显示设备： LED 点阵屏：大型的户外广告机；

LCD 屏：液晶屏

LCD 常的接口类型有 RGB、 CPU、 SPI、 MIPI、 MDDI、LVDS 和 VGA。

PPI 是每平方英寸所拥有的像素数目 ---- 分辨率

4k： 蓝光： 3900*2000

1080p： 超清： 1920*1080

720p： 高清： 1280*720

360p： 标清： 640*360

BPP 是每个像素使用多少位来表示其颜色 ---- 像素

1bpp： 黑白

2bpp： 4bpp， 8bpp：

16bpp： 伪彩， M3

24bpp： 真彩

32bpp： 真彩， 多出来的 8 位： 增加透明度

三原色 RGB：

光线有很多颜色，但是主要是三原色混合而成。而这三原色，对应我们的代码，实际上就是 0~255 的数字，由 8 位二进制数字组成。

8 位 8 位 8 位 8 位 ----- 32 位的数

一、LCD 驱动程序分析

设备驱动编写方法：

方法 1： 查看数据手册， 原理图编写驱动

----- 适合小型驱动(led、 key、 beep、 uart、 硬件的定时器等等)

方法 2： 内核提供了平台设备总线驱动层操作，你只需要给内核驱动层需要的资源，内核就可以完成，

平台设备总线：将驱动分为设备层、驱动层，匹配由总线完成。

设备层： 资源； 驱动层： 框架；

优点： 可移植性强(驱动层内核自带封装， 编写大型驱动只需要编写相应的设备资源即可)

例：LCD 屏驱动：数据线： 24 个 ，24 组寄存器 196 个寄存器；

所有的 LCD 屏驱动都是统一的操作，不同的只是屏的参数不同，所以不用每次都去编写 LCD 屏驱动。

驱动层 --- LCD 屏驱动框架性的东西（独立出来）；

设备层---- 资源，只需要编写设备层即可；

fbmem.c //内核自带， 实现 lcd 屏 framebuuffer 原理；

s3c-fb.c //三星驱动工程师编写实现 lcd 屏平台设备总线的驱动层；

framebuffer 原理见文档第二页。 主设备号为 29 设备节点名字为 fb%d；

帧缓冲（FrameBuffer）驱动程序主要依靠 4 个数据结构。定义在 include/linux/fb.h 程 序 内。它们分别是 fb_info、 fb_var_screeninfo、 fb_fix_screeninfo 和 fb_monospecs。 后 3 个结构可以在用户空间访问， 结构 fb_info 只能在内核空间访问。

fb_info 内部定义了 struct fb_ops， 结构 fb_ops 成员就是由一系列 Framebuffer 操作函数组成。

struct fb_info

{

```
atomic_t count;
int node; /*存放屏的序号，也可以说是次设备号*/
int flags;
struct mutex lock;                  /* Lock for open/release/ioctl funcs */
struct mutex mm_lock;               /* Lock for fb_mmap and smem_* fields */
struct fb_var_screeninfo var;       /*LCD 可变参数结构体*/
struct fb_fix_screeninfo fix;       /*LCD 固定参数结构体*/
struct fb_monspecs monspecs;        /*LCD 显示器标准*/
struct work_struct queue;           /*帧缓冲事件队列*/
struct fb_pixmap pixmap;            /*图像硬件 mapper*/
struct fb_pixmap sprite;            /*光标硬件 mapper*/
struct fb_cmap cmap;                /*当前的颜色表*/
struct list_head modelist;          /* mode list */
struct fb_videomode *mode;          /*当前的显示模式*/

#ifdefCONFIG_FB_BACKLIGHT
    struct backlight_device *bl_dev;    /*对应的背光设备*/
    /* Backlight level curve */
    struct mutex bl_curve_mutex;
    u8 bl_curve[FB_BACKLIGHT_LEVELS];/*背光调整*/
#endif
#ifdefCONFIG_FB_DEFERRED_IO
    struct delayed_work deferred_work;      /*延时工作队列*/
    struct fb_deferred_io *fbdefio;
#endif
struct fb_ops *fbops;           /*对底层硬件操作的函数指针*/
struct device *device;          /*内嵌的设备模型*/
struct device *dev;             /*fb 设备*/
int class_flag;                 /* private sysfs flags */
#ifdefCONFIG_FB_TILEBLITTING
    struct fb_tile_ops *tileops;    /*图块 Blitting*/
#endif
char __iomem *screen_base;      /*虚拟基地址*/
unsigned long screen_size;      /*LCD IO 映射的虚拟内存大小*/
void *pseudo_palette;           /*伪 16 色颜色表*/
#define FBINFO_STATE_RUNNING 0
#define FBINFO_STATE_SUSPENDED 1
u32 state;              /*LCD 的挂起或恢复状态*/
void *fbcon_par;        /* fbcon use-only private area */
void *par;              /*LCD 驱动的私有数据*/
struct apertures_struct
{
    unsigned int count;
    struct aperture
```

```
        {
            resource_size_t base;
            resource_size_t size;
        } ranges[0];
    } *apertures;
};
```

fb_var_screeninfo 结构体主要记录用户可以修改的控制器的参数， 比如屏幕的分辨率和每个像素的比特数等，该结构体定义如下：

```
struct fb_var_screeninfo
{
    __u32 xres;         /*可见屏幕一行有多少个像素点*/
    __u32 yres;         /*可见屏幕一列有多少个像素点*/
    __u32 xres_virtual; /*虚拟屏幕一行有多少个像素点*/
    __u32 yres_virtual; /*虚拟屏幕一列有多少个像素点*/
    /*虚拟到可见之间的偏移*/
    __u32 xoffset;     /*虚拟到可见屏幕之间的行偏移*/
    __u32 yoffset;     /*虚拟到可见屏幕之间的列偏移*/
    __u32 bits_per_pixel; /*每个像素的位数即 BPP*/
    __u32 grayscale;          /*非 0 时，指的是灰度*/
    struct fb_bitfield red;    /*fb 缓存的 R 位域*/
    struct fb_bitfield green; /*fb 缓存的 G 位域*/
    struct fb_bitfield blue;   /*fb 缓存的 B 位域*/
    struct fb_bitfield transp; /*透明度*/
    __u32 nonstd; /* != 0 非标准像素格式*/
    __u32 activate; /* see FB_ACTIVATE_* */
    __u32 height;      /*高度*/
    __u32 width;       /*宽度*/
    __u32 accel_flags; /* (OBSOLETE) see fb_info.flags */
    /*除 pixclock 本身外，其他都以像素时钟为单位*/
    __u32 pixclock; /*像素时钟(皮秒)*/
    __u32 left_margin; /*左边距， 对应 TFT 控制器时序的水平前沿信*/
    __u32 right_margin; /*右边距， 对应 TFT 控制器时序的水平后沿信*/
    __u32 upper_margin; /*上边距， 对应 TFT 控制器时序的垂直前沿信*/
    __u32 lower_margin; /*下边距， 对应 TFT 控制器时序的垂直后沿信*/
    __u32 hsync_len; /*水平同步的长度*/
    __u32 vsync_len; /*垂直同步的长度*/
    sync; /* see FB_SYNC_* */
    __u32 vmode; /* see FB_VMODE_* */
    __u32 rotate; /*顺时针旋转的角度*//* angle we rotate counter clockwise */
    __u32 reserved[5]; /* Reserved for future compatibility */
};
open("/dev/fbx");
mmap();
write(颜色值);   //显示指定的颜色
```

LCD 驱动： 固定的 主设备号 29

fbmem.c //内核自带 frmebuffer 原理实现

重点： fb_info、 fb_var_screeninfo；

内部函数实现编写在 linux-3.5/drivers/video/fbmem.c （1480 和 1816 行）

```
01480: static const struct file_operations fb_fops = {
01481:     .owner =    THIS_MODULE,
01482:     .read =     fb_read,
01483:     .write =    fb_write,
01484:     .unlocked_ioctl = fb_ioctl,
01485: #ifdef CONFIG_COMPAT
01486:     .compat_ioctl = fb_compat_ioctl,
01487: #endif
01488:     .mmap =     fb_mmap,
01489:     .open =     fb_open,
01490:     .release =  fb_release,
01491: #ifdef HAVE_ARCH_FB_UNMAPPED_AREA
01492:     .get_unmapped_area = get_fb_unmapped_area,
01493: #endif
```

```
01799: static int __init
01800: fbmem_init(void)
01801: {
01802:     proc_create("fb", 0, NULL, &fb_proc_fops);
01803:
01804:     if (register_chrdev(FB_MAJOR,"fb",&fb_fops))
01805:         printk("unable to get major %d for fb devs\n", FB_MAJOR);
01806:
01807:     fb_class = class_create(THIS_MODULE, "graphics");
01808:     if (IS_ERR(fb_class)) {
01809:         printk(KERN_WARNING "Unable to create fb class; errno = %ld\n", PTR_ERR(fb_class));
01810:         fb_class = NULL;
01811:     }
01812:     return 0;
01813: }
```

```
01816: module_init(fbmem_init);
01817: static void __exit
01818: fbmem_exit(void)
01819: {
01820:     remove_proc_entry("fb", NULL);
01821:     class_destroy(fb_class);
01822:     unregister_chrdev(FB_MAJOR, "fb");
01823: }
01824:
01825: module_exit(fbmem_exit);
```

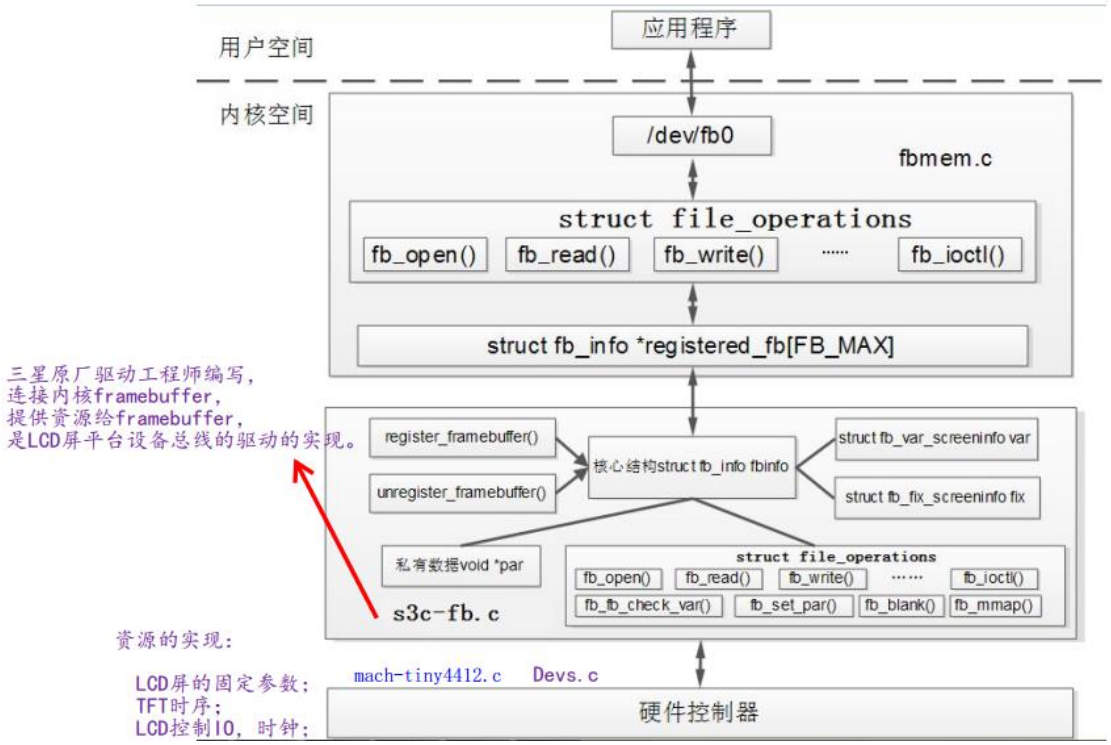
用户
操作LCD屏　　open　read　write等

fb0-----29

framebuffer核心层　　fb_open　fb_read fb_write　等

调用　struct fb_info *info;
　　struct fb_ops *fbops;　　(fb_open　fb_read等)

framebuffer 原理实现




```
file->private_data = info;
if (info->fbops->fb_open) {
    res = info->fbops->fb_open(info,1);
```

```
01448:      if (info->fbops->fb_open) {
01449:          res = info->fbops->fb_open(info,1);
```

fbmem.c//内核自带

s3c-fb.c //三星驱动工程师编写 ---- 我们需要搞懂整个操作框架，不需要编写；

以上是驱动层： 跳转到设备层 跳转见文档截图 Devs.c 419

Mach-tiny4412.c ---- 编写的设备资源 1330(资源定义) 2250(资源传递)

分析： 平台设备总线的驱动层 s3c-fb.c

```
static struct platform_driver s3c_fb_driver = {
    .probe      = s3c_fb_probe,
    .remove     = __devexit_p(s3c_fb_remove),
    .id_table   = s3c_fb_driver_ids,
    .driver     = {
        .name   = "s3c-fb",
        .owner  = THIS_MODULE,
        .pm = &s3cfb_pm_ops,
    },
};

module_platform_driver(s3c_fb_driver);
```

有 id_table 就用这个进行匹配设备层

层层跳转： Devs.c ---- 平台设备总线设备层注册的过程：

mach-tiny4412.c --- 加载

```
01922: static void __init smdk4x12_map_io(void)
 01926:     exynos_init_io(NULL, 0);
00381:     s3c_init_cpu(samsung_cpu_id, cpu_ids, ARRAY_SIZE(cpu_ids)
static struct cpu_table cpu_ids[] __initdata = {
384: static void __init exynos4_map_io(void)
00421:     s5p_fb_setname(0, "exynos4-fb");
```

LCD 屏平台设备总线的匹配过程、 由驱动层到设备层的过程。

```
03327:      .id_table   = s3c_fb_driver_ids,
00421:      s5p_fb_setname(0, "exynos4-fb");
00030: static inline void s5p_fb_setname(int id, char *name)
419: struct platform_device s5p_device_fimd0 = {
```

open("/dev/fd0");

fbmem.c //内部 frameburrer 核心层实现代码

s3c-fb.c //驱动层

Devs.c //设备层

关心资源： 假：修改 LCD 屏驱动、 改资源；

1、 找驱动层 --- 需要什么资源， 在 s3c-fb.c，platform_get_resource 函数

```
02730:    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
02744:    res = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
```

```
struct s3c_fb_platdata {
    void    (*setup_gpio)(void);

    struct s3c_fb_pd_win    *win[S3C_FB_MAX_WIN];
    struct fb_videomode     *vtiming;

    u32             vidcon0;
    u32             vidcon1;
};
```

```
struct s3c_fb_pd_win {
    unsigned short      default_bpp;
    unsigned short      max_bpp;
    unsigned short      xres;
    unsigned short      yres;
    unsigned short      virtual_x;
    unsigned short      virtual_y;
    unsigned short      width;
    unsigned short      height;
};
```

```
struct fb_videomode {
    const char *name;   /* optional */
    u32 refresh;        /* optional */
    u32 xres;
    u32 yres;
    u32 pixclock;
    u32 left_margin;
    u32 right_margin;
    u32 upper_margin;
    u32 lower_margin;
    u32 hsync_len;
    u32 vsync_len;
    u32 sync;
    u32 vmode;
    u32 flag;
};
```

```
02249:      tiny4412_fb_init_pdata(&smdk4x12_lcd0_pdata);
02250:      s5p_fimd0_set_platdata(&smdk4x12_lcd0_pdata);
```

设置平台驱动的设备层资源

```
s3c_set_platdata(pd, sizeof(struct s3c_fb_platdata),
        &s5p_device_fimd0);
```

需要：屏的固定参数、 TFT 时序、 LCD 屏控制器的时钟以及 GPIO

fbmem.c:内核维护人员编写， 在驱动层的基础上再次封装了一层， 提供了帧缓冲操作，
让用户更容易操作 LCD 屏。

s3c-fb.c：三星原厂驱动工程师 连接内核 framebuffer 提供资源给 framebuffer，
是 LCD 屏平台设备总线的驱动层实现。

Devs.c 和 mach-tiny4412.c --- 友善之臂编写， 平台设备总线的资源层实现，
传递的资源有屏的固定参数、 TFT 的时序、 LCD 控制器的 IO 以及时钟

主线：

1、 如何由驱动层 --- 设备层
2、 设备层和驱动层如何加载的
3、 驱动的探测函数资源分析
4、 设备层资源的传递
5、 io 分析、 时序、 时钟分析

二、LCD 显示汉字

1、HZK16 简介

HZK16字库是符合GB2312标准的16× 16点阵字库,HZK16的GB2312-80支持的汉字有6763个，符号 682 个。其中一级汉字有 3755 个，按声序排列，二级汉字有 3008 个，按偏旁部首排列。我们在一些应用场合根本用不到这么多汉字字模，所以在应用时就可以只提取部分字体作为己用。

HZK16 字库里的 16×16 汉字一共需要 256 个点来显示，也就是说需要 32 个字节才能达到显示一个普通汉字的目的。

我们知道一个 GB2312 汉字是由两个字节编码的，范围为 A1A1~FEFE。A1-A9 为符号区，B0 到 F7 为汉字区。每一个区有 94 个字符（注意：这只是编码的许可范围，不一定都有字型对应，比如符号区就有很多编码空白区域)。下面以汉字“我”为例，介绍如何在 HZK16 文件中找到它对应的 32 个字节的字模数据。

前面说到一个汉字占两个字节,这两个中前一个字节为该汉字的区号,后一个字节为该字的位号。其中，每个区记录 94 个汉字，位号为该字在该区中的位置。所以要找到“我”在 hzk16 库中的位置就必须得到它的区码和位码。(为了区别使用了区码和区号，其实是一个东西，别被我误导了)

区码：区号（汉字的第一个字节）-0xa0 (因为汉字编码是从 0xa0 区开始的，所以文件最前面就是从 0xa0 区开始，要算出相对区码) 。

位码：位号（汉字的第二个字节）-0xa0 。

这样我们就可以得到汉字在 HZK16 中的绝对偏移位置： offset=(94*(区码-1)+(位码-1))*32

注解：

1、 区码减 1 是因为数组是以 0 为开始而区号位号是以 1 为开始的

2、 (94*(区号-1)+位号-1)是一个汉字字模占用的字节数

3、 最后乘以 32 是因为汉字库文应从该位置起的 32 字节信息记录该字的字模信息(前面提到一个汉字要有 32 个字节显示)

有了偏移地址就可以从 HZK16 中读取汉字编码了，剩下的就是文件操作了。

2、LCD 显示汉字

现在我们来编写应用程序在 LCD 屏上显示文字，其中使用到一个 HZK16 文件，这个文件可以从网上下载到，存放的是 16X16 的点阵字模数据。另一个 C 文件 font_8x16.c 是 8*16 的 ASCII 文件字模数据，这个文件来自于 Linux 源码，路径 drivers\video\console\font_8x16.c,这个文件复制过来还不能直接使用，修改如下：

（1）屏蔽无关头文件，并删除掉“static”关键字。

```
//#include <linux/font.h>
//#include <linux/module.h>

#define FONTDATAMAX 4096

const unsigned char fontdata_8x16[FONTDATAMAX]
```

（2）删除无效的数组定义代码

把文件最后一段代码删除：

```
/*
const struct font_desc font_vga_8x16 = {
    .idx  = VGA8x16_IDX,
    .name    = "VGA8x16",
    .width   = 8,
    .height  = 16,
    .data    = fontdata_8x16,
    .pref    = 0,
};
EXPORT_SYMBOL(font_vga_8x16);
*/
```

3、内存映射

这里我们需要详细介绍一下：因为系统分为用户层和内核层（共同组成 4G 的内存空间），他们两个不能直接交换数据，以前我们把内核层的信息传递给用户层都是用函数 cpy_to_user 函数来实现的，但是这里因为我们的 LCD 控制器需要刷新很快，所以用这种方法显然很浪费时间。这里我们对显存进行映射（不是复制，大大加快了速度）。

Linux 提供了内存映射函数 mmap，它把文件内容映射到一段内存上(准确说是虚拟内存上)，通过对这段内存的读取和修改，实现对文件的读取和修改，

mmap 操作提供了一种机制，让用户程序直接访问设备内存，这种机制，相比较在用户空间和内核空间互相拷贝数据，效率更高。在要求高性能的应用中比较常用。mmap 映射内存必须是页面大小的整数倍，面向流的设备不能进行 mmap(fopen fread fwrite)，mmap 的实现和硬件有关。

先来看一下 mmap 的函数声明:

头文件:<unistd.h> <sys/mman.h>同时打开 fb0

原型:（在虚拟机上用命令 man 2 mmap 查看）

```
#include <sys/mman.h>

void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
```

返回值: 成功则返回映射区起始地址, 失败则返回 MAP_FAILED(-1).

参数:

addr: 指定映射的起始地址, 通常设为 NULL, 由系统指定.

length: 将文件的多大长度映射到内存.(显存的大小)

prot: 映射区的保护方式, 可以是:

PROT_EXEC: 映射区可被执行.

PROT_READ: 映射区可被读取.

PROT_WRITE: 映射区可被写入.

PROT_NONE: 映射区不能存取.

flags: 映射区的特性, 可以是:

MAP_SHARED: 对映射区域的写入数据会复制回文件, 且允许其他映射该文件的进程共享;

MAP_PRIVATE: 对映射区域的写入操作会产生一个映射的复制(copy-on-write),

对此区域所做的修改不会写回原文件;

此外还有其他几个 flags 不很常用, 具体查看 linux C 函数说明.

fd: 由 open 返回的文件描述符, 代表要映射的文件.

offset: 以文件开始处的偏移量, 必须是分页大小的整数倍, 通常 0, 表示从文件头开始映射;

显示汉字:

1、 先取模

2、 映射帧缓冲设备 ioctl 去获取设备的信息 mmap()映射设备

3、 编写画点的函数 求坐标偏移量， 将颜色值给缓冲区

4、 汉字模、 画点 --- 显示汉字 -- 电子书

三、LCD 屏显示图片

BMP 图象格式: BMP 是 bitmap 的缩写形式, bitmap 顾名思义, 就是位图也即 Windows 位图。它一般由 文件头信息块、图像描述信息块、颜色表(在真彩色模式无颜色表)和图像数据区组成。在系统中以 BMP 为扩展名保存。

1、文件头信息块 --他包含图形文件的类型, 内容尺寸以及初始的偏移量这些相关信息, 如下:

1：文件头，它包含BMP图像文件的类型、内容尺寸和起始偏移量等信息；

字节顺序	数据结构	描述
1,2	short	高8位为字母’B’，低8位为字母’M’
3,4,5,6	int	文件大小
7,8	short	保留字1
9,10	short	保留字2
11,12,13,14	int	数据部分偏移量

由上可以知道一共 14 个字节，我们可以定义一个结构体来描述以上信息：

```
/* 文件头结构 14byte */
typedef struct
{
    char cfType[2]; /* 文件类型，必须为 "BM" (0x4D42)*/
    char cfSize[4]; /* 文件的大小(字节) */
    char cfReserved[4]; /* 保留，必须为 0 */
    char cfoffBits[4]; /* 位图阵列相对于文件头的偏移量(字节)*/
    }__attribute__((packed)) BITMAPFILEHEADER;
```

2、图像描述信息块 --- 共 50 个字节

他描述了图像的参数，例如高、宽、像素等相关信息。

详细描述如下图：

15,16,17,18	int	当前结构体的大小，通常是40或56	
19,20,21,22	int	图像宽度（像素）	0x12~0x15是宽
23,24,25,26	int	图像高度（像素）	0x16~0x19是宽
27,28	short	这个字的值永远是1	说的是两个字节总和是1，
29,30（0x18,0x19）	short	每像素占用的位数，即bpp	每个像素所需的位数，必须是1(双色)、4(16色)、8(256色)、24(真彩色)之一

31,32,33,34	int	压缩方式	0x1e~0x21,值是0表示不压缩
35,36,37,38	int	水平分辨率，pixels-per-meter	
39,40,41,42	int	垂直分辨率，pixels-per-meter	
43,44,45,46	int	垂直分辨率，pixels-per-meter	
47,48,49,50	int	引用色彩数	
51,52,53,54	int	关键色彩数	

我们也可以定义一个结构体类型去描述他。

```
typedef struct
{
    char ciSize[4];          /* size of BITMAPINFOHEADER */
    char ciWidth[4];         /* 位图宽度(像素) */
    char ciHeight[4];        /* 位图高度(像素) */
    char ciPlanes[2];        /* 目标设备的位平面数, 必须置为1 */
    char ciBitCount[2];      /* 每个像素的位数, 1,4,8或24 */
    char ciCompress[4];      /* 位图阵列的压缩方法,0=不压缩 */
    char ciSizeImage[4];     /* 图像大小(字节) */
    char ciXPelsPerMeter[4];/* 目标设备水平每米像素个数 */
    char ciYPelsPerMeter[4];/* 目标设备垂直每米像素个数 */
    char ciClrUsed[4];       /* 位图实际使用的颜色表的颜色数 */
    char ciClrImportant[4]; /* 重要颜色索引的个数 */
}__attribute__((packed)) BITMAPINFOHEADER;
```

3、颜色表：每 4 字节表示一种颜色，并以 B（蓝色）、G（绿色）、R（红色）、alpha（32 位位图的透明度值，一般不需要）。即首先 4 字节表示颜色号 1 的颜色，接下来表示颜色号 2 的颜色，依此类推。

```
//位图的颜色结构体(888)
typedef struct
{
    unsigned short blue:8;
    unsigned short green:8;
    unsigned short red: 8;
}__attribute__((packed)) PIXEL;
```

4、图像数据区：颜色表接下来位为位图文件的图像数据区，在此部分记录着每点像素对应的颜色号，其记录方式也随颜色模式而定，既 2 色图像每点占 1 位（8 位为 1 字节）；16 色图像每点占 4 位（半字节）；256 色图像每点占 8 位（1 字节）；真彩色图像每点占 24 位（3 字节）。所以，整个数据区的大小也会随之变化。

究其规律而言，计算公式如下：图像数据信息大小=（图像宽度*图像高度*记录像素的位数）/8。

注意： 图像数据字节阵列，一般都是正向的，扫描行由底向上进行存储，

也就是说，阵列中最开始的字节表示图像的左下角。

5、位图数据：记录了位图的每一个像素值，记录顺序是在扫描行内是从左到右,扫描行之间是从下到上。

位图的一个像素值所占的字节数:

当 biBitCount=1 时，8 个像素占 1 个字节;

当 biBitCount=4 时，2 个像素占 1 个字节;

当 biBitCount=8 时，1 个像素占 1 个字节;

当 biBitCount=24 时,1 个像素占 3 个字节;

显示图片

1、 解析 bmp 图片(1、 文件信息头 2、 文件信息描述块 3、 数据区)

2、 读信息头(BM 文件大小 数据区的偏移量) //转换为需要的 int

3、 读取文件信息描述块(图片宽、 高、 bpp)

4、 画图，定位到 bmp 图片数据偏移区(存放图片的颜色值， 由左下角开始，由左到右)

-----显示任意大小的 bmp 图片

附部分函数介绍：

1、Linux 提供了内存映射函数 mmap, 它把文件内容映射到一段内存上(准确说是虚拟内存上), 通过对这段内存的读取和修改, 实现对文件的读取和修改, mmap 操作提供了一种机制，让用户程序直接访问设备内存, 这种机制, 相比较在用户空间和内核空间互相拷贝数据, 效率更高。在要求高性能的应用中比较常用。mmap 映射内存必须是页面大小的整数倍，面向流的设备不能进行 mmap，mmap 的实现和硬件有关原型:

void * mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offsize);

返回值: 成功则返回映射区起始地址, 失败则返回 MAP_FAILED(-1).

addr: 指定映射的起始地址, 通常设为 0, 由系统指定.

length: 将文件的多大长度映射到内存.

prot: 映射区的保护方式, 可以是:

PROT_EXEC: 映射区可被执行.

PROT_READ: 映射区可被读取.

PROT_WRITE: 映射区可被写入.

PROT_NONE: 映射区不能存取.

flags: 映射区的特性, 可以是:

MAP_SHARED: 对映射区域的写入数据会复制回文件, 且允许其他映射该文件的进程共享.

MAP_PRIVATE: 对映射区域的写入操作会产生一个映射的复制(copy-on-write),对此区域所做的修改不会写回原文件.

此外还有其他几个 flags 不很常用, 具体查看 linux C 函数说明.

fd: 由 open 返回的文件描述符, 代表要映射的文件.

offset: 以文件开始处的偏移量, 必须是分页大小的整数倍, 通常为 0, 表示从文件头开始映射

2、stat，lstat，fstat1 函数都是获取文件（普通文件，目录，管道，socket，字符，块)的属性。

函数原型#include <sys/stat.h>

int stat(const char *restrict pathname, struct stat *restrict buf);提供文件名字，

获取文件对应属性。

int fstat(int filedes, struct stat *buf);通过文件描述符获取文件对应的属性。失败返回

int lstat(const char *restrict pathname, struct stat *restrict buf);连接文件描述命，

获取文件属性。

```
struct stat
{
    mode_t st_mode; //文件对应的模式，文件，目录等
    ino_t st_ino; //inode 节点号
    dev_t st_dev; //设备号码
    dev_t st_rdev; //特殊设备号码
    nlink_t st_nlink; //文件的连接数
    uid_t st_uid; //文件所有者
    gid_t st_gid; //文件所有者对应的组
    off_t st_size; //普通文件，对应的文件字节数
    time_t st_atime; //文件最后被访问的时间
    time_t st_mtime; //文件内容最后被修改的时间
    time_t st_ctime; //文件状态改变时间
    blksize_t st_blksize; //文件内容对应的块大小
    blkcnt_t st_blocks; //文件内容对应的块数量
};
```

这三个系统调用都可以返回指定文件的状态信息，这些信息被写到结构 struct stat 的缓冲区中。
通过分析这个结构可以获得指定文件的信息。

3、exec 函数族

exec*由一组函数组成

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg , ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char * path, char *const argv[], char *const envp[]);
```

exec 函数族的作用是运行第一个参数指定的可执行程序。但其工作过程与 fork 完全不同，
fork 是在复制一份原进程，而 exec 函数执行第一个参数指定的可执行程序之后，这个新程序运行起来后也是一个进程，而这个进程会覆盖原有进程空间，即原有进程的所有内容都被新运行起来的进程全部覆盖了，所以 exec 函数后面的所有代码都不再执行，但它之前的代码当然是可以被执行的。

path 是包括执行文件名的全路径名 file 既可是全路径名也可是可执行文件名

arg 是可执行文件的全部命令行参数， 可以用多个，注意最后一个参数必须为 NULL。

argv 是一个字符串的数组 char *argv[]={“full path”,”param1”,”param2”,...NULL};

envp 指定新进程的环境变量 char *envp[]={“name1=val1”,”name2=val2

例如：

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    execl(“./hello_app”,”./hello_app”,”1.bmp”,NULL);
    retrurn 0;
}
```