# USB 手柄驱动报告

目录

[USB 手柄驱动报告 1](#_Toc406025474)

[各模块分工 1](#_Toc406025475)

[userapp.c 1](#_Toc406025476)

[描述: 1](#_Toc406025477)

[功能: 1](#_Toc406025478)

[pipe.h 2](#_Toc406025479)

[描述: 2](#_Toc406025480)

[xpad.c 2](#_Toc406025481)

[描述: 2](#_Toc406025482)

[makefile 4](#_Toc406025483)

[描述: 4](#_Toc406025484)

[浅谈Linux下USB驱动原理 5](#_Toc406025485)

[USB协议的获取 7](#_Toc406025486)

[设备驱动 12](#_Toc406025487)

# 各模块分工

## userapp.c

### **描述:**

用户空间的驱动程序 负责完成组合键 判断窗口类型等更搞层的功能

### 功能:

按键功能:

微软logo键:退出程序

x键:标左键

y键:标右键

手柄左杆:标移动

DPAD 上下:键盘上下

DPAD 左右:窗口切换

右大拇指键:回车

手柄右杆:鼠标滚轮

窗口识别:

目前识别窗口类型有:终端,sublime编辑器,火狐浏览器,chorme浏览器,文件管理器;

根据窗口类型改编按键策略,并实现如下功能:

如果是浏览器:

back键:后退

start键:前进

共同功能:

LT:关闭当前标签页

RT:新建标签页

LB:切换到前一个标签页

RB:切换到后一个标签页

Y键:新建窗口

B键:关闭窗口

## **pipe.h**

### 描述:

shell管道 ,得到活动窗口类名 , 被userapp.c调用

## xpad.c

### 描述:

内核空间的驱动程序,用来生成ko模块供linux内核加载.

首先移除linux自带的xpad模块.

命令:$sudo rmmod xpad

然后根据xpad.c生成ko文件

在xpad.c同一目录下创建makefile文件,在makefile文件中写如下代码:

**obj-m := xpad.o**

**all:**

**make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules**

**clean:**

**make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean**

然后执行命令:$make

在同一目录下会生成xpad.ko文件,这个文件是xpad的kernel module(内核驱动模块),

然后加载该内核模块:$sudo insmod xpad.ko

通过$cat /proc/bus/input/devices

查看当前usb设备是否成功识别

例如:

I: Bus=0003 Vendor=045e Product=028e Version=0114

N: Name="Microsoft XBox 360 Super Mouse"

P: Phys=usb-0000:00:1d.2-1/input0

S:Sysfs=/devices/pci0000:00/0000:00:1d.2/usb7/7-1/7-1:1.0/input/input22

U: Uniq=

H: Handlers=sysrq kbd mouse3 event13 js1

B: PROP=0

B: EV=f

B: KEY=c 0 0 0 0 0 0 0 7fd20000 430000 0 0 0 0 3792 2 417ff5ff fffffffc

B: REL=100

B: ABS=3f

输入设备节点文件都在/dev/input/文件夹下,前缀都是event

由以上信息的Handlers这一行可以知道,手柄的设备节点文件地址为

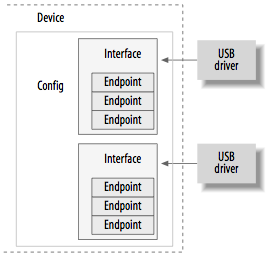
/dev/input/event13

## makefile

### 描述:

用来生成ko文件的make文件

# 浅谈Linux下USB驱动原理

Linux 支持两大类型的驱动,主机驱动 和 设备驱动. 本次软设,我们对主机驱动不做研究,主要讨论设备驱动. 

如图,一个USB驱动由一个或者多个config组成,其中每个config由一个或者多个接口,这些接口包括零个或者多个endpoint口.一个endpoint口总是单向传输的,要么是主机发往设备(OUT endpoint),要么是设备发往主机(IN endpoint).

有四种类型的endpoint并且每种类型的endpint传输数据的方式也不同.:

Control

Interrupt

Bulk

Isochronous

当主机向设备发送请求的时候,Interrupt(中断) endpoints 周期性传送固定大小的数据包.这种传送方式被键盘和鼠标设备广泛采用.

因为Xbox360手柄只涉及到Interrupt类型的传输,所以其他的类型不做讨论.

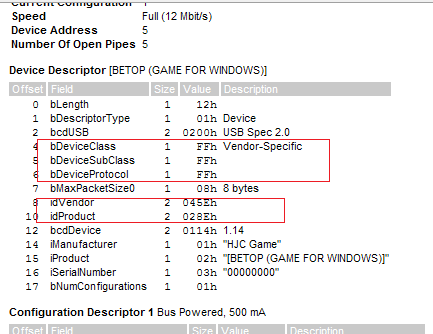
endpoint只是个口,需要URB来将数据从endpoint接收过来,或者将数据发送到endpoint上.

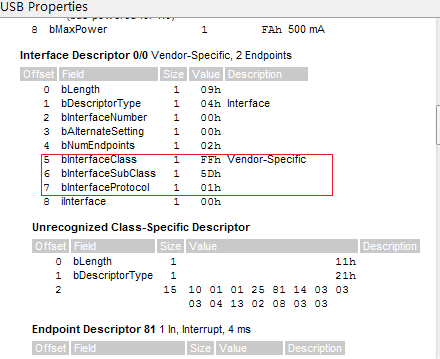
而 endpoint中的raw data是不规范化的数据,系统不能直接识别,需要对其解码然后转成系统能够识别的数据格式. 系统接收到能够识别的URB后根据事件的类型与取值做出相应的动作.具体请看xpad.c

# USB协议的获取

首先要对windows下的xbox360手柄驱动进行反向工程(或者叫做嗅探),获取到相应的USB协议.

有很多方法来进行USB嗅探:比如可以在VMware虚拟机下加载windows下的手柄设备驱动后,在主机环境下对虚拟机的数据进行截取,但是目前已经有了很多现成的USB传输分析工具,比如SnoopyPro,但是SnoopyPro仅支持xp的操作系统,最近新出了一款类似软件叫做USBlyzer,可以试用30天功能强大,而且支持win 7系统.





上图是USBlyzer返回的手柄信息,其中有用的信息有

idVendor: 0x045e (Microsoft)

idProduct: 0x028e (Standard version)

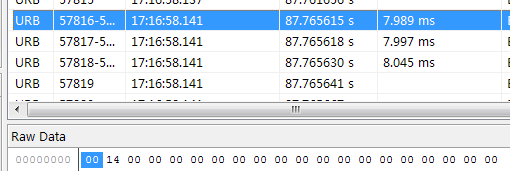
Class 255(0xFF), subclass 255(0xff), protocol 255(0xff)

其实xobx360手柄有三个interface(接口),但是只有第一个接口有用,其他的接口其实是为xbox360控制器同类型的其他设备预留的接口(比如xbox360电吉他,跳舞毯等等),其中第一个interface有in 和 out 两个endpoint.

第一个interface的协议为bInterfaceClass=0xffh bInterfaceSubClass=0x5D hbInterfaceProtocol=0x01h

这些在写驱动的时候作为参数提供给linux系统

然后对手柄各按键传输的raw data(原始数据)进行分析,



当手柄静止的时候raw data为(HEX) 00 14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

不难看出 raw data的长度为20byte. 而0x0014是固定不变的,代表数据包的长度0x0014=20

下面列出部分嗅探的结果:

游戏杆up

00000000 00 14 00 00 00 00 00 00 FF 7F 00 00 00 00 00 00 00 00 00 00 . . . . . . . . ÿ  . . . . . . . . . .

游戏杆down

00000000 00 14 00 00 00 00 00 00 00 80 00 00 00 00 00 00 00 00 00 00 . . . . . . . . .  . . . . . . . . . .

游戏杆left

00000000 00 14 00 00 00 00 00 80 0F FA 00 00 00 00 00 00 00 00 00 00 . . . . . . .  . ú . . . . . . . . . .

游戏杆right

00000000 00 14 00 00 00 00 FF 7F 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . . ÿ  . . . . . . . . . . . .

LT

00000000 00 14 00 00 FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . ÿ . . . . . . . . . . . . . . .

RT

00000000 00 14 00 00 00 FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . ÿ . . . . . . . . . . . . . .

LB

00000000 00 14 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . . . . . . . . . . . . . . . .

RB

00000000 00 14 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . . . . . . . . . . . . . . . .

back

00000000 00 14 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . . . . . . . . . . . . . . .

start

00000000 00 14 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . . . . . . . . . . . . . . . .

A

00000000 00 14 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . . . . . . . . . . . . . . . .

B

00000000 00 14 00 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . . . . . . . . . . . . . . .

X

00000000 00 14 00 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . @ . . . . . . . . . . . . . . . .

Y

00000000 00 14 00 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . .  . . . . . . . . . . . . . . . .

D-left

00000000 00 14 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . . . . . . . . . . . . . . . .

D-up

00000000 00 14 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . . . . . . . . . . . . . . . .

D-down

00000000 00 14 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . . . . . . . . . . . . . . . .

D-right

00000000 00 14 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . . . . . . . . . . . . . . . . . .

因为游戏杆是个杆....所以他的取值范围不是0~1而是 -32768~32767

例如游戏杆的y轴 最向上应该是32767

00 14 00 00 00 00 00 00 FF 7F 00 00 00 00 00 00 00 00 00 00

最向下应该是-32768

00 14 00 00 00 00 00 00 00 80 00 00 00 00 00 00 00 00 00 00

这里因为数据在raw data中是以小尾序存储的,要想看着舒服 我们要把它转化为大尾序,即把 FF 7F顺序颠倒一下,0x FF7F = -32768 ,0x8000=32767

在写驱动的时候这个地方也要把小尾序的数据转化成大尾序的数据传给系统.

下表是最终分析结果:

0x0014 0001 LB

0x0014 0002 RB

0x0014 0004 XBox button

0x0014 0008

0x0014 0010 A

0x0014 0020 B

0x0014 0040 X

0x0014 0080 Y

0x0014 0100 Dup

0x0014 0200 Ddown

0x0014 0400 Dleft

0x0014 0800 Dright

0x0014 1000 Start

0x0014 2000 Back

0x0014 4000 Left Axis Button

0x0014 8000 Right Axis Button

0x0014 0000 XX LR

0x0014 0000 00 XX RR

0x0014 0000 0000 LLHH Left X-Axis

0x0014 0000 0000 0000 LLHH Left Y-Axis

0x0014 0000 0000 0000 0000 LLHH Right X-Axis

0x0014 0000 0000 0000 0000 0000 LLHH Right Y-Axis

# 设备驱动

Linux一个非常良好的特性就是他可以动态加载对象代码来在内核下执行,这段代码被称为内核模块(kernel module),Linux能够分辨以下三个基本的设备类型:

字符类设备(Character devices)

块设备(Block devices)

网络接口(Network interfaces)

字符类设备在用户进城之间传送字节流.我们可以用open close read write ioctl等操作来管理模块.一个字符类设备看起来像一个文件,因为他根文件一样是可见的,而且大多数设备是线性的操控,即有序地进行write或者read数据的操作.xbox360手柄就属于字符类设备,这里块设备和网络接口不做讨论.

注册驱动有两种方式:第一种方式

init/exit型

整体框架如下:

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/usb.h>

static int pen\_probe(struct usb\_interface \*interface, const struct usb\_device\_id \*id)

{

    printk(KERN\_INFO "Pen drive (%04X:%04X) plugged\n", id->idVendor, id->idProduct);

    return 0;

}

static void pen\_disconnect(struct usb\_interface \*interface)

{

    printk(KERN\_INFO "Pen drive removed\n");

}

static struct usb\_device\_id pen\_table[] =

{

    { USB\_DEVICE(0x058F, 0x6387) },

    {} /\* Terminating entry \*/

};

MODULE\_DEVICE\_TABLE (usb, pen\_table);

static struct usb\_driver pen\_driver =

{

    .name = "pen\_driver",

    .id\_table = pen\_table,

    .probe = pen\_probe,

    .disconnect = pen\_disconnect,

};

static int \_\_init pen\_init(void)

{

    return usb\_register(&pen\_driver);

}

static void \_\_exit pen\_exit(void)

{

    usb\_deregister(&pen\_driver);

}

module\_init(pen\_init);

module\_exit(pen\_exit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Anil Kumar Pugalia <email\_at\_sarika-pugs\_dot\_com>");

MODULE\_DESCRIPTION("USB Pen Registration Driver");

第二种方法:

linux 3.4以后的内核版本中用module\_usb\_driver代替了module\_init （）和module\_exit(），作用一样

输入设备驱动的整体框架:

static struct usb\_device\_id xpad\_table[] =

{{

.match\_flags=

USB\_DEVICE\_ID\_MATCH\_VENDOR| USB\_DEVICE\_ID\_MATCH\_INT\_INFO,

.idVendor=(0x045e),

.bInterfaceClass=USB\_CLASS\_VENDOR\_SPEC,

.bInterfaceSubClass = 93,

.bInterfaceProtocol = (1)

},

{ }

};

static int dev\_probe (struct usb\_interface \*intf, const struct usb\_device\_id \*id);

static void dev\_disconnect (struct usb\_interface \*intf)

//标准usb驱动 表格 驱动信息 探针函数入口地址 断开函数入口地址

static struct usb\_driver dev\_driver =

{

.name = "usb dev",

.probe = dev\_probe,

.disconnect = dev\_disconnect,

.id\_table = dev\_table,

};

module\_usb\_driver(dev\_driver);

//

MODULE\_DEVICE\_TABLE(usb, dev\_table);

//驱动模块 作者 描述 许可

MODULE\_AUTHOR(“author”);

MODULE\_DESCRIPTION(“a hello world driver”);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR,MODULE\_DESCRIPTION,MODULE\_LICENSE,这些都是国际惯例,我就不细说了,需要说下MODULE\_DEVICE\_TABLE, MODULE\_DEVICE\_TABLE是为了当设备插入时,允许linux热插拔系统自动加载设备驱动,真正的即插即用,不插不用. .bInterfaceClass=USB\_CLASS\_VENDOR\_SPEC,

.bInterfaceSubClass = 93,

.bInterfaceProtocol = 1

就是xbox360手柄的第一个config中的第一个interface的信息: bInterfaceClass=0xffh bInterfaceSubClass=0x5D hbInterfaceProtocol=0x01h

idVendor=(0x045e) 就是xbox360手柄的idVendor

struct usb\_driver结构体中的

.name = "usb dev",

.probe = dev\_probe,

.disconnect = dev\_disconnect,

.id\_table = dev\_table,

id\_table 与MODULE\_DEVICE\_TABLE第二个参数一致 ,probe 是探针函数的入口,disconnect是设备断开后执行的操作

probe函数是在一个设备的信息与id\_table中的信息成功匹配的时候所执行的.

而disconnect函数在一个设备断开的时候执行的

probe函数代码如下:

static int xpad\_probe(struct usb\_interface \*intf, const struct usb\_device\_id \*id) //探针函数 相当于设备驱动的主函数

{

**//当设备与id\_table中的信息匹配成功后 系统调用此函数 intf是系统提供的usb接口**

**//id 是系统提供的usb设备id**

**// interface\_to\_usbdev(intf) 可以返回intf接口对应的usb设备**

**//他是一个宏 返回的是intf->dev.parent**

struct usb\_device \*udev = interface\_to\_usbdev(intf);

**//usb\_xpad是我自己定义的结构体 里面保存着跟usb设备有关的变量**

**//里面包含了 输入设备接口 (input\_dev) usb设备(usb\_device) usb接口(usb\_interface)**

**// in endpoint口的URB ; out endpoint口的URB ;等**

**//URB是用来从endpint 接收或者发送数据的一种异步方式的实现.**

struct usb\_xpad \*xpad;

struct input\_dev \*input\_dev;**//输入设备**

struct usb\_endpoint\_descriptor \*ep\_irq\_in;  **//in endpoint 的描述**

int ep\_irq\_in\_idx;

int i, error;

**//根据实际插入的设备的生产商id和产品id 找到匹配的设备**

for (i = 0; xpad\_device[i].idVendor; i++)

{

**// le16\_to\_cpu是将小尾序的数据转化成cpu能识别的尾序(因cpu型号的不同而不同)的函数**

if ((le16\_to\_cpu(udev->descriptor.idVendor) == xpad\_device[i].idVendor) &&

(le16\_to\_cpu(udev->descriptor.idProduct) == xpad\_device[i].idProduct))

break;

}

**//在内核空间为xpad指针申请内存空间**

xpad = kzalloc(sizeof(struct usb\_xpad), GFP\_KERNEL);

**//为输入设备在内核空间申请内存空间 input\_allocate\_device()这个函数是用kzalloc函数申请了空间后 然后对该内存进行了初始化**

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

**//如果申请空间失败**

if (!xpad || !input\_dev) {

error = -ENOMEM;

goto fail1;

}

**//** **usb\_alloc\_coherent**

**//分配DMA接口的缓冲区 返回 idata\_dma 和 idata（raw packet，原始数据包）**

**//DMA接口会给URB来使用**

xpad->idata = usb\_alloc\_coherent(udev, XPAD\_PKT\_LEN,

GFP\_KERNEL, &xpad->idata\_dma);

**//如果DMA借口申请失败**

if (!xpad->idata)

{

error = -ENOMEM;

goto fail1;

}

**//usb\_alloc\_urb函数 创建新的urb给设备使用**

**//如果使用的是中断类型的endpoint 函数的第一个参数为0**

xpad->irq\_in = usb\_alloc\_urb(0, GFP\_KERNEL);

**//如果urb创建失败**

if (!xpad->irq\_in) {

error = -ENOMEM;

goto fail2;

}

**// 将usb设备信息存入xpad**

xpad->udev = udev;

**// 将usb借口信息存入xpad**

xpad->intf = intf;

**//将输入设备信息的首地址存入xpad**

xpad->dev = input\_dev;

**//创建物理路径**

usb\_make\_path(udev, xpad->phys, sizeof(xpad->phys));

strlcat(xpad->phys, "/input0", sizeof(xpad->phys));

**//完善输入设备信息**

input\_dev->name = xpad\_device[i].name; **//保存输入设备名字**

input\_dev->phys = xpad->phys; **//保存输入设备的物理路径**

**//usb\_to\_input\_id函数 根据usb设备返回输入设备的id**

usb\_to\_input\_id(udev, &input\_dev->id);

**//一个设备往往有很多接口 而这次软设只用到了一个接口,而接口只对应一个input\_dev(输入设备)**

input\_dev->dev.parent = &intf->dev;

**//把xpad以私有数据存放于input\_dev中，以便后面方便调用**

input\_set\_drvdata(input\_dev, xpad);

**//输入设备的打开函数**

**// xpad\_open函数 需要在此函数中调用usb\_submit\_urb**

**//接收和发送数据都要调用usb\_submit\_urb 他负责将URB提交到USB core**

input\_dev->open = xpad\_open;

**//输入设备的关闭函数**

**// xpad\_close函数 需要在此函数中调用usb\_kill\_urb来停止URB的运作**

input\_dev->close = xpad\_close;

//**BIT\_MASK是个宏**

**//****BIT\_MASK(nr) (1UL<<((nr)%BITS\_PER\_LONG）)**

input\_dev->evbit[0] = BIT\_MASK(EV\_KEY);  **//注册键盘事件**

input\_dev->evbit[0] |= BIT\_MASK(EV\_REL); **//注册相对轴事件**

input\_dev->evbit[0] |= BIT\_MASK(EV\_ABS); **//注册绝对轴事件**

**/\* set up axes \*/**

for (i = 0; xpad\_abs[i] >= 0; i++) **///注册对应的绝对轴**

xpad\_set\_up\_abs(input\_dev, xpad\_abs[i]);

xpad\_set\_up\_rel(input\_dev, REL\_WHEEL); **//注册对应的相对轴**

for (i = 0 ; key\_need\_register[i] >= 0; i++) \_\_set\_bit(key\_need\_register[i], input\_dev->keybit);

**//注册按键**

for (i = 0; xpad\_common\_btn[i] >= 0; i++)

\_\_set\_bit(xpad\_common\_btn[i], input\_dev->keybit);

**//注册游戏手柄按键**

for (i = 0; xpad360\_btn[i] >= 0; i++)

\_\_set\_bit(xpad360\_btn[i], input\_dev->keybit);

**//注册LT RT按键**

for (i = 0; xpad\_abs\_triggers[i] >= 0; i++)

xpad\_set\_up\_abs(input\_dev, xpad\_abs\_triggers[i]);

**//初始化设备输出**

error = xpad\_init\_output(intf, xpad);

if (error)

goto fail3;

ep\_irq\_in\_idx = 0;

ep\_irq\_in = &intf->cur\_altsetting->endpoint[ep\_irq\_in\_idx].desc;

**//根据usb接口得到 in endpoint口的描述符**

**//函数结构 usb\_fill\_int\_urb(struct urb\* urb,struct usb\_device \* dev,unsigned int pipe,void \* transfer\_buffer,int buffer\_length,usb\_complete\_t complete,void \* context,int interval)**

**//根据 usb设备，usb管道，输入缓冲区的首地址，缓冲区长度，urb入口函数，手柄设备数据信息，in endpoint口的轮换间隔信息 得到 输入的urb**

usb\_fill\_int\_urb(xpad->irq\_in, udev,

usb\_rcvintpipe(udev, ep\_irq\_in->bEndpointAddress),

xpad->idata, XPAD\_PKT\_LEN, xpad\_irq\_in,

xpad, ep\_irq\_in->bInterval);

**//传输的DMA接口地址绑定到in endpoint的URB中**

xpad->irq\_in->transfer\_dma = xpad->idata\_dma;

**//传输标志(允许DMA方式传输) urb->transfer\_dma valid on submit**

xpad->irq\_in->transfer\_flags |= URB\_NO\_TRANSFER\_DMA\_MAP;

**//根据输入设备信息 注册设备**

error = input\_register\_device(xpad->dev);

if (error)

goto fail5;

**//把xpad以私有数据存放于usb\_interface中，以便后面方便调用**

usb\_set\_intfdata(intf, xpad);

**//成功 返回**

return 0;

**//如果执行错误执行 善后工作**

fail5: if (input\_dev)

input\_ff\_destroy(input\_dev); //free force feedback structures

fail4: xpad\_deinit\_output(xpad); //free out urb and out dma

fail3: usb\_free\_urb(xpad->irq\_in); //free in urb

fail2: usb\_free\_coherent(udev, XPAD\_PKT\_LEN, xpad->idata, xpad->idata\_dma); //free in dma

fail1: input\_free\_device(input\_dev); //free input\_dev

kfree(xpad); //free xpad

return error;

}

disconnect函数比较简单作用是释放之前申请的空间.

static void xpad\_disconnect(struct usb\_interface \*intf) **//设备断开函数**

{

struct usb\_xpad \*xpad = usb\_get\_intfdata (intf); **//根据usb接口得到手柄设备信息**

input\_unregister\_device(xpad->dev);  **//注销 该设备**

**//释放内核空间**

xpad\_deinit\_output(xpad);

usb\_free\_urb(xpad->irq\_in);

usb\_free\_coherent(xpad->udev, XPAD\_PKT\_LEN,

xpad->idata, xpad->idata\_dma);

kfree(xpad->bdata);

kfree(xpad);

usb\_set\_intfdata(intf, NULL);

}