
网名“鱼树”的学员聂龙浩,

学习“韦东山 Linux 视频第 2 期”时所写的笔记很详细,供大家参考。

也许有错漏,请自行分辨。

目录

触摸屏框架、原理和硬件操作步骤:	2
代码步骤:	2
一, "s3c2410_ts.c"是三星公司提供的触摸屏驱动。	4
简单分析此代码:	4
二, 触摸屏驱动框架:	5
1.习惯先写入口函	6
三, 触摸屏硬件操作:	9
四, 触摸屏使用过程: 按此思路写硬件驱动程序。	11
触摸屏驱动编写	12
查看电路图	12
一, 先看内核中自带的触摸屏驱动程序做的事件:	13
1, 使能时钟:CLKCON[15] 设置为"1"。	13
二, 查 2440 手册上硬件相关的说明。	16
A,2440 中有一个 10bit 的 CMOS ADC 模数转换器, 它最大工作频率是 2.5MHz。	16
B,Touch Screen Interface Mode 触摸屏接口模式:	17
三, 操作寄存器:	18
0, 使能时钟:	18
1, 操作寄存器之前, 要 ioremap ():	18
触摸屏使用过程:	19
四, 测试: 触摸笔按下、松开。	27
1. make menuconfig 去掉原来的触摸屏驱动程序	27
2. insmod s3c_ts.ko	29
按下/松开触摸笔	29
最后编译和测试。	46
使用:	46
安装:	47
使用: 安装驱动。	47
2. 触摸屏的程序是打开的文件	49

触摸屏框架、原理和硬件操作步骤：

触摸屏也是用“输入子系统”来做。

字符驱动程序编写：

APP: open, read, write. ----> 对应提供驱动程序的读写等函数。

驱动: drv_open, drv_read, drv_write

硬件

代码步骤:

1,确定主设备号:

可以自己确定，也可让内核分配。

2, 要构造驱动中的“open,read,write 等”

是将它们放在一个“file_operations”结构体中。

File_operations==> .open, .read, .write, .poll 等。

这里“open”函数会去配置硬件的相关引脚等，还有注册中断。

3, register_chrdev 注册字符设备:

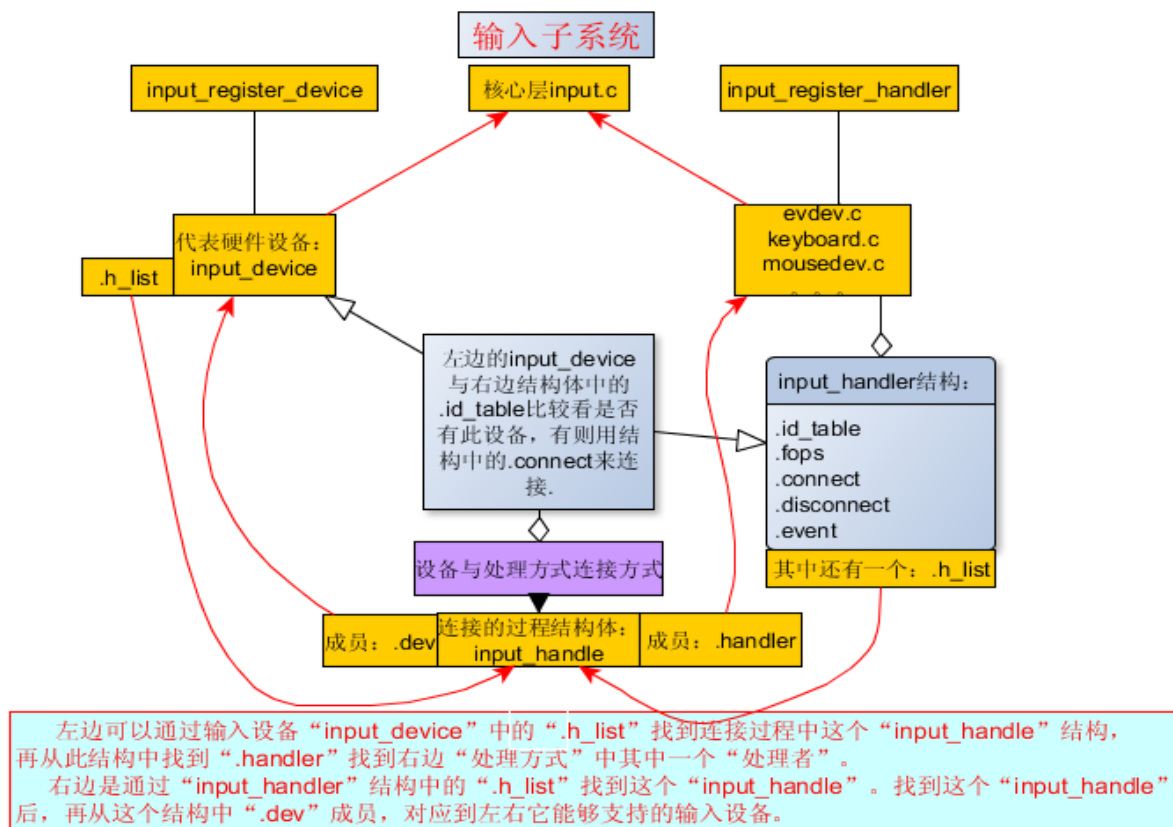
使用这个 file_operations 结构体。

Register_chrdev(主设备号, 设备号, file_operations 结构)。

4, 入口函数:

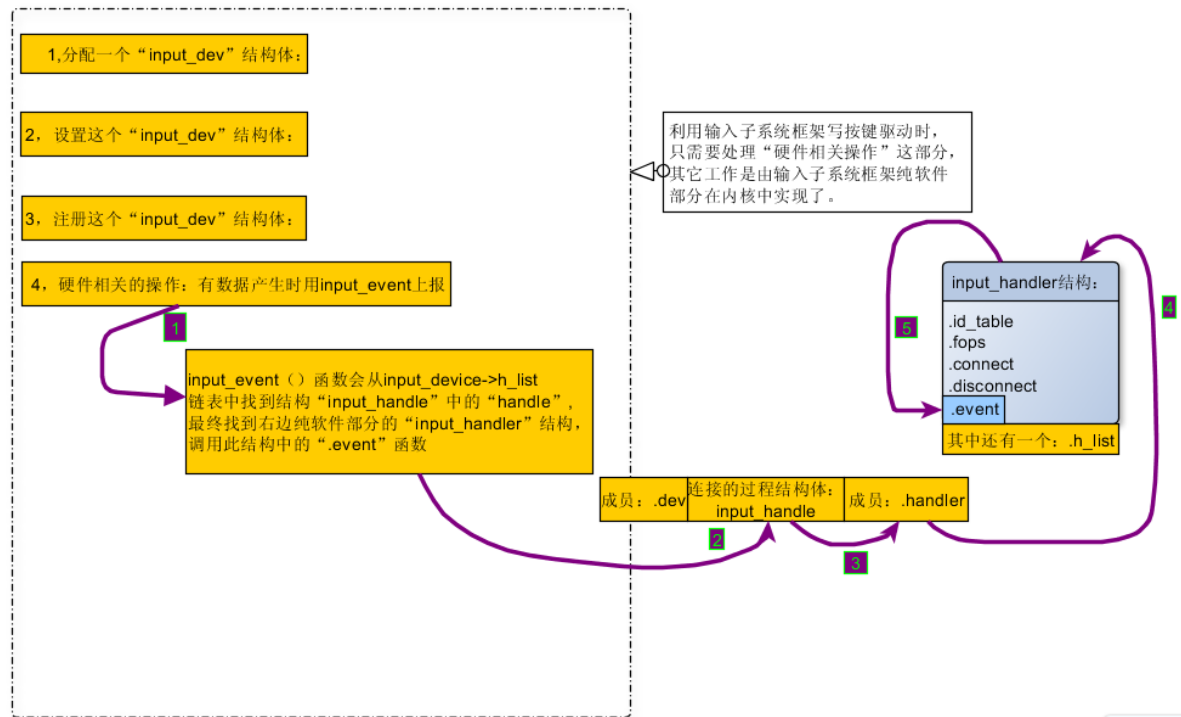
调用这个“register_chrdev()”函数。

5, 出口函数:



输入子系统肯定要提供读写等函数（如要读按键等信息），但发现输入子系统中 File_operations 结构体中，只发现一个“open”函数。这时就去看“input.c”中的“open”函数时，会发现它依赖其他文件如“evdev.c, keyboard.c”等文件注册的“input_handler”结构（这些结构中提供了 read, write 等函数）。

右边是纯软件的部分，左边是硬件部分，这个硬件部分分为如下步骤：



触摸屏也是用上面这一套框架来操作的。右边需要一个“evdev.c”文件。左边要分配一个“input_dev”结构。接着就看上图的硬件设备左边的过程:

分配一个“input_dev”结构体 --> 设置这个“input_dev”结构体 --> 注册这个“input_dev”结构体 --> 硬件相关的操作。

一, "s3c2410_ts.c"是三星公司提供的触摸屏驱动。

简单分析此代码:

1, 入口函数:

```
struct platform_driver s3c2410ts_driver = {
    .driver = {
        .name = "s3c2410-ts",
        .owner = THIS_MODULE,
    },
    .probe = s3c2410ts_probe,
    .remove = s3c2410ts_remove,
};
```

内核中有同名的平台设备时“s3c2410-ts”时,“s3c2410ts_driver”结构中的“.probe = s3c2410ts_probe,”就被调用。

```
int __init s3c2410ts_init(void)
```

```
-->platform_driver_register(&s3c2410ts_driver);
-->int __init s3c2410ts_probe(struct platform_device *pdev)
这个函数开始是平台设备 “platform_device *pdev” 这个形参获得一些管脚/* Configure
GPIOs */。主线：
-->input_dev = input_allocate_device(); 分配一个 input_dev 结构空格。
-->ts.dev = input_dev; 开始设置这个 input_dev 结构体。
    ts.dev->evbit[0] = BIT(EV_SYN) | BIT(EV_KEY) | BIT(EV_ABS); 能产生哪类事
        件:EV_KEY 按键类、EV_ABS 绝对位移类
    ts.dev->keybit[LONG(BTN_TOUCH)] = BIT(BTN_TOUCH); 能产生按键类事件中
BTN_TOUCH 触摸
屏事件
```

```
    input_set_abs_params(ts.dev, ABS_X, 0, 0x3FF, 0, 0); 开始设置绝对位移方面的参数。
    input_set_abs_params(ts.dev, ABS_Y, 0, 0x3FF, 0, 0);
    input_set_abs_params(ts.dev, ABS_PRESSURE, 0, 1, 0, 0);
-->input_register_device(ts.dev); 注册这个设备。当有事件发生时上报事件。
```

在 “void touch_timer_fire(unsigned long data)” 中有很多这类上报事件，如：

```
input_report_abs(ts.dev, ABS_X, ts.xp);
input_report_abs(ts.dev, ABS_Y, ts.yp);
```

```
input_report_key(ts.dev, BTN_TOUCH, 1);
input_report_abs(ts.dev, ABS_PRESSURE, 1);等等。
```

从 “input_report_abs()” 中看到这个函数就是 input_event() 上报事件。

```
void input_report_abs(struct input_dev *dev, unsigned int code, int value)
{
    input_event(dev, EV_ABS, code, value);
}
```

二，触摸屏驱动框架：

1，头文件：

```
#include <linux/errno.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/input.h>
#include <linux/init.h>
#include <linux/serio.h>
#include <linux/delay.h>
#include <linux/platform_device.h>
#include <linux/clock.h>
```

```
#include <asm/io.h>
#include <asm/irq.h>

#include <asm/plat-s3c24xx/ts.h>

#include <asm/arch/regs-adc.h>
#include <asm/arch/regs-gpio.h>
```

2, 按习惯先写“入口函数”与“出口函数”框架:

1. 习惯先写入口函数

```
static int s3c_ts_init(void)
{
    //1.1, 分配一个 input_dev 结构体。

    //1.2, 设置, 设置分为两大类: a, 能产生哪类事件; b, 能产生这类事件里的哪些事件.
    //1.3, 注册

    //1.4. 硬件相关的操作
    return 0;
}

static void s3c_ts_exit(void)
{
}

module_init(s3c_ts_init);
module_exit(s3c_ts_exit);

MODULE_LICENSE("GPL");
```

2, 分配“input_dev”结构空间:

①, 先声明一个“input_dev”结构变量“”

//1.1.1 声明一个 input_dev 结构体变量 s3c_ts_dev。

```
static struct input_dev *s3c_ts_dev;
```

②, 再为这个 input_dev *s3c_ts_dev 变量分配空间:

//1.1.2, 为这个 input_dev 结构分配空间.

```
s3c_ts_dev = input_allocate_device();
```

3, 设置这个 input_dev *s3c_ts_dev 变量:

设置分两类: 产生哪类事件; 产生这类事件中的哪些具体事件。

①，产生哪类事件：按键类事件。触摸屏的“绝对位移”事件。

//1.2.1,能产生哪类事件.

set_bit(EV_KEY, buttons_dev->evbit); //按键类事件.

set_bit(EV_ABS, s3c_ts_dev->evbit); //触摸屏是绝对位移事件(鼠标是相对位移).

```
/*
 * Event types
 */
触摸屏中的“绝对位移”事件
#define EV_SYN          0x00
#define EV_KEY          0x01
#define EV_REL          0x02
#define EV_ABS          0x03
#define EV_MSC          0x04
#define EV_SW           0x05
#define EV_LED          0x11
#define EV_SND          0x12
#define EV_REP          0x14
#define EV_FF           0x15
#define EV_PWR          0x16
#define EV_FF_STATUS    0x17
#define EV_MAX          0x1f
```

鼠标是“相对位移”事件，而触摸屏是“绝对位移”事件。

②，产生这类“按键”事件中的哪些具体事件：

BTN_TOUCH(button touch 这个抽象出来的按键)

能产生 s3c_ts_dev->keybit “按键”类里的 BTN_TOUCH 触摸屏事件。

//1.2.2,能产生这类事件中的哪些事件。

set_bit(BTN_TOUCH, s3c_ts_dev->keybit); //产生“按键”类里的触摸屏事件。

```
static inline void input_set_abs_params(struct input_dev *dev, int axis, int min, int max,
int fuzz, int flat)
{
    dev->absmin[axis] = min;
    dev->absmax[axis] = max;
    dev->absfuzz[axis] = fuzz;
    dev->absflat[axis] = flat;

    dev->absbit[LONG(axis)] |= BIT(axis);
}
```

参 1，input_dev 结构体。

参 2 “axis”是指 x 方向的坐标。

参 3，最小值。

参 4，最大值。

最后两个参数不是知道是什么意思。

input_set_abs_params(s3c_ts_dev, ABS_X, 0, 0x3FF, 0, 0); //x 方向

x 方向的最小值为 0，最大值为 0x3FF,这个最大值要看 2440 手册触摸屏部分。

FEATURES

— Resolution: 10-bit

手册上说这个触摸屏可以产生“10 位”。触摸屏实质上是一个 ADC 转换器。10 位就

是 0x3FF.0x3FF 的二进制是“1 111 111 111”一共 10 位。

input_set_abs_params(s3c_ts_dev, ABS_Y, 0, 0x3FF, 0, 0); //x 方向最大 0x3FF 即 10 位

input_set_abs_params(s3c_ts_dev, ABS_PRESSURE, 0, 1, 0, 0); //压力方向级别只有 0 与 1，要么压下要么松开。

画图板是压力越大，画出来的线条越粗（笔迹越粗）。里面的压力有很多级别。这里这个触摸屏的压力级别很简单，要么 0 要么是 1，即压力方向只有 0 与 1，要么按下要么松开。

4，开始注册：

//1.3,注册

//1.1.3,分配了 input_dev 结构之后，开始注册。

input_register_device(s3c_ts_dev);

注册完后，就是开始硬件操作了。

//1.习惯先写入口函数

```
static int s3c_ts_init(void)
{
```

//1.1,分配一个 input_dev 结构体。

//1.1.2,为这个 input_dev 结构分配空间。

s3c_ts_dev = input_allocate_device();

//1.2,设置,设置分为两大类:a,能产生哪类事件;b,能产生这类事件里的哪些事件。

//1.2.1,能产生哪类事件。

set_bit(EV_KEY, s3c_ts_dev->evbit); //按键类事件。

set_bit(EV_ABS, s3c_ts_dev->evbit); //触摸屏是绝对位移事件(鼠标是相对位移)。

//1.2.2,能产生这类事件中的哪些事件。

set_bit(BTN_TOUCH, s3c_ts_dev->keybit); //产生“按键”类里的触摸屏事件。

input_set_abs_params(s3c_ts_dev, ABS_X, 0, 0x3FF, 0, 0); //x方向最大0x3FF即10位

input_set_abs_params(s3c_ts_dev, ABS_Y, 0, 0x3FF, 0, 0); //y方向。

input_set_abs_params(s3c_ts_dev, ABS_PRESSURE, 0, 1, 0, 0); //压力方向级别只有0与1，要么压下要

//1.3,注册

//1.1.3,分配了input_dev结构之后，开始注册。

input_register_device(s3c_ts_dev);

//1.4.硬件相关的操作

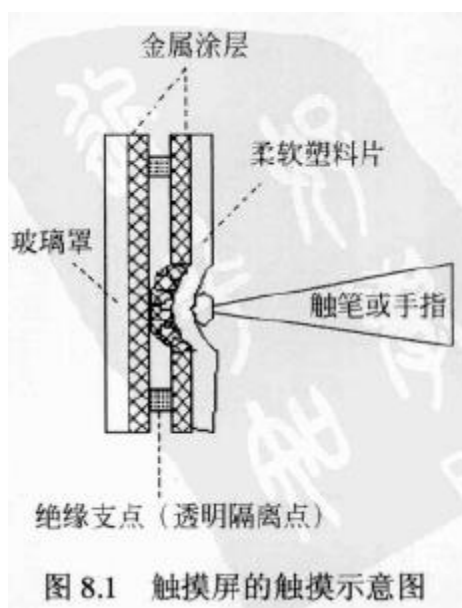
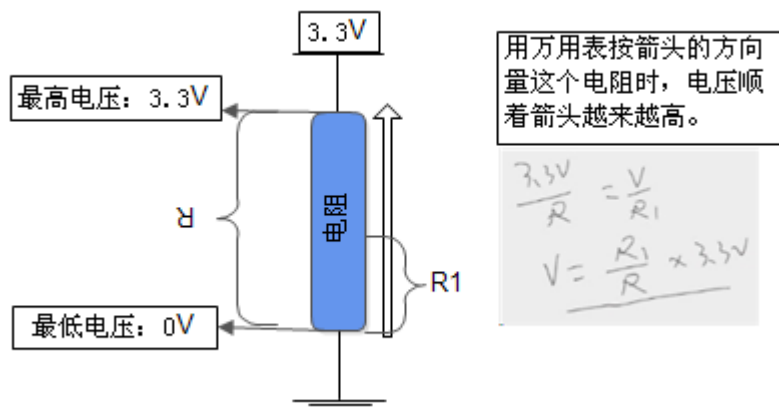
return 0;

} ? end s3c_ts_init ?

以上便是所有的触摸屏驱动的框架。

三，触摸屏硬件操作：

1. 触摸屏巧妙使用了“欧姆定律”。

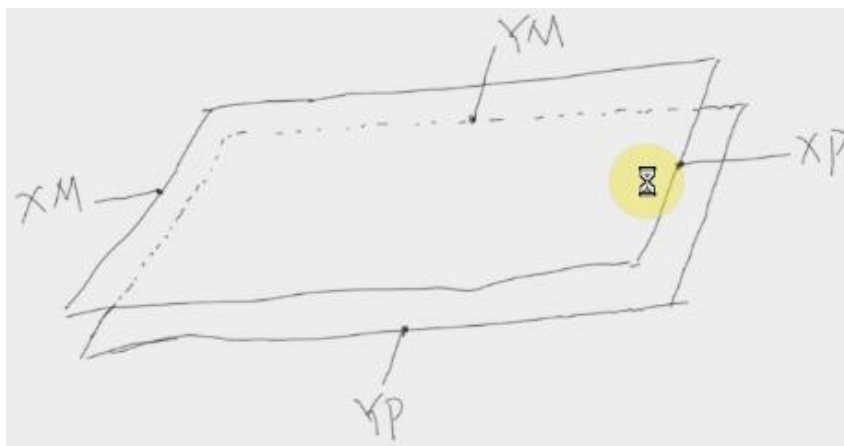


百问网
www.100ask.org

2，触摸屏硬件操作原理：

我们的开发板上的 LCD 屏上有触摸屏，触摸屏就是 2 层很薄的膜。覆盖在 LCD 屏上面。LCD 屏和触摸屏不同的器件，只是粘在一起了。

触摸屏两层膜引出 4 条线。



上面一层膜：

XP 边（P 表示正极的意思）。

XM 边（M 表示负极的意思）。

下面一层膜：

YP 边

YM 边

平时这两层膜接触不到，当有触摸时，即按压时，上下两层膜就会粘在一起。粘起来的那一点，如何测试 x,y 坐标：

A，测试 X 坐标：

- ①，XP 接 3.3V
- ②，XM 接地-0V。
- ③，YP 与 YM 都不接。这样上面一层相当于个电阻了。
- ④，测试 YP 电压。

电流从 XP 向 YP 流过去，当上面一层（YM,YP）与下一层由外力按压下粘有一个点时，这一个点就假设是一个万用表的触点了，就可以测量出 XP 与 XM 之间的任何一个“粘点”的电压，这个“粘点”越靠近 XM 接地端时，能“测量”到的电压就越低。

X 坐标只不过是一个 电压值。并不是 LCD 中，240*320 行、列里面的像素点坐标。

B，测试 Y 坐标：

- ①，YP 接 3.3V
- ②，YM 接地-0V。
- ③，XP 与 XM 都不接。这样上面一层相当于个电阻了。
- ④，测试 XP 电压。

也是当两层膜有外力按压时产生了一个“粘点”时，像万能表的指针一样可以量出这个 YP-
->YM（电流流向）间的电压值。

X,Y 测量时得到的是“电压值”，和 LCD 上的坐标没有关系。如何将的触摸屏与 LCD 屏对应起来，那是应用程序的事件。

电阻屏的手册屏经常要校验屏幕，校验就是使 LCD 屏 与 触摸屏 间建立一个联系。

3,一会再启动 ADC（模数转换）,把“粘点”的电压值测量出来成一个数值。

四，触摸屏使用过程：按此思路写硬件驱动程序。

1，按下（作为一个高效的系统，按下一般产生某个中断，不会是用轮询方式检测按下。）产生中断。

2，在“按下”中断处理程序里面，启动 ADC 转换 X、Y 坐标电压值。

3，这是启动 ADC，启动 ADC 不会瞬间完成。一般来说启动就不管了。ADC 结束，产生 ADC 中断（与‘按下’中断不同）。

4，在 ADC 中断处理函数里面，用 `input_event()` 来上报。

这个过程有个缺点，就是按下后只会启动一次，按下不松开时“粘点”划动，这样就不会新“按下”中断产生。

所以第 4 步里，上报后，启动“定时器”（处理长按、滑动的过程中也可以连续不断的转换这个 坐标电压值，上报出来）。

5，定时器 时间到，再到 第 2 步，再次启动 ADC 。就是按下时产生一个“粘点”中断，过了一段时间（定时器），就会再启动一次 ADC 中断，这样就能处理“粘点”滑动的过程。

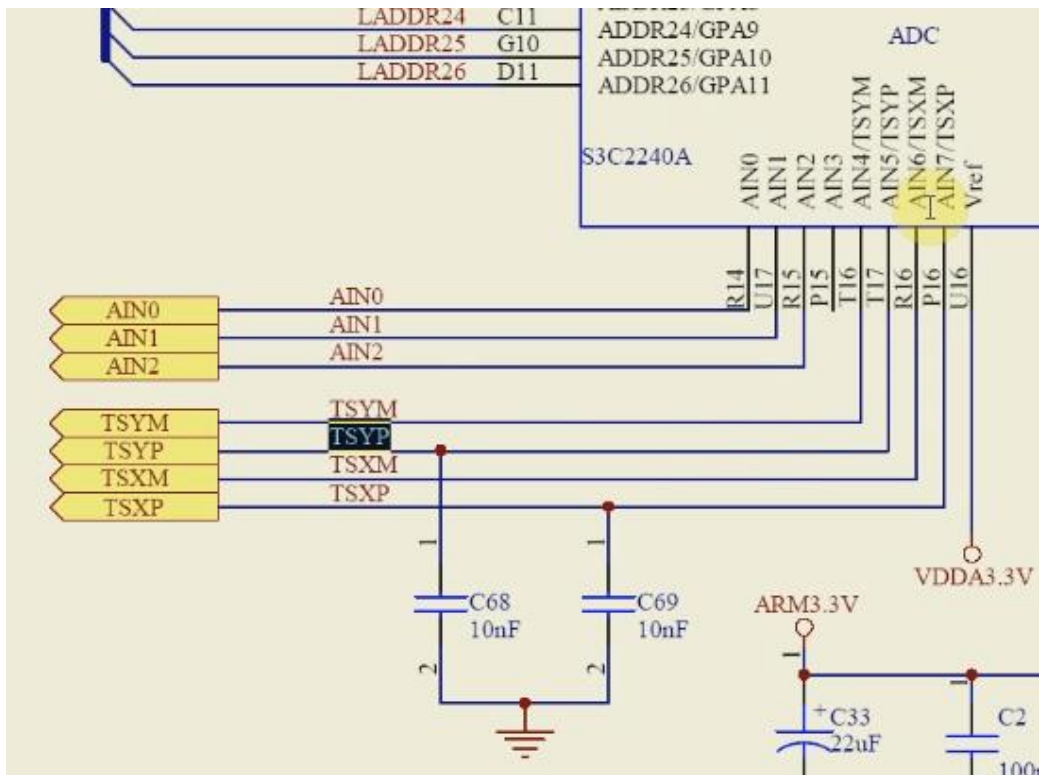
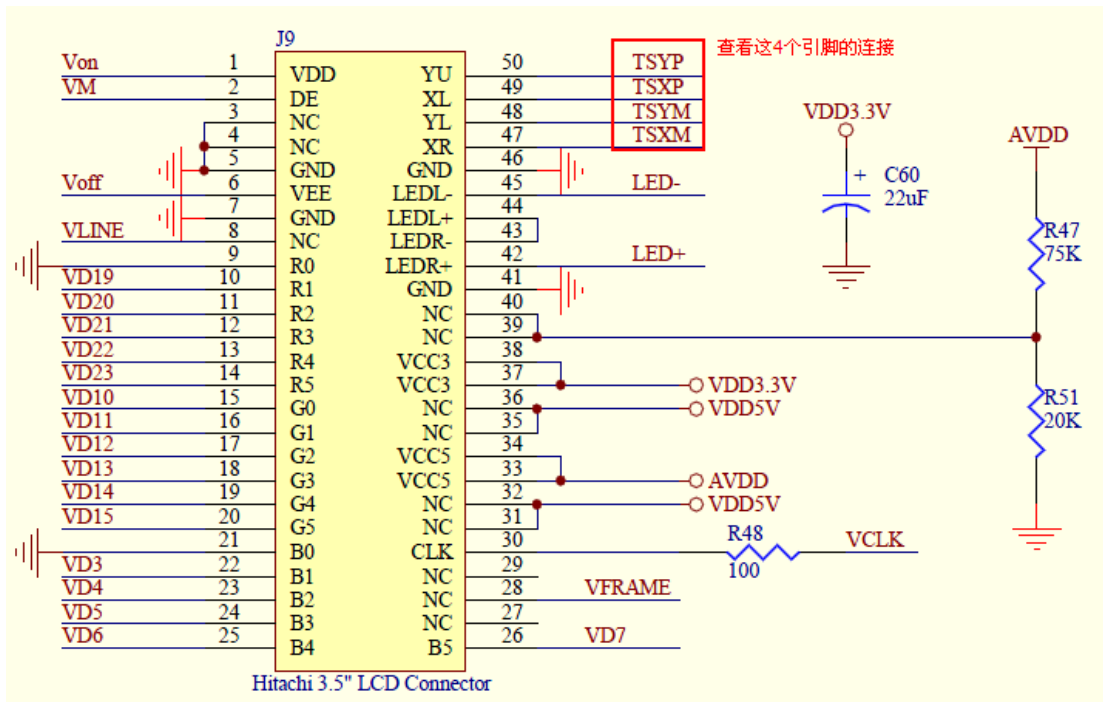
（这个定时器比如为 ms 级，人也反应不了这么快）。

6，松开。



触摸屏驱动编写

查看电路图



TSYP -- AIN5/TSYP
TSXP -- AIN7/TSXP
TSYM -- AIN4/TSYM
TSXM -- AIN6/TSXM

AIN4 ~ 7 不是一般的 GPIO 管脚，查 2440 手册。

SEL_MUX	[5:3]	Analog input channel select 000 = AIN 0 001 = AIN 1 010 = AIN 2 011 = AIN 3 100 = YM 101 = YP 110 = XM 111 = XP	0
---------	-------	-----------------------------------------------------------------------------------------------------------------------------------------	---

这几个位要么作为模拟输入，要么就用作触摸屏。看上面 100 = YM 就是 AIN4。

硬件相关的操作：

就是阅读 2440 第 16 章触摸屏这一章。

一，先看内核中自带的触摸屏驱动程序做的事件：

1，使能时钟:CLKCON[15] 设置为 “1”。

s3c2410_ts.c 中 “int __init s3c2410ts_probe(struct platform_device *pdev)”

-->adc_clock = clk_get(NULL, "adc"); //使能时钟.

-->clk_enable(adc_clock);

是设置 “CLKCON” 寄存器，将其 bit15 设置为 “1”。

为了省电，内核在启动时，对那些不是必须的模块都会先关掉。就是设置这个寄存器 “CLKCON”：

CLOCK CONTROL REGISTER (CLKCON)

Register	Address	R/W	Description	Reset Value
CLKCON	0x4C00000C	R/W	Clock generator control register	0xFFFFF0

2440 是所谓的片上系统（SOC）。它里面有很多的模块，如下：

CLKCON	Bit	Description	Initial State
AC97	[20]	Control PCLK into AC97 block. 0 = Disable, 1 = Enable	1
Camera	[19]	Control HCLK into Camera block. 0 = Disable, 1 = Enable	1
SPI	[18]	Control PCLK into SPI block. 0 = Disable, 1 = Enable	1
IIS	[17]	Control PCLK into IIS block. 0 = Disable, 1 = Enable	1
IIC	[16]	Control PCLK into IIC block. 0 = Disable, 1 = Enable	1
ADC(&Touch Screen)	[15]	Control PCLK into ADC block. 0 = Disable, 1 = Enable	1
RTC	[14]	Control PCLK into RTC control block. Even if this bit is cleared to 0, RTC timer is alive. 0 = Disable, 1 = Enable	1
GPIO	[13]	Control PCLK into GPIO block. 0 = Disable, 1 = Enable	1
UART2	[12]	Control PCLK into UART2 block. 0 = Disable, 1 = Enable	1
UART1	[11]	Control PCLK into UART1 block. 0 = Disable, 1 = Enable	1
UART0	[10]	Control PCLK into UART0 block. 0 = Disable, 1 = Enable	1
SDI	[9]	Control PCLK into SDI interface block. 0 = Disable, 1 = Enable	1
PWMTIMER	[8]	Control PCLK into PWMTIMER block. 0 = Disable, 1 = Enable	1
USB device	[7]	Control PCLK into USB device block. 0 = Disable, 1 = Enable	1
USB host	[6]	Control HCLK into USB host block. 0 = Disable, 1 = Enable	1
LCDC	[5]	Control HCLK into LCDC block. 0 = Disable, 1 = Enable	1
NAND Flash Controller	[4]	Control HCLK into NAND Flash Controller block. 0 = Disable, 1 = Enable	1
SLEEP	[3]	Control SLEEP mode of S3C2440A. 0 = Disable, 1 = Transition to SLEEP mode	0
IDLE BIT	[2]	Enter IDLE mode. This bit is not cleared automatically. 0 = Disable, 1 = Transition to IDLE mode	0
Reserved	[1:0]	Reserved	0

为了省电，内核一运行时，就会把这些不相关的（暂时不使用）的模块先关掉。我们要用哪一个模块时就要先把它打开。比如说上面的“ADC(&Touch Screen)”要打开时，就是把“CLKCON”寄存器的“bit[15]”设置为“1”-1 = Enable。

这里：adc_clock = clk_get(NULL, "adc"); //使能时钟。

clk_enable(adc_clock);

是设置“CLKCON”寄存器，将其bit15设置为“1”。在内核中搜索“"adc"”。在代码“clock.c”中：

/* standard clock definitions */

```
static struct clk init_clocks_disable[] = {
{
.name                = "nand",
.id                  = -1,
.parent              = &clk_h,
.enable              = s3c2410_clkcon_enable,
.ctrlbit             = S3C2410_CLKCON_NAND,
}, {
.name                = "sdi",
.id                  = -1,
.parent              = &clk_p,
```

```

.enable                = s3c2410_clkcon_enable,
.ctrlbit               = S3C2410_CLKCON_SDI,
}, {
.name                  = "adc",
.id                    = -1,
.parent                = &clk_p,
.enable                = s3c2410_clkcon_enable,
.ctrlbit               = S3C2410_CLKCON_ADC,
}, {
.name                  = "i2c",
.id                    = -1,
.parent                = &clk_p,
.enable                = s3c2410_clkcon_enable,
.ctrlbit               = S3C2410_CLKCON_IIC,
}, {
.name                  = "iis",
.id                    = -1,
.parent                = &clk_p,
.enable                = s3c2410_clkcon_enable,
.ctrlbit               = S3C2410_CLKCON_IIS,
}, {
.name                  = "spi",
.id                    = -1,
.parent                = &clk_p,
.enable                = s3c2410_clkcon_enable,
.ctrlbit               = S3C2410_CLKCON_SPI,
}
};

```

adc_clock = clk_get(NULL, "adc");是获得这个“adc”时钟，在 clock.c 中有这个时钟。里面有个 .enable = s3c2410_clkcon_enable, 哪一位呢：

```
#define S3C2410_CLKCON_ADC (1<<15)
```

clk_enable() 最终会调用到这个 “.enable = s3c2410_clkcon_enable,”

这个函数就是把那个寄存器 “CLKCON” 的 bit15，先读出 “S3C2410_CLKCON” 到 clkcon。

```
int s3c2410_clkcon_enable(struct clk *clk, int enable)
```

```
-->clkcon = __raw_readl(S3C2410_CLKCON); //先读出 S3C2410_CLKCON 到 clkcon。
```

```
-->if (enable) clkcon |= clocks; //若是开启时(开通 enable)，就或上 “1” (clocks = clk->ctrlbit)。
```

最后写进去：

```
clkcon &= ~(S3C2410_CLKCON_IDLE|S3C2410_CLKCON_POWER);
```

```
__raw_writel(clkcon, S3C2410_CLKCON);
```

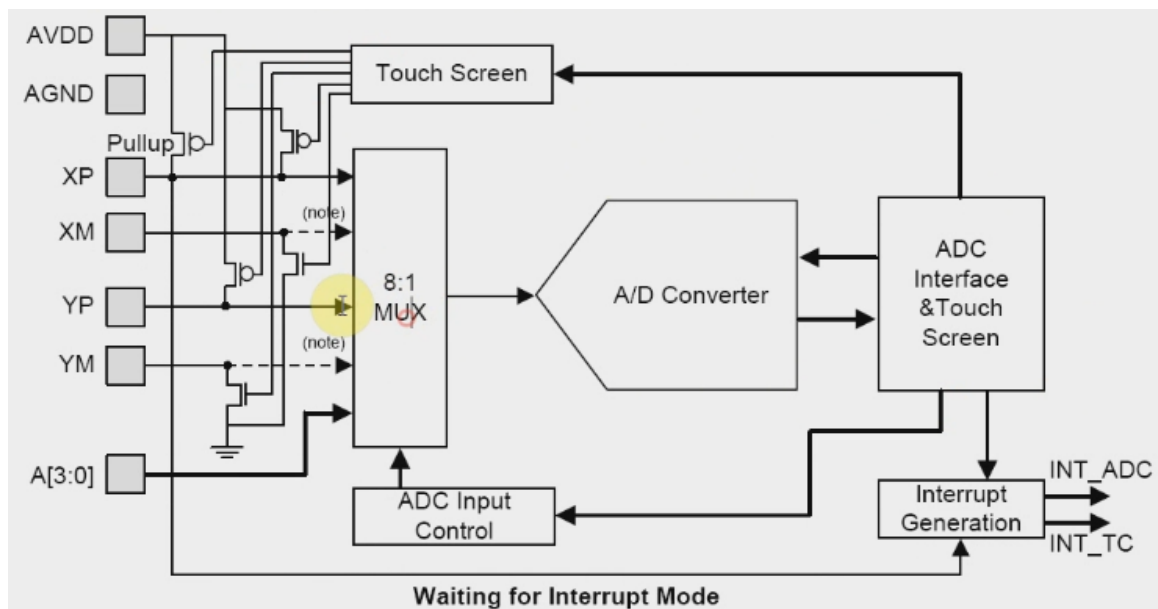
2, base_addr=ioremap(S3C2410_PA_ADC, 0x20);

二，查 2440 手册上硬件相关的说明。

A,2440 中有一个 10bit 的 CMOS ADC 模数转换器，它最大工作频率是 2.5MHz。

— Power Supply Voltage: 3.3V

最大输入电压是 3.3V。ADC 把模拟电压转成数字信号。因为这个 ADC 最大 10 位，则输入电压最大 3.3V 时，得到的就是“1 111 111 111”即 10 个“1”，输入 0V 时就得到“0”。每个刻度就是“3.3V/1023”，10bit 差不多 1024。3.3 除以 1024，每个刻度差不多 3 毫伏。



上面的“8：1”是指左边的 8 路 输入，选择一路出来，只有一个“ADC - A/D Converter”。就是指 8 路输入里面你选择转换哪一路。

A/D Conversion Time: ADC 转换时间。

若 PCLK 是 50MHz (2440 里 ADC 最大工作频率是 2.5MHz)，就要设置某些分频系数来降频。

若 (prescaler value is 49) 分频系数设为 49. 那么 ADC 的工作频率是：

$A/D \text{ converter freq.} = 50\text{MHz} / (49+1) = 1\text{MHz}$

转换一次需要 5 个周期 (5cycles)，1MHz 的 5 个周期就是 5 微秒。

$\text{Conversion time} = 1 / (1\text{MHz} / 5\text{cycles}) = 1 / 200\text{KHz} = 5 \text{ us}$

若要 ADC 工作在 2.5MHz，就把 2.5 带到上面的公式中算出 1 秒钟可以转换 500KSPS 次。

B,Touch Screen Interface Mode 触摸屏接口模式：

表 8.1 触摸屏接口的 4 个工作模式			
模式	AUTO_PST	XY_PST	说明
常规转换模式	AUTO_PST=0	XY_PST=0	通用的 ADC 转换
XY 坐标独立转换模式	AUTO_PST=0	XY_PST = 1	转换 X 坐标到 ADCDAT0 的 XPDATA，并产生 INT_ADC 中断
		XY_PST = 2	转换 Y 坐标到 ADCDAT1 的 YPDATA 并产生 INT_ADC 中断
XY 坐标自动转换模式	AUTO_PST=1	XY_PST=0	自动转换 X 坐标到 ADCDAT0 的 XPDATA；同时转换 Y 坐标到 ADCDAT1 的 YPDATA；并产生 INT_ADC 中断
等待中断模式		XY_PST=3	当触摸笔点击时产生中断，此时可以采用 XY 坐标独立转换模式或 XY 坐标自动转换模式读取坐标数据

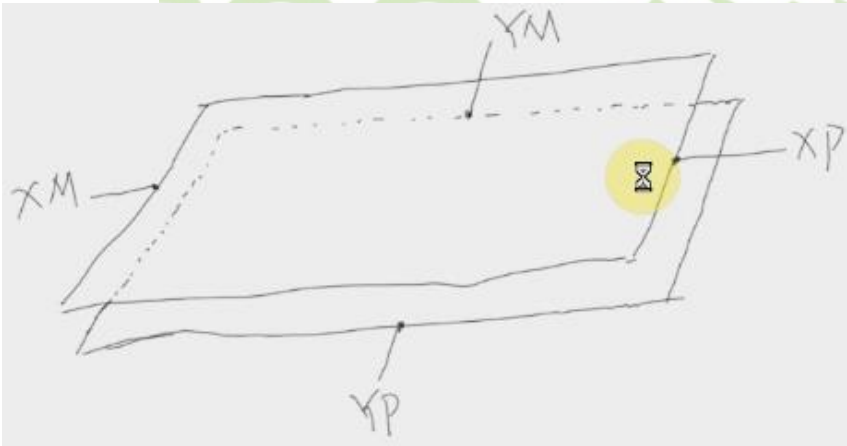
1,正常的转换模式：

进行一般的 ADC 操作。就是用于一般的 ADC 模数转换。

2，分离的 X,Y 坐标转换模式：

有两种模式—测量 X 坐标 或 测量 Y 坐标

转换 X,Y 坐标：



测量 X 坐标：要设置 XP 接 3.3V，XM 接地时，YM, YP 不接，再测量 YP 的电压。

测量 Y 坐标：要设置 YP 接 3.3V, YM 接地，XP, XM 不接，再测量 XP 的电压。

“分离的 X,Y 坐标转换模式”有两种模式，一种是测量 X 坐标，一种是测量 Y 坐标。

这些别人已经做好了，只要进入“分离的 X,Y 坐标转换模式”，选择进入测量 X 坐标模式后，会把 X 坐标的转换值写到这个“ADCDAT0”寄存器。这样然后产生一个中断。

当进入 Y 坐标转换模式之后，会把 Y 坐标的值写到“ADCDAT1”寄存器中去，然后产生一个中断。

3，“自动（连续的）X,Y 坐标转换模式”：

当进入这个模式之后，它会自动的帮你即转换 X 坐标，也转换 Y 坐标，

X 坐标的值写到“ADCDAT0”，Y 坐标的值写到“ADCDAT1”。然后产生一个中断。

第 2 种方式中是分两次操作，这里第 3 种方式中连续的操作。

4, 等待中断模式:

等待触摸笔按下、松开模式。

想产生中断, 就先让 2440 处于第 4 种“等待中断模式”。当触摸笔按下时触摸屏会产生“INT_TC”中断。

让“rADCTSC=0xd3”这个寄存器等于 0xd3 就可以。

C, Standby Mode 省电模式: 不用管。

D, 操作寄存器:

三, 操作寄存器:

0, 使能时钟:

//1.4.1, 定义一个时钟

```
struct clk* clk;
```

//1.4. 硬件相关的操作

//1.4.2, 使能时钟, 是设置 CLKCON[15]为 1.

```
clk = clk_get(NLLL, "adc");
```

```
clk_enable(clk);
```

1, 操作寄存器之前, 要 ioremap():

写一个结构体简化寄存器的 ioremap();

下面几个寄存器要 ioremap(); 寄存器间偏差刚好是 4 字节。

Register Name	Address (B. Endian)	Address (L. Endian)	Acc. Unit	Read/Write	Function
A/D Converter					
ADCCON	0x58000000	←	W	R/W	ADC control
ADCTSC	0x58000004				ADC touch screen control
ADCDLY	0x58000008				ADC start or interval delay
ADCDAT0	0x5800000C			R	ADC conversion data
ADCDAT1	0x58000010				ADC conversion data
ADCUPDN	0x58000014			R/W	Stylus up or down interrupt status

//1.4.2.1, 要 ioremap() 的触摸屏寄存器结构。

```
struct s3c_ts_regs {  
    unsigned long adcon;  
    unsigned long adctsc;  
    unsigned long adcdly;  
    unsigned long adcdat0;
```

```

    unsigned long adcdatl;
    unsigned long adcupdn;
};
//1.4.2.2, 定义一个此 s3c_ts_regs 结构的指针.
static volatile struct s3c_ts_regs *s3c_ts_regs;
定义了这个“s3c_ts_regs”结构的指针后，它指向哪里。（ioremap()后即可）。
//1.4.2.4, 对 ADC 转换相关寄存器 ioremap() ;
    s3c_ts_regs = ioremap(0x58000000, sizeof(s3c_ts_regs));

```

触摸屏使用过程：

按此思路写硬件驱动程序：TS 工作流程

- 1，按下（作为一个高效的系统，按下一般产生某个中断，不会是用轮询方式检测按下。）产生中断。
- 2，在“按下”中断处理程序里面，启动 ADC 转换 X、Y 坐标电压值。
- 3，这是启动 ADC，启动 ADC 不会瞬间完成。一般来说启动就不管了。ADC 结束，产生 ADC 中断（与‘按下’中断不同）。
- 4，在 ADC 中断处理函数里面，用 input_event() 来上报。
这个过程有个缺点，就是按下后只会启动一次，按下不松开时“粘点”划动，这样就不会新“按下”中断产生。
所以第 4 步里，上报后，启动“定时器”（处理长按、滑动的过程中也可以连续不断的转换这个 坐标电压值，上报出来）。
- 5，定时器 时间到，再到 第 2 步，再次启动 ADC 。就是按下时产生一个“粘点”中断，过了一段时间（定时器），就会再启动一次 ADC 中断，这样就能处理“粘点”滑动的过程。（这个定时器比如为 ms 级，人也反应不了这么快）。
- 6，松开。

以上是触摸屏的硬件使用过程，可以依照着设置并测试触摸屏是否设置成功。慢慢完美驱动程序。

A,先看下是否能检测到触摸屏，按下 和 松开。

A_1，开始设置 ADCCON 寄存器，ADCCON 寄存器： [15:0]位。

ADC CONTROL REGISTER (ADCCON)

Register	Address	R/W	Description	Reset Value
ADCCON	0x5800000	R/W	ADC control register	0x3FC4

①. ECFLG [15]：只读位，不用管。

ECFLG	[15]	End of conversion flag (Read only) 0 = A/D conversion in process 1 = End of A/D conversion	0
-------	------	--------------------------------------------------------------------------------------------------	---

ECFLG	[15]	AD 转换结束标志: 0=AD 转换进行中; 1=AD 转换结束	0
-------	------	----------------------------------------	---

②, PRSCEN [14]: 预分频使能。要设置为“1”。

PRSCEN	[14]	A/D converter prescaler enable 0 = Disable 1 = Enable	0
--------	------	-------------------------------------------------------------	---

PRSCEN	[14]	AD 转换预分频器使能: 0=停止; 1=使能	0
--------	------	-------------------------------	---

可以从内核的启动信息中得知“PCLK”的值:

```
CPU S3C2440A (id 0x32440001)
S3C244X: core 400.000 MHz, memory 100.000 MHz, peripheral 50.000 MHz
```

400MHz 是 FCLK (CPU 工作时钟)。

Memory 100MHz 是 HCLK。

Peripheral 50MHz 是 PCLK。

③, PRSCVL [13:6]: 预分频系数。这里也简单些, 设置为 49, 所以 ADC 时钟:

$ADCCLK = PCLK / (49 + 1) = 50MHz / (49 + 1) = 1MHz$

PRSCVL	[13:6]	A/D converter prescaler value Data value: 0 ~ 255 NOTE: ADC Frequency should be set less than PCLK by 5times. (Ex. PCLK=10MHZ, ADC Freq.< 2MHz)	0xFF
--------	--------	-------------------------------------------------------------------------------------------------------------------------------------------------------	------

PRSCVL	[13:6]	AD 转换预分频器数值 范围: 1~255; 除数为 (PRSCVL+1); 注意 ADC 频率应该设置成小于 PCLK 的 5 倍	0xFF
--------	--------	-----------------------------------------------------------------------------	------

④, SEL_MUX [5:3]: 多路选择。一共 8 路。

SEL_MUX	[5:3]	Analog input channel select 000 = AIN 0 001 = AIN 1 010 = AIN 2 011 = AIN 3 100 = YM 101 = YP 110 = XM 111 = XP	0
---------	-------	-----------------------------------------------------------------------------------------------------------------------------------------	---

SEL_MUX	[5:3]	模拟输入通道选择: [0, 7]对应 AIN0~AIN7	0
---------	-------	---------------------------------	---

若用作一般的 ADC 时, 就可以设置这个寄存器, 就是从这 8 路里想转换哪一路来模拟输入。这里我们先不用管。

⑤, STDBM [2]:

STDBM	[2]	Standby mode select 0 = Normal operation mode 1 = Standby mode	1
-------	-----	----------------------------------------------------------------------	---

STDBM	[2]	Standby 模式选择: 0=普通模式; 1=standby 模式	1
-------	-----	------------------------------------------	---

这里我们要设置为 0，要工作起来，要是设置为 1 了就进入省电模式了。

⑥，READ_START [1]：启动 ADC 有两种方式，1 为通过读操作来使能。

READ_START	[1]	A/D conversion start by read 0 = Disable start by read operation 1 = Enable start by read operation	0
------------	-----	-----------------------------------------------------------------------------------------------------------	---

READ_START	[1]	通过读取来启动 AD 转换: 0=停止通过读取来启动 AD 转换; 1=使能通过读取来启动 AD 转换	0
------------	-----	-----------------------------------------------------------	---

我们不需要这么智能来使能 AD 转换，所以直接设置为 0。

⑦，ENABLE_START [0]：

ENABLE_START	[0]	A/D conversion starts by enable. If READ_START is enabled, this value is not valid. 0 = No operation 1 = A/D conversion starts and this bit is cleared after the start-up.	0
--------------	-----	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---

ENABLE_START	[0]	启动 AD 操作: 如果 READ_START=1, 则这个值无效。 0=无操作; 1=启动 AD 转换, 启动后该位清零	0
--------------	-----	------------------------------------------------------------------------	---

将此位设置为 1，就启动 AD 转换。

当 AD 转换启动后，它会自动清零。这里我们不需要，可以以后来设置它。

A_2，注册 INT_TC (touch change) 中断：

```
int request_irq(unsigned int irq, irq_handler_t handler,
unsigned long irqflags, const char *devname, void *dev_id) {}
```

参 2 中断处理函数得自己写。

参 3 是 irqflags。

参 4 是名字。这里取为“ts_pen”触摸笔。

//1.4.3, 注册中断。

```
request_irq(IRQ_TC, pen_down_up_irq, IRQF_SAMPLE_RANDOM, "ts_pen", NULL);
```

注册完中断后，如何让按下触摸笔时这个触摸屏上产生中断，松开又产生中断。就是让它进入这个“等待中断模式”。

4. Waiting for Interrupt Mode

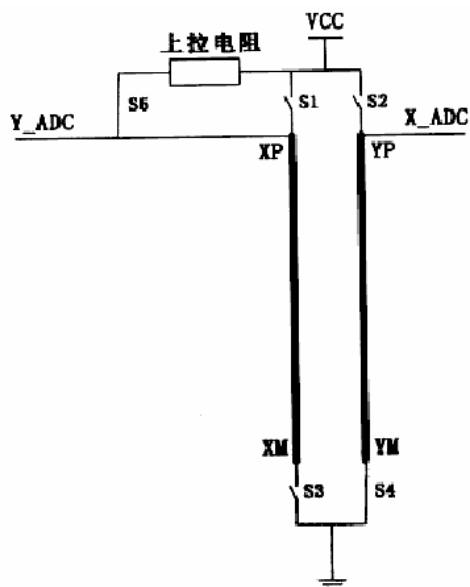
Touch Screen Controller generates interrupt (INT_TC) signal when the Stylus is down. Waiting for Interrupt Mode setting value is **ADCTSC=0xd3**; // XP_PU, XP_Dis, XM_Dis, YP_Dis, YM_En.

After Touch Screen Controller generates interrupt signal (INT_TC), Waiting for interrupt Mode must be cleared. (XY_PST sets to the No operation Mode)

要设置寄存器 adctsc = 0xd3; 进入“等待中断模式”。

为什么设置“adctsc = 0xd3”后就进入“等待中断模式”，分析“0xd3”：

当 bit7 为 1 时: YM 要输出。要接地。



5 触摸屏处于“等待中断模式”时的等效电路

图中“YM”是接地。那些位显然是用来控制图中“S1 ~ 5”这些开关的。

S4、S5 闭合，S1、S2、S3 断开，即 YM 接地、XP 上拉、XP 作为模拟输入（对 CPU 而言）、YP 作为模拟输入（对 CPU 而言）、XM 高阻。

平时触摸屏没有被按下时，由于上拉电阻的关系，Y_ADC 为高电平；当 x 轴和 y 轴受挤压而接触导通后，Y_ADC 的电压由于连通到 y 轴接地而变为低电平，此低电平可做为中断触发信号来通知 CPU 发生“Pen Down”事件，在 S3C2410/S3C2440 中，称为等待中断模式。

上接电阻“Y_ADC → S5 → 上拉电阻 → VCC”这条线平时是高电平。“YM → S4”是接地，当按下触摸笔时，两下两层膜（图中 XP, YP 两条粗黑线）相接时，就会有 VCC - 上接电阻 → S5 → Y_ADC → XP → YM → S4 这样就通了，VCC 电压就变低了。这样触摸屏里面模块的硬件就能检测引脚 Y_ADC 发生了变化，就知道这个触摸屏按下了。

这里 bit7 = 1，应该就上图里的意思，YM 要接地。S1, S2, S3, S5 要断开，上拉电阻要使能。

YP_SEN [6]:

YP_SEN	[6]	YP Switch Enable 0 = YP Output Driver Enable. 1 = YP Output Driver Disable.	1
--------	-----	-----------------------------------------------------------------------------------	---

YP_SEN	[6]	选择 nYPON 的输出值: 0=nYPON 输出 0 (YP=External voltage); 1=nYPON 输出 1 (YP 连接 AIN[5])	1
--------	-----	--------------------------------------------------------------------------------------	---

Bit6 为 1 时，表示 YP 输出要禁止，看 bit7 中描述的“等效电路”图中，YP 上与 VCC 断开着，即 YP 的输出禁止着。

XM_SEN [5]:

XM_SEN	[5]	XM Switch Enable 0 = XM Output Driver Disable. 1 = XM Output Driver Enable.	0
--------	-----	-----------------------------------------------------------------------------------	---

XM_SEN	[5]	选择 XMON 的输出值: 0=XMON 输出 0 (XM=Hi_Z); 1=XMON 输出 1 (XM=GND)	0
--------	-----	-----------------------------------------------------------------	---

Bit5 为 0 时, XM 要断开, 也是看上面的“等效电路”图, XM 用 S3 断开着。是断开的。

XP_SEN [4]:

XP_SEN	[4]	XP Switch Enable 0 = XP Output Driver Enable. 1 = XP Output Driver Disable.	1
--------	-----	-----------------------------------------------------------------------------------	---

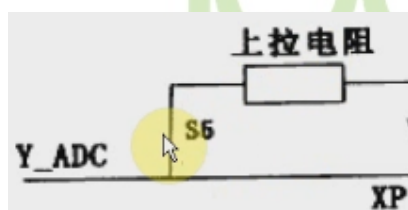
XP_SEN	[4]	选择 nXPON 的输出值: 0=nXPON 输出 0 (XP=External voltage); 1=nXPON 输出 1 (XP 连接 AIN[7])	1
--------	-----	--------------------------------------------------------------------------------------	---

Bit4 为 0 时, XP 要断开。看上面“等效电路”图, XP 用 S1 开关断开着。

PULL_UP [3]:

PULL_UP	[3]	Pull-up Switch Enable 0 = XP Pull-up Enable. 1 = XP Pull-up Disable.	1
---------	-----	----------------------------------------------------------------------------	---

PULL_UP	[3]	上拉开关: 0=允许 XP 上拉; 1=禁止 XP 上拉	1
---------	-----	------------------------------------	---



Bit3 为 0 时, 上是 XP 上拉使能。上图“等效电路”确实是 S5 连接着, XP 上拉使能着。

AUTO_PST [2]:

AUTO_PST	[2]	Automatically sequencing conversion of X-Position and Y-Position 0 = Normal ADC conversion. 1 = Auto Sequential measurement of X-position, Y-position.	0
----------	-----	--------------------------------------------------------------------------------------------------------------------------------------------------------------	---

AUTO_PST	[2]	模式选择: 0=正常 ADC 转换; 1=自动 XY 坐标转换	0
----------	-----	---------------------------------------	---

这个 AUTO_PST 位取 0 时为“正常的 ADC 转换”, 置 1 时是“自动 XY 坐标轮换”。我们这里是看第 4 种“等待中断模式”, 所以这个 AUTO_PST 位不用管。

XY_PST [1:0]:

XY_PST	[1:0]	Manually measurement of X-Position or Y-Position. 00 = No operation mode 01 = X-position measurement 10 = Y-position measurement 11 = Waiting for Interrupt Mode	0
--------	-------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---

XY_PST	[1:0]	测量模式:	0
		00=无操作;	
		01=X 坐标测量;	
		10=Y 坐标测量;	
		11=等待中断模式	

Bit[1:0] 设置为 11 - 等待中断模式。

所以代码如下:

一开始是 等待触摸笔 按下模式。

//从 ADCTSC UD_SEN-bit8, 此位为 0 时表示检测触摸笔按下中断信号(等待触摸笔按下)

```
static void enter_wait_pen_down_mode(void)
```

```
{
```

```
    //将此寄存器的 adctsc 寄存器设置为 0xd3 就进入“等待中断模式”。
```

```
    s3c_ts_regs->adctsc = 0xd3;
```

```
}
```

//从 ADCTSC UD_SEN-bit8, 此位为 1 时表示检测触摸笔松开中断信号(等待触摸笔松开)

```
static void enter_wait_pen_up_mode(void)
```

```
{
```

```
    //将此寄存器的 adctsc 寄存器设置为 0xd3 就进入“等待中断模式”。
```

```
    s3c_ts_regs->adctsc = 0xd3; //ACTSC 的 bit8 设置为 1.
```

```
}
```

//1. 4. 3. 1, 设置进入等待触摸笔按下或松开模式的函数. 是先按下模式.

```
enter_wait_pen_down_mode(void); //进入等待触摸笔按下模式。
```

```
//1. 4. 3. 2, 触摸笔按下中断模式后, 开始写 中断处理函数 pen_down_irq().
```

上面就是“进入中断”模式, 当进入中断时, 就要有中断处理函数。

```
//irqreturn_t (*irq_handler_t)(int, void *) {}
```

```
irqreturn_t pen_down_irq(int irq, void *dev_id)
```

```
{
```

```
    printk("pen down/up");
```

```
}
```

这个中断处理函数有按下和松开, 那么如何判断是触摸笔按下了还是松开了。这就看

“ADCDAT0” 寄存器。

ADC CONVERSION DATA REGISTER (ADCDAT0)

Register	Address	R/W	Description	Reset Value
ADCDAT0	0x580000C	R	ADC conversion data register	—

UPDOWN	[15]	Up or Down state of stylus at waiting for interrupt mode. 0 = Stylus down state. 1 = Stylus up state.		—
--------	------	-------------------------------------------------------------------------------------------------------------	--	---

UPDOWN	[15]	等待中断模式下触摸笔状态： 0=触摸笔按下； 1=触摸笔拿起	-
--------	------	--------------------------------------	---

bit15 置 0 时表示触摸笔按下状态，置 1 时表示触摸笔松开状态。

AUTO_PST	[14]	Automatic sequencing conversion of X-position and Y-Position 0 = Normal ADC conversion. 1 = Sequencing measurement of X-position, Y-position.	-
----------	------	-----------------------------------------------------------------------------------------------------------------------------------------------------	---

AUTO_PST	[14]	0=常规 ADC 转换； 1=XY 坐标顺序测量	-
----------	------	-----------------------------	---

XY_PST	[13:12]	Manually measurement of X-position or Y-position. 00 = No operation mode 01 = X-position measurement 10 = Y-position measurement 11 = Waiting for Interrupt Mode	-
--------	---------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---

XY_PST	[13:12]	00=无操作； 01=X 坐标测量； 10=Y 坐标测量； 11=等待中断模式	-
--------	---------	--------------------------------------------------	---

Reserved	[11:10]	Reserved	-
----------	---------	----------	---

Reserved	[11:10]	Reserved	-
----------	---------	----------	---

XPDATA (Normal ADC)	[9:0]	X-Position conversion data value (include normal ADC conversion data value) Data value: 0 ~ 3FF	-
------------------------	-------	----------------------------------------------------------------------------------------------------	---

XPDATA	[9:0]	X 坐标值或常规 ADC 转换值，支持到 0x3FF	-
--------	-------	----------------------------	---

则代码为：

```
//irqreturn_t (*irq_handler_t)(int, void *) {}
irqreturn_t pen_down_up_irq(int irq, void *dev_id)
{
    //2440 手册中 ADCDAT0 寄存器 bit15 置 0 时表示触摸笔按下状态，置 1 时表示触摸笔松开状态。
    if(s3c_ts_regs->adcdat0 & (1<<15))
    {
        printk("pen up"); //松开时就进入“等待按下模式”
        enter_wait_pen_down_mode(); 这样才可处理下次
        enter_wait_pen_up_mode();
    }
    else
```

```

    {
        printk("pen down"); //按下时就进入“等待松开”模式
enter_wait_pen_up_mode();
        enter_wait_pen_up_mode();
    }
    return IRQ_HANDLED;
}

```

这样操作后，一会测试触摸屏时，按下、松开就会的打印出来. 以上代码就先只作 按下，松开的处理。

写完“出口函数”，就开始编译测试下。

```

static void s3c_ts_exit(void)
{
    //释放 IRQ_TC.
    free_irq(IRQ_TC, NULL);
//释放 IRQ_ADC 中断
    free_irq(IRQ_ADC, NULL);
    //iounmap 掉寄存器
iounmap(s3c_ts_regs);
    //从内核释放此 s3c_ts_dev 结构体
input_unregister_device(s3c_ts_dev);
    //释放 s3c_ts_dev 结构空间.
input_free_device(s3c_ts_dev);
}

```



四，测试：触摸笔按下、松开。

```

book@book-desktop:/work/drivers_and_test/11th_ts/2th$ make
make -C /work/system/linux-2.6.22.6 M=`pwd` modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
  CC [M] /work/drivers_and_test/11th_ts/2th/s3c_ts.o
/work/drivers_and_test/11th_ts/2th/s3c_ts.c: In function `s3c_ts_init':
/work/drivers_and_test/11th_ts/2th/s3c_ts.c:94: warning: ignoring return value of `request_irq', declared with attribute warn_unused_result
  Building modules, stage 2.
  MODPOST 1 modules
  CC /work/drivers_and_test/11th_ts/2th/s3c_ts.mod.o
  LD [M] /work/drivers_and_test/11th_ts/2th/s3c_ts.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
book@book-desktop:/work/drivers_and_test/11th_ts/2th$ cp s3c_ts.ko /work/nfs_root/first_fs
book@book-desktop:/work/drivers_and_test/11th_ts/2th$

```

1. make menuconfig 去掉原来的触摸屏驱动程序

-> Device Drivers

-> Input device support (输入设备)

-> Generic input layer

-> Touchscreens

<> S3C2410/S3C2440 touchscreens

make uImage

```
book@book-desktop:/work/system/linux-2.6.22.6$ cp arch/arm/boot/uImage /work/nfs_root/uImage_nots
```

使用新内核启动

```
[q] Quit from menu
Enter your selection: q
OpenJTAG> nfs 30000000 192.168.1.5:/work/nfs_root/uImage_nots
dm9000 i/o: 0x20000000, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 08:00:3e:26:0a:5b
could not establish link
File transfer via NFS from server 192.168.1.5; our IP address is 192.168.1.17
Filename '/work/nfs_root/uImage_nots'.
Load address: 0x30000000
Loading: #####
#####
```

用 NFS 下载新的没有 TS 的内核镜像。

下载内核后，启动：

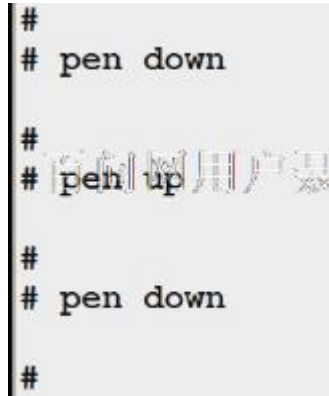
```
#####
done
Bytes transferred = 1841288 (1c1888 hex)
OpenJTAG> bootm 30000000
## Booting image at 30000000 ...
```

挂载 NFS 文件：

```
# ls
bin      lib      mnt      root     tmp
dev      linuxrc  opt      sbin     usr
etc      lost+found  proc    sys
# mount -t nfs -o nolock,vers=2 192.168.1.5:/work/nfs_root/first_fs /mnt
# cd /mnt/
# insmod s3c_ts.ko
input: Unspecified device as /class/input/input0
pen down
pen up
# █ Linux视频驱动第2期/100ask.taobao.com/00&Email:wei
```

2. insmod s3c_ts.ko

按下/松开触摸笔



可以看到测试成功的。

B, 按照上面的 TS 工作流程是, 在“按下”中断处理程序里面, 启动 ADC 转换 X、Y 坐标电压值。

```
//irqreturn_t (*irq_handler_t)(int, void *) {}  
irqreturn_t pen_down_up_irq(int irq, void *dev_id)  
{//2440 手册中 ADCDAT0 寄存器 bit15 置 0 时表示触摸笔按下状态, 置 1 时表示触摸笔松开状态。
```

```
    if(s3c_ts_regs->adcdat0 & (1<<15))  
    {  
        printk("pen up"); //松开时就进入“等待按下模式”  
        enter_wait_pen_down_mode(); 这样才可处理下次  
        enter_wait_pen_up_mode();  
    }  
    else  
    {  
        //printk("pen down"); //按下时就进入“等待松开”模式  
        enter_wait_pen_up_mode();  
        //enter_wait_pen_up_mode();  
        //3.1, 测量 XY 坐标模式。  
        enter_measure_xy_mode();  
        //3.2, 然后开始 adc 转换。  
        start_adc();  
    }  
    return IRQ_HANDLED;  
}
```

触摸屏有两种转换的方法:

2, “分离的 X,Y 坐标转换模式”有两种模式, 一种是测量 X 坐标, 一种是测量 Y 坐标。

这些别人已经做好了, 只要进入“分离的 X,Y 坐标转换模式”, 选择进入测量 X 坐标模式后, 会把 X 坐标的转换值写到这个“ADC DAT0”寄存器。这样然后产生一个中断。

当进入 Y 坐标转换模式之后，会把 Y 坐标的值写到“ADCDAT1”寄存器中去，然后产生一个中断。

3, “自动（连续的）X,Y 坐标转换模式”：当进入这个模式之后，它会自动的帮你即转换 X 坐标，也转换 Y 坐标

标，X 坐标的值写到“ADCDAT0”，Y 坐标的值写到“ADCDAT1”。然后产生一个中断。

第 2 种方式中是分两次操作，这里第 3 种方式中连续的操作。

这里我们用简单的第 3 种“自动（连续的）X,Y 坐标转换模式”。让其进入这种模式后，再启动 ADC 转换。

ADC 转换启动需要一定的时间，一般我们不会死等着它完成，它启动完成后会产生一个中断。所以再定义一个中断处理函数。

//1.4.3,注册中断。

```
request_irq(IRQ_TC, pen_down_up_irq, IRQF_SAMPLE_RANDOM, "ts_pen", NULL);
```

```
//3.2.2,ADC 启动完成后启用一个中断
```

```
request_irq(IRQ_ADC, adc_irq, IRQF_SAMPLE_RANDOM, "adc", NULL);
```

按下触摸笔，进入“pen_down_up_irq()”中断服务程序：

```
request_irq(IRQ_TC, pen_down_up_irq, IRQF_SAMPLE_RANDOM, "ts_pen", NULL);
```

```
-->irqreturn_t pen_down_up_irq(int irq, void *dev_id)()
```

在这个“pen_down_up_irq()”中断服务程序里面又再次让触摸屏进入“测量 XY 坐标模式”

```
-->enter_measure_xy_mode();
```

再启动 ADC 转换。

```
-->start_adc();
```

B_1,进入测量 XY 坐标模式：设置 s3c_ts_regs->adctsc 的值。

3. Auto(Sequential) X/Y Position Conversion Mode

Auto (Sequential) X/Y Position Conversion Mode is operated as the following. Touch Screen Controller sequentially converts X-Position and Y-Position that is touched. After Touch controller writes X-measurement data to ADCDAT0 and writes Y-measurement data to ADCDAT1, Touch Screen Interface is generating Interrupt source to Interrupt Controller in Auto Position Conversion Mode.

这里面没有说如何进入“测量 XY 坐标模式”的方法，这样就只能看寄存器的说明。

看“ADCTSC”寄存器，里面有各种开关的设置。

ADC TOUCH SCREEN CONTROL REGISTER (ADCTSC)

Register	Address	R/W	Description	Reset Value
ADCTSC	0x5800004	R/W	ADC Touch Screen Control Register	0x58

1. 当等待触摸屏中断时，XP_SEN 位应该被设置为‘1’（XP 输出禁止）并且 PULL_UP 位应该被设置为 ‘0’（XP 上拉使能）。
2. 只有在自动顺序 X/Y 方向转换时 AUTO_PST 位应该被设置为‘1’。
3. 在睡眠模式期间应该分离 XP、YP 与 GND 源以避免漏电电流。因为 XP、YP 将在睡眠模式中保持为‘H’状态。X/Y 方向转换的触摸屏引脚状态。

	XP	XM	YP	YM	ADC 通道选择
X 方向	V _{ref}	GND	高阻	高阻	YP
Y 方向	高阻	高阻	V _{ref}	GND	XP

（翻译）

1. While waiting for Touch screen Interrupt, XP_SEN bit should be set to ‘1’(XP Output disable) and PULL_UP bit should be set to ‘0’(XP Pull-up enable).

当进入等待触摸屏中断的模式时，XP_SEN 必须设置为“1”，并且“PULL_UP”必须设置为“0”。

3. XP, YP should be disconnected with GND source during sleep mode to avoid leakage current. Because XP, YP will be maintained as 'H' states in sleep mode. Touch screen pin conditions in X/Y position conversion.

	XP	XM	YP	YM	ADC ch. select
X Position	Vref	GND	Hi-Z	Hi-Z	YP
Y Position	Hi-Z	Hi-Z	Vref	GND	XP

分离的 X,Y 坐标转换模式时，如何设置那些开关。这个不用设置，因为我们这里选择“自动（连续的）X,Y 坐标转换模式”。

2. AUTO_PST bit should be set ‘1’ only in Automatic & Sequential X/Y Position conversion.

AUTO_PST 自动测量模式应该设置为“1”。（只有在 Touch Screen Interface Mode 第 3 种模式：Auto(Sequential) X/Y Position Conversion Mode 下这个 AUTO_PST 才设置成 1。）

从 NOTE2 的说明看，似乎就是把 AUTO_PST 位设置为“1”，就可以进入“

3. XP, YP should be disconnected with GND source during sleep mode to avoid leakage current. Because XP, YP will be maintained as 'H' states in sleep mode. Touch screen pin conditions in X/Y position conversion.

	XP	XM	YP	YM	ADC ch. select
X Position	Vref	GND	Hi-Z	Hi-Z	YP
Y Position	Hi-Z	Hi-Z	Vref	GND	XP

分离的 X,Y 坐标转换模式时，如何设置那些开关。这个不用设置，因为我们这里选择“自动（连续的）X,Y 坐标转换模式”。所以测试下此位的效果。

还是定义“ADCTSC”的位：

UD_SEN [8] 不用管。

YM_SEN [7]，有 XY 坐标两种模式，会自动的测量，所以：此位设置为“0”或“1”都不对。自动（连续）X,Y 坐标转换模式中 XY 会自动控制此开关。

YM Switch Enable

0 = YM Output Driver Disable.

1 = YM Output Driver Enable.

YP_SEN [6], 自动 (连续) X,Y 坐标转换模式中 XY 会自动控制此开关。

XM_SEN [5], 自动 (连续) X,Y 坐标转换模式中 XY 会自动控制此开关。

XP_SEN [4], 自动 (连续) X,Y 坐标转换模式中 XY 会自动控制此开关。

PULL_UP [3], 上拉使能。这个应该禁止掉。所以 bit3 = 1;

Pull-up Switch Enable

0 = XP Pull-up Enable.

1 = XP Pull-up Disable.

采样 X_ADC 电压, 得到 x 坐标, 等效电路如图 14.6 所示。

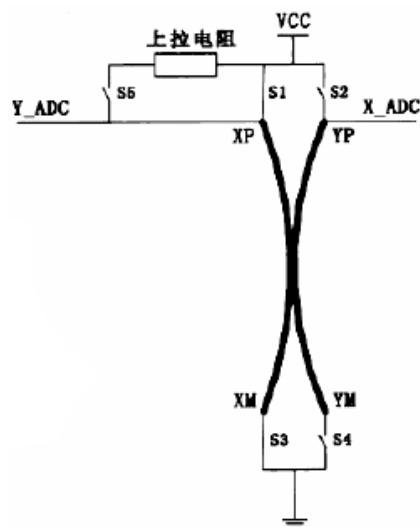


图 14.6 读取 x 坐标时的等效电路

S1、S3 闭合, S2、S4、S5 断开, 即 XP 接上电源、XM 接地、YP 作为模拟输入 (对 CPU 而言)、YM 高阻、XP 禁止上拉。这时, YP 即 X_ADC 就是 x 轴的分压点, 进行 A/D 转换后就得到 x 坐标。

采样 Y_ADC 电压, 得到 y 坐标, 等效电路如图 14.7 所示。

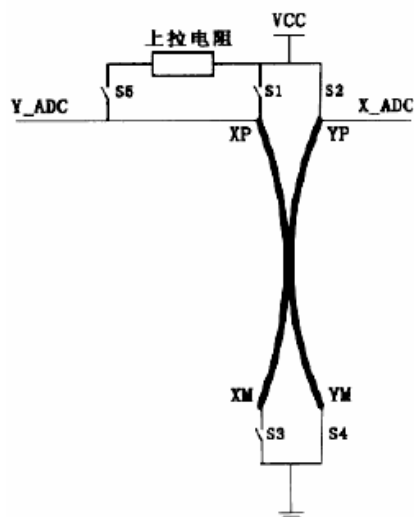


图 14.7 读取 y 坐标时的等效电路

S2、S4 闭合，S1、S3、S5 断开，即 YP 接上电源、YM 接地、XP 作为模拟输入（对 CPU 而言）、XM 高阻、XP 禁止上拉。这时，XP 即 Y_ADC 就是 y 轴的分压点，进行 A/D 转换后就得到 y 坐标。

AUTO_PST [2], 此位设置为“1”（1 = Auto Sequential measurement of X-position, Y-position.）

XY_PST [1:0]。不用管此两位。

结果是：bit3=1, bit2=1.

//3.1.1, 定义测量 XY 坐标模式的函数。

```
static void enter_measure_xy_mode(void)
```

```
{
```

```
    s3c_ts_regs->adctsc = (1<<3) | (1<<2); //ACTSC 的 bit3 设置为 1(禁用上拉使能). bit2 设置为 1.
```

```
}
```

B_2, 启动 ADC 转换：

查看 ADC 控制寄存器：

ADC CONTROL REGISTER (ADCCON)

Register	Address	R/W	Description	Reset Value
ADCCON	0x5800000	R/W	ADC control register	0x3FC4

ENABLE_START	[0]	A/D conversion starts by enable. If READ_START is enabled, this value is not valid. 0 = No operation 1 = A/D conversion starts and this bit is cleared after the start-up.		0
--------------	-----	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--	---

ENABLE_START [0]	启动 AD 操作: 如果 READ_START=1, 则这个值无效。 0=无操作; 1=启动 AD 转换, 启动后该位清零	0
------------------	------------------------------------------------------------------------	---

把 bit0 设置为 1 便可以启动 A D C。

//3.2.1, 定义开始 ADC 转换的函数。

```
static void start_adc(void)
{
    s3c_ts_regs->adcccon |= (1<<0); //ADCCON ADC 控制寄存器 bit0 设置 1 即可启动 ADC。
}
```

综述:

按下触摸笔, 进入“pen_down_up_irq()”中断服务程序:

```
request_irq(IRQ_TC, pen_down_up_irq, IRQF_SAMPLE_RANDOM, "ts_pen", NULL);
```

```
-->irqreturn_t pen_down_up_irq(int irq, void *dev_id)()
```

在这个“pen_down_up_irq()”中断服务程序里面又再次让触摸屏进入“测量 XY 坐标模式”

```
-->enter_measure_xy_mode();
```

再启动 ADC 转换。

```
-->start_adc();
```

B_2, ADC 的启动不可能瞬间完成(不会死等它), 待 ADC 启动完成后, 应该有个中断产生: 得到 XY 坐标电压值

```
irqreturn_t adc_irq(int irq, void *dev_id) //ADC 启动后的中断服务程序。(这里打印 XY 坐标电压值)
```

3. Auto(Sequential) X/Y Position Conversion Mode

Auto (Sequential) X/Y Position Conversion Mode is operated as the following. Touch Screen Controller sequentially converts X-Position and Y-Position that is touched. After Touch controller writes X-measurement data to ADCDAT0 and writes Y-measurement data to ADCDAT1, Touch Screen Interface is generating Interrupt source to Interrupt Controller in Auto Position Conversion Mode.

查看 XY 坐标电压值在哪里。X 坐标的电压值会存放在 ADCDAT0 寄存器, Y 坐标电压值放在 ADCDAT1 寄存器中。

ADC CONVERSION DATA REGISTER (ADCDAT0)

Register	Address	R/W	Description	Reset Value
ADCDAT0	0x580000C	R	ADC conversion data register	—
XPDATA (Normal ADC)	[9:0]		X-Position conversion data value (include normal ADC conversion data value) Data value: 0 ~ 3FF	—

值的范围是 10 位，0~3FF。这个 10 位是 ADCDAT0 这个寄存器的最低 10，即 ADCDAT0 的最低 10 位就是 X 坐标的电压值。这个坐标值是电压的数字表示而已，与 LCD 的 320*240 坐标没有关系。

ADC CONVERSION DATA REGISTER (ADCDAT1)

Register	Address	R/W	Description	Reset Value
ADCDAT1	0x5800010	R	ADC conversion data register	—
YPDATA	[9:0]		Y-position conversion data value Data value: 0 ~ 3FF	—

即 ADCDAT1 的最低 10 位就是 Y 坐标的电压值。

//3.2.1, 实现 ADC 启动完成后进入的中断服务程序，打开 XY 坐标值(电压值)。

```
static irqreturn_t adc_irq(int irq, void *dev_id)
{
    static int cnt = 0;
    //ADCDAT0 寄存的低 10 位就是 X 坐标值. ADCDAT1 低 10 就是 Y 坐标值.
    printk("adc_irq cnt = %d, x = %d, y = %d\n", ++cnt, s3c_ts_regs->adcdat0 &
0x3ff, s3c_ts_regs->adcdat1 & 0x3ff);
    return IRQ_HANDLED;
}
```

编译测试：

```
book@book-desktop:/work/drivers_and_test/11th_ts/3th$ make
make -C /work/system/linux-2.6.22.6 M='pwd' modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
  CC [M] /work/drivers_and_test/11th_ts/3th/s3c_ts.o
/work/drivers_and_test/11th_ts/3th/s3c_ts.c: In function `adc_irq':
/work/drivers_and_test/11th_ts/3th/s3c_ts.c:71: warning: int format, long unsigned int arg (arg 3)
/work/drivers_and_test/11th_ts/3th/s3c_ts.c:71: warning: int format, long unsigned int arg (arg 4)
/work/drivers_and_test/11th_ts/3th/s3c_ts.c: In function `s3c_ts_init':
/work/drivers_and_test/11th_ts/3th/s3c_ts.c:113: warning: ignoring return value of `request_irq', declared with
attribute warn_unused_result
/work/drivers_and_test/11th_ts/3th/s3c_ts.c:114: warning: ignoring return value of `request_irq', declared with
attribute warn_unused_result
/work/drivers_and_test/11th_ts/3th/s3c_ts.c: At top level:
/work/drivers_and_test/11th_ts/3th/s3c_ts.c:37: warning: `enter_wait_pen_up_mode' defined but not used
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /work/drivers_and_test/11th_ts/3th/s3c_ts.mod.o
  LD [M]  /work/drivers_and_test/11th_ts/3th/s3c_ts.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
book@book-desktop:/work/drivers_and_test/11th_ts/3th$
```

```
"
# rmmod s3c_ts
# insmod s3c_ts.ko
input: Unspecified device as /class/input/input1
#
```

卸载原来的驱动，装载现在的驱动。

```
# rmmod s3c_ts
# insmod s3c_ts.ko
input: Unspecified device as /class/input/input1
# adc_irq cnt = 1, x = 490, y = 895
```

按下有值后，再也没其他显示，这是因为按下松开或是划动时没有中断服务程序来处理。测量完后，应该再让它进入所谓的“等待松开模式—enter_wait_pen_up_mode()”这样才能连接操作。

再修改代码为：

//3.2.1, 实现 ADC 启动完成后进入的中断服务程序，打开 XY 坐标值(电压值)。

```
static irqreturn_t adc_irq(int irq, void *dev_id)
{
    static int cnt = 0;
    //ADCDAT0 寄存的低 10 位就是 X 坐标值. ADCDAT1 低 10 就是 Y 坐标值.
    printk("adc_irq cnt = %d, x = %d, y = %d\n", ++cnt, s3c_ts_regs->adcdat0 &
0x3ff, s3c_ts_regs->adcdat1 & 0x3ff);
    enter_wait_pen_up_mode();
    return IRQ_HANDLED;
}
```

```
book@book-desktop:/work/drivers_and_test/11th_ts/3th$ make && cp s3c_ts.ko /work/nfs_root/first_fs
make -C /work/system/linux-2.6.22.6 M=`pwd` modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
  CC [M] /work/drivers_and_test/11th_ts/3th/s3c_ts.o
/work/drivers_and_test/11th_ts/3th/s3c_ts.c: In function `adc_irq':
/work/drivers_and_test/11th_ts/3th/s3c_ts.c:71: warning: int format, long unsigned int arg (arg 3)
/work/drivers_and_test/11th_ts/3th/s3c_ts.c:71: warning: int format, long unsigned int arg (arg 4)
/work/drivers_and_test/11th_ts/3th/s3c_ts.c: In function `s3c_ts_init':
/work/drivers_and_test/11th_ts/3th/s3c_ts.c:114: warning: ignoring return value of `request_irq', declared with
attribute warn_unused_result
/work/drivers_and_test/11th_ts/3th/s3c_ts.c:115: warning: ignoring return value of `request_irq', declared with
attribute warn_unused_result
  Building modules, stage 2.
  MODPOST 1 modules
  LD [M] /work/drivers_and_test/11th_ts/3th/s3c_ts.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
book@book-desktop:/work/drivers_and_test/11th_ts/3th$
```

```
# rmmod s3c_ts
# insmod s3c_ts.ko
input: Unspecified device as /class/input/input2
#
# adc_irq cnt = 1, x = 476, y = 912 按下
pen up 松开
```

按下松开时的情况。


```

adc_irq cnt = 7, x = 285, y = 946
pen up
adc_irq cnt = 8, x = 229, y = 952
pen up
adc_irq cnt = 9, x = 248, y = 949
pen up
adc_irq cnt = 10, x = 18, y = 973
pen up

```

在 X 方向点动（此时还不是划动），这时 Y 坐标电压值变化不大，而 X 坐标电压值变化很大。

```

adc_irq cnt = 67, x = 18, y = 963
pen up
adc_irq cnt = 68, x = 330, y = 667
pen up

```

电压变化比较大，要作些优化。

按着不动，或是划动时并不能打印出来 XY 坐标值。这是要加“定时器”。

TS 工作流程其中的步骤：

4，在 ADC 中断处理函数里面，用 `input_event()` 来上报。

这个过程有个缺点，就是按下后只会启动一次，按下不松开时“粘点”划动，这样就不会新“按下”中断产生。

所以第 4 步里，上报后，启动“定时器”（处理长按、滑动的过程中也可以连续不断的转换这个坐标电压值，上报出来）。

5，定时器时间到，再到第 2 步，再次启动 ADC。就是按下时产生一个“粘点”中断，过了一段时间（定时器），就会再启动一次 ADC 中断，这样就能处理“粘点”滑动的过程。（这个定时器比如为 ms 级，人也反应不了这么快）。

下面的解决的问题：

坐标电压值不够精确。第二按下时不动或划动时并没有处理。

C，按照上面的 TS 工作流程是，精确坐标值，并且处理“按下不动”或“按下划动”时的情况：

C_1, 精确坐标电压值：在

优化措施 1：

触摸屏是利用“欧姆定律”。当按下时立刻就产生中断时，这时候的电压可能并没有稳定下来。这样测量到的电压可能就不准确。所以这里的优化方法就是等这个电压稳定下来时再去测量它。

当电压稳定后，才产生“触摸笔”按下和松开的中断。

这其中的“延时值”用 ADCDLY 寄存器来设置。

ADC START DELAY REGISTER (ADCDLY)

Register	Address	R/W	Description	Reset Value
ADCDLY	0x5800008	R/W	ADC Start or interval delay register	0x00ff

ADCDLY	Bit	Description	Initial State
DELAY	[15:0]	Normal Conversion Mode, XY position mode, auto position mode. → ADC conversion start delay value. Note: Don't use Zero value (0x0000)	00ff

DELAY [15:0] 常规转换模式、XY 坐标自动转换模式、XY 坐标自动转换模式下： 0x00ff
X/Y 坐标转换延时；
等待中断模式下：触摸笔按下以后产生中断的时间间隔；
注意不能使用 0 值

将“ADCDLY”寄存器设置为最大。

C_2, 扔掉不可靠的电压值：在打印 XY 坐标电压值前扔掉。

当一按下，就会产生一个中断 INT_TC，在这个 INT_TC 中断里面启动 ADC 转换，ADC 启动需要一定时间，若在 ADC 启动的过程松开了触摸笔，这时 ADC 转换成功后电压值就会不可靠。所以测量到这样的值要扔掉。

ADC CONVERSION DATA REGISTER (ADCDAT0)

Register	Address	R/W	Description	Reset Value
ADCDAT0	0x580000C	R	ADC conversion data register	—

ADCDAT0	Bit	Description	Initial State
UPDOWN	[15]	Up or Down state of stylus at waiting for interrupt mode. 0 = Stylus down state. 1 = Stylus up state.	—

ADCDAT0 寄存器中，bit15 是判断松开还是按下的状态。

ADC CONVERSION DATA REGISTER (ADCDAT1)

Register	Address	R/W	Description	Reset Value
ADCDAT1	0x5800010	R	ADC conversion data register	—

ADCDAT1	Bit	Description	Initial State
UPDOWN	[15]	Up or down state of stylus at waiting for interrupt mode. 0 = Stylus down state. 1 = Stylus up state.	—

ADCDAT1 寄存器，bit15 也是判断松开还是按下的状态的。

3.2.1, 实现 ADC 启动完成后进入的中断服务程序，打开 XY 坐标值(电压值)。

```
static irqreturn_t adc_irq(int irq, void *dev_id)
{
    static int cnt = 0;

    int adcdat0, adcdat1;
```

```

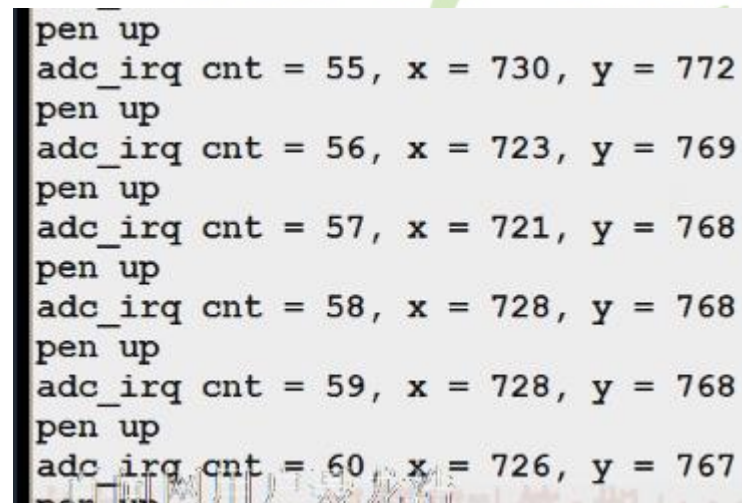
/* 4.2, 优化措施 2: 如果 ADC 完成时, 发现触摸笔已经松开, 则丢弃此次结果 */
adcdat0 = s3c_ts_regs->adcdat0;
adcdat1 = s3c_ts_regs->adcdat1;

    if (s3c_ts_regs->adcdat0 & (1<<15)) //ADCDAT0 bit15=1 时是松开状态
    {
/* 已经松开, 就等待触摸笔按下模式。这时不打印。 */
enter_wait_pen_down_mode();
    }
else //否则就是 ADCDAT0 bit15=0, 是触摸笔按下的状态, 这时打印 XY 坐标值。
{
    //ADCDAT0 寄存的低 10 位就是 X 坐标值. ADCDAT1 低 10 就是 Y 坐标值. 打印完后再
    等待松开模式.
    printk("adc_irq cnt = %d, x = %d, y = %d\n", ++cnt, adcdat0 & 0x3ff,
adcdat1 & 0x3ff);
    enter_wait_pen_up_mode();
}

return IRQ_HANDLED;
}

```

接着编译测试:



```

pen up
adc_irq cnt = 55, x = 730, y = 772
pen up
adc_irq cnt = 56, x = 723, y = 769
pen up
adc_irq cnt = 57, x = 721, y = 768
pen up
adc_irq cnt = 58, x = 728, y = 768
pen up
adc_irq cnt = 59, x = 728, y = 768
pen up
adc_irq cnt = 60, x = 726, y = 767

```

在同一个点不断按下松开, 发现电压值变化不大。可以多换几个点, 基本保持着稳定。

C_3, 优化措施 3:

多次的测量值, 再得到一个平均值可能更精确点。

//3.2.1, 实现 ADC 启动完成后进入的中断服务程序, 打开 XY 坐标值(电压值)。

```

static irqreturn_t adc_irq(int irq, void *dev_id)
{
    static int cnt = 0;

```

```

static int x[4], y[4]; //XY 坐标电压值都测量 4 次。
int adcdat0, adcdat1;

/* 4.2, 优化措施 2: 如果 ADC 完成时, 发现触摸笔已经松开, 则丢弃此次结果 */
adcdat0 = s3c_ts_regs->adcdat0;
adcdat1 = s3c_ts_regs->adcdat1;

if (s3c_ts_regs->adcdat0 & (1<<15)) //ADCDAT0 bit15=1 时是松开状态
{
/* 已经松开, 就等待触摸笔按下模式。这时不打印. */
    cnt = 0;
    enter_wait_pen_down_mode();
}
else //否则就是 ADCDAT0 bit15=0, 是触摸笔按下的状态, 这时打印 XY 坐标值。
{
    //4.4, 优化措施 3, 多次测量的求平均值。 printf("adc_irq cnt = %d, x
    = %d, y = %d\n", ++cnt, adcdat0 & 0x3ff, adcdat1 & 0x3ff);
    /* 优化措施 3: 多次测量求平均值 */
    x[cnt] = adcdat0 & 0x3ff; //测量后先把值放在这个静态变量的数组中。
    y[cnt] = adcdat1 & 0x3ff;
    ++cnt;
    if(cnt == 4) //要是测量的值已记数了 4 次就打印结果。
    {
        //ADCDAT0 寄存的低 10 位就是 X 坐标值, ADCDAT1 低 10 就是 Y 坐标值, 打印完
        后再等待松开模式。
        printf("x = %d, y = %d\n", (x[0]+x[1]+x[2]+x[3])/4,
        (y[0]+y[1]+y[2]+y[3])/4);
        cnt = 0; //再复位 cnt 值。
        enter_wait_pen_up_mode();
    }
    else //否则让它再测量一次,
    {
        enter_measure_xy_mode(); //进入测量 XY 坐标值模式
        start_adc(); //再次启动 ADC。总之是要得到上面定的 4 次的测量值。
    }
}

return IRQ_HANDLED;
}

再次编译后测量, 下面的值更精确了。

```



```

x = 691, y = 725
pen up
x = 690, y = 724
pen up
x = 692, y = 721
pen up
x = 693, y = 724
pen up
x = 694, y = 724
pen up
x = 697, y = 722
pen up
x = 696, y = 721
pen up
x = 694, y = 721
pen up
x = 694, y = 718

```

C__4，代码措施4：软件过滤。过滤成功时就打印否则不打印。

```

static int s3c_filter_ts(int x[], int y[])
{
#define ERR_LIMIT 10 //若定义了误差是10.这是一个经验值。

int avr_x, avr_y; //4次测量值，两两的平均值。
int det_x, det_y; //误差值。

avr_x = (x[0] + x[1])/2; //取平均值
avr_y = (y[0] + y[1])/2;

det_x = (x[2] > avr_x) ? (x[2] - avr_x) : (avr_x - x[2]); //求出误差值。
det_y = (y[2] > avr_y) ? (y[2] - avr_y) : (avr_y - y[2]);

if ((det_x > ERR_LIMIT) || (det_y > ERR_LIMIT)) //若误差大过 ERR_TIMET 就认为错
直接返回。
return 0;

avr_x = (x[1] + x[2])/2;
avr_y = (y[1] + y[2])/2;

det_x = (x[3] > avr_x) ? (x[3] - avr_x) : (avr_x - x[3]);
det_y = (y[3] > avr_y) ? (y[3] - avr_y) : (avr_y - y[3]);

if ((det_x > ERR_LIMIT) || (det_y > ERR_LIMIT))

```

```

return 0;

return 1;
}

static int s3c_filter_ts(int x[], int y[])
{
#define ERR_LIMIT 10 //若定义了误差是 10. 这是一个经验值。

int avr_x, avr_y; //4 次测量值，两两的平均值。
int det_x, det_y; //误差值。

avr_x = (x[0] + x[1])/2; //取平均值
avr_y = (y[0] + y[1])/2;

det_x = (x[2] > avr_x) ? (x[2] - avr_x) : (avr_x - x[2]); //求出误差值。
det_y = (y[2] > avr_y) ? (y[2] - avr_y) : (avr_y - y[2]);

if ((det_x > ERR_LIMIT) || (det_y > ERR_LIMIT)) //若误差大过 ERR_LIMIT 就认为错
直接返回。
return 0;

avr_x = (x[1] + x[2])/2;
avr_y = (y[1] + y[2])/2;

det_x = (x[3] > avr_x) ? (x[3] - avr_x) : (avr_x - x[3]);
det_y = (y[3] > avr_y) ? (y[3] - avr_y) : (avr_y - y[3]);

if ((det_x > ERR_LIMIT) || (det_y > ERR_LIMIT))
return 0;

return 1;
}

```

```

x = 600, y = 572
pen up
x = 602, y = 573
pen up
x = 599, y = 567
pen up
x = 596, y = 562
pen up
x = 597, y = 559
pen up
x = 593, y = 553
pen up
x = 593, y = 548
pen up
x = 592, y = 547
pen up

```

在同一个点上的测量效果，相差更小了。X坐标电压值没怎么变。
再编测试

C__5，最后一个优化：触摸笔长时间按下或是按下划动时的情况。要用到定时器。时间到了处理。

//5.1, 定义定时器。

```
static struct timer_list ts_timer;
```

定时器两要素：

超时时间：

超时后的处理函数：

```

# rmmod s3c_ts
# insmod s3c_ts.ko
Module len 4096 truncated
insmod: cannot insert 's3c_ts.ko': Invalid module format (-1): Exec format error
# █

```

加载时遇到这个问题，就重新编译（make clean 后）。说是文件被截断。

```
x = 236, y = 627
x = 236, y = 627
x = 236, y = 627
x = 236, y = 627
x = 236, y = 627
x = 235, y = 627
x = 235, y = 627
x = 235, y = 628
x = 235, y = 628
x = 233, y = 630
x = 231, y = 632
x = 229, y = 634
x = 227, y = 635
x = 226, y = 637
x = 224, y = 638
x = 221, y = 642
```

长按或滑动可以看到上面不断的打印 XY 坐标电压值。

五，上报事件：完整触摸屏驱动程序。

这时把里面的“printk()”改成“上报事件”即可。

1, s3c_2410_ts.c 中的上报事件操作如下：

```
input_report_abs(ts.dev, ABS_X, ts.xp);
input_report_abs(ts.dev, ABS_Y, ts.yp);

input_report_key(ts.dev, BTN_TOUCH, 1);
input_report_abs(ts.dev, ABS_PRESSURE, 1);
input_sync(ts.dev);
```

“input_report_abs()”的实质还是“input_event()”：

```
static inline void input_report_abs(struct input_dev *dev, unsigned int code, int value)
{
    input_event(dev, EV_ABS, code, value);
}
```



最后编译和测试。

- 1, 先查看有哪个 event 设备节点: `ls /dev/event*`
- 2, `insmod s3c_ts.ko` 。
- 3, 再查看设备节点: `ls /dev/event*`
- 4, `hexdump /dev/event*` 。 先加载的是触摸屏时, `dev/event0` 就对应于触摸屏。若先加载的是 LCD 则 `/dev/event0` 就对应于 LCD。

	秒	微秒	type	code	value
0000000	29a4	0000	8625	0008	0003 0000 0172 0000
0000010	29a4	0000	8631	0008	0003 0001 027c 0000
0000020	29a4	0000	8634	0008	0003 0018 0001 0000
0000030	29a4	0000	8638	0008	0001 014a 0001 0000
0000040	29a4	0000	863c	0008	0000 0000 0000 0000 (同步事件)
0000050	29a4	0000	c85e	0008	0003 0000 0171 0000
0000060	29a4	0000	c874	0008	0003 0001 027d 0000
0000070	29a4	0000	c87b	0008	0000 0000 0000 0000
0000080	29a4	0000	ed37	0008	0003 0018 0000 0000
0000090	29a4	0000	ed48	0008	0001 014a 0000 0000
00000a0	29a4	0000	ed4a	0008	0000 0000 0000 0000

0003 -- 绝对位移。

Code 0000 表示 X 方向 (ABS_X)。0001 表示 Y 方向 (ABS_Y)，0018 表示压力 (ABS_PRESSURE)。

Value 0172 0000 就是 X 坐标值。

Code 014a 是指哪个按键 “BTN_TOUCH”。

使用：

想用的触摸屏，就是想点击 LCD 屏上显示的内容，并对应起来。要用到库 “tslib”
编译：

```
tar xzf tslib-1.4.tar.gz
cd tslib
./autogen.sh
```

`mkdir tmp` 这个临时目录以后放他的编译结果。

```
echo "ac_cv_func_malloc_0_nonnull=yes" >arm-linux.cache
./configure --host=arm-linux --cache-file=arm-linux.cache --prefix=$(pwd)/tmp
make
```

make install 这个安装就是把编译的结果安装到“临时目录”tmp 中。

```
book@book-desktop:/work/drivers_and_test/11th_ts/tslib$ cd tmp/
book@book-desktop:/work/drivers_and_test/11th_ts/tslib/tmp$ ls
bin etc include lib
book@book-desktop:/work/drivers_and_test/11th_ts/tslib/tmp$ ls bin/
ts_calibrate ts_harvest ts_print ts_print_raw ts_test
book@book-desktop:/work/drivers_and_test/11th_ts/tslib/tmp$

book@book-desktop:/work/drivers_and_test/11th_ts/tslib/tmp$ ls etc/
ts.conf
book@book-desktop:/work/drivers_and_test/11th_ts/tslib/tmp$ ls include/
tslib.h
book@book-desktop:/work/drivers_and_test/11th_ts/tslib/tmp$ ls lib/
libts-0.0.so.0 libts-0.0.so.0.1.1 libts.la libts.so pkgconfig ts
book@book-desktop:/work/drivers_and_test/11th_ts/tslib/tmp$
```

安装：

cd tmp

cp * -rf /nfsroot （这只是一个示意）就是说要把文件拷贝到开发板的“根”目录下。就是把相关文件归到开发板的根文件系统的“bin”，“etc”，“lib”和“include”目录下。

```
book@book-desktop:/work/drivers_and_test/11th_ts/tslib$ cp tmp /work/nfs_root/first_fs/ts_dir -rfd
book@book-desktop:/work/drivers_and_test/11th_ts/tslib$
```

-rfd:r 递归，f 强制，d，保持链接。

```
# cd /mnt/ts_dir/
# ls
bin      etc      include  lib
# cp * / -rfd
#
```

使用：安装驱动。

```
1nfs 30000000 192.168.1.5:/work/nfs_root/uImage_notes
2bootm 30000000
3mount -t nfs -o nolock,vers=2 192.168.1.5:/work/nfs_root/first_fs /mnt
4
```

UBOOT 中从 NFS 下载新的内核 uImage。

第二步是 bootm 30000000

第三步是 mount -t nfs -o nolock,vers=2 192.168.。。。。。。。。。

先安装 s3c_ts.ko, lcd.ko

```
# cd /mnt/
# insmod cfb
cfbcopyarea.ko cfbfillrect.ko cfbimgblt.ko
# insmod cfbcopyarea.ko
# insmod cfbfillrect.ko
# insmod cfbimgblt.ko
# insmod lcd.ko
Segmentation fault
#
```

段错误。

这个错误可能和重新编译过内核有关，重新编译过内核后，就来再次编译下驱动程序，将在重新编译过的内核中重新编译过的驱动模块再来安装。

```
Please press Enter to activate this console.
starting pid 761, tty '/dev/s3c2410_serial0': '/bin/sh'
# mount -t nfs -o nolock,vers=2 192.168.1.5:/work/nfs_root/first_fs /mnt
# cd /mnt/
# insmod cfbcopyarea.ko
# insmod cfbimgblt.ko
# insmod cfb
cfbcopyarea.ko cfbfillrect.ko cfbimgblt.ko
# insmod cfbimgblt.ko
insmod: cannot insert 'cfbimgblt.ko': File exists (-1): File exists
# insmod cfbfillrect.ko
# insmod lcd.ko
Console: switching to colour frame buffer device 30x40
# ls /dev/ev*
ls: /dev/ev*: No such file or directory
# insmod s3c_ts.ko
input: Unspecified device as /class/input/input0
# ls /dev/ev*
/dev/event0
# ls /dev/fb0
/dev/fb0
#
```

1.

修改 /etc/ts.conf 第 1 行(去掉#号和第一个空格):

```
# module_raw input
```

改为:

```
module_raw input
```

```
# Uncomment if you wish to use the linux input layer event interface
module_raw input      前面的"# "-(#和空格)都要去掉

# Uncomment if you're using a Sharp Zaurus SL-5500/SL-5000d
# module_raw collie

# Uncomment if you're using a Sharp Zaurus SL-C700/C750/C760/C860
# module_raw corgi

# Uncomment if you're using a device with a UCB1200/1300/1400 TS interface
# module_raw ucb1x00

# Uncomment if you're using an HP iPaq h3600 or similar
# module_raw h3600

# Uncomment if you're using a Hitachi Webpad
# module_raw mk712

# Uncomment if you're using an IBM Arctic II
# module_raw arctic2

module_ptthres pmin=1
module_variance delta=30
- /etc/ts.conf [Modified] 2/25 8%
```

2. 触摸屏的程序是打开的文件

，可以通过环境变量来设置：先设置环境变量后，才能使用触摸屏程序。

```
export TSLIB_TSDEVICE=/dev/event0      //这是指触摸屏设备。
export TSLIB_CALIBFILE=/etc/pointercal //校验文件放在这里。
export TSLIB_CONFFILE=/etc/ts.conf     //配置文件放在这里。
export TSLIB_PLUGINDIR=/lib/ts         //插件放在这里
export TSLIB_CONSOLEDEVICE=none
export TSLIB_FBDEVICE=/dev/fb0        //显示屏
```

ts_calibrate //是指校验，在 SHELL 中输入这个程序后：

这时屏幕上会出现一些文字，左上角会有个“十字架”。

```
# ts_calibrate
xres = 240, yres = 320
Took 3 samples...
Top left : X = 713 Y = 806
Took 3 samples...
Top right : X = 231 Y = 819
Took 4 samples...
Bot right : X = 235 Y = 230
Took 4 samples...
Bot left : X = 719 Y = 230
Took 3 samples...
Center : X = 477 Y = 524
258.986328 -0.289881 -0.002477
359.473877 -0.005062 -0.377672
Calibration constants: 16972928 -18997 -162 23558480 -331 -24751 65536
#
```

输入 ts_calibrate 是指等待校验。
接着就是上下左右等的十字坐标的校验。

校验成功后生成下面这个校验文件：

```
# ls /etc/pointercal
/etc/pointercal
# ts
```

ts_test //在 SHELL 上输入这个命令，就可以测试了。这时点一下鼠标或滑动一下，触摸屏上的光标就会随着走。

225.510603:	157	211	1
225.530603:	158	211	1
225.610600:	161	207	1
225.610600:	162	203	1
225.710591:	165	201	1
225.750598:	170	198	1
225.770615:	173	197	1
225.790606:	176	197	1
225.830595:	178	197	1
225.850603:	180	197	1
225.870605:	183	199	1
225.890603:	186	199	1
225.910603:	188	200	0

还有其他一些测试程序：

```
# ts_
ts_cal.sh      ts_harvest    ts_print_raw
ts_calibrate  ts_print      ts_test
```

Ts_print:打印坐标。

# ts_print			
240.617177:	38	197	1
240.635592:	37	197	1
240.665051:	37	197	0
243.573941:	16	50	1
243.590595:	16	51	1
243.610607:	16	51	1
243.616056:	16	51	0

就是把 XY 坐标的电压值转换成 LCD 的 320*240 的坐标值。后面“1”表示按下，“0”表示松开的意思。

```
# ts_print_raw
284.265601:      795      784      1
284.285619:      794      783      1
284.311056:         0         0      0
284.311058:      794      783      0
285.150598:      650      742      1
285.196050:         0         0      0
285.196053:      650      742      0
285.389253:      635      740      1
```

这是打印原始数据，就是 XY 坐标的电压值。（上报的 ADC 的值）。

