最近学习了电容触摸屏的驱动及其上层工作原理，拿出来和大家分享！
转]Android 触摸屏校准程序的实现

一，校准的触摸算法如下：
触摸屏校准通用方法。
（XL, YL 是显示屏坐标，XT, YT 是触摸屏坐标，）
XL = XT*A+YT*B+C
YL = XT*D+YT*E+F
由于具体计算是希望是整数运算，所以实际中保存的 ABCDEF 为整数，而增加一个参数 Div
XL = (XT*A+YT*B+C) / Div
YL = (YT*D+YT*E+F) / Div
TSLIB 把以上的 7 个参数 ABCDEF Div 保存在 pointercal 文件中。
不校准的数据： A=1, B=0, C=0, D=0, E=1, F=0, Div=1
A B C D E F Div
-411 37818 -3636780 -51325 39 47065584 65536

二，Android 事件处理机制
android 事件的传入是从 EventHub 开始的，EventHub 是 事件的抽象结构，维护着系统设备的运行情况（设备文件放在/dev/input 里），设备类型包括 Keyboard、TouchScreen、TraceBall。它在系统启动的时候会通过 open_device 方法将系统提供的输入设备都增加到这个抽象结构中，并维护一个所有输入设备的文件描述符，如果输入设备是键盘的话还会读取 /system/usr/keylayout/目录下对应键盘设备的映射文件（修改./development/emulator/keymaps /qwerty.kl 来改变键值的映射关系），另外 getEvent 方法是对 EventHub 中的设备文件描述符使用 poll 操作等待驱动层事件的发生，如果发生的事件是键盘事件，则调用 Map 函数按照映射文件转换成相应的键值并将扫描码和键码返回给 KeyInputQueue.
frameworks/base/services/jni/com_android_server_KeyInputQueue.cpp
根据事件的类型以及事件值进行判断处理，从而确定这个事件对应的设备状态是否发生了改变并相应的改变对这个设备的描述结构 InputDevice。
Windowmanager 会创建一个线程（InputDispatcherThread），在这个线程里从事件队列中读取发生的事件 （QueuedEvent ev = mQueue.getEvent（））， 并根据读取到事件类型的不同分成三类（KEYBOARD、TOUCHSCREEN、TRACKBALL），分别进 行处理，例如键盘事件会调用 dispatchKey((KeyEvent)ev.event, 0, 0)以将事件通过 Binder 发送给具有焦点的窗口应用程序，然后调用 mQueue.recycleEvent(ev)继续等待键盘事件的发生;如果是触摸屏事件则调用 dispatchPointer(ev, (MotionEvent)ev.event, 0, 0)，这里会根据事件的种类（UP、DOWN、MOVE、OUT_SIDE 等）进行判断并处理，比如 Cancel 或将事件发送到具有权限的指定的窗口中 去;
移植方案
Android 本身并不带触摸屏校准。Android 获取到的数据就是驱动上报的原始数据。
方案一 ：移植 TSLIB，通过 TSLIB 产生 pointercal 校准参数文件。
方案二 ：从 Android 框架层获取 OnTouch 事件产生 pointercal 校准参数文件
方案一: 数据的校准在驱动中完成。 即把 pointercal 的参数数据通过某种方式(sysfs)传递给驱动程序进行校准。

方案二: 驱动上报原始点，原始点在框架层拦截后进行校验处理。
TSLIB 移植过程
修改源码以适应 android 的文件结构。
设定 Android.mk 编译选项，生成库即应用。
etc/ts.conf module_raw input
src/ts_config.c #define TS_CONF "/system/etc/ts.conf"
src/ts_load_module.c
char *plugin_directory="/system/lib/ts/plugins/";
tests/fbutils.c
char *defaultfbdevice = "/dev/graphics/fb0";
COPY ts.conf 到 /system/etc/ts.conf
init.rc. mkdir /data/etc/pointercal
通过 ts_calibrate 产生 pointercal 数据文件。
框架内获取参数文件制作 APK 应用，仿效 ts_calibrate 采点并计算出各参数，产生 pointercal 框架内实现触摸屏校准在 InputDevive.java 中 拦截触摸屏原始数据进行 pointercal 参数校验后再分发驱动内实现触摸屏校准
在 init.rc 中添加 event，在触摸屏加载后把 pointercal 参数输送给驱动。
结果-效果
实现细节:
扩展 init - proper_serivce 系统支持的属性权限，对自定义的特殊系统属性进行权限开放。
使用自定义系统属性在 init.rc 中 on property 事件中处理 pointercal 的读写权限。
使用自定义系统属性 触摸屏校准程序.apk 和 InputDevice.java 中的输入事件的同步。（在触摸屏校准期间 inputDevice 在输入事件中不能采用算法。 校准程序完成有 inputDevice 重新启用校准算法）
模拟器中至今无法进入 device.absX/Y != null 的代码， 需要了解以下 inputDevice 被调用的步骤。
三，触摸屏的时间流程:
驱动层:

```
/*
* Touchscreen absolute values
*
* These parameters are used to help the input layer discard out of
* range readings and reduce jitter etc.
*
* o min, max:- indicate the min and max values your touch screen returns
* o fuzz:- use a higher number to reduce jitter
*
* The default values correspond to Mainstone II in QVGA mode
*
* Please read
* Documentation/input/input-programming.txt for more details.
*/
```

```
static int abs_x[3] = {350, 3900, 5};
module_param_array(abs_x, int, NULL, 0);
MODULE_PARM_DESC(abs_x, "Touchscreen absolute X min, max, fuzz");

static int abs_y[3] = {320, 3750, 40};
module_param_array(abs_y, int, NULL, 0);
MODULE_PARM_DESC(abs_y, "Touchscreen absolute Y min, max, fuzz");

static int abs_p[3] = {0, 150, 4};
module_param_array(abs_p, int, NULL, 0);
MODULE_PARM_DESC(abs_p, "Touchscreen absolute Pressure min, max, fuzz");

/*
*  对设备进行初始化设置
*/
set_bit(EV_ABS, wm->input_dev->evbit);
set_bit(ABS_X, wm->input_dev->absbit);
set_bit(ABS_Y, wm->input_dev->absbit);
set_bit(ABS_PRESSURE, wm->input_dev->absbit);
input_set_abs_params(wm->input_dev, ABS_X, abs_x[0], abs_x[1],
abs_x[2], 0);
input_set_abs_params(wm->input_dev, ABS_Y, abs_y[0], abs_y[1],
abs_y[2], 0);
input_set_abs_params(wm->input_dev, ABS_PRESSURE, abs_p[0], abs_p[1],
abs_p[2], 0);

/*
* 事件发生时，提供原始点
*/
input_report_abs(wm->input_dev, ABS_X, data.x & 0xfff);
input_report_abs(wm->input_dev, ABS_Y, data.y & 0xfff);
input_report_abs(wm->input_dev, ABS_PRESSURE, data.p & 0xfff);

/*
* 提供给驱动外查询 input_dev 的接口
* struct input_absinfo info;
* ioctl(fd, EVIOCGABS(axis), &info)
* src file: evDev.c
*/
if ((_IOC_NR(cmd) & ~ABS_MAX) == _IOC_NR(EVIOCGABS(0))) {

t = _IOC_NR(cmd) & ABS_MAX;

abs.value = dev->abs[t];
```

```
abs.minimum = dev->absmin[t];
abs.maximum = dev->absmax[t];
abs.fuzz = dev->absfuzz[t];
abs.flat = dev->absflat[t];
```

Android 底层驱动

EventHub.cpp

```
static const char *device_path = "/dev/input";

openPlatformInput(void)
scan_dir(device_path);
open_device(devname);
fd = open(deviceName, O_RDWR);

/*
* 对外接口，getEvent，
* inotify 监控 device_path 目录， 使用 poll 机制轮询 inotify 和各个输入设备的可用状态。
解析事件或输入信息，放入各个传出参数中。
*/
bool EventHub::getEvent(int32_t* outDeviceId, int32_t* outType,
int32_t* outScancode, int32_t* outKeycode, uint32_t *outFlags,
int32_t* outValue, nsecs_t* outWhen)
```

JNI 部分： com_android_server_KeyInputQueue.cpp. 提供接口

```
static JNINativeMethod gInputMethods[] = {
/* name, signature, funcPtr */
{ "readEvent", "(Landroid/view/RawInputEvent;)Z",
(void*) android_server_KeyInputQueue_readEvent },
{ "getDeviceClasses", "(I)I",
(void*) android_server_KeyInputQueue_getDeviceClasses },
{ "getDeviceName", "(I)Ljava/lang/String;",
(void*) android_server_KeyInputQueue_getDeviceName },
{ "getAbsoluteInfo", "(IILcom/android/server/InputDevice$AbsoluteInfo;)Z",
(void*) android_server_KeyInputQueue_getAbsoluteInfo },
{ "getSwitchState", "(I)I",
(void*) android_server_KeyInputQueue_getSwitchState },
{ "getSwitchState", "(II)I",
(void*) android_server_KeyInputQueue_getSwitchStateDevice },
{ "getScancodeState", "(I)I",
```

```
(void*) android_server_KeyInputQueue_getScancodeState },
{ "getScancodeState", "(II)I",
(void*) android_server_KeyInputQueue_getScancodeStateDevice },
{ "getKeycodeState", "(I)I",
(void*) android_server_KeyInputQueue_getKeycodeState },
{ "getKeycodeState", "(II)I",
(void*) android_server_KeyInputQueue_getKeycodeStateDevice },
{ "hasKeys", "([I[Z)Z",
(void*) android_server_KeyInputQueue_hasKeys },
};
```

java service 部分： KeyInputQueue.java. 循环查询输入设备信息或目录状态并处理

```
Thread mThread = new Thread("InputDeviceReader") {
public void run() {
android.os.Process.setThreadPriority(
android.os.Process.THREAD_PRIORITY_URGENT_DISPLAY);

try {
RawInputEvent ev = new RawInputEvent();
while (true) {
InputDevice di;

// block, doesn't release the monitor

readEvent(ev);

boolean send = false;
boolean configChanged = false;

.....
```

//检测到新设备后

```
if (ev.type == RawInputEvent.EV_DEVICE_ADDED) {
synchronized (mFirst) {
di = newInputDevice(ev.deviceId);
mDevices.put(ev.deviceId, di);
configChanged = true;
}
}
```

```
//对触摸屏设备
InputDevice.AbsoluteInfo absX;
InputDevice.AbsoluteInfo absY;
InputDevice.AbsoluteInfo absPressure;
InputDevice.AbsoluteInfo absSize;
if ((classes&RawInputEvent.CLASS_TOUCHSCREEN) != 0) {
absX = loadAbsoluteInfo(deviceId, RawInputEvent.ABS_X, "X");
absY = loadAbsoluteInfo(deviceId, RawInputEvent.ABS_Y, "Y");
absPressure = loadAbsoluteInfo(deviceId, RawInputEvent.ABS_PRESSURE, "Pressure");
absSize = loadAbsoluteInfo(deviceId, RawInputEvent.ABS_TOOL_WIDTH, "Size");
} else {
absX = null;
absY = null;
absPressure = null;
absSize = null;
}

return new InputDevice(deviceId, classes, name, absX, absY, absPressure, absSize);
```

我们对触摸屏的数据修订是在 InputDevice.java 中基于 absX, absY, absPressure != null 的状态下的，当绝对原始点数据从驱动报上来之后，传递到 InputDevice.java 经过我们的校准后再 dispatch 出去到 windowManager -> activity 。 这样就是起到了校准效果。

四，需要注意的补助说明

EventHub 中有使用 IOCTL 对触摸屏的 EVIOCGABS(axis)进行了采样，取出内容 struct

```
input_absinfo info;
struct input_absinfo {
__s32 value;
__s32 minimum;
__s32 maximum;
__s32 fuzz;
__s32 flat;
__s32 resolution;
};

#define EVIOCGABS(abs)        _IOR('E', 0x40 + abs, struct input_absinfo)        /* get abs value/limits */
```

取出的是触摸屏的最大值，最小值等，这些值在 com_android_server_KeyInputQueue.cpp 中会把这些值传递给 InputDevice.cpp 中使用，即把报上来的位置通过算法进行计算成绝对坐标值

```
scaledX = ((scaledX-device.absX.minValue)
/ device.absX.range) * w;
```

当然如果出现了 X，Y 轴相反或者坐标反向等问题都可通过改写这条语句来进行实际操作。