LINUX

业界权威机构和专家强力推荐
多年培训、研发经验的总结

设备驱动开发详解

华清远见嵌入式培训中心 宋宝华 编著

Linux

附源代码光盘

人民邮电出版社
POSTS & TELECOM PRESS

第 18 章　LCD 设备驱动

在多媒体应用的推动下，彩色 LCD 越来越多地应用到了嵌入式系统中，掌上电脑（PDA）、手机等多采用 TFT 显示器件，支持彩色图形界面，能显示图片并进行视频媒体播放。帧缓冲（Framebuffer）是 Linux 为显示设备提供的一个接口，它允许上层应用程序在图形模式下直接对显示缓冲区进行读写操作。

本章主要讲解帧缓冲设备 Linux 驱动的架构及编程方法。

18.1 节讲解了 LCD 的底层硬件操作原理，18.2 节讲解了帧缓冲设备的概念及驱动中的重要数据结构和函数。

18.3 节讲解了帧缓冲设备驱动的整体结构，18.4～18.8 节分别讲解了帧缓冲设备的几个重要函数，18.3 节和 18.4～18.8 节的内容是整体与部分的关系。

18.9 节讲解了 Linux 帧缓冲设备用户空间的访问方法，并给出了两个常用的 GUI 环境—QT/E 及 Minigui 的基本用法。

18.10 节讲解了 S3C2410 LCD 控制器设备驱动的实例。

18.1 LCD 硬件原理

利用液晶制成的显示器称为 LCD，依据驱动方式可分为静态驱动、简单矩阵驱动以及主动矩阵驱动 3 种。其中，简单矩阵型又可再细分扭转向列型（TN）和超扭转式向列型（STN）两种，而主动矩阵型则以薄膜式晶体管型（TFT）为主流。表 18.1 列出了 TN、STN 和 TFT 显示器的区别。

表 18.1　　TN、STN 和 TFT 显示器的区别

类　别	TN	STN	TFT
原理	液晶分子，扭转 90°	扭转 180°～270°	液晶分子，扭转 90°
特性	黑白、单色低对比	黑白、彩色，低对比	彩色（1667 万色），可媲美 CRT 显示器的全彩，高对比
动画显示	否	否	是
视角	30° 以下	40° 以下	80° 以下
面板尺寸	1～3 英寸	1～12 英寸	37 英寸

TN 型液晶显示技术是 LCD 中最基本的，其他种类的 LCD 都以 TN 型为基础改进而得。TN 型 LCD 显示质量很差，色彩单一，对比度低，反映速度很慢，故主要用于简单的数字符与文字的显示，如电子表及电子计算器等。

STN LCD 的显示原理与 TN 类似，区别在于 TN 型的液晶分子将入射光旋转 90°，而 STN 则可将入射光旋转 180°～270°。STN 改善了 TN 视角狭小的缺点，并提高了对比度，显示品质较 TN 高。

STN 搭配彩色滤光片，将单色显示矩阵的任一像素分成 3 个子像素，分别透过彩色滤光片显示红、绿、蓝三原色，再经由三原色按比例调和，显示出逼近全彩模式的色彩。STN 显示的画面色彩对比度仍较小，反应速度也较慢，可以作为一般的操作显示接口。

随后出现的 DSTN 通过双扫描方式来显示，显示效果相对 STN 而言有了较大幅度的提高。DSTN 的反应速度可达到 100ms，但是在电场反复改变电压的过程中，每一像素的恢复过程较慢。因此，当在屏幕画面快速变化时，会产生“拖尾”现象。

TN 与 STN 型液晶显示器都是使用场电压驱动方式，如果显示尺寸加大，中心部位对电极变化的反应时间就会拉长，显示器的速度跟不上。为了解决这个问题，主动式矩阵驱动被提出，主动式 TFT 型的液晶显示器的结构较为复杂，它包括背光管、导光板、偏光板、滤光板、玻璃基板、配向膜、液晶材料和薄膜式晶体管等。

在 TFT 型 LCD 中，晶体管矩阵依显示信号开启或关闭液晶分子的电压，使液晶分子轴转向而成“亮”或“暗”的对比，避免了显示器对电场效应的依靠。因此，TFT

LCD 的显示质量较 TN/STN 更佳，画面显示对比度可达 150:1 以上，反应速度逼近 30ms 甚至更快，适用于 PDA、笔记本电脑、数码相机、MP4 等。

一块 LCD 屏显示图像不但需要 LCD 驱动器，还需要有相应的 LCD 控制器。通常 LCD 驱动器会以 COF/COG 的形式与 LCD 玻璃基板制作在一起，而 LCD 控制器则由外部电路来实现。许多 MCU 内部直接集成了 LCD 控制器，通过 LCD 控制器可以方便地控制 STN 和 TFT 屏。

TFT 屏是目前嵌入式系统应用的主流，图 18.1 给出了 TFT 屏的典型时序。时序图中的 VCLK、HSYNC 和 VSYNC 分别为像素时钟信号（用于锁存图像数据的像素时钟）、行同步信号和帧同步信号，VDEN 为数据有效标志信号，VD 为图像的数据信号。

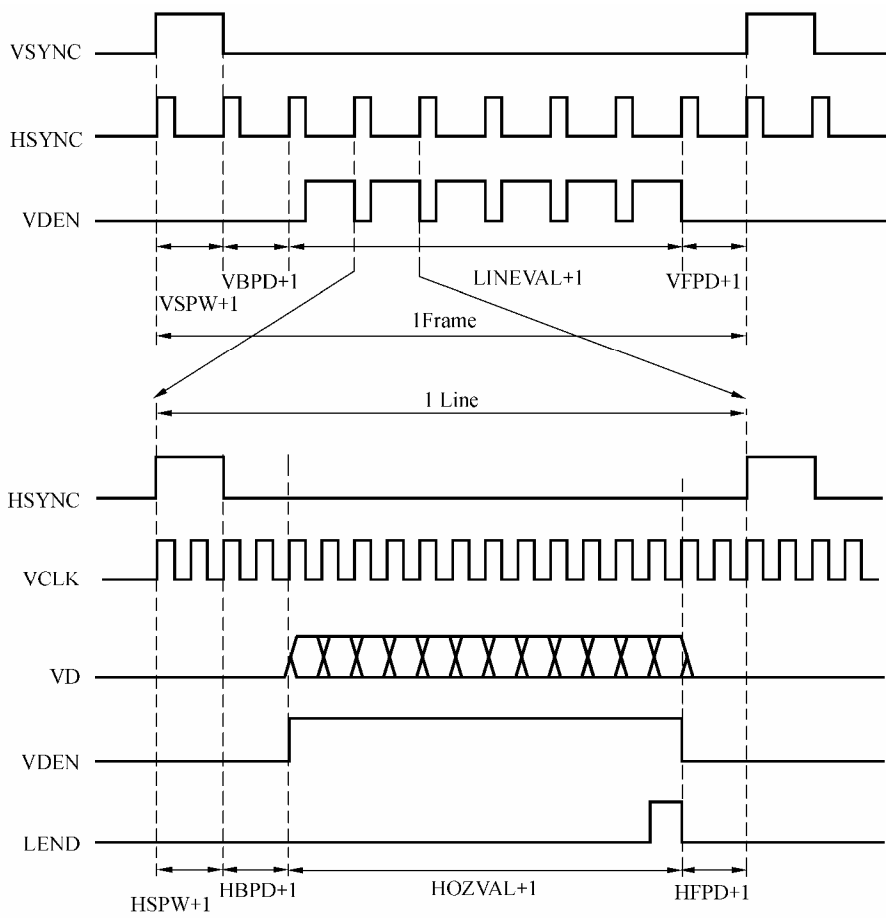


图 18.1　TFT 屏工作时序

作为帧同步信号的 VSYNC，每发出一个脉冲，都意味着新的一屏图像数据开始发送。而作为行同步信号的 HSYNC，每发出一个脉冲都表明新的一行图像资料开始发送。在帧同步以及行同步的头尾都必须留有回扫时间。这样的时序安排起源于 CRT 显示器电子枪偏转所需要的时间，但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 给出了 LCD 控制器中应该设置的 TFT 屏的参数，其中的上边界和下边界

即为帧切换的回扫时间，左边界和右边界即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 分辨率主要为 320*240、640*480 等。

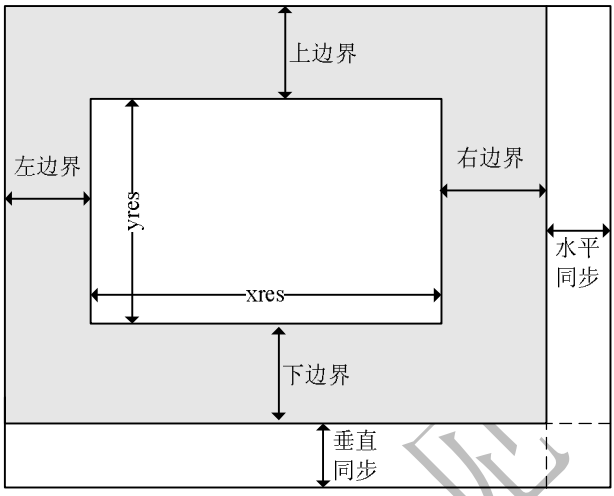


图 18.2　LCD 控制器中的时序参数设置

18.2 帧缓冲

18.2.1　帧缓冲的概念

帧缓冲（framebuffer）是 Linux 系统为显示设备提供的一个接口，它将显示缓冲区抽象，屏蔽图像硬件的底层差异，允许上层应用程序在图形模式下直接对显示缓冲区进行读写操作。用户不必关心物理显示缓冲区的具体位置及存放方式，这些都由帧缓冲设备驱动本身来完成。对于帧缓冲设备而言，只要在显示缓冲区中与显示点对应的区域写入颜色值，对应的颜色会自动在屏幕上显示，18.2.2 小节将讲解显示缓冲区与显示点的对应关系。

帧缓冲设备为标准字符设备，主设备号为 29，对应于/dev/fb%d 设备文件。帧缓冲驱动的应用非常广泛，在 Linux 的桌面系统中，Xwindow 服务器就是利用帧缓冲进行窗口的绘制。嵌入式系统中的 Qt/Embedded 等图形用户界面环境也基于帧缓冲而设计。另外，通过帧缓冲可支持汉字点阵的显示，因此帧缓冲也成为 Linux 汉化的可行方案。

18.2.2　显示缓冲区与显示点

在帧缓冲设备中，对屏幕显示点的操作通过读写显示缓冲区来完成，在不同的色彩模式下，显示缓冲区和屏幕上的显示点有不同的对应关系，表 18.2～表 18.4 分别给

出了 16 级灰度、8 位色和 16 位情况下显示缓冲区与显示点的对应关系。

表 18.2　　16 级灰度显示缓冲区与显示点的对应关系

位	31～28	27～24	23～20	19～16	15～12	11～8	7～4	3～0
0x0	点 7	点 6	点 5	点 4	点 3	点 2	点 1	点 0
0x04	点 15	点 14	点 13	点 12	点 11	点 10	点 9	点 8
…	…	…	…	…	…	…	…	…

续表

位	31～28	27～24	23～20	19～16	15～12	11～8	7～4	3～0
0x0	点 0	点 1	点 2	点 3	点 4	点 5	点 6	点 7
0x04	点 8	点 9	点 10	点 11	点 12	点 13	点 14	点 15
…	…	…	…	…	…	…	…	…

表 18.3　　8 位色时显示缓冲区与显示点的对应关系

RGB			BGR		
7～5	4～2	1～0	7～5	4～2	1～0
R	G	B			

表 18.4　　16 位色时显示缓冲区与显示点的对应关系

位	15～11	10～5	4～0
RGB565	R	G	B
RGB555	R	G	B

18.2.3　Linux 帧缓冲相关数据结构与函数

1．fb_info 结构体

帧缓冲设备最关键的一个数据结构体是 fb_info 结构体（为了便于记忆，我们把它简称为“FBI”），FBI 中包括了关于帧缓冲设备属性和操作的完整描述，这个结构体的定义如代码清单 18.1 所示。

代码清单 18.1　fb_info 结构体

```
1  struct fb_info
2  {
3     int node;
4     int flags;
5     struct fb_var_screeninfo var; /*可变参数 */
6     struct fb_fix_screeninfo fix; /*固定参数 */
7     struct fb_monspecs monspecs; /*显示器标准 */
8     struct work_struct queue; /* 帧缓冲事件队列 */
9     struct fb_pixmap pixmap; /* 图像硬件 mapper */
10    struct fb_pixmap sprite; /* 光标硬件 mapper */
11    struct fb_cmap cmap; /* 目前的颜色表*/
12    struct list_head modelist;
13    struct fb_videomode *mode; /* 目前的 video 模式 */
14
```

```
15   #ifdef CONFIG_FB_BACKLIGHT
16     struct mutex bl_mutex;
17     /* 对应的背光设备 */
18     struct backlight_device *bl_dev;
19     /* 背光调整 */
20     u8 bl_curve[FB_BACKLIGHT_LEVELS];
21    #endif
22
23    struct fb_ops *fbops; /* fb_ops，帧缓冲操作 */
24    struct device *device;
25    struct class_device *class_device; /
26    int class_flag; /* 私有 sysfs 标志 */
27    #ifdef CONFIG_FB_TILEBLITTING
28       struct fb_tile_ops *tileops; /* 图块 Blitting */
29    #endif
30    char __iomem *screen_base; /* 虚拟基地址 */
31    unsigned long screen_size; /* ioremapped 的虚拟内存大小 */
32    void *pseudo_palette; /* 伪 16 色颜色表 */
33    #define FBINFO_STATE_RUNNING 0
34    #define FBINFO_STATE_SUSPENDED       1
35    u32 state; /* 硬件状态，如挂起 */
36    void *fbcon_par;
37    void *par;
38 };
```

FBI 中记录了帧缓冲设备的全部信息，包括设备的设置参数、状态以及操作函数指针。每一个帧缓冲设备都必须对应一个 FBI。

2．fb_ops 结构体

FBI 的成员变量 fbops 为指向底层操作的函数的指针，这些函数是需要驱动程序开发人员编写的，其定义如代码清单 18.2 所示。

代码清单 18.2　fb_ops 结构体

```
1  struct fb_ops
2  {
3     struct module *owner;
4      /* 打开/释放 */
5     int(*fb_open)(struct fb_info *info, int user);
6     int(*fb_release)(struct fb_info *info, int user);
7
8     /* 对于非线性布局的/常规内存映射无法工作的帧缓冲设备需要 */
9     ssize_t(*fb_read)(struct file *file, char __user *buf, size_t
count,
10       loff_t*ppos);
11    ssize_t(*fb_write)(struct file *file, const char __user *buf,
size_t count,
12       loff_t *ppos);
13
14    /* 检测可变参数，并调整到支持的值*/
15       int(*fb_check_var)(struct  fb_var_screeninfo  *var,  struct
fb_info *info);
16
17   /* 根据 info->var 设置 video 模式 */
```

```
    18    int(*fb_set_par)(struct fb_info *info);
    19
    20    /* 设置 color 寄存器 */
    21    int(*fb_setcolreg)(unsigned regno, unsigned red, unsigned green,
unsigned
    22       blue, unsigned transp, struct fb_info *info);
    23
    24     /* 批量设置 color 寄存器，设置颜色表 */
    25     int(*fb_setcmap)(struct fb_cmap *cmap, struct fb_info *info);
    26
    27     /*显示空白 */
    28     int(*fb_blank)(int blank, struct fb_info *info);
    29
    30     /* pan 显示 */
    31       int(*fb_pan_display)(struct fb_var_screeninfo *var, struct
fb_info *info);
    32
    33     /* 矩形填充 */
    34         void(*fb_fillrect)(struct fb_info *info, const struct
fb_fillrect *rect);
    35     /* 数据复制 */
    36         void(*fb_copyarea)(struct fb_info *info, const struct
fb_copyarea *region);
    37     /* 图形填充 */
    38     void(*fb_imageblit)(struct fb_info *info, const struct fb_image
*image);
    39
    40     /* 绘制光标 */
    41     int(*fb_cursor)(struct fb_info *info, struct fb_cursor *cursor);
    42
    43     /* 旋转显示 */
    44     void(*fb_rotate)(struct fb_info *info, int angle);
    45
    46     /* 等待 blit 空闲 (可选) */
    47     int(*fb_sync)(struct fb_info *info);
    48
    49     /* fb 特定的 ioctl (可选) */
    50     int(*fb_ioctl)(struct fb_info *info, unsigned int cmd, unsigned
long arg);
    51
    52   /* 处理 32 位的 compat ioctl (可选) */
    53      int(*fb_compat_ioctl)(struct fb_info *info, unsigned cmd,
unsigned long arg);
    54
    55   /* fb 特定的 mmap */
    56   int(*fb_mmap)(struct fb_info *info, struct vm_area_struct *vma);
    57
    58   /* 保存目前的硬件状态 */
    59   void(*fb_save_state)(struct fb_info *info);
    60
    61   /* 恢复被保存的硬件状态 */
    62   void(*fb_restore_state)(struct fb_info *info);
    63 };
```

fb_ops 的 fb_check_var()成员函数用于检查可以修改的屏幕参数并调整到合适的值，而 fb_set_par()则使得用户设置的屏幕参数在硬件上有效。

3．fb_var_screeninfo 和 fb_fix_screeninfo 结构体

FBI 的 fb_var_screeninfo 和 fb_fix_screeninfo 成员也是结构体，fb_var_screeninfo 记录用户可修改的显示控制器参数，包括屏幕分辨率和每个像素点的比特数。fb_var_screeninfo 中的 xres 定义屏幕一行有多少个点，yres 定义屏幕一列有多少个点，bits_per_pixel 定义每个点用多少个字节表示。而 fb_fix_screeninfo 中记录用户不能修改的显示控制器的参数，如屏幕缓冲区的物理地址、长度。当对帧缓冲设备进行映射操作的时候，就是从 fb_fix_screeninfo 中取得缓冲区物理地址的。上述数据成员都需要在驱动程序中初始化和设置。

fb_var_screeninfo 和 fb_fix_screeninfo 结构体的定义分别如代码清单 18.3 和代码清单 18.4 所示。

代码清单 18.3　fb_var_screeninfo 结构体

```
1  struct fb_var_screeninfo
2  {
3     /* 可见解析度   */
4     __u32 xres;
5     __u32 yres;
6     /* 虚拟解析度    */
7     __u32 xres_virtual;
8     __u32 yres_virtual;
9     /* 虚拟到可见之间的偏移 */
10    __u32 xoffset;
11    __u32 yoffset;
12
13    __u32 bits_per_pixel; /* 每像素位数，BPP */
14    __u32 grayscale; /非 0 时指灰度 */
15
16    /* fb 缓存的 R\G\B 位域 */
17    struct fb_bitfield red;
18    struct fb_bitfield green;
19    struct fb_bitfield blue;
20    struct fb_bitfield transp; /* 透明度 */
21
22    __u32 nonstd; /* != 0 非标准像素格式 */
23
24    __u32 activate;
25
26    __u32 height; /*高度 */
27    __u32 width; /*宽度 */
28
29    __u32 accel_flags; /* 看 fb_info.flags */
30
31    /* 定时: 除了 pixclock 本身外，其他的都以像素时钟为单位 */
32    __u32 pixclock; /* 像素时钟（皮秒） */
```

```
33    __u32 left_margin; /* 行切换: 从同步到绘图之间的延迟    */
34    __u32 right_margin; /* 行切换: 从绘图到同步之间的延迟   */
35    __u32 upper_margin; /* 帧切换: 从同步到绘图之间的延迟  */
36    __u32 lower_margin; /* 帧切换: 从绘图到同步之间的延迟  */
37    __u32 hsync_len; /* 水平同步的长度     */
38    __u32 vsync_len; /* 垂直同步的长度     */
39    __u32 sync;
40    __u32 vmode;
41    __u32 rotate; /* 顺时钟旋转的角度 */
42    __u32 reserved[5]; /* 保留 */
43 };
```

代码清单 18.4　fb_fix_screeninfo 结构体

```
1  struct fb_fix_screeninfo
2  {
3     char id[16]; /* 字符串形式的标识符 */
4     unsigned long smem_start; /* fb 缓存的开始位置 */
5     __u32 smem_len; /* fb 缓存的长度 */
6     __u32 type; /* FB_TYPE_*       */
7     __u32 type_aux; /* 分界 */
8     __u32 visual; /* FB_VISUAL_* */
9     __u16 xpanstep; /* 如果没有硬件 panning ，赋 0 */
10    __u16 ypanstep;
11    __u16 ywrapstep;/
12    __u32 line_length; /* 1 行的字节数 */
13    unsigned long mmio_start; /* 内存映射 I/O 的开始位置 */
14    __u32 mmio_len; /* 内存映射 I/O 的长度  */
15    __u32 accel;
16    __u16 reserved[3]; /* 保留*/
17 };
```

代码清单 18.4 中第 8 行的 visual 记录屏幕使用的色彩模式，在 Linux 系统中，支持的色彩模式包括如下几种。

- Monochrome（FB_VISUAL_MONO01、FB_VISUAL_MONO10），每个像素是黑或白。
- Pseudo color（FB_VISUAL_PSEUDOCOLOR、FB_VISUAL_STATIC_PSEUDOCOLOR），即伪彩色，采用索引颜色显示。
- True color（FB_VISUAL_TRUECOLOR），真彩色，分成红、绿、蓝三基色。
- Direct color（FB_VISUAL_DIRECTCOLOR），每个像素颜色也是有红、绿、蓝组成，不过每个颜色值是个索引，需要查表。
- Grayscale displays，灰度显示，红、绿、蓝的值都一样。

4．fb_bitfield 结构体

代码清单 18.3 第 17、18、19 行分别记录 R、G、B 的位域，fb_bitfield 结构体描述每一像素显示缓冲区的组织方式，包含位域偏移、位域长度和 MSB 指示，如代码清单 18.5 所示。

代码清单 18.5　fb_bitfield 结构体

```
1  struct fb_bitfield
2  {
3     __u32 offset; /* 位域偏移    */
4     __u32 length; /* 位域长度    */
```

```
    5     __u32 msb_right; /* MSB */
    6 };
```

5．fb_cmap 结构体

fb_cmap 结构体记录设备无关的颜色表信息，用户空间可以通过 ioctl()的 FBIOGETCMAP 和 FBIOPUTCMAP 命令读取或设定颜色表。

代码清单 18.6　fb_cmap 结构体

```
    1 struct fb_cmap
    2 {
    3     __u32 start; /* 第1个元素入口    */
    4     __u32 len; /* 元素数量 */
    5     /* R、G、B、透明度*/
    6     __u16 *red;
    7     __u16 *green;
    8     __u16 *blue;
    9     __u16 *transp;
    10 };
```

代码清单 18.7 所示为用户空间获取颜色表的例程，若 BPP 为 8 位，则颜色表长度为 256；若 BPP 为 4 位，则颜色表长度为 16；否则，颜色表长度为 0，这是因为，对于 BPP 大于等于 16 的情况，使用颜色表是不划算的。

代码清单 18.7　用户空间获取颜色表例程

```
    1 // 读入颜色表
    2 if ((vinfo.bits_per_pixel == 8) || (vinfo.bits_per_pixel == 4))
    3 {
    4      screencols = (vinfo.bits_per_pixel == 8) ? 256 : 16;//颜色表大
小
    5      int loopc;
    6      startcmap = new fb_cmap;
    7      startcmap->start = 0;
    8      startcmap->len = screencols;
    9     //分配颜色表的内存
    10     startcmap->red = (unsigned short int*)malloc(sizeof(unsigned
short int)
    11        *screencols);
    12    startcmap->green = (unsigned short int*)malloc(sizeof(unsigned
short int)
    13        *screencols);
    14    startcmap->blue = (unsigned short int*)malloc(sizeof(unsigned
short int)
    15        *screencols);
    16    startcmap->transp = (unsigned short int*)malloc(sizeof(unsigned
short int)
    17        *screencols);
    18    //获取颜色表
    19    ioctl(fd, FBIOGETCMAP, startcmap);
    20    for (loopc = 0; loopc < screencols; loopc++)
    21    {
    22             screenclut[loopc] = qRgb(startcmap->red[loopc] >> 8,
startcmap
    23             ->green[loopc] >> 8, startcmap->blue[loopc] >> 8);
```

```
24   }
25 }
26 else
27 {
28     screencols = 0;
29 }
```

对于一个 256 色（BPP=8）的 800*600 分辨率的图像而言，若红、绿、蓝分别用一个字节描述，则需要 800*600*3=1440000Byte 的空间，而若使用颜色表，则只需要 800*600*1+256*3= 480768Byte 的空间。

6．文件操作结构体

作为一种字符设备，帧缓冲设备的文件操作结构体定义于/linux/drivers/vedio/fbmem.c 文件中，如代码清单 18.8 所示。

代码清单 18.8　帧缓冲设备文件操作结构体

```
1  static struct file_operations fb_fops =
2  {
3      .owner = THIS_MODULE,
4      .read = fb_read, //读函数
5      .write = fb_write, //写函数
6      .ioctl = fb_ioctl, //I/O 控制函数
7      #ifdef CONFIG_COMPAT
8         .compat_ioctl = fb_compat_ioctl,
9      #endif
10     .mmap = fb_mmap, //内存映射函数
11     .open = fb_open,  //打开函数
12     .release = fb_release, //释放函数
13     #ifdef HAVE_ARCH_FB_UNMAPPED_AREA
14        .get_unmapped_area = get_fb_unmapped_area,
15     #endif
16 };
```

帧缓冲设备驱动的文件操作接口函数已经在 fbmem.c 中被统一实现，一般不需要由驱动工程师再编写。

7．注册与注销帧缓冲设备

Linux 内核提供了 register_framebuffer()和 unregister_framebuffer()函数分别注册和注销帧缓冲设备，这两个函数都接受 FBI 指针为参数，原型为：

```
int register_framebuffer(struct fb_info *fb_info);
int unregister_framebuffer(struct fb_info *fb_info);
```

对于 register_framebuffer()函数而言，如果注册的帧缓冲设备数超过了 FB_MAX（目前定义为 32），则函数返回-ENXIO，注册成功则返回 0。

18.3 Linux 帧缓冲设备驱动结构

图 18.3 所示为 Linux 帧缓冲设备驱动的主要结构，帧缓冲设备提供给用户空间的

file_operations 结构体由 fbmem.c 中的 file_operations 提供，而特定帧缓冲设备 fb_info 结构体的注册、注销以及其中成员的维护，尤其是 fb_ops 中成员函数的实现则由对应的 xxxfb.c 文件实现，fb_ops 中的成员函数最终会操作 LCD 控制器硬件寄存器。

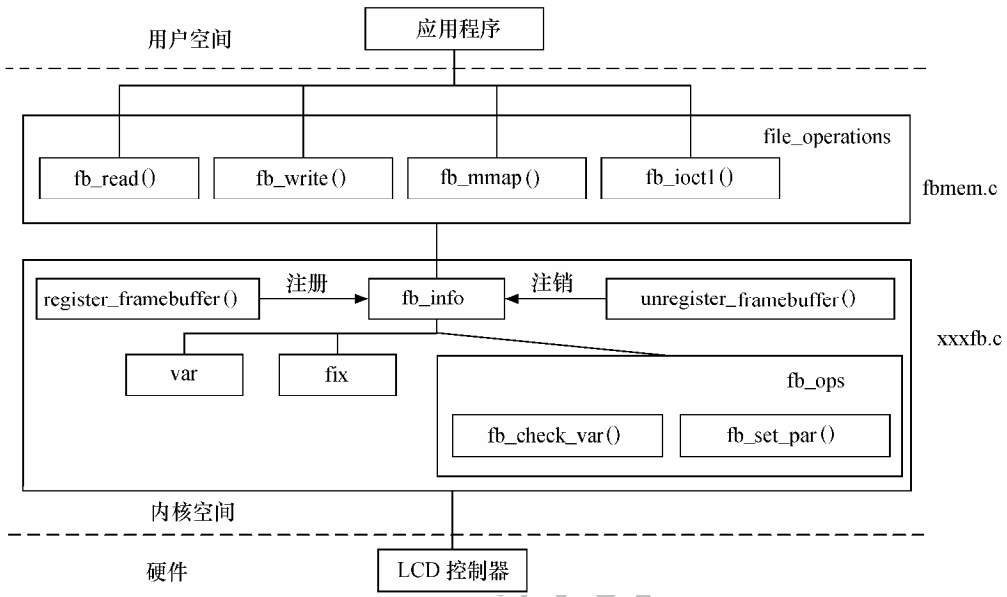


图 18.3　帧缓冲设备驱动的程序结构

18.4 帧缓冲设备驱动的模块加载与卸载函数

在帧缓冲设备驱动的模块加载函数中，应该完成如下 4 个工作。

- 申请 FBI 结构体的内存空间，初始化 FBI 结构体中固定和可变的屏幕参数，即填充 FBI 中 fb_var_screeninfo var 和 struct fb_fix_screeninfo fix 成员。
- 根据具体 LCD 屏幕的特点，完成 LCD 控制器硬件的初始化。
- 申请帧缓冲设备的显示缓冲区空间。
- 注册帧缓冲设备。

在帧缓冲设备驱动的模块卸载函数中，应该完成相反的工作，包括释放 FBI 结构体内存、关闭 LCD、释放显示缓冲区以及注销帧缓冲设备。

由于 LCD 控制器经常被集成在 SoC 上作为一个独立的硬件模块而存在（成为 platform_device），因此，LCD 驱动中也经常包含平台驱动，这样，在帧缓冲设备驱动的模块加载函数中完成的工作只是注册平台驱动，而初始化 FBI 结构体中的固定和可变参数、LCD 控制器硬件的初始化、申请帧缓冲设备的显示缓冲区空间和注册帧缓冲设备的工作则移交到平台驱动的探测函数中完成。

同样地，在使用平台驱动的情况下，释放 FBI 结构体内存、关闭 LCD、释放显示缓冲区以及注销帧缓冲设备的工作也移交到平台驱动的移除函数中完成。

代码清单 18.9 所示为帧缓冲设备驱动的模块加载和卸载以及平台驱动的探测和移除函数中的模板。

代码清单 18.9　帧缓冲设备驱动的模块加载/卸载及平台驱动的探测/移除函数的模板

```
1  /* 平台驱动结构体 */
2  static struct platform_driver xxxfb_driver =
3  {
4     .probe = xxxfb_probe,         //平台驱动探测函数
5     .remove = xxxfb_remove,      //平台驱动移除函数
6     .suspend = xxxfb_suspend,   .resume = xxxfb_resume, .driver =
7     {
8        .name = "xxx-lcd",  //驱动名
9        .owner = THIS_MODULE,
10    }
11 };
12
13 /* 平台驱动探测函数 */
14 static int __init xxxfb_probe(...)
15 {
16     struct fb_info *info;
17
18     /*分配 fb_info 结构体*/
19     info = framebuffer_alloc(...);
20
21     info->screen_base = framebuffer_virtual_memory;
22     info->var = xxxfb_var;  //可变参数
23    info->fix = xxxfb_fix;   //固定参数
24
25    /*分配显示缓冲区*/
26    alloc_dis_buffer(...);
27
28    /*初始化 LCD 控制器*/
29    lcd_init(...);
30
31    /*检查可变参数*/
32    xxxfb_check_var(&info->var, info);
33
34    /*注册 fb_info*/
35    if (register_framebuffer(info) < 0)
36       return  - EINVAL;
37
38     return 0;
39 }
40
41 /* 平台驱动移除函数 */
42 static void __exit xxxfb_remove(...)
43 {
44    struct fb_info *info = dev_get_drv_data(dev);
45
46    if (info)
47    {
48        unregister_framebuffer(info); //注销 fb_info
49        dealloc_dis_buffer(...); //释放显示缓冲区
50        framebuffer_release(info); //注销 fb_info
51   }
52
```

```
53     return 0;
54 }
55
56 /* 帧缓冲设备驱动模块加载与卸载函数  */
57 int __devinit xxxfb_init(void)
58 {
59     return platform_driver_register(&xxxfb_driver); //注册平台设备
60 }
61
62 static void __exit xxxfb_cleanup(void)
63 {
64      platform_driver_unregister(&xxxfb_driver); //注销平台设备
65 }
66
67 module_init(xxxfb_init);
68 module_exit(xxxfb_cleanup);
```

上述代码中第 35 行、48 行成对出现的 register_framebuffer() 和 unregister_framebuffer()分别用于注册和注销帧缓冲设备。

18.5 帧缓冲设备显示缓冲区的申请与释放

在嵌入式系统中，一种常见的方式是直接在 RAM 空间中分配一段显示缓冲区，典型结构如图 18.4 所示。

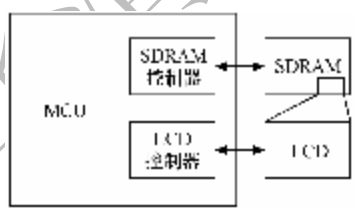


图 18.4　在 RAM 中分配显示缓冲区

在分配显示缓冲区时一定要考虑 cache 的一致性问题，因为系统往往会通过 DMA 方式搬移显示数据。合适的方式是使用 dma_alloc_writecombine()函数分配一段 writecombining 区域，对应的 writecombining 区域由 dma_free_writecombine()函数释放，如代码清单 18.10 所示。

writecombining 意味着“写合并”，它允许写入的数据被合并，并临时保存在写合并缓冲区（WCB）中，直到进行一次 burst 传输而不再需要多次 single 传输。通过 dma_alloc_ writecombine()分配的显示缓冲区不会出现 cache 一致性问题，这一点类似于 dma_alloc_ coherent()。

代码清单 18.10　帧缓冲设备显示缓冲区的分配与释放

```
1  static int __init xxxfb_map_video_memory(struct xxxfb_info *fbi)
2  {
3      fbi->map_size = PAGE_ALIGN(fbi->fb->fix.smem_len + PAGE_SIZE);
4      fbi->map_cpu = dma_alloc_writecombine(fbi->dev, fbi->map_size,
5          &fbi->map_dma,GFP_KERNEL); //分配内存
6
7      fbi->map_size = fbi->fb->fix.smem_len; //显示缓冲区大小
8
9      if (fbi->map_cpu)
10     {
11         memset(fbi->map_cpu, 0xf0, fbi->map_size);
12
13         fbi->screen_dma = fbi->map_dma;
14         fbi->fb->screen_base = fbi->map_cpu;
15           fbi->fb->fix.smem_start = fbi->screen_dma; //赋值 fix 的
smem_start
16   }
17
18      return fbi->map_cpu ? 0 :  - ENOMEM;
19 }
20
21 static inline void xxxfb_unmap_video_memory(struct s3c2410fb_info
*fbi)
22 {
23   //释放显示缓冲区
24         dma_free_writecombine(fbi->dev,fbi->map_size,fbi->map_cpu,
fbi->map_dma);
25 }
```

18.6 帧缓冲设备的参数设置

18.6.1 定时参数

FBI 结构体可变参数 var 中的 left_margin、right_margin、upper_margin、lower_margin、hsync_len 和 vsync_len 直接查 LCD 的数据手册就可以得到，图 18.5 所示为某 LCD 数据手册中直接抓图获得的定时信息。

由图 18.5 可知对该 LCD 而言，var 中各参数的较合适值分别为：left_margin = 104，right_margin =

显示模式	参数	符号	条件	时间			单位
				最小值	典型值	最大值	
常规	Vertical cycle	VP		648	—	—	行
	Vertical data start	VDS	VS+VBP	4	—	—	行
	Vertical Sync Pulse width	VS		2	—	—	行
	Vertical front porch	VFP		4			行
	Vertical Back porch	VBP		2			行
	Vertical blanking period	VBL	VS+VBP+VFP	8	—	—	行
	Vertical active area	VDISP		640			行
	Horizontal cycle	HP		520			点
	Horizontal front porch	HFP		24	—	—	点
	Horizontal Sync Pulse width	HS		8	—	—	点
	Horizontal Back porch	HBP		8			点
	Horizontal Data start	HDS	HS+HBP	36	—	—	点
	Horizontal active area	HDISP		480	—	—	点
	Clock Frequency	fclk		20	TBD	TBD	MHz
		tclk		50	TBD	TBD	ns

图 18.5　LCD 数据手册中定时参数示例

8，upper_margin = 2，lower_margin = 2，hsync_len = 2，vsync_len = 2。在 QVGA 模式下也可类似得到。

18.6.2　像素时钟

FBI 可变参数 var 中的 pixclock 意味着像素时钟，例如，如果为 28.37516 MHz，那么画 1 个像素需要 35242 ps（皮秒）：

```
1/(28.37516E6 Hz) = 35.242E-9 s
```

如果屏幕的分辨率是 640×480，显示一行需要的时间是：

```
640*35.242E-9 s = 22.555E-6 s
```

每条扫描线是 640，但是水平回扫和水平同步也需要时间，假设水平回扫和同步需要 272 个像素时钟，因此，画一条扫描线完整的时间是：

```
(640+272)*35.242E-9 s = 32.141E-6 s
```

可以计算出水平扫描率大约是 31kHz：

```
1/(32.141E-6 s) = 31.113E3 Hz
```

完整的屏幕有 480 线，但是垂直回扫和垂直同步也需要时间，假设垂直回扫和垂直同步需要 49 个象素时钟，因此，画一个完整的屏幕的时间是：

```
(480+49)*32.141E-6 s = 17.002E-3 s
```

可以计算出垂直扫描率大约是 59kHz：

```
1/(17.002E-3 s) = 58.815 Hz
```

这意味着屏幕数据每秒钟大约刷新 59 次。

18.6.3　颜色位域

FBI 可变参数 var 中的 red、green 和 blue 位域的设置直接由显示缓冲区与显示点的对应关系决定，例如，对于 RGB565 模式，查表 18.4，red 占据 5 位，偏移为 11 位；green 占据 6 位，偏移为 5 位；blue 占据 5 位，偏移为 0 位，即：

```
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;
```

18.6.4 固定参数

FBI 固定参数 fix 中的 smem_start 指示帧缓冲设备显示缓冲区的首地址，smem_len 为帧缓冲设备显示缓冲区的大小，计算公式为：

```
    smem_len = max_xres * max_yres * max_bpp
```

即：帧缓冲设备显示缓冲区的大小 = 最大的 *x* 解析度 * 最大的 *y* 解析度 * 最大的 BPP。

18.7 帧缓冲设备驱动的 fb_ops 成员函数

FBI 中的 fp_ops 是使得帧缓冲设备工作所需函数的集合，它们最终与 LCD 控制器硬件打交道。

fb_check_var()用于调整可变参数，并修正为硬件所支持的值；fb_set_par()则根据屏幕参数设置具体读写 LCD 控制器的寄存器以使得 LCD 控制器进入相应的工作状态。

对于 fb_ops 中的 fb_fillrect()、fb_copyarea()和 fb_imageblit()成员函数，通常直接使用对应的通用的 cfb_fillrect()、cfb_copyarea()和 cfb_imageblit()函数即可。cfb_fillrect()函数定义在 drivers/video/cfbfillrect.c 文件中，cfb_copyarea()定义在 drivers/video/cfbcopyarea.c 文件中，cfb_imageblit()定义在 drivers/video/cfbimgblt.c 文件中。

fb_ops 中的 fb_setcolreg()成员函数实现伪颜色表（针对 FB_VISUAL_TRUECOLOR、FB_ VISUAL_DIRECTCOLOR 模式）和颜色表的填充，其模板如代码清单 18.11 所示。

代码清单 18.11 fb_setcolreg()函数模板

```
    1  static int xxxfb_setcolreg(unsigned regno, unsigned red, unsigned
green,
    2      unsigned blue, unsigned transp, struct fb_info *info)
    3  {
    4      struct xxxfb_info *fbi = info->par;
    5      unsigned int val;
    6
    7      switch (fbi->fb->fix.visual)
```

```
8     {
9       case FB_VISUAL_TRUECOLOR:
10        /* 真彩色，设置伪颜色表 */
11        if (regno < 16)
12        {
13           u32 *pal = fbi->fb->pseudo_palette;
14
15          val = chan_to_field(red, &fbi->fb->var.red);
16          val |= chan_to_field(green, &fbi->fb->var.green);
17          val |= chan_to_field(blue, &fbi->fb->var.blue);
18
19          pal[regno] = val;
20        }
21        break;
22
23    case FB_VISUAL_PSEUDOCOLOR:
24        if (regno < 256)
25        {
26            /* RGB565 模式 */
27           val = ((red >> 0) &0xf800);
28           val |= ((green >> 5) &0x07e0);
29           val |= ((blue >> 11) &0x001f);
30
31           writel(val, XXX_TFTPAL(regno));
32           schedule_palette_update(fbi, regno, val);
33        }
34        break;
35      ...
36    }
37
38    return 0;
39 }
```

上述代码第 11 行对 regno < 16 的判断意味着伪颜色表只有 16 个成员，实际上，它们对应 16 种控制台颜色，logo 显示也会使用伪颜色表。

18.8 LCD 设备驱动的读写、mmap 和 ioctl 函数

虽然帧缓冲设备的 file_operations 中的成员函数，即文件操作函数已经由内核在 fbmem.c 文件中实现，一般不再需要驱动工程师修改，但分析这些函数对于巩固字符设备驱动的知识以及加深对帧缓冲设备驱动的理解是大有裨益的。

代码清单 18.12 所示为 LCD 设备驱动的文件操作读写函数的源代码，从代码结构及习惯而言，与本书第二篇所讲解的字符设备驱动完全一致。

代码清单 18.12　帧缓冲设备驱动的读写函数

```
1  static ssize_t fb_read(struct file *file, char __user *buf, size_t
count,
2      loff_t *ppos)
3  {
4      unsigned long p =  *ppos;
5      struct inode *inode = file->f_dentry->d_inode;
6      int fbidx = iminor(inode);
7      struct fb_info *info = registered_fb[fbidx]; //获得 FBI
8      u32 *buffer,  *dst;
9      u32 __iomem *src;
10     int c, i, cnt = 0, err = 0;
11     unsigned long total_size;
12
13     if (!info || !info->screen_base)
14         return  - ENODEV;
15
16     if (info->state != FBINFO_STATE_RUNNING)
17         return  - EPERM;
18
19     if (info->fbops->fb_read)  //如果 fb_ops 中定义了特定的读函数
20         return info->fbops->fb_read(file, buf, count, ppos);
21     /*获得显示缓冲区总的大小*/
22     total_size = info->screen_size;
23
24     if (total_size == 0)
25        total_size = info->fix.smem_len;
26
27     if (p >= total_size)
28        return 0;
29     /*获得有效的读长度*/
30     if (count >= total_size)
31        count = total_size;
32
33     if (count + p > total_size)
34        count = total_size - p;
35     /*分配用于临时存放显示缓冲区数据的 buffer*/
36       buffer = kmalloc((count > PAGE_SIZE) ? PAGE_SIZE : count,
GFP_KERNEL);
37     if (!buffer)
38        return  - ENOMEM;
39
40     src = (u32 __iomem*)(info->screen_base + p);  //获得源地址
41
42     if (info->fbops->fb_sync)
43         info->fbops->fb_sync(info);
```

```
44
45      while (count)
46      { /*读取显示缓冲区中的数据并复制到分配的 buffer*/
47        c = (count > PAGE_SIZE) ? PAGE_SIZE : count;
48        dst = buffer;
49        for (i = c >> 2; i--;)
50            *dst++ = fb_readl(src++);
51        if (c &3)
52        {
53            u8 *dst8 = (u8*)dst;
54            u8 __iomem *src8 = (u8 __iomem*)src;
55
56            for (i = c &3; i--;)
57                *dst8++ = fb_readb(src8++);
58
59            src = (u32 __iomem*)src8;
60        }
61
62        if (copy_to_user(buf, buffer, c))  //复制到用户空间
63        {
64            err =  - EFAULT;
65            break;
66       }
67       *ppos += c;
68       buf += c;
69       cnt += c;
70       count -= c;
71      }
72
73      kfree(buffer);
74
75      return (err) ? err : cnt;
76  }
77
78  static ssize_t fb_write(struct file *file, const char __user *buf,
size_t count,
79      loff_t *ppos)
80  {
81      unsigned long p =  *ppos;
82      struct inode *inode = file->f_dentry->d_inode;
83      int fbidx = iminor(inode);
84      struct fb_info *info = registered_fb[fbidx];
85      u32 *buffer,  *src;
86      u32 __iomem *dst;
87      int c, i, cnt = 0, err = 0;
88      unsigned long total_size;
89
90      if (!info || !info->screen_base)
91          return  - ENODEV;
92
93      if (info->state != FBINFO_STATE_RUNNING)
94          return  - EPERM;
```

```
    95
    96      if (info->fbops->fb_write) //如果 fb_ops 中定义了特定的写函数
    97          return info->fbops->fb_write(file, buf, count, ppos);
    98      /*获得显示缓冲区总的大小*/
    99      total_size = info->screen_size;
    100
    101     if (total_size == 0)
    102         total_size = info->fix.smem_len;
    103
    104     if (p > total_size)
    105         return 0;
    106     /*获得有效的写长度*/
    107     if (count >= total_size)
    108         count = total_size;
    109
    110     if (count + p > total_size)
    111         count = total_size - p;
    112     /*分配用于存放用户空间传过来的显示缓冲区数据的 buffer*/
    113       buffer = kmalloc((count > PAGE_SIZE) ? PAGE_SIZE : count,
GFP_KERNEL);
    114     if (!buffer)
    115         return  - ENOMEM;
    116
    117     dst = (u32 __iomem*)(info->screen_base + p); //要写的显示缓冲区
基地址
    118
    119     if (info->fbops->fb_sync)
    120         info->fbops->fb_sync(info);
    121
    122    while (count)
    123    { /*读取用户空间数据并复制到显示缓冲区*/
    124      c = (count > PAGE_SIZE) ? PAGE_SIZE : count;
    125      src = buffer;
    126
    127      if (copy_from_user(src, buf, c))
    128      {
    129        err =  - EFAULT;
    130        break;
    131      }
    132
    133      for (i = c >> 2; i--;)
    134          fb_writel(*src++, dst++);
    135
    136      if (c &3)
    137      {
    138         u8 *src8 = (u8*)src;
    139         u8 __iomem *dst8 = (u8 __iomem*)dst;
    140
    141       for (i = c &3; i--;)
    142          fb_writeb(*src8++, dst8++);
    143
    144       dst = (u32 __iomem*)dst8;
    145     }
    146
    147     *ppos += c;
```

```
    buf += c;
    cnt += c;
    count -= c;
  }

  kfree(buffer);

  return (err) ? err : cnt;
}
```

file_operations 中的 mmap()函数非常关键，它将显示缓冲区映射到用户空间，从而使得用户空间可以直接操作显示缓冲区而省去一次用户空间到内核空间的内存复制过程，提高效率，其源代码如代码清单 18.13 所示。

代码清单 18.13　帧缓冲设备驱动的 mmap 函数

```
static int fb_mmap(struct file *file, struct vm_area_struct *vma)
{
    int fbidx = iminor(file->f_dentry->d_inode);
    struct fb_info *info = registered_fb[fbidx];
    struct fb_ops *fb = info->fbops;
    unsigned long off;

    if (vma->vm_pgoff > (~0UL >> PAGE_SHIFT))
        return  - EINVAL;
   off = vma->vm_pgoff << PAGE_SHIFT;
    if (!fb)
        return  - ENODEV;
    if (fb->fb_mmap)  //FBI 中实现了 mmap，则调用 FBI 的 mmap
    {
       int res;
       lock_kernel();
       res = fb->fb_mmap(info, vma);
       unlock_kernel();
       return res;
  }

  /* !sparc32... */
  lock_kernel();

  /* 映射帧缓冲设备的显示缓冲区 */
  start = info->fix.smem_start; //开始地址
  len = PAGE_ALIGN((start &~PAGE_MASK) + info->fix.smem_len); //长
度
  if (off >= len)
  {
     /* 内存映射的 I/O */
     off -= len;
     if (info->var.accel_flags)
     {
        unlock_kernel();
        return  - EINVAL;
```

```
    }
    start = info->fix.mmio_start;
    len = PAGE_ALIGN((start &~PAGE_MASK) + info->fix.mmio_len);
}
unlock_kernel();
start &= PAGE_MASK;
if ((vma->vm_end - vma->vm_start + off) > len)
    return  - EINVAL;
off += start;
vma->vm_pgoff = off >> PAGE_SHIFT;
/* 这是 1 个 I/O 映射 */
vma->vm_flags |= VM_I/O | VM_RESERVED;
vma->vm_page_prot = pgprot_writecombine(vma->vm_page_prot);

if (io_remap_pfn_range(vma, vma->vm_start, off >> PAGE_SHIFT,
    vma->vm_end -vma->vm_start, vma->vm_page_prot)) //映射
  return  - EAGAIN;

return 0;
}
```

fb_ioctl()函数最终实现对用户 I/O 控制命令的执行，这些命令包括 FBIOGET_VSCREENINFO（获得可变的屏幕参数）、FBIOPUT_VSCREENINFO（设置可变的屏幕参数）、FBIOGET _FSCREENINFO（获得固定的屏幕参数设置，注意，固定的屏幕参数不能由用户设置）、FBIOPUTCMAP（设置颜色表）、FBIOGETCMAP（获得颜色表）等。代码清单 18.14 所示为帧缓冲设备 ioctl()函数的源代码。

代码清单 18.14 帧缓冲设备驱动的 ioctl 函数

```
static int fb_ioctl(struct inode *inode, struct file *file, unsigned
int cmd,
    unsigned long arg)
{
    int fbidx = iminor(inode);
    struct fb_info *info = registered_fb[fbidx];
    struct fb_ops *fb = info->fbops;
    struct fb_var_screeninfo var;
    struct fb_fix_screeninfo fix;
    struct fb_con2fbmap con2fb;
    struct fb_cmap_user cmap;
    struct fb_event event;
    void __user *argp = (void __user*)arg;
    int i;

    if (!fb)
       return  - ENODEV;
    switch (cmd)
    {
      case FBIOGET_VSCREENINFO:// 获得可变的屏幕参数
          return copy_to_user(argp, &info->var, sizeof(var)) ?  -
EFAULT: 0;
      case FBIOPUT_VSCREENINFO: //设置可变的屏幕参数
        if (copy_from_user(&var, argp, sizeof(var)))
            return  - EFAULT;
        acquire_console_sem();
```

```
    25          info->flags |= FBINFO_MISC_USEREVENT;
    26          i = fb_set_var(info, &var);
    27          info->flags &= ~FBINFO_MISC_USEREVENT;
    28          release_console_sem();
    29          if (i)
    30            return i;
    31          if (copy_to_user(argp, &var, sizeof(var)))
    32            return  - EFAULT;
    33          return 0;
    34      case FBIOGET_FSCREENINFO: //获得固定的屏幕参数设置
    35            return copy_to_user(argp, &info->fix, sizeof(fix)) ?  -
EFAULT: 0;
    36      case FBIOPUTCMAP:  //设置颜色表
    37          if (copy_from_user(&cmap, argp, sizeof(cmap)))
    38             return  - EFAULT;
    39          return (fb_set_user_cmap(&cmap, info));
    40      case FBIOGETCMAP:  //获得颜色表
    41          if (copy_from_user(&cmap, argp, sizeof(cmap)))
    42             return  - EFAULT;
    43          return fb_cmap_to_user(&info->cmap, &cmap);
    44      case FBIOPAN_DISPLAY:
    45          if (copy_from_user(&var, argp, sizeof(var)))
    46             return  - EFAULT;
    47          acquire_console_sem();
    48          i = fb_pan_display(info, &var);
    49          release_console_sem();
    50          if (i)
    51             return i;
    52          if (copy_to_user(argp, &var, sizeof(var)))
    53             return  - EFAULT;
    54          return 0;
    55      case FBIO_CURSOR:
    56          return  - EINVAL;
    57      case FBIOGET_CON2FBMAP:
    58          if (copy_from_user(&con2fb, argp, sizeof(con2fb)))
    59             return  - EFAULT;
    60          if (con2fb.console < 1 || con2fb.console > MAX_NR_CONSOLES)
    61             return  - EINVAL;
    62          con2fb.framebuffer =  - 1;
    63          event.info = info;
    64          event.data = &con2fb;
    65                                  notifier_call_chain(&fb_notifier_list,
FB_EVENT_GET_CONSOLE_MAP, &event);
    66            return copy_to_user(argp, &con2fb, sizeof(con2fb)) ?  -
EFAULT: 0;
    67      case FBIOPUT_CON2FBMAP:
    68          if (copy_from_user(&con2fb, argp, sizeof(con2fb)))
    69            return  - EFAULT;
    70          if (con2fb.console < 0 || con2fb.console > MAX_NR_CONSOLES)
    71            return  - EINVAL;
    72         if (con2fb.framebuffer < 0 || con2fb.framebuffer >= FB_MAX)
    73            return  - EINVAL;
```

```
74        #ifdef CONFIG_KMOD
75           if (!registered_fb[con2fb.framebuffer])
76              try_to_load(con2fb.framebuffer);
77         #endif /* CONFIG_KMOD */
78         if (!registered_fb[con2fb.framebuffer])
79            return  - EINVAL;
80         event.info = info;
81         event.data = &con2fb;
82         return notifier_call_chain(&fb_notifier_list,
83            FB_EVENT_SET_CONSOLE_MAP, &event);
84      case FBIOBLANK:
85         acquire_console_sem();
86         info->flags |= FBINFO_MISC_USEREVENT;
87         i = fb_blank(info, arg);
88         info->flags &= ~FBINFO_MISC_USEREVENT;
89         release_console_sem();
90         return i;
91      default:
92         if (fb->fb_ioctl == NULL)
93            return  - EINVAL;
94         return fb->fb_ioctl(info, cmd, arg);
95   }
96 }
```

18.9 帧缓冲设备的用户空间访问

通过/dev/fbns，应用程序可进行的针对帧缓冲设备的操作主要有如下几种。

- 读/写 dev/fbn：相当于读/写屏幕缓冲区。例如用 cp /dev/fb0 tmp 命令可将当前屏幕的内容复制到一个文件中，而命令 cp tmp > /dev/fb0 则将图形文件 tmp 显示在屏幕上。
- 映射操作：对于帧缓冲设备，可通过 mmap()映射操作将屏幕缓冲区的物理地址映射到用户空间的一段虚拟地址中，之后用户就可以通过读/写这段虚拟地址访问屏幕缓冲区，在屏幕上绘图了。而且若干个进程可以映射到同一个显示缓冲区。实际上，使用帧缓冲设备的应用程序都是通过映射操作来显示图形的。
- I/O 控制：对于帧缓冲设备，对设备文件的 ioctl()操作可读取/设置显示设备及屏幕的参数，如分辨率、显示颜色数、屏幕大小等。

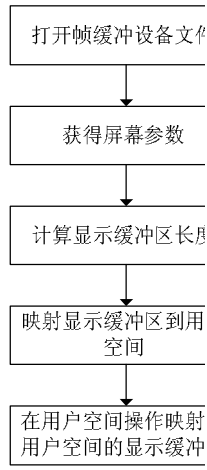


图 18.6 用户空间访帧缓冲设备流程

如图 18.6 所示，在应用程序中，操作/dev/fbn 的一般步骤如下。

（1）打开/dev/fbn 设备文件。

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

（3）将屏幕缓冲区映射到用户空间。

（4）映射后就可以直接读/写屏幕缓冲区，进行绘图和图片显示了。

代码清单 18.15 所示为一段用户空间访问帧缓冲设备显示缓冲区的范例，包含打开和关闭帧缓冲设备、得到和设置可变参数、得到固定参数、生成与 BPP 对应的帧缓冲数据及填充显示缓冲区。

代码清单 18.15　用户空间访问帧缓冲设备显示缓冲区范例

```
1  /*打开帧缓冲设备*/
2  int openFB(const char *name)
3  {
4     int fh;
5     char *dev;
6
7     if (name == NULL)
8     {
9        dev = getenv("FRAMEBUFFER");
10       if (dev)
11          name = dev;
12       else
13          name = DEFAULT_FRAMEBUFFER;
14    }
15
16    if ((fh = open(name, O_WRONLY)) ==  - 1)
17    {
18       fprintf(stderr, "open %s: %s\n", name, strerror(errno));
19       exit(1);
20    }
21    return fh;
22 }
23
24 /*关闭帧缓冲设备*/
25 void closeFB(int fh)
26 {
27    close(fh);
28 }
29
30 /*得到屏幕可变参数*/
31 void getVarScreenInfo(int fh, struct fb_var_screeninfo *var)
32 {
33   if (ioctl(fh, FBIOGET_VSCREENINFO, var))
34    {
35            fprintf(stderr,  "ioctl  FBIOGET_VSCREENINFO:  %s\n",
strerror(errno));
36       exit(1);
37    }
38 }
39
40 /*设置屏幕可变参数*/
41 void setVarScreenInfo(int fh, struct fb_var_screeninfo *var)
42 {
43    if (ioctl(fh, FBIOPUT_VSCREENINFO, var))
44    {
45             fprintf(stderr,  "ioctl  FBIOPUT_VSCREENINFO:  %s\n",
strerror(errno));
46       exit(1);
47    }
48 }
49
50 /*得到屏幕固定参数*/
51 void getFixScreenInfo(int fh, struct fb_fix_screeninfo *fix)
52 {
```

```
    53     if (ioctl(fh, FBIOGET_FSCREENINFO, fix))
    54     {
    55             fprintf(stderr, "ioctl FBIOGET_FSCREENINFO: %s\n",
strerror(errno));
    56         exit(1);
    57   }
    58 }
    59
    60 /*构造颜色表*/
    61 void make332map(struct fb_cmap *map)
    62 {
    63     int rs, gs, bs, i;
    64     int r = 8, g = 8, b = 4;
    65
    66     map->red = red;
    67     map->green = green;
    68     map->blue = blue;
    69
    70     rs = 256 / (r - 1);
    71     gs = 256 / (g - 1);
    72     bs = 256 / (b - 1);
    73
    74     for (i = 0; i < 256; i++)
    75     {
    76         map->red[i] = (rs *((i / (g *b)) % r)) *255;
    77         map->green[i] = (gs *((i / b) % g)) *255;
    78         map->blue[i] = (bs *((i) % b)) *255;
    79   }
    80 }
    81
    82 /*设置颜色表*/
    83 void set8map(int fh, struct fb_cmap *map)
    84 {
    85     if (ioctl(fh, FBIOPUTCMAP, map) < 0)
    86     {
    87         fprintf(stderr, "Error putting colormap");
    88         exit(1);
    89   }
    90 }
    91
    92 /*构造并设置颜色表*/
    93 void set332map(int fh)
    94 {
    95     make332map(&map332);
    96     set8map(fh, &map332);
    97 }
    98
    99 /*获得颜色表*/
   100 void get8map(int fh, struct fb_cmap *map)
   101 {
   102     if (ioctl(fh, FBIOGETCMAP, map) < 0)
   103     {
   104         fprintf(stderr, "Error getting colormap");
```

```
    105         exit(1);
    106   }
    107 }
    108 /*组成 BPP=16 时的缓冲数据*/
    109 inline unsigned short make16color(unsigned char r, unsigned char g,
unsigned char b)
    110 {
    111         return ((((r >> 3) & 31) << 11) |
    112                         (((g >> 2) & 63) << 5)  |
    113                         ((b >> 3) & 31) );
    114 }
    115
    116 /*把 RGB 数据按照屏幕的 BPP 调整*/
    117 void *convertRGB2FB(int fh, unsigned char *rgbbuff, unsigned long
count, int
    118     bpp, int *cpp)
    119 {
    120     unsigned long i;
    121     void *fbbuff = NULL;
    122     unsigned char *c_fbbuff;
    123     unsigned short *s_fbbuff;
    124     unsigned int *i_fbbuff;
    125
    126     switch (bpp)
    127     {
    128         case 8:
    129           //...
    130         case 15:
    131           //  ...
    132         case 16:
    133            *cpp = 2;
    134                         s_fbbuff  =  (unsigned  short*)malloc(count
*sizeof(unsigned short));
    135             for (i = 0; i < count; i++)
    136                s_fbbuff[i] = make16color(rgbbuff[i *3], rgbbuff[i
*3+1], rgbbuff
    137                [i *3+ 2]);
    138             fbbuff = (void*)s_fbbuff;
    139             break;
    140          case 24:
    141          case 32:
    142            //  ...
    143          default:
    144             fprintf(stderr, "Unsupported video mode! You've got:
%dbpp\n", bpp);
    145             exit(1);
    146     }
    147     return fbbuff;
    148 }
    149
    150 /*写显示缓冲区*/
    151 void blit2FB(int fh, void *fbbuff, unsigned int pic_xs, unsigned
int pic_ys,
    152       unsigned int scr_xs, unsigned int scr_ys, unsigned int xp,
unsigned int yp,
    153      unsigned int xoffs, unsigned int yoffs, int cpp)
```

```
154 {
155     int i, xc, yc;
156     unsigned char *cp;
157     unsigned short *sp;
158     unsigned int *ip;
159      cp = (unsigned char*)sp = (unsigned short*)ip = (unsigned
int*)fbbuff;
160
161     xc = (pic_xs > scr_xs) ? scr_xs : pic_xs;
162     yc = (pic_ys > scr_ys) ? scr_ys : pic_ys;
163
164     switch (cpp)
165     {
166       case 1:
167         ...
168       case 2:  //BPP=16
169         for (i = 0; i < yc; i++)
170         {
171                 lseek(fh, ((i + yoffs) *scr_xs + xoffs) *cpp,
SEEK_SET);
172             write(fh, sp + (i + yp) *pic_xs + xp, xc *cpp);
173         }
174         break;
175       case 4:
176         ...
177   }
178 }
179
180
181 /*帧缓冲设备显示*/
182 void fb_display(unsigned char *rgbbuff, int x_size, int y_size, int
x_pan, int
183    y_pan, int x_offs, int y_offs)
184 {
185    struct fb_var_screeninfo var;
186    unsigned short *fbbuff = NULL;
187    int fh =  - 1, bp = 0;
188
189     /* 打开帧缓冲设备 */
190     fh = openFB(NULL);
191
192     /* 获得可变参数 */
193     getVarScreenInfo(fh, &var);
194
195     /*校正 panning */
196     if (x_pan > x_size - var.xres)
197        x_pan = 0;
198     if (y_pan > y_size - var.yres)
199        y_pan = 0;
200     /* 校正 offset */
201     if (x_offs + x_size > var.xres)
202        x_offs = 0;
203     if (y_offs + y_size > var.yres)
```

```
    204         y_offs = 0;
    205
    206     /* 将 RGB 数据转换为与 var 位域对应的数据并填充到显示缓冲区 */
    207           fbbuff = convertRGB2FB(fh, rgbbuff, x_size *y_size,
var.bits_per_pixel, &bp);
    208     blit2FB(fh, fbbuff, x_size, y_size, var.xres, var.yres, x_pan,
y_pan, x_offs,
    209         y_offs, bp);
    210     free(fbbuff);
    211
    212     /* 关闭帧缓冲设备 */
    213     closeFB(fh);
    214 }
```

18.10 Linux 图形用户界面

18.10.1 Qt-X11/QtEmbedded/Qtopia

Qt 是 Trolltech（奇趣）公司所开发的一个跨平台 FrameWork 环境，它采用类似 C++的语法，在 Microsoft Windows 95/98/2000、Microsoft Windows NT、MacOS X、Linux、Solaris、HP-UX、Tru64 (Digital UNIX)、Irix、FreeBSD、BSD/OS、SCO、AIX 等许多平台上都可执行。

Trolltech 也针对嵌入式环境推出了 Qt/Embedded 产品。与桌面版本不同，Qt/Embedded 已经直接取代掉 X Server 及 X Library 等角色，所有的功能全部整合在一起。Qt/Embedded 是一个专门为小型设备提供图形用户界面的应用框架和窗口系统，它提供了丰富的窗口小部件（Widgets），并且还支持窗口部件的定制，因此它可以为用户提供漂亮的图形界面，许多基于 Qt 的 X Window 程序可以非常方便地移植到 Qt/Embedded 版本上。

Qt/Embedded 以原始 Qt 为基础，做了许多针对嵌入式环境的调整。Qt/Embedded 通过 Qt 的 API 与 Linux 的 I/O 直接交互，同 Qt/X11 相比，Qt/Embedded 不需要一个 X 服务器或是 Xlib 库，它在底层抛弃了 Xlib，而是采用帧缓冲作为底层图形接口，Qt/Embedded 的应用程序可以直接写显示缓冲区，无须借助繁琐的 Xlib/Server 系统，如图 18.7 所示。

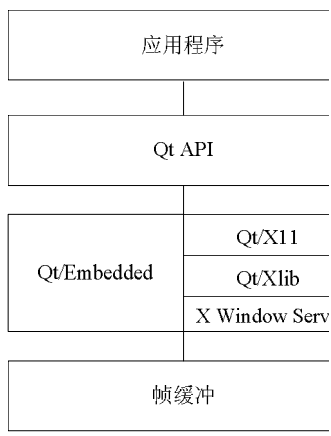


图 18.7 Qt/Embedded 与 Qt 的区

Qtopia 是建立在 Qt/Embedded 上的一种开放源代码窗口系统，它与实际的产品相似，专门针对 PDA、SmartPhone 这类运行嵌入式 Linux 的移动计算设备和手持设备而开发的。Trolltech 还发布了一款供应用开发人员使用的 Linux 手机“Qtopia Greenphone”。

在宿主机上可通过 qvfb（虚拟帧缓冲）来模拟帧缓冲。qvfb 是 X 窗口用来运行和测试 Qtopia 应用程序的系统程序，它使用了共享存储区域（虚拟的帧缓冲）来模拟帧缓冲并且在一个窗口中模拟一个应用来显示帧缓冲，允许我

们在桌面及其上开发 Qt 嵌入式程序。显示的区域被周期性地改变和更新，通过指定显示设备的宽度和颜色深度，虚拟出来的缓冲帧和物理的显示设备在每个像素上保持一致，从而方便程序的调试。

信号（Signal）和插槽（Slot）是 Qt 中一种用于对象间通信的调用机制，不同于传统的函数回调方式，信号和插槽是 Qt 中非常有特色的地方，是 Qt 编程区别于其他编程的标志。信号和插槽不是标准 C++功能，C++编译器不能理解这些语句，必须经过特殊的工具对象编辑器 MOC（Meta Object Compiler）将源代码中创建信号和插槽的语句翻译成 C++编译器能够理解的代码。

Qt 的窗口在事件发生后会激发信号。例如，一个按钮被点击时会激发一个“clicked”信号。程序员通过建立一个函数（称做一个插槽），然后调用 connect()函数把这个插槽和一个信号连接起来，这样就完成了一个事件和响应代码的连接。信号与插槽机制并不要求类之间互相知道细节，这样就可以相对容易的开发出代码可高重用的类。

例如，如果一个退出按钮的 clicked()信号被连接到了一个应用的退出函数 quit()插槽。那么一个用户点击退出键将使应用程序终止运行，完成上述连接过程的代码如下。

```
    connect( button, SIGNAL(clicked()), qApp, SLOT(quit()) );
```

代码清单 18.16 的应用程序创建一个 hello 窗口，该窗口显示一个动态字符串“Hello，World”，程序中添加了一个 Qtimer 定时器实例，以周期性刷新屏幕，从而得到动画的效果。

代码清单 18.16　Qt/Embedded 应用程序范例

```
    1   /*****************************************************
    2    ** 以下是 hello.h 的代码
    3    *****************************************************/
    4   #ifndef HELLO_H
    5      #define HELLO_H
    6      #include <qvariant.h>
    7      #include <qwidget.h>
    8      class QVBoxLayout;
    9      class QHBoxLayout;
    10     class QGridLayout;
    11     class Hello: public QWidget
    12     {
    13        Q_OBJECT public:
    14          Hello(QWidget *parent = 0, const char *name = 0, WFlags
fl = 0);
    15          ~Hello();
    16          //以下是手动添加的代码
    17          signals: void clicked();
    18         protected:
    19            void mouseReleaseEvent(QMouseEvent*);
    20            void paintEvent(QPaintEvent*);
    21            private slots: void animate();
```

```
    22          private:
    23              QString t;
    24              int b;
    25       };
    26    #endif // HELLO_H
    27
    28  /*************************************************************
    29   ** 以下是 hello.cpp 源代码
    30   *************************************************************/
    31  #include "hello.h"
    32  #include <qlayout.h>
    33  #include <qvariant.h>
    34  #include <qtooltip.h>
    35  #include <qwhatsthis.h>
    36  #include <qpushbutton.h>
    37  #include <qtimer.h>
    38  #include <qpainter.h>
    39  #include <qpixmap.h>
    40  /* 构造一个 Hello 窗口 */
    41   Hello::Hello(QWidget *parent, const char *name, WFlags fl):
QWidget(parent,
    42     name, fl)
    43  {
    44      if (!name)
    45          setName("Hello");
    46      resize(240, 320);
    47      setMinimumSize(QSize(240, 320));
    48      setMaximumSize(QSize(240, 320));
    49      setSizeIncrement(QSize(240, 320));
    50      setBaseSize(QSize(240, 320));
    51      QPalette pal;
    52      QColorGroup cg;
    53      cg.setColor(QColorGroup::Foreground, black);
    54      cg.setColor(QColorGroup::Button, QColor(192, 192, 192));
    55      cg.setColor(QColorGroup::Light, white);
    56      cg.setColor(QColorGroup::Midlight, QColor(223, 223, 223));
    57      cg.setColor(QColorGroup::Dark, QColor(96, 96, 96));
    58      cg.setColor(QColorGroup::Mid, QColor(128, 128, 128));
    59      cg.setColor(QColorGroup::Text, black);
    60      cg.setColor(QColorGroup::BrightText, white);
    61      cg.setColor(QColorGroup::ButtonText, black);
    62      cg.setColor(QColorGroup::Base, white);
    63      cg.setColor(QColorGroup::Background, white);
    64      cg.setColor(QColorGroup::Shadow, black);
    65      cg.setColor(QColorGroup::Highlight, black);
    66      cg.setColor(QColorGroup::HighlightedText, white);
    67      pal.setActive(cg);
    68      cg.setColor(QColorGroup::Foreground, black);
    69      cg.setColor(QColorGroup::Button, QColor(192, 192, 192));
    70      cg.setColor(QColorGroup::Light, white);
    71      cg.setColor(QColorGroup::Midlight, QColor(220, 220, 220));
    72      cg.setColor(QColorGroup::Dark, QColor(96, 96, 96));
    73      cg.setColor(QColorGroup::Mid, QColor(128, 128, 128));
    74      cg.setColor(QColorGroup::Text, black);
    75      cg.setColor(QColorGroup::BrightText, white);
```

```
    76      cg.setColor(QColorGroup::ButtonText, black);
    77      cg.setColor(QColorGroup::Base, white);
    78      cg.setColor(QColorGroup::Background, white);
    79      cg.setColor(QColorGroup::Shadow, black);
    80      cg.setColor(QColorGroup::Highlight, black);
    81      cg.setColor(QColorGroup::HighlightedText, white);
    82      pal.setInactive(cg);
    83      cg.setColor(QColorGroup::Foreground, QColor(128, 128, 128));
    84      cg.setColor(QColorGroup::Button, QColor(192, 192, 192));
    85      cg.setColor(QColorGroup::Light, white);
    86      cg.setColor(QColorGroup::Midlight, QColor(220, 220, 220));
    87      cg.setColor(QColorGroup::Dark, QColor(96, 96, 96));
    88      cg.setColor(QColorGroup::Mid, QColor(128, 128, 128));
    89      cg.setColor(QColorGroup::Text, black);
    90      cg.setColor(QColorGroup::BrightText, white);
    91      cg.setColor(QColorGroup::ButtonText, QColor(128, 128, 128));
    92      cg.setColor(QColorGroup::Base, white);
    93      cg.setColor(QColorGroup::Background, white);
    94      cg.setColor(QColorGroup::Shadow, black);
    95      cg.setColor(QColorGroup::Highlight, black);
    96      cg.setColor(QColorGroup::HighlightedText, white);
    97      pal.setDisabled(cg);
    98      setPalette(pal);
    99      QFont f(font());
    100     f.setFamily("adobe-helvetica");
    101     f.setPointSize(29);
    102     f.setBold(TRUE);
    103     setFont(f);
    104     setCaption(tr(""));
    105     t = "Hello,World";
    106     b = 0;
    107     QTimer *timer = new QTimer(this);   //创建定时器
    108     connect(timer, SIGNAL(timeout()), SLOT(animate()));  //连接信号
和插槽
    109     timer->start(40);
    110 }
    111
    112 /* 销毁对象，释放任何被分配的资源 */
    113 Hello::~Hello(){}
    114
    115 /* 每次定时器到期后调用插槽 */
    116 void Hello::animate()
    117 {
    118     b = (b + 1) &15;
    119     repaint(FALSE);   //重绘
    120 }
    121
    122 /*    处理 hello 窗口的鼠标按钮释放事件 */
    123 void Hello::mouseReleaseEvent(QMouseEvent *e)
    124 {
    125     if (rect().contains(e->pos()))
    126         emit clicked(); //激活 clicked()信号
```

```
127 }
128
129 /* 处理 hello 窗口的重绘事件 */
130 void Hello::paintEvent(QPaintEvent*)
131 {
132    static int sin_tbl[16] =
133    {
134       0, 38, 71, 92, 100, 92, 71, 38, 0, -38, -71, -92, -100, -92,
135      -71, -38
136    };
137    if (t.isEmpty())
138       return ;
139    // 1: 计算尺寸、位置
140    QFontMetrics fm = fontMetrics();
141    int w = fm.width(t) + 20;
142    int h = fm.height() *2;
143    int pmx = width() / 2-w / 2;
144    int pmy = height() / 2-h / 2;
145    // 2: 创建 pixmap，用窗口背景填充它
146    QPixmap pm(w, h);
147    pm.fill(this, pmx, pmy);
148    // 3: 绘制 pixmap
149    QPainter p;
150    int x = 10;
151    int y = h / 2+fm.descent();
152    int i = 0;
153    p.begin(&pm);
154    p.setFont(font());
155    while (!t[i].isNull())
156    {
157      int i16 = (b + i) &15;
158      p.setPen(QColor((15-i16) *16, 255, 255, QColor::Hsv));
159      p.drawText(x, y - sin_tbl[i16] *h / 800, t.mid(i, 1), 1);
160      x += fm.width(t[i]);
161      i++;
162   }
163   p.end();
164   // 4:复制 pixmap 到 Hello 窗口
165   bitBlt(this, pmx, pmy, &pm);
166 }
167
168 /*****************************************************
169  ** 以下是 main.cpp 的源代码
170  *****************************************************/
171 #include "hello.h"
172 #include <qapplication.h>
173 /*
174 The program starts here. It parses the command line and builds a
message
175 string to be displayed by the Hello widget.
176  */
177 #define QT_NO_WIZARD
178 int main(int argc, char **argv)
179 {
180     QApplication a(argc, argv);
```

```
    Hello dlg;
    QObject::connect(&dlg, SIGNAL(clicked()), &a, SLOT(quit()));
    a.setMainWidget(&dlg);
    dlg.show();
    return a.exec();
}
```

18.10.2 Microwindows/Nano-X

Microwindows 是嵌入式系统中广为使用的一种图形用户接口，其官方网站是 http://www. microwindows.org，Microwindows 完全支持 Linux 的帧缓冲技术。这个项目的早期目标是在嵌入式 Linux 平台上提供和普通个人电脑上类似的图形用户界面。作为 PC 上 X-Windows 的替代品，Microwindows 提供了和 X-Windows 类似的功能，但是占用的内存要少得多，根据用户的配置，Microwindows 占用的内存资源范围为 60KB～100KB。Microwindows 支持多种外部设备得输入，包括液晶显示器、鼠标和键盘等。

Microwindows 起源于 NanoGUI 项目，早期 Microwindows 有两个版本，一个版本包含了一组和微软的 WIN32 图形用户接口相似的 API，这个版本就是 Microwindows 版本；另外一个版本是基于 X-Windows 的一组 Xlib 风格的 API 函数库，这个版本允许 X11 的二进制代码直接在 Micro Windows 的 Nanx-X 服务器上运行，称之为 Nano-X。

Microwindows 的核心基于显示设备接口，因此可移植性很好，Microwindows 有自己的帧缓冲，并不局限于 Linux 开发平台，在 eCos、FreeBSD、RTEMS 等操作系统上都能很好地运行。此外，Microwindows 能在宿主机上仿真目标机，这一点与 Qt/Embedded 类似，可大大加快开发速度。Microwindows 的几个运行界面如图 18.8 所示。

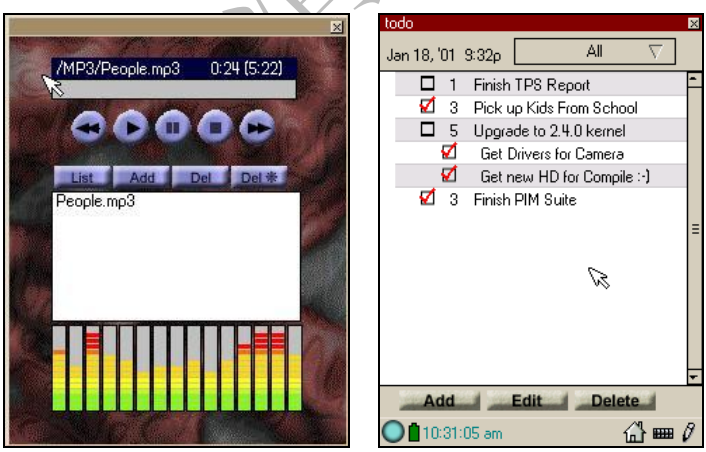


图 18.8 Microwindows 运行界面

Microwindows 支持新的 Linux 内核帧缓冲结构，提供每像素 1、2、4、8、16、24 和 32 位的支持，支持彩色显示和灰度显示，其中彩色显示包括真彩色（每像素 15、16 和 32 位）和调色板（每像素 1、2、4 和 8 位）两种模式。在彩色显示模式下，所有的颜色用 RGB 格式给出，系统再将它转换成与之最相似的可显示颜色，而在单色模式下中则是转

换成不同的灰度级。Microwindows 支持窗口覆盖和子窗口概念、完全的窗口和客户区剪切、比例和固定字体，还提供了字体和位图文件处理工具。

Microwindows 采用分层设计方法。在最底层，屏幕、鼠标/触摸屏以及键盘驱动程序提供了对物理设备访问的能力。在中间层，实现了一个可移植的图形引擎，支持行绘制、区域填充、剪切以及颜色模型等。在上层，实现多种 API 以适应不同的应用环境。

代码清单 18.17 所示为一个简单的 Microwindows 应用程序，它基于 Nano-X API 编写，创建一个窗口并显示“Hello World”，如图 18.9 所示。

代码清单 18.17　Microwindows 应用程序范例

```
#include <stdio.h>
#include <microwin/nano-X.h>

GR_WINDOW_ID wid;
GR_GC_ID gc;

void event_handler(GR_EVENT *event);

int main(void)
{
    if (GrOpen() < 0)
    {
        fprintf(stderr, "GrOpen failed");
        exit(1);
  }

  gc = GrNewGC();
  GrSetGCForeground(gc, 0xFF0000);
  //创建窗口
  wid = GrNewWindowEx(GR_WM_PROPS_APPFRAME |
                          GR_WM_PROPS_CAPTION |
                          GR_WM_PROPS_CLOSEBOX,
                            "jollen.org", GR_ROOT_WINDOW_ID, 0, 0,
200,
                          200,0xFFFFFF);
  //选择事件
  GrSelectEvents(wid, GR_EVENT_MASK_CLOSE_REQ|GR_EVENT_MASK_EXPOSURE);

  GrMapWindow(wid);
  GrMainLoop(event_handler); //挂接事件处理函数
}

void event_handler(GR_EVENT *event)
{
    switch (event->type)
    {
      case GR_EVENT_TYPE_EXPOSURE: //显示文本
        GrText(wid, gc, 50, 50, "Hello World",  - 1, GR_TFASCII);
        break;
      case GR_EVENT_TYPE_CLOSE_REQ:   //关闭
        GrClose();
```

```
41          exit(0);
42        default:
43          break;
44    }
45 }
```

图 18.9　Microwindows 使用范例

18.10.3　MiniGUI

MiniGUI 是由北京飞漫软件技术有限公司开发的面向实时嵌入式系统的轻量级图形用户界面支持系统，1999 年年初遵循 GPL 条款发布第一个版本以来，已广泛应用于手持信息终端、机顶盒、工业控制系统及工业仪表、彩票机、金融终端等产品和领域。目前，MiniGUI 已成为跨操作系统的图形用户界面支持系统，可在 Linux/uClinux、eCos、uC/OS-II、VxWorks 等操作系统上运行，已验证的硬件平台包括 Intel x86、ARM（ARM7/AMR9/StrongARM/xScale）、PowerPC、MIPS 和 M68K（DragonBall/ColdFire）等。

作为操作系统和应用程序之间的中间件，MiniGUI 将底层操作系统及硬件平台的差别隐藏了起来，并对上层应用程序提供了一致的功能特性，这些功能特性如下。

- 完备的多窗口机制和消息传递机制。
- 常用的控件类，包括静态文本框、按钮、单行和多行编辑框、列表框、组合框、进度条、属性页、工具栏、拖动条、树型控件、月历控件等。
- 对话框和消息框支持以及其他 GUI 元素，包括菜单、加速键、插入符、定时器等。
- 界面皮肤支持。用户可通过皮肤支持获得外观非常华丽的图形界面。
- 通过两种不同的内部软件结构支持低端显示设备（比如单色 LCD）和高端显示设备（比如彩色显示器），前者小巧灵活，而后者在前者的基础上提供了更加强大的图形功能。
- Windows 的资源文件支持，如位图、图标、光标等。

- 各种流行图像文件的支持，包括 JPEG、GIF、PNG、TGA、BMP 等。
- 多字符集和多字体支持，目前支持 ISO8859-1～ISO8859-15、GB2312、GBK、GB18030、BIG5、EUC-JP、Shift-JIS、EUC-KR、UNICODE 等字符集，支持等宽点阵字体、变宽点阵字体、Qt/Embedded 使用的嵌入式字体 QPF、TrueType 以及 AdobeType1 等矢量字体。
- 多种键盘布局的支持。MiniGUI 除支持常见的美式 PC 键盘布局之外，还支持法语、德语等语种的键盘布局。
- 简体中文（GB2312）输入法支持，包括内码、全拼、智能拼音等。用户还可以从飞漫软件获得五笔、自然码等输入法支持。
- 针对嵌入式系统的特殊支持，包括一般性的 I/O 流操作，字节序相关函数等。

如图 18.10 所示，基于 MiniGUI 的应用程序一般通过 ANSI C 库以及 MiniGUI 自身提供的 API 来实现自己的功能；MiniGUI 中的可移植层可将特定操作系统及底层硬件的细节隐藏起来，而上层应用程序则无须关心底层的硬件平台输出和输入设备。

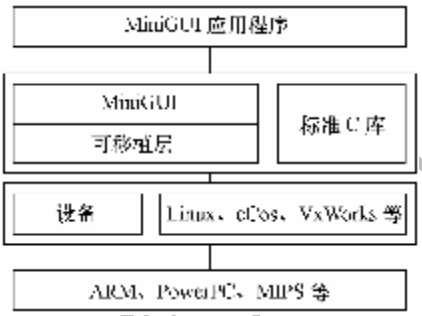


图 18.10　MiniGUI 和嵌入式操作系统的关系

因此，为了适合不同的操作系统环境，MiniGUI 可配置成以下 3 种运行模式。

1．MiniGUI-Threads

运行在 MiniGUI-Threads 上的程序可以在不同的线程中建立多个窗口，但所有的窗口在一个进程或者地址空间中运行。这种运行模式非常适合于大多数传统意义上的嵌入式操作系统，比如 uC/OS-II、eCos、VxWorks、pSOS 等。当然，在 Linux 和 uClinux 上，MiniGUI 也能以 MiniGUI-Threads 模式运行。

2．MiniGUI-Processes

和 MiniGUI-Threads 相反，MiniGUI-Processes 上的每个程序是单独的进程，每个进程也可以建立多个窗口，并且实现了多进程窗口系统。MiniGUI-Processes 适合于具有完整 UNIX 特性的嵌入式操作系统，比如嵌入式 Linux、VxWorks 等。

在 Linux 操作系统上运行的 MiniGUI 1.6.x 版本中，该运行模式称为“MiniGUI-Lite”。MiniGUI-Lite 为多进程环境的 Linux 操作系统提供了折中的解决方案，但没有解决进程间的窗口层叠问题。而 MiniGUI2.0.x 实现的 MiniGUI-Processes 模式为 Linux 等多进程操作系统提供了完整的图形界面解决方案。

3．MiniGUI-Standalone

在这种运行模式下，MiniGUI 可以以独立进程的方式运行，既不需要多线程也不

需要多进程的支持，这种运行模式适合功能单一的应用场合。比如在一些使用 uClinux 的嵌入式产品中，因为各种原因而缺少线程支持，这时，就可以使用 MiniGUI-Standalone 来开发应用软件。

一般而言，MiniGUI-Standalone 模式的适应面最广，可以支持几乎所有的操作系统；MiniGUI-Threads 模式的适用面次之，可运行在支持多任务的实时嵌入式操作系统，或者具备完整 UNIX 特性的普通操作系统；MiniGUI-Processes 模式的适用面较小，它仅适合于具备完整 UNIX 特性的嵌入式操作系统，比如 Linux 和 VxWorks。

MiniGUI 下的通信是一种类似于 Win32 的消息机制，如果有 WIN32 图形用户界面程序的编程基础，编写 MiniGUI 程序将没有门槛。代码清单 18.18 所示为一个完整的 MiniGUI 应用程序，该程序在屏幕上创建一个大小为 240×180 的应用程序窗口，并在窗口客户区的中部显示“Hello world!”，如图 18.11 所示。

代码清单 18.18　MiniGUI 应用程序范例

```
1  #include <stdio.h>
2  #include <minigui/common.h>
3  #include <minigui/minigui.h>
4  #include <minigui/gdi.h>
5  #include <minigui/window.h>
6  static int HelloWinProc ( HWND hWnd, int message, WPARAM wParam, LPARAM lParam )
7  {
8     HDC hdc;
9     switch (message)
10    {
11       case MSG_PAINT:
12          hdc = BeginPaint(hWnd);
13          TextOut(hdc, 60, 60, "Hello world!");//输出文本
14          EndPaint(hWnd, hdc);
15          return 0;
16       case MSG_CLOSE:
17          DestroyMainWindow(hWnd);//破坏窗口
18          PostQuitMessage(hWnd); //释放退出消息
19          return 0;
20     }
21     return DefaultMainWinProc(hWnd, message, wParam, lParam);
22 }
23
24 int MiniGUIMain(int argc, const char *argv[])
25 {
26      MSG Msg;
27      HWND hMainWnd;//主窗口句柄
```

```
28     MAINWINCREATE CreateInfo;
29     #ifdef _MGRM_PROCESSES
30        JoinLayer(NAME_DEF_LAYER, "helloworld", 0, 0);
31     #endif
32     CreateInfo.dwStyle = WS_VISIBLE | WS_BORDER | WS_CAPTION;
33     CreateInfo.dwExStyle = WS_EX_NONE;
34     CreateInfo.spCaption = "HelloWorld";
35     CreateInfo.hMenu = 0;
36     CreateInfo.hCursor = GetSystemCursor(0);
37     CreateInfo.hIcon = 0;
38     CreateInfo.MainWindowProc = HelloWinProc;//主窗口消息处理程序
39     CreateInfo.lx = 0;
40     CreateInfo.ty = 0;
41     CreateInfo.rx = 240; //水平尺寸
42     CreateInfo.by = 180; //垂直尺寸
43     CreateInfo.iBkColor = COLOR_lightwhite;
44     CreateInfo.dwAddData = 0;
45     CreateInfo.hHosting = HWND_DESKTOP;
46     hMainWnd = CreateMainWindow(&CreateInfo); //创建主窗口
47     if (hMainWnd == HWND_INVALID)
48         return  - 1;
49     ShowWindow(hMainWnd, SW_SHOWNORMAL);
50     while (GetMessage(&Msg, hMainWnd))
51    {
52         TranslateMessage(&Msg);//消息译码
53         DispatchMessage(&Msg); //派送消息
54   }
55   MainWindowThreadCleanup(hMainWnd);
56   return 0;
57 }
```

图 18.11　MiniGUI 使用范例

18.11 实例：S3C2410 LCD 设备驱动

18.11.1 S3C2410 LCD 控制器硬件描述

S3C2410 内部集成了 LCD 控制器，它支持 STN 和 TFT 屏，其特性如下。

1．STN 屏

支持 3 种扫描方式：4 位单扫描、4 位双扫描和 8 位单扫描。
支持单色、4 级灰度和 16 级灰度屏。
支持 256 色和 4096 色彩色 STN 屏（CSTN）。
支持分辨率为 640*480、320*240、160*160 以及其他规格的多种 LCD。

2．TFT 屏

支持单色、4 级灰度、256 色的调色板显示模式。
支持 64K 和 16M 色非调色板显示模式。
支持分辨率为 640*480，320*240 及其他多种规格的 LCD。

图 18.12 所示为 S3C2410 LCD 控制器内部的逻辑结构，REGBANK 是 LCD 控制器的寄存器组，用来对 LCD 控制器的各项参数进行设置。而 LCDCDMA 则是 LCD 控制器专用的 DMA 信道，负责将视频资料从系统总线（System Bus）上取来，通过 VIDPRCS 从 VD[23:0]发送给 LCD 屏。同时 TIMEGEN 和 LPC3600 负责产生 LCD 屏所需要的控制时序，例如 VSYNC、HSYNC、VCLK、VDEN，然后从 VIDEO MUX 送给 LCD 屏。

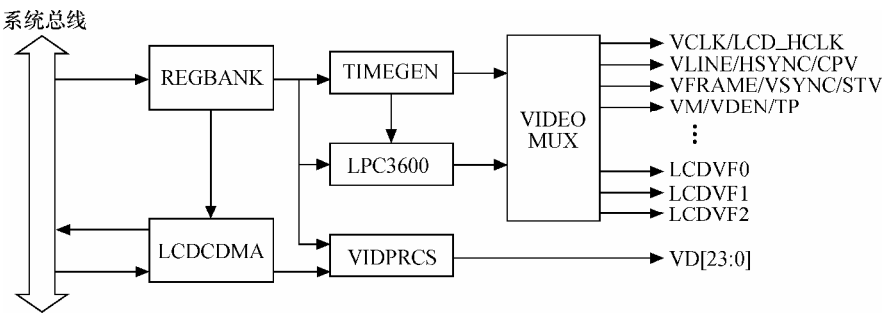
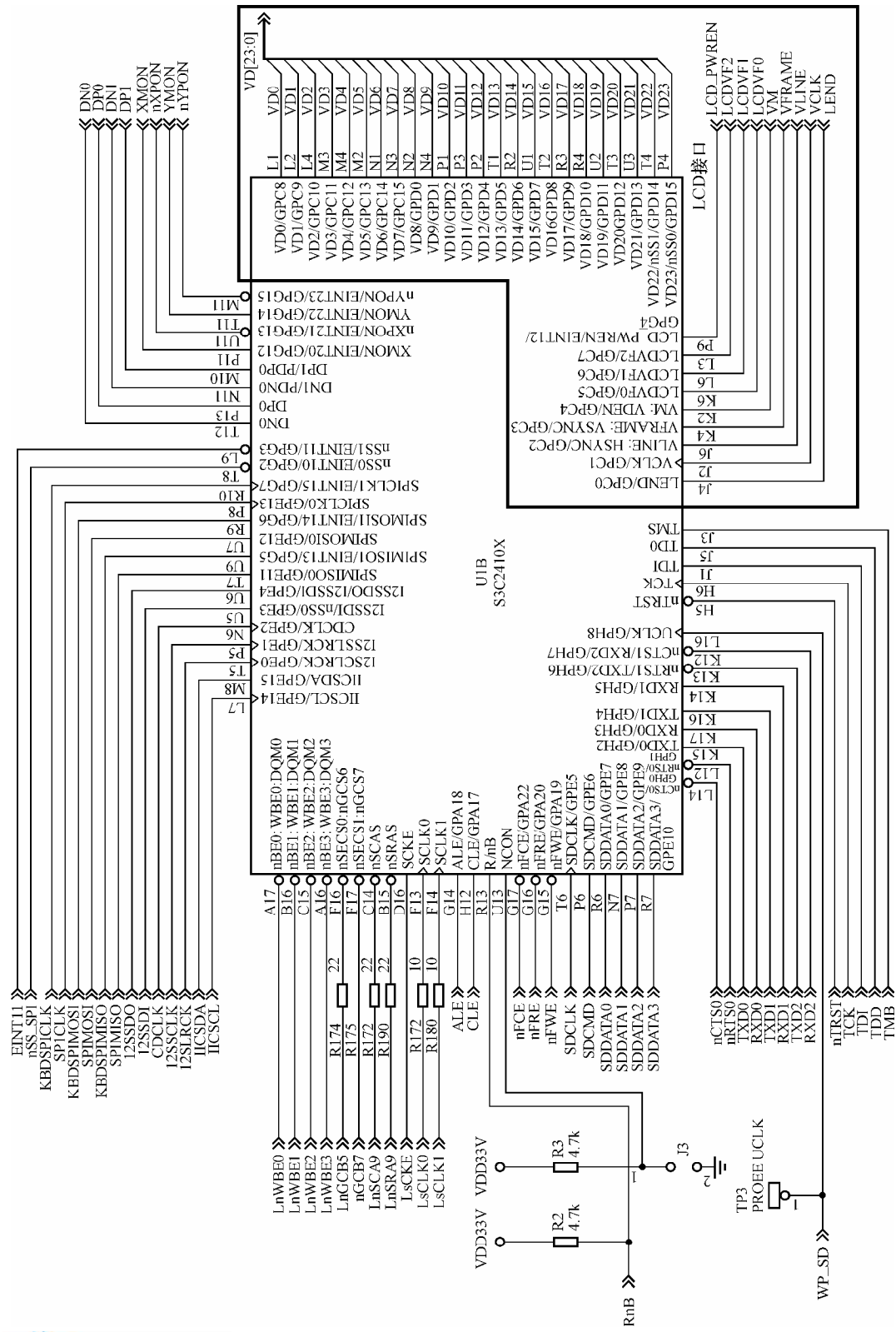


图 18.12　S3C2410 LCD 控制器逻辑结构

图 18.13 所示为 S3C2410 LCD 控制器外部的引脚连接，其中 VM 是 LCD 驱动器的 AC 信号。VM 信号被 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。VM 信号可以与每个帧同步，也可以与可变数量的 VLINE 信号同步。

S3C2410 LCD 控制器内部的寄存器可以控制 LCD 控制器接口的工作模式，如寄存器 1 记录屏幕类型（TFT、STN4、STN8、DSCAN4 等）和 BPP，寄存器 2 和寄存器 3 记录定时参数，代码清单 18.19 所示为 S3C2410 LCD 控制器的寄存器。


U1B
S3C2410X
LCD接口
VD[23:0]
LCD_PWREN
LCDVF2
LCDVF1
LCDVF0
VM
VFRAME
VLINE
VCLK
LEND
DN0
DP0
DN1
DP1
XMON
nXPON
YMON
nYPON
nBE0:WBE0:DQM0
nBE1:WBE1:DQM1
nBE2:WBE2:DQM2
nBE3:WBE3:DQM3
nSECS0:nGCS6
nSECS1:nGCS7
nSCAS
nSRAS
SCKE
SCLK0
SCLK1
ALE/GPA18
CLE/GPA17
R/nB
NCON
nFCE/GPA22
nFRE/GPA20
nFWE/GPA19
SDCLK/GPE5
SDCMD/GPE6
SDDATA0/GPE7
SDDATA1/GPE8
SDDATA2/GPE9
SDDATA3/GPE10
nCTS0/GPH0
nRTS0/GPH1
RXD0/GPH2
TXD0/GPH3
RXD1/GPH4
nRTS1/TXD2/GPH6
nCTS1/RXD2/GPH7
UCLK/GPH8
nTRST
TCK
TDI
TDO
TMS
LCD_PWREN/EINT12/GPG4
VD0/GPC8
VD1/GPC9
VD2/GPC10
VD3/GPC11
VD4/GPC12
VD5/GPC13
VD6/GPC14
VD7/GPC15
VD8/GPD0
VD9/GPD1
VD10/GPD2
VD11/GPD3
VD12/GPD4
VD13/GPD5
VD14/GPD6
VD15/GPD7
VD16GPD8
VD17/GPD9
VD18/GPD10
VD19/GPD11
VD20GPD12
VD21/GPD13
VD22/nSS1/GPD14
VD23/nSS0/GPD15
LENDGPC0
VCLK/GPC1
VLINE: HSYNC/GPC2
VFRAME: VSYNC/GPC3
VM: VDEN/GPC4
LCDVF0/GPC5
LCDVF1/GPC6
LCDVF2/GPC7
nXDPON/EINT21/GPG13
nYDPON/EINT23/GPG15
XMON/EINT20/GPG12
YMON/EINT22/GPG14
DP0
DN0
nSS0/EINT10/GPG2
nSS1/EINT11/GPG3
SPICLK1/EINT15/GPG7
SPICLK0/GPE13
SPIMOSI1/EINT14/GPG6
SPIMOSI0/GPE12
SPIMISO1/EINT13/GPG5
SPIMISO0/GPE11
I2SSDO/I2SSDI/GPE4
I2SSDI/nSS0/GPE3
CDCLK/GPE2
I2SSLRCK/GPE1
I2SCLRCK/GPE0
IICSDA/GPE15
IICSCL/GPE14
EINT11
nSS_SPI
KBDSPICLK
SPICLK
KBDSPIMOSI
SPIMOSI
KBDSPIMISO
SPIMISO
I2SSDO
I2SSDI
CDCLK
I2SSCLK
I2SLRCK
IICSDA
IICSCL
nCTS0
nRTS0
TXD0
RXD0
TXD1
RXD1
TXD2
RXD2
nTRST
TCK
TDI
TDD
TMB
LnWBE0
LnWBE1
LnWBE2
LnWBE3
LnGCB5
nGCB7
LnSCA9
LnSRA9
LsCKE
LsCLK0
LsCLK1
ALE
CLE
nFCE
nFRE
nFWE
SDCLK
SDCMD
SDDATA0
SDDATA1
SDDATA2
SDDATA3
RnB
WP_SD
VDD33V
R174
R175
R172
R190
R172
R180
R3
4.7k
R2
4.7k
J3
TP3
PROEE UCLK

代码清单 18.19　S3C2410 LCD 控制器寄存器

```
1  #define S3C2410_LCDREG(x) ((x) + S3C24XX_VA_LCD)
2
3  /* LCD 控制寄存器 */
4  #define S3C2410_LCDCON1        S3C2410_LCDREG(0x00)
5  #define S3C2410_LCDCON2        S3C2410_LCDREG(0x04)
6  #define S3C2410_LCDCON3        S3C2410_LCDREG(0x08)
7  #define S3C2410_LCDCON4        S3C2410_LCDREG(0x0C)
8  #define S3C2410_LCDCON5        S3C2410_LCDREG(0x10)
9  /*LCD 控制寄存器 1 的位定义*/
10 #define S3C2410_LCDCON1_CLKVAL(x)  ((x) << 8)
11 #define S3C2410_LCDCON1_MMODE      (1<<7)
12 #define S3C2410_LCDCON1_DSCAN4     (0<<5)
13 #define S3C2410_LCDCON1_STN4       (1<<5)
14 #define S3C2410_LCDCON1_STN8       (2<<5)
15 #define S3C2410_LCDCON1_TFT        (3<<5)
16
17 #define S3C2410_LCDCON1_STN1BPP    (0<<1)
18 #define S3C2410_LCDCON1_STN2GREY   (1<<1)
19 #define S3C2410_LCDCON1_STN4GREY   (2<<1)
20 #define S3C2410_LCDCON1_STN8BPP    (3<<1)
21 #define S3C2410_LCDCON1_STN12BPP   (4<<1)
22
23 #define S3C2410_LCDCON1_TFT1BPP   (8<<1)
24 #define S3C2410_LCDCON1_TFT2BPP    (9<<1)
25 #define S3C2410_LCDCON1_TFT4BPP    (10<<1)
26 #define S3C2410_LCDCON1_TFT8BPP    (11<<1)
27 #define S3C2410_LCDCON1_TFT16BPP   (12<<1)
28 #define S3C2410_LCDCON1_TFT24BPP   (13<<1)
29
30 #define S3C2410_LCDCON1_ENVID     (1)
31 #define S3C2410_LCDCON1_MODEMASK    0x1E
32
33 /*LCD 控制寄存器 2 的位定义*/
34
35 #define S3C2410_LCDCON2_VBPD(x)     ((x) << 24)
36 #define S3C2410_LCDCON2_LINEVAL(x)  ((x) << 14)
37 #define S3C2410_LCDCON2_VFPD(x)     ((x) << 6)
38 #define S3C2410_LCDCON2_VSPW(x)     ((x) << 0)
39
40 #define S3C2410_LCDCON2_GET_VBPD(x) ( ((x) >> 24) & 0xFF)
```

```
41 #define S3C2410_LCDCON2_GET_VFPD(x) ( ((x) >>  6) & 0xFF)
42 #define S3C2410_LCDCON2_GET_VSPW(x) ( ((x) >>  0) & 0x3F)
43
44 /*LCD 控制寄存器 3 的位定义*/
45
46 #define S3C2410_LCDCON3_HBPD(x)       ((x) << 19)
47 #define S3C2410_LCDCON3_WDLY(x)       ((x) << 19)
48 #define S3C2410_LCDCON3_HOZVAL(x)     ((x) << 8)
49 #define S3C2410_LCDCON3_HFPD(x)       ((x) << 0)
50 #define S3C2410_LCDCON3_LINEBLANK(x)  ((x) << 0)
51
52 #define S3C2410_LCDCON3_GET_HBPD(x) ( ((x) >> 19) & 0x7F)
53 #define S3C2410_LCDCON3_GET_HFPD(x) ( ((x) >>  0) & 0xFF)
54
55 /*LCD 控制寄存器 4 的位定义*/
56
57 #define S3C2410_LCDCON4_MVAL(x)    ((x) << 8)
58 #define S3C2410_LCDCON4_HSPW(x)    ((x) << 0)
59 #define S3C2410_LCDCON4_WLH(x)     ((x) << 0)
60
61 #define S3C2410_LCDCON4_GET_HSPW(x) ( ((x) >>  0) & 0xFF)
62
63 /*LCD 控制寄存器 5 的位定义*/
64
65 #define S3C2410_LCDCON5_BPP24BL    (1<<12)
66 #define S3C2410_LCDCON5_FRM565     (1<<11)
67 #define S3C2410_LCDCON5_INVVCLK    (1<<10)
68 #define S3C2410_LCDCON5_INVVLINE    (1<<9)
69 #define S3C2410_LCDCON5_INVVFRAME   (1<<8)
70 #define S3C2410_LCDCON5_INVVD       (1<<7)
71 #define S3C2410_LCDCON5_INVVDEN     (1<<6)
72 #define S3C2410_LCDCON5_INVPWREN    (1<<5)
73 #define S3C2410_LCDCON5_INVLEND     (1<<4)
74 #define S3C2410_LCDCON5_PWREN       (1<<3)
75 #define S3C2410_LCDCON5_ENLEND      (1<<2)
76 #define S3C2410_LCDCON5_BSWP        (1<<1)
77 #define S3C2410_LCDCON5_HWSWP        (1<<0)
78
79 /* 显示缓冲区开始位置寄存器 */
80 #define S3C2410_LCDSADDR1   S3C2410_LCDREG(0x14)
81 #define S3C2410_LCDSADDR2   S3C2410_LCDREG(0x18)
82 #define S3C2410_LCDSADDR3   S3C2410_LCDREG(0x1C)
83
84 #define S3C2410_LCDBANK(x)   ((x) << 21)
85 #define S3C2410_LCDBASEU(x)    (x)
86
87 #define S3C2410_OFFSIZE(x)  ((x) << 11)
88 #define S3C2410_PAGEWIDTH(x)   (x)
89
```

```
90  /* 查找表 */
91
92  #define S3C2410_REDLUT    S3C2410_LCDREG(0x20)
93  #define S3C2410_GREENLUT   S3C2410_LCDREG(0x24)
94  #define S3C2410_BLUELUT     S3C2410_LCDREG(0x28)
95
96  #define S3C2410_DITHMODE   S3C2410_LCDREG(0x4C)
97  #define S3C2410_TPAL       S3C2410_LCDREG(0x50)
98
99  #define S3C2410_TPAL_EN        (1<<24)
```

18.11.2 S3C2410 LCD 驱动的模块加载与卸载函数

在 S3C2410 中，作为一个相对独立的硬件单元，LCD 控制器被认定为平台设备，因此，在驱动的模块加载和卸载函数中，分别注册和注销对应的 platform_driver 即可，如代码清单 18.20 所示。

代码清单 18.20 S3C2410 LCD 驱动的模块加载与卸载函数

```
1  static struct platform_driver s3c2410fb_driver = {
2  .probe     = s3c2410fb_probe,//平台驱动探测函数
3  .remove    = s3c2410fb_remove,  //平台驱动移除函数
4  .suspend   = s3c2410fb_suspend,
5  .resume    = s3c2410fb_resume,
6  .driver      = {
7       .name  = "s3c2410-lcd",  //驱动名
8  .owner      = THIS_MODULE,
9  },
10 };
11
12 int __devinit s3c2410fb_init(void)
13 {
14  return platform_driver_register(&s3c2410fb_driver);//注册平台设备
15 }
16
17 static void __exit s3c2410fb_cleanup(void)
18 {
19  platform_driver_unregister(&s3c2410fb_driver);//注销平台设备
20 }
21
22 module_init(s3c2410fb_init);
23 module_exit(s3c2410fb_cleanup);
```

S3C2410 LCD 控制器平台设备定义于 arch/arm/mach-s3c2410 对应的目录中，其对应的 platform_device 结构体如代码清单 18.21 所示。

代码清单 18.21 S3C2410 LCD 驱动的平台设备结构体

```
1  struct platform_device s3c_device_lcd = {
2  .name      = "s3c2410-lcd", //平台设备名
```

```
3  .id        = -1,
4  .num_resources   = ARRAY_SIZE(s3c_lcd_resource),//资源数
5  .resource    = s3c_lcd_resource,//资源
6  .dev            = {
7    .dma_mask     = &s3c_device_lcd_dmamask,
8     .coherent_dma_mask   = 0xffffffffUL
9   }
10 };
```

第 4 行和第 5 行访问的 s3c_lcd_resource 为平台设备所使用的资源，S3C2410 LCD 平台设备所使用的资源如代码清单 18.22 所示。

代码清单 18.22　S3C2410 LCD 驱动的平台设备资源

```
1  static struct resource s3c_lcd_resource[] = {
2  [0] = {  //I/O 内存资源
3     .start = S3C24XX_PA_LCD,
4     .end   = S3C24XX_PA_LCD + S3C24XX_SZ_LCD - 1,
5     .flags = IORESOURCE_MEM,
6   },
7   [1] = {  //IRQ 资源
8        .start = IRQ_LCD,
9        .end   = IRQ_LCD,
10       .flags = IORESOURCE_IRQ,
11  }
12 };
```

另外，内核还定义了一个 s3c2410fb_mach_info 结构体，记录 LCD 的“机器信息”（即硬件特征），并将这个结构体赋值给平台设备的平台数据（platform_data），如代码清单 18.23 所示。

代码清单 18.23　S3C2410 LCD 驱动平台设备的平台数据

```
1  void __init s3c24xx_fb_set_platdata(struct s3c2410fb_mach_info *pd)
2  {
3    struct s3c2410fb_mach_info *npd;
4
5    npd = kmalloc(sizeof(*npd), GFP_KERNEL); //申请 s3c2410fb_mach_info
内存
6    if (npd)
7    {
8      memcpy(npd, pd, sizeof(*npd));
9      //设置设备的 platform_data 为 s3c2410fb_mach_info
10     s3c_device_lcd.dev.platform_data = npd;
11   }
12   else
13   {
14     printk(KERN_ERR "no memory for LCD platform data\n");
15   }
```

```
16 }
17
18 struct s3c2410fb_mach_info
19 {
20     unsigned char fixed_syncs; /* 不更新同步和边界 */
21
22     /* 屏幕尺寸 */
23     int width;
24     int height;
25
26     /* 屏幕信息 */
27     struct s3c2410fb_val xres;
28     struct s3c2410fb_val yres;
29     struct s3c2410fb_val bpp;
30
31     /* LCD 配置寄存器 */
32     struct s3c2410fb_hw regs;
33     ...
34 };
35
36 struct s3c2410fb_val
37 {
38     unsigned int defval; //默认值
39     unsigned int min;    //最小值
40     unsigned int max;    //最大值
41 };
42
43 struct s3c2410fb_hw
44 {
45    /*5 个 LCD 配置寄存器*/
46    unsigned long lcdcon1;
47    unsigned long lcdcon2;
48    unsigned long lcdcon3;
49    unsigned long lcdcon4;
50    unsigned long lcdcon5;
51 };
```

18.11.3 S3C2410 LCD 驱动的探测与移除函数

代码清单 18.20 所示的模块加载函数中对平台驱动的注册会导致其中探测函数的执行，这个探测函数完成 LCD 硬件的初始化、FBI 参数的填充、申请显示缓冲区并注册帧缓冲设备，如代码清单 18.24 所示；模块卸载函数中对平台驱动的注销会导致其中移除函数的执行，这个函数释放显示缓冲区并注销帧缓冲设备，如代码清单 18.25 所示。

代码清单 18.24 S3C2410 LCD 驱动的探测函数

```
1  static int __init s3c2410fb_probe(struct platform_device *pdev)
2  {
3      struct s3c2410fb_info *info;
4      struct fb_info *fbinfo;
5      struct s3c2410fb_hw *mregs;
6      int ret;
7      int irq;
8      int i;
9
10     mach_info = pdev->dev.platform_data; //获得 mach_info 结构体指针
11     if (mach_info == NULL)
12     {
13          dev_err(&pdev->dev, "no platform data for lcd, cannot
attach\n");
14       return  - EINVAL;
15     }
16
17     mregs = &mach_info->regs; //获得 mach_info 结构体中的寄存器指针
18
19     irq = platform_get_irq(pdev, 0); //获得 IRQ 号
20     if (irq < 0)
21     {
22        dev_err(&pdev->dev, "no irq for device\n");
23        return  - ENOENT;
24     }
25
26     //分配 FBI 结构体
27       fbinfo = framebuffer_alloc(sizeof(struct s3c2410fb_info),
&pdev->dev);
28     if (!fbinfo)
29     {
```

```
30          return  - ENOMEM;
31      }
32
33      info = fbinfo->par;
34      info->fb = fbinfo;
35      platform_set_drvdata(pdev, fbinfo);//设置平台设备数据
36
37      dprintk("devinit\n");
38
39      strcpy(fbinfo->fix.id, driver_name);
40
41      memcpy(&info->regs, &mach_info->regs, sizeof(info->regs));
42
43      info->mach_info = pdev->dev.platform_data;
44
45      //初始化固定参数
46      fbinfo->fix.type = FB_TYPE_PACKED_PIXELS;
47      fbinfo->fix.type_aux = 0;
48      fbinfo->fix.xpanstep = 0;
49      fbinfo->fix.ypanstep = 0;
50      fbinfo->fix.ywrapstep = 0;
51      fbinfo->fix.accel = FB_ACCEL_NONE;
52
53      //初始化可变参数
54      fbinfo->var.nonstd = 0;
55      fbinfo->var.activate = FB_ACTIVATE_NOW;
56      fbinfo->var.height = mach_info->height;
57      fbinfo->var.width = mach_info->width;
58      fbinfo->var.accel_flags = 0;
59      fbinfo->var.vmode = FB_VMODE_NONINTERLACED;
60
61      fbinfo->fbops = &s3c2410fb_ops;   //fb_ops
62      fbinfo->flags = FBINFO_FLAG_DEFAULT;
63      fbinfo->pseudo_palette = &info->pseudo_pal;
64
65      //初始化可变参数中的分辨率和 bpp
66      fbinfo->var.xres = mach_info->xres.defval;
67      fbinfo->var.xres_virtual = mach_info->xres.defval;
68      fbinfo->var.yres = mach_info->yres.defval;
69      fbinfo->var.yres_virtual = mach_info->yres.defval;
70      fbinfo->var.bits_per_pixel = mach_info->bpp.defval;
```

```
    71    //初始化可变参数中的上下边界和垂直同步
    72
fbinfo->var.upper_margin=3C2410_LCDCON2_GET_VBPD(mregs->lcdcon2) + 1;
    73                                  fbinfo->var.lower_margin        =
3C2410_LCDCON2_GET_VFPD(mregs->lcdcon2) + 1;
    74    fbinfo->var.vsync_len = S3C2410_LCDCON2_GET_VSPW(mregs->lcdcon2)
+ 1;
    75    //初始化可变参数中的左右边界和水平同步
    76                                   fbinfo->var.left_margin        =
S3C2410_LCDCON3_GET_HFPD(mregs->lcdcon3) + 1;
    77                                  fbinfo->var.right_margin        =
S3C2410_LCDCON3_GET_HBPD(mregs->lcdcon3) + 1;
    78    fbinfo->var.hsync_len = S3C2410_LCDCON4_GET_HSPW(mregs->lcdcon4)
+ 1;
    79    //设置可变参数中的 R/G/B 位数和位置
    80    fbinfo->var.red.offset = 11;
    81    fbinfo->var.green.offset = 5;
    82    fbinfo->var.blue.offset = 0;
    83    fbinfo->var.transp.offset = 0;
    84    fbinfo->var.red.length = 5;
    85    fbinfo->var.green.length = 6;
    86    fbinfo->var.blue.length = 5;
    87    fbinfo->var.transp.length = 0;
    88    fbinfo->fix.smem_len = mach_info->xres.max *mach_info->yres.max
    89        *mach_info ->bpp.max / 8;    //buffer 长度
    90
    91    for (i = 0; i < 256; i++)
    92         info->palette_buffer[i] = PALETTE_BUFF_CLEAR;
    93
    94    //申请内存区域
    95     if (!request_mem_region((unsigned long)S3C24XX_VA_LCD, SZ_1M,
"s3c2410-lcd"))
    96    {
    97        ret =  - EBUSY;
    98        goto dealloc_fb;
    99    }
    100   dprintk("got LCD region\n");
    101
```

```
    102   //申请中断
    103   ret = request_irq(irq, s3c2410fb_irq, SA_INTERRUPT, pdev->name,
info);
    104   if (ret)
    105   {
    106     dev_err(&pdev->dev, "cannot get irq %d - err %d\n", irq, ret);
    107     ret =  - EBUSY;
    108     goto release_mem;
    109   }
    110
    111   //获得时钟源并使能
    112   info->clk = clk_get(NULL, "lcd");
    113   if (!info->clk || IS_ERR(info->clk))
    114   {
    115      printk(KERN_ERR "failed to get lcd clock source\n");
    116      ret =  - ENOENT;
    117      goto release_irq;
    118   }
    119
    120   clk_enable(info->clk);
    121   dprintk("got and enabled clock\n");
    122
    123   msleep(1);
    124
    125   /* 初始化显示缓冲区 */
    126   ret = s3c2410fb_map_video_memory(info);
    127   if (ret)
    128   {
    129      printk(KERN_ERR "Failed to allocate video RAM: %d\n", ret);
    130      ret =  - ENOMEM;
    131      goto release_clock;
    132   }
    133   dprintk("got video memory\n");
    134
    135   ret = s3c2410fb_init_registers(info); //初始化 S3C2410 的 LCD 控制
器的寄存器
    136
    137   ret = s3c2410fb_check_var(&fbinfo->var, fbinfo);//检查可变参数
    138
    139   ret = register_framebuffer(fbinfo);//注册帧缓冲设备
```

```
    140   if (ret < 0)
    141   {
    142       printk(KERN_ERR "Failed to register framebuffer device: %d\n",
ret);
    143       goto free_video_memory;
    144   }
    145
    146   /* 创建设备文件 */
    147   device_create_file(&pdev->dev, &dev_attr_debug);
    148
    149   printk(KERN_INFO "fb%d: %s frame buffer device\n", fbinfo->node,
fbinfo
    150     ->fix.id);
    151
    152   return 0;
    153
    154   free_video_memory: s3c2410fb_unmap_video_memory(info);
    155   release_clock: clk_disable(info->clk);
    156   clk_put(info->clk);
    157   release_irq:
    158   free_irq(irq, info);
    159   release_mem:
    160             release_mem_region((unsigned    long)S3C24XX_VA_LCD,
S3C24XX_SZ_LCD);
    161   dealloc_fb:
    162   framebuffer_release(fbinfo);
    163   return ret;
    164 }
```

代码清单 18.25　S3C2410 LCD 驱动的移除函数

```
    1  static int s3c2410fb_remove(struct platform_device *pdev)
    2  {
    3     struct fb_info *fbinfo = platform_get_drvdata(pdev);
    4     struct s3c2410fb_info *info = fbinfo->par;
    5     int irq;
    6
    7     s3c2410fb_stop_lcd(); //停止 LCD
    8     msleep(1);
```

```
    9
    10    s3c2410fb_unmap_video_memory(info); //释放显示缓冲区
    11
    12    if (info->clk)
    13    {
    14       //释放时钟源
    15       clk_disable(info->clk);
    16       clk_put(info->clk);
    17       info->clk = NULL;
    18    }
    19
    20    irq = platform_get_irq(pdev, 0);   //获得平台设备使用的 IRQ
    21    free_irq(irq, info); //释放 IRQ
    22    //释放内存区域
    23              release_mem_region((unsigned    long)S3C24XX_VA_LCD,
S3C24XX_SZ_LCD);
    24
    25    unregister_framebuffer(fbinfo); //注销帧缓冲设备
    26
    27    return 0;
    28 }
```

代码清单 18.24 第 126 行调用的 s3c2410fb_map_video_memory()用于申请显示缓冲区，代码清单 18.25 第 10 行调用的 s3c2410fb_unmap_video_memory()用户释放显示缓冲区，如代码清单 18.26 所示。

代码清单 18.26 S3C2410 LCD 驱动的显示缓冲区申请和释放

```
    1  static int __init s3c2410fb_map_video_memory(struct s3c2410fb_info
*fbi)
    2  {
    3  fbi->map_size = PAGE_ALIGN(fbi->fb->fix.smem_len + PAGE_SIZE);
    4  fbi->map_cpu  = dma_alloc_writecombine(fbi->dev, fbi->map_size,
    5                    &fbi->map_dma, GFP_KERNEL);//分配内存
    6
    7  fbi->map_size = fbi->fb->fix.smem_len;//显示缓冲区长度
    8
    9  if (fbi->map_cpu) {
    10      memset(fbi->map_cpu, 0xf0, fbi->map_size);
    11
    12      fbi->screen_dma       = fbi->map_dma;
    13      fbi->fb->screen_base     = fbi->map_cpu;
    14      fbi->fb->fix.smem_start  = fbi->screen_dma;//显示缓冲区开始地址
    15
```

```
    dprintk("map_video_memory: dma=%08x cpu=%p size=%08x\n",
         fbi->map_dma, fbi->map_cpu, fbi->fb->fix.smem_len);
  }

 return fbi->map_cpu ? 0 : -ENOMEM;
}

static inline void s3c2410fb_unmap_video_memory(struct s3c2410fb_info *fbi)
{
  //释放内存
       dma_free_writecombine(fbi->dev,fbi->map_size,fbi->map_cpu, fbi->map_dma);
}
```

18.11.4 S3C2410 LCD 驱动挂起与恢复函数

如果内核配置了能量管理（即定义了 CONFIG_PM 宏），平台驱动部分应该增加挂起和恢复函数，在挂起函数中关闭 LCD 和其时钟源以节省能量，在恢复函数中开启 LCD 和其时钟源，如代码清单 18.27 所示。

代码清单 18.27 S3C2410 LCD 驱动挂起与恢复函数

```
#ifdef CONFIG_PM  //支持能量管理
    /* LCD 控制器的挂起和恢复支持 */

      static int s3c2410fb_suspend(struct platform_device *dev, pm_message_t state)
    {
       struct fb_info *fbinfo = platform_get_drvdata(dev);
       struct s3c2410fb_info *info = fbinfo->par;

       s3c2410fb_stop_lcd();//停止 LCD

      msleep(1);   //在停止时钟源前延迟
      clk_disable(info->clk); //停止时钟源

      return 0;
  }

  static int s3c2410fb_resume(struct platform_device *dev)
```

```
18  {
19      struct fb_info *fbinfo = platform_get_drvdata(dev);
20      struct s3c2410fb_info *info = fbinfo->par;
21
22      clk_enable(info->clk);   //启动时钟源
23      msleep(1);
24
25      s3c2410fb_init_registers(info); //初始化 LCD 控制器寄存器
26
27      return 0;
28  }
29
30 #else //不支持能量管理
31    #define s3c2410fb_suspend NULL
32    #define s3c2410fb_resume  NULL
33 #endif
```

18.11.5 S3C2410 LCD 驱动的 fb_ops 成员函数

1. fb_ops 结构体

S3C2410 LCD 驱动中定义了如代码清单 18.28 所示的 fb_ops，其中的 s3c2410fb_check_var()、s3c2410fb_set_par()、s3c2410fb_blank()和 s3c2410fb_setcolreg()成员函数特别针对 S3C2410 而实现。

代码清单 18.28 S3C2410 LCD 驱动的 fb_ops 结构体

```
1  static struct fb_ops s3c2410fb_ops = {
2  .owner          = THIS_MODULE,
3  .fb_check_var   = s3c2410fb_check_var,//检查可变参数
4  .fb_set_par     = s3c2410fb_set_par,  //设置参数
5  .fb_blank       = s3c2410fb_blank,  //显示空白
6  .fb_setcolreg   = s3c2410fb_setcolreg, //设置颜色表
7  .fb_fillrect    = cfb_fillrect, //矩形填充
8  .fb_copyarea    = cfb_copyarea,
9  .fb_imageblit   = cfb_imageblit,
10 };
```

2. 参数检查函数

s3c2410fb_check_var()函数用于检查用户设置的屏幕参数是否合理，譬如 xres、yres 和 BPP 都有一个范围 min～max，用户的设置如果在这个范围之外，则进行调整。另外，当 BPP 改变后，R、G、B 的位域也要进行相应调整，如代码清单 18.29 所示。

代码清单 18.29 S3C2410 LCD 驱动的参数检查函数

```
1  static int s3c2410fb_check_var(struct fb_var_screeninfo *var,
```

```
2                       struct fb_info *info)
3  {
4  struct s3c2410fb_info *fbi = info->par;
5
6  dprintk("check_var(var=%p, info=%p)\n", var, info);
7
8  /* 验证 x/y 解析度 */
9
10  if (var->yres > fbi->mach_info->yres.max)
11      var->yres = fbi->mach_info->yres.max;
12  else if (var->yres < fbi->mach_info->yres.min)
13       var->yres = fbi->mach_info->yres.min;
14
15  if (var->xres > fbi->mach_info->xres.max)
16      var->yres = fbi->mach_info->xres.max;
17  else if (var->xres < fbi->mach_info->xres.min)
18      var->xres = fbi->mach_info->xres.min;
19
20  /* 验证 bpp */
21
22  if (var->bits_per_pixel > fbi->mach_info->bpp.max)
23     var->bits_per_pixel = fbi->mach_info->bpp.max;
24  else if (var->bits_per_pixel < fbi->mach_info->bpp.min)
25     var->bits_per_pixel = fbi->mach_info->bpp.min;
26
27  /* 设置 R、G、B 位置 */
28
29  if (var->bits_per_pixel == 16) {
30       var->red.offset     = 11;
31       var->green.offset   = 5;
32       var->blue.offset    = 0;
33       var->red.length     = 5;
34       var->green.length   = 6;
35       var->blue.length    = 5;
36       var->transp.length = 0;
37  } else {
38       var->red.length     = var->bits_per_pixel;
39       var->red.offset     = 0;
40       var->green.length   = var->bits_per_pixel;
```

```
41      var->green.offset   = 0;
42      var->blue.length    = var->bits_per_pixel;
43      var->blue.offset    = 0;
44      var->transp.length  = 0;
45   }
46
47   return 0;
48 }
```

3．参数设置函数

s3c2410fb_set_par()函数根据用户设置的可变参数 var 调整固定参数 fix，修改 fix 的 line_length，最终将相应的修改在硬件上激活，如代码清单 18.30 所示。

代码清单 18.30　参数设置函数

```
1  static int s3c2410fb_set_par(struct fb_info *info)
2  {
3      struct s3c2410fb_info *fbi = info->par;
4      struct fb_var_screeninfo *var = &info->var;
5
6      if (var->bits_per_pixel == 16)
7        fbi->fb->fix.visual = FB_VISUAL_TRUECOLOR;
8     else
9        fbi->fb->fix.visual = FB_VISUAL_PSEUDOCOLOR;
10
11    fbi->fb->fix.line_length = (var->width *var->bits_per_pixel) /
8;
12
13   /* 激活新的配置 */
14
15   s3c2410fb_activate_var(fbi, var);
16   return 0;
17 }
18
19 /* 根据 var 和 fix 参数设置 LCD 控制器的硬件寄存器 */
20  static void s3c2410fb_activate_var(struct s3c2410fb_info *fbi,
struct
21     fb_var_screeninfo *var)
22 {
23     fbi->regs.lcdcon1 &= ~S3C2410_LCDCON1_MODEMASK;
24
25     dprintk("%s: var->xres  = %d\n", __FUNCTION__, var->xres);
26     dprintk("%s: var->yres  = %d\n", __FUNCTION__, var->yres);
```

```
    27              dprintk("%s:  var->bpp      =  %d\n",  __FUNCTION__,
var->bits_per_pixel);
    28
    29     switch (var->bits_per_pixel)
    30    {
    31      //设置 bpp
    32      case 1:
    33         fbi->regs.lcdcon1 |= S3C2410_LCDCON1_TFT1BPP;
    34         break;
    35      case 2:
    36         fbi->regs.lcdcon1 |= S3C2410_LCDCON1_TFT2BPP;
    37         break;
    38      case 4:
    39        fbi->regs.lcdcon1 |= S3C2410_LCDCON1_TFT4BPP;
    40        break;
    41      case 8:
    42        fbi->regs.lcdcon1 |= S3C2410_LCDCON1_TFT8BPP;
    43        break;
    44      case 16:
    45        fbi->regs.lcdcon1 |= S3C2410_LCDCON1_TFT16BPP;
    46        break;
    47   }
    48
    49   /* 检查是否需要更新同步和边界 */
    50
    51   if (!fbi->mach_info->fixed_syncs)
    52   {
    53              dprintk("setting  vert:  up=%d,  low=%d,  sync=%d\n",
var->upper_margin,
    54         var ->lower_margin, var->vsync_len);
    55
    56              dprintk("setting  horz:  lft=%d,  rt=%d,  sync=%d\n",
var->left_margin, var
    57         ->right_margin, var->hsync_len);
    58
    59     fbi->regs.lcdcon2 = S3C2410_LCDCON2_VBPD(var->upper_margin - 1)
|
    60         S3C2410_LCDCON2_VFPD(var->lower_margin - 1) |
```

```
61          S3C2410_LCDCON2_VSPW(var ->vsync_len - 1);
62
63      fbi->regs.lcdcon3 = S3C2410_LCDCON3_HBPD(var->right_margin - 1)
|
64          S3C2410_LCDCON3_HFPD(var->left_margin - 1);
65
66      fbi->regs.lcdcon4 &= ~S3C2410_LCDCON4_HSPW(0xff);
67      fbi->regs.lcdcon4 |= S3C2410_LCDCON4_HSPW(var->hsync_len - 1);
68    }
69
70    /* 更新 X/Y 信息 */
71
72    fbi->regs.lcdcon2 &= ~S3C2410_LCDCON2_LINEVAL(0x3ff);
73    fbi->regs.lcdcon2 |= S3C2410_LCDCON2_LINEVAL(var->yres - 1);
74
75    fbi->regs.lcdcon3 &= ~S3C2410_LCDCON3_HOZVAL(0x7ff);
76    fbi->regs.lcdcon3 |= S3C2410_LCDCON3_HOZVAL(var->xres - 1);
77
78    if (var->pixclock > 0)
79    {
80        int clkdiv = s3c2410fb_calc_pixclk(fbi, var->pixclock);
81
82        clkdiv = (clkdiv / 2) - 1;
83        if (clkdiv < 0)
84            clkdiv = 0;
85
86        fbi->regs.lcdcon1 &= ~S3C2410_LCDCON1_CLKVAL(0x3ff);
87        fbi->regs.lcdcon1 |= S3C2410_LCDCON1_CLKVAL(clkdiv);
88    }
89
90    /* 重写 LCD 控制器寄存器 */
91
92    dprintk("new register set:\n");
93    dprintk("lcdcon[1] = 0x%08lx\n", fbi->regs.lcdcon1);
94    dprintk("lcdcon[2] = 0x%08lx\n", fbi->regs.lcdcon2);
95    dprintk("lcdcon[3] = 0x%08lx\n", fbi->regs.lcdcon3);
96    dprintk("lcdcon[4] = 0x%08lx\n", fbi->regs.lcdcon4);
97    dprintk("lcdcon[5] = 0x%08lx\n", fbi->regs.lcdcon5);
98
99              writel(fbi->regs.lcdcon1    &~S3C2410_LCDCON1_ENVID,
```

```
S3C2410_LCDCON1);
    100   writel(fbi->regs.lcdcon2, S3C2410_LCDCON2);
    101   writel(fbi->regs.lcdcon3, S3C2410_LCDCON3);
    102   writel(fbi->regs.lcdcon4, S3C2410_LCDCON4);
    103   writel(fbi->regs.lcdcon5, S3C2410_LCDCON5);
    104
    105   /* set lcd address pointers */
    106   s3c2410fb_set_lcdaddr(fbi);
    107
    108   writel(fbi->regs.lcdcon1, S3C2410_LCDCON1);
    109 }
```

18.12 总结

帧缓冲设备是一种典型的字符设备，它统一了显存，将显示缓冲区直接映射到用户空间。帧缓冲设备驱动 file_operations 中 VFS 接口函数由 fbmem.c 文件统一实现。这样，驱动工程师的工作重点将是实现针对特定设备 fb_info 中的 fb_ops 的成员函数，另外，理解并能灵活地修改 fb_info 中的 var 和 fix 参数非常关键。fb_info 中的 var 参数直接和 LCD 控制器的硬件设置以及 LCD 屏幕对应。

在用户空间，应用程序直接按照预先设置的 R、G、B 位数和偏移写经过 mmap() 映射后的显示缓冲区就可实现图形的显示，省去了内存从用户空间到内核空间的复制过程。

- 出版教材：http://www.embedu.org/courses/course3.htm
- 实验设备：http://www.embedu.org/courses/course5.htm

推荐课程：华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班：

  http://www.farsight.com.cn/courses/TS-LinuxApp.htm

- 嵌入式 Linux 系统开发班：

  http://www.farsight.com.cn/courses/TS-LinuxEMB.htm

- 嵌入式 Linux 驱动开发班：

  http://www.farsight.com.cn/courses/TS-LinuxDriver.htm