数字系统II 实验 报告二

南京大学 电子科学与工程学院 吴康正 171180571
项目repo地址：https://git.nju.edu.cn/Minaduki/beaglebone_proj

实验目的

1. 使用Frame Buffer驱动HDMI显示器。
2. 编写画点、画线的简单API函数。
3. 尝试使用动态/静态链接库封装API。

实验环境

1. 硬件环境

实验使用了TI的BeagleBone Black开发板作为实验环境，其参数如下：

- 处理器：基于ARM Cortex-A8架构的TI AM3359 Sitara @ 1GHz
- 内存：板载512MiB DDR3
- 存储：板载4GiB 8-bit eMMC闪存
- 拓展存储：支持micro SD存储卡
- 网络界面：RJ-45接口百兆以太网
- 数字多媒体输出：micro HDMI接口
- 拓展接口：UART、GPIO、SPI、I2C等

使用的显示器：

- 分辨率：最大1920*1080
- 色彩空间：RGB565

2. 软件环境

- 上位机：使用Ubuntu 19.10系统的x86 PC
- Linux源码：版本4.4.155
- Busybox源码：版本1.30.1
- 编译器：The GNU Compiler Collection 9.2.1
- bootloader：U-boot

3. 网络环境

- 网关：192.168.208.254
- 上位机：192.168.208.35
- 开发板：192.168.208.121

实验原理

> 关于实验原理：
> 具体的实验原理中，实验指导书有所提及的在这里不再重复叙述。这里仅进行一些补充说明。

1. 关于screeninfo

1. fb_fix_screeninfo
   这个结构体在显卡模式设定后创建，用于描述显示卡的属性，如Frame Buffer在内存中的起始地址、占用大小，每行的长度、重映射的I/O的地址起止等。此结构体在系统正常运行时不可被修改。

```
struct fb_fix_screeninfo {
    char id[16];            /* identification string eg "TT Builtin" */
    unsigned long smem_start;/* Start of frame buffer mem */
    __u32 smem_len;         /* Length of frame buffer mem */
    __u32 type;             /* see FB_TYPE_*        */
    __u32 type_aux;         /* Interleave for interleaved Planes */
    __u32 visual;           /* see FB_VISUAL_*      */
    __u16 xpanstep;         /* zero if no hardware panning  */
    __u16 ypanstep;         /* zero if no hardware panning  */
    __u16 ywrapstep;        /* zero if no hardware ywrap    */
    __u32 line_length;      /* length of a line in bytes    */
    unsigned long mmio_start;/* Start of Memory Mapped I/O   */
    __u32 mmio_len;         /* Length of Memory Mapped I/O  */
    __u32 accel;            /* Indicate to driver which */
    __u16 reserved[3];      /* Reserved for future compatibility */
};
```

2. fb_var_screeninfo
   这个结构体中储存可供用户修改的变量，可以使用ioctl函数或者fbset命令进行全部/部分属性的更改。包含了屏幕的可见分辨率和虚拟分辨率、每个像素所占空间的大小、偏移量、灰度和显示模式等。

> 关于虚拟分辨率：
> 在实际使用中，常常通过设置两倍于真实分辨率的虚拟分辨率来达到预渲染的效果，以充分利用资源、防止画面卡顿，当然如果系统资源足够，也可以设置更高的虚拟分辨率。虚拟分辨率可以配合偏移量使用，在不进行重绘的情况下完成一些简单的动画效果。

```
struct fb_var_screeninfo {
    __u32 xres;         /* 行可见像素*/
    __u32 yres;         /* 列可见像素*/
    __u32 xres_virtual; /* 行虚拟像素*/
    __u32 yres_virtual; /* 列虚拟像素*/
    __u32 xoffset;      /* 水平偏移量*/
    __u32 yoffset;      /* 垂直偏移量*/
    __u32 bits_per_pixel;/*每个像素所占bit位数*/
    __u32 grayscale;    /* 灰色刻度*/
    struct fb_bitfield red; /* bitfield in fb mem if true color, */
    struct fb_bitfield green;   /* else only length is significant */
    struct fb_bitfield blue;
    struct fb_bitfield transp;  /* transparency         */
    __u32 nonstd;           /* != 0 Non standard pixel format */
```

```
    __u32 activate;         /* see FB_ACTIVATE_*        */
    __u32 height;           /* 图像高度*/
    __u32 width;            /* 图像宽度*/
    __u32 accel_flags;      /* (OBSOLETE) see fb_info.flags */
    __u32 pixclock;         /* pixel clock in ps (pico seconds) */
    __u32 left_margin;      /* time from sync to picture    */
    __u32 right_margin;     /* time from picture to sync    */
    __u32 upper_margin;     /* time from sync to picture    */
    __u32 lower_margin;
    __u32 hsync_len;        /* length of horizontal sync    */
    __u32 vsync_len;        /* length of vertical sync  */
    __u32 sync;         /* see FB_SYNC_*        */
    __u32 vmode;            /* see FB_VMODE_*       */
    __u32 rotate;           /* angle we rotate counter clockwise */
    __u32 reserved[5];      /* Reserved for future compatibility */
};
```

关于链接库

链接库主要是预编译的目标文件的集合，可以被编译器链接进程序，使用库文件主要是为了节约重复编译的时间成本和减少代码更新的工作复杂度。
理论来说，使用Makefile对目标文件进行管理也可以起到同样的效果，但是在不同系统之间迁移时，链接库显然更胜一筹。

1. 静态库
   静态库可以理解为目标文件的简单打包，由程序`ar`生成。
   静态库文件的内容会直接被打包到生成的二进制文件中，因此仅在编译环节需要，编译生成的二进制可以独立运行。

```
$ export AR=ar CC=gcc
$ AR -rcs <Library name> <Objective files>
$ CC -o main -lmylib <src>
```

2. 动态链接库
   动态链接库不需要打包在程序里，只会在程序里留下对应的接口，在运行时寻找对应的代码，因此体积更小，同时库文件升级时，如果函数接口没有变化，就不用重新编译。

```
$ export CC=gcc
$ CC -shared -fPIC -o libmylib.so <libsrc>
$ CC -o main -lmylib <src>
```

> 其中，第一句CC用于生成动态链接库，第二句CC则是使用动态链接库链接二进制程序。
> `-fPIC`参数代表生成位置无关代码(Position-Independent Code)。
> 如果没有使用`-l`的方式指定库文件位置，而是直接将其当作目标文件处理，则运行时库文件的位置必须与编译时一致。

实验流程

1. 准备工作

1.1 重新配置编译内核

使用`make menuconfig ARCH=arm`对内核进行自定义，需要保证如下条件满足：

1. `Device Drivers->Graphics support->DRM support for TI LCDC Display Controller`已被启用，这是开发板上的LCD控制器驱动，用于支持帧缓冲设备，启用后才能在设备目录下看见fb设备。
2. `Device Drivers->Graphics support->I2C encoder or helper chips->NXP semiconductors TDA668X HDMI encoder`已被启用，这是开发板上HDMI输出的驱动，用于将LCDC的输出编码为HDMI输出，启用后才能将Frame Buffer中的内容输出至HDMI显示器上。

配置完成后重新编译内核。

1.2 重新配置根文件系统

虚拟文件系统可以提供许多直接方便的硬件访问，因此重新配置启动脚本`/etc/rc`，在最后加上：

```
mount -t sysfs sys /sys
```

另外，实验中开发板的文件交换由NFS进行实现，因此在启动脚本中添加自动挂载NFS目录的命令：

```
mount -o nolock 192.168.208.35:/srv/nfs4 /mnt
```

由于系统使用NFS作为根文件系统，因此在启动时会自动配置网络，如果使用其他根文件系统启动，还应该在挂载命令之前添加：

```
ifconfig eth0 192.168.208.121
route add default gw 192.168.208.254
```

至此，所有准备工作完成，使用重新配置的内核和根文件系统启动。

2. 确定程序结构与封装

程序包含三部分源文件，分别为：

1. 通过syscall与底层LCDC帧缓冲驱动交互的程序`LCDC.c`和提供给用户的接口头文件`LCDC.h`。
2. 通过交互程序进行简单图形字符绘制的绘图程序`mydraw.c`、字体文件`FONT.H`和提供给用户的接口头文件`mydraw.h`。
3. 用户程序`main.c`。

三部分源文件互相独立，可以单独升级/替换任意一部分而不影响程序功能，比如，移植到其他设备时，只需要将`LCDC.c`替换，重新编译即可。

程序通过Makefile进行管理，将`LCDC.c`和`mydraw.c`文件编译为`libgraphics.so`的动态链接库，同时提供`LCDC.h`和`mydraw.h`用于应用开发。

文件	属于	依赖于	是否提供给用户
LCDC.h	系统交互	无	是
LCDC.c	系统交互	LCDC.h	否
FONT.H	绘图库	无	否
mydraw.h	绘图库	LCDC.h FONT.H	是
mydraw.c	绘图库	mydraw.h LCDC.o	否
libgraphics.so	二进制库文件	mydraw.o LCDC.o	是
main.c	用户程序	LCDC.h mydraw.h libgraphics.so	N/A

3. 编写程序

> 仅包含部分关键代码，完整源码请查看此处。

3.1 底层交互部分

1. `fb_open`函数

> 函数支持在支持的范围内指定分辨率，通过`fb_open(1, xres, yres)`的方式调用。

函数通过syscall对设备文件`/dev/fb0`进行操作。
使用`open`函数打开设备文件：

```
fd = open("/dev/fb0", O_RDWR);
```

使用`ioctl`函数，获取fb_var_screeninfo的信息，此信息包含屏幕分辨率和每个像素占存储的大小，用于下文mmap大小的确定。

```
if (ioctl(fd, FBIOGET_VSCREENINFO, &vinfo) < 0)
{
    perror("ioctl");
    return VINFO_READ_FAILED;
}
```

如果指定了分辨率，也使用`ioctl`将新的分辨率写回设备：

```

```

```
if (costumize)
{
    vinfo.xres = xres;
    vinfo.yres = yres;

    vinfo.xres_virtual = xres;
    vinfo.yres_virtual = yres;

    ioctl(fd, FBIOPUT_VSCREENINFO, &vinfo);
    ioctl(fd, FBIOGET_VSCREENINFO, &vinfo);
}
```

最后，根据获得的信息计算出需要映射的内存大小，使用mmap函数完成映射。

```
screen_size = vinfo.xres_virtual * vinfo.yres_virtual *
vinfo.bits_per_pixel / 8;
fbp = (unsigned char*)mmap(0, screen_size, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
```

2. 画点函数

程序提供了两种画点函数可供调用，可以分别指定RGB的8位数值，然后转换成RGB565的格式，或者直接给定一个16位的RGB565颜色，进行绘制：

```
int fb_draw_back_888(int x, int y, char Red, char Green, char Blue)
{
    long offset;
    short color;
    offset = (y * vinfo.xres + x) * vinfo.bits_per_pixel / 8;
    color = ((Red << 8) & 0xF800) | ((Green << 3) & 0x07E0) | ((Blue >> 3)
& 0x1F);
    *(unsigned char*)(fbp + offset + 0) = color & 0xFF;
    *(unsigned char*)(fbp + offset + 1) = (color >> 8) & 0xFF;
    return EXIT_SUCCESS;
}

int fb_draw_back_565(int x, int y, u_int16_t color)
{
    long offset;
    offset = (y * vinfo.xres + x) * vinfo.bits_per_pixel / 8;
    *(unsigned char*)(fbp + offset + 0) = color & 0xFF;
    *(unsigned char*)(fbp + offset + 1) = (color >> 8) & 0xFF;
    return EXIT_SUCCESS;
}
```

> fbp指针的偏移量与其类型有关，如果变量为uint16_t类型，就需要更改除8为除16。

> 程序中使用以下全局变量：

```
static int fd;
struct fb_var_screeninfo vinfo;
static char* fbp;
unsigned long screen_size;
```

> 这些变量封装于LCDC程序内，仅对此程序可见，除`fb_var_screeninfo`外，不建议于绘图库和用户程序内访问。

3.2 绘图库部分

绘图库提供以下绘图函数：

```
//m^n函数
//返回值:m^n次方.
long Pow(int m, int n);

void Clear(u_int16_t color);
void DrawPoint(int x, int y, u_int16_t color);
void DrawLine(int x1, int y1, int x2, int y2, u_int16_t color);
void DrawRectangle(int x1, int y1, int x2, int y2, u_int16_t color);
void DrawCircle(int x0, int y0, int r, u_int16_t color);

//在指定位置显示一个字符
//x,y:起始坐标
//num:要显示的字符:" "--->"~"
//size:字体大小 12/16/24/32
void ShowChar(int x, int y, char content, int size, u_int16_t color);
//显示数字,高位为0,则不显示
//x,y :起点坐标
//len :数字的位数
//size:字体大小
//color:颜色
//num:数值(0~4294967295);
void ShowNum(int x, int y, long num, u_int8_t len, int size, u_int16_t
color);
void ShowFloat(int x, int y, double num, u_int8_t precision, u_int8_t len,
u_int8_t size, u_int16_t color);
//显示字符串
//x,y:起点坐标
//width,height:区域大小
//size:字体大小
//*p:字符串起始地址
void ShowString(int x, int y, int width, int height, u_int8_t size, char
*p, u_int16_t color);
```

并提供部分颜色的16位RGB565值的宏定义，可以直接使用。
所有的绘图函数同时提供函数名以RGB为结尾的函数，功能相同，但是使用三个8位变量指定三种颜色比例，

如：

```
void DrawPoint_RGB(int x, int y, char red, char green, char blue);
```

> 画圆函数使用Bresenham算法实现：

```
void DrawCircle(int x0, int y0, int r, u_int16_t color)
{
        int a, b;
        int di;
        a = 0;
    b = r;
        di = 3 - (r << 1);              //判断下个点位置的标志
        while (a <= b)
        {
                DrawPoint(x0 + a, y0 - b, color);             //5
                DrawPoint(x0 + b, y0 - a, color);             //0
                DrawPoint(x0 + b, y0 + a, color);             //4
                DrawPoint(x0 + a, y0 + b, color);             //6
                DrawPoint(x0 - a, y0 + b, color);             //1
                DrawPoint(x0 - b, y0 + a, color);
                DrawPoint(x0 - a, y0 - b, color);             //2
                DrawPoint(x0 - b, y0 - a, color);             //7
                ++a;
                //使用Bresenham算法画圆
                if (di < 0)
        {
            di += 4 * a + 6;
        }
                else
                {
                        di += 10 + 4 * (a - b);
                        --b;
                }
        }
}
```

> 字符显示函数通过读取预设在FONT.H内的矩阵对应项目的值，获取点阵信息：

```
void ShowChar(int x, int y, char content, int size, u_int16_t color)
{
    unsigned char temp, t1, t;
        int y0 = y;
        u_int8_t csize = (size / 8 + ((size % 8) ? 1 : 0)) * (size / 2);//
得到字体一个字符对应点阵集所占的字节数
        content = content - ' ';//得到偏移后的值（ASCII字库是从空格开始取模，所以-
' '就是对应字符的字库）
        for (t = 0; t < csize; ++t)
```

```
        {
                if (size == 12)
        {
            temp = asc2_1206[content][t];                //调用1206字体
        }
        else if ...
                for (t1 = 0; t1 < 8; ++t1)
                {
                        if (temp & 0x80)
            {
                DrawPoint(x, y, color);
            }
                        temp <<= 1;
                        ++y;
                        if (y >= vinfo.yres)
            {
                return;         //超区域了
            }
                        if ((y - y0) == size)
                        {
                                y = y0;
                                ++x;
                                if (x >= vinfo.xres)
                {
                    return;     //超区域了
                }
                                break;
                        }
                }
        }
}
```

3.3 用户程序

用户程序中应当包含`LCDC.h`和`mydraw.h`两个头文件。
用户应先调用`fb_open()`函数打开Frame Buffer映射并设置分辨率(可选)，然后，用户可以调用绘图库的函数进行绘制，最后调用`fb_close()`函数关闭设备文件。

4. 创建动态链接库

使用Makefile对编译过程进行管理：

```
CC = arm-linux-gnueabihf-gcc
CFLAGS = -Wall -g
CPPFLAGS = -fPIC
LFLAGS = -L. -lgraphics
target = main
object = LCDC.o mydraw.o
mainobj = main.o
lib = libgraphics.so
```

```
$(target): $(object) $(mainobj)
        $(CC) $(CFLAGS) $(CPPFLAGS) -o $@ $^

install: main.c $(lib)
        $(CC) $(CFLAGS) -o $(target) $< $(LFLAGS)

$(lib): $(object)
        $(CC) $(CFLAGS) $(CPPFLAGS) -shared -o $@ $^

%.o: %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $<

clean:
        rm -rf $(target) *.o *.so
```

其中，各个目标的作用如下：

- $(target): 用于生成单文件的测试程序，可独立运行
- install: 用于生成依赖于动态链接库的用户程序，需要配合动态链接库才能运行。
- $(lib): 用于打包LCDC和mydraw的目标文件生成动态链接库
- %.o: 用于编译生成目标文件
- clean: 用于清理

将生成的库文件复制到BeagleBone的`/lib`目录下，执行生成的用户程序，即可看见绘图效果。

5. 绘图效果

由于虚拟文件系统的作用，可以很方便地通过`/dev/fb0`文件存取Frame Buffer中的图像。
使用`cat`或`cp`命令即可将`/dev/fb0`中的内容存储到一个普通文件中：

```
# cat /dev/fb0 > /mnt/demo.raw
```

> 这里的raw后缀名只是用于区分，并不是真正的相机raw文件。

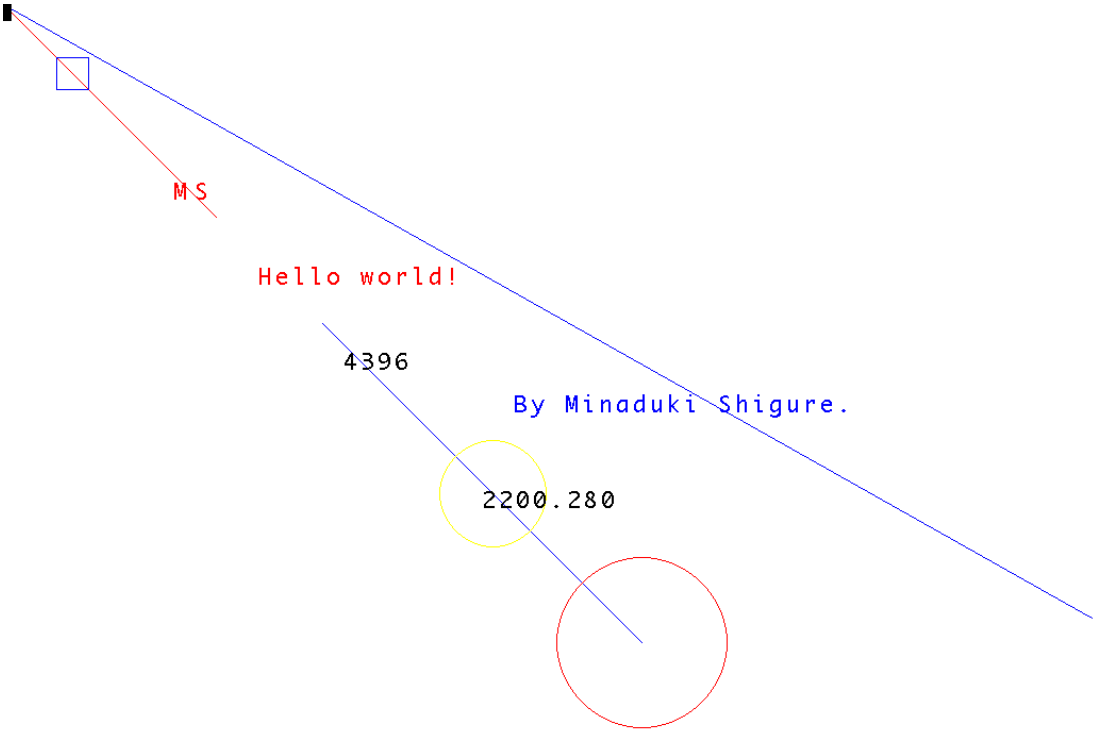
同理，如果普通文件存储了一幅Frame Buffer截图，也可以使用同样的方法显示出来：

```
# cat /mnt/demo.raw > /dev/fb0
```

获取图像后，可以在上位机通过`ffmpeg`程序转码为需要的图像封装格式：

```
$ ffmpeg -vcodec rawvideo -f rawvideo -pix_fmt rgb565 -s 1280X1024 -i
demo.raw -f image2 -vcodec png out-%d.png
```

运行本项目提供的用户程序demo，屏幕显示的效果如下：



> 当长时间没有被使用时，LCDC会进入休眠状态，可以通过向Frame Buffer的空白控制字段送一个数据而唤醒，只需输入以下命令：

```
# echo 0 > /sys/class/graphics/fb0/blank
```

小结

关于分辨率的问题

根据Frame Buffer的理论结构，修改fb_var_screeninfo中的物理和虚拟分辨率大小就可以更改输出显示的分辨率，但是实际操作中发现，如果在系统启动时连接了显示器，则分辨率最高只能修改到1280*1024，而如果在系统启动时没有连接显示器，则分辨率最高只能修改到1024*768。
如果使用fbset修改分辨率到更高值，程序会正常返回，但是分辨率并不会发生改变，如果使用ioctl函数强行修改了分辨率并尝试进行显示的话，则会报段错误而强制退出。
因此我认为，分辨率不能修改可能有以下两个原因：

1. 可能是由于开发板搭载的LCDC本身无法支持更高的分辨率，因此对可供Frame Buffer映射的I/O长度有限制。在I/O重映射时如果强行扩大映射范围，会访问到不属于自己的I/O，导致程序出错。但是这无法解释在开机时连接与不连接显示器时最大分辨率有差别的现象。

2. 可能是由于内核会在启动时划分内核内存空间的大小，而由于系统本身资源有限，内核更倾向于减少内核内存空间浪费以确保应用能够顺利运行，因此猜想在Frame Buffer在划分时就严格按照其长度分配内存空间，所占用的内存后方的连续空间也被其他需求占用了，因此无法扩大Frame Buffer的大小，也不能向后越界访问。这可以解释在不同的情况下启动时的分辨率差异问题，因为理论来说，系统内核应该可以控制LCDC识别显示设备的规格，也可以部分解释fbset无功而返的原因，但是依然缺乏足够的说服力。

关于封装

在编写绘图库时，使用了来自`LCDC.c`程序中的一个外部变量，用于确定绘图边界，这样的做法对封装有一定损伤。
一个想到的替代方案时，修改底层交互程序的`fb_open()`函数，新增加一个参数用于回传一个结构体，包含绘图需要的参数，在绘图库中设置一个初始化函数，初始化这样一个结构体，并且调用`fb_open()`函数配置Frame Buffer并获得需要的参数。
最后，将绘图库中的这个初始化函数提供给用户程序实现初始化，用以取代在用户空间访问`fb_open()`函数实现的功能，这样用户程序只需包含`mydraw.h`头文件，底层操作的两个文件对于用户完全透明。