图形用户接口实验报告

戴文谦
141180014

目录

1 实验环境介绍

1.1 硬件环境

硬件开发环境即为嵌入式实验系统开发板 x210v3。本次实验是实现嵌入式系统图形界面，故最重要的硬件资源是开发板自带的显示屏。包括显示屏在内，本次试验用到的硬件资源如下：

- 7 英寸，32 位 LCD，物理分辨率为 800*480
- 核心处理器：S5PV210，主频 1GHz
- 内存：512MB

- RS–232 接口 (UART2)
- RJ45 接口 (有线以太网 DM9000CEP)
- 功能按键 (开机/复位)

1.2 软件环境

软件环境主机端主要有以下配置：

- Ubuntu 桌面环境（包含 minicom 工具）。
- 位于主机上的 busybox 1.20.2[1]，作为嵌入式系统使用的 nfs 文件系统[2]
- Linux 内核 3.0.8 版本，包含三星设备驱动并开启了 frambuffer 相关功能[3]
- 针对 ARM 的交叉编译工具链 2016.08 版本，并加入环境变量中[4]

在嵌入式开发板 x210v3 端，厂商预装了底层的 bootloader 和 Android4.0.4 系统。开发板的 Bootloader 环境在嵌入式系统内核与文件系统实验中已经配置好网络地址和 nfs 文件系统启动的环境变量，可在本次实验中直接使用。本次实验与开发板板载 FLASH 芯片中的 Android 系统无关。

2 实验原理说明

2.1 Framebuffer 的相关概念

Frambuffer 是 linux2.2xx 内核以后提供的一种驱动程序接口。这种接口将显示设备抽象为帧缓冲区，允许应用软件通过 frambuffer 的接口来访问图像硬件设备。这样，软件无需了解涉及硬件底层驱动的操作细节。对于用户，可将 frambuffer 看做显示内存的一个映像，通过/dev 下的 fb0、fb1 等设备节点可对该设备进行访问。作为帧缓冲设备，frambuffer 也属于字符设备，访问方式与常规平台设备类似。frambuffer 获得用户应用程序的指令、数据后，与 LCD 控制器的驱动程序进行交互，进而实现访问显示设备的功能。

实际上，frambuffer 在开发中最主要的意义是在面向用户的层面上提供了一组标准化访问图形设备的接口和一组对图形设备的规范化描述，其本身的实现与一般的驱动程序有一定类似之处，都是基于面向对象的设计，不同之处在于其接口是抽象的，不包含物理层面的具体实现，因此才能实现在同一内核版本上的统一调用。

2.2 设备操作原理

对于用户程序而言，它和其他的设备并没有什么区别，用户可以把 frambuffer 看成是一块内存，既可以向内存中写数据，也可以读数据。Frambuffer 的显示缓冲区位于内核空间，应用程序可以把此空间映射到自己的用户空间，再进行操作。

[1] 安装地址为/srv/nfs4/nfslinux

[2] 已配置了开机自动挂载主机 nfs 文件夹至/mnt

[3] 实验前期使用 2.6.35 版本，后改用 3.0.8 版本

[4] 包含 GNU 工具：gcc compiler，binutil，C library，C header

在应用程序中，操作/dev/fbn 或其它设备文件一般调用以下函数：

1. `int open(const char *pathname, int flags)`：打开设备文件并返回设备文件描述符 fd，通过设备文件描述符为其它函数指明操作对象。
2. `int ioctl(int fd, int request, ...)`：传入控制字和参数，对设备的一些特性进行控制。在操作 frambuffer 时，一般用于读取或改变屏幕的参数属性。
3. `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)`：将指定大小的内核空间内的内存映射到用户空间，在操作 frambuffer 时常用于将屏幕缓冲区映射到用户空间，再向其中写入要显示的内容。
4. `int munmap(void *addr, size_t length)`：释放由 mmap() 函数映射的内存空间。在对 frambuffer 操作结束，退出程序前需要执行该函数来释放映射。
5. `int close(int fd)`：关闭设备文件，释放设备文件描述符 fd。

2.3 Framebuffer 内核实现分析

以下分析基于 linux2.6.35 版本内核源码。

由于本实验的主要目标是制作基于嵌入式系统的图形界面，对于 frambuffer 的研究主要基于其结构和内存操作。故主要分析以下三个内核文件：

```
linux-2.6.35.6/include/linux/fb.h
linux-2.6.35.6/drivers/video/fbmem.c
linux-2.6.35.6/drivers/video/samsung/s3cfb.c
```

其中最后一个文件是 frambuffer 调用的底层驱动程序。选取其中与图形界面实验相关的部分的代码进行分析：[1]

首先是声明 frambuffer 信息的结构体类型`fb_info`：

```
struct fb_info {
    struct fb_var_screeninfo var;
    struct fb_fix_screeninfo fix;
    struct fb_cmap cmap;          /* 当前的调色板 */
    struct fb_ops *fbops;         /* 说明操作frambuffer的函数 */
    struct device *device;        /* 指明该frambuffer设备的父设备 */
    struct device *dev;           /* 指明该frambuffer设备 */
    void *par;                    /* 该设备自己分配的空间 */
};
```

`fb_var_screeninfo` 是说明可变屏幕参数的结构体类型：

[1]分析方式采用为源代码加注释的方式。篇幅所限，进行了大量删减，只节选了与本次实验关联较大的部分分析，且对于底层驱动程序的调用全部写出篇幅过多，因此仅简单涉及。同样出于节省篇幅、突出重点的原因，对代码有微量改动

```
struct fb_var_screeninfo {
    ___u32 xres;                    /* 物理可见横向分辨率 */
    ___u32 yres;                    /* 物理可见纵向分辨率 */
    ___u32 xres_virtual;            /* 虚拟横向分辨率，对应可操作显存空间 */
    ___u32 yres_virtual;            /* 虚拟纵向分辨率 */
    ___u32 xoffset;                 /* 物理分辨率横向偏移显示 */
    ___u32 yoffset;                 /* 物理分辨率纵向偏移显示 */
    ___u32 bits_per_pixel;          /* 每个像素占用的显存空间位数 */
    ___u32 grayscale;               /* 该项不为0时用灰阶表示颜色 */
    struct fb_bitfield red;         /* 每个像素占用空间中红色的表示方式 */
    struct fb_bitfield green;       /* 每个像素占用空间中绿色的表示方式 */
    struct fb_bitfield blue;        /* 每个像素占用空间中蓝色的表示方式 */
    struct fb_bitfield transp;      /* 每个像素占用空间中透明度的表示方式 */
    ___u32 activate;                /* 激活状态 */
};
```

激活状态 activate 有以下不同定义以影响控制字`FBIOPUT_VSCREENINFO` 的行为：

```
#define FB_ACTIVATE_NOW 0 /* 可立即改变参数 */
#define FB_ACTIVATE_NXTOPEN 1 /* 下次打开设备文件时改变参数 */
#define FB_ACTIVATE_TEST 2 /* 不改变参数 */
#define FB_ACTIVATE_MASK 15 /* 通过按位与检测上述参数的标签 */
#define FB_ACTIVATE_FORCE 128 /* 强制更新屏幕可变参数 */
#define FB_ACTIVATE_INV_MODE 256 /* 关闭视频模式 */
```

`fb_bitfield` 是说明颜色表示方式的结构体类型：

```
struct fb_bitfield {
    ___u32 offset;                  /*  bitfield 偏移位数 */
    ___u32 length;                  /*  bitfield 占用的位数 */
    ___u32 msb_right;               /* 符号位是否在最右 */
};
```

`fb_fix_screeninfo` 是说明不可变屏幕参数的结构体类型：

```
struct fb_fix_screeninfo {
    char id[16];                    /* 屏幕的名称 */
    unsigned long smem_start;       /* framebuffer在物理内存中的开始地址 */
    ___u32 smem_len;                /* framebuffer占用物理内存长度 */
    ___u32 type;                    /* frambuffer类型 */
    ___u32 visual;                  /* 设备颜色信息 */
    ___u32 line_length;             /* 一行的字节数 */
    unsigned long mmio_start;       /* 寄存器映射在物理内存中的开始地址 */
    ___u32 mmio_len;                /* 寄存器映射占用物理内存长度 */
```

};

framebuffer 提供的一部分 ioctl 功能及其在 fb.h 中定义的宏如下：

```
#define FBIOGET_VSCREENINFO 0x4600
#define FBIOPUT_VSCREENINFO 0x4601
#define FBIOGET_FSCREENINFO 0x4602
#define FBIOPAN_DISPLAY 0x4606
#define FBIOBLANK       0x4611

static long do_fb_ioctl(struct fb_info *info, unsigned int cmd, unsigned long arg){
    switch (cmd) {
    case FBIOGET_VSCREENINFO:
        /* 将实际可变屏幕参数的结构体拷贝到用户空间 */
        var = info->var;
    case FBIOPUT_VSCREENINFO:
        /* 调用底层驱动程序按用户空间说明可变屏幕参数的结构体对屏幕进行设置 */
        ret = fb_set_var(info, &var);
    case FBIOGET_FSCREENINFO:
        /* 将实际不可变屏幕参数的结构体拷贝到用户空间 */
        fix = info->fix;
    case FBIOPAN_DISPLAY:
        /* 调用底层驱动程序按用户空间可变屏幕参数的结构体对屏幕进行设置 */
        /* 根据底层驱动代码，对于实验平台该函数只改变纵向偏移显示参数 */
        ret = fb_pan_display(info, &var);
    case FBIOBLANK:
        /* 调用底层驱动程序对屏幕节能选项进行设置 */
        ret = fb_blank(info, arg);
    ...
}
```

其中`FBIOBLANK` 控制字的参数有如下定义：

```
enum {
FB_BLANK_UNBLANK = VESA_NO_BLANKING, /* 关闭屏幕空白 */
FB_BLANK_NORMAL = VESA_NO_BLANKING + 1, /* 打开屏幕空白 */
FB_BLANK_VSYNC_SUSPEND =
                        VESA_VSYNC_SUSPEND + 1, /* 停止垂直同步 */
FB_BLANK_HSYNC_SUSPEND =
                        VESA_HSYNC_SUSPEND + 1, /* 停止水平同步 */
FB_BLANK_POWERDOWN = VESA_POWERDOWN + 1 /* 切断屏幕电源 */};
```

根据底层驱动程序定义，停止垂直同步、水平同步的功能不可用。

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

这两个是提供给下层图形设备驱动的接口，图形设备驱动通过这两函数向系统注册或注销自己。

3 实验过程与内容

3.1 软件结构的设计

本次实验的重点在开发一组图形用户接口，实际上是提供一组绘图函数和相应的头文件供应用程序调用。为供一般的应用程序调用，这组接口需要封装一定的操作细节，只需要用户提供画图参数就可以进行图形界面的设计；另外，为实现封装功能，需要把绘图函数独立成一个函数库，提供头文件和函数库文件供用户程序调用编译。

在 linux 操作系统下独立函数库主要有以下三种格式：

1. “.o”，即 C 语言源文件独立编译后的文件，缺点是无法把多个功能近似的源文件编成一个库文件，会造成文件结构臃肿，一般不适合作为分发的函数库，只作为对象模块而存在。[1]

2. “.a”，由归档管理器 ar 生成的静态库文件，可以将多个编译后的“.o”文件归档。在调用静态链接库生成程序时，与“.o”文件起到相同的作用，将被调用的库函数直接编入程序中。需要生成可独立执行的程序时，需要静态编译程序，不能通过动态链接库来生成，此时就需要静态链接库。实际上相当于多个“.o”文件的组合。[2]

3. “.so”，是将“.o”文件链接而成的动态链接库文件。在调用动态链接库生成程序时，实际上只是为程序中用到的库函数提供了指向动态链接库的符号链接，相应的函数执行过程只保存在动态库中，因此可以大大减小可执行文件的体积，有利于软件结构的模块化，为程序提供了一定的灵活性和可拓展性。执行动态编译的程序时，程序并无实际的函数功能代码，需要依赖于环境变量指定目录下的动态链接库，通过执行时加载库函数代码实现功能，因此不能独立运行。[3]

本次实验的文件存放在 `~/dwq/github/kernel_learn/frambuffer` 下，利用 git 进行版本控制，具体细节不再赘述。

函数库首先需要提供对 frambuffer 初始化、关闭等方式的框架。根据对用户暴露的底层细节和操作程度不同，有多种封装方式可以选择。具体而言，比如屏幕的参数，指向显存空间的指针，frambuffer 设备文件的设备描述符等是否对用户开放都是可选的。本次实验文件的编写选择了开放指向显存空间的指针。[4]

/ frambuffer.h */*
enum ScreenNum{V_SCREEN0, V_SCREEN1};

[1]类似 windows 操作系统下的.obj 文件
[2]类似 windows 操作系统下的.lib 文件
[3]类似 windows 操作系统下的.dll 文件
[4]实验报告中所有的代码都为节省篇幅做了一定调整删减。同时，在4.1节中探讨了该程序结构是否合理

```
typedef struct fb_var_screeninfo vinfo;
typedef struct fb_bitfield fbcolor;
typedef struct{unsigned char blue, green, red, alpha;}color_8;
/* frambuffer.c */
static enum ScreenNum SCREEN_USING;
unsigned char *fb_init(){
    vinfo fbinfo;
    fd = open ("/dev/fb0", O_RDWR);
    ioctl(fd, FBIOGET_VSCREENINFO, &fbinfo);
    xres = fbinfo.xres;yres = fbinfo.yres;xvres = fbinfo.xres_virtual;
    //虚拟分辨率为实际分辨率的两倍
    yvres = yres * (V_SCREEN1 - V_SCREEN0 + 1);
    fbinfo.yres_virtual = yvres;
    bits_per_pixel = fbinfo.bits_per_pixel;
    screenpixel = xres * yres;
    red = fbinfo.red;
    ...
    alpha = fbinfo.transp;
    //计算frambuffer占用内存字节数，需要按虚拟分辨率来计算分配内存以使用双屏功能
    screen0size = screenpixel * bits_per_pixel / BIT_TO_BYTE;
    screensize = screen0size * yvres / yres;
    //fb0屏幕初始化
    fbinfo.activate = FB_ACTIVATE_FORCE;
    fbinfo.yoffset = 0;
    SCREEN_USING = V_SCREEN0;
    ioctl(fd, FBIOPUT_VSCREENINFO, &fbinfo);
    return ((unsigned char *)mmap(NULL, screensize, PROT_READ
    | PROT_WRITE, MAP_SHARED, fd, 0));
}
void wipe_allscreen(unsigned char* buff){
    memset(buff, 0, screensize);
}
void release_fb(unsigned char* buff){
    munmap(buff, screensize);
}
void make_color(color_8 *out, unsigned char red, unsigned char green,
 unsigned char blue, unsigned char alpha){
    out->red = red;
    ....
}
```

为了检验函数库的功能，需要通过测试程序调用函数库模拟实际用户程序。下以本实验用到的 Makefile 说明程序的编译结构。每次更新程序后只需在终端下输入 `make` 与 `make cp` 即可完成编译与拷贝库文件、测试程序到开发板的工作。

```
CC = arm-linux-gcc
LIBS = libfb.so
PROGRAM = fbrun fbclean fbbmp fbflash fbblock fb2block
all : $(PROGRAM)
fb2block : fb2block.o libfb.so
    $(CC) -o $@ $< -L ./ -l fb
...
libfb.so : frambuffer.o
    $(CC) -shared -o $@ $<
%.o : %.c
    $(CC) -c  $<
.PHONY : clean cp
clean :
    rm -f *.o
cp :
    cp $(PROGRAM) /srv/nfs4/fb
    cp $(LIBS) /srv/nfs4/nfslinux/lib
```

3.2 基本绘图功能的实现

以下库函数分别提供画点、画线、画方块的功能。[1]

```
void draw_point(unsigned char *buff, uint32_t x, uint32_t y, color_8 color){
    //设置画点内存空间指针位置
    uint32_t offset = (y * xres + x) * bits_per_pixel / BIT_TO_BYTE;
    //由于实际设备单个像素占用4字节，故可以一次性写入单个像素提高效率
    *(uint32_t *)(buff + offset) = (color.alpha << alpha.offset) | (color.red << red.
    offset) | (color.green << green.offset) | (color.blue << blue.offset);
}
void draw_line(unsigned char *buff, int xmin, int xmax, int ymin, int ymax,
 color_8 color){
    double ratio;
    ratio = (double)(ymax - ymin) / (double)(xmax - xmin);
    for(x = xmin; x < xmax; x++){
        y = (uint32_t)(ratio * (x - xmin) + ymin);
        draw_point(buff, x, y, color);
    }
```

[1] 省略校正输入正确性部分

```
}
void draw_block(unsigned char *buff, int xmin, int xmax, int ymin, int ymax,
 color_8 color){
    for (int j = ymax; j > ymin; j--)
        for (int i = xmin; i < xmax; i++)
            draw_point(buff, i, j, color );
}
```

通过 minicom 进入开发板上的 linux 系统[1]，进入 `/mnt/fb` 文件夹运行测试程序 `fbrun` 可以测试以上功能，该测试程序运行后可以绘制线段，比较简单不再介绍。

3.3 绘制 BMP 图片的实现

BMP（全称 Bitmap）是 Windows 操作系统中的标准图像文件格式，本实验中根据文件定义结构来读入 BMP 文件并将其显示在开发板的屏幕上。

BMP 文件的格式定义如下：

```
//通知预编译器该部分结构体进行1字节对齐，若采用默认的4字节对齐则文件头读取出错
#pragma pack(1)
typedef struct{
    char f_Type[2];
    uint32_t f_Size;
    uint16_t f_Reserved_1;
    uint16_t f_Reserved_2;
    uint32_t f_Offset;          //图片内容相对文件头的偏移
}BMP_head;
typedef struct{
    uint32_t Size;
    int32_t Width;          //图片的宽度
    int32_t Height;         //图片的高度，大于0时图片行实际上为反向记录
    int16_t Planes;
    int16_t BitCount;       //图片的色彩位数
    uint32_t Compression;
    uint32_t SizeImage;
}BMP_info;
#pragma pack()
```

定义 BMP 文件头后，可以定义以下库函数，提供给应用程序画图功能：

```
void draw_BMP(unsigned char* buff, char path[], uint32_t xbias, uint32_t ybias){
    FILE *fp;
    if (fp = fopen(path, "r")){
```

[1]采用内核映像为 zImage3.0nfs，文件系统为 nfs，操作方式与嵌入式系统内核与文件系统实验中相同

```
        BMP_head head;
        BMP_info info;
        //读取BMP头，结构信息后跳转到位图内容
        if (1 != fread(&head, sizeof(BMP_head), 1, fp)){return;}
        if (1 != fread(&info, sizeof(BMP_info), 1, fp)){return;}
        if(fseek(fp, head.f_Offset, SEEK_SET)){return;}
        //由于BMP存在多种不同的位色彩设置，调用不同的子函数进行实际绘图
        switch(info.BitCount){
            case 24:draw_BMP_24(buff, fp, &info, xbias, ybias);   break;
            case 32:draw_BMP_32(buff, fp, &info, xbias, ybias);   break;
            default:break;
        }
        fclose(fp);
    }
}
//以常见的24位色（没有alpha通道）子函数示例，省略32位色的实现
void draw_BMP_24(unsigned char *buff, FILE *fp, BMP_info *info, uint32_t xbias,
 uint32_t ybias){
    //BMP文件格式中规定每行长度必须为4字节的倍数
    //24位色表示下每个像素点对应3字节，故BMP文件会在行末自动补0，显示时需要跳过
    //width_offset计算每行跳过的字节数
    int width_offset = info->Width * 3 / 4 * 4 + 4 - info->Width * 3;
    color_8 color;
    //高度信息大于0表示BMP图片记录顺序是从最末行向第一行的倒序，反之为正序
    if (info->Height > 0){
        for (y = info->Height + ybias; y > ybias; y--){
            for(x = xbias; x < info->Width + xbias; x++){
                //color_8结构体的定义与BMP格式中对色彩的定义一致，可直接读入
                fread(&color, 3, 1, fp);
                color.alpha = 0;
                //y定义为无符号整数，故此处做了一点调整
                draw_point(buff, x, y - 1, color);
            }
            fseek(fp, width_offset, SEEK_CUR);
        }
    }
    else if(info->Height < 0){
        int tmpheight = 0 - info->Height;
        for (y = ybias; y < tmpheight + ybias; y++)
        ....
```

```
    }
}
```

该功能的测试文件为 `fbbmp`，运行后会迅速闪过 5 幅图片，实现较为简单，不再赘述。

3.4 动态图形绘制的实现

动态图形绘制实际上是多帧绘制。想实现多帧绘制，绘制第二帧时需要擦除第一帧的内容。擦除可以在单屏或双屏的情况下实现。单屏擦除即直接清空屏幕并绘制下一帧；双屏擦除的方式是交替显示双屏内容，先清空不可见屏绘制下一帧图形，再重新设置显示偏移来显示不可见屏，以此往复。其中切换屏幕的函数和清空不可见屏的函数可以用如下方式实现：

```
/* frambuffer.c */
void switch_vscreen(){
    vinfo fbinfo;
    ioctl(fd, FBIOGET_VSCREENINFO, &fbinfo);
    switch(SCREEN_USING){
        case V_SCREEN0:fbinfo.yoffset = yres; break;
        case V_SCREEN1:fbinfo.yoffset = 0; break;
    }
    ioctl(fd, FBIOPAN_DISPLAY, &fbinfo);//此处不可以使用其它控制字
    SCREEN_USING = !SCREEN_USING;
}
void wipe_oldscreen(unsigned char* buff){
    unsigned char *tmp = buff + screen0size * (V_SCREEN1 - SCREEN_USING);
    memset(tmp, 0, screen0size);
}
```

利用函数库已有函数和双屏动态多帧显示的原理，设计了一个实现显示方块移动动画的函数：

```
void draw_flash_block(unsigned char *buff, int xmin, int xmax, int ymin, int ymax,
color_8 color, int steps, int delay){
    //在当前屏幕画初始方块帧并延迟一段时间
    draw_vblock(buff, xmin, xmax, ymin, ymax, color, SCREEN_USING);
    usleep(delay);
    //防止擦除屏幕外的内容
    if (xmin == 0 && steps >= 1){
        //在不可见屏绘制下一帧方块，切屏显示下一帧方块后延迟一定时间再继续下一帧
        draw_vblock(buff, xmin + i, xmax + i, ymin, ymax, color, V_SCREEN1 -
        SCREEN_USING);
        switch_vscreen();
        usleep(delay);
        for(i = 2; i < steps; i++){
```

```
            //在不可见屏已有方块的位置绘制黑色方块，减少擦除屏幕的读写量以加快速度
            draw_vblock(buff, xmin + i − 2, xmax + i − 2, ymin, ymax, black,
             V_SCREEN1 − SCREEN_USING);
             ... //继续按以上规则绘制下一帧方块
        }
    }else ...
}
```

测试程序为 `fbblock`，其中分别了采用两种方式绘制移动方块动画。由于液晶屏本身硬件参数限制，考虑程序执行绘制的时间，每秒帧数和移动长度等问题后，结合多次实验测试，每帧的延迟取 8ms 时，效果较好，此时移动时间基本只和设定的帧延迟时间和帧数有关，说明绘图程序的效率合理。

执行测试程序，可以看出单屏和双屏绘制动画的效果对比：双屏绘制的动画由于切屏后直接显示绘制好的图形，流畅连贯；而单屏绘制的动画由于存在明显擦除的动作，表现为动态图形移动时闪烁的效果，视觉效果比较差，证明采用双屏法绘制动画是有效实用的一种绘制方法。

3.5 多 frambuffer 设备的探究

多 framebuffer 设备的探究改用基于 3.0.8 内核版本的 linux 系统完成。前述实验中的代码在新内核版本中工作正常，故不再另外说明。经多次测试，发现 fb1-fb4 设备的初始化方式实际上与 fb0 设备有所不同，采用原 fb0 设备的初始化方式会导致程序执行报错。[1]以下为多次试验后为可用的实例：

```
unsigned char *fb1_init(){
    fd1 = open ("/dev/fb1", O_RDWR);
    vinfo fbinfo;
    ioctl(fd1, FBIOGET_VSCREENINFO, &fbinfo);
    //关键设置部分，以下控制字设置缺一不可
    fbinfo.activate = FB_ACTIVATE_FORCE;
    ioctl(fd1, FBIOPUT_VSCREENINFO, &fbinfo);
    ioctl(fd1, FBIOBLANK, FB_BLANK_UNBLANK);
    return ((unsigned char *)mmap(NULL, screensize, PROT_READ |
    PROT_WRITE, MAP_SHARED, fd1, 0));
}
```

测试程序是一个简单的层叠渐变混色效果测试，调用了 3 个 frambuffer 设备来实现，下为部分代码：

```
/* frambuffer.h */
//逐行慢速画出方块
void draw_sblock(unsigned char *buff, int xmin,int xmax, int ymin, int ymax,
```

[1]将在4.1节中详细解释原理

```
 color_8 color);
/* fb2block.c */
    unsigned char *fbp = fb_init(), *fbp1 = fb1_init(), *fbp2 = fb2_init();
    //原程序的最后一部分，在紫色上刷上两层透明度递减的白色层可以使得颜色变淡
    make_color(&color, 255, 0, 255, 255);
    draw_sblock(fbp, 0, 800, 0, 480, color);
    make_color(&color, 255, 255, 255, 127);
    draw_sblock(fbp1, 0, 800, 0, 480, color);
    make_color(&color, 255, 255, 255, 63);
    draw_sblock(fbp2, 0, 800, 0, 480, color);
}
```

在开发板上执行`fb2block` 可以看到测试程序的最终效果与注释一致。[1]

4 实验讨论与小结

4.1 部分实验问题讨论

* `Frambuffer` 中出现的部分问题。

– 操作 frambuffer 全屏显示时左上角有一块黑色部分无法覆盖 –

该问题是由 fb0 设备被系统的某一进程的控制台输出占用引起的。通过多次实验发现，关闭 fb0 设备后现象与关闭 fb1、fb2 设备不同，fb0 设备原本显示的图形不会消失。实际上，仅当 fb0 设备的打开次数多于 1 次时才会出现 fb0 设备关闭但不释放的情况。另外，还在测试程序时发现，fb0 设备的 `fb_var_screeninfo` 在被不断刷新。综合判断可以得出 fb0 设备在系统启动后即被某一非测试程序的进程打开并使用的结论。

– 通过 `FBIOPUT_VSCREENINFO` 控制字无法正确切屏的原因 –

在双屏显示动画实验中发现，利用`FBIOPUT_VSCREENINFO` 控制字可以改变屏幕信息，但却不能正常完成切屏后显示的功能，改用`FBIOPAN_DISPLAY` 控制字则问题解决。经过分析代码[2]，观察两个控制字调用的函数：

```
int fb_set_var(struct fb_info *info, struct fb_var_screeninfo *var){
    if ((var->activate & FB_ACTIVATE_FORCE) ||
        memcmp(&info->var, var, sizeof(struct fb_var_screeninfo))) {
        ...
        if ((var->activate & FB_ACTIVATE_MASK) == FB_ACTIVATE_NOW) {
            info->var = *var;
            ret = info->fbops->fb_set_par(info);
            fb_pan_display(info, &info->var);
}
```

[1] 为节约篇幅起见只说明了测试实验最后一段的实验现象

[2] 参照2.3节

可以发现无论利用哪种控制字都正确调用了 `fb_pan_display()` 函数，一般来说不会出现无法改变屏幕偏移的情况。后又经过多次实验运行程序 `fbflash` 发现，调用 `fb_set_var()` 函数后，屏幕会清空。动画函数的操作是先绘制好隐藏帧图像再进行切屏，如果在切屏时清空了屏幕自然无法正确显示切屏结果！

再仔细阅读内核代码，才分析出了屏幕清空的原因：[1]

`fb_set_var()` 函数中调用 `fb_pan_display()` 函数的前一行，调用了底层驱动程序中的如下函数：

```
static int s3cfb_set_par(struct fb_info *fb){
...     /* 原注释表明非默认fb设备是未经过下列初始化过程的 */
    if (win->id != pdata->default_win) {
        fb->fix.line_length = fb->var.xres_virtual *
                              fb->var.bits_per_pixel / 8;
        fb->fix.smem_len = fb->fix.line_length * fb->var.yres_virtual;
        s3cfb_map_video_memory(fb);
    }
        /* 读取并改变多项屏幕参数，会自动清空屏幕 */
        s3cfb_set_win_params(fbdev, win->id);...
}
```

– fb1-fb4 设备的初始化方式与 fb0 设备不同的原因 –

以下代码在初始化 fb0 时不需要，但在初始化非 fb0 的设备时必须用到：

```
fbinfo.activate = FB_ACTIVATE_FORCE;
ioctl(fd1, FBIOPUT_VSCREENINFO, &fbinfo);
ioctl(fd1, FBIOBLANK, FB_BLANK_UNBLANK);
```

原因与第一个问题有共同之处，即实际上 fb0 已经被其他进程初始化过一次，不必调用`FBIOBLANK` 关键字来开启屏幕，故对没有被初始化过的其它设备而言，第三行代码必须存在。经过实验发现，如果不用前两行代码强制调用一次`FBIOPUT_VSCREENINFO` 关键字，第三行代码同样会出错，故而分析以下驱动源码：

```
static int s3cfb_blank(int blank_mode, struct fb_info *fb){
    switch (blank_mode) {
    case FB_BLANK_UNBLANK:
        /* 注意到以下这个判断条件在s3cfb_set_par()函数中有涉及 */
        if (fb->fix.smem_start) {
            s3cfb_win_map_off(fbdev, win->id);
            s3cfb_set_window(fbdev, win->id, 1);
        }
}
```

[1]注释是分析源码时自行添加

通过前述已分析的s3cfb_set_par() 函数会发现，非默认 fb 设备确实不满足该判断条件，只有通过在用户层面改动activate 状态，强制通过fb_set_var() 函数调用s3cfb_set_par() 函数才能使该函数正常执行（即满足使用FBIOBLANK 关键字开启屏幕的条件）。

* 探讨软件结构的层次关系。

实际上通过阅读其它成熟的调用 frambuffer 的工程代码发现，此次实验中采取的软件结构还是比较混乱，在此对软件结构的层次关系进行一些讨论。

一是库函数全部放在一个.c 文件内是不太好的一个选择。一方面，一个太长的代码源文件不适合模块化管理、分享、调试、更新等活动，按功能分割成多个的源文件会更合适。同时，对于一组提供出去的接口，实际上用户得到的是一个或多个.h 文件和一个动态库文件.so，根据前面介绍的动态库生成原理，也支持由多个源文件生成一个动态库文件。

二是接口如何设计才比较合理。实际上接口函数需要调用的参数往往也是接口库对用户暴露的信息。本次实验的程序接口函数调用的参数是由mmap() 函数得到的指针，这并不是一个好选择，首先是不通用，往往为了某几个库函数需要重新设计全局变量的配置；同时，通过用户程序暴露一个指向内核空间的指针也不安全。相比之下，使用 frambuffer 设备打开后得到的文件描述符作为在 linux 内核的配合下具有很强的通用性，只需要这个文件描述符，函数几乎可以对 frambuffer 进行任何操作而不暴露任何其他内部信息。对用户暴露文件描述符，在 linux 操作系统下本身也是安全的行为。

还有一种设计思路是完全封装，屏蔽所有细节。比如按这样的思路设计图形用户接口时，往往使用较多用户程序无法调用的静态全局变量，或通过不同的头文件版本隔离出用户无法调用的内部函数。用户不需要知道任何关于 frambuffer 的信息和相关的文件描述符、指针等，甚至可以不知道软件调用了哪一个设备文件，当然这种思路也可以通过设计专门传出 frambuffer 内部参数的函数来提供一定的用户可定制性。

另外，对于图形接口，绘制图形的函数无论采取何种思路，由于嵌入式设备的特殊性，往往需要一定优化和显示上的调整，一般采取不同图形采用不同函数的绘图思路而非设计通用函数来解决问题，这也是嵌入式开发相较于个人计算机开发一个比较独特的地方。

4.2 实验小结

这次实验的主题是图形用户接口实验。一方面我通过对 frambuffer 设备的操作实现了 frambuffer 在图形用户接口领域一些初步的应用，比如绘制线段、图片、双屏法绘制动画，多 frambuffer 设备的操作实现类似图层的效果等。在这次实验的中后期，在 frambuffer 的操作上遇到了一些不理解的问题，促使我阅读了不少 Linux 内核中相关的源码、驱动来搞懂这些问题，并在后来的几次实验中验证了我的想法。这些探究的过程本身的收获，并不比本身实现 frambuffer 操作的收获少。

另一方面，通过这次实验也认识到了自己代码能力的不足，首先软件的结构比较混乱，而且后期发现有一部分无用的设计或是效率很低、代码复用性低的设计，实现的效果也不好。比如绘制函数的编写，由于实现效果一般，软件结构上想采用引用 C 标准数学库的做法也并不可行，在实验的后期基本属于废弃不用的函数，故在这次报告中也没有提及。