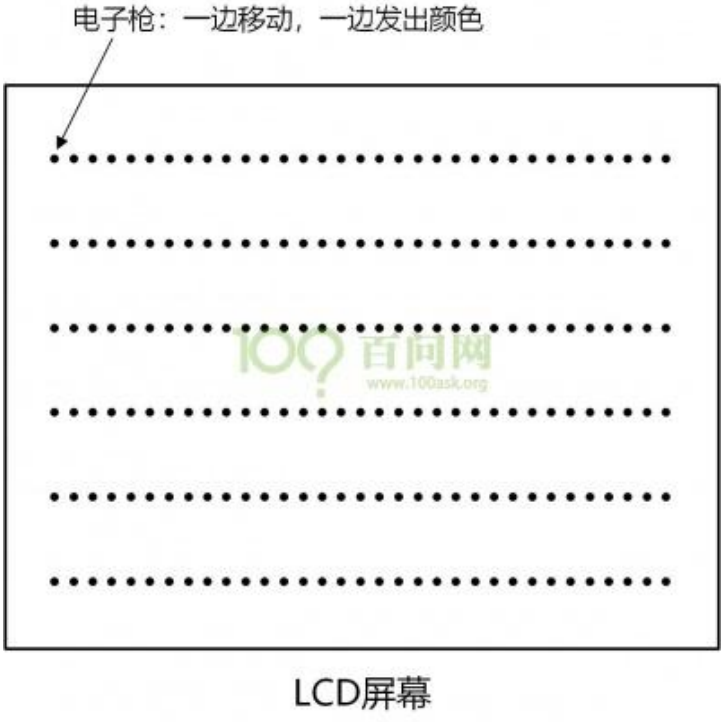
第017课 LCD

来自百问网嵌入式Linux wiki

目录

第001节_LCD硬件原理

先简单介绍下LCD的操作原理。 如下图的LCD示意图，里面的每个点就是一个像素点。



想象有一个电子枪，一边移动，一边发出各种颜色的光。这里有很多细节问题，我们一个一个的梳理。

- 1. 电子枪是如何移动的？

答：有一条CLK时钟线与LCD相连，每发出一次CLK(高低电平)，电子枪就移动一个像素。

- 2. 颜色如何确定？

答：由连接LCD的三组线：R(Red)、G(Green)、B(Blue)确定。

- 3. 电子枪如何得知应跳到下一行？

答：有一条HSYNC信号线与LCD相连，每发出一次脉冲(高低电平)，电子枪就跳到下一行。

- 4. 电子枪如何得知应跳到原点？

答：有一条VSYNC信号线与LCD相连，每发出一次脉冲(高低电平)，电子枪就跳到原点。

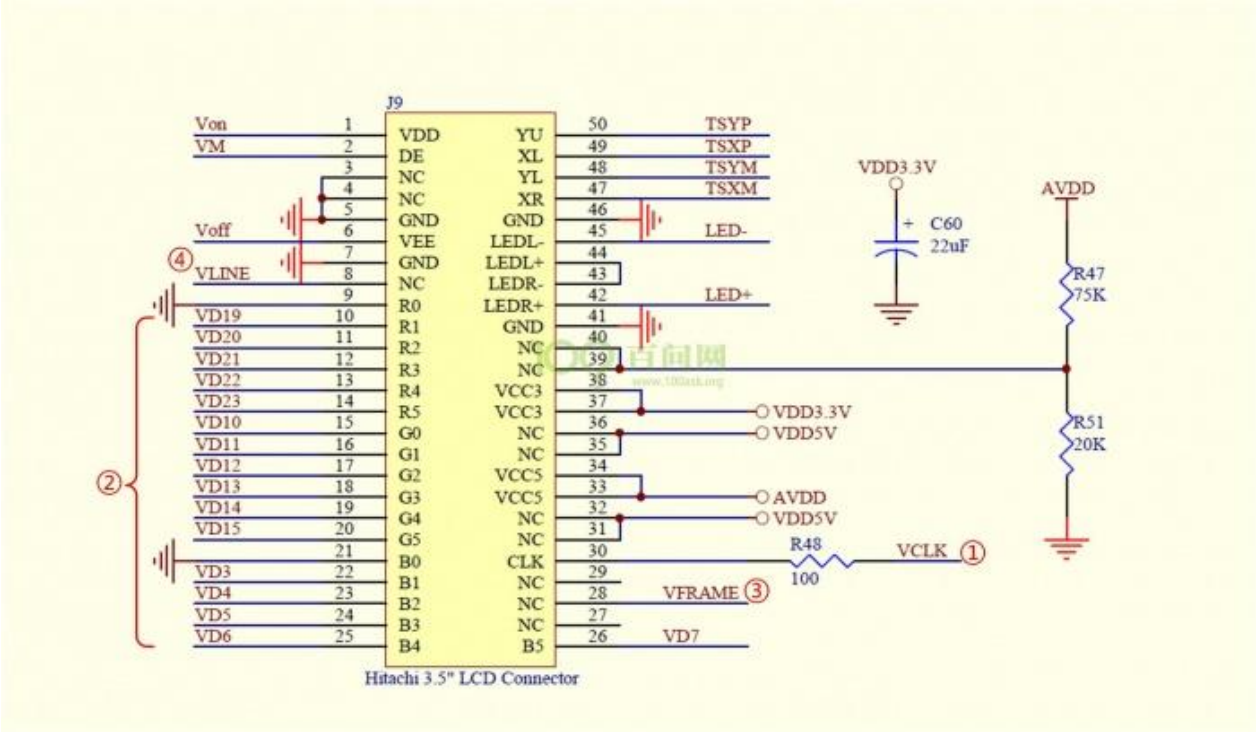
- 5. RGB线上的数据从何而来？

答：内存里面划分一块显存(FrameBuffer)，里面存放了要显示的数据，LCD控制器从里面将数据读出来，通过RGB三组线传给电子枪，电子枪再依次打到显示屏上。

- 6. 前面的信号由谁发给LCD？

答：有S3C2440里面的LCD控制器来控制发出信号。

通过JZ2440原理图对上面进行验证，下图的LCD控制器接口图。



①是时钟信号，每来一个CLK，电子枪就移动一个像素；

②是用来传输颜色数据；

③是垂直方向同步信号，FRAME(帧)；

④是水平方向同步信号，LINE(行)；

再来看看LCD的芯片手册。

Pin No.	Symbol	I/O	Function	Remark
1	V_{LED-}	P	Power for LED backlight cathode	
2	V_{LED+}	P	Power for LED backlight anode	
3	GND	P	Power ground	

3	GND	P	Power ground	
4	V_{DD}	P	Power voltage	
5	R0	I	Red data (LSB)	
6	R1	I	Red data	
7	R2	I	Red data	
8	R3	I	Red data	
9	R4	I	Red data	
10	R5	I	Red data	
11	R6	I	Red data	
12	R7	I	Red data (MSB)	
13	G0	I	Green data (LSB)	
14	G1	I	Green data	
15	G2	I	Green data	
16	G3	I	Green data	
17	G4	I	Green data	
18	G5	I	Green data	
19	G6	I	Green data	
20	G7	I	Green data (MSB)	
21	B0	I	Blue data (LSB)	
22	B1	I	Blue data	
23	B2	I	Blue data	
24	B3	I	Blue data	
25	B4	I	Blue data	
26	B5	I	Blue data	
27	B6	I	Blue data	
28	B7	I	Blue data (MSB)	

先是VLED+、VLED-背光灯电源。VDD、VDD是LCD电源。

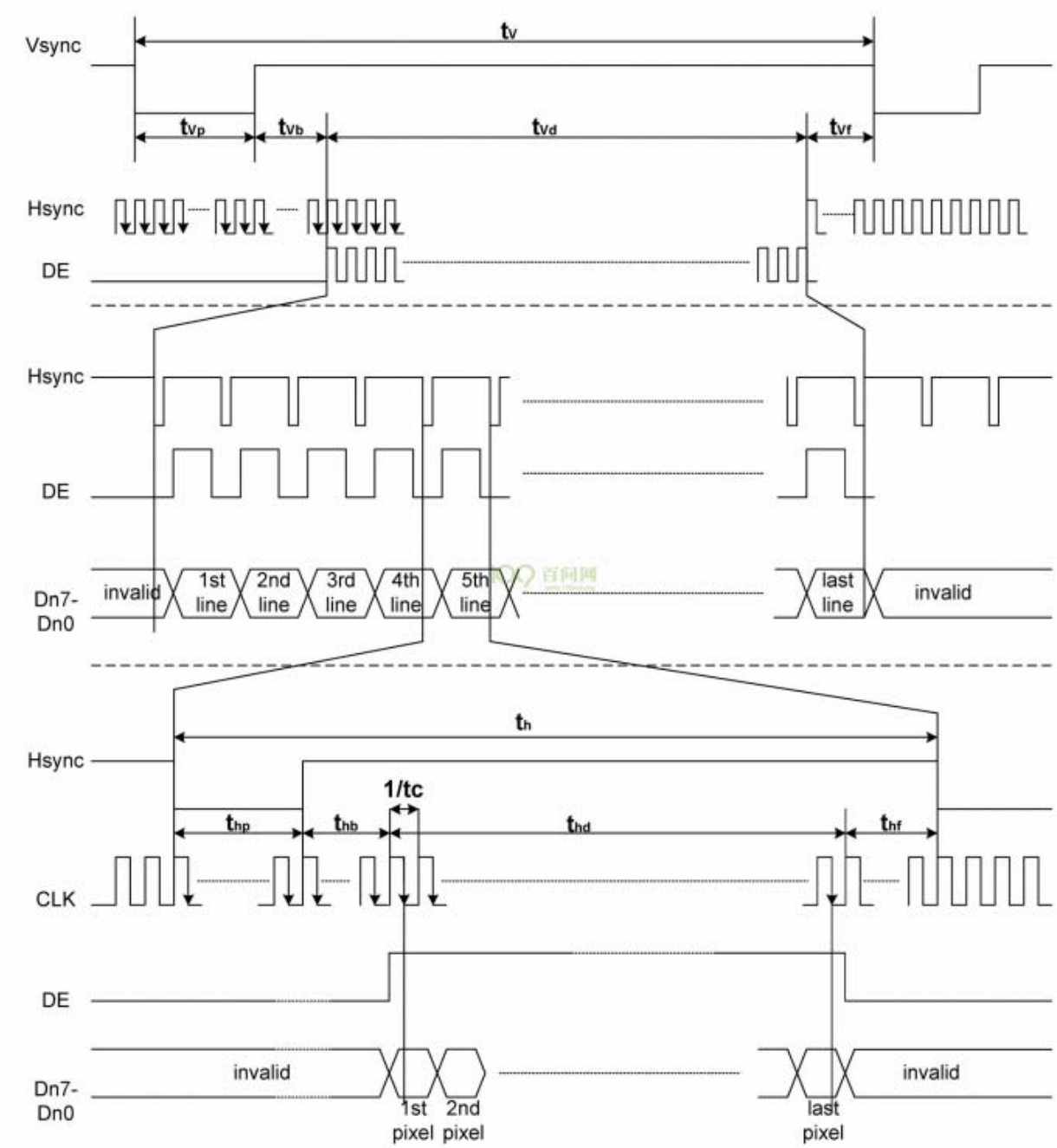
R0-R7、G0-G7、B0-B7是红绿蓝颜色信号。

PCLK是像素时钟信号。DISP是像素开关。

HSYNC、VSYNC分别是水平方向、垂直方向信号。

DE数据使能。X1、Y1、X2、Y2是触摸屏信号。

可以看出LCD有很多信号，这些信号要根据时序图传输才能正确显示。参考JZ2440_4.3寸LCD手册_AT043TN24的时序如下：



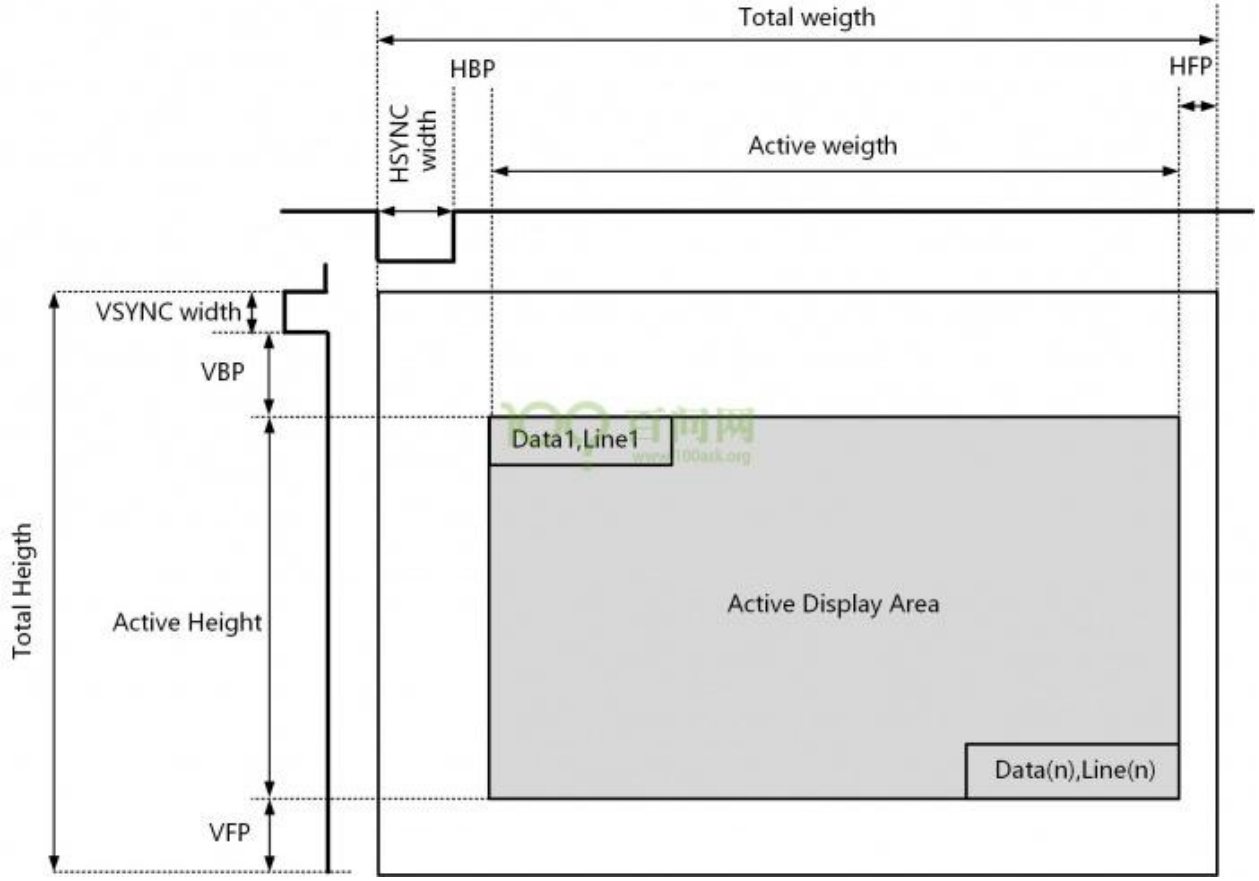
从最小的像素开始分析，电子枪每次在CLK下降沿(本开发板是下降沿)从数据线Dn0-Dn7上得到数据，发射到显示屏上，然后移动到下一个位置。Dn0-Dn7上的数据来源就是前面介绍的FrameBuffer。就这样从一行的最左边，一直移动到一行的最右边，完成了一行的显示，假设为x。

当打完一行的最后一个数据后，就会收到Hsync行同步信号，根据时序图，一个Hsync周期可以大致分为五部分组成：thp、thb、1/tc、thd、thf。thp称为脉冲宽度，这个时间不能太短，太短电子枪可能识别不到。电子枪正确识别到thp后，会从最右端移动最左端，这个移动的时间就是thb，称之为移动时间。thf表示显示完最右像素，再过多久Hsync才来。

同理，当电子枪一行一行的从上面移动到最下面时，Vsync垂直同步信号就让电子枪移动回最上边。Vsync中的tvp是脉冲宽度，tvb是移动时间，tvf表示显示完最下一行像素，再过多久Vsync才来。 假设一共有y行，则LCD的分辨率就是x*y。

关于显示原理，可以参考这篇博客：http://www.cnblogs.com/shangdawei/p/4760933.html

里面有一个LCD显示配置示意图如下：



当发出一个HSYNC信号后，电子枪就会从最右边花费HBP时长移动到最左边，等到了最右边后，等待HFP时长HSYNC信号才回来。因此，HBP和HFP分别决定了左边和右边的黑框。

同理，当发出一个VSYNC信号后，电子枪就会从最下边花费VBP时长移动到最上边，等到了最下边后，等待VFP时长VSYNC信号才回来。因此，VBP和VFP分别决定了上边和下边的黑框。 中间灰色区域才是有效显示区域。

再来解决最后一个问题：每个像素再FrameBuffer中，占据多少位BPP(Bits Per Pixels)？ 前面的LCD引脚功能图里，R0-R7、G0-G7、B0-B7，每个像素是占据3*8=24位的，即硬件上LCD的BPP是确定的。

虽然LCD上的引脚是固定的，但我们使用的时候，可以根据实际情况进行取舍，比如我们的JZ2440使用的是16BPP，因此LCD只需要R0-R4、G0-G5、B0-B4与SOC相连，5+6+6=16BPP，每个像素就只占据16位数据。

我们写程序的思路如下：

1. 查看LCD芯片手册，查看相关的时间参数、分辨率、引脚极性；

2. 根据以上信息设置LCD控制器寄存器，让其发出正确信号；

3. 在内存里面分配一个FrameBuffer，在里面用若干位表示一个像素，再把首地址告诉LCD控制器；

之后LCD控制器就能周而复始取出FrameBuffer里面的像素数据，配合其它控制信号，发送给电子枪，电子枪再让在LCD上显示出来。以后我们想显示图像，只需要编写程序向FrameBuffer填入相应数据即可，硬件会自动的完成显示操作。

第002节_S3C2440_LCD控制器

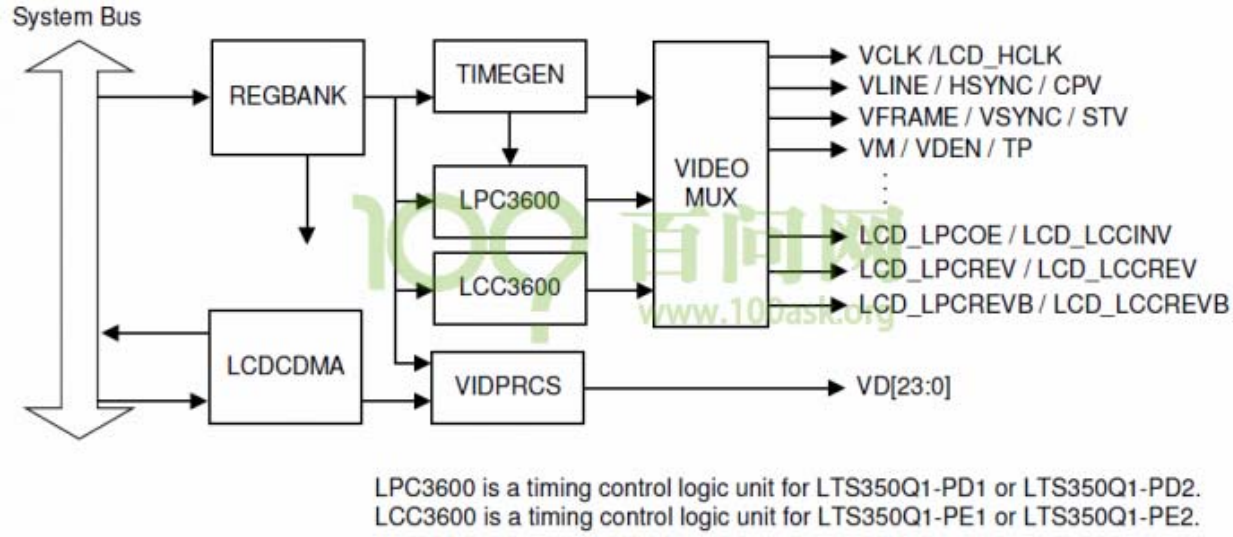
LCD控制器主要功能和需要的设置：

- 1. 取：从内存(FrameBuffer)取出某个像素的数据；之后需要把FrameBuffer地址、BPP、分辨率告诉LCD控制器；

- 2. 发：配合其它信号把FrameBuffer数据发给LCD；需要设置LCD控制器时序、设置引脚极性；

这里主要的难点就是如何配合其它信号，需要我们阅读LCD芯片手册，知道其时序要求，然后设置相应的LCD控制器。

先看下S3C2440芯片手册上的LCD控制器框图：

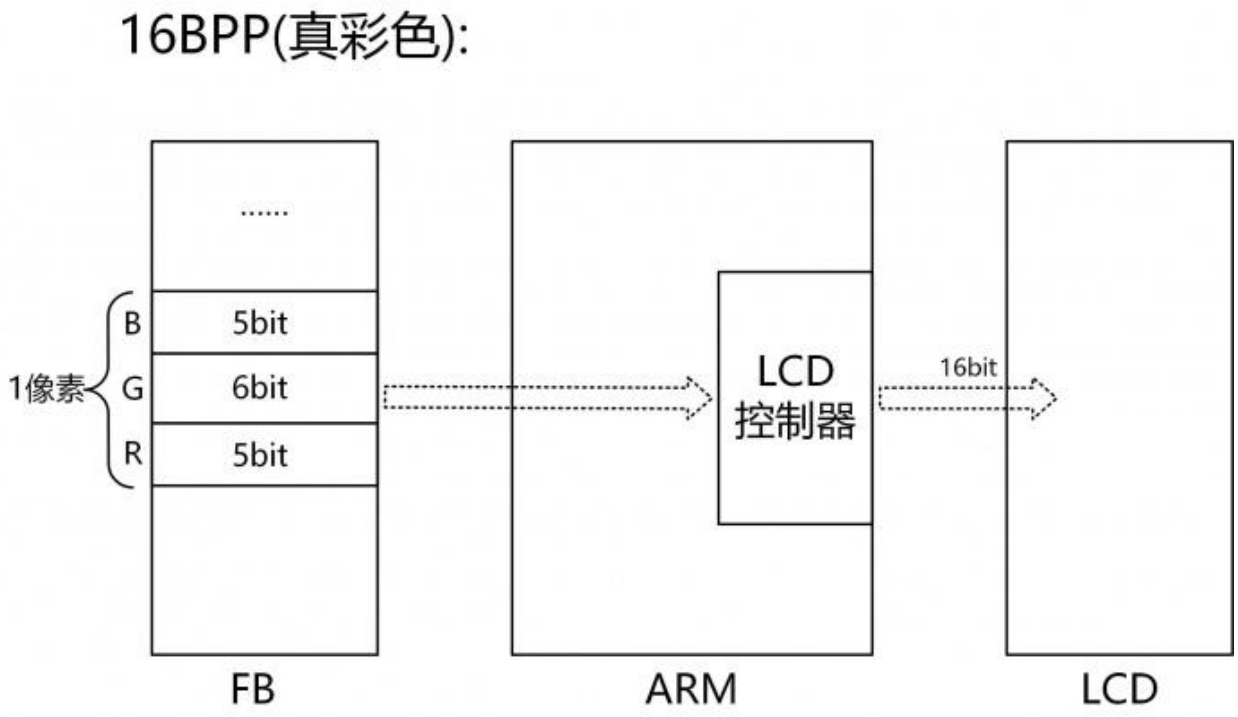


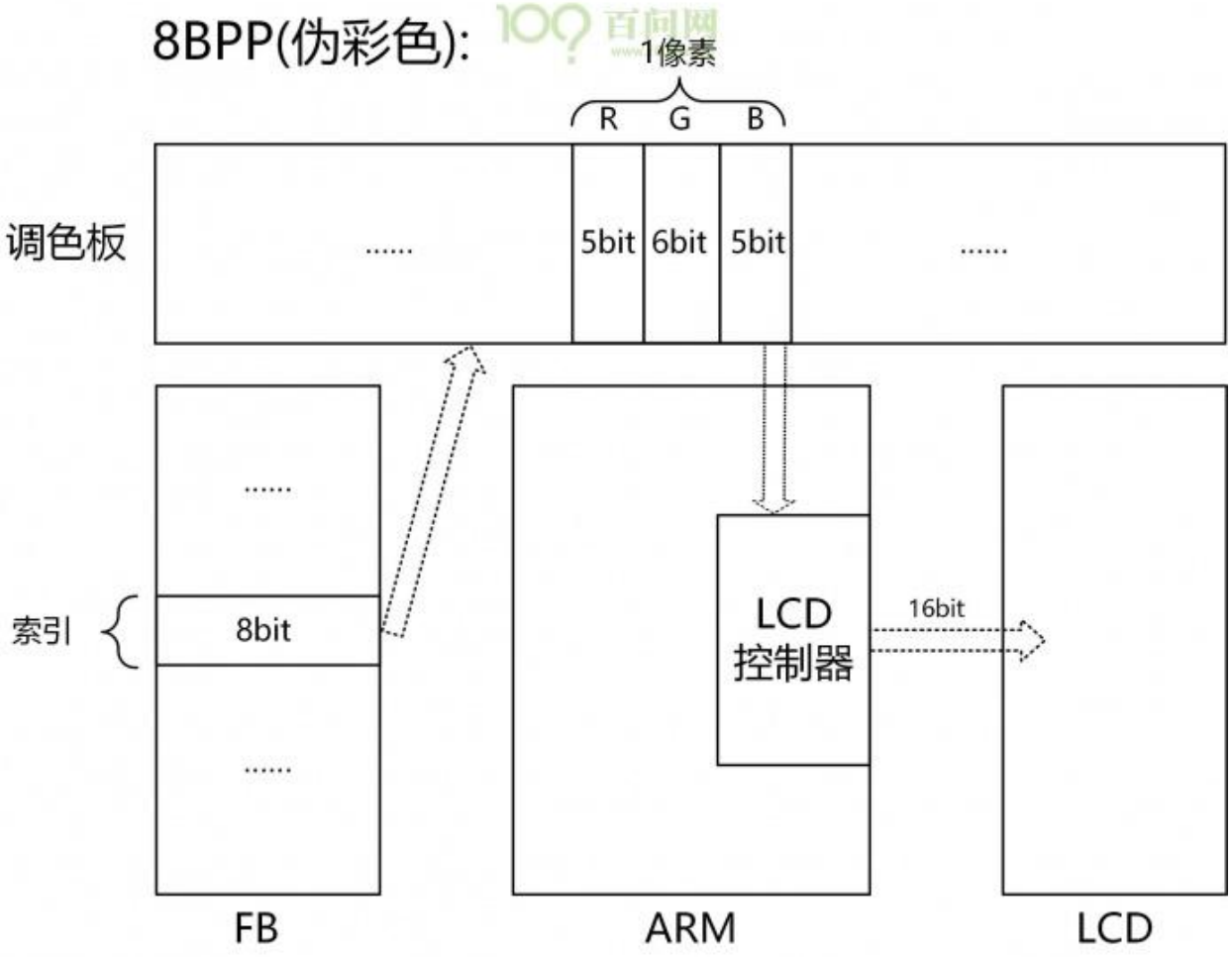
通过设置REGBANK(寄存器组)，LCDCDMA会自动(无需CPU参与)把内存上FrameBuffer里的数据，通过VIDPRCS发送到引脚VD[23:0]上，再配合VIDEOMUX引脚的控制信号，正确的显示出来。

S3C2440芯片手册介绍了LCD控制器支持TFT和STN两种LCD，我们常用的都是TFT材质的，因此主要看TFT相关的部分。

调色板的概念：

画油画的时候，通常先在调色板里配好想要的颜色，再用画笔沾到画布上作画。LCD控制器里也借用了这个概念，从FrameBuffer获得数据，这个数据作为索引从调色板获得对应数据，再发给电子枪显示出来。





如图，假如是16BPP的数据，LCD控制器从FB取出16bit数据，显示到LCD上。

当如果想节约内存，对颜色要求也没那么高，就可以采用调色板的方式，调色板里存放了256个16bit的数据，FB只存放每个像素的索引，根据索引去调色板找到对应的数据传给LCD控制器，再通过电子枪显示出来。

假设现在想要LCD只显示一种颜色怎么办？

如果是16BPP/24BPP需要修改FB里面的数据，填充同一个值。

如果是8BPP可以修改FB为同一种颜色，也可以设置调色板为同一种颜色，对于S3C22440有个临时调色板的特性，一旦使能了临时调色板，不管FB里面是什么数据，都只调用临时调色板的数据。

第003节_编程_框架与准备

本节主要有两个目的：

a. 讲解后续程序的框架；

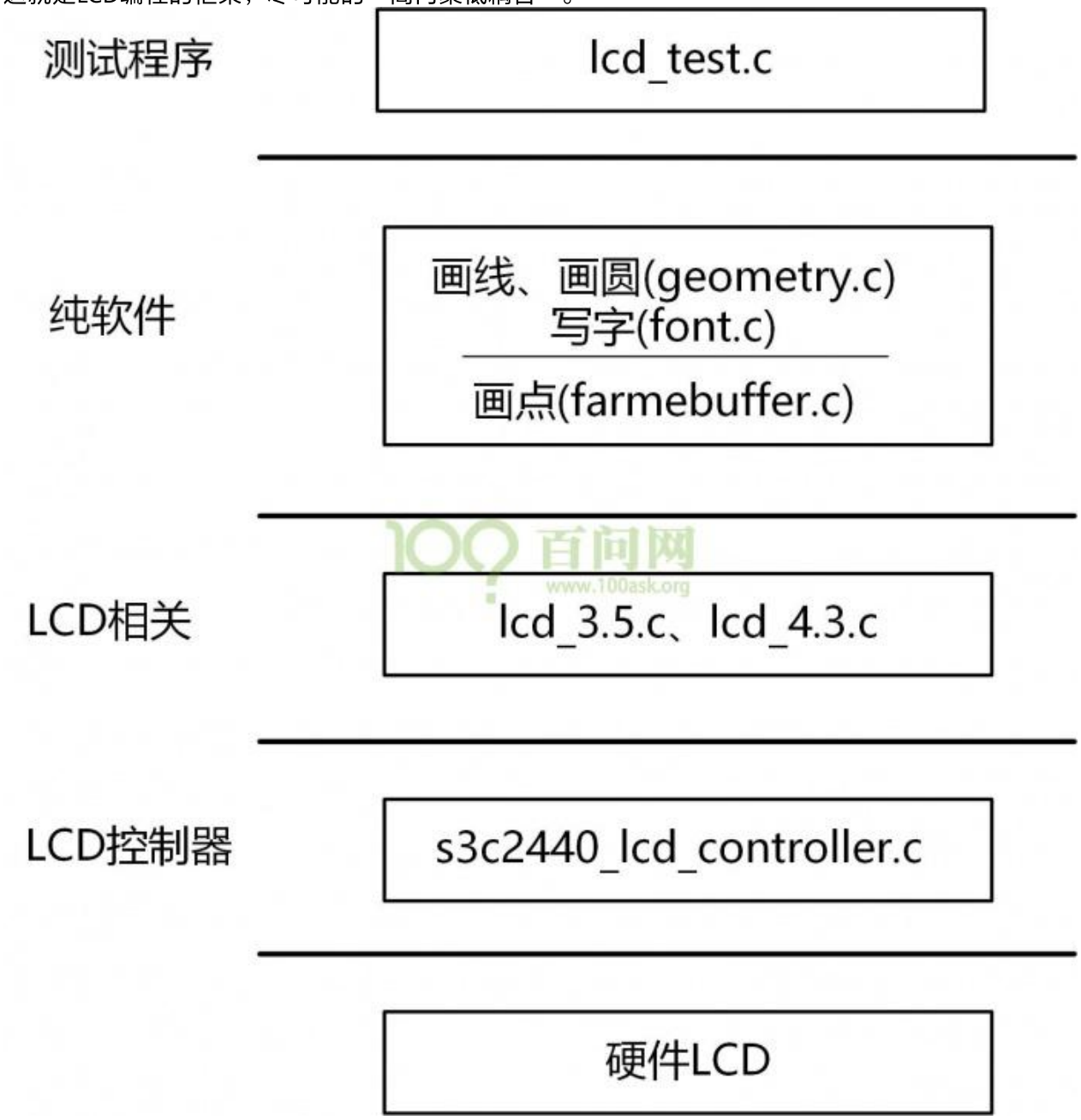
b. 准备一个支持NAND、NOR启动的程序；

我们的目的是在LCD显示屏上画线、画圆(geomentry.c)和写字(font.c)其核心是画点(farmebuffer.c)，这些都属于纯软件。此外还需要一个lcd_test.c测试程序提供操作菜单，调用画线、画圆和写字操作。

往下操作的是LCD相关的内容，不同的LCD，其配置的参数也会不一样，通过lcd_3.5.c或lcd_4.3.c来设置。

根据LCD的特性，来设置LCD控制器，对于我们开发板，就是s3c2440_lcd_controller.c，假如希望在其它开发板上也实现LCD显示，只需添加相应的代码文件即可。

这就是LCD编程的框架，尽可能的“高内聚低耦合”。



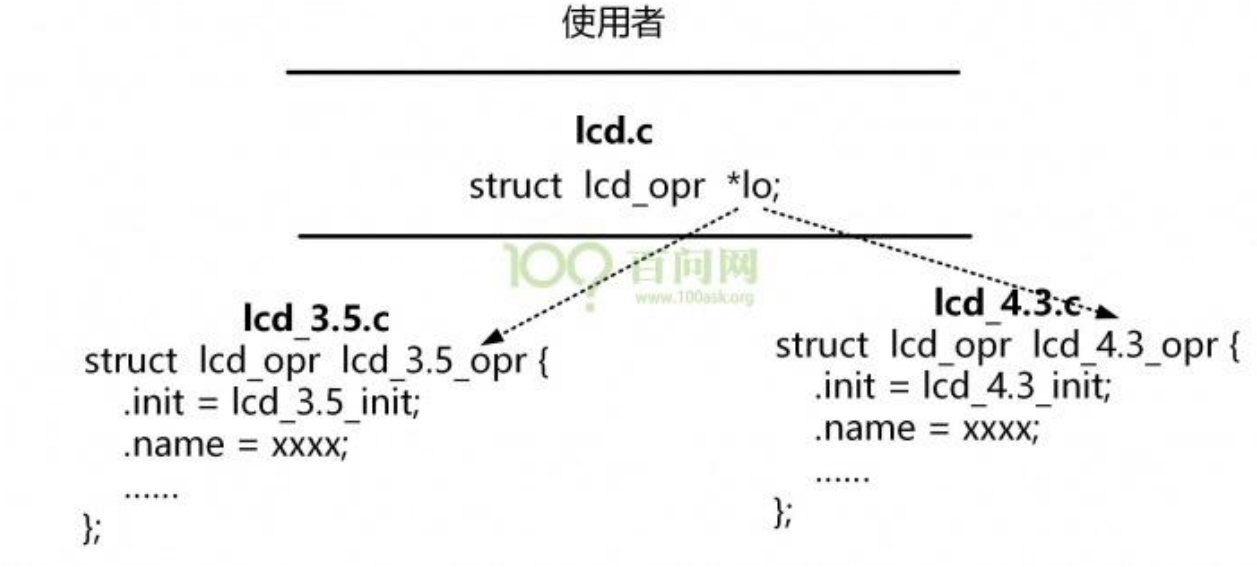
为了让程序更加好扩展，下面介绍“面向对象编程”的概念。

假如我们写好程序后，有两款尺寸大小的lcd，如何快速的在两个lcd上切换？

首先我们抽象出lcd_3.5.c和lcd_4.3.c的共同点，比如都有初始化函数init(),我们可以新建一个lcd.c，然后定义一个结构体：

```
struct lcd_opr{
    void (*init)(void);
};
```

用户不接触lcd_3.5.c和lcd_4.3.c，只需要在lcd.c里通过指针访问对应的结构体的函数，也就调用了不同init()。



前面我们的程序大小都没超过4K，因此无论Nor/Nand启动，都是正常的，现在的LCD相关代码比较大，超过4K，因此需要修改启动部分的代码。

目前还未讲解nand flash，因此直接将19课准备的nand_flash程序部分复制到当前代码里即可，关于这部分可以参考nand flash讲解部分。

第004节_编程_抽象出重要结构体

开始正式编写程序，根据前面的框架，新建如下文件：

font.c、framebuffer.c、geometry.c、lcd.c、lcd_4.3.c、lcd_controller.c、s3c2440_lcd_controller.c、lcd_test.c

首先编写lcd_controller.c，它向上要接收不同LCD的参数，向下要使用这些参数设置对应的LCD控制器。

前面我们列举了LCD的参数，例如引脚的极性、时序、数据的格式bpp、分辨率等，使用面向对象的思维方式，将这些封装成结构体放在lcd.h中：

```
enum {
    NORMAL = 0,
    INVERT = 1,
};

/* NORMAL : 正常极性
 * INVERT : 反转极性
 */
typedef struct pins_polarity {
    int vclk;  /* normal: 在下降沿获取数据 */
    int rgb;   /* normal: 高电平表示1 */
    int hsync; /* normal: 高脉冲 */
    int vsync; /* normal: 高脉冲 */
}pins_polarity, *p_pins_polarity;

typedef struct time_sequence {
    /* 垂直方向 */
    int tvp; /* vysnc脉冲宽度 */
    int tvb; /* 上边黑框, Vertical Back porch */
    int tvf; /* 下边黑框, Vertical Front porch */

    /* 水平方向 */
    int thp; /* hsync脉冲宽度 */
    int thb; /* 左边黑框, Horizontal Back porch */
    int thf; /* 右边黑框, Horizontal Front porch */

    int vclk;
}time_sequence, *p_time_sequence;


typedef struct lcd_params {
    /* 引脚极性 */
```

```
    pins_polarity pins_pol;

    /* 时序 */
    time_sequence time_seq;

    /* 分辨率, bpp */
    int xres;
    int yres;
    int bpp;

    /* framebuffer的地址 */
    unsigned int fb_base;
}lcd_params, *p_lcd_params;
```

以后就使用lcd_params结构体来表示lcd参数。

对于有多个lcd的情况，再定义个一个结构体，包含指针初始化函数和使能函数，放在lcd_controller.h里面：

```
typedef struct lcd_controller {
    void (*init)(p_lcd_params plcdparams);
    void (*enable)(void);
    void (*disable)(void);
}lcd_controller, *p_lcd_controller;
```

最后在lcd_controller.c里传入lcd参数，再通过指针函数初始化对应的lcd控制器：

```
void lcd_controller_init(p_lcd_params plcdparams)
{
    /* 调用2440的LCD控制器的初始化函数 */
    lcd_controller.init(plcdparams);
}
```

在s3c2440_lcd_controller.c还需构造一个当前soc的lcd控制器结构体：

```
struct lcd_controller s3c2440_lcd_controller = {
    .init    = xxx,
    .enalbe  = xxx,
    .disable = xxx,
};
```

第005节_编程_LCD控制器

继续上一节的代码，修改s3c2440_lcd_controller.c：

```
struct lcd_controller s3c2440_lcd_controller = {
    .init    = s3c2440_lcd_controller_init,
    .enalbe  = s3c2440_lcd_controller_enalbe,
    .disable = xs3c2440_lcd_controller_disable,
};
```

然后对每个函数进行功能实现，首先是s3c2440_lcd_controller_init，依次设置LCD控制器寄存器，先是**LCD寄存器1**：

LCD Control 1 Register

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

LCDCON1	Bit	Description	Initial State
LINECNT (read only)	[27:18]	Provide the status of the line counter. Down count from LINEVAL to 0	0000000000
CLKVAL	[17:8]	Determine the rates of VCLK and CLKVAL[9:0]. **STN**: VCLK = HCLK / (CLKVAL x 2) (CLKVAL ≥2) **TFT**: VCLK = HCLK / [(CLKVAL+1) x 2] (CLKVAL ≥ 0)	0000000000
MMODE	[7]	Determine the toggle rate of the VM. 0 = Each Frame 1 = The rate defined by the MVAL	0
PNRMODE	[6:5]	Select the display mode. 00 = 4-bit dual scan display mode (STN) 01 = 4-bit single scan display mode (STN)	00

[27:18]为只读数据位，不需要设置；

[17:8]用于设置CLKVAL(像素时钟频率)，我们使用的是TFT屏，因此采用的公式是VCLK = HCLK / [(CLKVAL+1) x 2]，其中HCLK为100M。LCD手册里面Clock cycle的要求范围为5-12MHz即可，即假设VCLK=9，根据公式9=100/[(CLKVAL+1)x2],算出CLKVAL≈4.5=5。VCLK为plcdparams->time_seq.vclk，则clkval = HCLK/plcdparams->time_seq.vclk/2-1+0.5;

[7]不用管，默认即可；

[6:5]TFT lcd配置为0b11；

[4:1]设置bpp模式，根据传入的plcdparams->bpp配置为相应的数值；

[0]LCD输出使能，先暂时关闭不输出；

寄存器2：

LCD Control 2 Register

Register	Address	R/W	Description	Reset Value
LCDCON2	0X4D000004	R/W	LCD control 2 register	0x00000000

LCDCON2	Bit	Description	Initial State
VBPD	[31:24]	**TFT**: Vertical back porch is the number of inactive lines at the start of a frame, after vertical synchronization period. **STN**: These bits should be set to zero on STN LCD.	0x00

对比2440LCD部分时序图和LCD时序图，得出两者之间关系，以后就可通过plcdparams传参数进来设置相关寄存器。

[31:24] : VBPD = tvb - 1

[23:14] : LINEVAL = line - 1

[13:6] : VFPD = tvf - 1

[5:0] : VSPW = tvp - 1

寄存器3:

LCD Control 3 Register

Register	Address	R/W	Description	Reset Value

[25:19] : HBPD = thb - 1

[18:8] : HOZVAL = 列 - 1

[7:0] ：HFPD = thf - 1

寄存器4：

LCD Control 4 Register

Register	Address	R/W	Description	Reset Value
LCDCON4	0X4D00000C	R/W	LCD control 4 register	0x00000000

LCDCON4	Bit	Description	Initial state
MVAL	[15:8]	**STN**: These bit define the rate at which the VM signal will toggle if the MMODE bit is set to logic '1'.	0X00
HSPW(TFT)	[7:0]	**TFT**: Horizontal sync pulse width determines the HSYNC pulse's high level width by counting the number of the VCLK.	0X00
WLH(STN)		**STN**: WLH[1:0] bits determine the VLINE pulse's high level width by counting the number of the HCLK. WLH[7:2] are reserved. 00 = 16 HCLK, 01 = 32 HCLK, 10 = 48 HCLK, 11 = 64 HCLK	

[7:0] ：HSPW = thp - 1

寄存器5：

LCD Control 5 Register

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

LCDCON5	Bit	Description	Initial state
Reserved	[31:17]	This bit is reserved and the value should be '0'.	0
VSTATUS	[16:15]	**TFT**: Vertical Status (read only). 00 = VSYNC 01 = BACK Porch 10 = ACTIVE 11 = FRONT Porch	00
HSTATUS	[14:13]	**TFT**: Horizontal Status (read only). 00 = HSYNC 01 = BACK Porch 10 = ACTIVE 11 = FRONT Porch	00

用来设置引

脚极性, 设置16bpp, 设置内存中象素存放的格式

[12] : BPP24BL

[11] : FRM565, 1-565

[10] : INVVCLK, 0 = The video data is fetched at VCLK falling edge

[9] : HSYNC是否反转

[8] : VSYNC是否反转

[7] : INVVD, rgb是否反转

[6] : INVVDEN

[5] : INVPWREN

[4] : INVLEND

[3] ：PWREN, LCD_PWREN output signal enable/disable

[2] ：ENLEND

[1] ：BSWP

[0] ：HWSWP

然后再设置framebuffer地址，先是 **LCDSADDR1**：

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

[29:21] : LCDBANK, A[30:22] of fb

[20:0] ：LCDBASEU, A[21:1] of fb

即用[29:0]表示起始地址的[30:1]。

LCDSADDR2：

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

[20:0] : LCDBASEL, A[21:1] of end addr

即framebuffer的结束地址。

最后还要设置相关引脚，包括背光控制引脚、LCD专用引脚、电源控制引脚：

```
void jz2440_lcd_pin_init(void)
{
    /* 初始化引脚 : 背光引脚 */
    GPBCON &= ~0x3;
    GPBCON |= 0x01;

    /* LCD专用引脚 */
```

```
    GPCCON = 0xaaaaaaaaa;
    GPDCON = 0xaaaaaaaaa;

    /* PWREN */
    GPGCON |= (3<<8);
}
```

LCD所有寄存器的具体设置如下：

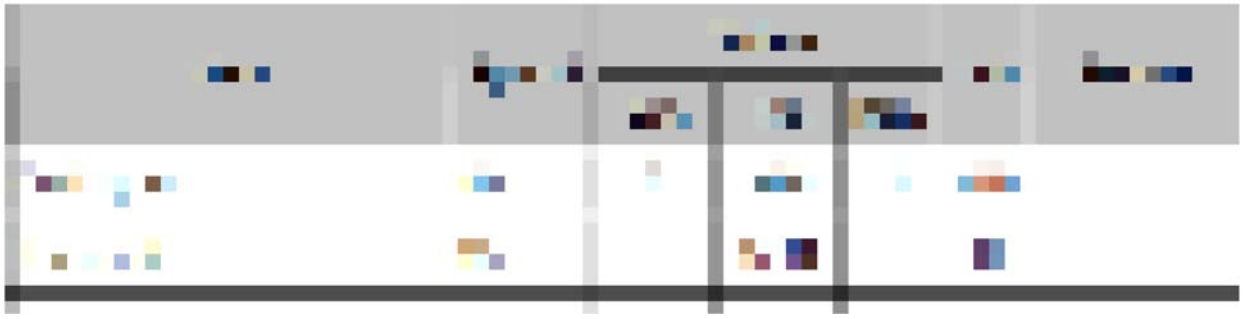
```
#define HCLK 100

void jz2440_lcd_pin_init(void)
{
    /* 初始化引脚 : 背光引脚 */
    GPBCON &= ~0x3;
    GPBCON |= 0x01;

    /* LCD专用引脚 */
    GPCCON = 0xaaaaaaaaa;
    GPDCON = 0xaaaaaaaaa;

    /* PWREN */
    GPGCON |= (3<<8);
}


/* 根据传入的LCD参数设置LCD控制器 */
void s3c2440_lcd_controller_init(p_lcd_params plcdparams)
{
    int pixelplace;
    unsigned int addr;

    jz2440_lcd_pin_init();

    /* [17:8]: clkval, vclk = HCLK / [(CLKVAL+1) x 2]
     *                  9   = 100M /[(CLKVAL+1) x 2], clkval = 4.5 = 5
     *                CLKVAL = 100/vclk/2-1
     * [6:5]: 0b11, tft lcd
     * [4:1]: bpp mode
     * [0]  : LCD video output and the logic enable/disable
     */
    int clkval = (double)HCLK/plcdparams->time_seq.vclk/2-1+0.5;
    int bppmode = plcdparams->bpp == 8  ? 0xb :\
                  plcdparams->bpp == 16 ? 0xc :\
                  0xd;  /* 0xd: 24bpp */
    LCDCON1 = (clkval<<8) | (3<<5) | (bppmode<<1) ;

    /* [31:24] : VBPD    = tvb - 1
     * [23:14] : LINEVAL = line - 1
     * [13:6]  : VFPD    = tvf - 1
     * [5:0]   : VSPW    = tvp - 1
     */
    LCDCON2 =   ((plcdparams->time_seq.tvb - 1)<<24) | \
                ((plcdparams->yres - 1)<<14)         | \
                ((plcdparams->time_seq.tvf - 1)<<6)  | \
                ((plcdparams->time_seq.tvp - 1)<<0);

    /* [25:19] : HBPD    = thb - 1
     * [18:8]  : HOZVAL  = 列 - 1
     * [7:0]   : HFPD    = thf - 1
     */
    LCDCON3 =   ((plcdparams->time_seq.thb - 1)<<19) | \
                ((plcdparams->xres - 1)<<8)          | \
                ((plcdparams->time_seq.thf - 1)<<0);

    /*
     * [7:0]   : HSPW    = thp - 1
     */
    LCDCON4 =   ((plcdparams->time_seq.thp - 1)<<0);

    /* 用来设置引脚极性, 设置16bpp, 设置内存中象素存放的格式
     * [12] : BPP24BL
     * [11] : FRM565, 1-565
     * [10] : INVVCLK, 0 = The video data is fetched at VCLK falling edge
     * [9]  : HSYNC是否反转
     * [8]  : VSYNC是否反转
     * [7]  : INVVD, rgb是否反转
     * [6]  : INVVDEN
     * [5]  : INVPWREN
     * [4]  : INVLEND
```

```
     * [3]  : PWREN, LCD_PWREN output signal enable/disable
     * [2]  : ENLEND
     * [1]  : BSWP
     * [0]  : HWSWP
     */

    pixelplace = plcdparams->bpp == 24 ? (0) : |\
                 plcdparams->bpp == 16 ? (1) : |\
                 (1<<1);  /* 8bpp */
    LCDCON5 = (plcdparams->pins_pol.vclk<<10) |\
              (plcdparams->pins_pol.rgb<<7)   |\
              (plcdparams->pins_pol.hsync<<9) |\
              (plcdparams->pins_pol.vsync<<8) |\
              (plcdparams->pins_pol.de<<6)    |\
              (plcdparams->pins_pol.pwren<<5) |\
              (1<<11) | pixelplace;

    /* framebuffer地址 */
    /*
     * [29:21] : LCDBANK, A[30:22] of fb
     * [20:0]  : LCDBASEU, A[21:1] of fb
     */
    addr = plcdparams->fb_base & ~(1<<31);
    LCDSADDR1 = (addr >> 1);

    /*
     * [20:0] : LCDBASEL, A[21:1] of end addr
     */
    addr = plcdparams->fb_base + plcdparams->xres*plcdparams->yres*plcdparams->bpp/8;
    addr >>=1;
    addr &= 0x1fffff;
    LCDSADDR2 = addr;//
}

void s3c2440_lcd_controller_enalbe(void)
{
    /* 背光引脚 : GPB0 */
    GPBDAT |= (1<<0);

    /* pwren    : 给LCD提供AVDD  */
    LCDCON5 |= (1<<3);

    /* LCDCON1'BIT 0 : 设置LCD控制器是否输出信号 */
    LCDCON1 |= (1<<0);
}

void s3c2440_lcd_controller_disable(void)
{
    /* 背光引脚 : GPB0 */
    GPBDAT &= ~(1<<0);

    /* pwren    : 给LCD提供AVDD  */
    LCDCON5 &= ~(1<<3);

    /* LCDCON1'BIT 0 : 设置LCD控制器是否输出信号 */
    LCDCON1 &= ~(1<<0);
}
```

这就完成了s3c2440_lcd_controller.c的编写，后面只需要向s3c2440_lcd_controller_init()传入构造好的参数即可。

第006节_编程_LCD设置

前面编写了s3c2440_lcd_controller.c，以后我们只需往里面传入参数即可控制LCD控制器，对于我们的4.3寸LCD，配合LCD手册时序的介绍，相关的设置如下:



```
#define LCD_FB_BASE 0x33c00000

lcd_params lcd_4_3_params = {
    .name = "lcd_4.3"
    .pins_polarity = {
        .de     = NORMAL,     /* normal: 高电平时可以传输数据 */
```

```
        .pwren = NORMAL,     /* normal: 高电平有效 */
        .vclk  = NORMAL,     /* normal: 在下降沿获取数据 */
        .rgb   = NORMAL,     /* normal: 高电平表示1 */
        .hsync = INVERT,     /* normal: 高脉冲 */
        .vsync = INVERT,     /* normal: 高脉冲 */
    },
    .time_sequence = {
        /* 垂直方向 */
        .tvp=   10, /* vysnc脉冲宽度 */
        .tvb=   2,  /* 上边黑框, Vertical Back porch */
        .tvf=   2,  /* 下边黑框, Vertical Front porch */

        /* 水平方向 */
        .thp=   41, /* hsync脉冲宽度 */
        .thb=   2,  /* 左边黑框, Horizontal Back porch */
        .thf=   2,  /* 右边黑框, Horizontal Front porch */

        .vclk=  9,  /* MHz */
    },
    .xres = 480,
    .yres = 272,
    .bpp  = 16,
    .fb_base = LCD_FB_BASE,
};
```

完成了lcd控制器和参数的代码，现在还需要一个管理的中间层将两者连在一起。

我们用lcd_controller.c管理s3c2440_lcd_controller.c，它向上接受传入的LCD参数，向下传给对应的LCD控制器。lcd_controller.c管理下级的控制器的思路如下：

- a. 用数组保存下面各种lcd_controller；
- b. 提供register_lcd_controller给下面的代码设置数组；
- c. 提供select_lcd_controller(name)给上面的代码选择某个lcd_controller;

lcd_controller.c代码如下：

```
#define LCD_CONTROLLER_NUM 10

static p_lcd_controller p_array_lcd_controller[LCD_CONTROLLER_NUM];
static p_lcd_controller g_p_lcd_controller_selected;

int register_lcd_controller(p_lcd_controller plcdcon)
{
    int i;
    for (i = 0; i < LCD_CONTROLLER_NUM; i++)
    {
        if (!p_array_lcd_controller[i])
        {
            p_array_lcd_controller[i] = plcdcon;
            return i;
        }
    }
    return -1;
}

int select_lcd_controller(char *name)
{
    int i;
    for (i = 0; i < LCD_CONTROLLER_NUM; i++)
    {
        if (p_array_lcd_controller[i] && !strcmp(p_array_lcd_controller[i]->name, name))
        {
            g_p_lcd_controller_selected = p_array_lcd_controller[i];
            return i;
        }
    }
    return -1;
}


/* 向上: 接收不同LCD的参数
 * 向下: 使用这些参数设置对应的LCD控制器
 */
```

```
int lcd_controller_init(p_lcd_params plcdparams)
{
    /* 调用所选择的LCD控制器的初始化函数 */
    if (g_p_lcd_controller_selected)
    {
        g_p_lcd_controller_selected->init(plcdparams);
        return 0;
    }
    return -1;
}

void lcd_contoller_add(void)
{
    s3c2440_lcd_contoller_add();
}
```

同时，在s3c2440_lcd_controller.c里注册控制器：

```
void s3c2440_lcd_contoller_add(void)
{
    register_lcd_controller(&s3c2440_lcd_controller);
}
```

这样，s3c2440_lcd_controller.c里的register_lcd_controller()将自己放在p_array_lcd_controller[]这个数组，然后上层的lcd_controller.c调用select_lcd_controller()传入要选择的LCD控制器，然后在数组里面找到名字名字匹配的LCD控制器进行相应的初始化。

同理，也通过lcd.c去管理lcd_4.3.c,思路如下：

- a. 有一个数组存放各类lcd的参数；
- b. 有一个register_led给下面的lcd程序来设置数组；
- c. 有一个select_lcd，供上层选择某款LCD；

参考前面的lcd_controller.c编辑lcd_controller.c如下：

```
#define LCD_NUM 10

static p_lcd_params p_array_lcd[LCD_NUM];
static p_lcd_params g_p_lcd_selected;

int register_lcd(p_lcd_params plcd)
{
    int i;
    for (i = 0; i < LCD_NUM; i++)
    {
        if (!p_array_lcd[i])
        {
            p_array_lcd[i] = plcd;
            return i;
        }
    }
    return -1;
}

int select_lcd(char *name)
{
    int i;
    for (i = 0; i < LCD_NUM; i++)
    {
        if (p_array_lcd[i] && !strcmp(p_array_lcd[i]->name, name))
        {
            g_p_lcd_selected = p_array_lcd[i];
            return i;
        }
    }
    return -1;
}
```

在lcd_4.3.c里面把lcd参数注册进去：

void lcd_4_3_add(void) { register_lcd(&lcd_4_3_params); }

以后只需要在lcd.c里面选择某款lcd和某款lcd控制器即可，底层的只管添加种类即可。

在lcd.c里面添加初始化函数如下：

```
int lcd_init(void)
{
    /* 注册LCD */
    lcd_4_3_add();

    /* 注册LCD控制器 */
    lcd_contoller_add();

    /* 选择某款LCD */
    select_lcd("lcd_4.3");

    /* 选择某款LCD控制器 */
    select_lcd_controller("s3c2440");

    /* 使用LCD的参数, 初始化LCD控制器 */
    lcd_controller_init(g_p_lcd_selected);
}
```

第007节_编程_简单测试

首先向lcd_test.c里面添加一个测试函数lcd_test()，用于向framebuffer写数据，所需步骤如下：

- a. 初始化LCD
- b. 使能LCD
- c. 获取LCD参数: fb_base, xres, yres, bpp
- d. 往framebuffer中写数据

- a. 初始化LCD

```
lcd_init();
```

- b. 使能LCD

```
lcd_enable();
```

该函数实际调用的是lcd_controller_enable()

- c. 获取LCD参数: fb_base, xres, yres, bpp

只有获取到LCD的参数信息，才能根据这些信息进行相应显示。

```
get_lcd_params(&fb_base, &xres, &yres, &bpp);
```

该函数是在lcd.c里面实现：

```
void get_lcd_params(unsigned int *fb_base, int *xres, int *yres, int *bpp)
{
    *fb_base = g_p_lcd_selected->fb_base;
    *xres = g_p_lcd_selected->xres;
    *yres = g_p_lcd_selected->yres;
    *bpp = g_p_lcd_selected->bpp;
}
```

- d. 往framebuffer中写数据

假设现在BPP=16，想让全屏显示红色，就需要从framebuffer基地址开始一直填充对应的颜色数据。 对于16BPP，RGB=565，想显示红色，即[15:11]全为1表示红色，[10:5]全为0表示无绿色，[4:0]全为0表示无蓝色，0b1111100000000000=0xF700。

以基地址为起点，分别以xres和yres为边界，依次填充颜色。

```
        p = (unsigned short *)fb_base;
        for (x = 0; x < xres; x++)
            for (y = 0; y < yres; y++)
                *p++ = 0xf700;
```

编写好程序后，修改Makefile:

```
objs = start.o led.o uart.o init.o nand_flash.o main.o exception.o interrupt.o timer.o nor_flash.o my_printf.o string_utils.o lib1fun

objs += lcd/font.o
objs += lcd/framebuffer.o
objs += lcd/geometry.o
objs += lcd/lcd.o
objs += lcd/lcd_4.3.o
objs += lcd/lcd_controller.o
objs += lcd/lcd_test.o
objs += lcd/s3c2440_lcd_controller.o

all: $(objs)
    #arm-linux-ld -Ttext 0 -Tdata 0x30000000  start.o led.o uart.o init.o main.o -o sdram.elf
    arm-linux-ld -T sdram.lds $^ -o sdram.elf
    arm-linux-objcopy -O binary -S sdram.elf sdram.bin
    arm-linux-objdump -D sdram.elf > sdram.dis
clean:
    rm *.bin *.o *.elf *.dis

%.o : %.c
    arm-linux-gcc -march=armv4 -c -o $@ $<

%.o : %.S
    arm-linux-gcc -march=armv4 -c -o $@ $<
```

然后把工程文件放到虚拟机上交叉编译，根据编译提示结果进行对应修改。

常见问题包括：头文件未添加、数据类型错误等。

同理，假如现在是24BPP，即RGB:888，每个颜色占8位，一共占据24位。

虽然颜色数据只占据24位，但实际中是占据的32位(1字节)方便存储计算，即[23:0]存放的是数据，[31：24]空闲无数据。

对于32BPP，大多数情况下和24BPP差不多的，即RGB:888，每个颜色占8位，一共占据24位。因此想依次显示红绿蓝，代码如下：

```
        /* 0xRRGGBB */
        /* red */
        p2 = (unsigned int *)fb_base;
        for (x = 0; x < xres; x++)
            for (y = 0; y < yres; y++)
                *p2++ = 0xff0000;

        /* green */
        p2 = (unsigned int *)fb_base;
        for (x = 0; x < xres; x++)
            for (y = 0; y < yres; y++)
                *p2++ = 0x00ff00;

        /* blue */
        p2 = (unsigned int *)fb_base;
        for (x = 0; x < xres; x++)
            for (y = 0; y < yres; y++)
                *p2++ = 0x0000ff;
```

之前我们讲数据是8BPP的时候，可以通过调色板转成16BPP，在LCD上显示出相应颜色。

那前面的24BPP、32BPP是怎样在 只能接收16BPP(硬件上只有16根数据线)的LCD上显示的呢？

这是因为在使用24BPP时，发出的8条红色，8条绿色，8条蓝色数据，只用了高5条红色，高6条绿色，高5条蓝色与LCD相连。

第008节_编程_画点线圆

本节将在LCD上画点画圆，无论是何种图形，都是基于点来构成的，因此我们需要先实现画点。

在前面**第003节_编程_框架与准备**所讲的框架里，计划的是在farmebuffer.c实现画点，在geomentry.c实现画线、画圆，font.c实现写字。

我们先在farmebuffer.c实现画点，一个点(x，y)在FB中的位置如图：

可以得出其计算公式：

```
(x，y)像素起始地址=fb_base+(xres*bpp/8)*y + y*bpp/8
```

同时利用yres来分辨y方向的边界。

因此，需要先从LCD中获取参数：fb_base、xres、yres、bpp;

```
static unsigned int fb_base;
static int xres, yres, bpp;

void fb_get_lcd_params(void)
{
```

```
    get_lcd_params(&fb_base, &xres, &yres, &bpp);
}
```

再实现画点操作：

```
void fb_put_pixel(int x, int y, unsigned int color)
{
    unsigned char  *pc;  /* 8bpp */
    unsigned short *pw;  /* 16bpp */
    unsigned int   *pdw; /* 32bpp */

    unsigned int pixel_base = fb_base + (xres * bpp / 8) * y + x * bpp / 8;

    switch (bpp)
    {
        case 8:
            pc = (unsigned char *) pixel_base;
            *pc = color;
            break;
        case 16:
            pw = (unsigned short *) pixel_base;
            *pw = convert32bppto16bpp(color);
            break;
        case 32:
            pdw = (unsigned int *) pixel_base;
            *pdw = color;
            break;
    }
}
```

对于8PP，每个像素只占据8位(1字节)，因此采用unsigned char类型；

对于16PP，每个像素只占据16位(2字节)，因此采用unsigned short类型；

对于32PP，每个像素只占据32位(4字节)，因此采用unsigned int类型；

再根据传入的x,y坐标，计算出对应的显存位置。

根据BPP的不同，修改相应位的显存数据。

传入的颜色数据一般都是32bit的，即格式为：0x00RRGGBB,

对于8PP，通过的是调色板索引实现的，这个后续再讲解，直接*pc = color即可。

对于16PP，需要进行颜色转换。

对于32PP，大小刚好对应，直接*pc = color即可。

使用convert32bppto16bpp()函数进行颜色数据转换：

```
unsigned short convert32bppto16bpp(unsigned int rgb)
{
    int r = (rgb >> 16)& 0xff;
    int g = (rgb >> 8) & 0xff;
    int b = rgb & 0xff;

    /* rgb565 */
    r = r >> 3;
    g = g >> 2;
    b = b >> 3;

    return ((r<<11) | (g<<5) | (b));
}
```

先分别取出RGB，再相应的清除低位数据，实现将RGB888变为RGB565，最后再组成unsigned short类型数据返回。

画圆画线的具体原理不是我们的主要内容，我们直接百度“C语言 LCD 画圆”可以得到相关的实现代码，比如这篇博客：http://blog.csdn.net/p1126500468/article/details/50428613

新建一个geometry.c，复制博客中代码，替换里面的描点显示函数即可。

最后在主函数测试程序里，加上画圆画线的测试代码：

```
    /* 画线 */
    draw_line(0, 0, xres - 1, 0, 0xff0000);
    draw_line(xres - 1, 0, xres - 1, yres - 1, 0xffff00);
    draw_line(0, yres - 1, xres - 1, yres - 1, 0xff00aa);
    draw_line(0, 0, 0, yres - 1, 0xff00ef);
    draw_line(0, 0, xres - 1, yres - 1, 0xff4500);
    draw_line(xres - 1, 0, 0, yres - 1, 0xff0780);

    delay(1000000);

    /* 画圆 */
    draw_circle(xres/2, yres/2, yres/4, 0xff00);
```

希望在LCD上显示如下图形，由6条线和一个圆组成。

将线的起始坐标作为参数传入画线函数。

将圆心和半径作为参数传入画圆函数。

第009节_编程_显示文字

文字也是由点构成的，一个个点组成的点阵，宏观的来看，就是文字。

可以参考Linux内核源码中的相关操作，在内核中搜索“font”，打开font_8x16.c，可以看到里面的A字符内容如下：

```
    /* 65 0x41 'A' */
    0x00, /* 00000000 */
    0x00, /* 00000000 */
    0x10, /* 00010000 */
    0x38, /* 00111000 */
    0x6c, /* 01101100 */
    0xc6, /* 11000110 */
    0xc6, /* 11000110 */
```

```
    0xfe, /* 11111110 */
    0xc6, /* 11000110 */
    0xc6, /* 11000110 */
    0xc6, /* 11000110 */
    0xc6, /* 11000110 */
    0x00, /* 00000000 */
    0x00, /* 00000000 */
    0x00, /* 00000000 */
    0x00, /* 00000000 */
```

根据这些数据，在一个8*16的区域里，将为1的点显示出来，为0的则不显示，最终将呈现一个字母“A”。

新建一个font.c，根据字母的点阵在LCD上描画文字，需要的步骤如下：

- a. 根据带显示的字符的ascii码在fontdata_8x16中得到点阵数据
- b. 根据点阵来设置对应象素的颜色
- c. 根据点阵的某位决定是否描颜色

```
void fb_print_char(int x, int y, char c, unsigned int color)
{
    int i, j;

    /* 根据c的ascii码在fontdata_8x16中得到点阵数据 */
    unsigned char *dots = &fontdata_8x16[c * 16];

    unsigned char data;
    int bit;

    /* 根据点阵来设置对应象素的颜色 */
    for (j = y; j < y+16; j++)
    {
        data = *dots++;
        bit = 7;
        for (i = x; i < x+8; i++)
        {
            /* 根据点阵的某位决定是否描颜色 */
            if (data & (1<<bit))
                fb_put_pixel(i, j, color);
            bit--;
        }
    }
}
```

在font_8x16.c里面，每个字符占据16位，因此想要根据ascii码找到对应的点阵数据，需要对应的乘16，再取地址，得到该字符的首地址。

再根据每个点阵数据每位是否为1，来调用描点函数fb_put_pixel()。这样，依次显示16个点阵数据，获得字符图形。

同样的，在显示之前，还需要获取LCD参数：

```
extern const unsigned char fontdata_8x16[];
/* 获得LCD参数 */
static unsigned int fb_base;
static int xres, yres, bpp;

void font_init(void)
{
    get_lcd_params(&fb_base, &xres, &yres, &bpp);
}
```

如果想显示字符串，那就在每显示完一个字符后，x轴加8即可，同时考虑是否超出屏幕显示范围进行换行处理：

```
/* "abc\n\r123" */
void fb_print_string(int x, int y, char* str, unsigned int color)
{
    int i = 0, j;

    while (str[i])
    {
        if (str[i] == '\n')
```

```
            y = y+16;
        else if (str[i] == '\r')
            x = 0;

        else
        {
            fb_print_char(x, y, str[i], color);
            x = x+8;
            if (x >= xres) /* 换行 */
            {
                x = 0;
                y = y+16;
            }
        }
        i++;
    }
}
```

最后在在主函数里，加上显示字符串的函数，传入希望显示的字符串。

第010节_编程_添加除法

在s3c2440_lcd_controller.c里，以前我们使用如下除法：

```
int clkval = (double)HCLK/plcdparams->time_seq.vclk/2-1+0.5;
```

编译的时候会提示出错：

我们的lib1funcs.S里面是有除法的，但除法的功能不够强，将前面除法计算代码中的double改成精度更小的float还是不行。

对于未实现的函数：

- a. 去uboot中查找
- b. 去内核源码中查找
- c. 去库函数中查找(一般来说编译器自带有很多库)

在库函数中查找步骤：

- a. 输入命令arm-linux-gcc -v，查看当前使用的交叉编译工具链；
- b. 输入命令echo $PATH，在环境变量中找到当前使用的交叉编译工具链所在的路径；
- c. 进入交叉编译工具链所在目录搜索相关函数，例如grep "__floatsisf" * -nR；
- d. 提取出其中的静态库(.a后缀文件)，复制文件到代码文件；
- e. 修改Makefile，依次尝试加入的每个静态库，直至编译成功；

注意:如果你更换了编译器，需要自己去编译器目录里找出对应的libgcc.a;

有可能有多个libgcc.a，逐个尝试；

第011节_编程_使用调色板

前面我们写的程序都是采用的16BPP或者24BPP(也就是32BPP)，假如我们要使用8PP，就得使用调色板。

如图所示8PP工作原理示意图，在FB只存放8bit得每个像素索引，根据这个索引，在去去调色板找到对应的数据传给LCD控制器，再通过电子枪显示出来。

调色板里面有2^8(256)个颜色数据，每个颜色数据为16bit，表示一种颜色。

在硬件上，我们要初始化这个调色板，才能通过索引得到颜色。

根据第三节的软件框架，调色板的初始化应该放在s3c2440_lcd_controller.c里面。

在lcd_controller结构体里添加调色板初始化函数：

```
struct lcd_controller s3c2440_lcd_controller = {
    .name     = "s3c2440",
    .init     = s3c2440_lcd_controller_init,
    .enable   = s3c2440_lcd_controller_enalbe,
    .disable  = s3c2440_lcd_controller_disable,
    .init_palette = s3c2440_lcd_controller_init_palette,
};
```

调色板对应一块内存，我们需要找到他的位置和格式。

从芯片手册了解到，其起始地址为0x4D000400，且设置调色板前，需要关闭LCD控制器。

```
void s3c2440_lcd_controller_init_palette(void)
{
    volatile unsigned int *palette_base =  (volatile unsigned int *)0x4D000400;
    int i;

    int bit = LCDCON1 & (1<<0);

    /* LCDCON1'BIT 0 : 设置LCD控制器是否输出信号 */
    if (bit)
        LCDCON1 &= ~(1<<0);

    for (i = 0; i < 256; i++)
    {
        /* 低16位 : rgb565 */
        *palette_base++ = i;
    }

    if (bit)
        LCDCON1 |= (1<<0);
}
```

设置调色板前，先判断LCD控制器是否打开，如果打开了就先关闭，且设置完成后再打开。

再网上搜索了一下，没有找到调色板数据数组，这里作为实验，就随便设置，让其为i。

我们让调色板数据等于i，0-255，只占据8位，最后的颜色范围为B5G3R0，因此会比较偏蓝。

修改lcd_4.3.c，将BPP改为8，

再修改lcd_controller.c的初始化函数，加入调色板初始化函数。

```
int lcd_controller_init(p_lcd_params plcdparams)
{
    /* 调用所选择的LCD控制器的初始化函数 */
    if (g_p_lcd_controller_selected)
    {
```

```
        g_p_lcd_controller_selected->init(plcdparams);
        g_p_lcd_controller_selected->init_palette();
        return 0;
    }
    return -1;
}
```

再修改lcd_test.c，加入bpp=8的情况：

```
    if (bpp == 8)
    {
        /* 让LCD输出整屏的红色 */

        /* bpp: palette[12] */

        p0 = (unsigned char *)fb_base;
        for (x = 0; x < xres; x++)
            for (y = 0; y < yres; y++)
                *p0++ = 12;

        /* palette[47] */
        p0 = (unsigned char *)fb_base;
        for (x = 0; x < xres; x++)
            for (y = 0; y < yres; y++)
                *p0++ = 47;

        /* palette[88] */
        p0 = (unsigned char *)fb_base;
        for (x = 0; x < xres; x++)
            for (y = 0; y < yres; y++)
                *p0++ = 88;

        /* palette[0] */
        p0 = (unsigned char *)fb_base;
        for (x = 0; x < xres; x++)
            for (y = 0; y < yres; y++)
                *p0++ = 0;

    }
```

随便让其全屏显示某个颜色。

最后在画线画圆，显示文字的函数里，修改下颜色：

```
    /* 画线 */
    draw_line(0, 0, xres - 1, 0, 0x23ff77);
    draw_line(xres - 1, 0, xres - 1, yres - 1, 0xffff);
    draw_line(0, yres - 1, xres - 1, yres - 1, 0xff00aa);
    draw_line(0, 0, 0, yres - 1, 0xff00ef);
    draw_line(0, 0, xres - 1, yres - 1, 0xff45);
    draw_line(xres - 1, 0, 0, yres - 1, 0xff0780);

    delay(1000000);

    /* 画圆 */
    draw_circle(xres/2, yres/2, yres/4, 0xff);

    /* 输出文字 */
    fb_print_string(10, 10, "www.100ask.net\n\r100ask.taobao.com", 0xff);
```

上面的颜色数据，其实只有低两位有效，因为前面的调色板映射范围是0-255。

《《所有章节目录》》

第006课 开发板熟悉与体验(免费)
第007课 裸机开发步骤和工具使用(免费))
第008课 第1个ARM裸板程序及引申(部分免费)
第009课 gcc和arm-linux-gcc和Makefile
第010课 掌握ARM芯片时钟体系
第011课 串口(UART)的使用
第012课 内存控制器与SDRAM
第013课 代码重定位
第014课 异常与中断
第015课 NOR Flash
第016课 Nand Flash
第017课 LCD
第018课 ADC和触摸屏
第019课 I2C
第20课 SPI

分类： ARM裸机加强版

- 本页面最后修改于2018年1月29日 (星期一) 10:33。