

Linux下I2C接口触摸屏驱动分析

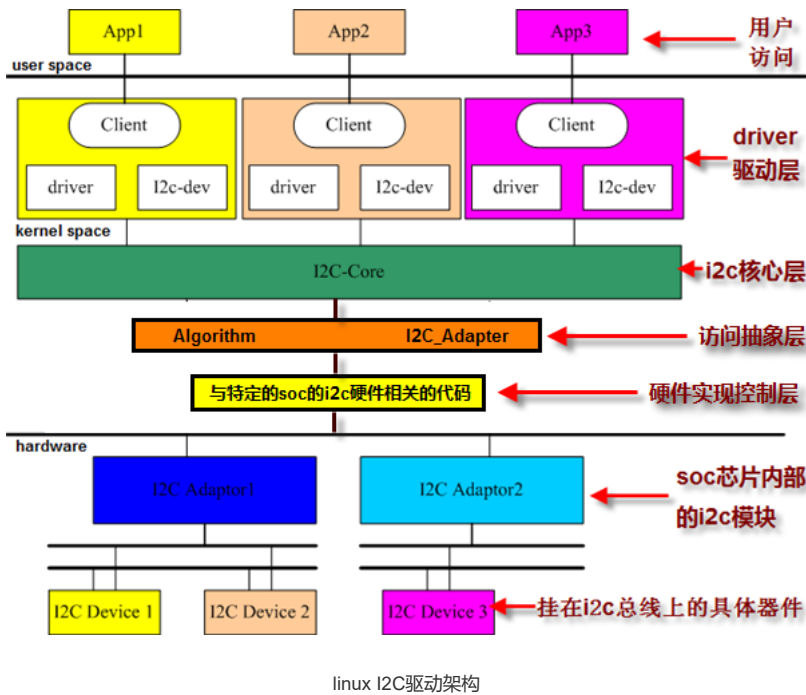
2016年06月06日 19:48:50

2279

linux下触摸屏驱动的移植主要包括这几个步骤：

- ...定触摸屏IC接口，了解对应接口的API函数，注册设备并加入到相应总线上
- ...联设备与驱动，并编写具体的驱动代码
- ...悉linux 输入设备驱动，在驱动代码中分配一个输入设备并初始化相应数据结构，在驱动实现中引用
- ...对应上面几部分，分析I2C接口下触摸屏驱动的实现。先介绍linux下I2C接口与输入子系统架构，然后基于F
- ...码逐层展开整个移植过程的实现。

一、I2C驱动架构



linux I2C驱动架构

具体的实现可分为四个层次：

1、提供adapter的硬件驱动，探测、初始化i2c adapter（如申请i2c的io地址和中断号），驱动soc控制的i2c adapter在硬件上产生信号（start、stop、ack）以及处理i2c中断。覆盖图中的硬件实现层。主要数据结构struct i2c_adapter，这里adapter的驱动在文件/drivers/i2c/busses/i2c-omap.c中

```
[cpp]
01. struct i2c_adapter {
02.     struct module *owner;
03.     unsigned int id;
04.     unsigned int class;
05.     struct i2c_algorithm *algo; /* the algorithm to access the bus */
06.     void *algo_data;
07.
08.     /* --- administration stuff. */
09.     int (*client_register)(struct i2c_client *);
10.     int (*client_unregister)(struct i2c_client *);
11.
12.     /* data fields that are valid for all devices */
13.     struct mutex bus_lock;
14.     struct mutex clist_lock;
15.
16.     int timeout;
17.     int retries;
18.     struct device dev; /* the adapter device */
19.     struct class_device class_dev; /* the class device */
20.
21.     int nr;
```

水木无痕

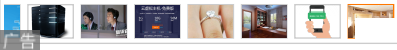
关注

原创	粉丝	喜欢	评论
172	23	8	11

等级： 博客 5 访问量： 24万+
积分： 4271 排名： 8658



四十平小户型装修



他的最新文章 更多文章

- struct security_operations {}函数结构
为设备服务的文件系统sysfs--kobject和kset的关系
为设备服务的文件系统sysfs--sysfs文件的读写
为设备服务的文件系统sysfs--sysfs文件的打开操作
为设备服务的文件系统sysfs--文件目录的创建

文章分类

C语言	7篇
内核	8篇
驱动	8篇
杂	2篇
通信 网络	3篇
视频	2篇

展开

文章存档

2017年10月	22篇
2017年9月	25篇
2017年8月	5篇
2017年7月	2篇
2017年6月	25篇
2017年5月	

展开

```

22.     struct list_head clients;
23.     struct list_head list;
24.     char name[I2C_NAME_SIZE];
25.     struct completion dev_released;
26.     struct completion class_dev_released;
27. };

```

这个结构体对应一个控制器。其中包含了控制器名称，algorithm数据，控制器设备等。



块加载后，会调用驱动的probe函数寻找对应的adapter完成驱动与adapter的匹配。这里的probe函数为

```

[cpp]
static int __devinit omap_i2c_probe(struct platform_device *pdev)
{
    struct omap_i2c_dev *dev;
    struct i2c_adapter *adap;
    struct resource *mem, *irq, *ioarea;
    struct omap_i2c_bus_platform_data *pdata = pdev->dev.platform_data;
    irq_handler_t isr;
    int r;
    u32 speed = 0;

    /* NOTE: driver uses the static register mapping */
    mem = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    if (!mem) {
        dev_err(&pdev->dev, "no mem resource?\n");
        return -ENODEV;
    }
    irq = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
    if (!irq) {
        dev_err(&pdev->dev, "no irq resource?\n");
        return -ENODEV;
    }

    ioarea = request_mem_region(mem->start, resource_size(mem),
                                pdev->name);
    if (!ioarea) {
        dev_err(&pdev->dev, "I2C region already claimed\n");
        return -EBUSY;
    }

    dev = kzalloc(sizeof(struct omap_i2c_dev), GFP_KERNEL);
    if (!dev) {
        r = -ENOMEM;
        goto err_release_region;
    }

    if (pdata != NULL) {
        speed = pdata->clkrate;
        dev->set_mpu_wkup_lat = pdata->set_mpu_wkup_lat;
    } else {
        speed = 100; /* Default speed */
        dev->set_mpu_wkup_lat = NULL;
    }

    dev->speed = speed;
    dev->dev = &pdev->dev;
    dev->irq = irq->start;
    dev->base = ioremap(mem->start, resource_size(mem));
    if (!dev->base) {
        r = -ENOMEM;
        goto err_free_mem;
    }

    platform_set_drvdata(pdev, dev);

    dev->reg_shift = (pdata->flags >> OMAP_I2C_FLAG_BUS_SHIFT__SHIFT) & 3;

    if (pdata->rev == OMAP_I2C_IP_VERSION_2)
        dev->regs = (u8 *)reg_map_ip_v2;
    else
        dev->regs = (u8 *)reg_map_ip_v1;

    pm_runtime_enable(dev->dev);
    pm_runtime_get_sync(dev->dev);

    dev->rev = omap_i2c_read_reg(dev, OMAP_I2C_REV_REG) & 0xff;

    if (dev->rev <= OMAP_I2C_REV_ON_3430)
        dev->errata |= I2C_OMAP3_IP153;

    if (!(pdata->flags & OMAP_I2C_FLAG_NO_FIFO)) {
        u16 s;

        /* Set up the fifo size - Get total size */

```

他的热门文章

JAVA 点击按钮后跳到另一个界面

17659

安装mysql出现* Could't find MySQL s
erver (/usr/bin/mysqld_safe)

14452

如何在shell脚本中，判断一个基本命令执
行是否成功？

9499

inet addr、bcast、mask

6848

/usr/bin/env: php: No such file or dire
ctory

4876

在shell中,拼接一个字符串,形成一条命令

4809

取模和求余的区别

4068

如何检验mysql安装好了没

3799

嵌入式系统分为硬件层、驱动层、从做系
统层、应用层4层

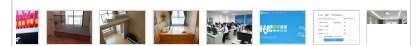
3698

Linux下Apache与httpd的区别与关系

3594



榻榻米床好不好



联系我们



请扫描二维码联系客服

✉ webmaster@csdn.net

☎ 400-660-0108

👤 QQ客服 🗣 客服论坛

关于 招聘 广告服务 百度

©1999-2018 CSDN版权所有

京ICP证09002463号

经营性网站备案信息

网络110报警服务

中国互联网举报中心

北京互联网违法和不良信息举报中心

```

74.         s = (omap_i2c_read_reg(dev, OMAP_I2C_BUFSTAT_REG) >> 14) & 0x3;
75.         dev->fifo_size = 0x8 << s;
76.
77.         /*
78.          * Set up notification threshold as half the total available
79.          * size. This is to ensure that we can handle the status on int
80.          * call back latencies.
81.          */
82.         dev->fifo_size = (dev->fifo_size / 2);
83.
84.         if (dev->rev >= OMAP_I2C_REV_ON_3530_4430)
85.             dev->b_hw = 0; /* Disable hardware fixes */
86.         else
87.             dev->b_hw = 1; /* Enable hardware fixes */
88.
89.         /* calculate wakeup latency constraint for MPU */
90.         if (dev->set_mpu_wkup_lat != NULL)
91.             dev->latency = (1000000 * dev->fifo_size) /
92.                             (1000 * speed / 8);
93.     }
94.
95.     /* reset ASAP, clearing any IRQs */
96.     omap_i2c_init(dev);
97.
98.     isr = (dev->rev < OMAP_I2C_OMAP1_REV_2) ? omap_i2c_omap1_isr :
99.                                                omap_i2c_isr;
100.
101.     r = request_irq(dev->irq, isr, 0, pdev->name, dev);
102.
103.     if (r) {
104.         dev_err(dev->dev, "failure requesting irq %i\n", dev->irq);
105.         goto err_unuse_clocks;
106.     }
107.
108.     dev_info(dev->dev, "bus %d rev%d.%d.%d at %d kHz\n", pdev->id,
109.              pdev->rev, dev->rev >> 4, dev->rev & 0xf, dev->speed);
110.
111.     pm_runtime_put(dev->dev);
112.
113.     adap = &dev->adapter;
114.     i2c_set_adapdata(adap, dev);
115.     adap->owner = THIS_MODULE;
116.     adap->class = I2C_CLASS_HWMON;
117.     strcpy(adap->name, "OMAP I2C adapter", sizeof(adap->name));
118.     adap->algo = &omap_i2c_algo;
119.     adap->dev.parent = &pdev->dev;
120.
121.     /* i2c device drivers may be active on return from add_adapter() */
122.     adap->nrx = pdev->id;
123.     r = i2c_add_numbered_adapter(adap);
124.     if (r) {
125.         dev_err(dev->dev, "failure adding adapter\n");
126.         goto err_free_irq;
127.     }
128.
129.     return 0;
130.
131. err_free_irq:
132.     free_irq(dev->irq, dev);
133. err_unuse_clocks:
134.     omap_i2c_write_reg(dev, OMAP_I2C_CON_REG, 0);
135.     pm_runtime_put(dev->dev);
136.     iounmap(dev->base);
137. err_free_mem:
138.     platform_set_drvdata(pdev, NULL);
139.     kfree(dev);
140. err_release_region:
141.     release_mem_region(mem->start, resource_size(mem));
142.
143.     return r;
144. }

```

2、提供adapter的algorithm，用具体适配器的xxx_xfer()函数来填充i2c_algorithm的master_xfer函数指针，并把赋值后的i2c_algorithm再赋值给i2c_adapter的algo指针。覆盖图中的访问抽象层、i2c核心层。主要数据结构struct i2c_algorithm

这个结构体中定义了一套控制器使用的通信方法。其中关键函数是master_xfer(), 实现了在物理层面上数据传输的过程。omap平台的实现函数为omap_i2c_xfer()

```
[cpp]
01. omap_i2c_xfer(struct i2c_adapter *adap, struct i2c_msg msgs[], int num)
02. {
03.     struct omap_i2c_dev *dev = i2c_get_adapdata(adap);
04.     int i;
05.     int r;
06.
07.     pm_runtime_get_sync(dev->dev);
08.
09.     r = omap_i2c_wait_for_bb(dev);
10.     if (r < 0)
11.         goto out;
12.
13.     if (dev->set_mpu_wkup_lat != NULL)
14.         dev->set_mpu_wkup_lat(dev->dev, dev->latency);
15.
16.     for (i = 0; i < num; i++) {
17.         r = omap_i2c_xfer_msg(adap, &msgs[i], (i == (num - 1)));
18.         if (r != 0)
19.             break;
20.     }
21.
22.     if (dev->set_mpu_wkup_lat != NULL)
23.         dev->set_mpu_wkup_lat(dev->dev, -1);
24.
25.     if (r == 0)
26.         r = num;
27.
28.     omap_i2c_wait_for_bb(dev);
29. out:    pm_runtime_put(dev->dev);
30.
31.     return r;
32. }
```

I2C消息传输方法

可以看到, 最终用于传输的是omap_i2c_xfer_msg()函数, 函数定义在drivers/i2c/busses/i2c-omap.c文件中, 具体的实现可以先不管。

3、实现i2c设备驱动中的i2c_driver接口, 由结构i2c_client中的数据填充, 覆盖图中的driver驱动层。主要数据结构

```
[cpp]
01. struct i2c_client {
02.     unsigned int flags; /* div., see below */
03.     unsigned short addr; /* chip address - NOTE: 7bit */
04.     /* addresses are stored in the */
05.     /* _LOWER_ 7 bits */
06.     struct i2c_adapter *adapter; /* the adapter we sit on */
07.     struct i2c_driver *driver; /* and our access routines */
08.     int usage_count; /* How many accesses currently */
09.     /* to the client */
10.     struct device dev; /* the device structure */
11.     struct list_head list;
12.     char name[I2C_NAME_SIZE];
13.     struct completion released;
14. };
```


这个结构体中的内容是描述设备的。包含了芯片地址, 设备名称, 设备使用的中断号, 设备所依附的控制器, 设备所依附的驱动等内容。

4、实现i2c设备所对应的具体device的驱动。覆盖图中的driver驱动层。主要数据结构struct i2c_driver

```
[cpp]
01. struct i2c_driver {
02.     int id;
03.     unsigned int class;
04.
05.     int (*attach_adapter)(struct i2c_adapter *);
06.     int (*detach_adapter)(struct i2c_adapter *);
07.
08.     int (*detach_client)(struct i2c_client *);
09.
10.     int (*command)(struct i2c_client *client, unsigned int cmd, void *arg);
11.     struct device_driver driver;
12.     struct list_head list;
13. };
```

这个结构体对应了驱动方法，重要成员函数有probe，remove，suspend，resume与中断处理函数，也是我们需要实现的函数。另外包括一个重要的数据结构: struct i2c_device_id *id_table; 如果驱动可以支持好几个设备，那么这里面就要包含这些设备的ID。

第一层和第二层是i2c总线驱动，属于芯片内部的驱动，在linux驱动架构中已经实现，不需要我们编写或更改。第三第四属于i2c设备驱动，需要我们根据内核提供的接口去完成具体的代码。

 就是i2c_core层，起到了承上启下的作用，包含在开发中需要用到的核心函数。源代码位于drivers/i2c/i2c-core.c中。在这里可以看到几个重要的函数。

 加/删除i2c控制器的函数

[...]

```
01. int i2c_add_adapter(struct i2c_adapter *adapter)
02. int i2c_del_adapter(struct i2c_adapter *adap)
```

(2)增加/删除i2c设备的函数

[cpp]

```
01. struct i2c_client *i2c_new_device(struct i2c_adapter *adap, struct i2c_board_info const
    *info);
02. void i2c_unregister_device(struct i2c_client *client)
```

(3)增加/删除设备驱动的函数

[cpp]

```
01. int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
02. void i2c_del_driver(struct i2c_driver *driver)
```

(4)i2c传输、发送和接收函数

[cpp]

```
01. int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num);
02. int i2c_master_send(struct i2c_client *client, const char *buf, int count);
03. int i2c_master_recv(struct i2c_client *client, char *buf, int count);
```

send和receive分别都调用了transfer函数，而transfer也不是直接和硬件交互，而是调用algorithm中的master_xfer()函数。

二、linux input输入子系统架构

linux输入系统框架

linux输入子系统（linux input subsystem）从上到下由三层实现，分别为：输入子系统事件处理层（Event Handler）、输入子系统核心层（Input Core）和输入子系统设备驱动层。在图中可以看出输入设备不止包括触摸屏，还有很多其他种类的输入设备，如键盘，鼠标等等，所以linux内核已经把各式各样的输入设备进行了抽象，提供了统一的接口，让我们把设备接入到系统中。而我们要做的工作就是申请一个输入设备结构体对象，并填充其里面的数据。

1，输入子系统事件处理层：与userspace衔接，提供处理设备事件的接口，以模块编译进内核，数据结构为input_handler

[cpp]

```
01. struct input_handler {
02.
03.     void *private;
04.
05.     void (*event)(struct input_handle *handle, unsigned int type, unsigned int code,
int value);
06.     bool (*filter)(struct input_handle *handle, unsigned int type, unsigned int code,
int value);
07.     bool (*match)(struct input_handler *handler, struct input_dev *dev);
08.     int (*connect)(struct input_handler *handler, struct input_dev *dev, const struct input_device_id *id);
09.     void (*disconnect)(struct input_handle *handle);
10.     void (*start)(struct input_handle *handle);
11.
12.     const struct file_operations *fops;
13.     int minor;
14.     const char *name;
15.
16.     const struct input_device_id *id_table;
```

```

17.
18.         struct list_head      h_list;
19.
20.         struct list_head      node;
21.     };

```

其中，struct input_device_id在后面讲到handler与input_dev匹配时将用到，还分别用到match与connect函数，具体过程下面再说。

👍 t file_operations提供用户空间的访问方法，这里的evdev形式的fops为：

```

0
[cpp]
static const struct file_operations evdev_fops = {
02.     .owner          = THIS_MODULE,
    .read             = evdev_read,
    .write            = evdev_write,
05.     .poll           = evdev_poll,
06.     .open           = evdev_open,
07.     .release        = evdev_release,
08.     .unlocked_ioctl = evdev_ioctl,
09. #ifdef CONFIG_COMPAT
10.     .compat_ioctl   = evdev_ioctl_compat,
11. #endif
12.     .fasync          = evdev_fasync,
13.     .flush           = evdev_flush,
14.     .llseek          = no_llseek,
15. };

```

event函数记录输入事件的值，驱动input_report_xxx()上报的值最终会通过这个函数保存到内核区，供用户空间访问，存储这个数值的数据结构为input_event。

```

[cpp]
01. struct input_event {
02.     struct timeval time;
03.     __u16 type;
04.     __u16 code;
05.     __s32 value;
06. };

```

2.输入子系统核心层: 提供构建输入设备核心方法实现与数据结构定义，在文件input.h与input.c文件，包括

(1)注册/注销设备

```

input_register_device()
input_unregister_device()

```

(2)注册/注销handler

```

input_register_handler()
input_unregister_handler()

```

(3)注册注销handle

```

input_register_handle()
input_unregister_handle()

```

handle是device与handler链接的纽带。其定义为

```

[cpp]
01. <pre name="code" class="cpp">struct input_handle {
02.
03.     void *private;
04.
05.     int open;
06.     const char *name;
07.
08.     struct input_dev *dev;
09.     struct input_handler *handler;
10.
11.     struct list_head      d_node;
12.     struct list_head      h_node;
13. };</pre>
14. <pre></pre>
15. 详细过程如图所示:

```

```

16. <p></p>
17. <p align="left"><br>
18. </p>
19. <p align="left"> 当新生成一个input_dev时, 需要匹配一个input_handler, 即特定输入设备相应的处理方法, 使用input_attach_handler()<br>
20. </p>
21. <p align="left"></p>
22. <pre name="code" class="cpp"><pre name="code" class="cpp">static int input_attach_handler(struct input_dev *dev, struct input_handler *handler)
    {
23.
24.         const struct input_device_id *id;
25.         int error;
26.
27.         id = input_match_device(handler, dev);
28.         if (!id)
29.             return -ENODEV;
30.
31.         error = handler->connect(handler, dev, id);
32.         if (error && error != -ENODEV)
33.             pr_err("failed to attach handler %s to device %s, error: %d\n",
34.                 handler->name, kobject_name(&dev->dev.kobj), error);
35.
36.         return error;
37.     }</pre>
38. <pre></pre>
39. (1) 使用input_match_device()函数匹配, 采用dev匹配handler, handler被匹配。首先调用input_match_device()函数
40. <p></p>
41. <p align="left"></p>
42. <pre name="code" class="cpp"><pre name="code" class="cpp">static const struct input_device_id *input_match_device(struct input_handler *handler, struct input_dev *dev)
    {
43.
44.         const struct input_device_id *id;
45.         int i;
46.
47.         for (id = handler->id_table; id->flags || id->driver_info; id++) {
48.
49.             if (id->flags & INPUT_DEVICE_ID_MATCH_BUS)
50.                 if (id->bustype != dev->id.bustype)
51.                     continue;
52.
53.             if (id->flags & INPUT_DEVICE_ID_MATCH_VENDOR)
54.                 if (id->vendor != dev->id.vendor)
55.                     continue;
56.
57.             if (id->flags & INPUT_DEVICE_ID_MATCH_PRODUCT)
58.                 if (id->product != dev->id.product)
59.                     continue;
60.
61.             if (id->flags & INPUT_DEVICE_ID_MATCH_VERSION)
62.                 if (id->version != dev->id.version)
63.                     continue;
64.
65.             MATCH_BIT(evbit, EV_MAX);
66.             MATCH_BIT(keybit, KEY_MAX);
67.             MATCH_BIT(relbit, REL_MAX);
68.             MATCH_BIT(absbit, ABS_MAX);
69.             MATCH_BIT(mscbit, MSC_MAX);
70.             MATCH_BIT(ledbit, LED_MAX);
71.             MATCH_BIT(sndbit, SND_MAX);
72.             MATCH_BIT(ffbit, FF_MAX);
73.             MATCH_BIT(swbit, SW_MAX);
74.
75.             if (!handler->match || handler->match(handler, dev))
76.                 return id;
77.         }
78.
79.         return NULL;
80.     }</pre>
81. <p><br>
82. </p>
83. <pre></pre>
84. handler有某一个flag设置了, input设备对应的条件必须具备。使用MATCH_BIT宏进行匹配。<br>
85. <br>
86. (2) 匹配成功, 调用handler->connect()函数关联。触摸屏是事件设备, 所以将匹配一个input_handler evdev_handler, 对应的函数为evdev_connect, 代码存放在evdev.c中。
87. <p></p>
88. <p align="left"></p>
89. <pre name="code" class="cpp"><pre name="code" class="cpp">static int evdev_connect(struct input_handler *handler, struct input_dev *dev, const struct input_device_id *id)
    {
90.
91.         struct evdev *evdev;
92.         int minor;
93.         int error;
94.
95.         for (minor = 0; minor < EVDEV_MINORS; minor++)

```

```

96.         if (!evdev_table[minor])
97.             break;
98.
99.         if (minor == EVDEV_MINORS) {
100.             pr_err("no more free evdev devices\n");
101.             return -ENFILE;
102.         }
103.
104.         evdev = kzalloc(sizeof(struct evdev), GFP_KERNEL);
105.         if (!evdev)
106.             return -ENOMEM;
107.
108.         INIT_LIST_HEAD(&evdev->client_list);
109.         spin_lock_init(&evdev->client_lock);
110.         mutex_init(&evdev->mutex);
111.         init_waitqueue_head(&evdev->wait);
112.
113.         dev_set_name(&evdev->dev, "event%d", minor);
114.         evdev->exist = true;
115.         evdev->minor = minor;
116.
117.         evdev->handle.dev = input_get_device(dev);
118.         evdev->handle.name = dev_name(&evdev->dev);
119.         evdev->handle.handler = handler;
120.         evdev->handle.private = evdev;
121.
122.         evdev->dev.devt = MKDEV(INPUT_MAJOR, EVDEV_MINOR_BASE + minor);
123.         evdev->dev.class = &input_class;
124.         evdev->dev.parent = &dev->dev;
125.         evdev->dev.release = evdev_free;
126.         device_initialize(&evdev->dev);
127.
128.         error = input_register_handle(&evdev->handle);
129.         if (error)
130.             goto err_free_evdev;
131.
132.         error = evdev_install_chrdev(evdev);
133.         if (error)
134.             goto err_unregister_handle;
135.
136.         error = device_add(&evdev->dev);
137.         if (error)
138.             goto err_cleanup_evdev;
139.
140.         return 0;
141.
142. err_cleanup_evdev:
143.         evdev_cleanup(evdev);
144. err_unregister_handle:
145.         input_unregister_handle(&evdev->handle);
146. err_free_evdev:
147.         put_device(&evdev->dev);
148.         return error;
149. }</pre>
150. <pre></pre>
151. (1)定义一个比输入设备更为具体的设备——事件设备evdev，触摸屏接入到用户空间也就是事件设备。
152. <p></p>
153. <p align="left"></p>
154. <pre name="code" class="cpp"><pre name="code" class="cpp">struct evdev {
155.     int open;
156.     int minor;
157.     struct input_handle handle;
158.     wait_queue_head_t wait;
159.     struct evdev_client __rcu *grab;
160.     struct list_head client_list;
161.     spinlock_t client_lock; /* protects client_list */
162.     struct mutex mutex;
163.     struct device dev;
164.     bool exist;
165. };</pre>
166. <p><br>
167. </p>
168. <pre></pre>
169. <p></p>
170. <p align="left"> (2)初始化里面的数据。</p>
171. <p align="left"> (3)设置evdev里面的值。</p>
172. <p align="left"> (4)注册handle。</p>
173. <p align="left"> (5)因为evdev_handler可以匹配多个evdev，所以有一个数据是专门用来存储这些evdev的，
    所以用evdev_install_chrdev()函数把它放入数组。</p>
174. <pre name="code" class="cpp"><pre name="code" class="cpp">static int evdev_install_chrdev(s
    struct evdev *evdev)
175. {
176.     /*
177.      * No need to do any locking here as calls to connect and
178.      * disconnect are serialized by the input core
179.      */
180.     evdev_table[evdev->minor] = evdev;

```



```

181.         return 0;
182.     }</pre>
183. </pre></pre>
184.     (6)把设备加入内核。
185. <p align="left"> 3, 输入子系统设备驱动层: 是真正设备驱动层, 有具体的硬件接口, 数据结构为input_dev
186. </p>
187. <p align="left"></p>
188. <pre name="code" class="cpp"><pre name="code" class="cpp">struct input_dev {
189.     const char *name;
190.     const char *phys;
191.     const char *uniq;
192.     struct input_id id;
193.
194.     unsigned long propbit[BITS_TO_LONGS(INPUT_PROP_CNT)];
195.
196.     unsigned long evbit[BITS_TO_LONGS(EV_CNT)];
197.     unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];
198.     unsigned long relbit[BITS_TO_LONGS(REL_CNT)];
199.     unsigned long absbit[BITS_TO_LONGS(ABS_CNT)];
200.     unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)];
201.     unsigned long ledbit[BITS_TO_LONGS(LED_CNT)];
202.     unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];
203.     unsigned long ffbbit[BITS_TO_LONGS(FF_CNT)];
204.     unsigned long swbit[BITS_TO_LONGS(SW_CNT)];
205.
206.     unsigned int hint_events_per_packet;
207.
208.     unsigned int keycodemax;
209.     unsigned int keycodesize;
210.     void *keycode;
211.
212.     int (*setkeycode)(struct input_dev *dev, const struct input_keymap_entry *ke, unsigned
213.     int *old_keycode);
214.     int (*getkeycode)(struct input_dev *dev, struct input_keymap_entry *ke);
215.
216.     struct ff_device *ff;
217.
218.     unsigned int repeat_key;
219.     struct timer_list timer;
220.
221.     int rep[REP_CNT];
222.
223.     struct input_mt *mt;
224.
225.     struct input_absinfo *absinfo;
226.
227.     unsigned long key[BITS_TO_LONGS(KEY_CNT)];
228.     unsigned long led[BITS_TO_LONGS(LED_CNT)];
229.     unsigned long snd[BITS_TO_LONGS(SND_CNT)];
230.     unsigned long sw[BITS_TO_LONGS(SW_CNT)];
231.
232.     int (*open)(struct input_dev *dev);
233.     void (*close)(struct input_dev *dev);
234.     int (*flush)(struct input_dev *dev, struct file *file);
235.     int (*event)(struct input_dev *dev, unsigned int type, unsigned int code, int value);
236.
237.     struct input_handle __rcu *grab;
238.
239.     spinlock_t event_lock;
240.     struct mutex mutex;
241.
242.     unsigned int users;
243.     bool going_away;
244.
245.     struct device dev;
246.
247.     struct list_head h_list;
248.     struct list_head node;
249.
250.     unsigned int num_vals;
251.     unsigned int max_vals;
252.     struct input_value *vals;
253. };</pre>
254. </pre></pre>
255.     这里包含了所有可能输入设备的数据域, 我们驱动要定义的输入设备的参数就包含在这个数据结构中。<br>
256. <br>
257.     4, 把设备注册输入子系统里一般要经过以下步骤: <br>
258. <br>
259.     (1) 申请一个输入设备, 定义事件集合。关于输入事件的介绍可以看这里: Linux 内核点触摸接口协议<br>
260. <br>
261.     (2) 上报从设备中读到的数据。可使用函数
262. <p></p>
263. <p align="left"></p>
264. <pre name="code" class="cpp"><pre name="code" class="cpp">void input_report_key(struct input_dev *dev, unsigned int code, int value); // 上报按键事件

```

```

263. void input_report_rel(struct input_dev *dev, unsigned int code, int value); //上报相对坐
    标事件
264. void input_report_abs(struct input_dev *dev, unsigned int code, int value); //上报绝对坐
    标事件</pre>
265. <pre></pre>
266. 三、基于OMAP FT5x0x驱动源码分析<br>
267. <br>
268. 1、板级初始化代码<br>
    ...
    <br>
    板级初始化代码与具体的平台有关，负责相应平台的初始化工作，这里使用的pandaboard平台的代码，在目录a
    rch/arm/mach-omap2/board-omap4panda.c中。触摸屏设备当然需要在这里先声明并注册进系统。具体代码如下：
    <p></p>
    <p align="left"></p>
    <pre name="code" class="cpp"><pre name="code" class="cpp">static struct i2c_board_info __in
    itdata panda_i2c2_boardinfo[] = {
    {
    275.     I2C_BOARD_INFO("ft5x06_ts", 0x38),
    276.     .irq = OMAP_GPIO_IRQ(34),
    277. },
    278. };
    279. omap_register_i2c_bus(2, 100, panda_i2c2_boardinfo, ARRAY_SIZE(panda_i2c2_boardinfo));</pre>
    280. <pre></pre>
    281. <p></p>
    282. <p align="left"> 声明一个i2c_board_info结构的数组，这个数组包含pandaboard板上i2c接口的信息。先使
    用宏I2C_BOARD_INFO宏声明设备名为ft5x06_ts，地址为0x38。要说明的是这里的地址并不是指处理器中i2c寄存器
    的硬件地址，而是在系统加载后，位于/sys/bus/i2c/devices下的设备名。再申请这个i2c接口的中断口为GPIO34，
    这从板的硬件连接图可以看出。<br>
    283. <br>
    284. 然后把这个结构体注册进i2c总线，这里跟具体的平台有关，使用omap_register_i2c_bus函数注册。函数定义
    在arch/arm/plat-omap/i2c.c。尝试跟踪中断信息存储的位置，我们首先看omap_register_i2c_bus()</p>
    285. <p align="left"></p>
    286. <pre name="code" class="cpp"><pre name="code" class="cpp">int __init omap_register_i2c_bus(in
    t bus_id, u32 clkrate, struct i2c_board_info const *info, unsigned len)
    287. {
    288.     int err;
    289.
    290.     BUG_ON(bus_id < 1 || bus_id > omap_i2c_nr_ports());
    291.
    292.     if (info) {
    293.         err = i2c_register_board_info(bus_id, info, len);
    294.         if (err)
    295.             return err;
    296.     }
    297.
    298.     if (!i2c_pdata[bus_id - 1].clkrate)
    299.         i2c_pdata[bus_id - 1].clkrate = clkrate;
    300.
    301.     i2c_pdata[bus_id - 1].clkrate &= ~OMAP_I2C_CMDLINE_SETUP;
    302.
    303.     return omap_i2c_add_bus(bus_id);
    304. }</pre>
    305. <pre></pre>
    306. 信息info跳入i2c_register_board_info()函数
    307. <p></p>
    308. <p align="left"></p>
    309. <pre name="code" class="cpp">i2c_register_board_info(int busnum, struct i2c_board_info const
    *info, unsigned len)
    310. {
    311.     int status;
    312.
    313.     down_write(&__i2c_board_lock);
    314.
    315.     /* dynamic bus numbers will be assigned after the last static one */
    316.     if (busnum >= __i2c_first_dynamic_bus_num)
    317.         __i2c_first_dynamic_bus_num = busnum + 1;
    318.
    319.     for (status = 0; len; len--, info++) {
    320.         struct i2c_devinfo *devinfo;
    321.
    322.         devinfo = kzalloc(sizeof(*devinfo), GFP_KERNEL);
    323.         if (!devinfo) {
    324.             pr_debug("i2c-core: can't register boardinfo!\n");
    325.             status = -ENOMEM;
    326.             break;
    327.         }
    328.
    329.         devinfo->busnum = busnum;
    330.         devinfo->board_info = *info;
    331.         list_add_tail(&devinfo->list, & i2c_board_list);
    332.     }
    333.
    334.     up_write(&__i2c_board_lock);
    335.
    336.     return status;
    337. }</pre> 可以看到用链表存储这些设备信息，存储节点为i2c_devinfo结构
    338. <p></p>

```

```

339. <p align="left"></p>
340. <pre name="code" class="cpp"><pre name="code" class="cpp">struct i2c_devinfo {
341.     struct list_head    list;
342.     int                 busnum;
343.     struct i2c_board_info board_info;
344. };</pre>
345. <pre></pre>
346. 这个结构会在i2c_scan_static_board_info()函数被引用
347. <p></p>
348. <p align="left"></p>
349. <pre name="code" class="cpp"><pre name="code" class="cpp">static void i2c_scan_static_board
350. _info(struct i2c_adapter *adapter)
351. {
352.     struct i2c_devinfo    *devinfo;
353.
354.     down_read(&__i2c_board_lock);
355.     list_for_each_entry(devinfo, &__i2c_board_list, list) {
356.         if (devinfo->busnum == adapter->nr
357.             && !i2c_new_device(adapter,
358.                                 &devinfo->board_info))
359.             dev_err(&adapter->dev,
360.                     "Can't create device at 0x%02x\n",
361.                     devinfo->board_info.addr);
362.     }
363.     up_read(&__i2c_board_lock);
364. }</pre>
365. <pre></pre>
366. 这个函数遍历设备信息链表，找出与adapter匹配的设备，并使用i2c_new_device()添加进去。这个函数在注册
367. adapter时被i2c_register_adapter()调用。
368. <p></p>
369. <p align="left"></p>
370. <pre name="code" class="cpp"><pre name="code" class="cpp">struct i2c_client *i2c_new_device(
371. struct i2c_adapter *adap, struct i2c_board_info const *info)
372. {
373.     struct i2c_client    *client;
374.     int                 status;
375.
376.     client = kzalloc(sizeof *client, GFP_KERNEL);
377.     if (!client)
378.         return NULL;
379.
380.     client->adapter = adap;
381.
382.     client->dev.platform_data = info->platform_data;
383.
384.     if (info->archdata)
385.         client->dev.archdata = *info->archdata;
386.
387.     client->flags = info->flags;
388.     client->addr = info->addr;
389.     client->irq = info->irq;
390.
391.     strcpy(client->name, info->type, sizeof(client->name));
392.
393.     /* Check for address validity */
394.     status = i2c_check_client_addr_validity(client);
395.     if (status) {
396.         dev_err(&adap->dev, "Invalid %d-bit I2C address 0x%02hx\n",
397.                 client->flags & I2C_CLIENT_TEN ? 10 : 7, client->addr);
398.         goto out_err_silent;
399.     }
400.
401.     /* Check for address business */
402.     status = i2c_check_addr_busy(adap, client->addr);
403.     if (status)
404.         goto out_err;
405.
406.     client->dev.parent = &client->adapter->dev;
407.     client->dev.bus = &i2c_bus_type;
408.     client->dev.type = &i2c_client_type;
409.     client->dev.of_node = info->of_node;
410.
411.     /* For 10-bit clients, add an arbitrary offset to avoid collisions */
412.     dev_set_name(&client->dev, "%d-%04x", i2c_adapter_id(adap),
413.                 client->addr | ((client->flags & I2C_CLIENT_TEN
414.                                     ? 0xa000 : 0));
415.     status = device_register(&client->dev);
416.     if (status)
417.         goto out_err;
418.
419.     dev_dbg(&adap->dev, "client [%s] registered with bus id %s\n",
420.             client->name, dev_name(&client->dev));
421.
422.     return client;
423.
424. out_err:
425.     dev_err(&adap->dev, "Failed to register i2c client %s at 0x%02x "

```

创建client

```

423.         "(%d)\n", client->name, client->addr, status);
424. out_err_silent:
425.     kfree(client);
426.     return NULL;
427. }</pre>
428. <pre></pre>
429. 这里创建了i2c_client对象, 并把devinfo内的信息存储在它里面, 下面设备驱动调用的probe函数传进的i2c_client就是这里的client。<br>
...
433. 设备与驱动绑定的代码位于目录drivers/input/touchscreen/ft5x0x.c中, 这个文件包含了i2c设备的具体代码, 以模块的形式编译进内核。
434.
435.
436.
437. {
438.     { FT5X0X_NAME, 0 }, { }
439. };
440. MODULE_DEVICE_TABLE(i2c, ft5x0x_ts_id);
441.
442. static struct i2c_driver ft5x0x_ts_driver =
443. {
444.     .probe      = ft5x0x_ts_probe,
445.     .remove     = __devexit_p(ft5x0x_ts_remove),
446.     .id_table   = ft5x0x_ts_id,
447.     .driver     =
448.     {
449.         .name    = FT5X0X_NAME,
450.         .owner   = THIS_MODULE,
451.     },
452. };
453.
454. static int __init ft5x0x_ts_init(void)
455. {
456.     return i2c_add_driver(&ft5x0x_ts_driver);
457. }
458.
459. static void __exit ft5x0x_ts_exit(void)
460. {
461.     i2c_del_driver(&ft5x0x_ts_driver);
462. }
463.
464. module_init(ft5x0x_ts_init);
465. module_exit(ft5x0x_ts_exit);
466.
467. MODULE_AUTHOR("<wenfs@Focaltech-systems.com>");
468. MODULE_DESCRIPTION("FocalTech ft5x0x TouchScreen driver");
469. MODULE_LICENSE("GPL");</pre>
470. <pre></pre>
471. <p></p>
472. <p align="left"> 首先建立设备与驱动的映射表ft5x0x_ts_id, 通过宏MODULE_DEVICE_TABLE关联起来, MODULE_DEVICE_TABLE第一个参数表示这个设备id是i2c总线上了, 第二个参数是指设备, 当系统找到这个设备时, 就会通过FT5X0X_NAME把设备与这个模块关联起来。<br>
473. 这是个i2c驱动模块, 当然需要定义一个i2c_driver结构体来标识这个驱动, 并说明实现的驱动函数有probe()和remove()函数。<br>
474. <br>
475. 之后就是添加模块初始化和退出函数, 分别是在模块初始化时调用i2c_add_driver()宏加入i2c驱动, i2c_add_driver()实际上调用到的是 i2c_register_driver()函数。</p>
476. <p align="left"></p>
477. <pre name="code" class="cpp"><pre name="code" class="cpp">int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
478. {
479.     int res;
480.
481.     /* Can't register until after driver model init */
482.     if (unlikely(WARN_ON(!i2c_bus_type.p)))
483.         return -EAGAIN;
484.
485.     /* add the driver to the list of i2c drivers in the driver core */
486.     driver->driver.owner = owner;
487.     driver->driver.bus = &i2c_bus_type;
488.
489.     /* When registration returns, the driver core
490.      * will have called probe() for all matching-but-unbound devices.
491.      */
492.     res = driver_register(&driver->driver);
493.     if (res)
494.         return res;
495.
496.     /* Drivers should switch to dev_pm_ops instead. */
497.     if (driver->suspend)
498.         pr_warn("i2c-core: driver [%s] using legacy suspend method\n",
499.             driver->driver.name);
500.     if (driver->resume)
501.         pr_warn("i2c-core: driver [%s] using legacy resume method\n",

```

```

502.         driver->driver.name);
503.
504.         pr_debug("i2c-core: driver [%s] registered\n", driver->driver.name);
505.
506.         INIT_LIST_HEAD(&driver->clients);
507.         /* Walk the adapters that are already present */
508.         i2c_for_each_dev(driver, __process_new_driver);
509.
510.         return 0;
511.     }
512. }
513.
514. <p align="left">    在模块退出时调用i2c_del_driver删除i2c驱动。</p>
515. <p align="left">    3、设备驱动程序实现<br>
516.    从上面i2c_driver结构可以看出，ft5x0x实现了驱动的两个接口函数，分别为probe()和remove()函数。<br>
517.    >
518.    probe()具体实现函数为ft5x0x_ts_probe(),这个函数在内核初始化时，如果设备和驱动匹配，将调用，实现i
519.    2c设备的初始化。具体实现：</p>
520. <p align="left"></p>
521. <pre name="code" class="cpp"><pre name="code" class="cpp">static int ft5x0x_ts_probe(struct
522.     i2c_client *client, const struct i2c_device_id *id)
523. {
524.     struct ft5x0x_ts_data *ft5x0x_ts;
525.     struct input_dev *input_dev;
526.     int err = 0;
527.     int rev_id = 0;
528.
529.     if (!i2c_check_functionality(client->adapter, I2C_FUNC_I2C)) {
530.         err = -ENODEV;
531.         goto err_out;
532.     }
533.
534.     ft5x0x_ts = kzalloc(sizeof(*ft5x0x_ts), GFP_KERNEL);
535.     if (!ft5x0x_ts) {
536.         err = -ENOMEM;
537.         goto err_free_mem;
538.     }
539.
540.     this_client = client;
541.     i2c_set_clientdata(client, ft5x0x_ts);
542.
543.     rev_id = i2c_smbus_read_byte_data(client, FT5X0X_REG_FT5201ID);
544.     if (rev_id != FT5X06_ID)
545.     {
546.         err = -ENODEV;
547.         dev_err(&client->dev, "failed to probe FT5X0X touchscreen device\n");
548.         goto err_free_mem;
549.     }
550.
551.     INIT_WORK(&ft5x0x_ts->pen_event_work, ft5x0x_ts_pen_irq_work);
552.     ft5x0x_ts->ts_workqueue = create_singlethread_workqueue(dev_name(&client->dev));
553.     if (!ft5x0x_ts->ts_workqueue)
554.     {
555.         err = -ESRCH;
556.         goto err_free_thread;
557.     }
558.
559.     err = request_irq(client->irq, ft5x0x_ts_interrupt, IRQF_DISABLED | IRQF_TRIGGER_RISING,
560. "ft5x0x_ts", ft5x0x_ts);
561.     if (err < 0)
562.     {
563.         dev_err(&client->dev, "request irq failed\n");
564.         goto err_free_irq;
565.     }
566.
567.     input_dev = input_allocate_device();
568.     if (!input_dev)
569.     {
570.         err = -ENOMEM;
571.         dev_err(&client->dev, "failed to allocate input device\n");
572.         goto err_free_input;
573.     }
574.
575.     ft5x0x_ts->input_dev = input_dev;
576.
577. #ifdef CONFIG_FT5X0X_MULTITOUCH
578.     set_bit(ABS_MT_TOUCH_MAJOR, input_dev->absbit);
579.     set_bit(ABS_MT_POSITION_X, input_dev->absbit);
580.     set_bit(ABS_MT_POSITION_Y, input_dev->absbit);
581.     set_bit(ABS_MT_WIDTH_MAJOR, input_dev->absbit);
582.     set_bit(ABS_PRESSURE, input_dev->absbit);
583.
584.     input_set_abs_params(input_dev,
585.         ABS_MT_POSITION_X, 0, SCREEN_MAX_X, 0, 0);
586.     input_set_abs_params(input_dev,
587.         ABS_MT_POSITION_Y, 0, SCREEN_MAX_Y, 0, 0);
588.     input_set_abs_params(input_dev,

```

```

585.         ABS_MT_TOUCH_MAJOR, 0, PRESS_MAX, 0, 0);
586.     input_set_abs_params(input_dev,
587.         ABS_MT_WIDTH_MAJOR, 0, 200, 0, 0);
588. #else
589.     set_bit(ABS_X, input_dev->absbit);
590.     set_bit(ABS_Y, input_dev->absbit);
591.     set_bit(ABS_PRESSURE, input_dev->absbit);
592.     set_bit(BTN_TOUCH, input_dev->keybit);
593.     ...
594.     input_set_abs_params(input_dev, ABS_X, 0, SCREEN_MAX_X, 0, 0);
595.     input_set_abs_params(input_dev, ABS_Y, 0, SCREEN_MAX_Y, 0, 0);
596.     input_set_abs_params(input_dev, ABS_PRESSURE, 0, PRESS_MAX, 0, 0);
597. #endif
598.
599.     set_bit(EV_ABS, input_dev->evbit);
600.     set_bit(EV_KEY, input_dev->evbit);
601.
602.     input_dev->name = FT5X0X_NAME;
603.     err = input_register_device(input_dev);
604.     if (err)
605.     {
606.         dev_err(&client->dev, "failed to register input device: %s\n",
607.             dev_name(&client->dev));
608.         goto err_free_input;
609.     }
610.
611. #ifdef CONFIG_HAS_EARLYSUSPEND
612.     ft5x0x_ts->early_suspend.level = EARLY_SUSPEND_LEVEL_BLANK_SCREEN + 1;
613.     ft5x0x_ts->early_suspend.suspend = ft5x0x_ts_suspend;
614.     ft5x0x_ts->early_suspend.resume = ft5x0x_ts_resume;
615.     register_early_suspend(&ft5x0x_ts->early_suspend);
616. #endif
617.
618.     return 0;
619.
620. err_free_input:
621.     input_free_device(input_dev);
622. err_free_irq:
623.     free_irq(client->irq, ft5x0x_ts);
624. err_free_thread:
625.     cancel_work_sync(&ft5x0x_ts->pen_event_work);
626.     destroy_workqueue(ft5x0x_ts->ts_workqueue);
627.     i2c_set_clientdata(client, NULL);
628. err_free_mem:
629.     kfree(ft5x0x_ts);
630. err_out:
631.     return err;
632. }</pre>
<pre></pre>
633. <pre></pre>
634.     这个函数代码比较长，但是可以分部分去看，每一部分对应不同的初始化工作。大概可以分为这几个部分：<br>
635. <br>
636.     (1)检测适配器是否支持I2C_FUNC_I2C的通信方式：i2c_check_functionality(client->adapter, I2C_FUNC_I2C)<br>
637. <br>
638.     (2)申请内存存储驱动的数据：ft5x0x_ts = kzalloc(sizeof(*ft5x0x_ts), GFP_KERNEL); 并把驱动数据赋值给系统传过来的i2c_client数据：this_client = client; i2c_set_clientdata(client, ft5x0x_ts);<br>
639. <br>
640.     (3)验证将要通信的设备ID：rev_id = i2c_smbus_read_byte_data(client, FT5X0X_REG_FT5201ID);<br>
641. <br>
642.     (4)创建触摸事件的工作队列并初始化工作队列的处理函数：INIT_WORK(&ft5x0x_ts->pen_event_work, ft5x0x_ts_pen_irq_work); ft5x0x_ts->ts_workqueue = create_singlethread_workqueue(dev_name(&client->dev));<br>
643. <br>
644.     (5)申请系统中断并声明中断处理函数：request_irq(client->irq, ft5x0x_ts_interrupt, IRQF_DISABLED | IRQF_TRIGGER_RISING, "ft5x0x_ts", ft5x0x_ts);<br>
645. <br>
646.     (6)分配一个输入设备实例，初始化数据并注册进系统：input_dev = input_allocate_device(); input_register_device(input_dev);<br>
647. <br>
648.     (7)如果定义了earlysuspend，声明其处理函数。earlysuspend用于对触摸屏类设备的电源管理，降低功耗。<br>
649. <br>
650.     (8)最后对执行过程中可能出现的错误进行处理。<br>
651. <br>
652.     当有触摸动作时，触摸屏将会执行(5)中声明的中断处理函数ft5x0x_ts_interrupt，具体实现：<br>
653. <pre name="code" class="cpp"><pre name="code" class="cpp">static irqreturn_t ft5x0x_ts_interrupt(int irq, void *dev_id)
654. {
655.     struct ft5x0x_ts_data *ft5x0x_ts = dev_id;
656.
657.     if (!work_pending(&ft5x0x_ts->pen_event_work))
658.         queue_work(ft5x0x_ts->ts_workqueue, &ft5x0x_ts->pen_event_work);
659.
660.     return IRQ_HANDLED;
661. }</pre>

```

```

662. <pre></pre>
663. 可以看到，中断处理就是在判断工作队列在有没有被挂起的情况下在触摸时间加入到工作队列中去，等待工作队列
    处理函数ft5x0x_ts_pen_irq_work()的实现。ft5x0x_ts_pen_irq_work()的实现: <br>
664. <pre name="code" class="cpp"><pre name="code" class="cpp">static void ft5x0x_ts_pen_irq_wor
    k(struct work_struct *work)
665. {
666.     int ret = -1;
667.     ...
    ret = ft5x0x_read_data();
    if (ret == 0)
        ft5x0x_report_value();
671. }</pre>
    <pre></pre>
    概括来说这里只做了两件事：从设备中读取数据；把数据上报到输入子系统中。<br>
    (1)从设备中读取数据由函数ft5x0x_read_data()实现<br>
    <pre name="code" class="cpp">static int ft5x0x_read_data(void)
677. {
678.     struct ft5x0x_ts_data *data = i2c_get_clientdata(this_client);
679.     struct ts_event *event = &data->event;
680.     u8 buf[32] = {0};
681.     int ret = -1;
682.     int status = 0;
683.
684. #ifdef CONFIG_FT5X0X_MULTITOUCH
685.     ret = ft5x0x_i2c_rxdata(buf, 31);
686. #else
687.     ret = ft5x0x_i2c_rxdata(buf, 7);
688. #endif
689.     if (ret < 0)
690.     {
691.         printk("%s read_data i2c_rxdata failed: %d\n", __func__, ret);
692.         return ret;
693.     }
694.
695.     memset(event, 0, sizeof(struct ts_event));
696.     event->touch_point = buf[2] & 0x07;
697.
698.     if (event->touch_point == 0)
699.     {
700.         ft5x0x_ts_inactivate();
701.         return 1;
702.     }
703.
704. #ifdef CONFIG_FT5X0X_MULTITOUCH
705.     switch (event->touch_point)
706.     {
707.         case 5:
708.             event->x5 = (s16)(buf[0x1b] & 0x0F)<<8 | (s16)buf[0x1c];
709.             event->y5 = (s16)(buf[0x1d] & 0x0F)<<8 | (s16)buf[0x1e];
710.             status = (s16)((buf[0x1b] & 0xc0) >> 6);
711.             event->touch_ID5=(s16)(buf[0x1D] & 0xF0)>>4;
712.             if (status == 1) ft5x0x_ts_release();
713.
714.         case 4:
715.             event->x4 = (s16)(buf[0x15] & 0x0F)<<8 | (s16)buf[0x16];
716.             event->y4 = (s16)(buf[0x17] & 0x0F)<<8 | (s16)buf[0x18];
717.             status = (s16)((buf[0x15] & 0xc0) >> 6);
718.             event->touch_ID4=(s16)(buf[0x17] & 0xF0)>>4;
719.             if (status == 1) ft5x0x_ts_release();
720.
721.         case 3:
722.             event->x3 = (s16)(buf[0x0f] & 0x0F)<<8 | (s16)buf[0x10];
723.             event->y3 = (s16)(buf[0x11] & 0x0F)<<8 | (s16)buf[0x12];
724.             status = (s16)((buf[0x0f] & 0xc0) >> 6);
725.             event->touch_ID3=(s16)(buf[0x11] & 0xF0)>>4;
726.             if (status == 1) ft5x0x_ts_release();
727.
728.         case 2:
729.             event->x2 = (s16)(buf[9] & 0x0F)<<8 | (s16)buf[10];
730.             event->y2 = (s16)(buf[11] & 0x0F)<<8 | (s16)buf[12];
731.             status = (s16)((buf[0x9] & 0xc0) >> 6);
732.             event->touch_ID2=(s16)(buf[0x0b] & 0xF0)>>4;
733.             if (status == 1) ft5x0x_ts_release();
734.
735.         case 1:
736.             event->x1 = (s16)(buf[3] & 0x0F)<<8 | (s16)buf[4];
737.             event->y1 = (s16)(buf[5] & 0x0F)<<8 | (s16)buf[6];
738.             status = (s16)((buf[0x3] & 0xc0) >> 6);
739.             event->touch_ID1=(s16)(buf[0x05] & 0xF0)>>4;
740.             if (status == 1) ft5x0x_ts_release();
741.             break;
742.
743.         default:
744.             return -1;
745.     }
746. #else

```

```

747.     if (event->touch_point == 1)
748.     {
749.         event->x1 = (s16)(buf[3] & 0x0F)<<8 | (s16)buf[4];
750.         event->y1 = (s16)(buf[5] & 0x0F)<<8 | (s16)buf[6];
751.     }
752. #endif
753.     event->pressure = 200;
754.
755.     dev_dbg(&this_client->dev, "%s: 1:%d %d 2:%d %d \n", __func__,
756.         event->x1, event->y1, event->x2, event->y2);
757.
758.     return 0;
759. }</pre>
760. <p></p>
761. <p align="left"></p>
762. <pre name="code" class="cpp"><pre name="code" class="cpp">static int ft5x0x_i2c_rxdata(char
763. *rxdata, int length)
764. {
765.     int ret;
766.
767.     struct i2c_msg msgs[] =
768.     {
769.         {
770.             .addr = this_client->addr,
771.             .flags = 0,
772.             .len = 1,
773.             .buf = rxdata,
774.         },
775.         {
776.             .addr = this_client->addr,
777.             .flags = I2C_M_RD,
778.             .len = length,
779.             .buf = rxdata,
780.         },
781.     };
782.
783.     ret = i2c_transfer(this_client->adapter, msgs, 2);
784.     if (ret < 0)
785.         pr_err("msg %s i2c read error: %d\n", __func__, ret);
786.
787.     return ret;
788. }</pre>
789. <pre></pre>
790. 这个函数里面会构建一个i2c_msg结构去调用i2c_transfer()函数读取数据，在前面已经说过，i2c_trans
791. fer()函数最终调用的就是i2c_algorithm中master_xfer()函数进行实际的数据传输，这个函数在adapter驱动中实
792. 现。<br>
793. 对读取到的数据处理涉及到IC里面的数据格式，查找芯片的数据手册可以看到寄存器的数据分步。
794. <p></p>
795. <p align="left"><br>
797. </p>
798. <p align="left">ft5x0x寄存器映射表<br>
799. <br>
800. 这里为缩小篇幅，只截取一部分寄存器，其余寄存器格式可参照既给出的寄存器信息。结合图片可以看出，代
801. 码先根据是否多点触摸，从寄存器读取数据长度，若是只需要单点触摸就读取7个字节信息，若是多点触摸，就读取31
802. 个字节的信息，从代码可以看出最多支持五点触摸。读出信息的第三个字节的低4位为触摸点数，所以event->touch
803. point = buf[2] & 0x07;判断触摸点数之后便分别计算触摸点的坐标，这里以第一个点为例，其余点可以依此类推。
804. 第一个点的坐标信息包含在地址03h~06h中，坐标信息由12bit组成，分布在两个字节中，另外还包含flag标志信
805. 息，用于表示触摸点的状态，还包含触摸点的ID识别。<br>
806. <br>
807. (2)把数据上报到输入子系统中由函数ft5x0x_report_value()实现</p>
808. <p align="left"></p>
809. <pre name="code" class="cpp"><pre name="code" class="cpp">static void ft5x0x_report_value(v
810. oid)
811. {
812.     struct ft5x0x_ts_data *data = i2c_get_clientdata(this_client);
813.     struct ts_event *event = &data->event;
814.
815. #ifdef CONFIG_FT5X0X_MULTITOUCH
816.     switch(event->touch_point)
817.     {
818.     case 5:
819.         input_report_abs(data->input_dev, ABS_MT_TRACKING_ID, event->touch_ID5);
820.         input_report_abs(data->input_dev, ABS_MT_TOUCH_MAJOR, event->pressure);
821.         input_report_abs(data->input_dev, ABS_MT_POSITION_X, SCREEN_MAX_X - event->x5);
822.         input_report_abs(data->input_dev, ABS_MT_POSITION_Y, event->y5);
823.         input_report_abs(data->input_dev, ABS_MT_WIDTH_MAJOR, 1);
824.         input_mt_sync(data->input_dev);
825.     case 4:
826.         input_report_abs(data->input_dev, ABS_MT_TRACKING_ID, event->touch_ID4);
827.         input_report_abs(data->input_dev, ABS_MT_TOUCH_MAJOR, event->pressure);
828.         input_report_abs(data->input_dev, ABS_MT_POSITION_X, SCREEN_MAX_X - event->x4);
829.         input_report_abs(data->input_dev, ABS_MT_POSITION_Y, event->y4);
830.         input_report_abs(data->input_dev, ABS_MT_WIDTH_MAJOR, 1);
831.         input_mt_sync(data->input_dev);
832.     case 3:
833.         input_report_abs(data->input_dev, ABS_MT_TRACKING_ID, event->touch_ID3);

```



```
824.         input_report_abs(data->input_dev, ABS_MT_TOUCH_MAJOR, event->pressure);
825.         input_report_abs(data->input_dev, ABS_MT_POSITION_X, SCREEN_MAX_X - event->x3);
826.         input_report_abs(data->input_dev, ABS_MT_POSITION_Y, event->y3);
827.         input_report_abs(data->input_dev, ABS_MT_WIDTH_MAJOR, 1);
828.         input_mt_sync(data->input_dev);
829.     case 2:
830.         input_report_abs(data->input_dev, ABS_MT_TRACKING_ID, event->touch_ID2);
831.         input_report_abs(data->input_dev, ABS_MT_TOUCH_MAJOR, event->pressure);
832.         input_report_abs(data->input_dev, ABS_MT_POSITION_X, SCREEN_MAX_X - event->x2);
833.         input_report_abs(data->input_dev, ABS_MT_POSITION_Y, event->y2);
834.         input_report_abs(data->input_dev, ABS_MT_WIDTH_MAJOR, 1);
835.         input_mt_sync(data->input_dev);
836.     case 1:
837.         input_report_abs(data->input_dev, ABS_MT_TRACKING_ID, event->touch_ID1);
838.         input_report_abs(data->input_dev, ABS_MT_TOUCH_MAJOR, event->pressure);
839.         input_report_abs(data->input_dev, ABS_MT_POSITION_X, SCREEN_MAX_X - event->x1);
840.         input_report_abs(data->input_dev, ABS_MT_POSITION_Y, event->y1);
841.         input_report_abs(data->input_dev, ABS_MT_WIDTH_MAJOR, 1);
842.         input_mt_sync(data->input_dev);
843.     default:
844.         break;
845. }
846. #else /* CONFIG_FT5X0X_MULTITOUCH*/
847.     if (event->touch_point == 1)
848.     {
849.         input_report_abs(data->input_dev, ABS_X, SCREEN_MAX_X - event->x1);
850.         input_report_abs(data->input_dev, ABS_Y, event->y1);
851.         input_report_abs(data->input_dev, ABS_PRESSURE, event->pressure);
852.     }
853.     input_report_key(data->input_dev, BTN_TOUCH, 1);
854. #endif /* CONFIG_FT5X0X_MULTITOUCH*/
855.     input_sync(data->input_dev);
856.
857.     dev_dbg(&this_client->dev, "%s: 1:%d 2:%d 3:%d\n", __func__,
858.         event->x1, event->y1, event->x2, event->y2);
859. }</pre>
860. <pre></pre>
861. 因为触摸屏使用的是绝对坐标系，上报数据使用input_report_abs()函数，参数要注意选择合适的事件集。上报
玩数据要记得同步，使用input_mt_sync()表示单个手指信息结束，使用input_sync()表示整个触摸动作的结束。<b
r>
862. <br>
863.     remove()函数的具体实现是ft5x0x_ts_remove(),做与probe()相反的工作，释放内存。
864. <p></p>
865. <p align="left"></p>
866. <pre name="code" class="cpp"><pre name="code" class="cpp">static int __devexit ft5x0x_ts_remo
ove(struct i2c_client *client)
867. {
868.     struct ft5x0x_ts_data *ft5x0x_ts = i2c_get_clientdata(client);
869.
870.     unregister_early_suspend(&ft5x0x_ts->early_suspend);
871.     free_irq(client->irq, ft5x0x_ts);
872.     input_unregister_device(ft5x0x_ts->input_dev);
873.     kfree(ft5x0x_ts);
874.     cancel_work_sync(&ft5x0x_ts->pen_event_work);
875.     destroy_workqueue(ft5x0x_ts->ts_workqueue);
876.     i2c_set_clientdata(client, NULL);
877.
878.     return 0;
879. }</pre>
880. <pre></pre>
881. 至此，我们从i2c接口及输入子系统的接口清楚了整个触摸屏驱动的结构，并详细分析了触摸屏驱动中的具体实
现。
```



严禁讨论涉及中国之军/政相关话题，违者会被禁言、封号！

nanopi2-触摸屏-I2C Note



u010623392 2016年09月25日 12:37 857

1、I2C驱动 drivers/i2c/busses/i2c-nxp.c 2、触摸屏驱动 drivers/input/touchscreen/ft5x0x_ts.c it7260_mts.c ...

linux I2C驱动分析整理



lushengchu2003 2013年11月27日 10:41 2482

一直以为对kernel I2C很熟悉了，可是最近用到时候，发现对其中一些函数理解还不透彻，加上以前分析的也没有做下笔记，现在重新整理一份，供以后参考。 平台是allwinner A10 linux3...

佩服啊！PHP这么牛逼的原因是？

为什么说2018年你必须学php?这份学习指南给你..



Linux下触摸屏驱动程序分析



fengel_cs 2016年01月22日 16:31 789

本文主要分析Linux3.5--Exynos4412平台，分析触摸屏驱动核心内容。Linux下触摸屏驱动(以ft5x06_ts为例)需要了解如下知识：1. I2C协议 2. Exynos4412处理器...



linux 触摸屏驱动分析



wujianguizhen 2013年12月04日 10:46 1426

mini2440驱动分析系列之-----Mini2440触摸屏程序分析 By JeefJiang July, 8th, 2009 ...

【Linux内核驱动】编写I2C外设驱动读取触摸屏固件版本

编写I2C外设驱动步骤 注册I2C设备，一般在板级文件中，定义i2c_board_info 注册I2C驱动：i2c_register_driver，i2c_del_driver 利用i2c_client...



wr132 2017年11月18日 17:04 131

I2C接口的OLED在树莓派3上的应用（已过时）

==本文驱动已过时，最新资讯请参见 <http://blog.csdn.net/ki1381/article/details/79291138>参考了<http://shumeipai.nxez.com/2...>



ki1381 2018年02月08日 11:35 5685

I2C接口的OLED在树莓派3上的应用（luma.oled驱动）

2016年9月写过一篇blog记录了如何将12864的OLED小显示屏应用到树莓派上。时至今日，当初给出的链接虽然还在，但已经有了 翻天覆地的更新，因此有必要重新学习下新的方法。参考资料：<http://...>



ki1381 2018年02月08日 17:04 533

全网络对Linux input子系统最清晰、详尽的分析

本文应是全网对linux input子系统分析最有系统逻辑性和最清晰的分析文章了，主要结构input-core, input-handler和input-device三者的关系以及应用open和rea...



yueqian_scut 2015年08月27日 14:27 2563

Linux I2C工具查看配置I2C设备



zjy900507 2017年11月28日 15:11 454

1.安装 I2C驱动载入和速率修改请查看博文【树莓派学习笔记——I2C设备载入和速率设置】。2.I2C总线扫描通过i2cdetect -l指令可以查看树莓派上的I...

mini2440 平台上挂载I2C接口触摸屏的驱动开发过程

本篇记录在友善之臂 mini2440 平台上挂载I2C接口触摸屏的驱动开发过程。内核版本linux-2.6.32.2, 平台是ARM9 S3C2440+I2C接口的触摸屏 如上篇Li...



yhf19881015 2012年11月15日 23:16 1800