華中科技大學

2024

嵌入式系统 课程实验报告

题　　目：	香橙派相关实验
专　　业：	计算机科学与技术
班　　级：	大数据 2201
学　　号：	U202215566
姓　　名：	刘师言
电　　话：	18571580627
邮　　件：	shyl@hust.edu.cn

目录

1 实验一 系统烧录

1.1 实验要求

实验一的主要要求包括如下几点：

1. **准备**
   a) 开发板连接
   b) 触摸屏连接
2. **香橙派镜像的烧录**
   a) 开发板镜像烧录
   b) 开发板启动
3. **香橙派的使用**
   a) 开发板网络配置
   b) SSH 服务启动
4. **Linux 应用程序开发**
   a) 在开发板上编译简单应用程序

1.2 实验过程

1.2.1 开发板的准备

1. 开发板连接

按照实验指导书中的连接方式，依次对开发板的各接口进行连接，如图 1. 1 所示：

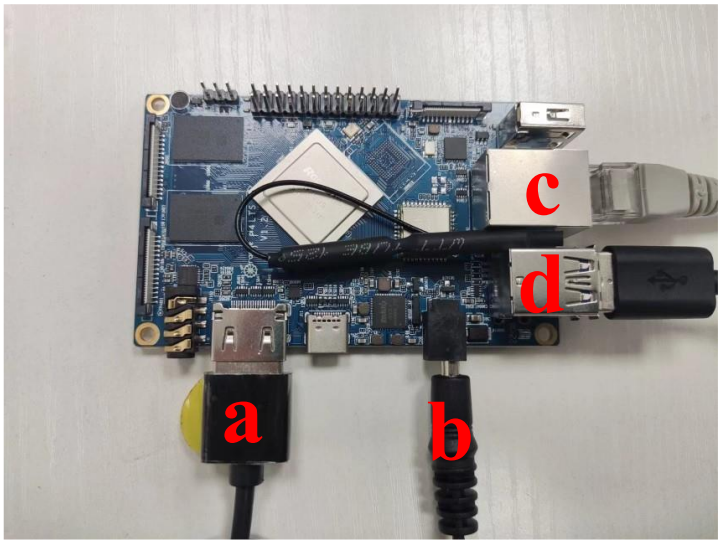

图 1. 1 开发板的连接

其中，各符号表示的接口和所连接的对象分别如下所示：

a) HDMI 接口，连接显示屏的 HDMI 接口；

b) 电源接口，DC 接口，连接电源；

c) 网线接口，连接开发主机的以太网口（本实验使用无线通信，故未用到）；

d) USB 接口，连接显示屏的 touch 接口。

为了方便开发板的操作和实验的调试，还额外使用了开发板的两个 USB 接口来分别连接鼠标和键盘。

2. 触摸屏连接

同样地，按照实验指导书中的连接方式，依次对触摸屏的各接口进行连接，如图 1.2 所示：

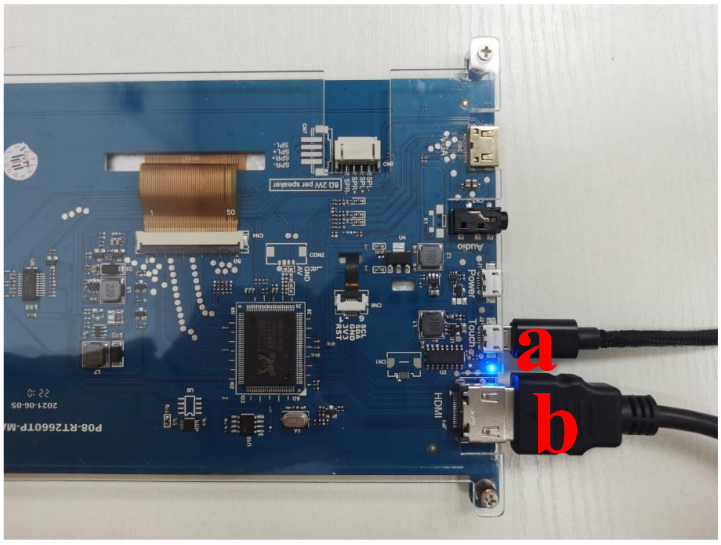

图 1.2 触摸屏的连接

其中，各符号表示的接口和所连接的对象分别如下所示：

a) Micro USB 接口，用于给显示屏供电以及发送触摸信号；

b) HDMI 接口，接收音频及视频信号。

1.2.2 香橙派镜像的烧录

1. 开发板镜像烧录

首先，下载镜像压缩文件至开发主机。在下载目录打开终端，输入如下命令来解压文件：

```
1. unzip Orangepi4-lts_3.0.6_ubuntu_jammy_desktop_xfce_linux5.18.5.zip
```

解压完成后，得到镜像文件 Orangepi4-lts_3.0.6_ubuntu_jammy_desktop_xfce_linux5.18.5.img。接着，将开发板的 SD 卡插入读卡器，并连接开发主机。鼠标右键准备好的烧录软件 balenaEtcher，选择以管理员身份运行，如**错误!未找到引用源。**所示。最后，选择相应的镜像和 SD 卡进行烧录，烧录完成的结果如**错误!未找到引用源。**所示。

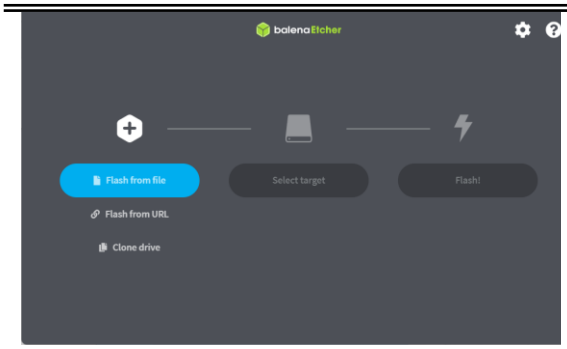

图 1.3 烧录软件 balenaEtcher

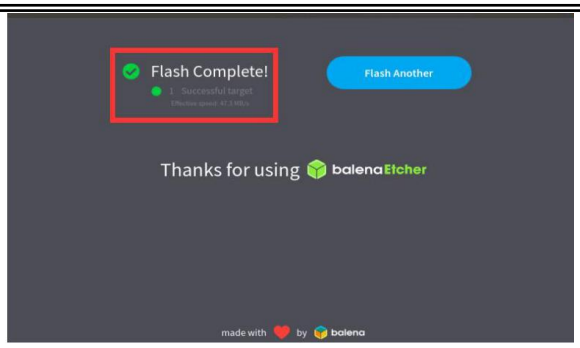

图 1.4 烧录完成的结果

2. 开发板启动

开发板镜像烧录完成后，在开发板断电的情况下插入 SD 卡，然后连接电源启动开发板，发现触摸屏上能正常显示开发板的启动界面，如图 1. 5 所示：

图 1.5 开发板启动

稍作等待后，使用开发板已有的账号密码（Username 和 Password 分别为 root 和 orangepi）进行登录，进入到系统桌面，如图 1. 6 所示：

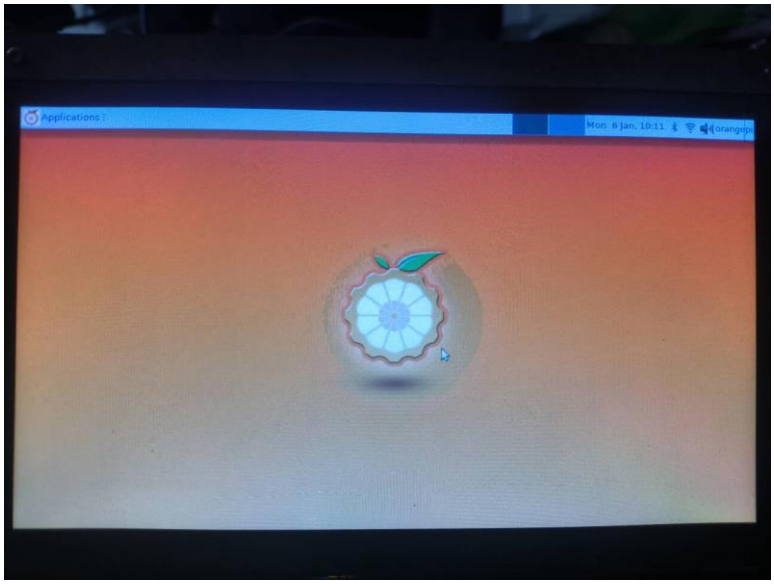

图 1.6 开发板进入到系统桌面

1.2.3 香橙派的使用

1. 开发板网络配置

本实验使用无线通信方式，通过手机热点进行开发板的远程登录。首先，依次点击触摸屏左上角的 Applications 和 Terminal Emulator，打开开发板的终端。接着，使用如下命令扫描周围的 WIFI 热点：

```
1. nmcli dev wifi
```

然后，使用如下命令连接扫描到的 WIFI 热点，其中 wifi_name 为手机热点的名称，wifi_passwd 为手机热点的密码：

```
1. nmcli dev wifi connect wifi_name password wifi_passwd
```

手机热点连接成功后，终端输出如图 1. 7 所示：

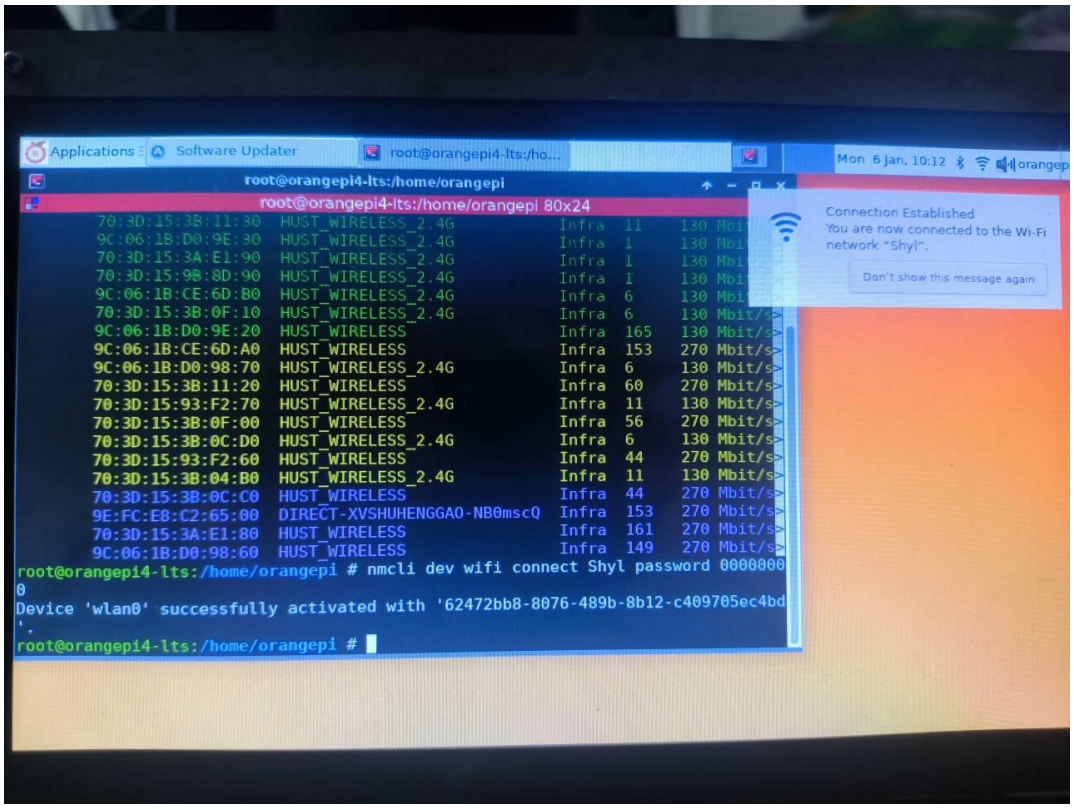

图 1. 7 开发板网络配置

2. SSH 服务启动

开发板网络配置完成后，通过 SSH 服务将开发主机连接到开发板，从而实现开发板的远程登录。同样地，开发主机也需连接到手机热点。接着获取开发板的 IP，使用如下命令可以查看开发板所连接 WIFI 的 IP 地址：

```
1. ip addr show wlan0
```

得到开发板的 IP 地址为 192.168.255.7，如图 1. 8 所示：

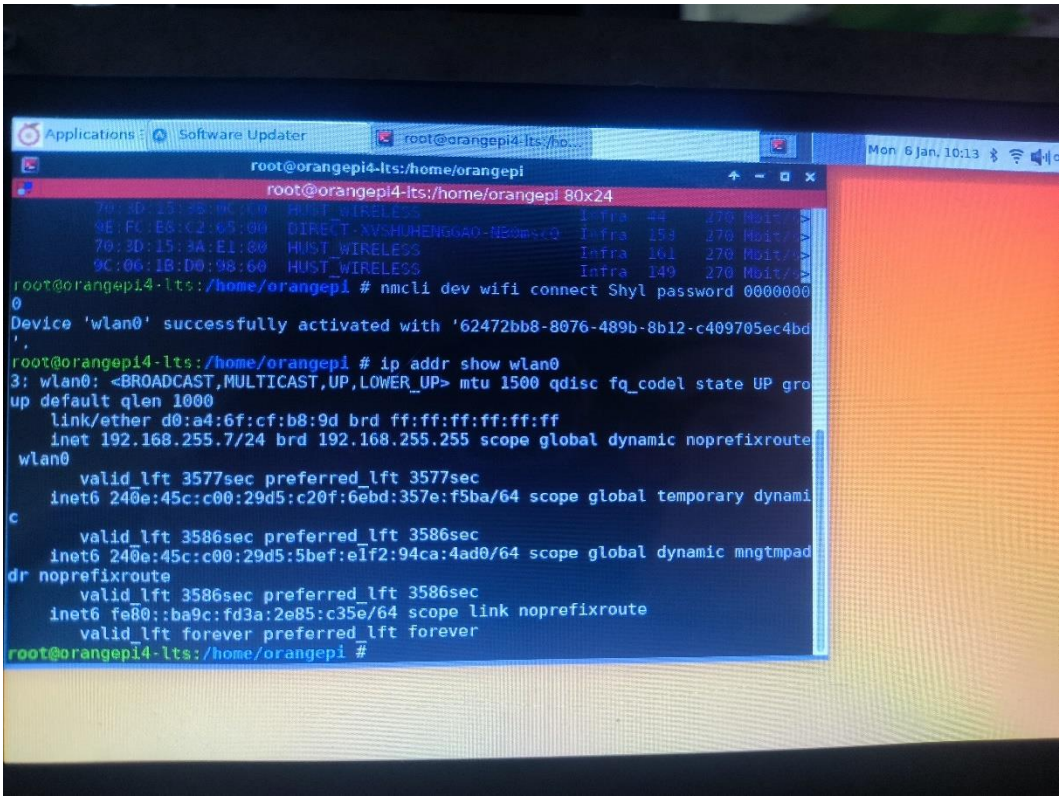

图 1.8 查看开发板的 IP 地址

开发主机使用 VS Code 进行开发，通过如下的 SSH 命令连接到开发板：

```
1. ssh root@192.168.255.7
```

连接过程输入开发板的登录密码，即可完成连接。连接成功后，开发主机在 VS Code 中可以读写开发板的指定目录，如图 1.9 所示：

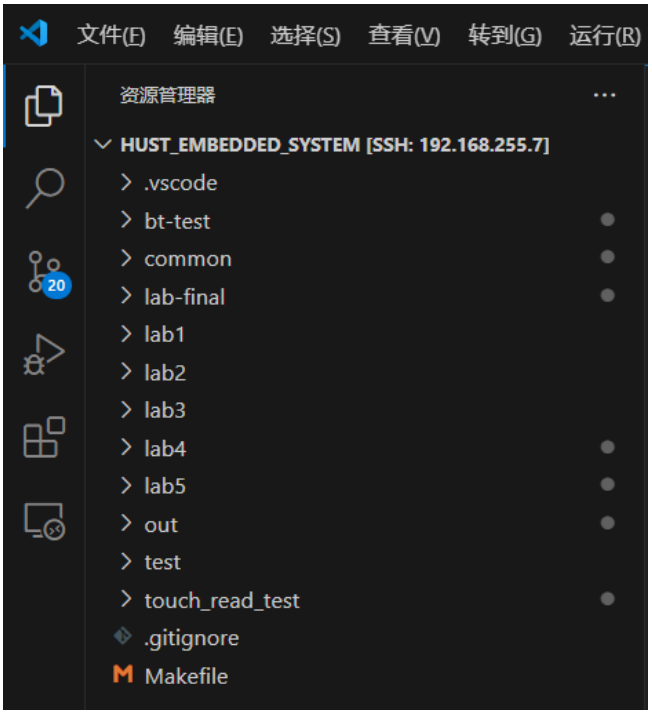

图 1.9 开发主机读写开发板的指定目录

1.2.4 Linux 应用程序开发

1. 在开发板上编译简单应用程序

首先，使用 SCP 将实验源代码 lab-2022-st.tgz 上传到开发板的 root 目录下，也可以通过 VS Code 完成便捷的拖拽上传。由于上一步中开发主机已通过 SSH 远程登录到开发板，因此开发主机可在本地执行开发板的终端命令。

实验源代码上传完成后，进入 root 目录，使用如下命令解压源代码：

```
1. tar -xzvf ./lab-2022-st.tgz
```

本实验使用开发板的本地编译环境对源代码进行编译，编译前需确保实验源代码目录中的编译器配置正确。在文件./lab-2022-st/common/rules.mk 中，删去 CC 的赋值中的 CROSS_COMPILE（交叉编译环境）字段，从而使用开发板的本地编译环境。

接下来通过键盘操控开发板，进入./lab-2022-st/lab1 文件夹，使用 make 命令进行编译，编译生成的 lab1 可执行文件将自动拷贝到../out 目录下。最后，直接进入../out 目录，使用如下命令即可运行编译好的 lab1 可执行文件。

```
1. ./lab1
```

由于实验一不涉及图形化界面的输出，因此上述运行可执行文件的命令也可以在开发主机端操控开发板的终端完成。

1.3 实验结果

运行编译好的 lab1 可执行文件后，观察到触摸屏上成功输出了“Hello embedded linux!”内容，如图 1. 10 所示：

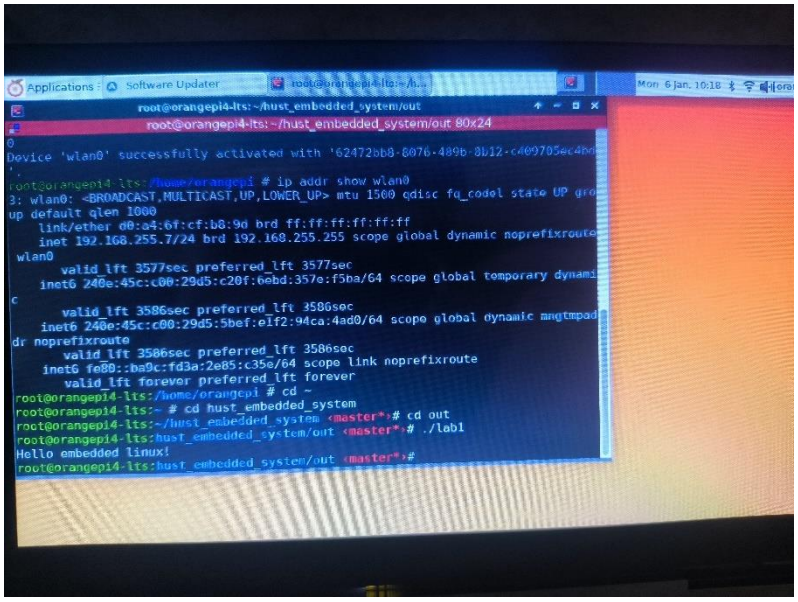

图 1. 10 实验一运行结果

实验结果表明上述开发板的准备、开发板镜像的烧录、开发板的使用和 Linux 应用程序开发等实验过程均正确无误，实验一顺利完成。

2 实验二 图形界面基础

2.1 实验要求

实验一的主要要求包括如下几点：

1. **Linux 下的 LCD 显示驱动接口**
   a) framebuffer 的使用原理
2. **基本图形的显示**
   a) 点区域
   b) 线区域
   c) 矩形区域
3. **双缓冲机制**

2.2 实验过程

2.2.1 Linux 下的 LCDC 显示驱动接口

1. **framebuffer 的使用原理**

Linux 中 framebuffer 是位于内存空间的一个二维数组，共有 600 行（屏幕宽度）和 1024 列（屏幕高度），数组中的每个元素表示屏幕上的一个点，占用 4 个字节的内存空间，地址从高到低每 8 位分别表示透明度（A）、红色像素值（R）、绿色像素值（G）和蓝色像素值（B）。framebuffer 的驱动原理如图 2. 1 所示：

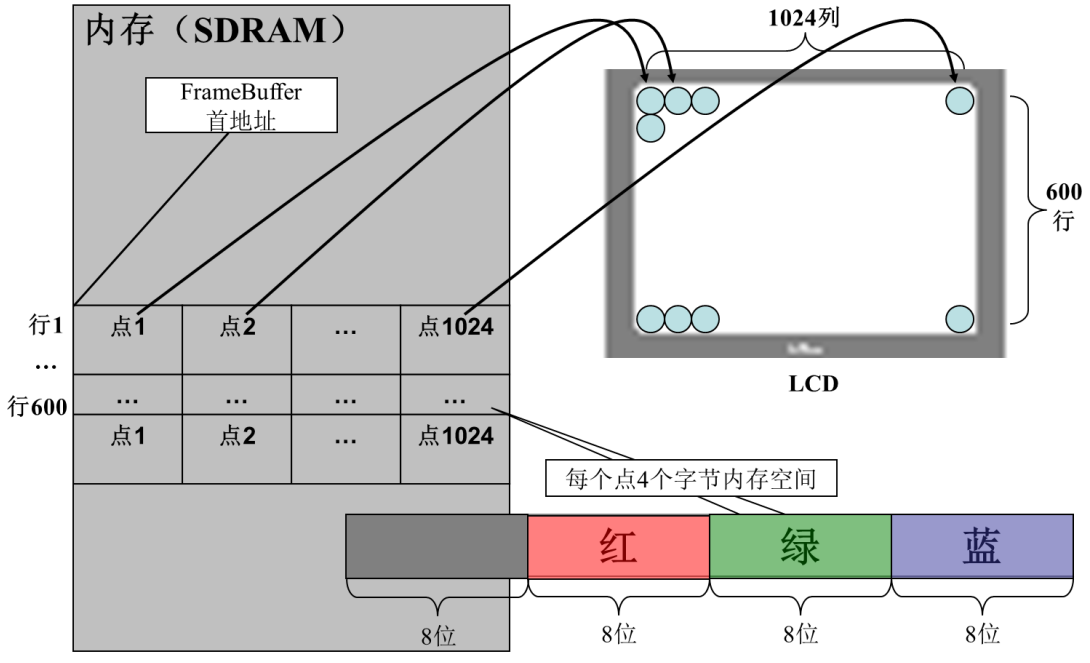

图 2. 1 framebuffer 驱动原理

本实验在 common/graphic.c 文件中提供了用于 framebuffer 设备操作的 LCD 显示驱动接口，其中 framebuffer 初始化的实现如下所示：

```
1. void fb_init(char *dev)
2. {
3.     fb_fd = open(dev, O_RDWR);
4.     if(fb_fd < 0){
5.             printf("Open fb device error!\n");
6.             return;
7.     }
8.     LCD_MEM_BASE = mmap(NULL,
9.                     SCREEN_WIDTH*SCREEN_HEIGHT*4,
                        PROT_READ|PROT_WRITE,
10.                     MAP_SHARED, fb_fd, 0);
11.    memset(LCD_MEM_BASE, 0, SCREEN_WIDTH*SCREEN_HEIGHT*4);
12. }
```

2.2.2 双缓冲机制

双缓冲机制确保界面内容一次性全部出现在屏幕上，在“基本图像的显示”一节用到，因此先于“基本图形的显示”一节介绍。

一般情况下，最终的用户界面都需要经过若干次绘图才能完成，比如要先擦除之前的部分界面内容，再逐步绘制新的界面内容。如果这些中间绘图是直接在 framebuffer 上操作的，那么在 LCD 屏幕上就会看到这些中间结果，比如看到屏幕先被清除，再逐渐显示部分界面，而不是界面内容一次性全部出现在屏幕上。

解决这个问题的办法就是双缓冲，所有的绘图都先绘制在一个后缓冲（和 framebuffer 同样大小的一块内存）中，绘制完毕后再把最终屏幕内容拷贝到 framebuffer 中。

双缓冲机制的绘图过程具体包括如下步骤：

1. 所有的绘图函数都在后缓冲中绘图：

```
1. void fb_draw_xxxx(…)
```

2. 所有的绘图函数中都要记录本次的绘图区域：

```
1. void _update_area(int x, int y, int w, int h)
```

3. 绘图完毕后，把后缓冲中所有需要更新的绘图区域内容拷贝到前缓冲（framebuffer），同时清空全局更新的绘图区域：

```
1. void fb_update(void);
```

使用流程图对双缓冲机制的绘图过程进行直观呈现，如图 2. 2 所示：

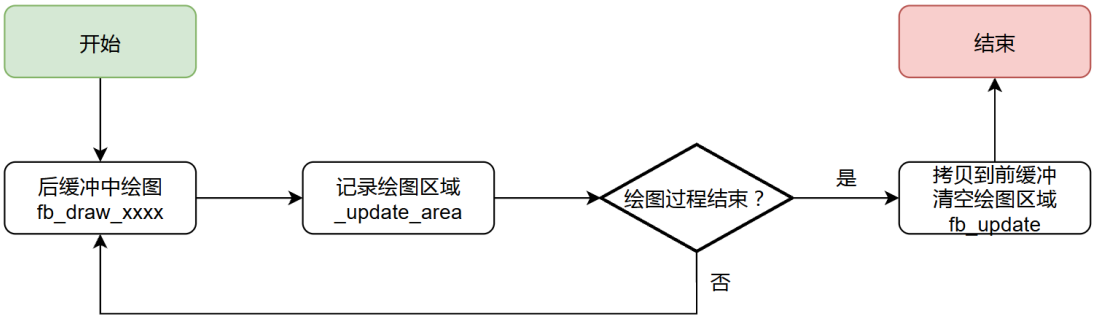

图 2.2 双缓冲机制

本实验的 lab2/main.c 文件中已经实现了双缓冲刷新的过程，如下所示：

```
#include “../common/common.h”
int main(int argc, char *argv[])
{
    fb_init("/dev/fb0");
    fb_draw_rect(0, 0, SCREEN_WDITH, SCREEN_HEIGHT, BLACK);
    fb_update();
    //......
    fb_draw_line(100,10,1000,580,FB_COLOR(0,0,100));
    fb_update( ); /*双缓冲刷新*/
    //fb_draw_xxxx;
    //fb_update();
    return 0;
}
```

2.2.3 基本图形的显示

1. 点区域

点区域的显示通过画点函数完成。本实验的 common/graphic.c 文件中定义了所有画图函数，包括画点函数、画矩形函数、画线函数等。

由上一节双缓冲机制的原理可知，该过程主要分为后缓冲中绘图、记录绘图区域和更新绘图区域（双缓冲刷新）三个阶段，而双缓冲刷新过程在画图函数调用时（lab2/main.c 文件）完成，因此画图函数的具体实现只需包括后缓冲中绘图和记录绘图区域两个过程。

本实验的 common/grahic.c 文件中已经实现了使用双缓冲机制的画点函数，如下所示：

```
void fb_draw_pixel(int x, int y, int color)
{
    if(x<0 || y<0 || x>=SCREEN_WIDTH || y>=SCREEN_HEIGHT) return;
    int *buf = _begin_draw(x,y,1,1);
/*-------------------------------------------------*/
    *(buf + y * SCREEN_WIDTH + x) = color;
```

```
7. /*-------------------------------------------------*/
8.     return;
9. }
```

其中，_begin_draw 函数对记录绘图区域的过程进行了封装，最终返回 DRAW_BUF 指针表示后缓冲区原点的地址。对于要绘制的点的坐标(x, y)，令 buf 为后缓冲区原点的指针，SCREEN_WIDTH 为屏幕宽度，color 为 4 个字节依次表示 ARGB 的像素值，则该点的指针为 buf + y * SCREEN_WIDTH + x，将该地址的内容设为 color，则完成了后缓冲中绘图的过程。

此外，为确保函数的健壮性，在绘图前需先判断要绘制的点是否正确位于屏幕范围内，否则直接返回。如果忽视了对绘图区域范围的检查，可能导致后续绘图过程产生预料之外的结果或导致程序异常。

最后，在画点函数调用后执行双缓冲刷新过程，即可实现点区域的显示。

2. 矩形区域

矩形区域的显示通过画矩形函数实现。画矩形函数可视作由若干个画点函数组成，但直接调用画点函数会带来大量函数调用（包括画点函数、_begin_draw 函数的调用等）的跳转开销，导致程序性能下降，因此考虑在画矩形函数内对 _begin_draw 记录的绘图区域进行扩展，并通过指针的顺序寻址来批量修改后缓冲的像素值。

同样地，在绘图前需先判断要绘制的矩形是否正确位于屏幕范围内，否则调整矩形区域，使其正确位于屏幕范围内，或者直接返回。

_begin_draw 函数接收四个参数的输入，分别为开始绘图的横、纵坐标和绘图区域的宽、高，这样函数内部就会根据现有绘图区域和传入的绘图区域来更新绘图区域。在画点函数中，传入的宽、高均为 1，而画矩形函数只需一次性将传入的宽、高分别设为矩形的宽、高即可完成记录绘图区域的过程。

接着，对于矩形的起始点（矩形的左上角顶点）(x, y)、宽 w 和高 h，令 buf 为后缓冲区原点的指针，SCREEN_WIDTH 为屏幕宽度，color 为 4 个字节依次表示 ARGB 的像素值，则矩形中(x+j, y+i)位置上的点的指针为 buf + (y + i) * SCREEN_WIDTH + (x + j)，将该地址的内容设为 color，则完成了后缓冲中绘制矩形中一点的过程。使用双重循环对 i, j 的值依次遍历，满足 i 小于 h 且 j 小于 w，在后缓冲绘制矩形中的所有点，则完成了后缓冲中绘制矩形的过程。

画矩形函数的具体实现如下所示：

```
1. void fb_draw_rect(int x, int y, int w, int h, int color)
2. {
3.     if(x < 0) { w += x; x = 0;}
4.     if(x+w > SCREEN_WIDTH) { w = SCREEN_WIDTH-x;}
5.     if(y < 0) { h += y; y = 0;}
6.     if(y+h >SCREEN_HEIGHT) { h = SCREEN_HEIGHT-y;}
```

```
    if(w<=0 || h<=0) return;
    int *buf = _begin_draw(x,y,w,h);
/*-------------------------------------------------*/
    for (int i = 0; i < h; i++) {
        for (int j = 0; j < w; j++) {
            *(buf + (y + i) * SCREEN_WIDTH + (x + j)) = color;
        }
    }
/*-------------------------------------------------*/
    return;
}
```

然而，双重循环的绘制过程效率较低，为提高程序性能，使用 SIMD 指令（如 memcpy 函数）替代双重循环来对绘图过程进行优化，从而使时间复杂度由$O(hw)$降低为$O(h)$，其中 h 和 w 分别表示矩形的高和宽。

优化后，画矩形函数的具体实现如下所示：

```
for (int i = 0; i < h; i++) {
    memcpy(buf + (y + i) * SCREEN_WIDTH + x, &color, w * sizeof(color));
}
```

最后，在画矩形函数调用后执行双缓冲刷新过程，即可实现矩形区域的显示。

3. 线区域

线区域的显示通过画线段函数实现。画线段函数给定线段两个端点的坐标，在屏幕上绘制一条尽可能流畅的线段。

同样地，在绘图前需先判断要绘制的线段是否正确位于屏幕范围内，即检查两个端点是否均正确位于屏幕范围内，否则直接返回。

画线段函数中，绘图区域等同于分别以两个端点为对角顶点的矩形区域，因此_begin_draw 函数的使用与画矩形函数基本相同。然而，画线段函数的实现相比画点函数和画矩形函数更为复杂，因为线段上的点需要通过一定的手段计算得到，这就用到计算机图形学的相关知识。

首先，计算绘制线段的步数 n，每一步绘制线段上的一个点，步数通过取横、纵坐标变化量的较大者得到。接着，计算横、纵坐标每步的增量，通过变化量与步数的比值得到。最后，从任意一个顶点开始，记录当前点的浮点数坐标表示，然后循环 n 次，每次在后缓冲中绘制当前点的整数坐标表示（通过 int 类型转换进行向下取整），并使当前点的浮点数坐标表示的横、纵坐标分别增加各自的增量。通过这种方法，可以近似地完成绘制线段的过程。

画线段函数的具体实现如下所示：

```
void fb_draw_line(int x1, int y1, int x2, int y2, int color)
{
/*-------------------------------------------------*/
```

```
4.      if(x1<0 || y1<0 || x1>=SCREEN_WIDTH || y1>=SCREEN_HEIGHT) return;
5.      if(x2<0 || y2<0 || x2>=SCREEN_WIDTH || y2>=SCREEN_HEIGHT) return;
6.      int x_min = x1 < x2 ? x1 : x2;
7.      int y_min = y1 < y2 ? y1 : y2;
8.      int x_max = x1 > x2 ? x1 : x2;
9.      int y_max = y1 > y2 ? y1 : y2;
10.     int *buf = _begin_draw(x_min, y_min, x_max - x_min, y_max - y_min);
11.
12.     // x、y 的总变化量
13.     float dx = x2 - x1;
14.     float dy = y2 - y1;
15.     // 计算步数，取变化量较大者
16.     float steps = abs(dx) > abs(dy) ? abs(dx) : abs(dy);
17.     // x、y 每步的增量
18.     float x_inc = dx / steps;
19.     float y_inc = dy / steps;
20.
21.     float x = x1;
22.     float y = y1;
23.
24.     for(int i = 0; i <= steps; i++) {
25.         *(buf + (int)y * SCREEN_WIDTH + (int)x) = color;
26.         x += x_inc;
27.         y += y_inc;
28.     }
29. /*-------------------------------------------------*/
30.     return;
31. }
```

2.3 实验结果

运行可执行文件前，需要注意的是，由于实验二及后续实验均涉及图形化界面的输出，因此执行过程需在开发板完成，从而使图像能直接显示在 LCD 屏上。此外，开发板在桌面模式下终端无法正常渲染图像，因此需禁用桌面，仅使用终端界面。输入如下命令打开香橙派的配置菜单：

```
1. sudo orangepi-config
```

打开配置菜单后，依次选择 System、Desktop，然后选择<Stop>，最后在终端输入 reboot 命令重启 Linux 系统，发现此时触摸屏上只显示终端的画面（如图 2. 3 所示），表明桌面已经成功禁用，接下来可继续进行实验的测试。

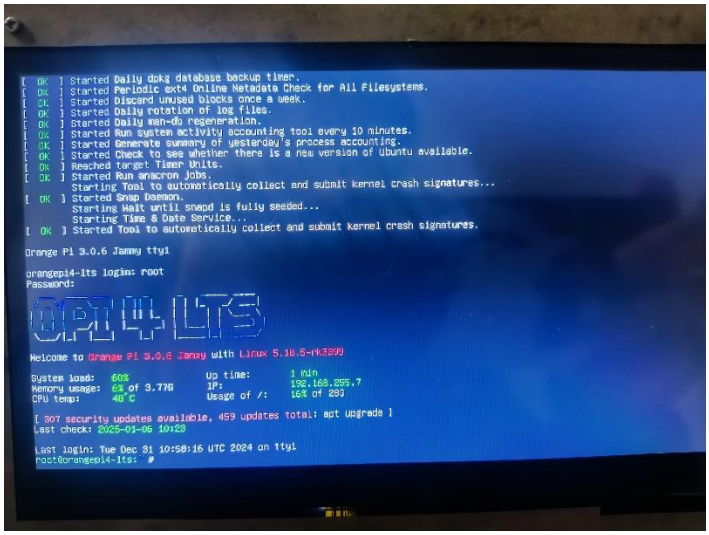

图 2.3 开发板禁用桌面模式

运行编译好的 lab2 可执行文件后，观察到触摸屏上依次进行了点（如图 2.4 所示）、矩形（如图 2.5 所示）的绘制，并在最后展示了三者的综合绘制结果，如图 2.6 所示：

图 2.4 点的绘制

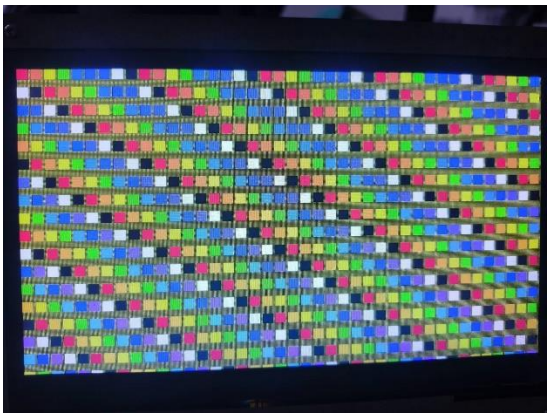

图 2.5 矩形的绘制

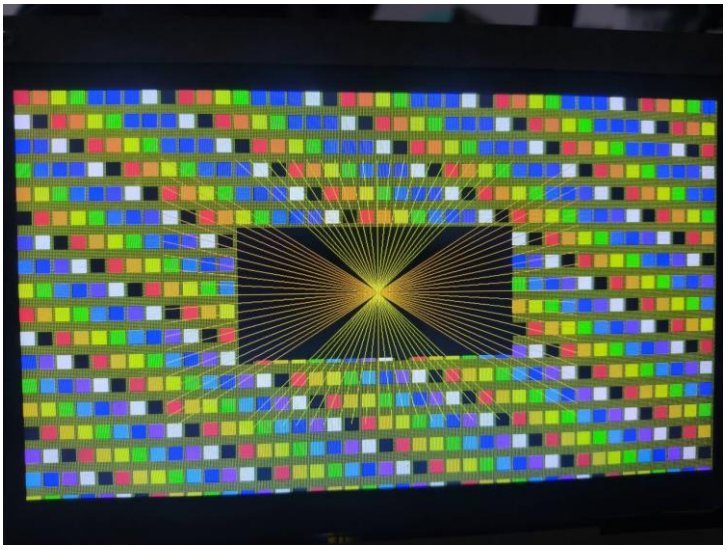

图 2.6 实验二结果

由测试结果可以看出，各种图像均能正确绘制，表明实验二成功。

3 实验三 图片文字显示

3.1 实验要求

实验三的主要要求包括如下几点：

1. **jpg 不透明图片显示**
2. **png 半透明图片显示**
3. **矢量字体显示**
   a) 字模的提取
   b) 字模的显示（只有 alpha 值的位图）

3.2 实验过程

3.2.1 jpg 不透明图片显示

在进行本实验之前，首先需对图片结构体 fb_image 的数据结构有所了解，其定义如下所示：

```
1. typedef struct {
2.         int color_type; /*图片中的像素颜色类型*/
3.         int pixel_w, pixel_h; /*图片的像素宽和高*/
4.         int line_byte;  /*图片中存储一行像素颜色所占的字节数*/
5.         char *content; /*图片的像素颜色值*/
6. } fb_image;
```

其中，color_type 的取值为 1、2 和 3，分别由 FB_COLOR_RGB_8880、FB_COLOR_RGBA_8888 和 FB_COLOR_ALPHA_8 三个宏定义，依次表示不透明 jpg 图片、透明 png 图片和矢量字体的字模图片。与 framebuffer 的表示相同，不透明 jpg 和 png 图片中一个像素的颜色值也占用 4 字节的内存空间，从高到低每 8 位分别表示 A、R、G、B。两种图片的像素颜色类型如图 3. 1 所示：

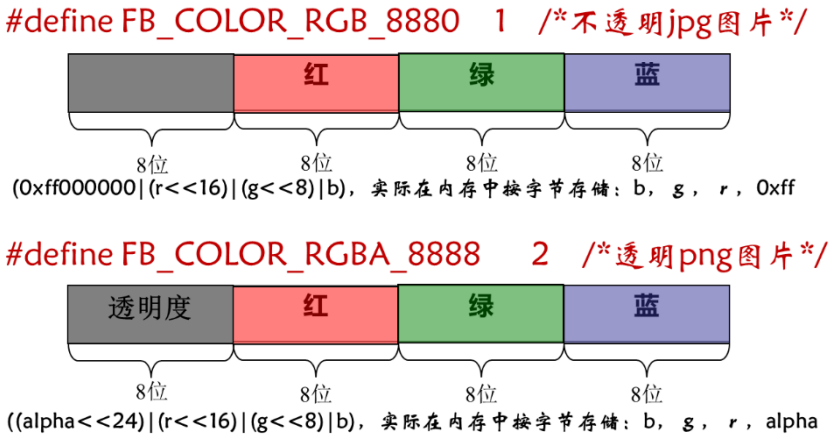

图 3. 1 jpg 和 png 图片的像素颜色类型

矢量字体的字模图片中一个像素的颜色值只占用 1 字节的内存空间，仅包括 A 字段而没有 R、G、B 字段，其像素颜色类型如图 3.2 所示：

图 3.2 矢量字体字模图片的像素颜色类型

接下来在 common/graphic.c 中完成画图函数的具体实现。画图函数根据传入的图片类型执行对应的绘制过程，对于不透明 jpg 图片，简单拷贝图片内存到后缓冲的相应位置即可。该拷贝过程通过与画矩形函数类似的双重循环实现，值得注意的是，由于不透明 jpg 图片中一个像素的颜色值占用 4 字节的内存空间，因此使用循环索引时也需将其放大为原来的 4 倍。

绘制不透明 jpg 图片的具体实现如下所示：

```
1. char *dst = (char *)(buf + y*SCREEN_WIDTH + x);
2. char *src; //不同的图像颜色格式定位不同
3.
4. if(image->color_type == FB_COLOR_RGB_8880) /*lab3: jpg*/
5. {
6.     src = image->content + iy * w * 4 + ix * 4;
7.     for (int i = 0; i < h; i++) {
8.         for (int j = 0; j < w; j++) {
9.         int* src_ = src + i * w * 4 + j * 4;
10.        int* dst_ = dst + i * SCREEN_WIDTH * 4 + j * 4;
11.        *dst_ = *src_;
12.    }
13. }
```

与画矩形函数类似，双重循环的拷贝过程效率较低，为提高程序性能，使用 SIMD 指令替代双重循环来对拷贝过程进行优化，从而使时间复杂度由$O(hw)$降低为$O(h)$，其中 h 和 w 分别表示图片的高和宽。

优化后，绘制不透明 jpg 图片的具体实现如下所示：

```
1. char *dst = (char *)(buf + y*SCREEN_WIDTH + x);
2. char *src; //不同的图像颜色格式定位不同
3.
4. if(image->color_type == FB_COLOR_RGB_8880) /*lab3: jpg*/
5. {
6.     src = image->content + iy * w * 4 + ix * 4;
7.     for (int i = 0; i < h; i++) {
8.         memcpy(dst + i * SCREEN_WIDTH * 4, src + i * w * 4, w * 4);
9.     }
10. }
11.
```

3.2.2 png 半透明图片显示

透明 png 图片相比不透明 jpg 图片增加了对 A 字段的使用。在透明图片中，A 字段的取值与该区域的透明程度的关系如表 3- 1 所示：

表 3- 1 透明程度与 alpha 取值的关系

区域的透明程度	A 字段取值 alpha
空白的区域	alpha == 0
完全不透明实体的区域	alpha == 1.0
半透明的区域	0 < alpha < 1.0

由此可知，理论上透明度的取值范围为浮点数[0, 1.0]。当透明图片与底图叠加绘制时，令透明图片的颜色为 color1、透明度为 alpha1，底图的颜色为 color2、没有透明度或透明度为 1.0，则该像素最终的颜色值 color 的计算表达式如下：

$$color = color1 \times alpha1 + color2 \times (1 - alpha1)$$

因此，对于透明 png 图片，不能使用 SIMD 指令进行批量拷贝，需要使用双重循环逐点计算，并要根据各像素的透明度单独计算每个颜色通道上的颜色值。同样地，使用循环索引时也需将其放大为原来的 4 倍。绘制透明 png 图片中的各像素点时，先比较像素点透明度 alpha 的取值，如果 alpha 等于 0，表示该像素为空白区域，可直接跳过；如果 alpha 等于 1.0，表示该区域为完全不透明实体的区域，可直接拷贝图片内存到后缓冲的相应位置；如果 alpha 在 0 和 1.0 之间，表示该区域为半透明的区域，则单独计算每个颜色通道上的颜色值再写入后缓冲的相应位置。

由于内存空间是从低到高排列的，因此对于一个像素表示的 4 个字节内存空间 src，将其视为一个 char 类型数组，则数组中依次存放 B、G、R 和 A 字段，可通过索引提取各字段的具体值（如 src[3]表示 A 字段的值，即透明度）。

绘制透明 png 图片的具体实现如下所示：

```
1. else if(image->color_type == FB_COLOR_RGBA_8888) /*lab3: png*/
2. {
3.     src = image->content + iy * w * 4 + ix * 4;
4.     for (int i = 0; i < h; i++) {
5.         for (int j = 0; j < w; j++) {
6.             char* src_ = src + i * w * 4 + j * 4;   // B G R A
7.             char* dst_ = dst + i * SCREEN_WIDTH * 4 + j * 4;
8.             alpha = src_[3];
9.             switch (alpha) {
10.                case 0:
11.                    break;
12.                case 255:
13.                    dst_[0] = src_[0];
14.                    dst_[1] = src_[1];
```

```
                    dst_[2] = src_[2];
                    break;
                default:
                    dst_[0] += (((src_[0] - dst_[0]) * alpha) >> 8);
                    dst_[1] += (((src_[1] - dst_[1]) * alpha) >> 8);
                    dst_[2] += (((src_[2] - dst_[2]) * alpha) >> 8);
                    break;
            }
        }
    }
}

```

3.2.3 矢量字体显示

1. 字模的提取

字模包括字模图片和字模图片信息两项内容，类型分别为 fb_image 的指针和 fb_font_info 的指针。字模从矢量字体文件（.ttf 或.ttc 字体文件）提取，因此需先对矢量字体文件进行初始化。

矢量字体文件初始化后，使用 fb_read_font_image 函数对字模进行提取，函数的定义如下所示：

```
fb_image * fb_read_font_image(
            const char *text,
            int pixel_size,
            fb_font_info *info);
```

其中，函数的返回值是字模图片的指针，字模示意图如图 3. 3 所示：

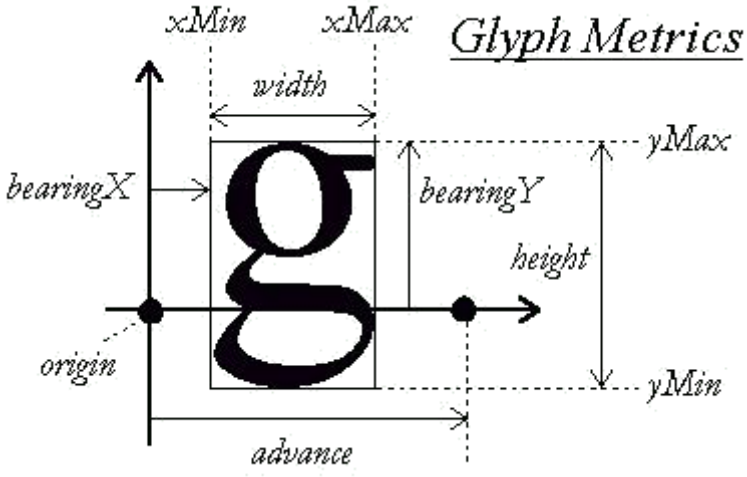

图 3. 3 字模示意图

函数参数的 info 字段是字模图片信息 fb_font_info 结构体的指针，结构体的定义如下所示：

```
1. struct fb_font_info {
2.       int bytes;
3.       int advance_x;
4.       int left;
5.       int top;
6. }
```

其中，left 字段即上图中的 bearingX，top 字段即上图中的 bearingY。根据字模图片信息结构体的有关字段，可以从字模图片中提取出包含完整字体的最小区域，以便后续过程中矢量字体的绘制。

2. 字模的显示

绘制矢量字体的过程与透明 png 图片较为类似，都涉及透明图片与底图的叠加操作，因此需单独计算每个颜色通道上的颜色值。由于字模图片只有透明度字段，没有颜色值字段，其颜色值在绘制时由函数参数决定。

此外，字模图片中一个像素只占 1 个字节的内存空间，因此在双重循环过程中，仅访问后缓冲的内存时需要将循环索引扩大到原来的 4 倍，而访问字模图片的内存时则使用循环索引本身。

绘制矢量字体的具体实现如下所示：

```
 1. else if(image->color_type == FB_COLOR_ALPHA_8) /*lab3: font*/
 2. {
 3.     char r = color >> 16;
 4.     char g = color >> 8;
 5.     char b = color;
 6.
 7.     src = image->content + iy * w + ix;
 8.     for (int i = 0; i < h; i++) {
 9.         for (int j = 0; j < w; j++) {
10.             char* src_ = src + i * w + j;   // A
11.             char* dst_ = dst + i * SCREEN_WIDTH * 4 + j * 4;
12.             alpha = *src_;
13.             switch (alpha) {
14.                 case 0:
15.                     break;
16.                 case 255:
17.                     dst_[0] = b;
18.                     dst_[1] = g;
19.                     dst_[2] = r;
20.                     break;
21.                 default:
22.                     dst_[0] += (((b - dst_[0]) * alpha) >> 8);
23.                     dst_[1] += (((g - dst_[1]) * alpha) >> 8);
24.                     dst_[2] += (((r - dst_[2]) * alpha) >> 8);
```

```
25.                     break;
26.             }
27.         }
28.     }
29. }
```

3.3 实验结果

运行编译好的 lab3 可执行文件后，观察触摸屏上分别显示了不透明 jpg 图片（羊驼）、透明 png 图片（绿色带状物）和矢量字体（诗句、“嵌”字），如图 3.4 所示。且各元素尺寸、颜色和叠加关系均正常，表明画图函数的设计与实现是合理的，实验三成功。

图 3.4 实验三结果

最后，编译 lab-2022-st/test/test.c 测试脚本文件，运行编译好的 test 可执行文件，以对实验二和实验三中实现的所有画图函数（点、矩形、线、两种图像和文本）进行性能测试，测量每种画图函数的渲染时间。

运行 test 可执行文件后，屏幕上依次完成了各图案的绘制过程并记录下了各自的渲染时间，最后以表格的形式进行汇总，如图 3.5 所示：

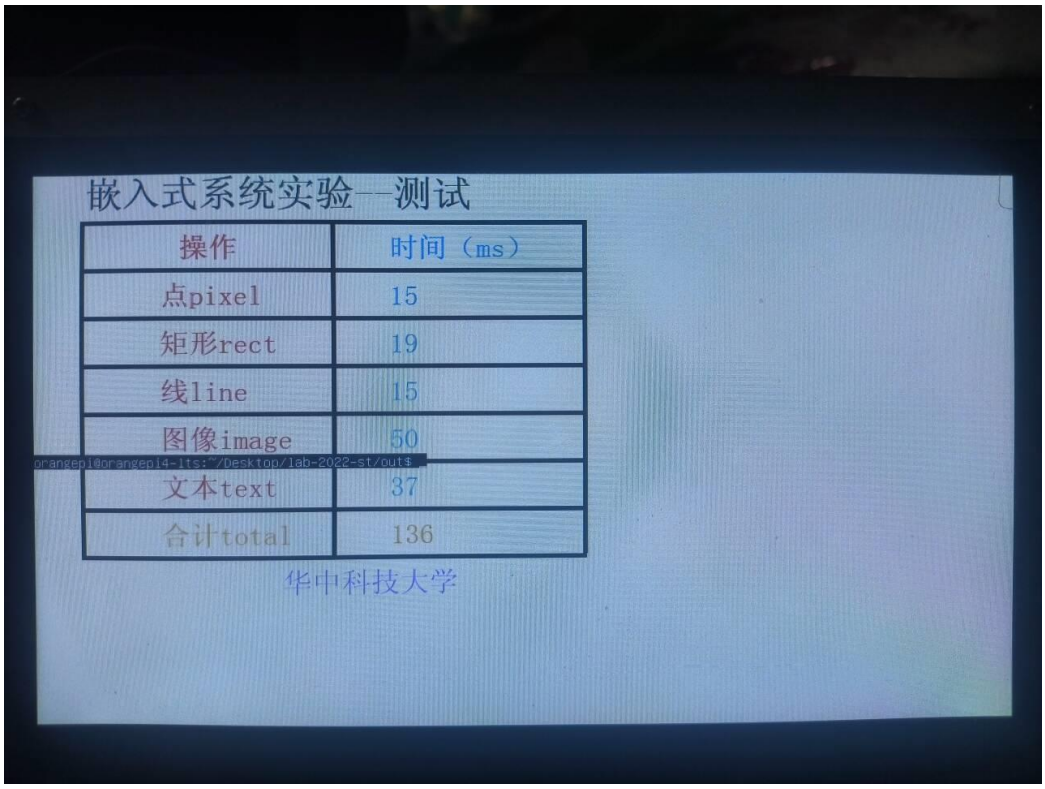

图 3.5 实验三测试结果

由测试结果可以看到，所有画图操作的合计时间仅为 136ms（由于程序运行时间和运行时的机器状况紧密相关，因此每次测试的运行时间在正常范围内波动），询问老师后得知较好的画图操作的合计时间在 120~200ms 范围左右，表明程序的性能是极佳的，反映了此前各种优化手段（如 SIMD 等）的有效性。

4 实验四 多点触摸开发

4.1 实验要求

实验四的主要要求包括如下几点：

1. **Linux 下的触摸屏驱动接口**
   a) Input event 的使用
   b) 多点触摸协议（Multi-touch Protocol）
2. **获取多点触摸的坐标**
3. **多点触摸开发**

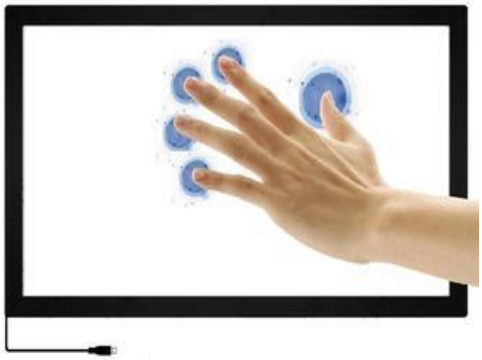

图 4.1 多点触摸开发效果

a) 对应每个手指触点的圆的颜色不同
b) 实时跟踪触点，只显示当前位置
c) 之前位置的圆要清除掉
d) 屏幕不能明显闪烁

4.2 实验过程

4.2.1 Linux 下的触摸屏驱动接口

1. Input event 的使用

在 Linux 下，触摸屏通过驱动接口 Input event 与开发板进行信息传递。具体而言，触摸屏将点击事件实时写入文件/dev/input/eventX 中，其中不同设备的 X 值可能不同，一般为 0、1、2、3 等，需要通过查看/proc/bus/input/devices 来确认。开发板中的程序通过不断读取/dev/input/eventX 文件中的内容，并将信息封装到 input_event 结构体中来获取点击事件，结构体的定义如下所示：

```
1. struct input_event {
2.      struct timeval time;
3.      __u16 type;
```

```
4.      __u16 code;
5.      __s32 value;
6. };
```

其中，字段 type 表示点击事件的类型，包括 EV_ABS、EV_SYN 等；字段 code 表示点击事件的返回码，用来区分点击事件中的不同数据，如点击位置的横、纵坐标，轨迹值等；字典 value 表示点击事件相应数据的返回值。这种文件式的交互，使得程序能通过触摸屏驱动接口来实时获取点击事件。

2. 多点触摸协议

多点触摸协议运行触摸屏上同时发生多个点击事件，比如触摸屏上多个手指同时按下、滑动等。其实现原理为：当有一个触摸事件发生时，连续读取多条记录，并只发送变化的内容到调用触摸屏驱动接口的应用程序。其中，记录存储在上述的 input_event 结构体中，多点触摸协议规定的记录内容分别如表 4- 1 所示：

表 4- 1 多点触摸协议

data.type	data.code	data.value
EV_ABS	ABS_MT_POSITION_X	坐标值(0 ~ screen_width-1)
EV_ABS	ABS_MT_POSITION_Y	坐标值(0 ~ screen_height-1)
EV_ABS	ABS_MT_SLOT	轨迹值(0,1,2,3,4)
EV_ABS	ABS_MT_TRACKING_ID	轨迹值(>=0, -1)
EV_SYN	SYN_REPORT	无

应用程序通过 touch_read 函数读取多点触摸设备文件/dev/input/eventX 的内容并返回一条点击事件记录，touch_read 函数封装在 touch_event_cb 函数中，该函数在程序主函数添加到任务循环中，当多点触摸设备文件可读时，会自动调用多个 touch_event_cb 函数，依次读取多点触摸设备文件中的点击事件记录，并交由应用程序处理，从而完成多点触摸的响应。

4.2.2 获取多点触摸的坐标

多点触摸时点击区域的坐标会记录在文件中，读取记录和封装成 input_event 结构体的操作均已在 common/touch.c 文件中实现。然而，驱动上传的横、纵坐标范围均为$[0, 4096)$，与触摸屏的实际尺寸 1024×600 并不相符，因此在读取坐标时需对驱动记录的坐标进行转换。对于驱动上传的横坐标x、纵坐标y，屏幕的宽 SCREEN_WIDTH、高 SCREEN_HEIGHT，变换得到的屏幕范围的横坐标x'、纵坐标y'的计算表达式分别如下所示：

$$x = x \cdot \text{SCREEN\_WIDTH} \gg 12$$

$$y = y \cdot \text{SCREEN\_HEIGHT} \gg 12$$

经过上述变换，touch_read 函数的返回值 x、y 即为屏幕范围内多点触摸的坐标。在后续多点触摸开发中，程序根据获取到的多点触摸的坐标即可在触摸屏上实现图形的精确绘制。

4.2.3 多点触摸开发

本实验要求使用多点触摸开发完成如图 4. 1 的交互程序。其中，每个手指的触摸点处要进行圆的绘制，因此须先在 common/graphic.c 文件中完成画圆函数的设计与实现。画圆函数接收五个参数，分别为圆心的横、纵坐标，圆的半径 r，圆的粗细 b 和圆的颜色值。其中，圆的粗细表示圆的绘制范围为半径为 r 的圆和半径为 r—b 的同心圆之间的圆环区域。

尽管传统的双重循环方法实现较为简单，但渲染时间较长，而本实验的多点触摸决定了触摸屏上要同时渲染多个圆的图像，且随着手指的移动将频繁更新。因此，实验对图形渲染的性能要求较高，该绘制方法会使触摸交互不流畅，影响使用体验。最终，参考计算机图形学的有关知识，选用 Bresenham 算法来实现圆的绘制。

Bresenham 算法的原理是通过整数运算来避免浮点运算，提高绘制效率并减少误差，从而在离散像素网格上高效地绘制直线和圆。Bresenham 算法绘制圆的原理图如图 4. 2 所示：

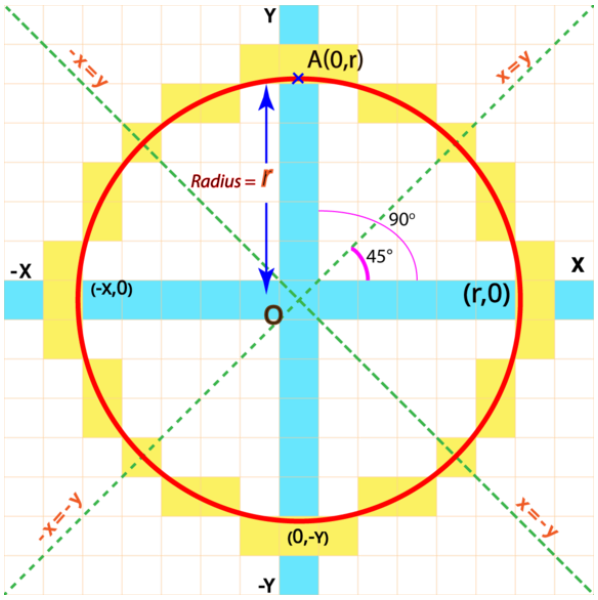

图 4. 2 Bresenham 算法绘制圆的原理图

而对于圆的粗细 b，只需通过循环，依次使用 Bresenham 算法绘制半径为整数且在$[r-b+1,\ r]$范围内的所有圆即可。画圆函数的具体实现如下所示：

```
1. void fb_draw_circle(int x0, int y0, int r, int b, int color) {
2.     int *buf = _begin_draw(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT);
3.     void plot(int x, int y) {
4.         if(x >= 0 && x < SCREEN_WIDTH && y >= 0 && y <SCREEN_HEIGHT) {
5.             *(buf + y * SCREEN_WIDTH + x) = color;
6.         }
7.     }
8.     for(int w = 0; w < b; w++) {
9.         int radius = r - w;
10.        int x = 0;
11.        int y = radius;
12.        int d = 3 - 2 * radius;
13.        while(x <= y) {
14.            // 在每个八分圆弧上画更多的点
15.            plot(x0 + x, y0 + y);
16.            plot(x0 + x + 1, y0 + y); // 填补横向间隙
17.            plot(x0 - x, y0 + y);
18.            plot(x0 - x - 1, y0 + y);
19.            plot(x0 + x, y0 - y);
20.            plot(x0 + x + 1, y0 - y);
21.            plot(x0 - x, y0 - y);
22.            plot(x0 - x - 1, y0 - y);
23.
24.            plot(x0 + y, y0 + x);
25.            plot(x0 + y, y0 + x + 1); // 填补纵向间隙
26.            plot(x0 - y, y0 + x);
27.            plot(x0 - y, y0 + x + 1);
28.            plot(x0 + y, y0 - x);
29.            plot(x0 + y, y0 - x - 1);
30.            plot(x0 - y, y0 - x);
31.            plot(x0 - y, y0 - x - 1);
32.            if(d < 0) {
33.                d = d + 4 * x + 6;
34.            } else {
35.                d = d + 4 * (x - y) + 10;
36.                y--;
37.            }
38.            x++;
39.        }
40.    }
41. }
42.
```

完成画圆函数的实现后，接下来对 lab4/main.c 文件中的 touch_event_cb 函数进行修改来增加对多点触摸的响应，即根据多点触摸的信息在触摸屏上绘图。

首先，对屏幕背景和 5 根手指对应的圆分别使用不同的颜色来表示，并使用手指颜色数组来管理各手指对应的圆的颜色，颜色和数组的定义如下所示：

```
1. #define COLOR_BACKGROUND    FB_COLOR(0x00, 0x00 ,0x00)
2. #define COLOR_RED           FB_COLOR(0xff, 0x00, 0x00)
3. #define COLOR_ORANGE        FB_COLOR(0xff, 0xa5, 0x00)
4. #define COLOR_YELLOW        FB_COLOR(0xff, 0xff, 0x00)
5. #define COLOR_GREEN         FB_COLOR(0x00, 0xff, 0x00)
6. #define COLOR_BLUE          FB_COLOR(0x00, 0x00, 0xff)
7. static int colors[5] = {COLOR_RED, COLOR_ORANGE, COLOR_YELLOW,
COLOR_GREEN, COLOR_BLUE};
```

其次，使用 FingerState 结构体保存手指触摸状态，并使用结构体数组来管理各手指的手指触摸状态，结构体和数组的定义如下所示：

```
1. // 手指状态结构体
2. typedef struct {
3.     int active; // 当前手指是否点击
4.     int x;  // 最后的 x 坐标
5.     int y;  // 最后的 y 坐标
6. } FingerState;
7. static FingerState fingers[5] = {0};
```

其中，在 fingers 手指触摸状态数组中通过各手指的索引来获取相应的手指触摸状态。对于一个 FingerState 手指触摸状态结构体，active 字段表示当前手指是否点击，x 和 y 字段分别表示当前手指最后一次点击的横坐标和纵坐标，即屏幕上最新的手指触摸坐标。

在 touch_event_cb 函数内部，每次调用时通过 touch_read 函数获取当前事件的手指 id（根据当前触摸屏上的手指个数自增）、手指触摸坐标和点击事件类型。先判断手指 id 是否在 0~4（分别表示 5 根手指）范围内，否则直接返回。接下来对点击事件类型进行枚举，可能的点击事件类型包括按压（TOUCH_PRESS）、移动（TOUCH_MOVE）、释放（TOUCH_RELEASE）和错误（TOUCH_ERROR），并根据点击事件类型作出相应的响应。

对于按压事件类型，表示手指 id 对应的手指首次在触摸屏上点击，因此根据手指 id 获取对应的手指触摸状态结构体，将 active 字段设为 1，x、y 字段分别设为手指触摸坐标。最后，调用此前实现的画圆函数，圆心坐标为手指触摸坐标，圆的半径和粗细设为固定值，圆的颜色根据手指 id 从手指颜色数组中获取。调用画圆函数后，同样使用 fb_update 函数进行双缓冲刷新的操作。这样便在触摸屏上完成了手指触摸点处圆的绘制，且不同手指触摸时渲染的圆将具有不同的颜色，同时该手指 id 的触摸信息也保存在了手指触摸状态结构体中以方便维护。

处理按压事件类型的具体实现如下所示：

```
1. case TOUCH_PRESS:
```

```
2.      fingers[finger].active = 1;
3.      fingers[finger].x = x;
4.      fingers[finger].y = y;
5.      fb_draw_circle(x, y, 50, 5, colors[finger]);
6.      fb_update();
7.      break;
```

对于移动事件类型，表示手指 id 对应的手指已在触摸屏上点击，并进行了一段移动，到达当前的手指触摸坐标。然而，由于此前该手指对应的圆已在触摸屏上渲染，因此须先将其擦除，再对当前手指触摸坐标处的圆进行绘制，来达到圆的移动的效果。擦除旧的轨迹时，可以使用画矩形函数重绘整张屏幕的方法，但效率显然较低，因此采用只更新发生变化的区域的方法，即根据手指 id 获取对应的手指触摸状态结构体，以原来的 x、y 字段为圆心坐标，调用画圆函数，并指定颜色为屏幕背景颜色 COLOR_BACKGROUND，从而实现了旧轨迹的高效擦除。

接下来与按压事件类型的执行过程基本相同，先将手指触摸状态结构体中的 x、y 字段分别更新为当前的手指触摸坐标，再对新的轨迹进行重绘，最后进行双缓冲刷新。可以发现，移动事件类型的处理相比按压事件类型仅仅增加了对旧轨迹的擦除操作。

移动事件类型的具体实现如下所示：

```
1. case TOUCH_MOVE:
2.      // 只更新发生变化的区域
3.      fb_draw_circle(fingers[finger].x, fingers[finger].y, 50, 5,
COLOR_BACKGROUND);
4.      fingers[finger].x = x;
5.      fingers[finger].y = y;
6.      fb_draw_circle(x, y, 50, 5, colors[finger]);
7.      fb_update();
8.      break;
```

对于释放事件类型，表示手指 id 对应的手指已在触摸屏上点击，并在当前时刻释放。因此，先根据手指 id 获取对应的手指触摸状态结构体，再使用与移动事件类型处理中的相同方法擦除旧的轨迹，同时进行双缓冲刷新，最后将手指触摸状态结构体的 active 字段置为 0。

释放事件类型的具体实现如下所示：

```
1. case TOUCH_RELEASE:
2.      fb_draw_circle(fingers[finger].x, fingers[finger].y, 50, 5,
COLOR_BACKGROUND);
3.      fb_update();
4.      fingers[finger].active = 0;
5.      break;
```

对于错误事件类型，表示多点触摸过程产生了异常，此时在终端上输出相应的报错提示，并关闭多点触摸设备文件句柄，最后删除多点触摸设备文件，从而停止多点触摸协议的运行。

错误事件类型的具体实现如下所示：

```
1. case TOUCH_ERROR:
2.     printf("close touch fd\n");
3.     close(fd);
4.     task_delete_file(fd);
5.     return;
6. }
```

通过主函数中的任务循环，和以上各种事件的枚举处理，从而实现了本实验要求的多点触摸开发应用程序。

4.3 实验结果

运行编译好的 lab4 可执行文件，进入交互界面，分别使用不同个数的手指在触摸屏上点击和移动，观察触摸屏上圆的渲染情况，如图 4.3 和图 4.4 所示：

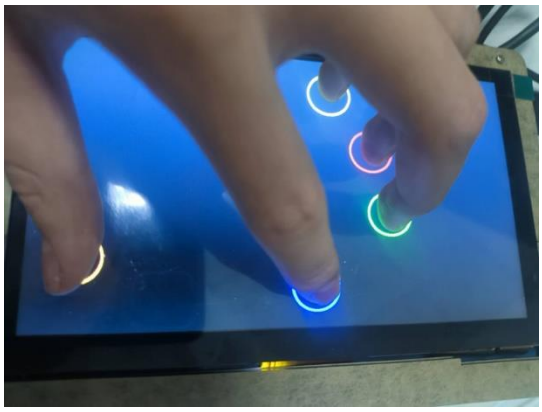

图 4.3 实验四结果

图 4.4 实验四结果

手指移动过程中，屏幕上各手指位置所渲染的圆均能流畅更新，且没有出现拖尾和重影等现象，表明实验四成功。

5 实验五 蓝牙通讯

5.1 实验要求

实验五的主要要求包括如下几点：

1. 正确配置并启动蓝牙服务
2. 熟练使用蓝牙的常用命令工具
3. 通过 RFCOMM 串口（蓝牙串行通讯）实现无线通讯
4. 测试通过 lab5 实验代码

5.2 实验过程

5.2.1 配置并启动蓝牙服务

首先，使用如下命令打开并编辑开发板上的配置文件：

```
1. nano /etc/systemd/system/dbus-org.bluez.service
```

在 ExecStart 赋值中增加兼容性标志-C，并在下方添加新行，对 ExecStartPost 进行赋值，如下所示：

```
1. ExecStart =/usr/lib/bluetooth/bluetoothd -C
2. ExecStartPost=/usr/bin/sdptool add SP
```

配置好的配置文件如所示，接下来使用 reboot 命令重启开发板。

重启后，使用 hciconfig 命令查看是否存在蓝牙的设备节点，如果存在，说明蓝牙初始化正常。然后，使用 bluetoothctl 命令进入蓝牙 shell，并使用如下命令分别启动蓝牙、设置可发现模式、设置可配对模式，最后扫描周围的蓝牙设备：

```
1. power on
2. discoverable on
3. pairable on
4. scan on
```

开始扫描后，打开手机蓝牙，观察到终端上扫描到手机蓝牙设备并显示了蓝牙的相关信息。记下手机蓝牙设备的 MAC 地址，使用如下命令进行配对：

```
1. pair 蓝牙 MAC
```

配对成功后，手机蓝牙界面会有所提示，如图 5. 1 所示：

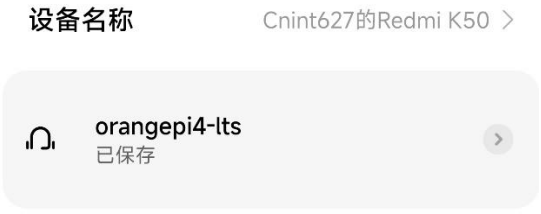

图 5. 1 蓝牙配对成功

5.2.2 蓝牙串行通信实现

蓝牙设备之间的通讯编程方式和网络 socket 类似，具体过程为：服务器建立蓝牙 socket，并监听（listen）客户端连接；客户端连接（connect）到服务器。Linux 下蓝牙技术的实现框架 bluez 提供了 rfcomm 工具程序，可以将蓝牙 socket 转换成串行设备，方便用户编程开发。

本实验中，使用开发板作为服务器，手机作为客户端，进行蓝牙串行通信测试。开发板与手机蓝牙配对成功后，运行如下命令启动服务器监听：

```
1. rfcomm -r watch 0 1
```

其中，watch 语句表示客户端退出后，rfcomm 不退出，继续监听；如果使用 listen 语句，则服务器连接只创建一次，客户端退出后，rfcomm 也退出。两种连接方式可根据实际需要选取。0 表示连接建立后创建的终端设备是/dev/rfcomm0，1 表示蓝牙的连接通道序号，-r 表示终端设备进入 raw 模式（内核对数据不加工，不回显）。

手机上安装“SPP 蓝牙串口”程序，在蓝牙串口程序中与已配对的开发板建立连接。连接成功后，观察开发板终端和手机蓝牙串口程序的输出分别如图 5. 2 和图 5. 3 所示：

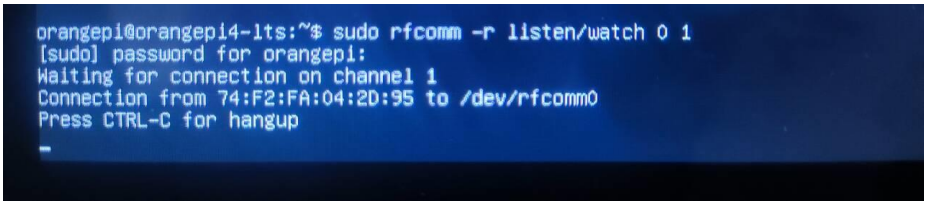

图 5. 2 开发板蓝牙连接结果

图 5. 3 手机蓝牙连接结果

最后，在开发板的另一终端上编译 lab5 测试程序，并运行 lab5 可执行文件。

5.3 实验结果

运行编译好的 lab5 可执行文件后，先使用手机蓝牙串口程序向开发板发送信息。发送“Hello World”、“你好，华中科技大学”和“2024”等信息后，观察到开发板在触摸屏上均能正常显示出对应内容，如图 5.4 所示：

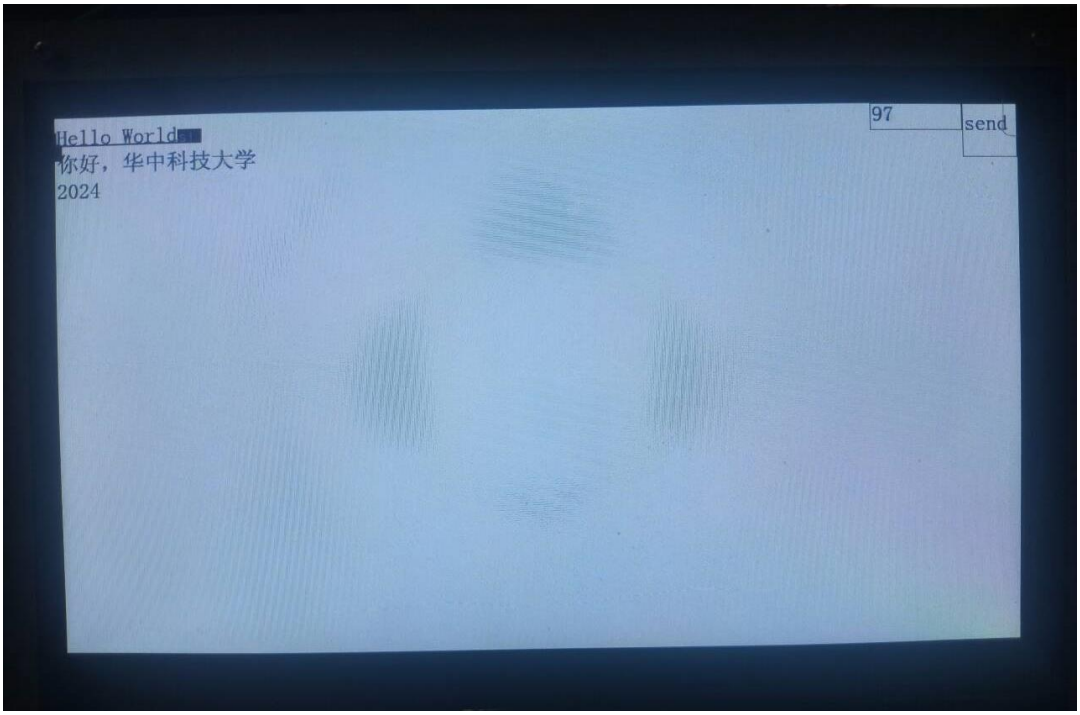

图 5.4 实验五开发板接收消息结果

接着，再使用开发板向手机蓝牙串口程序发送信息。点击触摸屏右上角的“send”按钮，开发板会向手机蓝牙串口程序发送“hello”信息，观察手机上也能正常显示出对应内容，如图 5.5 所示：

图 5.5 实验五手机接收消息结果

以上实验结果表明，开发板与手机之间能正常进行蓝牙串行通信，且能适配多种类型的文本传输（通过 UTF-8 编解码实现），实验五成功。

6 实验六 综合实验

6.1 实验要求

综合运用所学内容，设计并实现一个在多个开发板之间进行蓝牙互联协作的程序或一个功能比较复杂的单机程序。例如：

联机程序：

1. 共享白板：两个开发板之间共享屏幕手绘内容
2. 共享文件：显示远程开发板上的目录文件列表,选中指定文件，显示指定文件内容（文本和 png/jpg 图片）
3. 语音消息：两个开发板之间语音转文字并发送到对方屏幕
4. 联网游戏：比如五子棋等
5. 其他

单机程序：

1. 图片浏览器：图片放大、缩小、图片尺寸超过显示区域时手指拖动显示
2. 视频播放器
3. 语音识别程序：语音控制屏幕显示
4. 游戏或其他

6.2 实验过程

6.2.1 综合实验设计

本次综合实验的选题是**联网游戏**，具体为**双人蓝牙象棋对战游戏**（后简称“象棋游戏”）。象棋游戏通过两台开发板之间的蓝牙串行通信实现数据交换，具备完善的图形化对战界面（如 6.3 实验结果一节所示），并严格遵从中国象棋的行棋规则。本实验的所有代码已在 GitHub 仓库中开源，仓库地址如下：

https://github.com/cnint0627/hust_embedded_system

象棋游戏采用模块化的系统架构设计，自顶向下依次为：

1. **流程控制层**
2. **游戏逻辑层**
3. **棋子逻辑层**
4. **行棋日志层、蓝牙传输层**
5. **UI 层**

其中，流程控制层负责程序整体流程的控制，包括蓝牙连接、点击事件处理、各层模块的交互等；游戏逻辑层负责游戏运行流程的控制，包括棋子选择、棋子移动等；棋子逻辑层负责棋子移动逻辑的控制，规定了具体的行棋规则，并对每次移动进行合法性检查；行棋日志层和蓝牙传输层都接受棋子逻辑层的输出（棋

子的移动信息）作为输入，其中行棋日志层根据移动信息生成相应的日志，蓝牙传输层将移动信息传输给远程主机，然后等待远程主机的响应；最后，UI 层根据本轮棋子的移动信息和日志实现屏幕的重绘。

这种模块化的设计构成了低耦合、高内聚的层次化系统，提高了代码的复用性、可扩展性和可维护性，从而使程序设计更为灵活、清晰。系统的整体架构图如图 6. 1 所示：

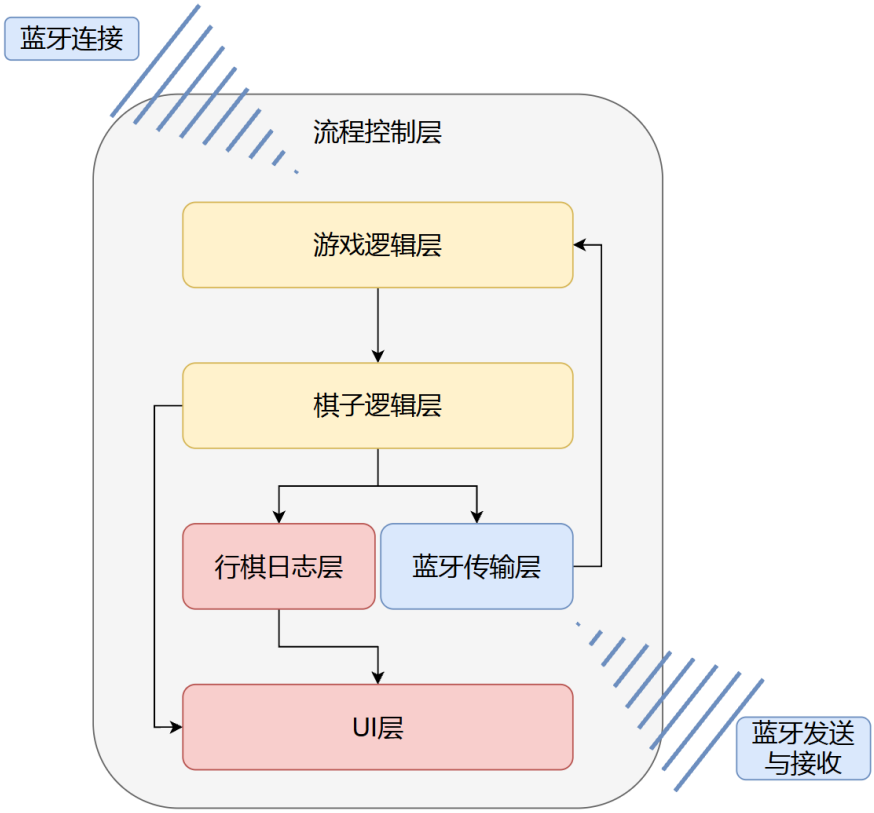

图 6. 1 系统整体架构图

6.2.2 综合实验实现

象棋游戏的实现过程仍然按系统的模块自顶向下进行阐述：

1. 流程控制层

流程控制层负责程序整体流程的控制，对应根目录的 main.c 文件。执行象棋游戏的可执行程序 lab-final 时，传入额外的两个参数，命令如下所示：

```
1. ./lab-final [event_id] [player_id]
```

其中，event_id 为多点触摸设备文件的 id，根据设备的实际情况传入（一般为 0、1、2、3 之一）；player_id 为本次游戏的玩家 id（0 或 1），两台设备联机游戏时，先按照实验 5 中的方法进行蓝牙配对和串口通信连接，再由 player_id 确定不同设备在游戏内的身份和行棋顺序（0 号玩家为红方，1 号玩家为黑方，红方先手）。

程序启动并接收上述两个参数后，分别对 UI、多点触摸协议、蓝牙、游戏控制器等组件进行初始化。其中，UI 将“游戏菜单”界面初次渲染，多点触摸协议和蓝牙为硬件层面的初始化，游戏控制器对全局游戏状态初始化，具体内容将在游戏逻辑层进行详细介绍。

初始化完成后，进入象棋游戏的主循环。主循环通过如下的命令设置延迟时间为 16ms，即使帧率稳定在 60fps 左右，从而在确保画面流畅刷新的同时，减少程序的性能开销：

```
1. task_delay(16);  // ~60fps
```

主循环中通过多点触摸协议不断获取当前帧屏幕触摸的坐标，如果当前回合为自身回合，则根据触屏坐标作出相应响应，如果当前回合为对手回合，则等待远程蓝牙消息传输。

当前回合为自身回合时，首先判断触屏坐标处是否为按钮，并根据按钮类型划分为 GAME_BTN_START（开始游戏）、GAME_BTN_RESTART（重新开始）、GAME_BTN_EXIT（退出游戏）、GAME_BTN_NONE（触屏坐标处非按钮）四个类型。其中，对于开始游戏和退出游戏按钮，由于游戏菜单（SCREEN_MENU）和游戏对战（SCREEN_GAME）界面均出现这两个按钮，因此还需通过判断当前界面来执行不同的点击事件。

如果触屏坐标处不为按钮，说明可能为棋子或无关区域，此时需要关注的是棋子点击事件。同样地，先判断当前界面是否为游戏对战界面，并将触屏坐标通过计算转换为棋盘坐标（由连续的触屏坐标到离散的棋盘坐标），从而获取当前选中的棋盘格。进而，通过游戏逻辑层、棋子逻辑层完成棋子选择、棋子移动等操作，然后通过行棋日志层记录日志，并通过蓝牙传输层将本轮行棋信息传输给对方，最后通过 UI 层更新本地界面。

当前回合为对方回合时，关于棋子点击事件的监听将失效，转而持续等待远程蓝牙消息传输。当收到远程蓝牙消息（即对方的行棋信息）时，相当于对方程序代替我们完成了游戏逻辑层和棋子逻辑层的任务，并得到了本轮行棋信息，此时通过行棋日志层记录日志，最后通过 UI 层更新本地界面即可。

流程控制层所控制的程序整体流程如图 6. 2 所示。

2. 游戏逻辑层

游戏逻辑层负责游戏运行逻辑的控制，对应根目录下的 game.c 文件。对于流程控制层中提及的游戏初始化、棋子选择、棋子移动等事件均在游戏逻辑层完成。其中，棋子移动的操作还需与棋子逻辑层配合，在合理的行棋规则下进行。

游戏初始化的过程完成了游戏结构体 Game 初始化，Game 结构体的具体内容如下所示：

```
1. // 游戏结构体
2. typedef struct {
```

```
    Chess chess;                    // 棋盘状态
    enum GameState state;           // 游戏状态
    int player_id;                  // 当前玩家(0:红方,1:黑方)
    int current_player;             // 当前下棋的玩家
    bool has_selected;              // 是否已选中棋子
    int selected_x, selected_y;     // 选中的棋子位置
} Game;
```

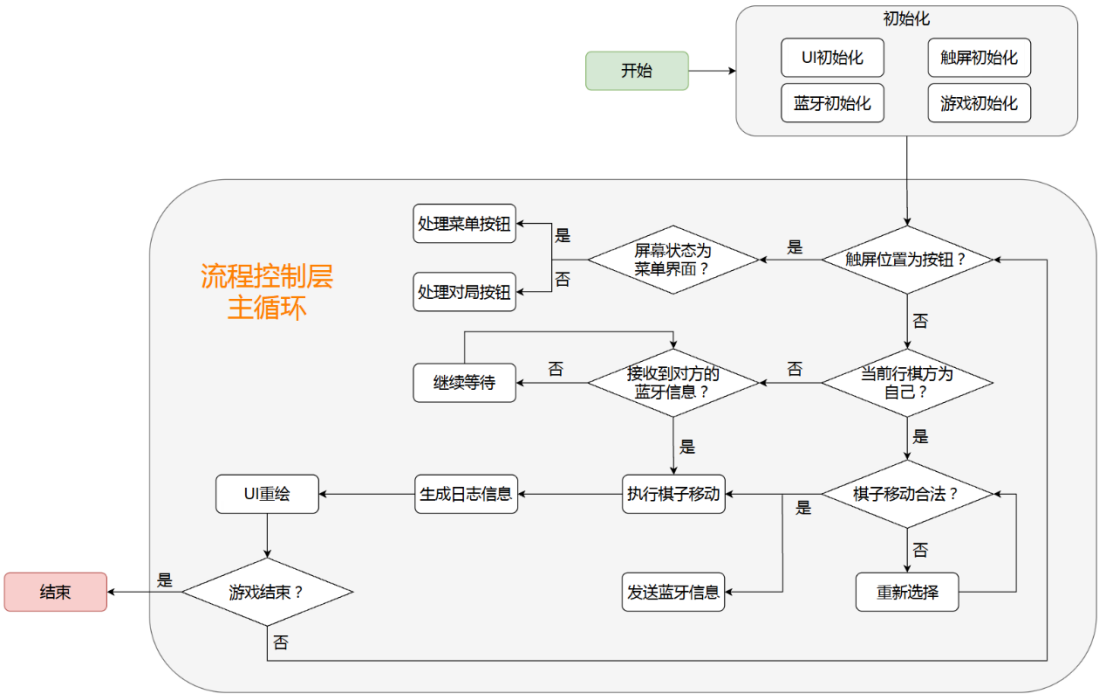

图 6.2 流程控制层整体运行流程

其中，chess 字段表示棋盘状态，记录棋盘的当前格局（layout）；state 字段表示当前游戏状态，包括 GAME_PLAYING（游戏中）、GAME_WIN_RED（红方胜）、GAME_WIN_BLACK（黑方胜），用作游戏进行和结束的标记；player_id 字段表示当前设备对应的玩家 id；current_player 表示当前回合负责行棋的玩家；has_selected、selected_x 和 selected_y 字段分别记录当前是否已选择棋子和所选择棋子的棋盘坐标。

初始化完成后，游戏逻辑层主要负责处理由流程控制层发起的棋盘点击事件时，包括棋子选择和棋子移动，在 game_handle_select 函数中实现，其具体实现如下所示：

```
void game_handle_select(Game *game, int x, int y) {
    if(game->state != GAME_PLAYING) {
        return;
    }

    if(!game->has_selected) {
        // 选中棋子
        Piece *piece = &game->chess.board[y][x];
```

```
 9.         if(piece->type != PIECE_NONE && piece->color ==
game->current_player) {
10.             game->selected_x = x;
11.             game->selected_y = y;
12.             game->has_selected = true;
13.         }
14.     } else {
15.         // 移动棋子
16.         if(chess_is_valid_move(&game->chess,
17.             game->selected_x, game->selected_y, x, y)) {
18.
19.             // 执行移动
20.             chess_move_piece(&game->chess,
21.                 game->selected_x, game->selected_y, x, y);
22.
23.             // 检查游戏是否结束
24.             if(chess_is_game_over(&game->chess)) {
25.                 game->state = game->current_player == 0 ?
26.                             GAME_WIN_RED : GAME_WIN_BLACK;
27.             } else {
28.                 // 切换玩家
29.                 game->current_player = !game->current_player;
30.             }
31.         }
32.         game->has_selected = false;
33.     }
34. }
35.
```

该函数接收三个参数，分别为 Game 结构体指针和棋盘点击事件中的棋盘横、纵坐标。首先，根据 Game 结构体判断当前游戏状态是否为“游戏中”，如果是则继续，再判断当前是否已选择棋子。

如果当前未选择棋子，判断传入的棋盘坐标处是否为本方棋子，如果是则选中。如果当前已选择棋子，且棋盘坐标处为已选择棋子本身，则取消选中；如果棋盘坐标处为其他本方棋子，则更换已选择棋子为该棋子；如果棋盘坐标处为空格或对方棋子，则进行棋子移动操作，该操作通过棋子逻辑层提供的行棋规则进行合法性校验。

当移动成功时，检查游戏是否结束并切换当且回合负责行棋的玩家为对方。最后，由流程控制层调用游戏逻辑层的 game_get_move_info 函数获取本轮的行棋信息，并封装到 MoveInfo 结构体中，分别用于行棋日志层和蓝牙传输层的数据传递。

3. 棋子逻辑层

棋子逻辑层负责棋子移动逻辑的控制，对应根目录下的 chess.c 文件。其规定了完整的行棋规则，主要作用为对游戏逻辑层发起的棋子移动事件进行合法性校验。同时，棋子逻辑层还定义了各种棋子的类型、棋盘的布局、棋子的初始布局等操作，并在合法性校验通过后负责棋盘格局的更新。

棋子逻辑层定义的棋子类型和棋子结构体分别如下所示，包括中国象棋中的所有棋子，并将同类型的不同名称棋子（如“车”和“車”）归为一类，使用棋子结构体中的颜色代码（0 表示红方，1 表示黑方）进行区分：

```
// 棋子类型
enum PieceType {
    PIECE_NONE = 0,
    PIECE_KING,     // 将/帅
    PIECE_ADVISOR,  // 士/仕
    PIECE_BISHOP,   // 象/相
    PIECE_KNIGHT,   // 马
    PIECE_ROOK,     // 车
    PIECE_CANNON,   // 炮
    PIECE_PAWN      // 卒/兵
};
```

```
// 棋子结构
typedef struct {
    enum PieceType type;  // 类型
    int color;            // 0:红方, 1:黑方
    const char *name;     // 显示名称
} Piece;
```

棋盘格局为高为 10、宽为 9 的二维数组，即数组的布局为 10 行 9 列。流程控制层对棋盘进行了初始化，即通过棋子逻辑层完成了棋盘清空和双方初始棋子放置的过程。

当游戏逻辑层发起棋子移动事件时，首先调用棋子逻辑层的移动规则检查函数 chess_is_valid_move 进行合法性校验，并返回本次移动是否合法。如果本次移动合法，游戏逻辑层继续调用棋子逻辑层的棋子移动函数 chess_move_piece 进行棋盘格局的更新，同时调用棋子逻辑层的游戏结束检查函数 chess_is_game_over 判断双方的将/帅是否还在棋盘中，从而对当前游戏状态进行检查。

在移动规则检查函数中，使用大量的 switch-case 语句分别对各类棋子的行棋规则进行了规定，通过判断传入的棋子移动信息是否满足该类棋子的移动规则来完成合法性校验。例如，“象”（PIECE_BISHOP）的行棋规则较为复杂，包括不能过河、走田字和象眼检查（田字中间不能有其他棋子）等限制，其具体实现如下所示：

```
case PIECE_BISHOP: {
```

```
    // 不能过河
    if(color == 0 && to_y > 4) return false;
    if(color == 1 && to_y < 5) return false;
    // 走田字
    if(dx != 2 || dy != 2) return false;
    // 象眼检查
    return chess->board[(from_y + to_y)/2][(from_x + to_x)/2].type ==
PIECE_NONE;
}
```

可见，游戏逻辑层实际上是对游戏运行逻辑的上层封装，而实际的棋子移动则通过棋子逻辑层进行下层调用。这种分层的模块化设计确保了系统运行的高稳定性与可扩展性。

4. 行棋日志层

行棋日志层负责根据棋子移动信息生成相应的行棋日志，并将日志信息传递给 UI 层进行渲染，对应根目录下的 log.c 文件。流程控制层对行棋日志层进行初始化的过程完成了行棋日志结构体的初始化，其具体内容如下所示：

```
// 游戏日志结构
typedef struct {
    char logs[MAX_LOG_LINES][64];  // 日志内容
    int log_count;                 // 日志行数
} GameLog;
```

其中，日志内容为长度为 MAX_LOG_LINES 的字符串数组，因此当日志行数超过 MAX_LOG_LINES 会进行滚动刷新。

流程控制层通过游戏逻辑层获取本回合的行棋信息后，调用行棋日志层进行日志生成和记录。日志生成的过程根据当前回合行棋信息中的原坐标、新坐标计算棋子移动类型和移动距离。其中，移动类型使用中国象棋中的专用术语表示，包括 MOVE_ADVANCE（进）、MOVE_RETREAT（退）和 MOVE_HORIZONTAL（平）三种，并与棋子移动距离组合后作为一条新日志保存。这种专用术语的表示方法，使得象棋游戏能尽可能标准地对中国象棋的对局模式进行复刻，从而优化玩家的游戏体验。

同理，如果当前回合为对方回合，则根据对方通过远程蓝牙传输得到的行棋信息进行日志生成和记录，实现了网络化数据传输与本地化数据处理的分离，在一定程度上减轻了系统的网络依赖，提高了系统的稳定性。

5. 蓝牙传输层

蓝牙传输层负责游戏的远程通信，即发送行棋信息到对方设备和从对方设备接收行棋信息，对应根目录下的 bluetooth.c 文件。流程控制层对蓝牙传输层的初始化过程完成了蓝牙硬件和蓝牙协议的初始化。

流程控制层中，如果当前回合为自身回合，则通过游戏逻辑层获取棋子移动

信息，并通过蓝牙传输层将行棋信息发送到对方设备；如果当前回合为对方回合，则持续等待并监听对方设备通过远程蓝牙发送的行棋信息，以此作为游戏逻辑层的执行结果来进行后续行棋日志层和 UI 层等处理流程。

蓝牙传输中使用了序列化和反序列化技术进行数据交换。发送数据时，将行棋信息字符串转换为对应的二进制形式；接收数据时，将二进制形式的行棋信息转换回相应的字符串，并使用 C 语言的字符串函数从中提取出关键信息（如棋盘坐标），从而完成了一次完整的数据传输过程。

6. UI 层

UI 层负责根据本轮棋子移动信息和日志内容进行屏幕的重绘，对应根目录下的 ui.c 文件。流程控制层对 UI 层初始化时，UI 层完成了游戏画面的初次渲染，即“游戏菜单”界面的绘制，包括“开始游戏”和“退出游戏”两个按钮。

当点击“开始游戏”按钮后，流程控制层将当前屏幕状态由 SCREEN_MENU 切换为 SCREEN_GAME，从而使 UI 层在其后完成游戏对局界面的绘制，并根据游戏结构体 Game 中的棋盘格局和日志结构体 GameLog 分别完成棋盘对应位置处棋子的绘制和右侧日志内容的绘制。

游戏对局界面上包括“重新开始”和“退出游戏”两个按钮，点击后流程控制层分别处理对应的点击事件。值得注意的是，UI 层始终根据当前游戏状态而不是游戏事件进行绘制，即“状态驱动”而不是“事件驱动”，从而实现了 UI 层和游戏主体逻辑的解耦。

在流程控制层，每隔 16ms（即帧率为 60FPS）进行一次屏幕重绘，使 UI 层根据当前游戏状态完成屏幕内容的绘制。设置 16ms 延迟的作用是防止 UI 层频繁屏幕重绘消耗大量的硬件资源，确保屏幕适时刷新的同时提高了程序的流畅程度，且象棋游戏的特点（不需要灵敏的实时响应）也保证了这一设定的合理性。

6.3 实验结果

使用两台设备依次进行蓝牙配对和蓝牙串行通信后，分别执行以下的命令。其中第一个参数 x 表示设备自身的多点触摸协议设备号，第二个参数表示 player_id，使得两台设备分别作为红方和黑方进行游戏：

```
1. ./lab_final x 0
```

```
1. ./lab_final x 1
```

分别执行上述命令后，观察到两台设备的屏幕上均正确出现了“游戏菜单”界面，如图 6. 3 所示：

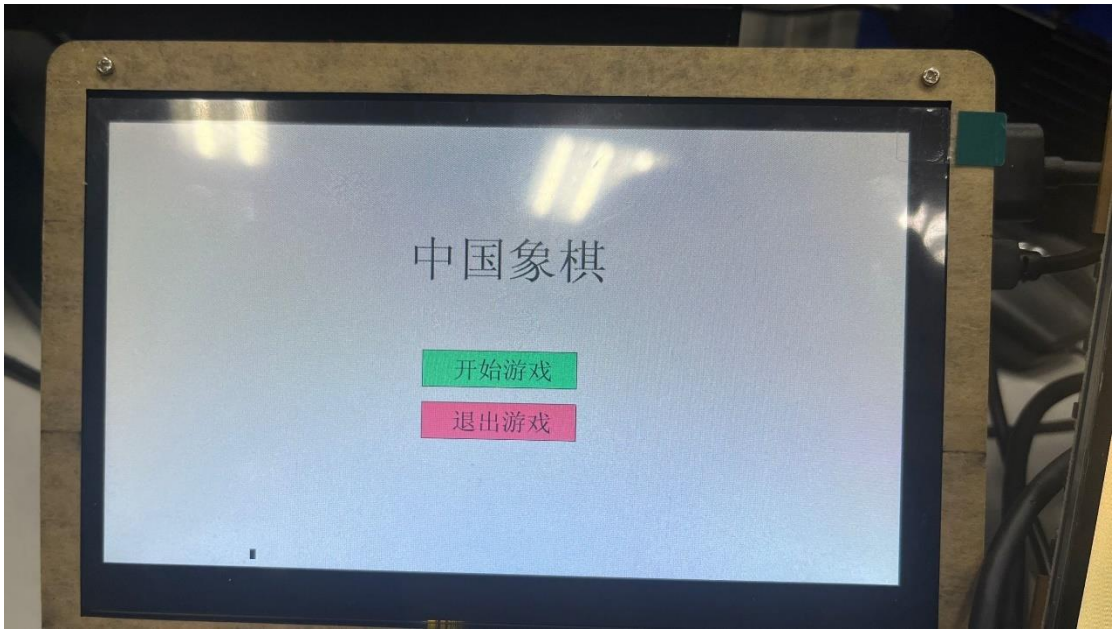

图 6.3 游戏菜单界面

接下来，分别点击“开始游戏”按钮，两台设备进入游戏对局界面，屏幕左侧显示完整棋盘和相应位置处的双方棋子，右侧显示日志内容。首轮由红方设备负责行棋，此时另一方设备点击棋盘将无反应。红方设备选择棋子后，观察到相应棋子周围出现蓝色提示框，如图 6.4 所示：

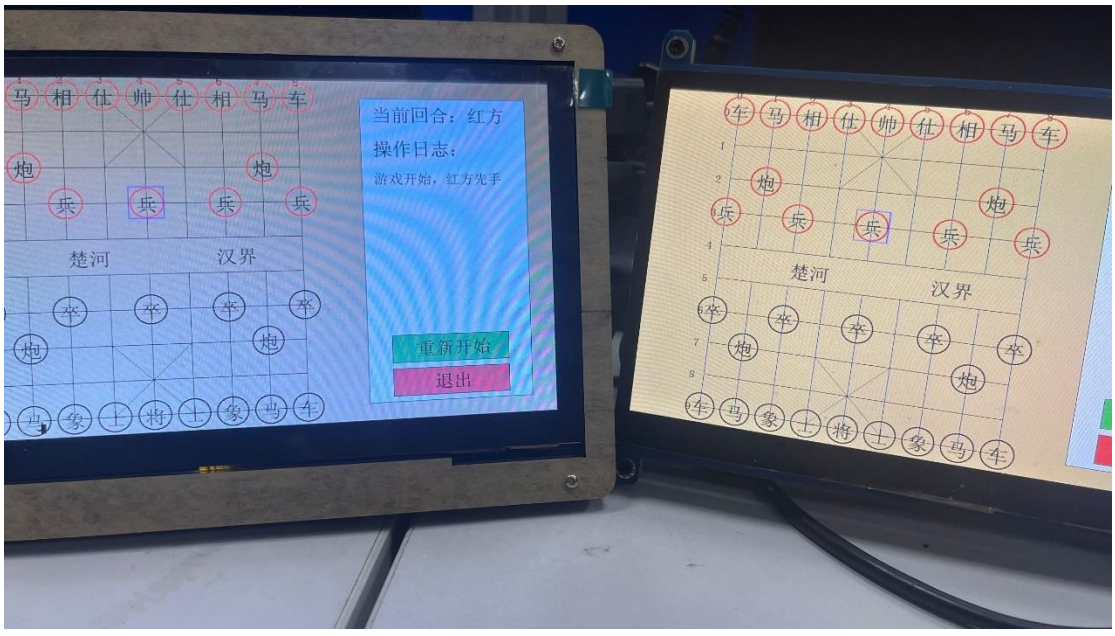

图 6.4 棋盘及选择棋子界面

选择棋子后，再次点击该棋子或点击棋盘格选定的棋子移动路径不合法，则取消棋子选择状态。如果点击棋盘格选定的棋子移动路径合法，则两台设备的屏幕上均会同步更新该棋子的位置，且行棋日志区将显示对应的内容（本轮红方行棋信息为“兵 5 四进一”），如图 6.5 所示：

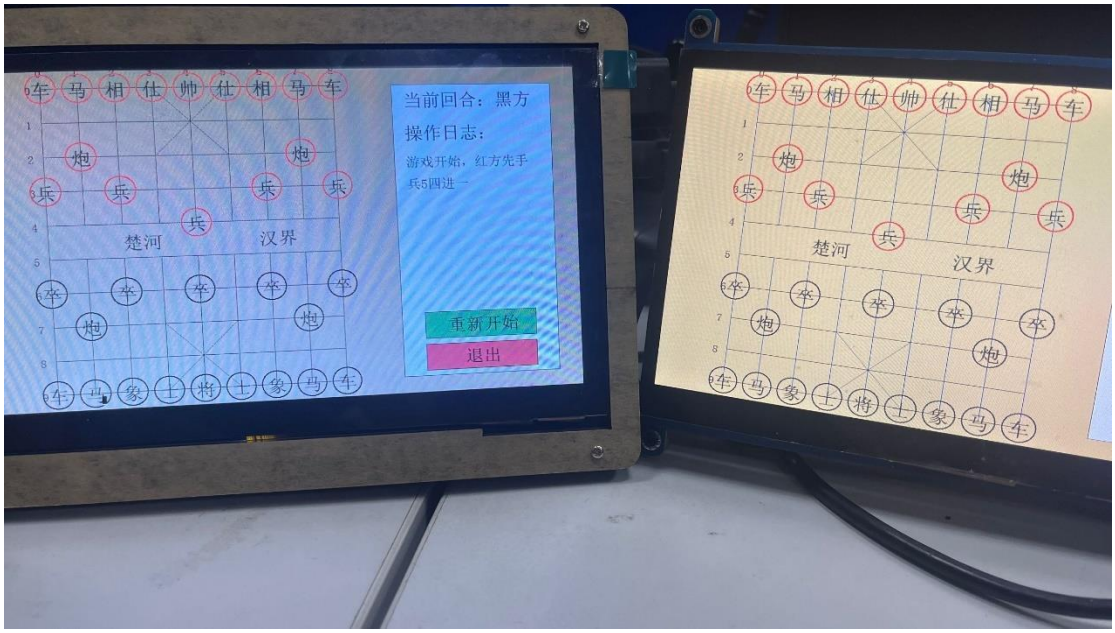

图 6.5 棋子移动界面

同理，黑方继续行棋，行棋后两台设备的屏幕上均会同步更新该棋子的位置，且行棋日志区将显示对应的内容（本轮黑方行棋信息为“卒 5 四退一”），如图 6.6 所示：

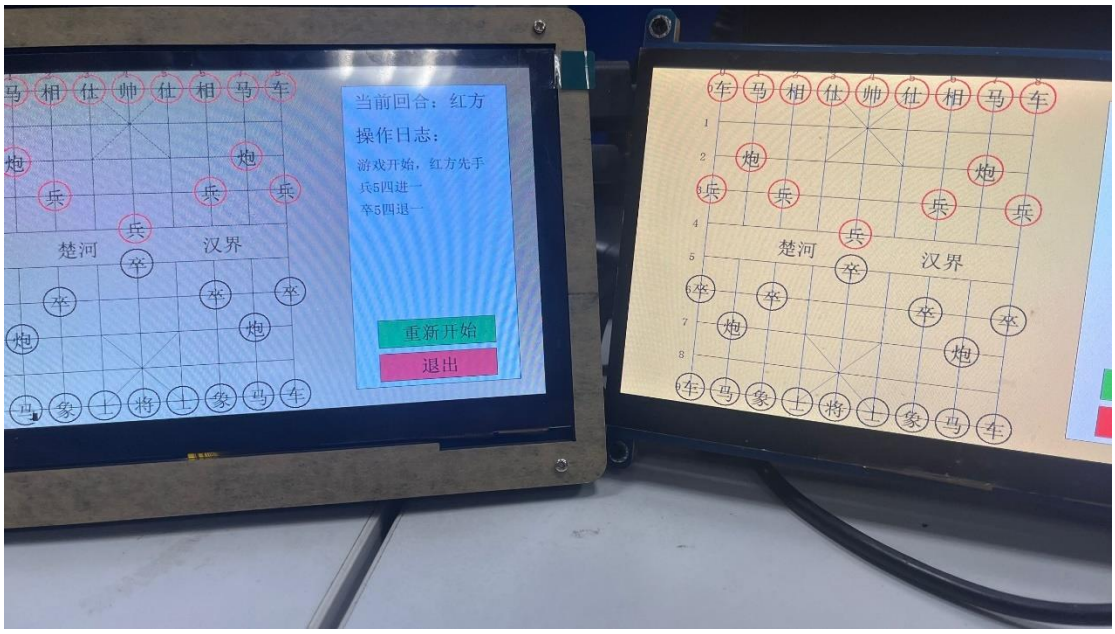

图 6.6 棋子移动界面

当发生“吃子”事件时，被吃棋子会被当前移动的棋子覆盖，两台设备的屏幕上均会同步更新棋子的位置，且行棋日志区将显示对应的内容（本轮红方行棋信息为“兵 5 五进一”），如图 6.7 所示：

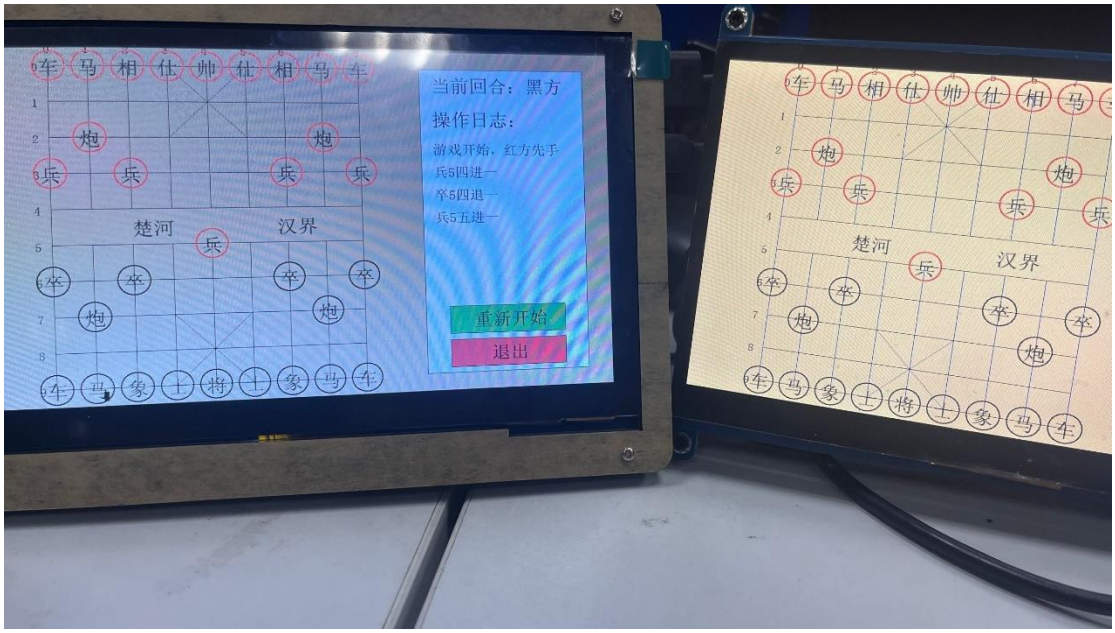

图 6.7 吃子界面

最后，当一方吃掉对方的将/帅后，游戏结束，双方的屏幕上同时显示本次游戏的获胜方，并提供“重新开始”和“退出游戏”选项，如图 6.8 所示：

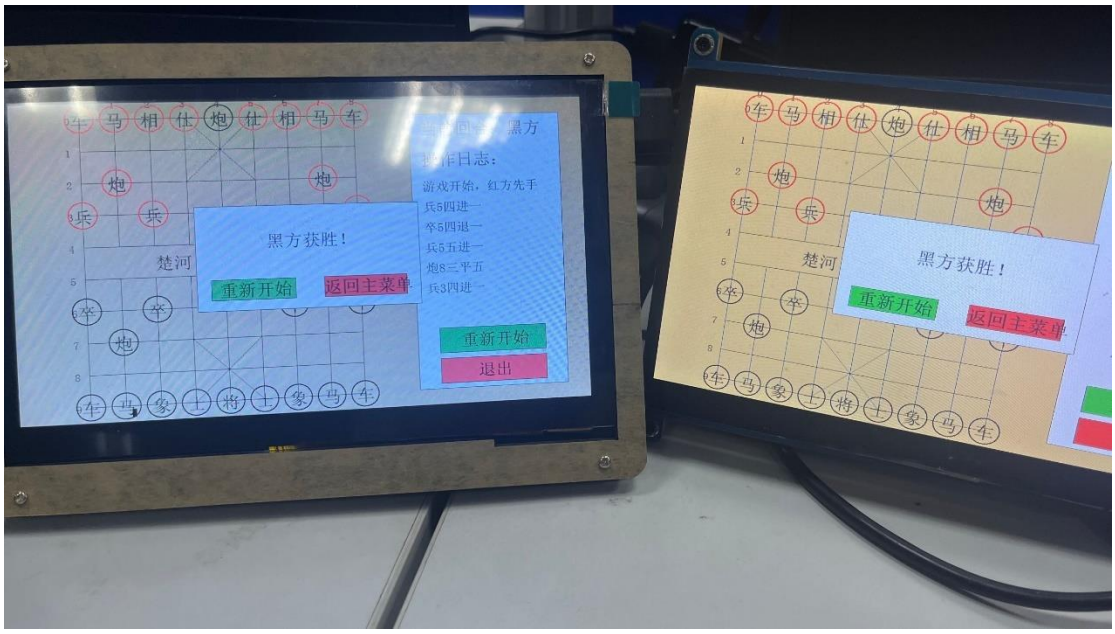

图 6.8 游戏结束界面

以上实验结果展示了象棋游戏运行的全部流程，表明所设计与实现的“双人蓝牙象棋对战游戏”十分标准地对中国象棋的游戏过程进行了复刻，本次综合实验成功。

7 实验总结与建议

7.1 实验总结

通过本学期的六次嵌入式系统实验，我系统地学习了嵌入式系统开发的基础知识和实战技能。从最基础的系统烧录、图形界面开发，到图片文字显示、多点触摸交互，再到蓝牙通信，最后完成综合性的双人蓝牙象棋对战游戏，实验内容由浅入深、循序渐进，使我对嵌入式系统有了更深入的理解。

在基础实验中，我掌握了香橙派开发板的基本使用方法，包括系统烧录、远程登录等操作。通过图形界面实验，我深入理解了 Linux 下 framebuffer 的工作原理和双缓冲机制，掌握了基本图形的绘制方法。在图片文字显示实验中，我学会了处理不同类型图片的显示技术，包括对透明度的处理。多点触摸实验则让我理解了触摸屏驱动的工作原理和事件处理机制。蓝牙通信实验使我掌握了设备间无线通信的基本方法。

在综合实验中，我设计并实现了一个完整的双人蓝牙象棋对战游戏。这个项目综合运用了前面所学的各项技术，采用模块化的系统架构，实现了流程控制、游戏逻辑、棋子逻辑、行棋日志、蓝牙传输、UI 显示等多个功能模块。通过这个综合项目的开发，我不仅巩固了各项技术要点，更重要的是学会了如何进行系统设计、模块划分，以及如何处理多个功能模块之间的协作。

7.2 实验建议

本次实验课程总体上收获颇丰，体会到了软硬件结合的乐趣。建议在实验内容中增加更多底层驱动开发的内容。目前的实验主要基于 Linux 系统提供的驱动框架，如 framebuffer、input event 等进行应用开发。如果能增加一些设备驱动的开发实验，让学生了解驱动程序的工作原理、中断处理机制、设备树配置等内容，将有助于加深对嵌入式系统底层工作机制的理解。

此外，建议加入一些实时操作系统的相关实验内容。通过让学生实现简单的任务调度、中断响应、资源管理等功能，加深对嵌入式系统中实时性要求、任务优先级管理等核心概念的理解。这些内容将帮助学生更好地理解嵌入式系统的特点和设计思想。

最后，建议增加一些硬件资源管理的实验内容。比如内存管理、电源管理、外设管理等方面的实验，让学生了解如何在资源受限的嵌入式环境下进行系统设计和优化。这将有助于培养学生在实际嵌入式系统开发中的系统思维和优化意识。