDCON","GPBCON","GPBDAT",查看下“GPB？”是什么引脚：

这是一张给 LCD 提供背光电路的原理图。下面一张是 LCD 本身的电源。



背光电路是背光电源亮起来后，LCD 里面的东西才看的到。

查看“背光电路”中的“KEYBOARD”引脚是哪个引脚：
KEYBOARD --- J6 TOUT0/GPB0 (GPB0 要配置成输出引脚：GPBCON |= GPB0_out;)
其他的引脚如下：

```
GPCUP   = 0xffffffff;   // 禁止内部上拉
GPCCON  = 0xaaaaaaaa;   // GPIO 管脚用于
VD[7:0],LCDVF[2:0],VM,VFRAME,VLINE,VCLK,LEND
GPDUP   = 0xffffffff;   // 禁止内部上拉
GPDCON  = 0xaaaaaaaa;   // GPIO 管脚用于 VD[23:8]
GPBCON &= ~(GPB0_MSK);  // Power enable pin
GPBCON |= GPB0_out;
GPBDAT &= ~(1<<0);      // Power off
```

等等...就是要看清要配置哪些引脚。

a,**定义要映射的引脚：**

```
//4.1.1,定义要映射的 GPB? GPC? GPG? 寄存器引脚。
static volatile unsigned long *gpbcon;
static volatile unsigned long *gpbdat;

static volatile unsigned long *gpccon;
static volatile unsigned long *gpcdat;
```

```
static volatile unsigned long *gpgcon;
```

b,**开始映射物理地址：**ioremap()

```
void * __ioremap(unsigned long phys_addr, unsigned long size,
unsigned long flags)
```

入口： phys_addr：要映射的起始的 IO 地址；

size：要映射的空间的大小；

flags：要映射的 IO 空间的和权限有关的标志；

功能： 将一个 IO 地址空间映射到内核的虚拟地址空间上去，便于访问；

从芯片手册上查 GPBCON、GPBDAT 等的物理地址。

GPBCON	0x56000010				Port B control
GPBDAT	0x56000014				Port B data
GPBUP	0x56000018				Pull-up control B

```
gpbcon = ioremap(0x56000010, 8); //映射 GPBCON 寄存器物理地址 8 字节。
gpbdat = gpbcon + 1; //指针加 1 相当于加 4 字节.
```

再看 GPCCON 等物理地址：

GPCCON	0x56000020				Port C control
GPCDAT	0x56000024				Port C data
GPCUP	0x56000028				Pull-up control C

```
gpccon = ioremap(0x56000020, 4); //不会是单独映射 4 字节，而是一页页的
```

映射,这里写 4 或 1024 都没关
系，肯定都是一页页映射.

GPDCON	0x56000030				Port D control
GPDDA1T	0x56000034				Port D data
GPDUP	0x56000038				Pull-up control D

```
gpdcon = ioremap(0x56000030, 4); //gpccon 全都用着 LCD 引脚。
```

GPGCON	0x56000060				Port G control
GPGDAT	0x56000064				Port G data
GPGUP	0x56000068				Pull-up control G

```
gpgcon = ioremap(0x56000060, 4); //gpdcon 也是全都用作 LCD 引脚.
```

c,**管脚配置：**

映射好后，就可以操作了，把它们配置成 LCD 引脚。
查看之前的裸机代码：

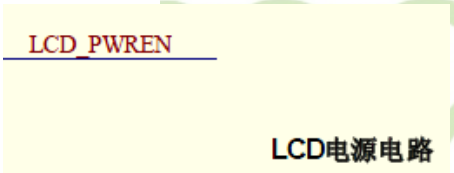
```
GPCUP    = 0xffffffff;    // 禁止内部上拉
```

```
GPCCON  = 0xaaaaaaaa;   // GPIO 管脚用于
VD[7:0],LCDVF[2:0],VM,VFRAME,VLINE,VCLK,LEND
GPDUP   = 0xffffffff;   // 禁止内部上拉
GPDCON  = 0xaaaaaaaa;   // GPIO 管脚用于 VD[23:8]

GPBCON &= ~(GPB0_MSK);  // Power enable pin
//#define GPB0_out     (1<<(0*2))
//#define GPB0_MSK     (3<<(0*2)) 是先把 bit0~1 清为零。
GPBDAT &= ~(1<<0);      // Power off 然后打它设置为“1”输出引脚。
```

对于“GPCUP = 0xffffffff; // 禁止内部上拉”可以不设置。

```
*gpbcon &= ~(3); /* GPB0 设置为输出引脚 */
*gpbcon |= 1;    /* 并且先让它输出低电平.先不使用 */
*gpbdat &= ~1;   /* 输出低电平.与上 '1 的取反' 即与上 0，结果为 0，输
出低电平 。意思是先不让背光开启.*/
```

接着还有一个“PWERN”:这是 LCD 本身的电源。LCD 工作起来后，想看到 LCD 里的显示的内容，需要在后面点亮一个“背光灯”。



"LCD_PWREN" -- "P11" -- "LCD_PWREN/EINT12/GPG4".
这个引脚的连接是“GPG4”，所以要设置“GPGCON”寄存器。

PORT G CONTROL REGISTERS (GPGCON, GPGDAT)

If GPG0–GPG7 will be used for wake-up signals at Sleep mode, the ports will be set in interrupt mode.

Register	Address	R/W	Description	Reset Value
GPGCON	0x56000060	R/W	Configures the pins of port G	0x0
GPGDAT	0x56000064	R/W	The data register for port G	Undef.
GPGUP	0x56000068	R/W	Pull-up disable register for port G	0xfc00

GPG4	[9:8]	00 = Input 10 = EINT[12]	01 = Output 11 = LCD_PWRDN

需要把它的“GPG4”来设置下。即“bit8~9”设置为“11 = LCD_PWRDN”。

```
*gpgcon |= (3<<8); /* GPG4 用作 LCD 电源使能引脚'LCD_PWREN'.芯片手册上
GPG4 是[9:8]两位，功能描述中 11 = LCD_PWRDN . 故(3<<8) */
```

②，根据 LCD 手册配置控制器，让它可以发出正确的信号(如 VCLK 频率等).

a,将 LCD 控制器物理地址构造一个结构：

LCD 控制 1 寄存器

寄存器	地址	R/W	描述	复位值
LCDCON1	0X4D000000	R/W	LCD 控制 1 寄存器	0x00000000

LCD 控制器					
LCDCON1	0X4D000000	←	W	R/W	LCD 控制 1
LCDCON2	0X4D000004				LCD 控制 2
LCDCON3	0X4D000008				LCD 控制 3
LCDCON4	0X4D00000C				LCD 控制 4
LCDCON5	0X4D000010				LCD 控制 5
LCDSADDR1	0X4D000014				STN/TFT：帧缓冲区开始地址 1
LCDSADDR2	0X4D000018				STN/TFT：帧缓冲区开始地址 2
LCDSADDR3	0X4D00001C				STN/TFT：虚拟屏幕地址设置
REDLUT	0X4D000020				STN：红色查找表
GREENLUT	0X4D000024				STN：绿色查找表
BLUELUT	0X4D000028				STN：蓝色查找表
DITHMODE	0X4D00004C				STN：抖动模式
TPAL	0X4D000050				TFT：临时调色板
LCDINTPND	0X4D000054				LCD 中断等待
LCDSRCPND	0X4D000058				LCD 中断源
LCDINTMSK	0X4D00005C				LCD 中断屏蔽
TCONSEL	0X4D000060				TCON(LPC3600/LCC3600)控制

LCDCON1 寄存器一个一个的“ioremap()”会很麻烦。这里把它们全放在一个结构中。将上面诸如“LCDCON1 - 0X4D000000”,"LCDCON2 - 0X4D000004"..."LCDSADDR1 - 0X4D000013"..."TCONSEL - 0X4D000060".一个一个排在结构中。注意的是两个地址间相差 4 字节 。若之间跳跃太大，就要加些“保留字节”。

```
unsigned long                bluelut;        // 0X4D000028 STN：蓝色查找表
unsigned long                reserved[9];    //保留 9 字节
unsigned long                dithmode;       // 0X4D00004C STN：抖动模式
```

“0X4D000028 ”与“0X4D00004C ”相差了：
0X4D000028 - 0X4D00004C = 0x24 转成十进制为：36.36 再除以 4 则为 9.所以中间相差了 9 字节。

```
/*4.2.1,LCD 控制寄存器 LCDCON1 需要 ioremap 的地址:2410/2440 一致 */
struct lcd_regs {
unsigned long               lcdcon1;        // 0X4D000000 LCD 控制器
1。
unsigned long               lcdcon2;        // 0X4D000004 LCD 控制 2
unsigned long               lcdcon3;        // 0X4D000008 LCD 控制 3
unsigned long               lcdcon4;        // 0X4D00000C LCD 控制 4
unsigned long               lcdcon5;        // 0X4D000010 LCD 控制 5
    unsigned long               lcdsaddr1;      // 0X4D000014
STN/TFT：帧缓冲区开始地址 1
    unsigned long               lcdsaddr2;      // 0X4D000018
STN/TFT：帧缓冲区开始地址 2
    unsigned long               lcdsaddr3;      // 0X4D00001C
STN/TFT：虚拟屏幕地址设置
    unsigned long               redlut;         // 0X4D000020 STN：
红色查找表
    unsigned long               greenlut;       // 0X4D000024 STN：
绿色查找表
    unsigned long               bluelut;        // 0X4D000028 STN：
蓝色查找表
    unsigned long               reserved[9];    // 0X4D000028 -
0X4D00004C = 0x24 十进制:36.36/4=9.故中间相差 9 字节。
    unsigned long               dithmode;       // 0X4D00004C STN：
抖动模式
    unsigned long               tpal;           // 0X4D000050 TFT：
临时调色板
    unsigned long               lcdintpnd;      // 0X4D000054 LCD 中
断等待
    unsigned long               lcdsrcpnd;      // 0X4D000058 LCD 中
断源
    unsigned long               lcdintmsk;      // 0X4D00005C LCD 中
断屏蔽
    unsigned long               lpcsel;         // 0X4D000060
TCON(LPC3600/LCC3600)控制
};
```

b,**接着需要定义一个结构体指针，并将其** ioremap();

//4.2.2,定义一个 lcd_regs 结构体变量.

```
static volatile struct lcd_regs *lcd_regs;
```

LCD 控制寄存器所在的物理地址基地址是：

LCDCON1	0X4D000000				LCD 控制 1

Ioremap()：

```
//4.2.3,定义了一个 lcd_regs 结构指针变量后，就开始 ioremap();
    lcd_regs = ioremap(0x4D000000, sizeof(struct lcd_regs));
```

到这里好像明白了 ioremap 的用法，就是一个寄存器基地址为起始，这个寄存器所占的大小为要 ioremap 的大小。这样 ioremap 后，就可以直接操作里面的寄存器了。

c,接着设置这个寄存器：

先设置“LCDCON1”:一位一位的设置。

LCD 控制 1 寄存器

寄存器	地址	R/W	描述	复位值
LCDCON1	0X4D000000	R/W	LCD 控制 1 寄存器	0x00000000

LCDCON1	位	描述	初始状态
LINECNT(只读)	[27:18]	提供行计数器的状态。从 LINEVAL 递减计数到 0。	0000000000
CLKVAL	[17:8]	决定 VCLK 的频率和 CLKVAL[9:0]。 **STN**：VCLK = HCLK / (CLKVAL × 2)　（CLKVAL≥2） **TFT**：VCLK = HCLK / [(CLKVAL + 1) × 2]　（CLKVAL≥0）	0000000000
MMODE	[7]	决定 VM 的触发频率 0 = 每帧　1 = 由 MVAL 定义此频率	0
PNRMODE	[6:5]	选择显示模式 00 = 4 位双扫描显示模式（STN）　01 = 4 位单扫描显示模式（STN） 10 = 8 位单扫描显示模式（STN）　11 = TFT LCD 面板	00
BPPMODE	[4:1]	选择 BPP（位每像素）模式 0000 = STN 的 1 bpp，单色模式　0001 = STN 的 2 bpp，4 阶灰度模式 0010 = STN 的 4 bpp，16 阶灰度模式　0011 = STN 的 8 bpp，彩色模式(256 色) 0100 = STN 的封装 12 bpp，彩色模式（4096 色） 0101 = STN 的未封装 12 bpp，彩色模式（4096 色） 0110 = STN 的封装 16 bpp，彩色模式（4096 色） 1000 = TFT 的 1 bpp　1001 = TFT 的 2 bpp 1010 = TFT 的 4 bpp　1011 = TFT 的 8 bpp 1100 = TFT 的 16 bpp　1101 = TFT 的 24 bpp	0000
ENVID	[0]	LCD 视频输出和逻辑使能/禁止。 0 = 禁止视频输出和 LCD 控制信号　1 = 允许视频输出和 LCD 控制信号	0

VCLK 设置：Bit[17:8]

CLKVAL:这里设置 VCLK 输出的时钟（喷枪从左往右移动的速度）。要设置 LCD 控制器让它发出合适的时钟，这个 LCD 屏才会反应的过来。若 LCD 控制器发的时钟太快，LCD 屏可能没法响应。若是太慢了点，则就会看到“喷枪”（光标）在屏幕上一点一点“描”（移动）。所以这个 VCLK（时钟）要设置。

有公式：TFT：VCLK = HCLK / [(CLKVAL + 1) × 2

这里“HCLK”可以从内核的输出信息知道：100 MHz

```
# dmesg
Linux version 2.6.22.6 (book@book-desktop) (gcc version 3.4.5) #1 Thu Jan 27 22:22:51 CST 2011
CPU: ARM920T [41129200] revision 0 (ARMv4T), cr=c0007177
Machine: SMDK2440
Memory policy: ECC disabled, Data cache writeback
On node 0 totalpages: 16384
  DMA zone: 128 pages used for memmap
  DMA zone: 0 pages reserved
  DMA zone: 16256 pages, LIFO batch:3
  Normal zone: 0 pages used for memmap
CPU S3C2440A (id 0x32440001)
S3C244X: core 400.000 MHz, memory 100.000 MHz, peripheral 50.000 MHz
S3C24XX Clocks, (c) 2004 Simtec Electronics
CLOCK: Slow mode (1.500 MHz), fast, MPLL on, UPLL on
CPU0: D VIVT write-back cache
CPU0: I cache: 16384 bytes, associativity 64, 32 byte lines, 8 sets
```

可以通过某些函数得到这个 HCLK(memory 100 MHz)。

现在：VCLK = 100MHz/［(CLKVAL + 1) × 2

这里 VCLK 取多少决定于 LCD 屏的手册：这个 VCLK 要看不同的 LCD 屏手册来调整。

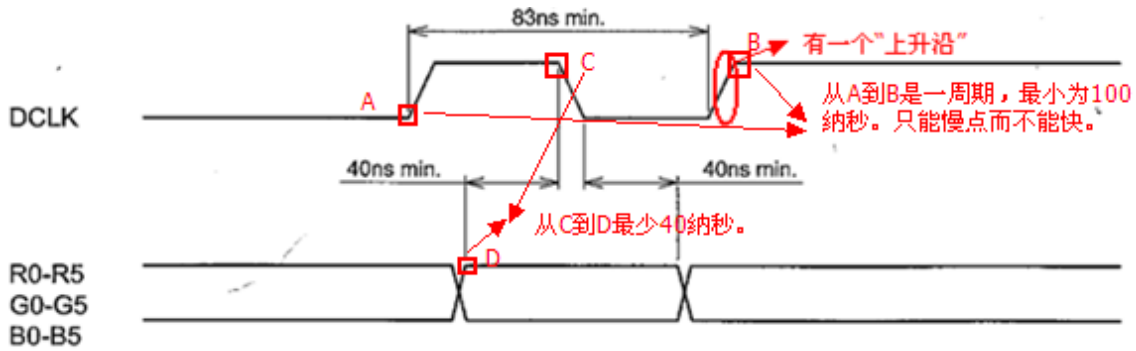
	Item	Symbol	Min.	Typ.	Max.	Unit.	Remark
Source Driver	Clock cycle time	Trate	100	- VCLK	-	ns	-
	Clock low level width	Tcwl	35	-	-	ns	-
	Clock high level width	Tcwh	35	-	-	ns	-
	Data set up time	Tds	25	-	-	ns	-
	Data hold time	Tdh	25	-	-	ns	-
	Start pulse set up time	Tss	25	-	-	ns	-
	Start pulse hold time	Tsh	25	-	-	ns	-
	CL1 high level width	Tcl1wh	10	-	-	us	-
	CL1 start pulse	Tscl1	100	-	-	ns	-
	STH start pulse	Tssth	100	-	-	ns	-
	M set up time	Tms	50	-	-	ns	-
	M hold time	Tmh	50	-	-	ns	-

“clock cycle time”（时钟周期）为 100 纳秒。

“clock low level width”(低周期) 最少 35 纳秒。

“clock high level width”(高周期) 最少 35 纳秒。

高、低周期最少加起来为 70 纳秒。（与 100 纳秒不吻合）



从上两个图看，这个 LCD 的 VCLK 应该是 100 纳秒。详细得查网络。从 A 到 B 处是一个周期，最小为 100 纳秒。因为只能慢点点而不能快。高电平时间是“40 纳秒”。上图只能看出这么多东西（不好看明白）。不过 VCLK 就是 100 纳秒。
频率等于周期的倒数。100ns 返过来频率是：10MHz.
用计算器计算的方法：
输入 10 进制 100 --> 点(科学计算器）Exp 结果为“100.e+0″ --> 点数字 9，结果为“100.e+9” --> 再点“+/-”号结果为“100.e-9”--> 最后按“1/x”键，结果为“10 000 000”。即 10MHz.

10MHz= 100MHz/ [(CLKVAL + 1) × 2 ----> 最后算得：CLKVAL = 4.

MMODE 设置： bit[7]，“决定 VM 的触发频率”这个和我们这里设置没什么关系，不用管。

PNRMODE 设置：bit[6:5]，“选择显示模式”
我们是 TFT LCD.则设置成“11 = TFT LCD 面板”。

BPPMODE 设置： bit[4:1] ，是像素的格式。选择 BPP（位每像素）模式。每个像素用多少位来表示。
我们这里之前设置过一个像素为 RGB--565,丢弃了 2 位，是 16 位（2410 不支持 18 位的），所以这里我们选
择：1100 = TFT 的 16 bpp

ENVID 设置： bit[0]，LCD 视频输出和逻辑使能/禁止。设置完后再使能。所以这里先禁止。

lcd_regs->lcdcon1 = (4<<8)| (3<<5) | (0x0c<<1);
一位一位的设置：
VCLK 为 Bit[17:8]位，最后算出来的结果 CLKVAL 等于 4，则 bit[17:8]为 4，即(4<<8)
PNRMODE 为 bit[6:5]两位，我们选择的显示模式是“TFT LCD 面板”，芯片手册上规定 TFT LCD 面板时这两位设置为“0b11”，则(3<<5),3 的二进制为“0b11”，这个意思就是说把 0b11 械移 5 位，也就是 bit[6:5]两个位置处从“初始状态：00”被设置成了“11”。
BPPMODE 为 bit[4:1],“初始状态：0000”，我们的 2440 每像素是占 16 位(″RGB--565″),根据芯片手册规定每像素为 16 位时，此 bit[4:1]为“0b1100”，0b1100 十六进制是 C，则为（0x0c<<1）,就是把 0b1100 左移了 1 位（因为寄存器的位是从 0 开始，BPPMODE 是占了从位 1 到位 4 这 4 个位，所以是左移 1 位。）。

再设置“LCDCON2”:LCD 控制器 2.垂直方面的时间参数。

LCD 控制 2 寄存器

寄存器	地址	R/W	描述	复位值
LCDCON2	0X4D000004	R/W	LCD 控制 2 寄存器	0x00000000

表 6.2 LCDCON2 描述

LCDCON2	位	描述	初始值
VBPD	[31:24]	TFT:帧同步后，帧数据开始前，无效行信号的数量； STN:必须为 0	0x00
LINEVAL	[23:14]	LCD 面板的垂直尺寸	0000000000
VFPD	[13:6]	TFT: 帧数据结束后，帧同步前，无效行信号的数量； STN:必须为 0	00000000
VSPW	[5:0]	TFT: 通过计算无效行的数量，决定帧同步信号脉冲高电平的宽度； STN:必须为 0	000000

VBPD 设置： bit[31:24]：

表 27-12. TFT LCD 控制器模块信号时序常数

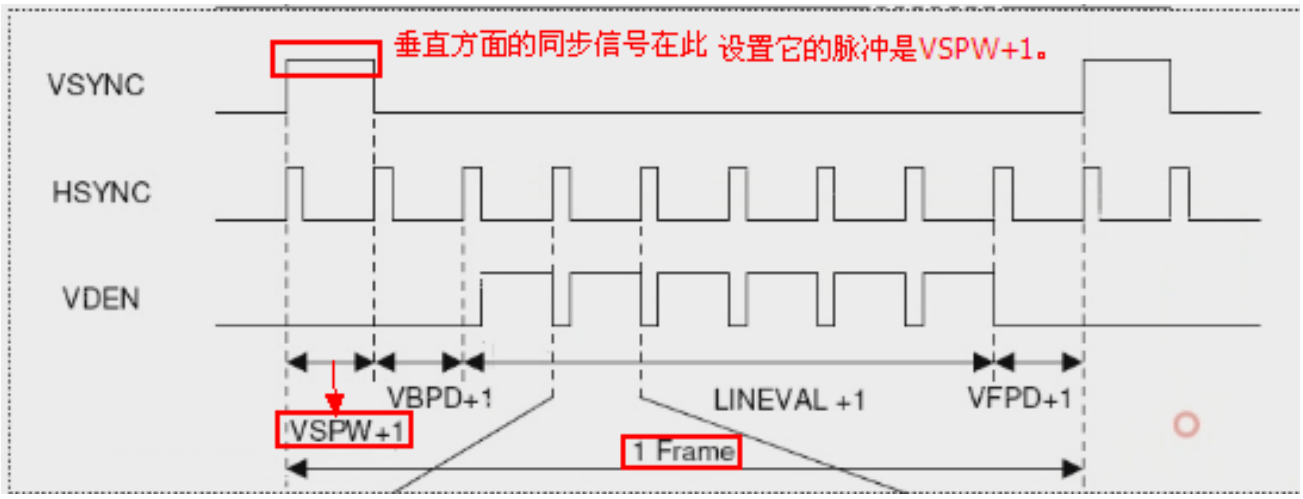
(V_{DD} = 1.2 V ± 0.05 V , T_A = –40 至 85 °C , V_{EXT} = 3.3V ± 0.3V)

参数	符号	最小值	典型值	最大值	单位
垂直脉冲宽度	Tvspw	VSPW + 1	–	–	Phclk(1)
垂直后沿延迟	Tvbpd	VBPD+1	–	–	Phclk
垂直前沿延迟	Tvfpd	VFPD+1	–	–	Phclk
VCLK 脉冲宽度	Tvclk	1	–	–	Phclk(2)
VCLK 脉冲高电平宽度	Tvclkh	0.5	–	–	Phclk
VCLK 脉冲低电平宽度	Tvclkl	0.5	–	–	Phclk
Hsync 建立至 VCLK 下降沿	Tl2csetup	0.5	–	–	Phclk
VDEN 建立至 VCLK 下降沿	Tde2csetup	0.5	–	–	Phclk
VDEN 保持至 VCLK 下降沿	Tde2chold	0.5	–	–	Phclk
VD 建立至 VCLK 下降沿	Tvd2csetup	0.5	–	–	Phclk
VD 保持至 VCLK 下降沿	Tvd2chold	0.5	–	–	Phclk
VSYNC 建立至 HSYNC 下降沿	Tf2hsetup	HSPW + 1	–	–	Phclk
VSYNC 保持至 HSYNC 下降沿	Tf2hhold	HBPD + HFPD +HOZVAL + 3	–	–	Phclk

注释：

1. HSYNC 周期
2. VCLK 周期

可以搜索 2440 上这个寄存域是什么意思，看看有没有图：



2440 上 LCD 控制器 的 帧时序

VCLK：是 LCD 控制器和 LCD 驱动器之间的像素时钟信号（用于锁存图像数据的像素时钟），由 LCD 控制器发

出数据在 VCLK 的上升沿处送出，在 VCLK 的下降沿处被 LCD 驱动器采样。

HSYNC：是 LCD 控制器与 LCD 驱动器间的行同步信号。每发出一个脉冲都表明新的一行图像资料开始发送。

VSYNC：是 LCD 控制器与 LCD 驱动器间的帧同步信号。每发出一个脉冲，都意味着新的一屏图像数据开始发送。

VDEN：数据有效标志信号。是 LCD 驱动器的 AC 信号，它可以被 LCD 驱动器用于改变行和列的电压极性，从

而控制像素点的显示或熄灭。

在帧同步以及行同步的头尾都必须留有回扫时间。这样安排起源于 CRT 显示器电子枪偏转所需要的时间，后来成为实际上的工业标准，因此 TFT 屏也包含了回扫时间。

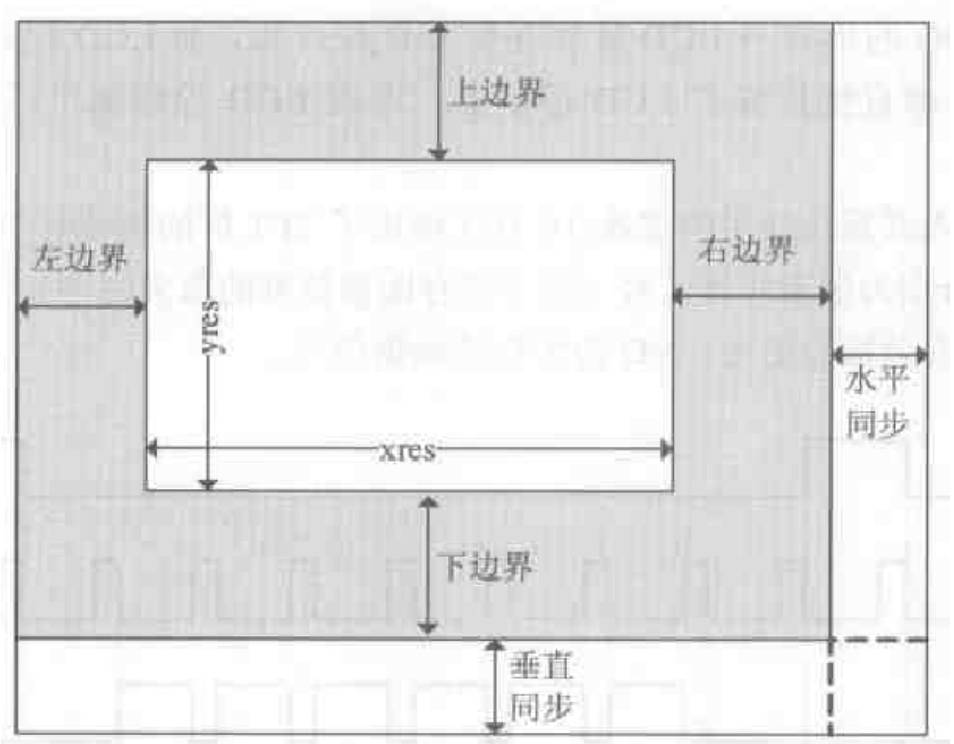
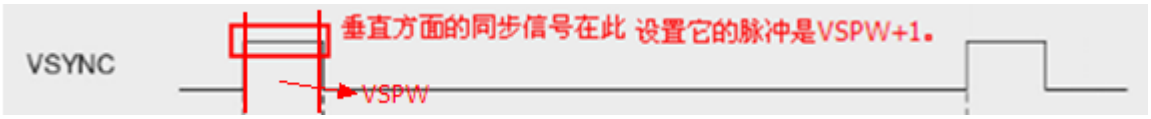


图 18.2　LCD 控制器中的时序参数设置

其中的上边界和下边界即为帧切换的回扫时间。
左边界和右边界为行切换的回扫时间。
水平同步和垂直同步分别是行和帧同步本身需要的时间。
xres 和 yres 则分别是屏幕的水平和垂直分辨率，常见的嵌入式设备的 LCD 分辨离主要为 320*240、640
*480 等。

上面这是一帧的截图。这里一帧的时序图中有“VBPD”寄存器域。LCD 可以想像成后面有一个“喷枪-电子枪”从左至右、从上至下 的移动。当移到最下一行时，收到一个“垂直方向的同步信号--VSYNC”时就会跳回到 LCD 屏的最上边的最左端。这里要想像，这个跳回去不能是瞬间跳回去的，并且这个脉冲，这个垂直方向的同步信号“VSYNC”，这个脉冲要维持多长时间才能感应的到？这个就要设置。

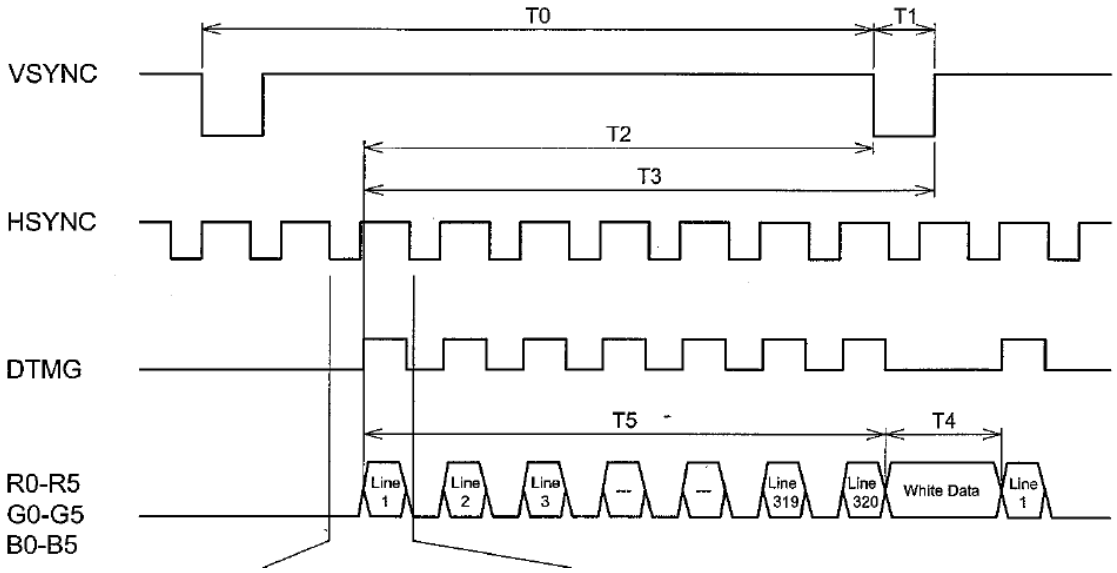


从上面的时序图看到一个“VSYNC”垂直方向的同步信号的脉冲是“VSPW+1”,VSPW 域设置为 0 时，就是一个“行同步信号 -- HSYNC”，即维持多少行的时间。

Hsync 建立至 VCLK 下降沿.
从下面跳到最上边时不可能瞬间就能跳转回去，要跳多长时间，跳回去之后要过多久才能重新发出一行的
数据？要过“VBPD”这么长时间才能重新发出数据。

VDEN 建立至 VCLK 下降沿.
VDEN 保持至 VCLK 下降沿.
从上面扫描到下面之后，并不是立刻跳转回去的，要过多长时间才能收到下一个行，垂直方向同步信号跳
回去。
有多少行可以设置“LINEVAL +1”。
从上图可知有 4 个参数可以去设置:VSPW 、VBPD 、LINEVAL 、VFPD.查看 LCD 手册看这些参数的意
义。

VSYNC 垂直方向的同步信号，下面是 LCD 屏手册上的“帧时序图”：



要从这个时序图反推上面 2410 是的“VSPW 、VBPD 、LINEVAL 、VFPD”的值。

先看 LCD 屏手册上这个时序图上的"T0"等于多少：

	MIN.	TYP.	MAX.	UNIT	SYMBOL
Vertical Total	-	327	-	Line	T0
Vertical Sync Width	1	1	-	Line	T1
Vertical Sync Start	-	322	-	Line	T2
Vertical Sync End	-	323	-	Line	T3
Vertical Blank Time	5	7	-	Line	T4
Vertical Display End	-	320	-	Line	T5

T0 = 327. T1 = 1, T2 = 322, T3 = 323, T4 = 7, T5 = 320.

垂直方向的时间参数:

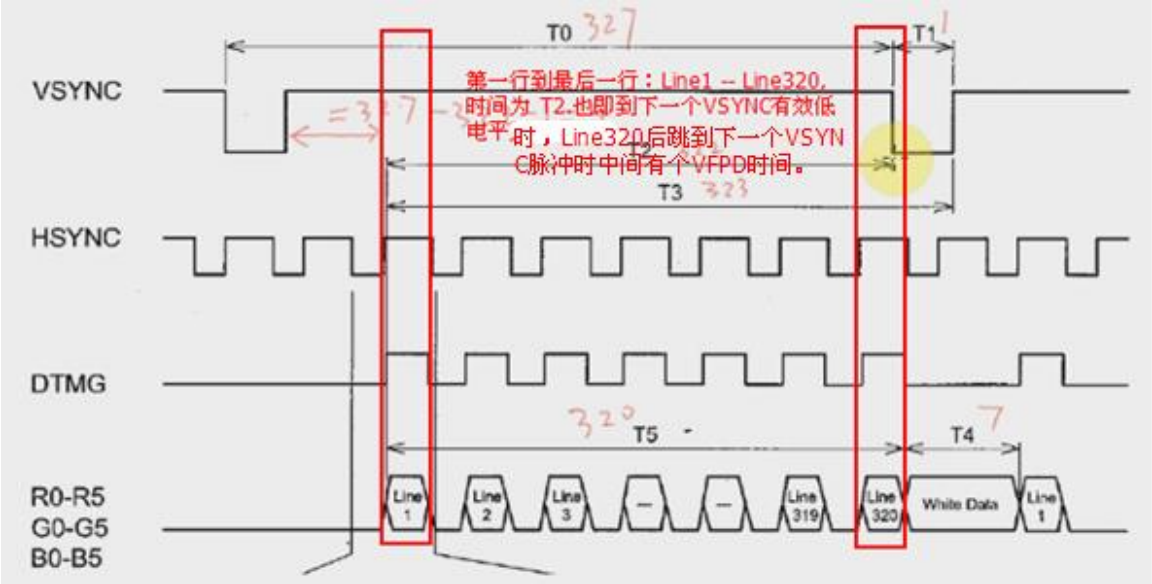
a, 在 LCD 屏手册上，VSYNC 垂直方向的同步信号，平时是高电平，有效时是“低电平”。而“2440”手册上的时序图上显示 VSYNC 是，平时是“低电平”，而有效时是“高电平”。所以要设置某些寄存器，把这个 2440 上的 VSYNC 的极性反过来使之“有效电平”与 LCD 屏手册规定的时序相同。

b, VSPW bit[5:0] : VSYNC 信号的脉冲宽度,上图 T1 为就是显示的 VSYNC 信号的脉冲宽度。 从 LCD 屏手册知脉冲 T1=1,2440 上一个 VSYNC 脉冲是 VSPW+1,所以反推，1=VSPW+1,则 VSPW 域要设置为 0.

c，LINEVAL bit[23:14]: 多少行, 320, 所以 LINEVAL=320-1=319
2440 手册上的时序是：LINEVAL+1 .
LCD 手册时序中为 320 行。

所以 LINEVAL+1=320, 故，LINEVAL=319.

d,VFPD bit[13:6] : 发送到最后一行数据之后，要过多长时间再收到下一个 VSYNC 垂直方向的的同步信号。
LCD 手册时序上显示最后一行数据是到“T2”结束.



而从上面的 LCD 屏时序 可以看到一帧 320 行的时间为 Line1 到 Line320 行的时长是 T5。

而 2440LCD 时序中一个帧的第一行到最后一行后，要开始跳到下一个帧时，中间有个缓冲的时间，这个时间就是 VFPD 。

VFPD+1 = T2 - T5 = 322-320 = 2.

所以 VFPD = 1;

e,VBPD bit[31:24]: VSYNC 之后再过多长时间才能发出第 1 行数据。VSYNC 垂直方向的同步信号脉冲

后多长时间后，才能发出第一行的数据。收到 VSYNC 同步时钟信号后要跳到最上去开始下一帧，

并不是一跳过去后瞬间有能有数据发送出来，要设置一个“缓冲”（差不多是缓冲）的时间-

VBPD。LCD 手册上的时序 VBPD 是 TO-T2 后再减去一个 VSYNC 低电平周期（脉冲宽度 T1）.

即：VBPD +1 = TO-T2-T1 = 327-322-1 = 4 。故：VBPD = 3.

所以，LCDCON2 ，垂直方向的时间参数设置为：

```
lcd_regs->lcdcon2 = (3<<24) | (319<<14) | (1<<6) | (0<<0);
```

再设置“LCDCON3”:LCD 控制器

水平方面的时间参数。

LCD Control 3 Register

Register	Address	R/W	Description	Reset Value
LCDCON3	0X4D000008	R/W	LCD control 3 register	0x00000000

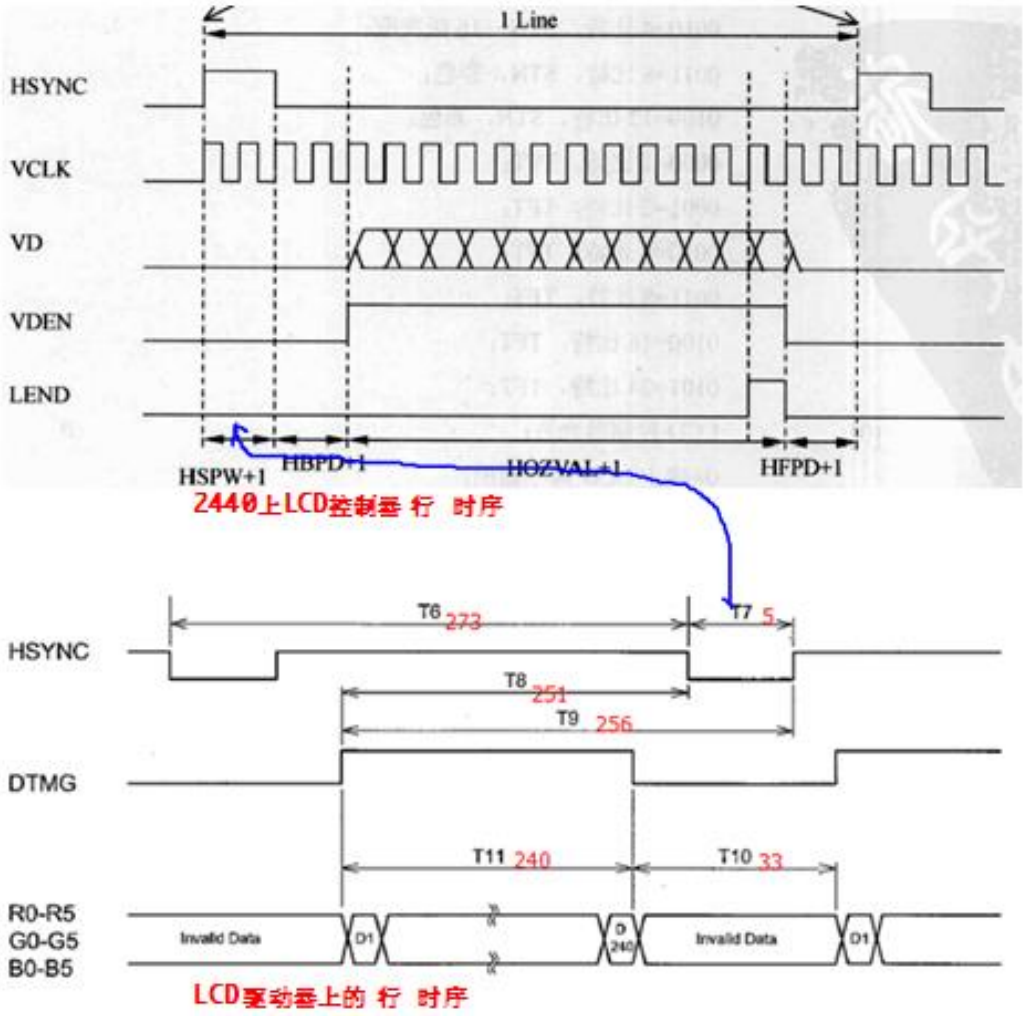
表 6.3　LCDCON3 描述

LCDCON3	位	描述	初始值
HBPD（TFT） WDLY（STN）	[25:19]	TFT: 行同步下降沿后，行数据开始前，无效 VCLK 的数量 STN: WDLY[1：0]决定 VLINE 和 VCLK 之间的延迟。 00=16HCLK；01=32HCLK；10=64HCLK；11=128HCLK	0000000
HOZVAL	[18:8]	TFT/STN：LCD 面板的水平尺寸	00000000000
HFPD（TFT） LINEBLANK （STN）	[7:0]	TFT: 行数据结束后，行同步上升沿前，无效 VCLK 的数量 STN:表示每行内插入的空时间，这些位可以精确地调整 VLINE 的频率 BLANKTIME= LINEBLANK×HCLK×8	0x00

表 6.4　LCDCON4 描述

LCDCON4	位	描述	初始值
MVAL	[15:8]	STN: MMOD=1 时，决定 VM 信号的翻转频率	0x00
HSPW（TFT） WLH（STN）	[7:0]	TFT: 决定行同步信号脉冲高电平的宽度。单位是 VCLK STN:WLH[1:0]决定 VLINE 脉冲高电平的宽度，单位是 HCLK 00=16HCLK，01=32HCLK，10=64HCLK，11=128HCLK	0x00

我们的实验中是 TFT 屏，则另一个“STN”类型的不用配置。

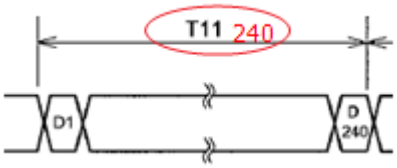


	MIN.	TYP.	MAX.	UNIT	SYMBOL
Horizontal Total	258	273	509	Pixel Clock	T6
Horizontal Sync Width	4	5	10	Pixel Clock	T7
Horizontal Sync Start	246	251	307	Pixel Clock	T8
Horizontal Sync End	250	256	317	Pixel Clock	T9
Horizontal Blank Time	18	33	269	Pixel Clock	T10
Horizontal Display End	-	240	-	Pixel Clock	T11

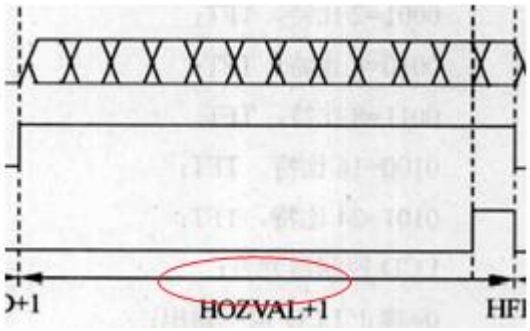
LCD 驱动器上 行 时序上的参数定义如上。
T6 的典型值是 273，这里直接用曲型值好了。其他 T7 ~ 11 都使用典型值。

HSPW(TFT) [7:0]：决定行同步信号脉冲高电平的宽度。单位是 VCLK。这个寄存器域是 LCDCON4 中的。是指 HSYNC 同步信号的脉冲宽度。
T7 是脉冲宽度。2440LCD 控制器行时序中，HSYNC 一个脉冲是“HSPW+1”,在 LCD 驱动器的行时序中 HSYNC 的一个脉冲宽度是“T7”，所以:T7 = HSPW+1 = 5,则 HSPW = 4.

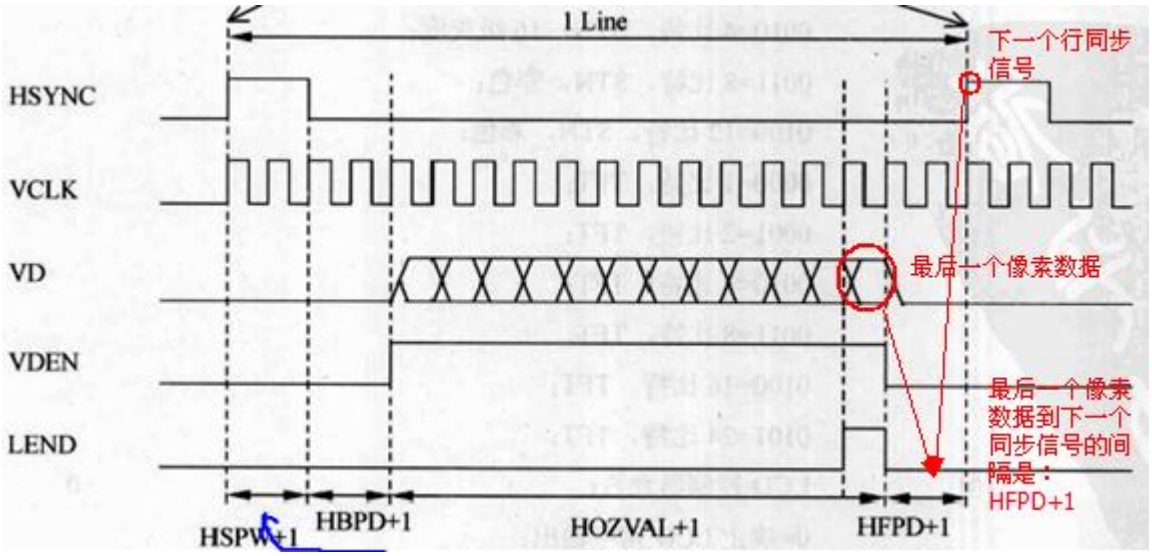
HOZVAL [18:8]:有多少列数据,LCD 面板的水平尺寸。LCD 驱动器时序图上显示有 240 列。即一行有多少列像素。

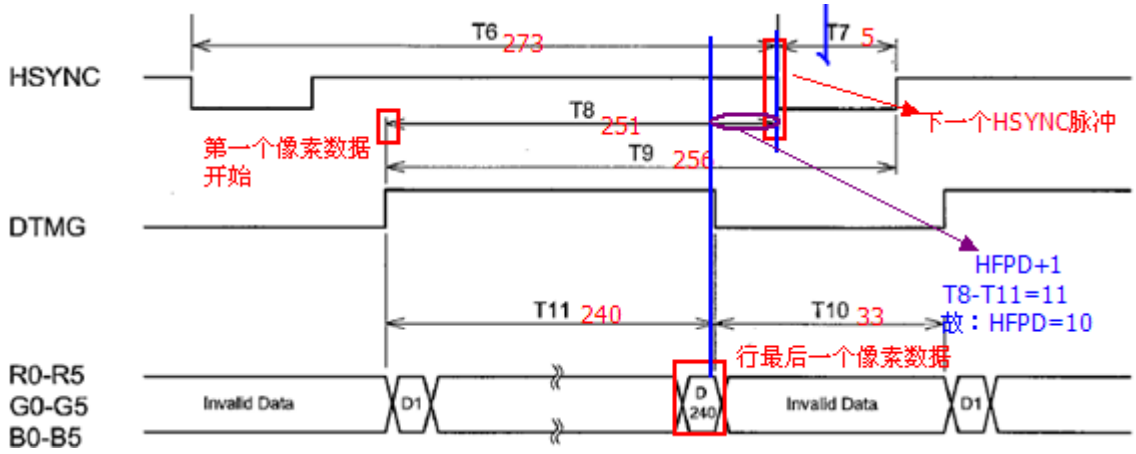


下面是 2440 LCD 控制器 行时序：HOZVAL+1 就是行数据。



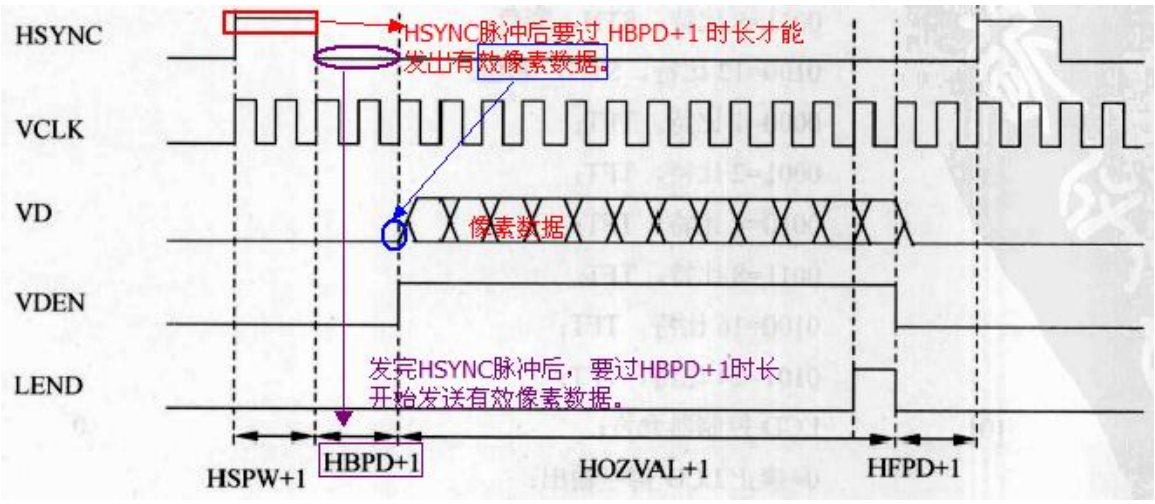
HFPD (TFT) [7:0]: 行数据结束后，行同步上升沿前，无效的 VCLK 的数量。即发出一行的最后一个像素数据后，再过多久时间发出下一个行同步信号“HSYNC”。



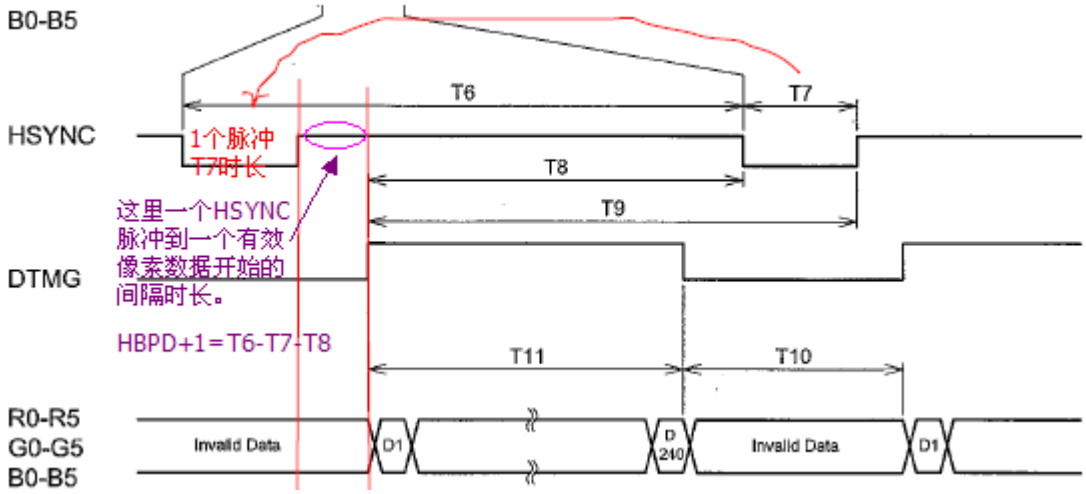


从 2440 LCD 控制器行时序上看,发出最后一个像素的数据之后。要多长时间能接收到水平方向的同步信号。最后一个像素数据是：T8 - T11=251-240=11

HBPD (TFT) [25:19]：行同步下降沿后，行数据开始前，无效 VCLK 的数量。即 HSYNC 后过多入才能发出第一个像素的数据。



喷枪从行的左边扫描到行的右边。然后收到一个水平方向的同步信号“HSYNC”后，就跳回到最左边。跳到最左边时并不能立即发出另一个像素值，这需要点时间。这个时间长，看 LCD 驱动器行时序可知道：



最后：HBPD+1 = T6-T7-T8=273-5-251=17,则 HBPD = 16.

以上便算出了 LCDCON3,LCDCON4 中要设置的 域：
最后代码为：

```
//4.2.4.3,设置 LCD 控制器 2:LCDCON3
        /* 水平方向的时间参数
      * bit[25:19]: HBPD, VSYNC 之后再过多长时间才能发出第 1 行数据
      *              LCD 驱动器行时序 T6-T7-T8=17
      *              HBPD=16
      * bit[18:8]: 多少列(一行多少个像素), 此 LCD 驱动器手册上说的是
240 列, 所以 HOZVAL=240-1=239
      * bit[7:0] : HFPD, 发出最后一行里最后一个象素数据之后，再过多长
时间才发出下一个 HSYNC
      *              LCD 手册 T8-T11=251-240=11, 所以 HFPD=11-1=10
      */
```

```
    lcd_regs->lcdcon3 = (16<<19) | (239<<8) | (10<<0);

      //4.2.4.4,设置 LCD 控制器 2:LCDCON4
      /* 水平方向的同步信号
     * bit[7:0]                    : HSPW, HSYNC 信号的脉冲宽度, LCD 手册
T7=5, 所以 HSPW=5-1=4
     */
    lcd_regs->lcdcon4 = 4<<0;
```

其他 LCD 控制器：

LCDCON5:只是控制的是引脚。就是那些极性。

LCD Control 5 Register

Register	Address	R/W	Description	Reset Value
LCDCON5	0X4D000010	R/W	LCD control 5 register	0x00000000

VSTATUS	[16:15]	**TFT**: Vertical Status (read only). 帧状态 00 = VSYNC 01 = BACK Porch 10 = ACTIVE 11 = FRONT Porch	00
HSTATUS	[14:13]	**TFT**: Horizontal Status (read only). 行状态 00 = HSYNC 01 = BACK Porch 10 = ACTIVE 11 = FRONT Porch	00

以上两个都是“只读”，就不能配置，不用管。

BPP24BL	[12]	**TFT**: This bit determines the order of 24 bpp video memory. 0 = LSB valid 1 = MSB Valid	0

我们实验的 TFT 屏是每像素 16 位，这个寄存器域说的是 24bpp 视频内存的存放顺序。所以不用管。

FRM565	[11]	**TFT**: This bit selects the format of 16 bpp output video data. 0 = 5:5:5:1 Format 1 = 5:6:5 Format	0

我们的 TFT 屏每像素是 16 位，RGB--565,所以“FRM565”要设置成“1”。

INVVCLK	[10]	**STN/TFT**: This bit controls the polarity of the VCLK active edge. 0 = The video data is fetched at VCLK falling edge 1 = The video data is fetched at VCLK rising edge	0

INVVCLK	[10]	STN/TFT:控制 VCLK 的有效电平。 0=在 VCLK 的下降沿读数据； 1=在 VCLK 的上升沿读数据	0

反转 VCLK。查看我们实验的 LCD 屏的 VCLK 是否需要返转：

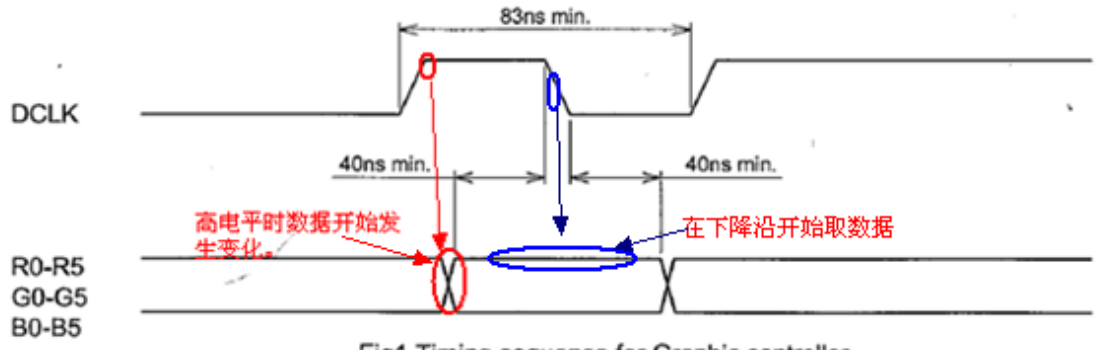


Fig1.Timing sequence for Graphic controller

在高电平的时候，数据发生变化。是在下降沿时开始取数据。

再看 2440 LCD 控制器中的规定：在 LCDCON5 的 bit[10]直接说了。为 0 时，就和 LCD 屏（LCD 驱动器）中规定的一样了。就不需要返转了，直接匹配。

INVVCLK	[10]	STN/TFT:控制 VCLK 的有效电平。 0=在 VCLK 的下降沿读数据； 1=在 VCLK 的上升沿读数据	0

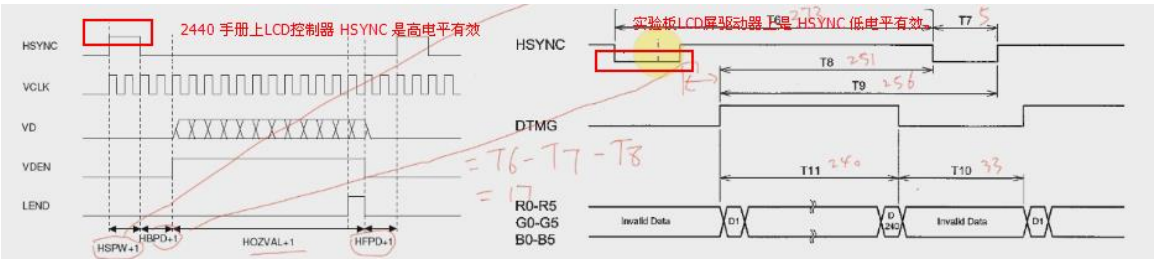
所以，这里“INVVCLK [10]”直接设置为“0”。

INVVCLK	[10]	**STN/TFT**: This bit controls the polarity of the VCLK active edge. 0 = The video data is fetched at VCLK falling edge 1 = The video data is fetched at VCLK rising edge	0

INVVLINE [9]：水平方向的同步信号正常还是要反转？

INVVLINE	[9]	STN/TFT:VLINE/HSYNC 脉冲的电平。 0=正常；1=翻转	0

在 2440 手册上，HSYNC 是高电平有效。而 实验板上的 LCD 驱动器是 HSYNC 低电平有效：



从 2440 手册和 LCD 驱动器时序图看来，这里需要反转。即按 LCD 屏的实际情况，是低电平有效。故：

INVVLINE bit[9] : 1 = HSYNC 信号要反转,即低电平有效

INVVFRAME	[8]	**STN/TFT**: This bit indicates the VFRAME/VSYNC pulse polarity. 0 = Normal 1 = Inverted	0

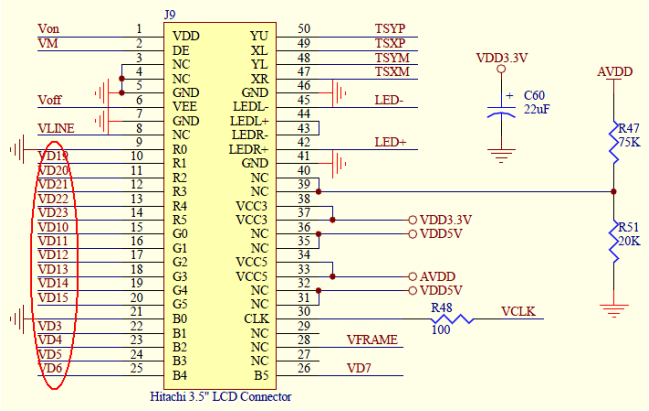
INVVFRAME	[8]	STN/TFT:VFRAME/VSYNC 脉冲的电平。 0=正常；1=翻转	0

INVVFRAME [8]：是指 帧同步信号是否要反转。我们实际上的 LCD 驱动器与 2440 LCD 控制器的帧同步信号也要反转。故：

INVVFRAME bit[8] : 1 = VSYNC 信号要反转,即低电平有效 。

INVVD	[7]	**STN/TFT**: This bit indicates the VD (video data) pulse polarity. 0 = Normal 1 = VD is inverted.	0

INVVD [7] STN/TFT:数据线脉冲的电平。 0
0=正常；1=翻转

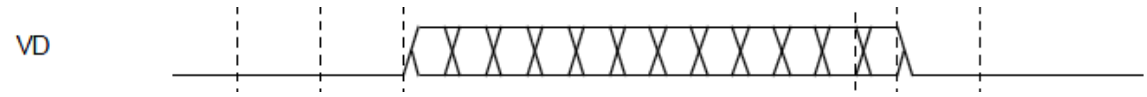


INVVD [7]：数据一般用 0 表示 1，VD0 ~ VD23 这些数据引脚。这个不需要反转。故：
INVVD [7]：0 数据脉冲的电平不需要反转。

INVVDEN	[6]	**TFT**: This bit indicates the VDEN signal polarity. 0 = normal 1 = inverted	0

查看 LCD 屏驱动器时序：DTMG 就是 VDEN

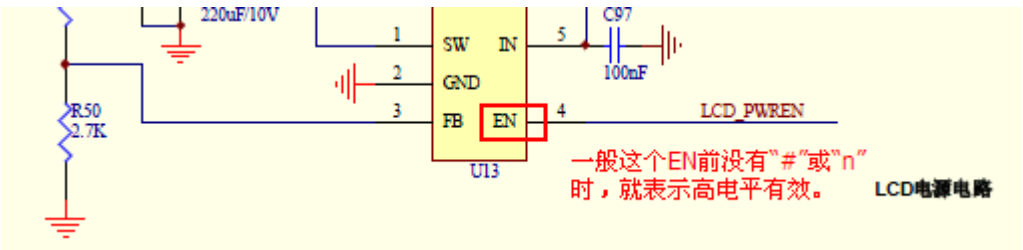
再看 2440 手册上 LCD 控制器时序：



可见都是“上升沿”有效，所以这里不需要反转。故：
INVVDEN [6] ：设置为 0 不需要反转。

INVPWREN	[5]	**STN/TFT**: This bit indicates the PWREN signal polarity. 0 = normal 1 = inverted	0

INVPWREN [5] STN/TFT:PWREN 脉冲的电平。 0
0=正常；1=翻转



INVPWREN [5]：PWREN 信号也不需要反转。PWREN 电源使能信号。高电平有效。原理图上这种“EN”信号若前面没有一个“#”号，或者没有一个“n”，一般就表示是高电平有效。故：
INVPWREN [5] ： 0 。表示高电平有效不需要反转。

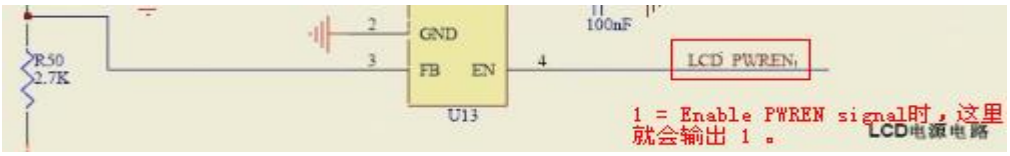
INVLEND	[4]	**TFT**: This bit indicates the LEND signal polarity. 0 = normal 1 = inverted	0

这个“LEND”信号没有用到，这里不管。

PWREN	[3]	**STN/TFT**: LCD_PWREN output signal enable/disable. 0 = Disable PWREN signal 1 = Enable PWREN signal	0

PWREN	[3]	STN/TFT:LCD_PWREN 信号输出允许。 0=禁止；1=允许	0

是否使能这个 LCD_PWREN 引脚。“1 = Enable PWREN signal”时就会输出 1.



这里先不要，等设置完后，再来使能它。故：
PWREN [3]：先设置为“0”（输出低电平），以后再使能它。

ENLEND	[2]	**TFT**: LEND output signal enable/disable. 0 = Disable LEND signal 1 = Enable LEND signal	0

ENLEND	[2]	TFT:LEND 信号输出允许。 0=禁止；1=允许	0

ENLEND [2]：来能管这位。

BSWP	[1]	**STN/TFT**: Byte swap control bit. 0 = Swap Disable 1 = Swap Enable	0
HWSWP	[0]	**STN/TFT**: Half-Word swap control bit. 0 = Swap Disable 1 = Swap Enable	0

BSWP	[1]	STN/TFT:Byte swap 控制位。 0=禁止；1=允许	0
HWSWP	[0]	STN/TFT:Half-Word swap 控制位。 0=禁止；1=允许	0

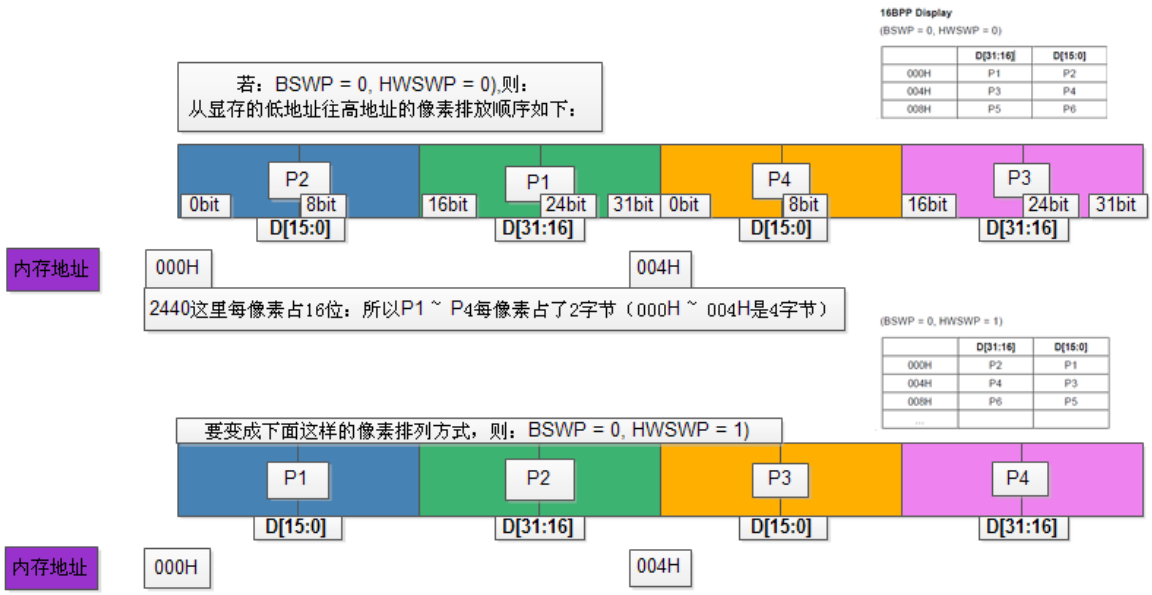
指字节交换。搜索 2440 手册中的 BSWP 和 HWSWP。见到如下图：像素在显存中的排放问题。

16BPP Display

(BSWP = 0, HWSWP = 0)

	D[31:16]	D[15:0]
000H	P1	P2
004H	P3	P4
008H	P5	P6
...		

我们 2440 是每像素为 16 位，就对应上面 16BPP 显示。这个图是指显存。上面“(BSWP = 0, HWSWP = 0)”，则当显存如下从 000H 开始 004H 处增大时（内存以字节为单位），第一个像素 P1 在高位[31:16].(000H 是内存地址，0 地址。)，而 P2 在低 16 位处。这样使得 P2 在 P1 的内存地址前面。



故：bit[1] : 0 = BSWP bit[0] : 1 = HWSWP

③，显卡自带了显存但这里没有：

从内存里分配显存(framebuffer)，并把地址告诉 LCD 控制器.
设置好了 LCD 控制器后，就会自动从显存里取一个像素的值通过 VD0~VD23 发送出像素数据到 LCD 屏上去。然后再取下一个，周而复始，取到显存中最后一个时（显存就那么大）就再到显存开始处取。

a， **故而要分配显存。**

这个显存要让它 物理地址 连续（因为 LCD 控制器没有那么智能，所以要连续。）。分配时不能用 kmolloc(),要用专
门的函数来分配这个块显存。

查看内核中“s3c2410fb.c”中分配显存的方式：
dma_alloc_writecombine()

```
/* Initialize video memory */
ret = s3c2410fb_map_video_memory(info);
if (ret) {
     printk( KERN_ERR "Failed to allocate video RAM: %d\n", ret);
     ret = -ENOMEM;
     goto release_clock;
}
dprintk("got video memory\n");
```

```
void *
dma_alloc_writecombine(struct device *dev, size_t size, dma_addr_t *handle, gfp_t gfp)
{
    return __dma_alloc(dev, size, handle, gfp,
              pgprot_writecombine(pgprot_kernel));
}
```

s3c2410fb_map_video_memory(info)
 -->fbi->map_cpu = dma_alloc_writecombine(fbi->dev,
fbi->map_size,&fbi->map_dma, GFP_KERNEL);
参 1 是设备：我们写为 NULL。
参 2 是大小：上面已经算出来，“s3c_lcd->fix.smem_len = 240*320*16/8;
//显存的长度,查 LCD 手册.每像素
占 16 位即 2 字节.”
参 3“dma_addr_t”实际上就是物理地址。

```
/* Dma addresses are 32-bits wide.  */

typedef u32 dma_addr_t;
typedef u32 dma64_addr_t;
```

实际上就是 unsigned int 类型。
这个物理地址就是我们要填写的：s3c_lcd->fix.smem_start = xxx; //显存的物理地址.
参 4 是标记：从 s3c2410fb.c 中借鉴为：GFP_KERNEL

```
# define GFP_KERNEL 0
```

返回值：这块物理内存的虚拟地址。之前配置 LCD 参数时这个虚拟地址没有配置：
//s3c_lcd->screen_base = ?; //显存的虚拟地址.在
fbmem.c 的 fb_write()有用到.在分配了显存后再设置.
所以：

```
s3c_lcd->screen_base = dma_alloc_writecombine(NULL,
s3c_lcd->fix.smem_len, &s3c_lcd->fix.smem_start, GFP_KERNEL);
```

b,以上只是分配了这块显存，接着要把这块显存的地址告诉 LCD 控制器。这样才知道从哪里取像素数据。
查看 2410 手册：之前已经设置了 LCDCON1~5 个寄存器，还有其他的：

LCDSADDR1 ： 帧缓冲器开始地址 1 寄存器

FRAME BUFFER START ADDRESS 1 REGISTER

Register	Address	R/W	Description	Reset Value
LCDSADDR1	0X4D000014	R/W	**STN/TFT**: Frame buffer start address 1 register	0x00000000

LCDSADDR1	Bit	Description	Initial State
LCDBANK	[29:21]	These bits indicate A[30:22] of the bank location for the video buffer in the system memory. LCDBANK value cannot be changed even when moving the view port. LCD frame buffer should be within aligned 4MB region, which ensures that LCDBANK value will not be changed when moving the view port. So, care should be taken to use the malloc() function.	0x00
LCDBASEU	[20:0]	For dual-scan LCD : These bits indicate A[21:1] of the start address of the upper address counter, which is for the upper frame memory of dual scan LCD or the frame memory of single scan LCD. For single-scan LCD : These bits indicate A[21:1] of the start address of the LCD frame buffer.	0x000000

这些位表明系统存储器中视频缓冲器的 bank 位置的 A[30:22]。即使当移动视口时也不能改变 LCDBANK 的值。LCD 帧缓冲器应该在 4MB 连续区域内，以保证当移动视口时不会改变 LCDBANK 的值。因此应该谨慎使用 malloc()函数。

对于单扫描 LCD：这些位表明 LCD 帧缓冲器的开始地址的 A[21:1]。

Frame buffer 开始地址寄存器 1.

LCDBANK [29:21]：代表“video buffer”视频缓冲区的 A[30:22]

LCDBASEU [20:0]：对双列扫描，但我们是单列一行一行扫描的。所以用下面单列扫描。

For single-scan LCD : These bits indicate A[21:1] of the start address
of the LCD frame buffer.

所以上面 lcdaddr1 寄存器是存放显存地址的 A[30:22] A[21:1]（A30 ~ A1）就是 bit0（A0）和 bit31(最高位）不要。
故，这个“LCDSADDR1”寄存器存放 A30 ~ A1 .这个寄存器的设置：物理地址左移一位。

```
lcd_regs->lcdsaddr1 = &s3c_lcd->fix.smem_start << 1; //最低位不要。
```

最高位也不要，但最高位设置时不用关心了。

LCDSADDR1 是[29:21] [20:0] 是最高两位 bit30~31 不需要。所以把最高两们清为 0 ：
~(3<<30)；3 左移 30 位后，最高两位就是“11”，再取反就是“00”。
最后结果是：
lcd_regs->lcdsaddr1 = (s3c_lcd->fix.smem_start << 1) && ~(3<<30);

LCDSADDR2 - 帧缓冲器开始地址 2 寄存器：

FRAME Buffer Start Address 2 Register

Register	Address	R/W	Description	Reset Value
LCDSADDR2	0X4D000018	R/W	**STN/TFT**: Frame buffer start address 2 register	0x00000000

LCDSADDR2	Bit	Description	Initial State
LCDBASEL	[20:0]	For dual-scan LCD: These bits indicate A[21:1] of the start address of the lower address counter, which is used for the lower frame memory of dual scan LCD. For single scan LCD: These bits indicate A[21:1] of the end address of the LCD frame buffer. LCDBASEL = ((the frame end address) >>1) + 1 = LCDBASEU + (PAGEWIDTH+OFFSIZE) x (LINEVAL+1)	0x0000

对于单扫描 LCD：这些位表明 LCD 帧缓冲器的结束地址的 A[21:1]。
LCDBASEL = ((帧结束地址) >> 1) + 1
= LCDBASEU +
(PAGEWIDTH+OFFSIZE) × (LINEVAL+1)
我们是单列扫描，是指结束地址的 A[21:1].我们的结束地址是“s3c_lcd->fix.smem_start + s3c_lcd->fix.smem_len”，
就是“显存物理地址--s3c_lcd->fix.smem_start”加上“显存大小--s3c_lcd->fix.smem_len”。这样就能表示显存的结束地址了。
结束地址的 A[21:1]，就是右移一位。

然后它只需要 [20:0] 这 21 位，就要 “&”上 “0x1fffff”。5 个 f 就是 20 位。

结果是：
lcd_regs->lcdsaddr2 = ((s3c_lcd->fix.smem_start + s3c_lcd->fix.smem_len) >> 1) & 0x1fffff;

帧缓冲器开始地址 3 寄存器：

FRAME Buffer Start Address 3 Register

Register	Address	R/W	Description	Reset Value
LCDSADDR3	0X4D00001C	R/W	**STN/TFT**: Virtual screen address set	0x00000000

LCDSADDR3	Bit	Description	Initial State
OFFSIZE	[21:11]	Virtual screen offset size (the number of half words). This value defines the difference between the address of the last half word displayed on the previous LCD line and the address of the first half word to be displayed in the new LCD line.	00000000000
PAGEWIDTH	[10:0]	Virtual screen page width (the number of half words). This value defines the width of the view port in the frame.	000000000

OFFSIZE [21:11]：
虚拟屏偏移尺寸（半字数）。
此值定义了显示在之前 LCD 行的最后半字的地址与要在新 LCD 行中显示的第一半字的地址之间的差。

我们的真实屏幕分辨率和虚拟分辨率是一样的（上面有定义），所以这之间的偏移值是 0.即 OFFSIZE 域设置为 0.

PAGEWIDTH [10:0]：
虚拟屏页宽度（半字数）。
此值定义了帧中视口的宽度。
PAGEWIDTH 就是一行所占据的大小。单位是“半字”，即 2 字节。
一个像素是 16 位，单位是“半字”即 16 位。

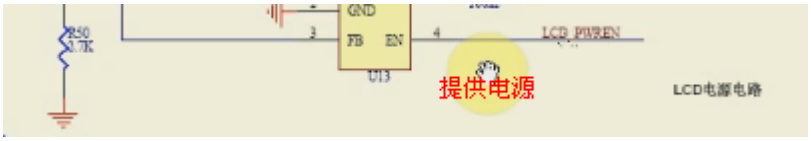
我们的 LCD 屏，X 方向即行是 240 分辨率，Y 方向即垂直方向上是 320 分辨率。这里 PAGEWIDTH 是说一行所占据的大小，所以，应该是 （分辨率 * 像素大小 -- 我们这里 RGB--565,即每像素 16 位），即 240 * 16.这里 PAGEWIDTH 的单位是半字即 16bit。故结果如下。

结果：
```
lcd_regs->lcdsaddr3 = (240*16/16); //一行的长度(单位: 2 字节)
```

看 2440 手册上的例 3：（看不懂）
LCD 面板 = 320 × 240，彩色，单扫描
帧开始地址 = 0x0C500000
偏移点数 = 1024 个点（512 半字）
LINEVAL = 240 - 1 = 0xEF
PAGEWIDTH = 320 x 8 / 16 = 0xA0
OFFSIZE = 512 = 0x200
LCDBANK = 0x0C500000 >> 22 = 0x31
LCDBASEU = 0x100000 >> 1 = 0x80000
LCDBASEL = 0x80000 + (0xA0 + 0x200) x (0xEF + 1) = 0xA7600

以上设置完后，就可以注册 framebuffer 了（使能它）：
1，打开背光：

2，打开 LCD 驱动器本身的电源。



可以直接在 LCD 硬件操作中直接打开，这样别人不能控制打开的接口。但查看 s3c2410fb.c 中的使能 framebuffer 代码时，看提供的一些接口可以让别人打开代码中涉及到 电源管理。所以这里就直接打开算了，电源管理以后再单独研究。

先启动 LCD 驱动器本身.只要设置 LCD 控制寄存器 1 即可。

LCD Control 1 Register

Register	Address	R/W	Description	Reset Value
LCDCON1	0X4D000000	R/W	LCD control 1 register	0x00000000

ENVID	[0]	LCD video output and the logic enable/disable. 0 = Disable the video output and the LCD control signal. 1 = Enable the video output and the LCD control signal.	0

ENVID 设置 ：bit[0] ：0 LCD 视频输出和逻辑使能/禁止。设置完后再使能。之前是禁止的，这里要开启，即让此位域 ENVID 为 “1”。
所以直接把 LCDCON1 或上 1 即可：
lcd_regs->lcdcon1 |= (1<<0); //使能 LCD 控制器

还要加上一句代码：最后测试时“cat lcd.ko >/dev/fb0”没有显示花屏，是这句少了：
lcd_regs->lcdcon3 |= (1<<3); //这个要设置，不然 LCD 不显示。这是使能 LCD 本身。

打开 LCD 背光：看 GPIO 中 GPB0 输出高电平。原来是输出低电平：
gpbdat &= ~1; / 输出低电平.与上 ’1 的取反’ 即与上 0，结果为 0，输出低电平 。意思是先不让背光开启.*/
现在要打开背光，即：
gpbdat |= 1; / 或上 1 也是先使其输出高电平，意思是打开背光.*/

开启 LCD 能使用：LCD 控制器使能，LCD 本身电源使能，LCD 背光开启。

```
    //4.3.3,启动LCD。
        //4.3.3.1, 先启动LCD控制器 和 驱动器本身.(一个是LCD控制器，一个是LCD电源，一个是LCD背光
lcd_regs->lcdcon1 |= (1<<0); //使能LCD控制器
lcd_regs->lcdcon3 |= (1<<3); //这是全能LCD本身(LCD驱动器)这个要设置，不然LCD不显示。
        //4.3.3.2,启动LCD背光。
 *gpbdat |= 1;     /* 或上1也是先使其输出高电平，意思是打开背光灯.*/
```

以上代码基本完成，作一个备份：3_lcd_LCD 硬件操作完成_基本可以作实验去了.c

代码写到这里，基本上就完成了 LCD 硬件的操作，就可以去做实验了。但可能还有点问题。就是调色板的问题。上面代码的调色板还没有设置。并且在 fb_ops s3c_lcdfb_ops 结构中还有一个“.fb_setcolreg”没有完成。

```
//3.4，其他设置(看结构体fb_info的定义中还有什么,参考别人的代码):
//s3c_lcd->pseudo_palette = ?;  //假的调色码   调色板
//s3c_lcd->screen_base = ?;     //显存的虚拟地址.在fbmem.c的fb_write()有用到.在分配了显存后再设置.
s3c_lcd->screen_size = 240*320*16/8;      //显存的大小
```

//3.3.1,为此 LCD 驱动定义一个 fb_ops 结构变量 s3c_lcdfb_ops。成员定义如下:

```
static struct fb_ops s3c_lcdfb_ops = {
    .owner      = THIS_MODULE,
    //.fb_setcolreg   = s3c_lcdfb_setcolreg, 以后会用到。
    .fb_fillrect    = cfb_fillrect,
    .fb_copyarea    = cfb_copyarea,
    .fb_imageblit   = cfb_imageblit, //上面三个一般都有，是别人提供的.
};
```

调色板：

从硬件 LCD 手册上知道，每个像素需要 16 位的像素数据。（LCD 每像素需要 16bpp），显存也是给每像素留着 16 位，这刚好合适，LCD 控制器直接把显存数据取过来直接发给 LCD 屏就可以。

但是当我想节省内存时，在显存里只想给一个像素 8 位空间，这时 LCD 控制器从显存中取到一个 8 位的数据后，如何转换成 一个硬件 LCD 上需要的 16bit 的数据？这个中间就引入了一个 转换 的过程：（调色板）

调色板是一块内存。

LCD 控制器从显存里得到的 8bit 数据，此时并不是直接发给 LCD 屏。而是发给调色板，调色板是块内存，里面存放了真正的 16bit 数据。LCD 控制器从显存得到 8bit 数据后，以它作为 索引，假如一个 8 位是 100，这时就从“调色板”（像个数组）中找到 100 项，把这个第 100 项处的 16bit 取出来。发送给 LCD 屏。

调色板，就像一个里面放好颜色的颜料盒，要取哪种颜色就通过 索引来检索出来哪个颜色在调色板这个颜色盒的哪里，找到后拿出来。

这里调色板是块内存，事先配置好几种颜色，LCD 控制器要用时，从显存中读到一个 8 bit 数据，以此 8bit 数据作为一个“索引”值，从“调色板”这个数组里取出一个真正的颜色，再把这个真正的颜色发送给 LCD 硬件。

而当我们 LCD 屏每像素是 RGB-565,是 16 位时，显存中也是每 16 位存一个像素数据，这样就用不到调色板了。但可能是为了兼容以前的程序，我们的代码里要提供一个“假的调色板”。

假的调色板设置：

在“s3c2410fb.c”中：

```
fbinfo->pseudo_palette       = &info->pseudo_pal;
```

而“pseudo_pal”是一个数组。

```
u32                                                pseudo_pal[16];
```

其他 LCD 程序中也有这个“u32 pseudo_pal[16]”，所以我们的代码里为了兼容也写个这个的数组（调色板）。

先在 info_fb 结构中加上这个 pseudo_palette[16] 成员定义：

故代码结果：

```
//4.3.4,假的调色板需要这样一个数据.
Static u32                 pseudo_palette[16]; //我们自定义的数组名是 pseudo_palette
//3.4，其他设置(看结构体 fb_info 的定义中还有什么,参考别人的代码):
s3c_lcd->pseudo_palette = pseudo_palette;  //假的调色码
```

再看这个假的调色板由谁来设置：

就是由“struct fb_ops s3c_lcdfb_ops”结构中的“.fb_setcolreg = s3c_lcdfb_setcolreg,”来设置。

查看“s3c2410fb.c”中“s3c2410fb_setcolreg（）”：

```
int s3c2410fb_setcolreg(unsigned regno,
       unsigned red, unsigned green, unsigned blue,
       unsigned transp, struct fb_info *info)
```

我们仿照的写：

```
//4.3.5,设置调用假的调色板的函数。函数在 struct fb_ops s3c_lcdfb_ops 中定义。
u32                                                pseudo_pal[16];
static int s3c_lcdfb_setcolreg(unsigned regno,
       unsigned red, unsigned green, unsigned blue,
       unsigned transp, struct fb_info *info)
{
}
```

调色板就像放红，绿，蓝颜料的画具"调色板"。

参 1 是指哪个“画具--颜料盘“调色板”这里就是指那个数组“调色板就是定义的一个数组”，如：

U32 pseudo_pal[16];

参 2 为红。
参 3 为绿。
参 4 为蓝。
参 5 为透明度。
参 6 为 fb_info 结构：我们的代码这里就是“s3c_lcdfb_ops”。

这里面主要是用了一个“chan_to_field（）”函数。

下面分析下流程：

先用参 1：

在固定参数里设置过“真彩色”：所以我们只看“s3c2410fb.c->s3c2410fb_setcolre()”下面部分代码：
参考 s3c2410fb.c 中假调色板中的设置：

```
switch (fbi->fb->fix.visual) {
    case FB_VISUAL_TRUECOLOR: //真彩色
        /* true-colour, use pseuo-palette */

        if (regno < 16) { //真彩色里若这个调色板数组中小于16个组元就设置否则返回1
            .....
            .....
        }
        break;
    default:
        return 1;    /* unknown type */
```

“FB_VISUAL_TRUECOLOR”宏为 2 是指“真彩色”。

```
/* 2.1 设置固定的参数 */
strcpy(s3c_lcd->fix.id, "mylcd");
s3c_lcd->fix.smem_len = 240*320*16/8;
s3c_lcd->fix.type     = FB_TYPE_PACKED_PIXELS;
s3c_lcd->fix.visual   = FB_VISUAL_TRUECOLOR; /* TFT */
s3c_lcd->fix.line_length = 240*2;
```

unsigned int val; //,设置后的值。
    if(regno > 16) //若我们的假调色板数组组元大于 16 就返回 1.我们定义时 pseude_pal[16]为 16 组元.
        return 1; //返回 1 是参考 s3c2410fb.c 中的假调色板函数.

再用红，绿，蓝三个颜色，得到一个 val 值：参考 s3c2410fb.c

```
u32 *pal = fbi->fb->pseudo_palette;

val  = chan_to_field(red,   &fbi->fb->var.red);
val |= chan_to_field(green, &fbi->fb->var.green);
val |= chan_to_field(blue,  &fbi->fb->var.blue);
```

把 val 值放到调色板里：

```
pal[regno] = val;
```

其中这个“chan_to_field（）”的分析如下：

```
s3c_lcd->var.red.offset   = 11; //RGB--565:红色的偏移值从bit11开始.
s3c_lcd->var.red.length   = 5;  //RGB--565,则红色占5位.

s3c_lcd->var.green.offset = 5; //RGB--565:绿色的偏移值从bit5开始.
s3c_lcd->var.green.length = 6;  //RGB--565,则绿色占6位.

s3c_lcd->var.blue.offset  = 0; //RGB--565:蓝色的偏移值从bit0开始.
s3c_lcd->var.blue.length  = 5;  //RGB--565,则蓝色占5位.
```

val = chan_to_field(red, &fbi->fb->var.red);
把 red 色根据“&fbi->fb->var.red”这个位。Var.red 是表示红色从哪一位开始，占据多少长度。这里就是根

据这个值来存放。imxfb_activate_var（）”定义：
看看“s3c2410fb.c”中“”

```
static inline unsigned int chan_to_field(unsigned int chan, struct fb_bitfield *bf)
{
    chan &= 0xffff;
    chan >>= 16 - bf->length;
    return chan << bf->offset;
}
```

形参 chan 是指“red”或者是"green"再或者是"blue"。
chan &= 0xffff; 意思是只保留低 16 位（4 个 f 为 16 位）。形参 chan 是 int 型为 32 位，这里只保留低 16 位。
chan >>= 16 - bf->length; 假设这个“bf->length”是 RGB-565 中的 5 位（红色点一个像素点 16 位中的 5 位）。则这里是 chan >> 16-5 .即“chan”右移 11 位，就是把低 11 位全部清掉了。这时就保留了高的 5 位了。

```
s3c_lcd->var.red.offset   = 11; //RGB--565:红色的偏移值从bit11开始.
s3c_lcd->var.red.length   = 5;  //RGB--565,则红色占5位.
```

接着把这个“高 5 位”左移一个偏移。就得到一个值为 red 来用。
return chan << bf->offset; 上面得到高 5 位后，再左移一个“偏移”。就表示了红色点一个像素中 16 位的[15:11]这 5 位。

最后，我们的代码则：

```
//4.3.4,假的调色板需要这样一个数据.
static u32                  pseudo_pal[16];

//4.3.6.1,构造这个 chan_to_field()函数。
static inline u_int chan_to_field(u_int chan, struct fb_bitfield *bf)
{
chan &= 0xffff;
chan >>= 16 - bf->length;
return chan << bf->offset;
}

//4.3.5,设置调用假的调色板的函数。函数在 struct fb_ops s3c_lcdfb_ops
中定义。
//调色板就像放红，绿，蓝颜料的画具"调色板"。
static int s3c_lcdfb_setcolreg(unsigned regno,
       unsigned red, unsigned green, unsigned blue,
       unsigned transp, struct fb_info *info)
{
    unsigned int val;   //a,设置后的值。
    if(regno > 16)      //若我们的假调色板数组组元大于 16 就返回 1.我
们定义时 pseude_pal[16]为 16 组元.
        return 1;       //返回 1 是参考 s3c2410fb.c 中的假调色板函数.

    //4.3.6,用红，绿，蓝三原色构造出 val 值。
    val  = chan_to_field(red,   &info->var.red);
    val |= chan_to_field(green, &info->var.green);
val |= chan_to_field(blue,  &info->var.blue);

    //((u32)(info->pseudo_palette))[regno] = val; //因为
pseudo_palette 是 void*型，这是强制转换.这样写比较通用.
    pseudo_palette[regno] = val; //因为是设置自己的假调色板，就直接这
样写不加 info。
    return 0;
}
```

最后写完“出口函数”：

```
//1.2,出口函数;
static void lcd_exit(void)
```

```
{    //1.2.1,注释去这个结构。
    unregister_framebuffer(s3c_lcd);
    //1.2.2,省电
    lcd_regs->lcdcon1 &= ~(1<<0); //关掉 LCD 本身的电.
    *gpbdat &= ~1; //输出低电平，禁用 LCD 背光.
    //1.2.3，释放掉分配的显存。
    dma_free_writecombine(NULL, s3c_lcd->fix.smem_len,
s3c_lcd->screen_base, s3c_lcd->fix.smem_start);
    //1.2.4,iounmap();
    iounmap(lcd_regs);
iounmap(gpbcon);
iounmap(gpccon);
iounmap(gpdcon);
iounmap(gpgcon);
    //1.2.5,释放 framebuffer 结构体.
framebuffer_release(s3c_lcd);
}
```

测试：

make menuconfig 去掉原来的驱动程序，

不然两个驱动程序会冲突。
-> Device Drivers
  -> Graphics support
<M> S3C2410 LCD framebuffer support
上面“<*> S3C2410 LCD framebuffer support”是之前自带的驱动程序，这里去掉后以<M>模块方式编译进内核。
设置为<M>是下面三个函数：

```
static struct fb_ops s3c_lcdfb_ops = {
    .owner      = THIS_MODULE,
    .fb_setcolreg   = atmel_lcdfb_setcolreg,
    .fb_fillrect    = cfb_fillrect,
    .fb_copyarea    = cfb_copyarea,
    .fb_imageblit   = cfb_imageblit,
};
```

这三个函数一会也得编译成模块文件（.ko）

2. make uImage

make modules
主要是想得到上面三个函数所得到的三个.ko 文件。

```
Entry Point:  0x30008000
  Image arch/arm/boot/uImage is ready
book@book-desktop:/work/system/linux-2.6.22.6$ cp arch/arm/boot/uImage /work/nfs_root/uImage_nolcd
```

3. 使用新的 uImage 启动开发板：

```
[b] Boot the system
[r] Reboot u-boot
[q] Quit from menu
Enter your selection: q
OpenJTAG> nfs 30000000 192.168.1.5:/work/nfs_root/uImage_nolcd
dm9000 i/o: 0x20000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 08:00:3e:26:0a:5b
```

重启开发板，用新的内核来启动。

通过网络把“uImage_nolcd”新的内核镜像烧到 30000000 地址处。

```
MAC: 08:00:3e:26:0a:5b
could not establish link
File transfer via NFS from server 192.168.1.5; our IP address is 192.168.1.17
Filename '/work/nfs_root/uImage_nolcd'.
Load address: 0x30000000
Loading: ###################################################################
         ###################################################################
         ###################################################################
         ###################################################################
         ###################################################################
         ##################################
done
Bytes transferred = 1842392 (1c1cd8 hex)
OpenJTAG> bootm 30000000
```

再通过“bootm 30000000”从 30000000 处启动内核。

编译：出错：

```
--
/work/nfs_root/romfs/10_lcd/lcd.c: At top level:
/work/nfs_root/romfs/10_lcd/lcd.c:256: warning: function declaration isn't a prototype
```

```
00255: static void lcd_exit() 要写void形参
00256: {    /*1.2.1.注释去这个结构。*/
```

最后：

```
root@ian:/work/nfs_root/romfs/10_lcd# make
make -C /work/system/linux-2.6.22.6 M=`pwd` modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
  CC [M]  /work/nfs_root/romfs/10_lcd/lcd.o
/work/nfs_root/romfs/10_lcd/lcd.c: In function `lcd_init':
/work/nfs_root/romfs/10_lcd/lcd.c:235: warning: passing arg 3 of `dma_alloc_writecombine' from incompatible pointer t
ype
  Building modules, stage 2.
  MODPOST 1 modules
  LD [M]  /work/nfs_root/romfs/10_lcd/lcd.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
root@ian:/work/nfs_root/romfs/10_lcd#
```

中间那个 警告 不知道是什么意思。

接着，将 lcd.ko 拷贝到 nfs 根文件系统。再挂载：

```
starting pid 767, tty '/dev/s3c2410_serial0': '/bin/sh'
#
# mount -t nfs -o nolock,vers=2 192.168.1.5:/work/nfs_root/first_fs /mnt
# cd /mnt/
```

直接挂载时，说下面三个函数不识别：

```
# insmod lcd.ko
lcd: Unknown symbol cfb_fillrect
lcd: Unknown symbol cfb_imageblit
lcd: Unknown symbol cfb_copyarea
insmod: cannot insert 'lcd.ko': Unknown symbol in module (-1): No such file or directory
#
```

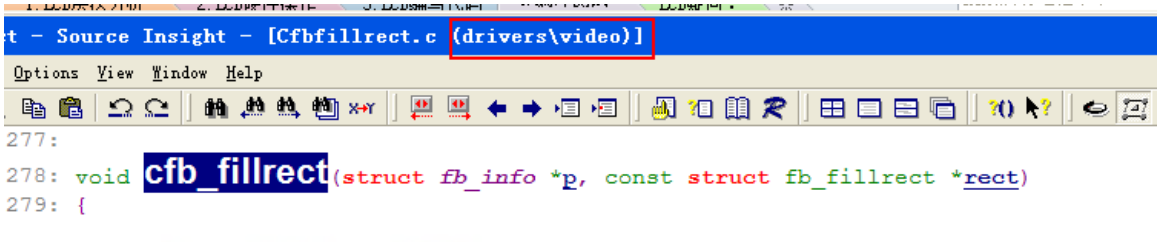
意思要先按着 这三个 模块。（它们三个本身并不是驱动，只是提供了三个函数的模块）

编译内核时，make menuconfig -->make uImage & make modules 了，

```
//3.3.1,为此LCD驱动定义一个 fb_ops 结构变量 s3c_lcdfb_ops。成员定义如下：
static struct fb_ops s3c_lcdfb_ops = {
    .owner      = THIS_MODULE,
    .fb_setcolreg   = s3c_lcdfb_setcolreg, //以后会用到。设置假的调色板会用到.
    .fb_fillrect    = cfb_fillrect,
    .fb_copyarea    = cfb_copyarea,
    .fb_imageblit   = cfb_imageblit, //上面三个一般都有，是别人提供的.
};
```

这三个驱动模块点击一个跳转

发现它们在“drivers\video”目录下：




```
book@book-desktop:/work/drivers_and_test/10th_lcd/4th$ cp lcd.ko /work/nfs_root/first_fs
book@book-desktop:/work/drivers_and_test/10th_lcd/4th$ cd -
/work/system/linux-2.6.22.6
book@book-desktop:/work/system/linux-2.6.22.6$ cp drivers/video/
Display all 190 possibilities? (y or n)
book@book-desktop:/work/system/linux-2.6.22.6$ cp drivers/video/cfb
cfbcopyarea.c      cfbcopyarea.mod.o  cfbfillrect.ko     cfbfillrect.o      cfbimgblt.mod.c
cfbcopyarea.ko     cfbcopyarea.o      cfbfillrect.mod.c  cfbimgblt.c        cfbimgblt.mod.o
cfbcopyarea.mod.c  cfbfillrect.c      cfbfillrect.mod.o  cfbimgblt.ko       cfbimgblt.o
book@book-desktop:/work/system/linux-2.6.22.6$ cp drivers/video/cfb*.ko /work/nfs_root/first_fs
book@book-desktop:/work/system/linux-2.6.22.6$
```

将它们拷贝到 nfs 网络文件系统中。

4. 依次加载这个函数的模块：

insmod cfbcopyarea.ko

insmod cfbfillrect.ko

insmod cfbimgblt.ko

接着就可以装载“lcd.ko”，在装载前：

```
# ls /dev/fb*
ls: /dev/fb*: No such file or directory
#
```

装载前没有 fb* 设备文件。

insmod lcd.ko

```
# insmod lcd.ko
Console: switching to colour frame buffer device 30x40
# ls /dev/fb*
/dev/fb0
#
```

这时也可以看到 LCD 屏亮起来了。

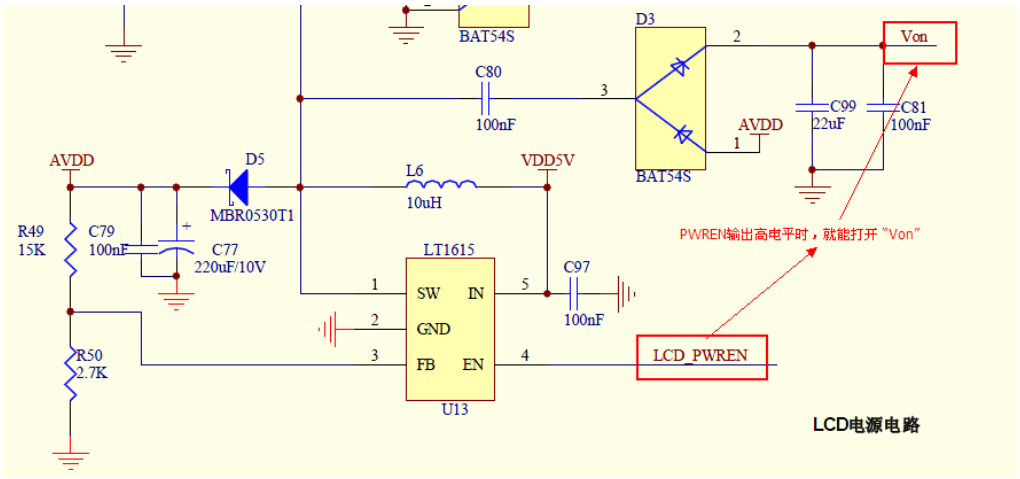
接着看是否可以操作 LCD 屏。

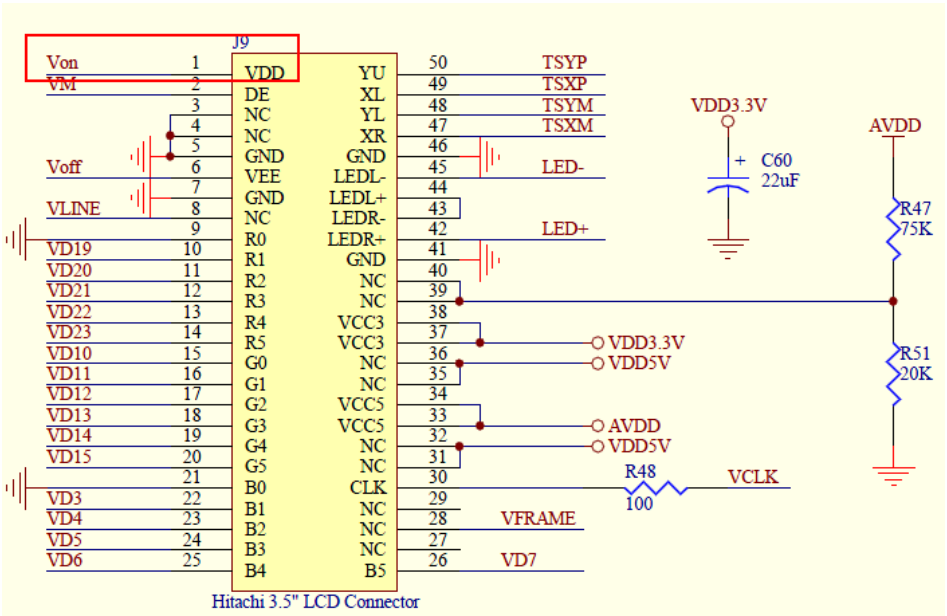
接着测试下 LCD 是否可显示：

```
# cat lcd.ko > /dev/fb0
#
Ready
```

虽然把一个内容送到 LCD，上面的操作应该会使 LCD 花屏。但这时却没有显示：

应该是 bit3 这里的问题：





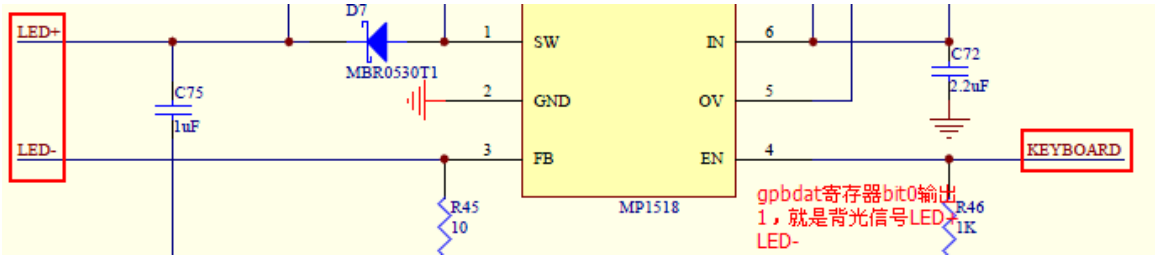
让这个 LCD_PWREN 引脚变成 高电平，就能打开 “Von” ,给这个 LCD 本身提供 电平。

```
     * bit[4] : 0 = VDEN不用反转
     * bit[3] : 0 = PWREN输出0(输出低电平) .控制是否使能LCD本身(LCD驱动器)
     * bit[1] : 0 = BSWP
     * bit[0] : 1 = HWSWP 2440手册P413
     */
lcd_regs->lcdcon5 = (1<<11) | (0<<10) | (1<<9) | (1<<8) | (1<<0);//bit?左移0位可以不写。


//4.3.显卡自带了显存但这里没有:从内存里分配显存(framebuffer)，并把地址告诉LCD控制器.
    //4.3.1,分配显存
s3c_lcd->screen_base = dma_alloc_writecombine(NULL, s3c_lcd->fix.smem_len, &s3c_lcd->fix.sm
    //4.3.2,将显存的地址告诉LCD控制器.设置FRAME Buffer Start Address 1~3 Register
lcd_regs->lcdsaddr1 = (s3c_lcd->fix.smem_start << 1) && ~(3<<30);
lcd_regs->lcdsaddr2 = ((s3c_lcd->fix.smem_start + s3c_lcd->fix.smem_len) >> 1) & 0x1fffff;
lcd_regs->lcdsaddr3 = (240*16/16); //一行的长度(单位: 2字节)

//s3c_lcd->fix.smem_start = xxx; //显存的物理地址.

    //4.3.3,启动LCD。
        //4.3.3.1, 先启动LCD控制器 和 驱动器本身.(一个是LCD控制器，一个是LCD电源，一个是LCD背光)
lcd_regs->lcdcon1 |= (1<<0); //使能LCD控制器
lcd_regs->lcdcon3 |= (1<<3); //这是使能LCD本身(LCD驱动器)，输入高电平后，LCD本身才有电平。
        //4.3.3.2,启动LCD背光。
 *gpbdat |= 1;     /* 或上1也是先使其输出高电平，意思是打开背光灯.*/
```

开启 LCD 能使用：LCD 控制器使能，LCD 本身电源使能，LCD 背光开启。

```
    //4.3.3,启动LCD。
        //4.3.3.1, 先启动LCD控制器 和 驱动器本身.(一个是LCD控制器，一个是LCD电源，一个是LCD背光)
lcd_regs->lcdcon1 |= (1<<0); //使能LCD控制器
lcd_regs->lcdcon3 |= (1<<3); //这是使能LCD本身(LCD驱动器)，输入高电平后，LCD本身才有电平。
        //4.3.3.2,启动LCD背光。
 *gpbdat |= 1;     /* 或上1也是先使其输出高电平，意思是打开背光灯.*/
```


Gpbdat 寄存器的 bit0 设置为“1”，就是输出背光信号“LED+”“LED1”.



重新编译代码，再加载驱动：

```
# rmmod lcd
rmmod: lcd: Resource temporarily unavailable
# reboot
```

卸载不了，就重启开发板。

```
# mount -t nfs -o nolock,vers=2 192.168.1.5:/work/nfs_root/first_fs /mnt
# cd /mnt/
# insmod cfb
cfbcopyarea.ko  cfbfillrect.ko  cfbimgblt.ko
# insmod cfbcopyarea.ko
# insmod cfbfillrect.ko
# insmod cfbimgblt.ko
# insmod lcd.ko
Console: switching to colour frame buffer device 30x40
#
#
# echo hello > /dev/tty1
#
```

这时 LCD 上就有显示了。

echo hello > /dev/tty1 // 可以在 LCD 上看见 hello

cat lcd.ko > /dev/fb0 // 花屏.是直接把 lcd.ko 中的内容直接扔到 LCD 上，根本不知道里面是什么格式的内容，肯定是花屏。

5. 修改 /etc/inittab

tty1::askfirst:-/bin/sh

用新内核重启开发板

```
s3c2410_serial0::askfirst:-/bin/sh
```

这是启动一个“-/bin/sh”程序，这个 shell 程序从“s3c2410_serial0”串口 0 得到输入，把输出输出串口 0 上来。

现在再启动一个“shell”程序，这个 shell 程序从 device“tty1”(tty1 就对应我们的键盘)，device tty1 输出时就对应这里的“LCD”。

```
# /etc/inittab
::sysinit:/etc/init.d/rcS
s3c2410_serial0::askfirst:-/bin/sh
tty1::askfirst:-/bin/sh
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
```

/dev/tty1 是用的如下驱动程序：

linux/drivers/video/fbcon.c -- Low level frame buffer based console driver

这个驱动程序最终也会用到“framebuffer”.

```
fb_info = registered_fb[con2fb_map[ops->currcon]];

if (info == fb_info) {
    struct display *p = &fb_display[ops->currcon];

    if (rotate < 4)
        p->con_rotate = rotate;
    else
        p->con_rotate = 0;

    fbcon_modechanged(info);
}
```

从这个结构体“registered_fb”里面得到了我们的“fb_info”结构。“fb_info”结构里有显存。
会帮我们操作，显示文字时得到文字的字模，然后在 LCD 的显存里面描出这个文字。

修改了“/etc/inittab”后，再用新内核“uImage”重启开发板：

```
Enter your selection: q
OpenJTAG> nfs 30000000 192.168.1.5:/work/nfs_root/uImage_nolcd
dm9000 i/o: 0x20000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 08:00:3e:26:0a:5b
could not establish link
File transfer via NFS from server 192.168.1.5; our IP address is 192.168.1.17
Filename '/work/nfs_root/uImage_nolcd'.
Load address: 0x30000000
Loading: ################################################################
         ################################################################
         ################################################################
         ################################################################
         ################################################################
         ##################################
done
Bytes transferred = 1842392 (1c1cd8 hex)
OpenJTAG> bootm 30000000
## Booting image at 30000000 ...
   Image Name:   Linux-2.6.22.6
   Created:      2011-12-22   1:14:43 UTC
```

```
# mount -t nfs -o nolock,vers=2 192.168.1.5:/work/nfs_root/first_fs /mnt
# cd /mnt
# insmod cfbcopyarea.ko
# insmod cfbfillrect.ko
insmod cfbimgblt.ko
insmod lcd.ko
insmod buttons.ko
# insmod cfbimgblt.ko
# insmod lcd.ko
Console: switching to colour frame buffer device 30x40
# insmod buttons.ko
input: Unspecified device as /class/input/input1
#
#
```

一开始装载驱动后，LCD 上就显示了下一行：

```
Please press Enter to activate this console.
```

提示要“回车”激活控制台。
这时通过开发板上的“按键---回车”就可以激活这个控制台。这时出现下面一行：

```
starting pid 767, tty '/dev/s3c2410_serial0': '/bin/sh'
```

PID 不是上面的，而是下面的：

```
767 0            3096 S    -sh
768 0            3096 S    -sh
```

这样就激活这个 shell 。

```
# ls -l /proc/768/fd
lrwx------    1 0        0               64 Jan  1 00:02 0 -> /dev/tty1
lrwx------    1 0        0               64 Jan  1 00:02 1 -> /dev/tty1
lrwx------    1 0        0               64 Jan  1 00:02 10 -> /dev/tty
lrwx------    1 0        0               64 Jan  1 00:02 2 -> /dev/tty1
#
```

这个“tty1”就是对应之前的“buttons.ko”：上面有加载这个 按键 驱动。

```
# insmod buttons.ko
input: Unspecified device as /class/input/input1
#
```

接着用 buttons 按键按 l+s+回车时，就能在 LCD 屏上显示目录了。

```
insmod cfbcopyarea.ko
insmod cfbfillrect.ko
insmod cfbimgblt.ko
insmod lcd.ko
insmod buttons.ko
```

内核提供的 LCD 驱动“s3c2410fb.c”，是用了“总线+设备+驱动”模型：

```
static struct platform_driver s3c2410fb_driver = {
    .probe      = s3c2410fb_probe,
    .remove     = s3c2410fb_remove,
    .suspend    = s3c2410fb_suspend,
    .resume     = s3c2410fb_resume,
    .driver     = {
        .name   = "s3c2410-lcd",
        .owner  = THIS_MODULE,
    },
};

int __devinit s3c2410fb_init(void)
{
    return platform_driver_register(&s3c2410fb_driver);
}

static void __exit s3c2410fb_cleanup(void)
{
    platform_driver_unregister(&s3c2410fb_driver);
}
```

```
int __devinit s3c2410fb_init(void)
  -->platform_driver_register(&s3c2410fb_driver);
```

注册了一个“s3c2410fb_driver”平台驱动。

.name = "s3c2410-lcd", 当内核中有同名的设备时，就调用：.probe 函数。

```
.probe = s3c2410fb_probe,
```

```
int __init s3c2410fb_probe(struct platform_device *pdev)
```

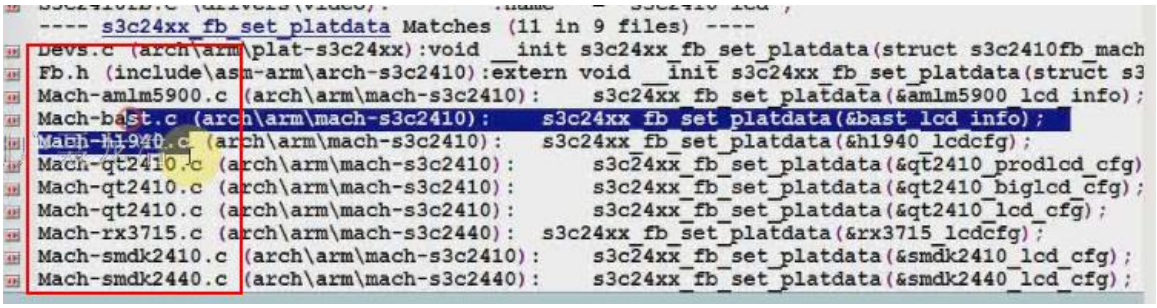
根据平台设备的一些信息（看形参,是指这个平台设备的一些信息）。来设备这个 LCD 控制器。拆分成两部分，一部分是比较稳定的代码（软件部分），另一部分是“硬件相关的代码”。

内核这个 LCD 的硬件相差的代码在：linux/arch/arm/plat-s3c24xx/devs.c

这里面有一个“平台设备”：

```
struct platform_device s3c_device_lcd = {
.name = "s3c2410-lcd",
.id = -1,
.num_resources = ARRAY_SIZE(s3c_lcd_resource),
.resource = s3c_lcd_resource,
.dev = {
.dma_mask = &s3c_device_lcd_dmamask,
.coherent_dma_mask = 0xffffffffUL
}
};
```

它们调用某个函数，设置平台设备里面的“私有数据”：s3c24xx_fb_set_platdata()。
查看这个函数“__init s3c24xx_fb_set_platdata(struct s3c2410fb_mach_info *pd)”
设置的地方：
搜索内核：



每种开发板都有。如“linux/arch/arm/mach-s3c2440/mach-smdk2440.c”中：

```
static void __init smdk2440_machine_init(void)
{
    s3c24xx_fb_set_platdata(&smdk2440_lcd_cfg);

    platform_add_devices(smdk2440_devices, ARRAY_SIZE(smdk2440_devices));
    smdk_machine_init();
}
```

如上面“s3c24xx_fb_set_platdata(&smdk2440_lcd_cfg);”这个平台私有数据里有什么东西？
有“smdk2440_lcd_cfg”相差寄存器的设置：

```
/* 320x240 */
static struct s3c2410fb_mach_info smdk2440_lcd_cfg_320x240 __initdata = {
    .regs   = {
        .lcdcon1 =  S3C2410_LCDCON1_TFT16BPP | \
                S3C2410_LCDCON1_TFT | \
                S3C2410_LCDCON1_CLKVAL(0x04),

        .lcdcon2 =  S3C2410_LCDCON2_VBPD(1) | \
                S3C2410_LCDCON2_LINEVAL(239) | \
                S3C2410_LCDCON2_VFPD(5) | \
                S3C2410_LCDCON2_VSPW(1),

        .lcdcon3 =  S3C2410_LCDCON3_HBPD(36) | \
                S3C2410_LCDCON3_HOZVAL(319) | \
                S3C2410_LCDCON3_HFPD(19),

        .lcdcon4 =  S3C2410_LCDCON4_MVAL(13) | \
                S3C2410_LCDCON4_HSPW(5),

        .lcdcon5 =  S3C2410_LCDCON5_FRM565 |
                S3C2410_LCDCON5_INVVLINE |
```

```
                S3C2410_LCDCON5_INVVFRAME |
S3C2410_LCDCON5_INVVDEN |
                S3C2410_LCDCON5_PWREN |
                S3C2410_LCDCON5_HWSWP,
    },

    .gpccon      =  0xaaaa56aa,
    .gpccon_mask =  0xffffffff,
    .gpcup       =  0xffffffff,
    .gpcup_mask  =  0xffffffff,

    .gpdcon      =  0xaaaaaaaa,
    .gpdcon_mask =  0xffffffff,
    .gpdup       =  0xffffffff,
    .gpdup_mask  =  0xffffffff,

    .fixed_syncs =  1,
    .type        =  S3C2410_LCDCON1_TFT,
    .width       =  320,
    .height      =  240,

    .xres   = {
.min                = 320,
.max                = 320,
.defval = 320,
},

    .yres   = {
        .max    =   240,
        .min    =   240,
        .defval =   240,
    },

.bpp                                = {
.min                = 16,
.max                = 16,
.defval = 16,
},
};
```

想用内核支持自己的 LCD 时，就需要上面这个结构体即可。