

全网络对Linux input子系统最清晰、详尽的分析

原创2015年08月27日 14:27:40

1

标签：linux input / 输入子系统 / input_handler / input_device / input核心层

2596

Linux input分析之二：解构input_handler、input_core、input_device

输入输出是用户和产品交互的手段，因此输入驱动开发在Linux驱动开发中很常见。同时，input子系统的分层架构思想在Linux驱动设计中极具代表性和先进性，因此对Linux input子系统进行深入分析很有意义。

本文继续在《Linux input子系统分析之一：软件分层》的基础上继续深入研究Linux输入子系统的分层架构思想以及其实现。软件分层探讨的是输入消息从底层硬件到内核、应用层的消息传递和使用过程，而本文则是专注剖析Linux内核驱动层对输入设备的抽象分层管理和实现。

一、input子系统知识点回顾

详细请看《Linux input子系统分析之一：软件分层》一文。输入子系统对linux的输入设备驱动进行了高度抽象，最终分成了三层，包括input核心层、input事件处理层和input设备驱动层。input核心层（input-core）对input设备（input-device）和input事件处理（input-handler）进行管理并进行消息转发。如下图：

设备驱动层

核心层

事件层

用户空间

触摸屏输入设备

鼠标输入设备

键盘输入设备

Input Core

event Handler

TS Handler

Mouse Handler

Keyboard Handler

设备节点访问

所有的输入设备的主设备号都是13，input-core通过次设备号来将输入设备进行分类，如0-31是游戏杆，32-63是鼠标（对应Mouse Handler）、64-95是事件设备（如触摸屏，对应Event Handler）。

二、input核心层的任务

核心层input-core完成的工作包括：

1）直接跟字符设备驱动框架交互，字符设备驱动框架根据主设备号来进行管理，而input-core则是依赖于次设备号来进行分类管理。Input子系统的所有输入设备的主设备号都是13，其对应input-core定义的struct file_operations input_fops。驱动架构层通过主设备号13获取到input_fops，之后的处理便交给input_fops进行。

2）提供接口供事件处理层（input-handler）和输入设备（input-device）注册，并为输入设备找到匹配的事件处理者。

3）将input-device产生的消息（如触屏坐标和压力值）转发给input-handler，或者将input-handler的消息传递给input-device（如鼠标的闪光灯命令）。

三、input子系统初始化

1. input-core初始化

--driver/input/input.c

在设备模型/sys/class目录注册设备类，在/proc/bus/input目录产生设备信息，向字符设备驱动框架注册input子系统的接口操作集合（主设备号13和input_fops）。

吴跃前

博客专家

关注

原创134

粉丝959

喜欢224

等级：博客6

访问量：51

积分：6898

排名：414

单身公寓

广告

他的最新文章

RTX-TINY AND C51工具链相关

基于C语言的状态机框架和实现

tensorflow 安装

微信跑步机

混合编程接口规范

文章分类

嵌入式开发、软件架构设计

LINUX内核、驱动

微信硬件开发和物联网

集成电路设计

交叉工具链

机器学习

展开

博主专栏

SoC嵌入式软件架

27843

10篇

微信硬件平台解决

286928

45篇

```
static const struct file_operations input_fops = {
    .owner = THIS_MODULE,
    .open = input_open_file,
};

static int __init input_init(void)
{
    int err;
    input_init_abs_bypass();
    err = class_register(&input_class);
    err = input_proc_init();
    err = register_chrdev(13, "input", &input_fops);
}
```

2.input-handler初始化

以支持触摸屏TS的event-handler为例说明。

--driver/input/evdev.c

```
static const struct input_device_id evdev_ids[] = {
    { .driver_info = 1 }, /* Matches all devices */
    {}, /* Terminating zero entry */
};

static struct input_handler evdev_handler = {
    .event = evdev_event,
    .connect = evdev_connect,
    .disconnect = evdev_disconnect,
    .fops = &evdev_fops,
    .minor = EVDEV_MINOR_BASE,
    .name = "evdev",
    .id_table = evdev_ids,
};

static int __init evdev_init(void)
{
    return input_register_handler(&evdev_handler);
}
```

继续展开input_register_handler接口：

--driver/input/input.c

```
static struct input_handler *input_table[8];
static LIST_HEAD(input_handler_list);

int input_register_handler(struct input_handler *handler)
{
    struct input_dev *dev;
    if (handler->fops->minor < EVDEV_MINOR_BASE || handler->minor > EVDEV_MINOR_MAX)
        return -EINVAL;
    input_table[handler->minor - EVDEV_MINOR_BASE] = handler;
    list_add_tail(&handler->node, &input_handler_list);
    list_for_each_entry(dev, &input_dev_list, node)
        input_attach_handler(dev, handler);
}
```

3.input-device初始化

以触摸屏TSC2007为例，该触摸屏是I2C总线接口访问。

--driver/input/touchscreen/tsc2007.c



嵌入式Linux内核驱动分析
14 篇 46188

文章存档

2018年3月
2018年1月
2017年12月
2017年10月
2017年9月
2017年8月

展开

他的热门文章

从零开始搭建微信硬件开发环境：
——1小时掌握微信硬件开发流程
24964

全球最低功耗蓝牙单芯片DA145
体系 - 层次架构和BLE消息事件
13642

蓝牙DA14580开发：固件格式、
和烧写
13399

揭开智能配置上网（微信AirKiss
面纱
11624

如何提高蓝牙BLE的传输速率和
11581

基于微信硬件公众平台的智能控
程
9164

微信硬件平台的基础接入和硬件
入分析
8980

以蓝牙开发的视觉解读微信Air
8848

Android Activity使用拾遗
8721

一张图读懂基于微信硬件平台的
构
8702

JD.COM 京东

无氟变频

一级能效 一键自清洁

直降900

7999

3月30日-4月3日 4月4日-00:00

¥4499.00 支付尾款

联系我们

请扫描二维码联系
webmaster@
400-660-011



I2C总线的管理类似于平台总线，在注册I2C设备驱动接口i2c_add_driver中也会匹配其管理的I2C设备链表元素，匹配成功后则会调用i2c_driver的probe接口。有关总线、设备和驱动的关系请参看《从需求的角度去理解Linux：总线、设备和驱动》。

继续跟踪tsc2007_probe之前先看看input-device的数据结构：

```
struct input_dev {
    const char *name;
    const char *phys;
    const char *uniq;    input-device 名称, ID等属性
    struct input_id id;

    unsigned long evbit[BITS_TO_LONGS(EV_CNT)];    支持的事件类型, 如同步事件、按键、绝对值等
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];
    unsigned long relbit[BITS_TO_LONGS(REL_CNT)];
    unsigned long absbit[BITS_TO_LONGS(ABS_CNT)];
    unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)];
    unsigned long ledbit[BITS_TO_LONGS(LED_CNT)];
    unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];
    unsigned long ffbbit[BITS_TO_LONGS(FF_CNT)];
    unsigned long swbit[BITS_TO_LONGS(SW_CNT)];

    int (*open)(struct input_dev *dev);    设备操作接口
    void (*close)(struct input_dev *dev);
    int (*flush)(struct input_dev *dev, struct file *file);
    int (*event)(struct input_dev *dev, unsigned int type, unsigned int code,

    struct input_handle *grab;    设备通过input-handle关联到handler
    struct device dev;
    struct list_head h_list;    内嵌基础设备类
    struct list_head node;    关联到全局链表
    ...
};
```

继续跟踪tsc2007_probe：

```
static int tsc2007_probe(struct i2c_adapter *adapter, const struct i2c_device_id *id)
{
    struct i2c_client *client;
    struct tsc2007 *ts;
    struct input_dev *input_dev;
    //省去i2c设备相关的适配检测代码

    input_dev = input_allocate_device();

    input_dev->name = "TSC2007 Touchscreen";
    input_dev->phys = ts->phys;
    input_dev->id.bustype = BUS_I2C;
    //注册按键事件和绝对值坐标事件
    input_dev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_ABS);
    input_dev->keybit[BIT_WORD(BTN_TOUCH)] = BIT_MASK(BTN_TOUCH);

    input_set_abs_params(input_dev, ABS_X, 0, MAX_12BIT, 0, 0);
    input_set_abs_params(input_dev, ABS_Y, 0, MAX_12BIT, 0, 0);
    input_set_abs_params(input_dev, ABS_PRESSURE, 0, MAX_12BIT, 0, 0);

    //注册触屏引起的外部硬件中断
    err = request_irq(ts->irq, tsc2007_irq, 0, client->dev.driver->name, ts);
    //注册输入设备input-device
    err = input_register_device(input_dev);
}
```

继续展开input_register_device接口：

--driver/input/input.c

```
static LIST_HEAD(input_dev_list);

int input_register_device(struct input_dev *dev)
{
    static atomic_t input_no = ATOMIC_INIT(0);
    struct input_handler *handler;
    const char *path;
    int error;

    /* Every input device generates EV_SYN/SYN_REPORT events. */
    __set_bit(EV_SYN, dev->evbit);
    dev_set_name(&dev->dev, "input%d",
        (unsigned long) atomic_inc_return(&input_no) - 1);
    //设备驱动模型视图增加一个input设备
    error = device_add(&dev->dev);
    //连接到全局输入设备链表
    list_add_tail(&dev->node, &input_dev_list);
    //匹配全局事件处理handler链表, 匹配支持该输入设备的handler
    list_for_each_entry(handler, &input_handler_list, node)
        input_attach_handler(dev, handler);
}
```

4.input-core关联匹配input-device和input-handler

在input_register_handler和input_register_device最后都会使用input_attach_handler接口来匹配输入设备和对应的事件处理器。

```
static int input_attach_handler(struct input_dev *dev, struct input_handler *handler)
{
    const struct input_device_id *id;
    int error;

    //匹配handler和device的ID, event_handler默认处理所有的事件类型设备
    id = input_match_device(handler, dev);
    //匹配成功后调用handler的connect接口, 即evdev_connect
    error = handler->connect(handler, dev, id);
}
```

继续跟踪evdev_connect:

--driver/input/evdev.c

```
//每类handler管理最多32个设备
#define EVDEV_MINORS 32
//evdev_table的每一个元素对应一个input-device
static struct evdev *evdev_table[EVDEV_MINORS];
static int evdev_connect(struct input_handler *handler, struct input_dev *dev,
    const struct input_device_id *id)
{
    struct evdev *evdev;
    int minor;
    //从0开始找到一个未用的索引号
    for (minor = 0; minor < EVDEV_MINORS; minor++)
        if (!evdev_table[minor])
            break;
    evdev = kzalloc(sizeof(struct evdev), GFP_KERNEL);

    INIT_LIST_HEAD(&evdev->client_list);
    spin_lock_init(&evdev->client_lock);
    mutex_init(&evdev->mutex);
    init_waitqueue_head(&evdev->wait);
    //这里我们能看到最终生成的设备文件名event0、event1等等
    dev_set_name(&evdev->dev, "event%d", minor);
    evdev->exist = 1;
    evdev->minor = minor;
    evdev->handle.dev = input_get_device(dev);
    evdev->handle.name = dev_name(&evdev->dev);
    evdev->handle.handler = handler;
    evdev->handle.private = evdev;
    //在设备驱动视图/sys/class/input/和/sys/devices/目录下产生eventx设备
    //最终依赖event机制和mdev在/dev目录生成对应的设备文件
    evdev->dev.devt = MKDEV(INPUT_MAJOR, EVDEV_MINOR_BASE + minor);
    evdev->dev.class = &input_class;
    evdev->dev.parent = &dev->dev;
    evdev->dev.release = evdev_free;
    device_initialize(&evdev->dev);
    error = input_register_handle(&evdev->handle);
    error = evdev_install_chrdev(evdev);
}
```

Struct evdev evdev_table代表evdev_handler所管理的底层input-device (通过input-handle管理) 和应用层已打开该设备的进程、同步的相关结构和消息队列 (evdev_client记录)。

```

struct evdev {
    int exist;
    int open;
    int minor;
    //input_handle管理handler对应的device
    struct input_handle handle;
    wait_queue_head_t wait;
    //记录应用层已经open过该eventx设备的进程、同步异步相关结构和消息池
    struct evdev_client *grab;
    struct list_head client_list;
    spinlock_t client_lock; /* protects client_list */
    struct mutex mutex;
    struct device dev;
};

```



input-handle关联input-device和input-handler一目了然。

```

struct input_handle {

    void *private;
    int open;
    const char *name;
    struct input_dev *dev;
    struct input_handler *handler;

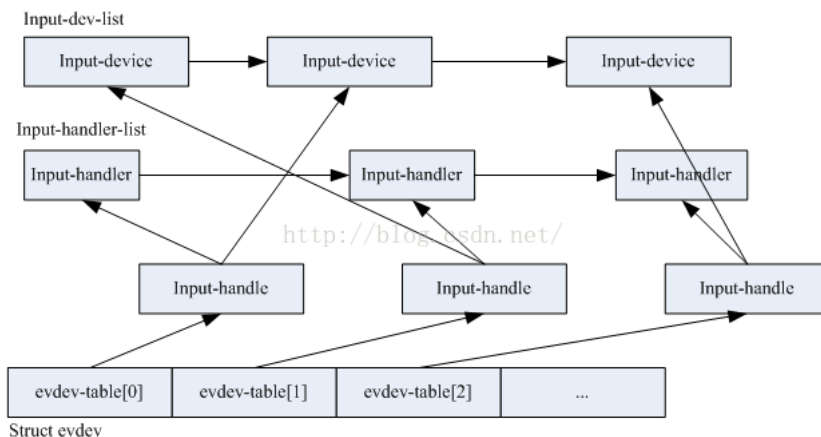
    struct list_head d_node;
    struct list_head h_node;
};

```

所以input_register_handle的接口很容易想到是通过input-handle通过自身的d-node和h-node关联到input-device和input-handler实例中。这样通过input-handler可以快速找到input-device，通过input-device也可以快速找到input-handler。

至于evdev_install_chrdev即是将一个evdev实例记录到evdev_table数组，宣告其存在。

至此，我们可以得到以下evdev-handler管理下的示意图：



四、应用open过程

假设触摸屏驱动在注册输入设备过程中生成/dev/input/event0设备文件。我们来跟踪打开这个设备的过程。

Open("/dev/input/event0")

1.vfs_open打开该设备文件，读出文件的inode内容，得到主设备号13和次设备号64。

2.chardev_open 字符设备驱动框架的open根据主设备号13得到输入子系统的input_fops操作集。

3.input_fops->open, 即input_open_file

```

static int input_open_file(struct inode *inode, struct file *file)
{
    struct input_handler *handler;
    const struct file_operations *old_fops, *new_fops = NULL;
    //iminor(inode)即是次设备号64，所以是input_table[2]，即evdev_handler
    handler = input_table[iminor(inode) >> 5];
    if (handler)
        //即evdev_handler的evdev_fops
        new_fops = fops_get(handler->fops);
    //即evdev_fops的evdev_open
    err = new_fops->open(inode, file);
}

```


4.继续跟踪input-handler层的evdev-open,至此evdev不仅关联了底层具体的input-device,而且记录了应用层进程打开该设备的信息。之后input-device产生的消息可以传递到evdev的client中的消息队列。便于上层读取。

```
static int evdev_open(struct inode *inode, struct file *file)
{
    struct evdev *evdev;
    struct evdev_client *client;
    //i=64-64=0
    int i = iminor(inode) - EVDEV_MINOR_BASE;
    evdev = evdev_table[i];

    client = kzalloc(sizeof(struct evdev_client), GFP_KERNEL);
    spin_lock_init(&client->buffer_lock);
    //根据当前进程产生client的名称
    snprintf(client->name, sizeof(client->name), "%s-%d",
             dev_name(&evdev->dev), task_tgid_vnr(current));
    wake_lock_init(&client->wake_lock, WAKE_LOCK_SUSPEND, client->name);
    //将代表打开该设备的进程相关的数据结构client和evdev绑定
    client->evdev = evdev;
    evdev_attach_client(evdev, client);
    //执行input-device层的open
    error = evdev_open_device(evdev);
    if (error)
        return error;
}
```

5. input-device层的open。

```
input_open_device(&evdev->handle)

int input_open_device(struct input_handle *handle)
{
    struct input_dev *dev = handle->dev;
    if (!dev->users++ && dev->open)
        retval = dev->open(dev);
}
```

实际上, tsc2007驱动并没有定义input_dev的open接口。

五、触屏消息传递过程

1. open获得的fd句柄对应的file_operations是evdev_handler的evdev_fops。因此read接口最终会调用到evdev_fops的read接口,即evdev_read。接下来我们来跟踪这个接口的实现过程。我们先看看struct evdev的成员evdev_client的定义,其即是代表打开该输入设备的进程相关的数据结构。

```
struct evdev_client {
    //消息队列, input-core转发的底层触屏消息会放到这里
    //上层应用进程也会从这里读取
    struct input_event buffer[EVDEV_BUFFER_SIZE];
    //指示消息队列的指针
    int head;
    int tail;
    //读写锁
    spinlock_t buffer_lock;
    //异步通知上层等待进程
    struct fasync_struct *fasync;
    //管理的evdev
    struct evdev *evdev;
    struct list_head node;
    struct wake_lock wake_lock;
    char name[28];
};
```

2. evdev_read

```
static ssize_t evdev_read(struct file *file, char __user *buffer,
                        size_t count, loff_t *ppos)
{
    struct evdev_client *client = file->private_data;
    struct evdev *evdev = client->evdev;
    struct input_event event;
    int retval;
    //判断读取长度是否小于单个input-event的长度
    if (count < input_event_size())
        return -EINVAL;
    //判断消息队列消息是否为空
    if (client->head == client->tail && evdev->exist &&
        (file->f_flags & O_NONBLOCK))
        return -EAGAIN;
    //等待消息队列不为空的事件
    retval = wait_event_interruptible(evdev->wait,
        client->head != client->tail || !evdev->exist);
    if (retval)
        return retval;

    if (!evdev->exist)
        return -ENODEV;
    //将消息队列的消息取出并通过copy_to_user填到buffer返回
    while (retval + input_event_size() <= count &&
        evdev_fetch_next_event(client, &event)) {

        if (input_event_to_user(buffer + retval, &event))
            return -EFAULT;

        retval += input_event_size();
    }
}
```

3. 假设消息队列为空时，则上层进程将会睡眠，直到被唤醒再进行消息读取。谁来唤醒它呢？由底层input-device的硬件中断发起，最终将触屏消息送达该消息队列后即会发出唤醒信号。tsc2007_probe中注册的外部硬件中断服务函数即是发起者。

来看看该中断服务函数tsc2007_irq：

```
static irqreturn_t tsc2007_irq(int irq, void *handle)
{
    struct tsc2007 *ts = handle;

    if (!ts->get_pendown_state || likely(ts->get_pendown_state())) {
        disable_irq_nosync(ts->irq);
        //延时防抖动，超时即调用ts->work
        schedule_delayed_work(&ts->work,
            msecs_to_jiffies(TS_POLL_DELAY));
    }
}
```

ts-work即是tsc2007_work：

```
static void tsc2007_work(struct work_struct *work)
{
    struct tsc2007 *ts =
        container_of(to_delayed_work(work), struct tsc2007, work);
    struct ts_event tc={0};

    //通过I2C接口读出触屏坐标
    tsc2007_read_values(ts, &tc);
    //报告x, y坐标和压力值
    input_report_abs(input, ABS_X, tc.x);
    input_report_abs(input, ABS_Y, tc.y);
    input_report_abs(input, ABS_PRESSURE, rt);
    //发出同步事件
    input_sync(input);
}
```

跟踪input_report_abs接口：

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

void input_event(struct input_dev *dev, unsigned int type,
{
    //input-device初始化声明是否支持绝对值坐标(EV_ABS)事件
    if (is_event_supported(type, dev->absbit, EV_MAX)) {
        //产生系统随机数，对input没有影响
        add_input_randomness(type, code, value);
        input_handle_event(dev, type, code, value);
    }
}

static void input_pass_event(struct input_dev *dev, unsigned
{
    //即输入设备对应的input-handle，其handler就是evdev_handler
    handle = rcu_dereference(dev->grab);
    if (handle)
        //即evdev_event
        handle->handler->event(handle, type, code, value);
}

static void input_handle_event(struct input_dev *dev, unsigned i
{
    switch (type) {
    case EV_ABS:
        //检测input-device初始化声明是否支持该编码ABS_X
        if (is_event_supported(code, dev->absbit, ABS_MAX)) {
            //误差校准
            value = input_defuzz_abs_event(value,
                dev->abs[code], dev->absfuzz[code]);
            if (dev->abs[code] != value) {
                dev->abs[code] = value;
                //该消息要传递给对应的input-handler
                disposition = INPUT_PASS_TO_HANDLERS;
            }
        }
        //非同步事件
        if (disposition != INPUT_IGNORE_EVENT && type != EV_SYN)
            dev->sync = 0;
        //明显不是传递给input-device层处理，如果是鼠标电灯信号就会
        if ((disposition & INPUT_PASS_TO_DEVICE) && dev->event
            dev->event(dev, type, code, value);
        //继续转发
        if (disposition & INPUT_PASS_TO_HANDLERS)
            input_pass_event(dev, type, code, value);
    }
}
```

1

有一点需要注意，每次触屏消息产生后，在tsc2007_work中要input-report-abs报告x坐标，y坐标和压力值，最后再通过input-sync接口发出同步事件，向上层应用发出异步通知进行读取。

敬请关注微信公众号：嵌入式企鹅圈，百分百原创，嵌入式Linux和物联网开发技术经验，资深嵌入式软件架构师撰文。



本文已收录于以下专栏：[嵌入式Linux内核驱动情景分析](#)

写下你的评论...

[回复](#)

顶一个

leesagacious 2015-12-13 10:58:35 697engshengq23 2012-05-10 15:24:16 3175

你知道AI人工智能工程师有多缺乏吗？

21世纪什么最贵？人才啊.....



👍

1

[Linux]input 子系统学习笔记（简单范例和四个基本函数）

入子系统是为了将输入设备的功能呈现给应用程序。 它支持 鼠标、键盘、蜂鸣器、触摸屏、传感器等需要不断上报数据的设备。 分析了四个函数： 1. input_allocate_device 在内存中...

💬

dearsq

2016-05-19 14:54:51

📖 5984

Linux Input子系统浅析（二）-- 模拟tp上报键值

通过前一节的分析得到，linux Input子系统上传数据本质上是将input_dev的数据，上报给input_handler，当用户读入event时，驱动层只需要利用copy_to_user将数...

👤 xiaopangzi313

2016-08-31 12:36:04

📖 1368

舆情监测系统

舆情监测系统公司排名

百度广告



Linux input子系统

一、Input子系统分层思想 input子系统是典型的字符设备。首先分析输入子系统的工作机理。底层设备(按键、触摸等)发生动作时，产生一个事件(抽象)，CPU读取事件数据放入缓冲区，字符设备驱...

👤 bianyuke

2015-12-03 09:45:48

📖 360

Linux/Android——input子系统核心 (三)

之前的博客有涉及到linux的input子系统，这里学习记录一下input模块. input子系统，作为管理输入设备与系统进行交互的中枢，任何的输入设备驱动都要通过input向内核注册其设备，常...

👤 jscese

2014-12-26 15:10:07

📖 5104

linux input输入子系统分析《四》：input子系统整体流程全面分析

主要讲述本人在学习Linux内核input子系统的全部过程，如有分析不当，多谢指正。以下方式均可联系，文章欢迎转载，保留联系信息，以便交流。 邮箱：eabi010@gmail.com 主页：www...

👤 wangpengqi

2013-01-05 23:01:49

📖 4633

linux input输入子系统分析《一》：初识input输入子系统

主要讲述本人在学习Linux内核input子系统的全部过程，如有分析不当，多谢指正。以下交流方式，文章欢迎转载，保留联系信息，以便交流。 邮箱：eabi010@gmail.com 主页：www.i...

👤 ielife

2012-07-29 14:33:24

📖 11687

Linux Input子系统--概述

水平有限，描述不当之处还请指出，转载请注明出处http://blog.csdn.net/vanbreaker/article/details/7714188 输入设备总类繁杂，包括按键...

👤 vanbreaker

2012-07-04 19:54:25

📖 6064

Linux input子系统分析之一:软件层次

输入输出是用户和产品交互的手段，因此输入驱动开发在Linux驱动开发中很常见。同时，input子系统的分层架构思想在Linux驱动设计中极具代表性和先进性，因此对Linux input子系统。 ...

👤 yueqian_scut

2015-08-23 10:49:35

📖 3295

https://blog.csdn.net/yueqian_scut/article/details/48026955

9/10

linux input 子系统分析 三

YAOZHENGUO2006 2011-09-14 19:48:59 7636

linux input子系统分析--子系统核心.事件处理层.事件传递过程 一. 输入子系统核心分析。 1.输入子系统核心对应与/drivers/in
put/input.c文件,这个也...

Linux驱动框架之——Input子系统

fanwenjieok 2014-08-12 00:23:24 882

Linux驱动框架之——Input子系统 输入(Input)子系统是分层架构的，总共分为 3 层，从上到下分别是：事件处理层(Event Ha
ller)、输入子系统核心层(Input ...

input子系统框架分析

duan_xiaosu 2017-03-30 15:47:50 405

1. input框架介绍： Linux input子系统主要分为三层：驱动、输入core、事件处理层。 驱动根据core提供的接口，向上报告发生
的动作（input_report_*）。 cor...

input子系统分析

ly601579033 2015-08-27 11:29:50 695

linux系统中，input输入子系统驱动主要可以分为：设备驱动层、input core层和input handler事件处理层。 设备驱动层提供具
体用户设备驱动，由struct input_dev ...

Linux input子系统分析之二：深入剖析input_handler、input_core、input_device


yueqian_scut 2015-09-28 23:42:49 3131

本文继续在《Linuxinput子系统分析之一：软件分层》的基础上继续深入研究Linux输入子系统的分层架构思想以及其实现。软件
分层探讨的是输入消息从底层硬件到内核、应用层的消息传递和使用过程，而本文...

舆情监测系统

舆情监控系统有哪些

百度广告



闲聊linux中的input设备(转)

beckdon 2016-01-28 10:48:46 706

转自：http://blog.csdn.net/lmm670/article/details/6080998 用过linux的哥们都知道，linux所有的设备都是以文件的形式实
现的，要访问一...

Linux内核Input输入子系统浅解

G_linuxer_ 2016-07-01 16:03:37 1828

Linux输入设备总类繁杂，常见的包括有按键、键盘、触摸屏、鼠标、摇杆等等，他们本身就是字符设备，而linux内核将这些设备
的共同性抽象出来，简化驱动开发建立了一个input子系统。子系统共分为三层， ...

linux驱动之input子系统之按键驱动编写流程(三)

u013256018 2015-11-21 14:46:09 858

的撒旦撒

linux input系列-----input的初始化

u013308744 2016-10-19 11:39:24 674

static int __init input_init(void) { int err; err = class_register(&input_class); err = input_pr...

linux input 子系统分析

coder_jack 2015-05-01 18:28:25 636

转自：http://blog.csdn.net/yaozhenguo2006/article/details/6769482 linux input子系统分析--概述与数据结构 ...