触摸屏移植综合实验报告

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。本次实验是实现触摸屏的移植与应用，故最重要的硬件资源是开发板提供的 LCD 触摸屏。本次实验用到的硬件资源如下：

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

- tslib 触摸屏库 1.4 版本源代码包
- 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
- 针对 ARM 的交叉编译工具链 2016.08 版本，并加入环境变量中[3]

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 Linux 对触摸屏的支持

触摸屏是现在嵌入式系统中常常采取的一种人机交互方式，具有直观、简单、方便的特点，手机由键盘机发展到触屏机的过程就可以证明这一点。常用的触摸屏一般有电阻屏和电容屏：电阻屏是在触摸点被按压时，上下两层导电层接触，在 x 和 y 方向形成电阻分压，通过 A/D 传感器采样电压得出按压点坐标，因而一般只支持单点触控；电容屏是利用人体电容和触摸屏触摸板形成耦合使电流变化，通过 A/D 采样四个角的电流比例得到触摸点的位置，可以支持多点触控，是主流智能手机的标配。

Linux 内核包含多种触摸屏设备驱动，可以为多种触摸屏提供支持。与触摸屏相关的内核配置选项通过以下方式调用：`make menuconfig`: `Device Drivers-->Input device support`

1. `Event interface`：在实验板上，配置此内核选项可以通过 `/dev/input/event2` 设备存取输入设备的事件，使用 tslib 库时要配置。
2. `Touchscreens`：选择触摸屏驱动。对于 x210 开发板，需要配置对应的触摸屏驱动项 `x210 touchscreen driver`。

Linux 操作系统内核一般完成的是 A/D 转换部分，而触摸屏库则基于内核之上提供一组方便用户调用的接口。

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

2.2 触摸屏库的分析

触摸屏库的原理分析也是基于对其源码的分析。由于其源码规模不小，为节约本篇报告篇幅起见，只根据源码分析结果对其主要代码结构做一定总结介绍。

tslib 库实现类似面向对象的抽象机制方式，是通过 `tsdev` 结构体来代表被操作的触摸屏对象，该对象可以通过 tslib 库的提供的一组接口进行操作，而接口的实现封存在各个模块中，其操作方式与 frambuffer 的机制类似。

src 目录下包括主要的头文件和接口函数的实现：

- `ts_open.c`：独立于模块而存在，负责打开 eventX 设备文件并将文件标识符写入触摸屏对象中。
- `ts_load_module.c`：tslib 库的基础是通过在触摸屏对象中加载不同的模块实现对触摸屏设备的读取，数据处理等，该文件内函数的作用是静态或动态地加载模块库并根据参数将其与触摸屏对象关联。
- `ts_read.c` 等:实现根据触摸屏对象的属性调用具体模块操作的接口函数,如 `ts_read()`，`ts_close()` 等。
- `tslib.h`: 声明 tslib 公开的函数，供用户程序调用其中的函数来实现对触摸屏的操作，并定义了触摸屏单采样点、多采样点的格式 `struct ts_sample`，`struct ts_sample_mt`。
- `tslib-private.h`：声明 tslib 不公开的函数，用于加载模块和报错功能。同时还包括触摸屏对象格式的定义 `struct tsdev`。
- `tslib-filter.h`：定义 tslib 模块的属性 `struct tslib_module_info`，包括触摸屏对象的指针（指向调用该模块的触摸屏对象），模块链表（用于连接对触摸屏对象进行下一步处理的模块），以及该模块对触摸屏对象操作方法的定义。

plugins 目录下是各模块的实现,此处主要分析读取 eventX 设备使用的模块 `input-raw.c`:

- `tslib_input` 结构体，定义了读取设备文件时用到的一些属性并储存一部分读取到的信息。此处程序编程采取了一个简单的技巧：`tslib_input` 结构体的第一个成员是 `tslib_module_info` 结构体，故指向该成员的内存指针也是指向父结构体的指针，通过转换指针类型可以实现对父结构体和子结构体内容的灵活访问，利用聚合的方法实现了类似于 C++ 中的继承机制。
- `ts_input_read()` 和 `ts_input_read_mt()` 函数是对上层接口 `ts_read()`，`ts_read_mt()` 的实现，通过读取 eventX 设备得到触摸屏的触摸状态，触摸点等信息。
- 模块中还包含模块初始化函数的实现，检查文件标识符，字符串格式化函数等功能。

2.3 测试程序的分析

以下直接对源代码 `tests` 目录下 `ts_test.c` 程序中涉及的主要代码进行分析，部分调用到的函数将在注释中直接说明功能。

```
/* 由font*.c提供文字点显示映射，fbutils.c/h提供了绘图相关操作函数 */
/* 由 testutils .c/h提供按钮相关功能 */
/* 定义用到的颜色和触摸按钮 */
static int palette [] ={
0x000000, 0xffe080, 0 xffffff , 0xe0c0a0, 0x304050, 0x80b8c0};
#define NR_COLORS (sizeof (palette) / sizeof (palette [0]))
#define NR_BUTTONS 3
static struct ts_button buttons [NR_BUTTONS];

/* 每次切换模式，刷新屏幕时先以全黑填充屏幕再重绘提示信息和按钮 */
static void refresh_screen (){
    int i;
     fillrect (0, 0, xres - 1, yres - 1, 0);
    put_string_center(xres/2, yres/4, "TSLIB test program", 1);
    put_string_center(xres/2, yres/4+20,"Touch screen to move crosshair", 2);
    for (i = 0; i < NR_BUTTONS; i++) button_draw (&buttons [i]);
}
int main(){
    /* 读取TSLIB_TSDEVICE环境变量作为被使用的触摸屏设备，如无则用指定设备 */
    /* 以阻塞方式打开该该设备 */
    if( (tsdevice = getenv("TSLIB_TSDEVICE")) != NULL ) {
        ts = ts_open(tsdevice,0);
    } else {
        if (!( ts = ts_open("/dev/input/event0", 0)))
            ts = ts_open("/dev/touchscreen/ucb1x00", 0);
    }
    /* 打开触摸屏设备，配置文件，frambuffer出错则报错退出 */
    if (!ts) {
        perror("ts_open");
        exit (1);
    }
    if (ts_config(ts)) {
        perror("ts_config");
        exit (1);
    }
    if (open_framebuffer()) {
        close_framebuffer();
        exit (1);
    }
    x = xres/2;
```

```
y = yres/2;
/* 填写颜色表 */
for (i = 0; i < NR_COLORS; i++) setcolor (i, palette [i]);
/* 对触摸按键的属性进行初始化，w为宽度，h为高度，x为最左点横向坐标， *
 * y为最上点纵向坐标，text为文字内容 */
memset (&buttons, 0, sizeof (buttons));
buttons [0].w = buttons [1].w = buttons [2].w = xres / 4;
buttons [0].h = buttons [1].h = buttons [2].h = 20;
buttons [0].x = 0;
buttons [1].x = (3 * xres) / 8;
buttons [2].x = (3 * xres) / 4;
buttons [0].y = buttons [1].y = buttons [2].y = 10;
buttons [0].text = "Drag";
buttons [1].text = "Draw";
buttons [2].text = "Quit";
refresh_screen();
/* 在没有按下退出之前重复处理触摸屏的触摸事件 */
while (1) {
    struct ts_sample samp;
    int ret;
    /* 拖动光标模式下的操作逻辑：在上次循环结束采样的位置画光标 */
    if ((mode & 15) != 1) put_cross(x, y, 2 | XORMODE);
    /* 每次循环读一次触摸数据，直到完全读完event设备已有触摸事件 */
    /* 此处read为阻塞方式，即没有触摸数据时会停在该语句处等待触摸 */
    ret = ts_read(ts, &samp, 1);
    /* 读取到新的触摸后以异或模式画旧光标，相当于擦除 */
    if ((mode & 15) != 1) put_cross(x, y, 2 | XORMODE);
    /* read出错则退出程序 */
    if (ret < 0) {
        perror("ts_read");
        close_framebuffer();
        exit(1);
    }
    /* 读到不止一个点则抛弃 */
    if (ret != 1) continue;
    /* 检测各个按钮是否被按下，如果被按下则切换到对应模式 */
    for (i = 0; i < NR_BUTTONS; i++)
        if (button_handle(&buttons [i], samp.x, samp.y, samp.pressure))
            switch (i) {
            case 0:
```

```
                mode = 0;
                refresh_screen(); break;
            case 1:
                mode = 1;
                refresh_screen(); break;
            case 2:
                quit_pressed = 1;
            }
        /* 在终端上打印采样到的触摸信息 */
        printf("%ld.%06ld:␣%6d␣%6d␣%6d\n", samp.tv.tv_sec, samp.tv.tv_usec,
            samp.x, samp.y, samp.pressure);
        /* 判断触摸是否已经抬起 */
        if (samp.pressure > 0) {
            /* 还在触摸且在画线模式下则从上个触摸位置向新位置画线 */
            if (mode == 0x80000001) line (x, y, samp.x, samp.y, 2);
            /* 还在触摸且在拖拽模式下则更新光标位置 */
            x = samp.x; y = samp.y;
            mode |= 0x80000000;
        } else mode &= ~0x80000000;
        if (quit_pressed) break;
    }
    close_framebuffer();
    return 0;
}
```

3 实验过程与内容

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

3.1 触摸屏库的移植与测试

本次实验使用的是触摸屏库 tslib 1.4 版本。对于触摸屏库的移植，采取和应用程序类似的方法：利用交叉编译器在主机上交叉编译源代码。与其不同的是，触摸屏库内部代码中对不同的处理器环境做了不同的实现，故在执行安装操作之前需要进入程序解压目录执行一些预配置命令。

```
$ export CC=arm-none-linux-gnueabi-gcc
$ ./autogen.sh
$ ./configure --host=arm-linux --prefix=/home/dwq/tslib
$ make -j8
```

```
$ make install -j8
$ cp -r /home/dwq/tslib /srv/nfs4/nfslinux
```

执行以上操作后在 /home/dwq/tslib 目录下产生安装好的库和测试程序，目录结构符合 linux 根目录规范的要求，并拷贝到主机上作为目标机根文件系统的 nfs 文件系统中。

根据对触摸屏库的分析，要使触摸屏库正常运行，需要定义一些程序执行时需要读取的环境变量。为了方便起见，在存放实验文件的文件夹下创建如下脚本文件：

```
# ./tslibenv.sh
export TSLIB_TSDEVICE=/dev/input/event2
export TSLIB_CONFFILE=/etc/ts.conf
export TSLIB_PLUGINDIR=/lib/ts
export TSLIB_CONSOLEDEVICE=none
export TSLIB_FBDEVICE=/dev/fb0
export TSLIB_CALIBFILE=/etc/pointercal
```

再对触摸屏库的配置文件 /etc/ts.conf 进行修改，去掉以下几项的注释以调用 tslib 的一些模块：

```
# 使用 linux 的 event 设备作为输入设备的模块
module_raw input
# 过滤 AD 转换器的噪声
module variance delta=30
# 去除触点抖动的模块
module dejitter delta=100
# 线性坐标变换模块
module linear
```

修改好以上环境变量和配置文件后，尝试在目标机上执行 ts_test 测试文件，发现运行时缺少库文件：error while loading shared libraries: libdl.so.2: cannot open shared object file: No such file or directory。将交叉编译器动态链接库下的对应文件和其符号链接源文件都复制到目标机的 /lib 下后，以上错误不再出现，但程序执行时报段错误，错误函数为 fopen()。通过研究代码和几次试验发现，是打开校准文件 /etc/pointercal 时出错，原因是目标机由于采用 nfs 文件系统，无法创建、修改这个在主机上实际不存在的文件。

在主机上创建 pointercal 文件后，在目标机下执行 ts_test 程序，打开了触摸屏库的自带测试程序，触摸屏工作正常，可以通过三个按钮分别调用拖动光标、连续画线、退出功能，minicom 终端中输出的触摸信息也与实际情况符合，移植成功。

3.2 扩充自带测试程序功能

为标识与自带测试程序的不同，将源代码中的 ts_test.c，testutils.c/h，fbutils.c/h 分别复制为ts_main.c，newtest.c/h，fbconfig.c/h。为了扩充自带测试程序的功能，进而

利用 tslib 库提供的资源和库函数进行编程，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
PROGRAM = ts_main pop
OBJ = font_8x8.o font_8x16.o fbconfig.o newtest.o

default : $(PROGRAM)
pop : pop.o $(OBJ)
    $(CC) -o $@ $^ -L ./ libts.so
ts_main : ts_main.o  $(OBJ)
    $(CC) -o $@ $^ -L ./ libts.so
%.o : %.c
    $(CC) -c  $<

.PHONY : cp
cp :
    cp $(PROGRAM) /srv/nfs4/touchscreen
```

本次实验中对自带测试程序的扩充是移植用户图形界面实验中的 BMP 绘制函数：

```
/* fbconfig.c */
//测试程序提供的指向显存空间的指针
static unsigned char *fbuffer;
//临时借用的颜色板号，用于画点时映射原实验中颜色格式指定的颜色
#define COLORMAP_TMP 255
//原实验与测试文件中画点方式不同，故提供此函数作为给原BMP绘制函数画点的接口
void draw_point(int x, int y, color_8 color){
    unsigned colortmp;
    unsigned red;
    unsigned green;
    unsigned blue;
    union multiptr loc;     //loc.p8是一个(char *)型指针
    //超出虚拟分辨率范围内的点不画
    if ((x < 0) || ((__u32)x >= var.xres_virtual) ||
        (y < 0) || ((__u32)y >= var.yres_virtual)) return;
    //根据测试程序定义计算的画点位置指针
    loc.p8 = line_addr[y] + x * bytes_per_pixel;
    //将传入颜色由原实验中的格式转为以24位整数表示RGB来建立颜色板映射
    colortmp = ((unsigned)color.red << 16) | ((unsigned)color.green << 8)
     | ((unsigned)color.blue);
```

```
    setcolor(COLORMAP_TMP, colortmp);
    //在指定位置绘制临时颜色板对应颜色的点，与传入的颜色相同
    ___setpixel (loc, xormode, colormap [COLORMAP_TMP]);
}
/* fbconfig.h */
//原实验中的颜色格式定义
typedef struct{
    unsigned char blue;
    unsigned char green;
    unsigned char red;
    unsigned char alpha;
}color_8;
//原实验中对BMP文件格式的定义
#pragma pack(1)
typedef struct{...}BMP_head;
typedef struct{...}BMP_info;
#pragma pack()
//原实验中绘制BMP相关的函数，去掉传入的显存指针，函数中以测试程序提供的指针代替
void draw_point(int x, int y, color_8 color);
void make_color(color_8 *out, unsigned char red, unsigned char green,
                unsigned char blue, unsigned char alpha);
void draw_BMP(char path[], ___u32 xbias, ___u32 ybias);
void draw_BMP_24(FILE *fp, BMP_info *info, ___u32 xbias, ___u32 ybias);
void draw_BMP_32(FILE *fp, BMP_info *info, ___u32 xbias, ___u32 ybias);
}
/* ts_main.c */
//只涉及修改部分
#define NR_BUTTONS 4
buttons[0].text = "Drag";
buttons[1].text = "Draw";
buttons[2].text = "Bmp";
buttons[3].text = "Quit";
//主循环中加入画图功能
while(1)
{...
    if (mode == 0x00000002){
    //画方块版本
    // fillrect (oldx, oldy, oldx + 30, oldy + 30, 0);
    //put_string_center (xres/2, yres/4,"Touch screen to move crosshair", 2);
    //for (i = 0; i < NR_BUTTONS; i++)
```

```
    //button_draw (&buttons [i]);
    // fillrect  (samp.x, samp.y, samp.x + 30, samp.y + 30, 1);
    //oldx = samp.x;
    //oldy = samp.y;
        //画图版本
        refresh_screen();
        draw_BMP("1234.bmp", samp.x, samp.y);
    }
}
```

修改并编译以上结果后在目标机上执行测试程序 `ts_main`，对于画图和画方块版本，都实现了按下 `Bmp` 按钮后，以抬起触摸作为按下" 按键" 的信号，擦除旧屏幕内容并在触摸点绘制图片或指定大小方块的功能，证明该功能函数块结合触摸屏的移植成功。

3.3 连连看小游戏的实现

基于以上对自带测试程序功能的扩充，重新设计主程序后可以实现各种应用。连连看小游戏非常适合通过触屏方式进行交互，规则也相对简单，故选择实现一个简单的连连看小游戏作为基于触摸屏库开发的嵌入式应用：

```
/* pop.c */
#define X_BLOCK_NUMBER 6 //x轴图片块个数
#define Y_BLOCK_NUMBER 3 //y轴图片块个数
#define BLOCK_NUMBER (X_BLOCK_NUMBER * Y_BLOCK_NUMBER)

//以下部分定义程序的基本数据结构
//图片块类型Block，是游戏的基本元素
typedef struct ts_block{
    int x;                  //左上角x轴坐标
    int y;                  //左上角y轴坐标
    int imgid;              //显示的是第imgid种图案
    char *on_img_path; //选中状态时的图片样式
    char *off_img_path; //未选中时的图片样式
    char exist_flag;        //图片块是否还未被消标志位，图片块仍存在则为1
}Block;

//游戏全局地图类型Block_map，封装所有游戏数据
typedef struct ts_block_map{
    Block Block [BLOCK_NUMBER]; //所有的图片块
    int pressed_block;              //已选中的图片块编号
    int size;                       //已初始化的方块数
    int x_offset;                   //游戏区域左上角x轴坐标
```

```
    int y_offset;                    //游戏区域左上角y轴坐标
    int block_length;                //图片块长度
    int block_height;                //图片块高度
    int exist_blocks;                //还未被消掉的图片块数
    char reset_flag[BLOCK_NUMBER]; //用于标记某个方块是否还未被重排
}Block_map;

//获取图片i的路径字符串，0表示非选中状态，1为选中状态
void get_imgpath(char *string, int i, int state){
    if(state == 0) sprintf(string, "./bmp/smooth-%d.bmp", i);
    else if(state == 1) sprintf(string, "./bmp/smooth-%d.bmp", i + 43);
}
//根据对应图片初始化图片块
void register_block(Block_map *map, char *off_path, char *on_path){
    Block *block = (map->Block) + map->size;
    block->y = map->size / X_BLOCK_NUMBER * map->block_height +
                map->y_offset;
    block->x = map->size % X_BLOCK_NUMBER * map->block_length +
                map->x_offset;
    block->imgid = map->size / 2;
    block->on_img_path = on_path;
    block->off_img_path = off_path;
    block->exist_flag = 1;
    (map->size)++;
}
//重置地图状态，令所有图片块都可用
void reset_map(Block_map *map){
    int i;
    for(i = 0; i < BLOCK_NUMBER; i++) map->Block[i].exist_flag = 1;
    map->exist_blocks = BLOCK_NUMBER;
}
//重新绘制整个游戏区域的仍存在的图片块
void show_map(Block_map *map){
    int i;
    Block *block;
    for(i = 0; i < BLOCK_NUMBER; i++){
        block = (map->Block) + i;
        if(block->exist_flag)
            draw_BMP(block->off_img_path, block->x, block->y);
    }
```

```
}
//初始化地图所有的游戏数据和图片块
void init_map(Block_map *map){
    int i;
    Block *block = map->Block;
    char *string_on = NULL;
    char *string_off = NULL;
    map->pressed_block = -1;
    map->size = 0;
    map->x_offset = 120;
    map->y_offset = 70;
    map->block_length = 96;
    map->block_height = 132;
    map->exist_blocks = BLOCK_NUMBER;
    for(i = 0; i < BLOCK_NUMBER; i++){
        //为保存图片路径的字符数组分配储存空间
        string_off = (char *)malloc(25);
        string_on = (char *)malloc(25);
        map->reset_flag[i] = 0;
        get_imgpath(string_off, i/2, 0);
        get_imgpath(string_on, i/2, 1);
        register_block(map, string_off, string_on);
    }
}
//清空游戏全局地图，释放为储存字符串手动申请的内存空间
void exit_map(Block_map *map){
    int i;
    for(i = 0; i < BLOCK_NUMBER; i++){
        free(map->Block[i].on_img_path);
        free(map->Block[i].off_img_path);
    }
}
//返回点击的图片块序号，若图片块不存在或在范围外返回-1
int get_block_id(Block_map *map, int x, int y){
    int idy = (y - map->y_offset) / map->block_height;
    int idx = (x - map->x_offset) / map->block_length;
    int id = idx + idy * X_BLOCK_NUMBER;
    //检查点击是否在游戏区域内并判断图片块是否还存在
    if(y >= map->y_offset && idy < Y_BLOCK_NUMBER &&
       x >= map->x_offset && idx < X_BLOCK_NUMBER){
```

```
        if ((map->Block[id]).exist_flag) return id;
    }
    return -1;
}
//处理按下指定图片块的操作
void press_block(Block_map *map, int id){
    Block *block = (map->Block) + id;
    Block *pressed = (map->Block) + map->pressed_block;
    int x = block->x; int y = block->y;
    int px = pressed->x; int py = pressed->y;
    //按下已选中图片则取消选择
    if (map->pressed_block == id){
        fillrect (x, y, x + map->block_length, y + map->block_height, 0);
        draw_BMP(block->off_img_path, x, y);
        map->pressed_block = -1;
    }
    //没有已选中图片则选中图片
    else if(map->pressed_block == -1) {
        fillrect (x, y, x + map->block_length, y + map->block_height, 0);
        draw_BMP(block->on_img_path, x, y);
        map->pressed_block = id;
    }
    //按下图片与已选择图片不同则改为选择按下图片
    else if(block->imgid != pressed->imgid){
        fillrect (px, py, px + map->block_length, py + map->block_height, 0);
        draw_BMP(pressed->off_img_path, px, py);
        fillrect (x, y, x + map->block_length, y + map->block_height, 0);
        draw_BMP(block->on_img_path, x, y);
        map->pressed_block = id;
    }
    //按下图片与已选择图片相同则消去两图片块
    else{
        fillrect (px, py, px + map->block_length, py + map->block_height, 0);
        fillrect (x, y, x + map->block_length, y + map->block_height, 0);
        map->pressed_block = -1;
        block->exist_flag = 0;
        pressed->exist_flag = 0;
        map->exist_blocks -= 2;
    }
}
```

```
//随机打乱所有图片块
void change_map(Block_map *map){
    int i, random;
    Block blocktmp[BLOCK_NUMBER];
    //有已选择图片块则取消选择
    if(map->pressed_block >= 0) press_block(map, map->pressed_block);
    for(i = 0; i < BLOCK_NUMBER; i++){
        blocktmp[i] = map->Block[i];
        map->reset_flag[i] = 1;
    }
    //打乱算法采取旧地图随机抽出一个块作为新地图的第一块，依次重复
    //若旧地图随机抽出的块已被加入新地图，选择下一个块
    for(i = 0; i < BLOCK_NUMBER; i++){
        random = (rand()) % BLOCK_NUMBER;
        while(!(map->reset_flag[random])){
            random = (random + 1) % BLOCK_NUMBER;
        }
        map->Block[i] = blocktmp[random];
        map->Block[i].y = i / X_BLOCK_NUMBER * map->block_height +
                          map->y_offset;
        map->Block[i].x = i % X_BLOCK_NUMBER * map->block_length +
                          map->x_offset;
        map->reset_flag[random] = 0;
    }
}
//主函数，省略部分非关键且与前文重复代码
int main(){
    int x, y;
    unsigned int i;
    unsigned int mode = 0; //=1时为进入游戏模式
    int quit_pressed = 0;
    int newclick_flag = 1;  //消抖使用的标志位
    Block_map map;          //定义保存所有游戏数据的地图并
    srand(time(NULL));      //根据当前时间重取随机数种子
    ... //省略对触摸屏的操作
    //定义三个不同按钮：（重新）开始，重排图片块，退出
    buttons[0].text = "(Re)start";
    buttons[1].text = "Remap";
    buttons[2].text = "Quit";
    refresh_screen();
```

```
//屏幕上显示开始信息并初始化游戏数据
put_string_center (xres/2, yres/4,"Touch button to start", 2);
init_map(&map);
//进入读取触摸屏状态的循环
while (1) {
    ret = ts_read(ts, &samp, 1);...
    for (i = 0; i < NR_BUTTONS; i++)
        if (button_handle(&buttons [i], samp.x, samp.y, samp.pressure))
            switch (i + 1) {
            //点击（重新）开始的处理
            case 1:
                mode = 1;
                refresh_screen();
                reset_map(&map);
                change_map(&map);
                show_map(&map);break;
            //点击重排图片块的处理
            case 2:
                refresh_screen();
                change_map(&map);
                show_map(&map);break;
            //点击退出时销毁游戏数据
            case 3:
                quit_pressed = 1;
                exit_map(&map);
                break;
            }
    //消抖部分，取触摸屏幕到抬起之间所有采样数据的加权平均
    if (samp.pressure > 0) {
        if(newclick_flag) {
            x = samp.x;
            y = samp.y;
            newclick_flag = 0;
        }
        else{
            x = (samp.x + x) >> 1;
            y = (samp.y + y) >> 1;
        }
        mode |= 0x80000000;
    } else mode &= ~0x80000000;
```

```
        //手指离开屏幕后重置消抖信息并得出一个选中点
        if ((mode == 1) || (mode == 0))
        {
            newclick_flag = 1;
            printf("click=(%d,%d)\n", x, y);
            //仅在游戏模式下执行游戏逻辑
            if(mode == 1){
                int id = get_block_id(&map, x, y);
                if (id >= 0){
                    press_block(&map, id);
                    //若游戏结束在屏幕上显示消息并退出游戏模式
                    if(map.exist_blocks == 0){
                        put_string_center(xres/2, yres/4,"Congratulations!", 2);
                        mode = 0;
                    }
                }
            }
        }
        if (quit_pressed) break;
    }
    //省略退出后清空frambuffer等操作
}
```

利用与 3.2 节相同的方法可以完成对程序进行编译、拷贝到目标机上的工作。

至此一个规则为只要选中图片相同，就可以相消的连连看制作成功，具备触摸按键开始/重新开始游戏，重排图片顺序，退出程序的功能，并在目标机上已经通过多次测试可玩。具体游戏效果可以运行实验室的目标机 `/mnt/touchscreen/`文件夹内程序 `pop` 进行体验。

4 实验讨论与小结

4.1 部分实验问题讨论

* 配置环境变量出现的问题

在 3.1 节中曾经提到可以用脚本文件快速配置 tslib 运行所需的环境变量。但在实际执行时，直接在 minicom 终端中运行 `./tslibenv.sh` 会发现并未成功配置好其中的环境变量，多次尝试均是如此，若在终端中分别敲入脚本中的每一行语句并运行则可以正常配置。

解决方法是：在以上运行命令的前方加入 . （点 + 空格）后可以正常配置环境变量。

通过查询 linux 手册，得到了问题的解释：终端下执行一个脚本时，会先开启一个子终端环境，在该子终端环境内运行脚本内命令，那么脚本配置的环境变量就只会在子终端环境内起作用，而在运行脚本后的原终端中并未配置环境变量。在执行脚本的命令前加入的 . （点 + 空格）称为点命令，可以使脚本在当前终端中执行，因此可以正常配置配置好环境变量。

4.2 实验小结

本次实验对触摸屏库 tslib 的原理和实现进行了学习分析，完成了将触摸屏库移植到嵌入式系统上的工作并基于该库开发了用户程序——连连看游戏。通过这次实验，研究了触摸屏库的程序结构和实现，对 linux 软件的开发有了更深的了解；同时掌握了在实验主机上移植以触摸屏库为代表的应用程序，使之能在嵌入式系统上正常运行的方法。最后利用 C 语言编程知识和触摸屏库提供的接口实现了一个具有实用价值的应用程序，且这种程序在通过相应环境版本的交叉编译器编译后，可以在任何安装 tslib 触摸屏库的 linux 操作系统上运行而不需要考虑兼容性问题，是一次成功的跨硬件设备开发软件的经验。

由于时间限制，没有进一步开发完成触屏连连看游戏的限制规则，有一点遗憾。