第7章 输入设备驱动

内核的输入子系统是为了对分散的、多种不同类别的输入设备（如键盘、鼠标、跟踪球、操纵杆、辊轮、触摸屏、加速计和手写板）进行统一处理的驱动。输入子系统带来了如下好处：

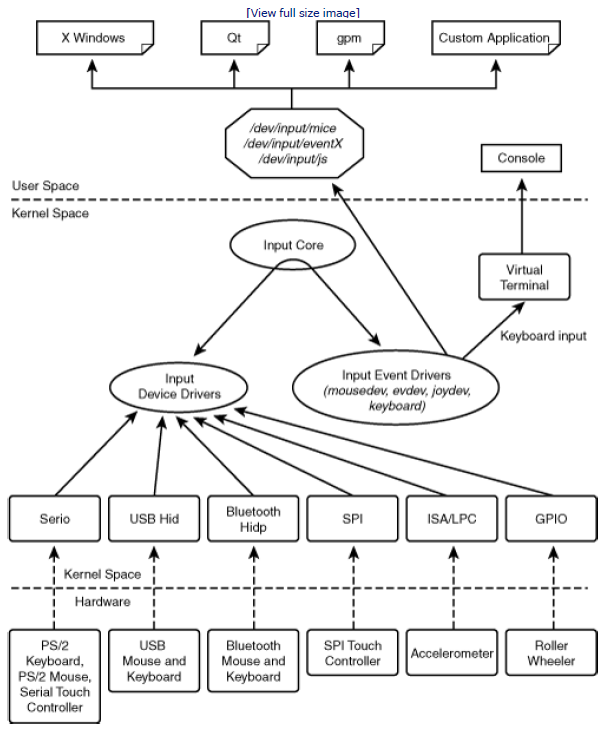
· 统一了物理形态各异的相似的输入设备的处理功能。例如，各种鼠标，不论PS/2、USB，还是蓝牙，都被同样处理。

· 提供了用于分发输入报告给用户应用程序的简单的事件（event）接口。你的驱动不必创建、管理/dev节点以及相关的访问方法。因此它能很方便的调用输入API以发送鼠标移动、键盘按键，或触摸事件给用户空间。X Windows这样的应用程序能够无缝地运行于输入子系统提供的event接口之上。

· 抽取出了输入驱动的通用部分，简化了驱动，并提供了一致性。例如，输入子系统提供了一个底层驱动（成为serio）的集合，支持对串口和键盘控制器等硬件输入设备的访问。

图7.1展示了输入子系统的操作。此子系统包括一前一后运行的两类驱动：事件驱动和设备驱动。事件驱动负责和应用程序的接口，而设备驱动负责和底层输入设备的通信。鼠标事件产生者mousedev，是前者的实例；而PS/2鼠标驱动是后者的实例。事件驱动和设备驱动都可以利用输入子系统的高效、可重用的核心提供的服务。

图 7.1. 输入子系统



事件驱动是标准的，对所有的输入类都是可用的，所以你更可能的是实现设备驱动而不是事件驱动。你的设备驱动可以利用一个已经存在的、合适的事件驱动通过输入核心和用户应用程序接口。需要注意的是本章使用的名辞“设备驱动”指的是输入设备驱动，而不是输入事件驱动。

输入事件驱动

输入子系统提供的事件接口已经发展成为很多图形窗口系统理解的标准。事件驱动提供一个硬件无关的抽象，以和输入设备交互；如同帧缓冲接口（在第12章《视频设备驱动》中讨论）提供一个通用的机制以和显示设备通信一样。事件驱动和帧缓冲驱动一起，将图形用户接口（GUI）和各种各样的底层硬件隔离开来。

Evdev接口

Evdev是一个通用的输入事件驱动。Evdev产生的每个事件包都有如下格式，定义于include/linux/input.h：

struct input\_event {

struct timeval time; /\* Timestamp \*/

\_\_u16 type; /\* Event Type \*/

\_\_u16 code; /\* Event Code \*/

\_\_s32 value; /\* Event Value \*/

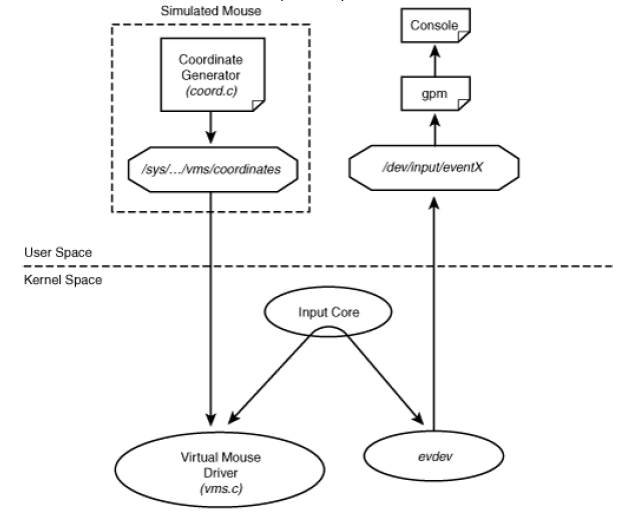
};

为了学习如何使用evdev，让我们来实现一个虚拟鼠标的输入设备驱动。

设备例子：虚拟鼠标

我们的虚拟鼠标工作过程如下：一个应用程序（coord.c）模拟鼠标移动，并通过一个sysfs节点/sys/devices/platform/vms/coordinates分发坐标信息给虚拟鼠标驱动（vms.c）。虚拟鼠标驱动（vms驱动）通过evdev向上层传送这些移动信息。图7.2展示了详细过程：

图 7.2. 虚拟鼠标的输入驱动



通用目的鼠标（gpm）是服务器，让你在文本模式下使用鼠标，而无需X服务器。Gpm能够理解evdev消息，因此vms驱动能够直接和其通信。一切就绪后，你将会看到随着由coord.c产生虚拟鼠标移动，光标在屏幕上跳动。

清单7.1包含coord.c，它连续产生随机的X和Y坐标。鼠标与操纵杆或触摸屏不同，它产生的是相对坐标[MS1] 而非绝对坐标，这就是coord.c所做的工作。vms驱动在清单7.2中。

清单 7.1. 模拟虚拟移动的应用程序（coord.c）

Code View:

#include

int

main(int argc, char \*argv[])

{

int sim\_fd;

int x, y;

char buffer[10];

/\* Open the sysfs coordinate node \*/

sim\_fd = open("/sys/devices/platform/vms/coordinates", O\_RDWR);

if (sim\_fd

#include

#include

#include

#include

struct input\_dev \*vms\_input\_dev; /\* Representation of an input device \*/

static struct platform\_device \*vms\_dev; /\* Device structure \*/

/\* Sysfs method to input simulated

coordinates to the virtual

mouse driver \*/

static ssize\_t

write\_vms(struct device \*dev,

struct device\_attribute \*attr,

const char \*buffer, size\_t count)

{

int x,y;

sscanf(buffer, "%d%d", &x, &y);

/\* Report relative coordinates via the

event interface \*/

input\_report\_rel(vms\_input\_dev, REL\_X, x);

input\_report\_rel(vms\_input\_dev, REL\_Y, y);

input\_sync(vms\_input\_dev);

return count;

}

/\* Attach the sysfs write method \*/

DEVICE\_ATTR(coordinates, 0644, NULL, write\_vms);

/\* Attribute Descriptor \*/

static struct attribute \*vms\_attrs[] = {

&dev\_attr\_coordinates.attr,

NULL

};

/\* Attribute group \*/

static struct attribute\_group vms\_attr\_group = {

.attrs = vms\_attrs,

};

/\* Driver Initialization \*/

int \_\_init

vms\_init(void)

{

/\* Register a platform device \*/

vms\_dev = platform\_device\_register\_simple("vms", -1, NULL, 0);

if (IS\_ERR(vms\_dev)) {

PTR\_ERR(vms\_dev);

printk("vms\_init: error\n");

}

/\* Create a sysfs node to read simulated coordinates \*/

sysfs\_create\_group(&vms\_dev->dev.kobj, &vms\_attr\_group);

/\* Allocate an input device data structure \*/

vms\_input\_dev = input\_allocate\_device();

if (!vms\_input\_dev) {

printk("Bad input\_alloc\_device()\n");

}

/\* Announce that the virtual mouse will generate

relative coordinates \*/

set\_bit(EV\_REL, vms\_input\_dev->evbit);

set\_bit(REL\_X, vms\_input\_dev->relbit);

set\_bit(REL\_Y, vms\_input\_dev->relbit);

/\* Register with the input subsystem \*/

input\_register\_device(vms\_input\_dev);

printk("Virtual Mouse Driver Initialized.\n");

return 0;

}

/\* Driver Exit \*/

void

vms\_cleanup(void)

{

/\* Unregister from the input subsystem \*/

input\_unregister\_device(vms\_input\_dev);

/\* Cleanup sysfs node \*/

sysfs\_remove\_group(&vms\_dev->dev.kobj, &vms\_attr\_group);

/\* Unregister driver \*/

platform\_device\_unregister(vms\_dev);

return;

}

module\_init(vms\_init);

module\_exit(vms\_cleanup);

让我们仔细阅读清单7.2中的代码。初始化期间，vms驱动注册自身为输入设备驱动。为此，它首先使用内核API input\_allocate\_device()分配input\_dev结构：

vms\_input\_dev = input\_allocate\_device();

然后，声明虚拟鼠标产生相对性事件：

set\_bit(EV\_REL, vms\_input\_dev->evbit); /\* 事件类型为EV\_REL \*/

下一步，声明虚拟鼠标产生的事件的编码：

set\_bit(REL\_X, vms\_input\_dev->relbit); /\* Relative 'X' movement \*/

set\_bit(REL\_Y, vms\_input\_dev->relbit); /\* Relative 'Y' movement \*/

如果你的虚拟鼠标也能产生按钮点击事件，还需要将其加入vms\_init()：

set\_bit(EV\_KEY, vms\_input\_dev->evbit); /\* Event Type is EV\_KEY \*/

set\_bit(BTN\_0, vms\_input\_dev->keybit); /\* Event Code is BTN\_0 \*/

最后，进行注册：

input\_register\_device(vms\_input\_dev);

write\_vms()是sysfs中的store()方法，和/sys/devices/platform/vms/coordinates相关联。当coord.c写入X/Y坐标对进此文件时，write\_vms()做如下操作：

input\_report\_rel(vms\_input\_dev, REL\_X, x);

input\_report\_rel(vms\_input\_dev, REL\_Y, y);

input\_sync(vms\_input\_dev);

第一条语句会产生REL\_X事件，或设备在X方向的相对移动。第二条语句产生REL\_Y事件，或设备在Y方向的相对移动。input\_sync()表明此事件已经完成，因此输入子系统将这两个事件组成一个evdev包，并通过/dev/input/eventX发送出去，eventX中的X是分配给vms驱动的接口序号。读该文件的应用程序将以前面描述的input\_event格式接收事件包。为了将gpm关联至此事件接口，从而追逐光标的跳动，需要做如下操作：

bash> gpm -m /dev/input/eventX -t evdev

在“触摸控制器”和后面的“加速度传感器”章节中讨论的ADS7846触摸控制器驱动以及加速度传感器驱动，也都使用了evdev。

更多的事件接口

vms驱动利用通用的evdev事件接口，但像键盘、鼠标和触摸屏这些输入设备必须使用定制的事件驱动。在讨论相应的设备驱动时，我们将一一学习。

为了编写你自己的事件驱动，并通过/dev/input/mydev提供给用户空间，你必须利用input\_handler结构体，并将其向输入核心注册：

Code View:

static struct input\_handler my\_event\_handler = {

.event = mydev\_event, /\* Handle event reports sent by

input device drivers that use

this event driver's services \*/

.fops = &mydev\_fops, /\* Methods to manage

/dev/input/mydev \*/

.minor = MYDEV\_MINOR\_BASE, /\* Minor number of

/dev/input/mydev \*/

.name = "mydev", /\* Event driver name \*/

.id\_table = mydev\_ids, /\* This event driver can handle

requests from these IDs \*/

.connect = mydev\_connect, /\* Invoked if there is an

ID match \*/ .disconnect = mydev\_disconnect, /\* Called when the driver unregisters

\*/

};

/\* Driver Initialization \*/

static int \_\_init

mydev\_init(void)

{

/\* ... \*/

input\_register\_handler(&my\_event\_handler);

/\* ... \*/

return 0;

}

mousedev（drivers/input/mousedev.c）的具体实现中可以看到完整的例子。

输入设备驱动

下面，让我们将注意转向键盘、鼠标以及触摸屏这些通用输入设备的驱动上。但首先，让我们快速浏览一下那些输入驱动可以利用的、现成的硬件访问方法。

Serio

Serio层提供了访问老式输入硬件（如i8042兼容键盘控制器和串口）的库例程。PS/2键盘和鼠标与前者相连，串行触摸控制器和后者连接在一起。为了与serio提供服务的硬件通信，如发送命令给PS/2鼠标，需要用serio\_register\_driver()向serio注册规定的回调例程。

为了添加新的驱动作为serio的一部分，需要用serio\_register\_port()注册open()/close()/start()/stop()/write()入口函数。drivers/input/serio/serport.c中可以看到具体的例子。

正如你在图7.1中所看到的，serio仅仅是访问底层硬件的一条路径。有些输入设备驱动依赖的是底层的总线层的支持，例如USB和SPI。

键盘

键盘多种多样――从老式的PS/2，到USB，蓝牙以及红外等。每种类型都有特定的输入设备驱动，但所有的都使用相同的键盘事件驱动，以确保提供给用户一致的接口。然而，和其它的事件驱动相比，键盘事件驱动有其独特之处：它传送数据给另一个内核子系统（tty层），而不是通过/dev节点传送给用户空间。

PC键盘

PC键盘（也成为PS/2键盘或AT键盘）通过i8042兼容键盘控制器与处理器接口。桌上电脑通常由专用的键盘控制器，而笔记本电脑的键盘接口由通用嵌入式控制器负责（参见第20章“嵌入式控制器”中的“更多的设备及驱动”）。当你在PC键盘上按下一个键时，会产生以下步骤：

1. 键盘控制器（或嵌入式控制器）扫描键盘矩阵，译码，并做按键去抖处理。

2. 键盘设备驱动在serio的帮助下，针对每个键的按下与释放，从键盘控制器读取原始的扫描码。按下与释放之间的区别在最高位，在释放时最高位被置位。例如，当“a”键被按下、释放后，会产生一对扫描码：0x1e和0x9e。专用键用0xE0做转义，因此按下、释放右箭头键，产生的序列为（0xE0 0x4D 0xE0 0xCD）。你可以使用showkey工具观察控制器发出的扫描码（跟在－>符号后面的是对前面内容的解释）：

bash> showkey -s

kb mode was UNICODE

[ if you are trying this under X, it might not work since

the X server is also reading /dev/console ]

press any key (program terminates 10s after last

keypress)...

[MS2] ...

0x1e 0x9e －> "a" 键按下与松开

3. 基于输入模式，键盘设备驱动转换接收到的扫描码为键值。为了查看和“a”键对应的键值：

bash> showkey

...

keycode 30 press －> A push of the "a" key

keycode 30 release －> Release of the "a" key

为了向上报告键值，驱动产生一个输入事件，将控制权交给键盘事件驱动。

4. 根据加载的键盘映射，键盘事件驱动进行键值翻译。（查看loadkeys的man帮助，以及在/lib/kbd/keymaps中提供的map文件）。它检查翻译后的键值是否和虚拟控制台或系统重启等相联系。添加如下代码至键盘事件驱动（drivers/char/keyboard.c）的Ctrl+Alt+Del处理程序，可将Ctrl+Alt+Del的行为设置为点亮CAPSLOCK和NUMLOCK灯，而不是重启系统，

static void fn\_boot\_it(struct vc\_data \*vc,

struct pt\_regs \*regs)

{

+ set\_vc\_kbd\_led(kbd, VC\_CAPSLOCK);

+ set\_vc\_kbd\_led(kbd, VC\_NUMLOCK);

- ctrl\_alt\_del();

}

5. 对于一般的键，译码后得到的键值被送至相联的虚拟终端，以及N\_TTY线路规程（在第6章“串行驱动”中讨论过虚拟终端与线路规程）。这些由drivers/char/keyboard.c中的代码完成：

/\* Add the keycode to flip buffer \*/

tty\_insert\_flip\_char(tty, keycode, 0);

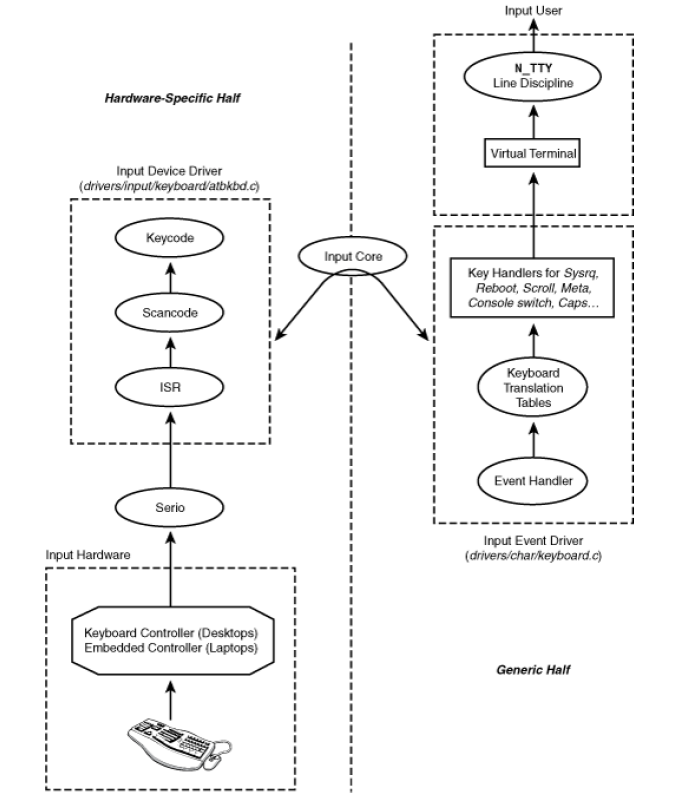
/\* Schedule \*/

con\_schedule\_flip(tty);

N\_TTY线路规程处理从键盘接收的输入，回送至虚拟控制台，以使用户空间的应用程序从与虚拟控制台相连的/dev/ttyX节点读取字符。

图7.3显示了从你按下键盘开始，到回显至虚拟控制台的整个过程的数据流。图的左半部是和特定硬件相关的，右半部是通用的。按照输入子系统的设计目的，底层硬件接口对键盘事件驱动和tty层是透明的。输入核心和定义明确的事件接口将input用户程序和复杂的硬件隔离开来。

图 7.3. PS/2兼容键盘的数据流



USB与蓝牙键盘

USB规范中有关人机接口设备（HID）的部分规定了USB键盘、鼠标、小键盘(Keypad)以及其他输入外设使用的通信协议。在Linux上，它们是通过usbhid USB客户端驱动来实现的，它负责USB HID类（0x03）设备。Usbhid注册自身作为输入设备驱动。它和输入API保持一致，并报告输入事件至相连的HID。

为了理解USB键盘的代码流，回到图7.3中，并修改左半部中的硬件相关部分。将输入硬件框中的键盘控制器用USB控制器代替，serio用USB核心层代替，输入设备驱动框用usbhid驱动代替即可。

对于蓝牙键盘，在图7.3中将键盘控制器用蓝牙芯片代替，serio用蓝牙核心层代替，输入设备驱动框用蓝牙hidp驱动代替。

USB和蓝牙在第11章“通用串行总线”和第16章“无线Linux”中分别详细讨论。

鼠标

类似于键盘，鼠标接口选项多种多样。让我们从通用的看起。

PS/2鼠标

鼠标在X和Y坐标上产生相对移动，有一个或多个按钮，一些还有滚轮（scroll wheel）。PS/2兼容的老式鼠标依赖于serio层和底层的控制器交互。鼠标的输入事件驱动称为mousedev，通过/dev/input/mice报告鼠标事件给用户应用程序。

设备例子：辊轮鼠标（Roller Mouse）

为了感受真实的鼠标设备驱动，让我们将第4章“打下基础”中所讨论的辊轮（roller wheel）变换为通用PS/2鼠标的变种。“辊轮鼠标（roller mouse）”在Y轴产生一维的移动。辊轮（wheel）顺时针和逆时针的旋转分别产生正的和负的相对Y坐标（类似鼠标上的滚轮scroll wheel），按下辊轮则会产生鼠标左按钮事件。辊轮鼠标对于在智能手机、手持设备以及音乐播放器等设备上操作菜单是理想的选择。

辊轮鼠标设备驱动实现于清单7.3，工作于像X Windows这样的窗口系统。查看roller\_mouse\_init()可清楚此驱动如何实现类似鼠标的功能。不像第4章清单4.1中的辊轮驱动，辊轮鼠标驱动不需要read()和poll()方法，因为这些事件通过输入API来报告。辊轮中断处理例程roller\_isr()也做了相应的改变。中断处理例程中的事务管理使用了一个等待队列，一个自旋锁，以及store\_movement()例程用于支持read()和poll()。

清单7.3中开始的“+”和“-”指示了和第4章清单4.1中辊轮驱动实现的区别。

清单 7.3. 滚轮鼠标驱动

Code View:

+ #include

+ #include

+ /\* Device structure \*/

+ struct {

+ /\* ... \*/

+ struct input\_dev dev;

+ } roller\_mouse;

+ static int \_\_init

+ roller\_mouse\_init(void)

+ {

+ /\* Allocate input device structure \*/

+ roller\_mouse->dev = input\_allocate\_device();

+

+ /\* Can generate a click and a relative movement \*/

+ roller\_mouse->dev->evbit[0] = BIT(EV\_KEY) | BIT(EV\_REL);

+ /\* Can move only in the Y-axis \*/

+ roller\_mouse->dev->relbit[0] = BIT(REL\_Y);

+

+ /\* My click should be construed as the left button

+ press of a mouse \*/

+ roller\_mouse->dev->keybit[LONG(BTN\_MOUSE)] = BIT(BTN\_LEFT);

+ roller\_mouse->dev->name = "roll";

+

+ /\* For entries in /sys/class/input/inputX/id/ \*/

+ roller\_mouse->dev->id.bustype = ROLLER\_BUS;

+ roller\_mouse->dev->id.vendor = ROLLER\_VENDOR;

+ roller\_mouse->dev->id.product = ROLLER\_PROD;

+ roller\_mouse->dev->id.version = ROLLER\_VER;

+ /\* Register with the input subsystem \*/

+ input\_register\_device(roller\_mouse->dev);

+}

/\* Global variables \*/

- spinlock\_t roller\_lock = SPIN\_LOCK\_UNLOCKED;

- static DECLARE\_WAIT\_QUEUE\_HEAD(roller\_poll);

/\* The Roller Interrupt Handler \*/

static irqreturn\_t

roller\_interrupt(int irq, void \*dev\_id)

{

int i, PA\_t, PA\_delta\_t, movement = 0;

/\* Get the waveforms from bits 0, 1 and 2

of Port D as shown in Figure 7.1 \*/

PA\_t = PORTD & 0x07;

/\* Wait until the state of the pins change.

(Add some timeout to the loop) \*/

for (i=0; (PA\_t==PA\_delta\_t); i++){

PA\_delta\_t = PORTD & 0x07;

}

movement = determine\_movement(PA\_t, PA\_delta\_t);

- spin\_lock(&roller\_lock);

-

- /\* Store the wheel movement in a buffer for

- later access by the read()/poll() entry points \*/

- store\_movements(movement);

-

- spin\_unlock(&roller\_lock);

-

- /\* Wake up the poll entry point that might have

- gone to sleep, waiting for a wheel movement \*/

- wake\_up\_interruptible(&roller\_poll);

-

+ if (movement == CLOCKWISE) {

+ input\_report\_rel(roller\_mouse->dev, REL\_Y, 1);

+ } else if (movement == ANTICLOCKWISE) {

+ input\_report\_rel(roller\_mouse->dev, REL\_Y, -1);

+ } else if (movement == KEYPRESSED) {

+ input\_report\_key(roller\_mouse->dev, BTN\_LEFT, 1);

+ }

+ input\_sync(roller\_mouse->dev);

return IRQ\_HANDLED;

}

指点杆（Trackpoint）

指点杆是一个定点设备，在一些笔记本电脑上和PS/2类型的键盘集成在一起。此设备包括位于键盘中间的操作杆和位于空白键下方的鼠标按钮。指点杆的本质功能类似于鼠标，因此你可以用PS/2鼠标驱动对其操作。

不像普通的鼠标，指点杆提供更多的移动控制。你可以控制指点杆控制器改变其灵敏度和惯量等属性。内核有一个特别的驱动drivers/input/mouse/trackpoint.c，用于创建和管理相应的sysfs节点。对于所有的指点配置选项的设置，可在/sys/devices/platform/i8042/serioX/serioY/下查看。

触摸板（Touchpad）

触摸板是类似于鼠标的定点设备，在笔记本电脑上很常见。与传统的鼠标不同，触摸板没有移动组件。它能产生和鼠标兼容的相对坐标，但通常被操作系统用于在功能更强大的模式下产生绝对坐标。在绝对模式下的通信协议类似于PS/2鼠标协议，但不兼容于PS/2。

对于那些和基本PS/2鼠标协议各种的变种相一致的设备，基本的PS/2鼠标驱动能够对其提供支持。对于新鼠标协议，通过psmouse结构体可实现协议驱动，从而可在基本驱动的基础上添加对该新鼠标协议的支持。例如，如果你的笔记本电脑在绝对模式下使用Synaptics触摸板，基本的PS/2鼠标驱动使用Synaptics协议驱动的服务去解释数据流。为了更好的理解Synaptics协议和基本PS/2驱动共同工作的原理，请阅读下面从清单7.4收集的四个代码片段：

· PS/2鼠标驱动：drivers/input/mouse/psmouse-base.c，用支持的鼠标协议信息（包括Synaptics触摸板协议）实例化了psmouse\_protocol结构体。

· psmouse结构：定义于drivers/input/mouse/psmouse.h，和各种PS/2协议绑定在一起。

· synaptics\_init()：使用对应协议的处理函数地址填充psmouse结构体。

· 协议处理函数synaptics\_process\_byte()（由synaptics\_init()填充）：当serio感知到鼠标移动时，从中断上下文中被调用。如果查看synaptics\_process\_byte()，将会发现触摸板的移动通过mousedev报告给用户应用程序。

清单7.4. Synaptics 触摸板的PS/2鼠标协议驱动

Code View:

drivers/input/mouse/psmouse-base.c:/\* List of supported PS/2 mouse protocols \*/static struct psmouse\_protocol psmouse\_protocols[] = { { .type = PSMOUSE\_PS2, /\* The bare PS/2 handler \*/ .name = "PS/2", .alias = "bare",

.maxproto = 1, .detect = ps2bare\_detect, }, /\* ... \*/ { .type = PSMOUSE\_SYNAPTICS, /\* Synaptics TouchPad Protocol \*/ .name = "SynPS/2", .alias = "synaptics", .detect = synaptics\_detect, /\* Is the protocol detected? \*/ .init = synaptics\_init, /\* Initialize Protocol Handler \*/ }, /\* ... \*/} drivers/input/mouse/psmouse.h:/\* The structure that ties various mouse protocols together \*/struct psmouse { struct input\_dev \*dev; /\* The input device \*/ /\* ... \*/ /\* Protocol Methods \*/ psmouse\_ret\_t (\*protocol\_handler) (struct psmouse \*psmouse, struct pt\_regs \*regs); void (\*set\_rate)(struct psmouse \*psmouse, unsigned int rate); void (\*set\_resolution) (struct psmouse \*psmouse, unsigned int resolution); int (\*reconnect)(struct psmouse \*psmouse); void (\*disconnect)(struct psmouse \*psmouse); /\* ... \*/}; drivers/input/mouse/synaptics.c:/\* init() method of the Synaptics protocol \*/int synaptics\_init(struct psmouse \*psmouse){ struct synaptics\_data \*priv; psmouse->private = priv = kmalloc(sizeof(struct synaptics\_data), GFP\_KERNEL); /\* ... \*/ /\* This is called in interrupt context when mouse movement is sensed \*/ psmouse->protocol\_handler = synaptics\_process\_byte; /\* More protocol methods \*/ psmouse->set\_rate = synaptics\_set\_rate; psmouse->disconnect = synaptics\_disconnect; psmouse->reconnect = synaptics\_reconnect; /\* ... \*/} drivers/input/mouse/synaptics.c:/\* If you unfold synaptics\_process\_byte() and look at synaptics\_process\_packet(), you can see the input events being reported to user applications via mousedev \*/static void synaptics\_process\_packet(struct psmouse \*psmouse){ /\* ... \*/ if (hw.z > 0) { /\* Absolute X Coordinate \*/ input\_report\_abs(dev, ABS\_X, hw.x); /\* Absolute Y Coordinate \*/ input\_report\_abs(dev, ABS\_Y, YMAX\_NOMINAL + YMIN\_NOMINAL - hw.y); } /\* Absolute Z Coordinate \*/ input\_report\_abs(dev, ABS\_PRESSURE, hw.z); /\* ... \*/ /\* Left TouchPad button \*/ input\_report\_key(dev, BTN\_LEFT, hw.left); /\* Right TouchPad button \*/ input\_report\_key(dev, BTN\_RIGHT, hw.right); /\* ... \*/}

USB和蓝牙鼠标

和USB键盘一样，USB鼠标也由同样的输入驱动usbhid所驱动。类似的，支持蓝牙键盘的hidp也支持蓝牙鼠标。

正如你所预想的，USB和蓝牙鼠标的驱动通过mousedev来传输设备数据。

触摸控制器

在第6章中，以N\_TCH线路规程的形式实现了串行触摸控制器的设备驱动。输入子系统提供了更方便、更简单的方式以实现驱动。通过如下修改，重新以输入设备驱动的形式实现有限状态机：

1. serio提供serport线路规程，以访问和串口连接的设备。使用serport的服务和触摸控制器交互。

2. 类似在清单7.2中针对虚拟鼠标所做的，通过evdev产生输入报告，而不是传送坐标信息给tty层。

通过这些改变，通过/dev/input/eventX，用户空间可访问触摸屏。实际的驱动实现留做练习。

Analog Devices公司的ADS7846芯片是一个不使用串口的触摸控制器的例子，它使用的是SPI（Serial Peripheral Interface）接口。此设备的驱动使用的是SPI提供的服务，而不是serio。第8章“I2C协议”中的“串行外设接口总线”对SPI有详细讨论。类似于大多数触摸驱动，ADS7846驱动使用evdev接口分发触摸信息给用户应用程序。

一些触摸控制器接口为USB。其例子为3M USB触摸控制器，由drivers/input/touchscreen/usbtouchscreen.c所驱动。

很多PDA都在LCD上叠加有四线电阻触摸板。触摸板的X和Y电极（每个坐标轴两线）和模数转换器（ADC）相连，当触摸屏幕时会产生模拟电压值的变化，ADC将其变换为数字输出。输入驱动接收从ADC来的坐标并分发至用户空间。

由于制造过程的细微差别，同一款触摸板的多个产品可能会产生略有不同的坐标范围（X和Y方向的最大值）。为了使应用程序不受此差别之影响，使用之前会对触摸屏校准。校准通常由GUI来发起，通过在屏幕边界和其它的位置显示交叉记号，并要求用户点击这些点。如果支持自校准，产生的坐标通过使用相应的命令被编程进触摸控制器；否则用于对坐标数据流使用软件进行校正。

输入子系统也包括tsdev事件驱动，tsdev根据Compaq触摸屏协议产生坐标信息。如果你的系统通过tsdev报告触摸事件，支持此协议的应用程序能从/dev/input/tsX获取触摸输入。然而，此驱动将从主线内核中移去，因为用户空间的tslib库受到更多的支持.

Documentation/feature-removal-schedule.txt列出了那些将从内核源码树中移除的特性。

加速度传感器

加速度传感器用于测量加速度。某些IBM/联想的笔记本电脑配有检测突然移动的加速度传感器。产生的信息用于保护硬盘免于损坏。它所使用的机制称为硬盘活动保护系统（Hard Drive Active Protection System，HDAPS），类似于汽车中用于保护乘员免于伤害的安全气囊。HDAPS驱动被实现为平台驱动，向输入子系统注册。它使用evdev对检测的加速度的X和Y分量形成数据流。应用程序能够通过/dev/input/eventX读取加速度事件，以检测诸如振动之类的情况，并执行保护措施，例如停止硬盘驱动头。如果你移动笔记本电脑，执行下列命令将不不断输出信息（假设event3分配给HDAPS）：

bash> od –x /dev/input/event3

0000000 a94d 4599 1f19 0007 0003 0000 ffed ffff

...

加速度传感器也提供例如温度、键盘和鼠标的活动性等这些信息，所有这些信息可通过访问/sys/devices/platform/hdaps/中的文件获得。鉴于此，HDAPS驱动是内核源码中硬件监控（hwmon）子系统的一部分。在下一章的“带LM传感器的硬件监控”一节中我们将讨论硬件监控。

输出事件

一些设备驱动也处理输出事件。例如，键盘驱动能点亮CAPSLOCK LED，PC 扬声器驱动能够发出“嘟嘟”声。让我们重点来讨论后者。在初始化期间，扬声器驱动会通过设置相应的evbits以表明其有输出能力，并注册用来处理输出事件的回调函数：

Code View:

drivers/input/misc/pcspkr.c:

static int \_\_devinit pcspkr\_probe(struct platform\_device \*dev)

{

/\* ... \*/

/\* Capability Bits \*/

pcspkr\_dev->evbit[0] = BIT(EV\_SND);

pcspkr\_dev->sndbit[0] = BIT(SND\_BELL) | BIT(SND\_TONE);

/\* The Callback routine \*/

pcspkr\_dev->event = pcspkr\_event;

err = input\_register\_device(pcspkr\_dev);

/\* ... \*/

}

/\* The callback routine \*/

static int pcspkr\_event(struct input\_dev \*dev, unsigned int type,

unsigned int code, int value)

{

/\* ... \*/

/\* I/O programming to sound a beep \*/

outb\_p(inb\_p(0x61) | 3, 0x61);

/\* set command for counter 2, 2 byte write \*/

outb\_p(0xB6, 0x43);

/\* select desired HZ \*/

outb\_p(count & 0xff, 0x42);

outb((count >> 8) & 0xff, 0x42);

/\* ... \*/

}

为了使扬声器发声，键盘事件驱动产生一个声音事件（EV\_SND）：

input\_event(handle->dev, EV\_SND, /\* Type \*/

SND\_TONE, /\* Code \*/

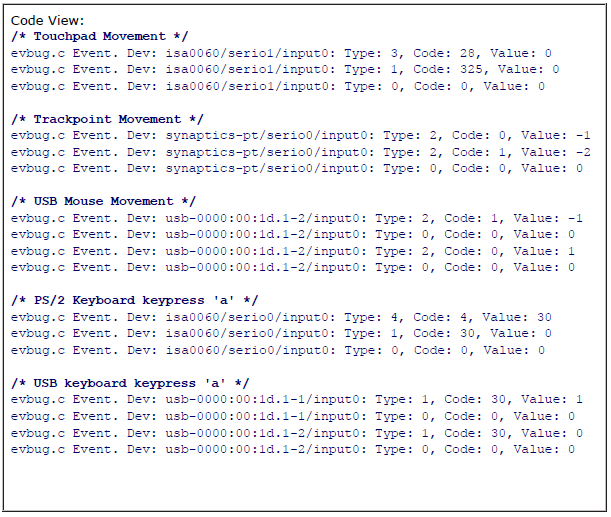
hz /\* Value \*/);

这会触发回调函数pcspkr\_event()的执行，你将听到“嘟嘟”声。

调试

如果你正在调试输入驱动，你可以使用evbug模块辅助调试。它输出和输入子系统产生的事件对应的（type, code, value）（查看前面定义的input\_event结构体）。图7.4是在操作某输入设备时evbug所捕获的数据。

图 7.4. Evbug输出



Code View:

/\* Touchpad Movement \*/

evbug.c Event. Dev: isa0060/serio1/input0: Type: 3, Code: 28, Value: 0

evbug.c Event. Dev: isa0060/serio1/input0: Type: 1, Code: 325, Value: 0

evbug.c Event. Dev: isa0060/serio1/input0: Type: 0, Code: 0, Value: 0

/\* Trackpoint Movement \*/

evbug.c Event. Dev: synaptics-pt/serio0/input0: Type: 2, Code: 0, Value: -1

evbug.c Event. Dev: synaptics-pt/serio0/input0: Type: 2, Code: 1, Value: -2

evbug.c Event. Dev: synaptics-pt/serio0/input0: Type: 0, Code: 0, Value: 0

/\* USB Mouse Movement \*/

evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 2, Code: 1, Value: -1

evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 0, Code: 0, Value: 0

evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 2, Code: 0, Value: 1

evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 0, Code: 0, Value: 0

/\* PS/2 Keyboard keypress 'a' \*/

evbug.c Event. Dev: isa0060/serio0/input0: Type: 4, Code: 4, Value: 30

evbug.c Event. Dev: isa0060/serio0/input0: Type: 1, Code: 30, Value: 0

evbug.c Event. Dev: isa0060/serio0/input0: Type: 0, Code: 0, Value: 0

/\* USB keyboard keypress 'a' \*/

evbug.c Event. Dev: usb-0000:00:1d.1-1/input0: Type: 1, Code: 30, Value: 1

evbug.c Event. Dev: usb-0000:00:1d.1-1/input0: Type: 0, Code: 0, Value: 0

evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 1, Code: 30, Value: 0

evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 0, Code: 0, Value: 0

为了更好的理解图7.4中的输出，需要牢记的是触摸板产生绝对坐标（EV\_ABS）或事件类型0x03，指点杆产生相对坐标（EV\_REL）或事件0x02，键盘发出键盘事件（EV\_KEY）或事件0x01。事件0x0对应于input\_sync()的调用，其操作如下：

input\_event(dev, EV\_SYN, SYN\_REPORT, 0);

此操作将转换为（type, code, value）组（0x0, 0x0, 0x0），并完成每个输入事件。

查看源代码

大多数输入事件驱动位于drivers/input/ directory目录。然而键盘事件驱动为drivers/char/keyboard.c，这是因为键盘和虚拟终端绑定，而非与/dev/input/下的设备节点绑定。

你可以在几个位置发现输入设备驱动。老式的键盘、鼠标和操纵杆驱动，位于drivers/input/下单独的子目录。蓝牙输入驱动在net/bluetooth/hidp/下。你也可以在drivers/hwmon/和drivers/media/video/等地方发现输入驱动。事件类型、代码和值定义于include/linux/input.h。

serio子系统位于drivers/input/serio/。serport线路规程的源码文件为drivers/input/serio/serport.c。Documentation/input/包括不同输入接口的更多细节。

表7.1概括了本章中使用的主要数据结构，及其在源码树中的位置。表7.2列出了本章所用到的主要内核编程接口，以及它定义的位置。

表 7.1. 数据结构概述

数据结构

位置

描述

input\_event

include/linux/input.h

evdev产生的每个事件包都采用此格式。

input\_dev

include/linux/input.h

代表一个输入设备。

input\_handler

include/linux/serial\_core.h

事件驱动支持的入口函数。

psmouse\_protocol

drivers/input/mouse/psmouse-base.c

所支持的PS/2鼠标协议驱动相关的信息。

psmouse

drivers/input/mouse/psmouse.h

PS/2鼠标驱动支持的方法。

表 7.2. 内核编程接口概述

内核接口

位置

描述

input\_register\_device()

drivers/input/input.c

向input核心注册一个设备。

input\_unregister\_device()

drivers/input/input.c

从input核心移除一个设备。

input\_report\_rel()

include/linux/input.h

在某个方向产生相对移动。

input\_report\_abs()

include/linux/input.h

在某个方向产生绝对移动。

input\_report\_key()

include/linux/input.h

产生一个按键或按钮按击。

input\_sync()

include/linux/input.h

表明输入子系统能收集以前产生的事件，将这些事件组成一个evdev包，并通过/dev/input/ inputX发送给用户空间。

input\_register\_handler()

drivers/input/input.c

注册一个用户事件驱动。

sysfs\_create\_group()

fs/sysfs/group.c

用特定属性创建sysfs节点组。

sysfs\_remove\_group()

fs/sysfs/group.c

移除用sysfs\_create\_group()创建的sysfs组。

tty\_insert\_flip\_char()

include/linux/tty\_flip.h

发送一个字符给线路规程层。

platform\_device\_register\_simple()

drivers/base/platform.c

创建一个简单平台设备。

platform\_device\_unregister()

drivers/base/platform.c

卸载一个平台设备。

--------------------------------------------------------------------------------

[MS1]原译文意思译反

[MS2]原文为命令执行结果，不能翻译

本文来自CSDN博客，转载请标明出处：http://blog.csdn.net/nhczp/archive/2009/10/08/4643974.aspx