
网名“鱼树”的学员聂龙浩,

学习“韦东山 Linux 视频第 2 期”时所写的笔记很详细,供大家参考。

也许有错漏,请自行分辨。

目录

一、 USB 概念:	2
现象:	2
问 1. 既然还没有"驱动程序",为何能知道是"android phone"。	3
问 2. USB 设备种类非常多,为什么一接入电脑,就能识别出来?	3
问 3. PC 机上接有非常多的 USB 设备,怎么分辨它们?	3
问 4. USB 设备刚接入 PC 时,还没有编号;那么 PC 怎么把"分配的编号"告诉它?	4
问 5. 为什么一接入 USB 设备,PC 机就能发现它?	4
其他概念:	5
1. USB 是主从结构的:	5
2. USB 的传输类型:	5
3. USB 传输的对象: 端点(endpoint)	6
4. 每一个端点都有传输类型,传输方向。	6
5. 术语里、程序里说的输入(IN)、输出(OUT) "都是" 基于 USB 主机的立场说的。	6
6. USB 总线驱动程序的作用:	6
USB 驱动程序框架:	6
二、 USB 总线驱动概念:	7
一、USB 总线驱动程序的作用:	7
1. 识别 USB 设备。	7
2. 查找并安装对应的设备驱动程序	8
3. 提供 USB 读写函数	8
二、USB 主机控制器:	8
1, 有三种规范-- UHCI OHCI EHCI	8
2, 搜索内核源码:	11
三、USB 总线驱动程序的流程:	11
1. 识别 USB 设备。	11
2. 查找并安装对应的设备驱动程序	11
3. 提供 USB 读写函数	12
HUB 的概念:	13
设备描述符:	16
端点描述符:	18
从"hub_port_connect_change()"开始分析如下的过程:	19

总结:	22
三、 USB 驱动设备简单编写	22
目标:	23
在“.probe”函数里做下面 4 件事情:	23
怎么写 USB 设备驱动程序?	23
1. 分配/设置 usb_driver 结构体	23
2. 注册	24
代码框架:	25
一, id_table:	25
二, probe 函数:	26
三, disconnect 函数:	27
四, 测试:	28
六, 在 probe 函数中打印 厂家 ID 和 设备 ID:	29
1, 通过“usb_interface”USB 接口得到“usb_device”(USB 设备)结构体:	30
四、 USB 鼠标驱动	31
一, 分配一个 input_dev 结构。	32
二, 设置这个 input_dev 结构	32
三, 注册	33
四, 硬件相关操作	33
A. 数据传输 3 要素: 源, 目的, 长度。	33
B, 使用三要素:	34
测试 3th:	37
测试 4th:	40
总结:	41

一、 USB 概念:

现象:

在 WINDOWS 下:

把 USB 设备接到 PC

1. 右下角弹出“发现什么 USB 新设备”。如发现“android phone”。
2. 跳出一个对话框, 提示你安装驱动程序。

问 1. 既然还没有"驱动程序", 为何能知道是"android phone"。

答 1. windows 里已经有了 USB 的总线驱动程序, 接入 USB 设备后, 是"总线驱动程序"知道你是"android phone"

提示你安装的是"设备驱动程序"。

一个层次出来了: USB 分为两层。

APP:

设备驱动 -- 自己写

。 。 。 。 。

USB 总线驱动程序 -- 一般为内核自带

硬 件

USB 总线驱动程序负责: 识别 USB 设备, 给 USB 设备找到对应的驱动程序。

问 2. USB 设备种类非常多, 为什么一接入电脑, 就能识别出来?



答 2. PC 和 USB 设备都得遵守一些规范。

比如: USB 设备接入电脑后, 总线驱动程序就会发什么命令出来, 如 PC 机会发出"你是什么"?

USB 设备就必须回答"我是 xxx", 并且回答的语言必须是中文。(回答的格式一样)

USB 总线驱动程序会发出某些命令想获取设备信息(描述符),

USB 设备必须返回"描述符"给 PC。(回复的信息都有一个定的格式, 有格式化的信息称为"描述符")。

问 3. PC 机上接有非常多的 USB 设备, 怎么分辨它们?

USB 接口只有 4 条线: 5V, GND, D-, D+。

所有的数据都是通过"D+, D-"来传输, 所有设备都是挂接到 D+, D1 上。则那么多设备是如何选择一个 USB 设备进行通讯的。

答 3. 每一个 USB 设备接入 PC 时，USB 总线驱动程序都会给它分配一个编号。（在 USB 总线上每个设备都有编号）

接在 USB 总线上的每一个 USB 设备都有自己的编号(在驱动程序里称为'地址')。

PC 机想访问某个 USB 设备时，发出的命令都含有对应的编号信息(地址)。

问 4. USB 设备刚接入 PC 时，还没有编号；那么 PC 怎么把"分配的编号"告诉它？

答 4. 新接入的 USB 设备的默认编号是 0，在未分配新编号前，PC 使用 0 编号和它通信。

接着就可以通过 0 号编号分配它

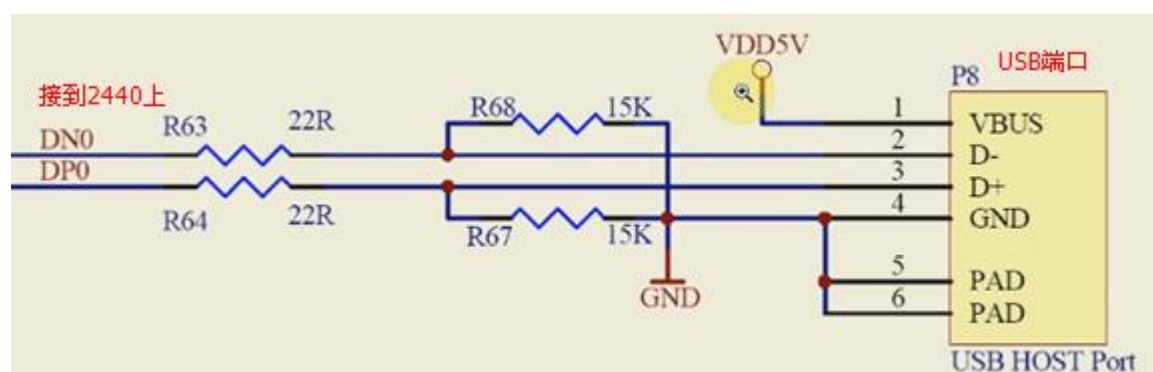
新的编号（通信）。

问 5. 为什么一接入 USB 设备，PC 机就能发现它？

只有 4 条线（没什么中断引脚），如何快速发现的。这是 USB 总线里做了某些手脚。

答 5. PC 的 USB 口内部，D-和 D+接有 15K 的下拉电阻，平时未接 USB 设备时为低电平

USB 设备的 USB 口内部，D-或 D+接有 1.5K 的上拉电阻；它一接入 PC，就会把 PC USB 口的 D-或 D+拉高（有 5V 的一边才 1.5K 电阻，另一边是 15K 电阻接地，这样电流能流过来。所以原本被 15K 拉低的一端，也有电流过来了。），从硬件的角度通知 PC 有新设备接入。



这是一个 USB 主机，会发现 'D+, D-' 里面都接有一个 '15K' 的下拉电阻（保证没信号的时候是什么电位）。

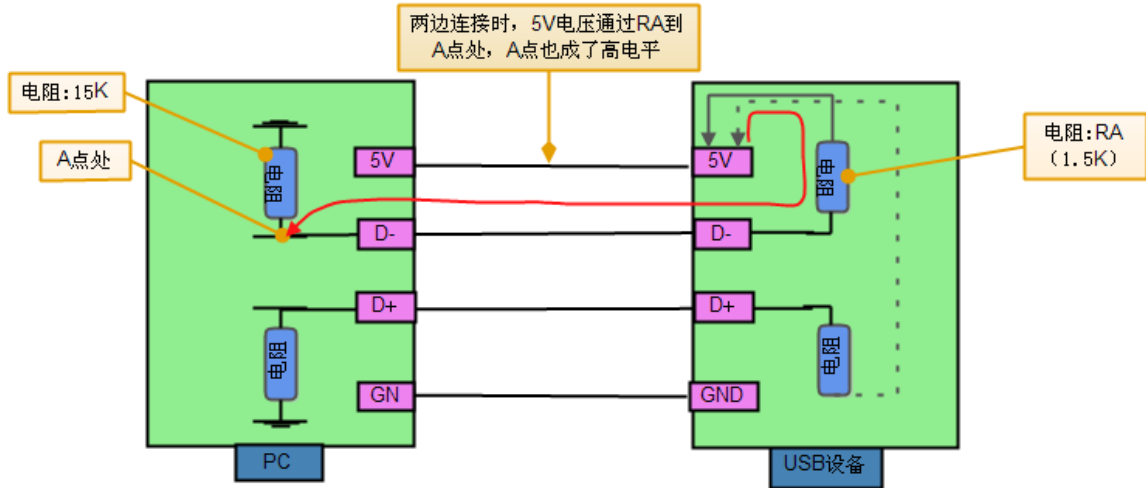
上拉，就是把电位拉高，比如拉到 VCC

下拉，就是把电压拉低，拉到 GND

一般就是刚上电的时候，端口电压不稳定，为了让他稳定为高或低，就会用到上拉或下拉电阻。

有些芯片内部集成了上拉电阻，所以外部就不用上拉电阻了。但是有一些开漏的，外部必须加上拉电阻。

这是 PC 里面的情况，若没有接外设时，这 15K 电阻处的引脚是低电平。



USB 设备中，D+ 或 D- 那里要接一个“上拉电阻”接到 5V。

一开始“PC--USB 设备”间没有连接时，PC 端“D-, D+”两个引脚都是低电平，两边连接时，USB 设备端的 5V 通过上拉电阻 RA 一直过来到 PC 端，这时 PC 端 D- 引脚的 A 点就变成高电平了。这时 PC 端的硬件就能发现新设备。

在“D-”上接 1.5K 上拉电阻就是“全速设备”（12M 每秒）。

在“D+”上接 1.5K 上拉电阻就是“高速设备”（480M 每秒）。

其他概念:

1. USB 是主从结构的:

所有的 USB 传输，都是从 USB 主机这方发起；USB 设备（从机）没有“主动”通知 USB 主机的能力。

例子：USB 鼠标滑动一下立刻产生数据，但是它没有能力通知 PC 机来读数据，只能被动地等得 PC 机来读。PC 主机会不断的来查询数据。

2. USB 的传输类型:

- 控制传输：可靠，时间有保证，比如：USB 设备的识别过程
- 批量传输：可靠，时间没有保证，比如：U 盘（从 PC 上删除 U 盘时有时很快有时很慢）
- 中断传输：可靠，实时，比如：USB 鼠标。（借助了中断的概念，实质是用 USB 主机查询方式来实现实时性，

不断读 USB 鼠标。USB 设备并不能发起中断，USB 设备并没有发起通知给 USB 主机的能力。)

d. 实时传输：不可靠，实时，比如：USB 摄像头。（数据的不可靠可以接受，要是可靠，可能因为某一帧发送错误时会因可靠机制来重传，则产生滞后感。)

3. USB 传输的对象：端点(endpoint)

我们说“读 U 盘”、“写 U 盘”，可以细化为：把数据写到 U 盘的端点 1，从 U 盘的端点 2 里读出数据

除了端点 0 外，每一个端点只支持一个方向的数据传输。

端点 0 用于控制传输，既能输出也能输入。

4. 每一个端点都有传输类型，传输方向。

5. 术语里、程序里说的输入(IN)、输出(OUT) "都是" 基于 USB 主机的立场说的。

比如鼠标的数据是从鼠标传到 PC 机，对应的端点称为“输入端点”。（基于 USB 主机立场这是输入）。

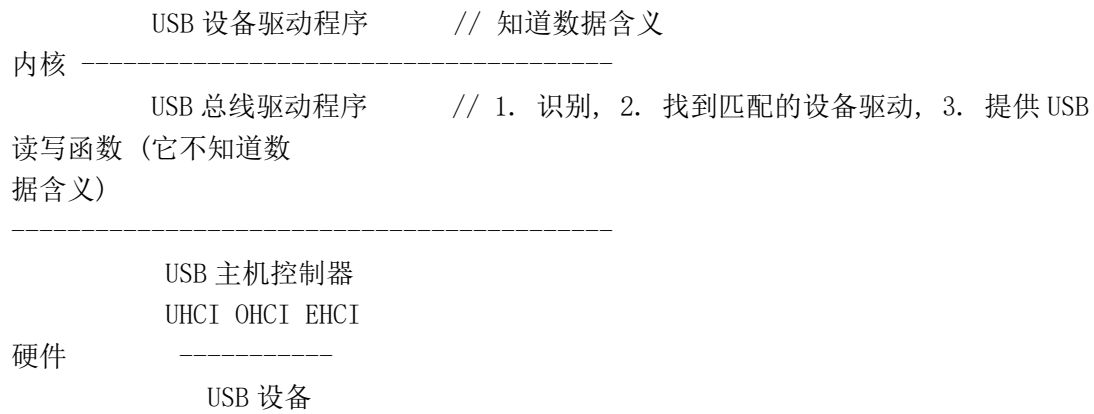
6. USB 总线驱动程序的作用：

- a. 识别 USB 设备。
- b. 查找并安装对应的设备驱动程序。
- c. 提供 USB 读写函数。（只提供读写函数，数据的含义不知道。只是“送信人”。不了解数据含意。USB 设备驱动程序知道数据含意。）

USB 总线驱动程序已经有‘读写’函数，可以让应用程序访问这些函数（跨过“设备驱动程序”）。Libusb 就是封装了“USB 总线驱动程序”提供出的“读写”函数。直接调用“USB 总线驱动程序”中的函数很复杂，libusb 就封装了这些函数。这样就绕过了“设备驱动”直接访问。但这不是这节的内容，这里是要如何写“USB 设备驱动”。

USB 驱动程序框架：

app:



二、 USB 总线驱动概念:



UHCI: intel, 低速(1.5Mbps)/全速(12Mbps)
 OHCI: microsoft 低速/全速
 EHCI: 高速(480Mbps)

USB 总线驱动程序就是用来支持 “硬件 — USB 主机控制器” 的。可以认为 “USB 总线驱动程序” 就是 “USB 主机控制器” 的驱动程序。

USB 设备驱动程序就是用来支持 “硬件 — USB 设备” 的。但是为了访问 “USB 设备”，就要用到了 “USB 总线驱动程序” 提供的函数，将命令或数据发给 “USB 主机控制器”，由 “USB 主机控制器” 产生信号发送给 “USB 设备”。

一、 USB 总线驱动程序的作用:

1. 识别 USB 设备。

- 1.1 分配地址。
- 1.2 并告诉 USB 设备(set address)。
- 1.3 发出命令获取描述符。

描述符的信息可以在 include/linux/usb/Ch9.h 看到。

2. 查找并安装对应的设备驱动程序

3. 提供 USB 读写函数

二、USB 主机控制器：

1, 有三种规范-- UHCI OHCI EHCI

UHCI : intel 定义出来的如何使用这个“主机控制器”。

适用于 低速 (USB 1.1) 或全速 (USB2.0) 的 USB 设备。当说到 USB2.0 时有“全速”和“高速”。

intel 是生产 CPU 的, 所以用 UHCI 规范做出来的 USB 主机控制器, 硬件功能强大, 软件实现稍微简单点。

OHCI : 是微软定义。这套规范做出来的主机控制器, 硬件稍弱, 但软件复杂。

EHCI : 支持高速 (480Mbps)。EHCI 专用于高速设备。

在系统中:

```
[ 8537.416114] usb 1-2: new full-speed USB device number 3 using ohci_hcd
[ 8537.989067] Initializing USB Mass Storage driver...
[ 8537.989144] scsi5 : usb-storage 1-2:1.0
[ 8537.989224] usbcore: registered new interface driver usb-storage
[ 8537.989226] USB Mass Storage support registered.
[ 8538.011072] usbcore: registered new interface driver uas
[ 8539.103917] scsi 5:0:0:0: Direct-Access Kingston DT 101 G2 1.00 PQ
: 0 ANSI: 4
[ 8539.109110] sd 5:0:0:0: Attached scsi generic sg4 type 0
[ 8539.123078] sd 5:0:0:0: [sdd] 15131636 512-byte logical blocks: (7.74 GB/7.21
GiB)
[ 8539.132891] sd 5:0:0:0: [sdd] Write Protect is off
[ 8539.132897] sd 5:0:0:0: [sdd] Mode Sense: 45 00 00 00
[ 8539.141067] sd 5:0:0:0: [sdd] Write cache: disabled, read cache: enabled, doe
sn't support DPO or FUA
[ 8539.215593] sdd: sdd1
[ 8539.269517] sd 5:0:0:0: [sdd] Attached SCSI removable disk
nie@nie:~$
```

下面是在路由器上:


```
# cat /proc/bus/usb/devices

T: Bus=02 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#= 1 Spd=12 MxCh= 1
B: Alloc= 0/900 us ( 0%), #Int= 0, #Iso= 0
D: Ver= 1.10 Cls=09(hub ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1
P: Vendor=0000 ProdID=0000 Rev= 2.06
S: Manufacturer=Linux 2.6.21 ohci_hcd
S: Product=RT3xxx OHCI Controller
S: SerialNumber=rt3xxx-ohci
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr= 0mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=hub
E: Ad=81(I) Atr=03(Int.) MxPS= 2 Iv1=255ms

T: Bus=01 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#= 1 Spd=480 MxCh= 1
B: Alloc= 0/800 us ( 0%), #Int= 0, #Iso= 0
D: Ver= 2.00 Cls=09(hub ) Sub=00 Prot=01 MxPS=64 #Cfgs= 1
P: Vendor=0000 ProdID=0000 Rev= 2.06
S: Manufacturer=Linux 2.6.21 ehci_hcd
S: Product=Ralink EHCI Host Controller
S: SerialNumber=rt3xxx
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr= 0mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=hub
E: Ad=81(I) Atr=03(Int.) MxPS= 4 Iv1=256ms
#
```



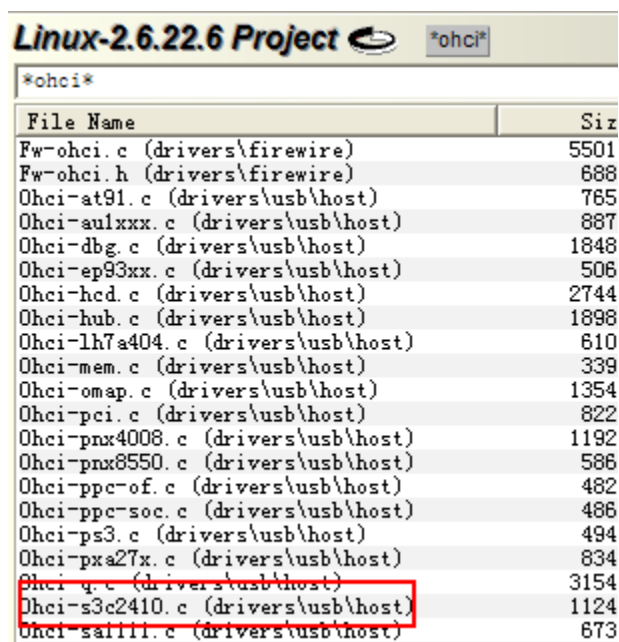
```
# cat /proc/bus/usb/devices

T: Bus=02 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#= 1 Spd=12 MxCh= 1
B: Alloc= 0/900 us ( 0%), #Int= 0, #Iso= 0
D: Ver= 1.10 Cls=09(hub ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1
P: Vendor=0000 ProdID=0000 Rev= 2.06
S: Manufacturer=Linux 2.6.21 ohci_hcd
S: Product=RT3xxx OHCI Controller
S: SerialNumber=rt3xxx-ohci
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr= 0mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=hub
E: Ad=81(I) Atr=03(Int.) MxPS= 2 Iv1=255ms

T: Bus=01 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#= 1 Spd=480 MxCh= 1
B: Alloc= 0/800 us ( 0%), #Int= 0, #Iso= 0
D: Ver= 2.00 Cls=09(hub ) Sub=00 Prot=01 MxPS=64 #Cfgs= 1
P: Vendor=0000 ProdID=0000 Rev= 2.06
S: Manufacturer=Linux 2.6.21 ehci_hcd
S: Product=Ralink EHCI Host Controller
S: SerialNumber=rt3xxx
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr= 0mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=hub
E: Ad=81(I) Atr=03(Int.) MxPS= 4 Iv1=256ms

T: Bus=01 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#= 2 Spd=480 MxCh= 0
D: Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1
P: Vendor=0951 ProdID=1642 Rev= 1.00
S: Manufacturer=Kingston
S: Product=DT 101 G2
S: SerialNumber=001CC0EC33B0FC31170C2BA4
C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=100mA
I:* If#= 0 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50 Driver=usb-storage
E: Ad=81(I) Atr=02(Bulk) MxPS= 512 Iv1=0ms
E: Ad=02(O) Atr=02(Bulk) MxPS= 512 Iv1=31875us
*
```

2, 搜索内核源码:



File Name	Size
Fw-ohci.c (drivers/firewire)	5501
Fw-ohci.h (drivers/firewire)	688
Ohci-at91.c (drivers/usb/host)	765
Ohci-aulxxx.c (drivers/usb/host)	887
Ohci-dbg.c (drivers/usb/host)	1848
Ohci-ep93xx.c (drivers/usb/host)	506
Ohci-hcd.c (drivers/usb/host)	2744
Ohci-hub.c (drivers/usb/host)	1898
Ohci-lh7a404.c (drivers/usb/host)	610
Ohci-mem.c (drivers/usb/host)	339
Ohci-omap.c (drivers/usb/host)	1354
Ohci-pci.c (drivers/usb/host)	822
Ohci-pnx4008.c (drivers/usb/host)	1192
Ohci-pnx8550.c (drivers/usb/host)	586
Ohci-ppc-of.c (drivers/usb/host)	482
Ohci-ppc-soc.c (drivers/usb/host)	486
Ohci-ps3.c (drivers/usb/host)	494
Ohci-pxa27x.c (drivers/usb/host)	834
Ohci-q.c (drivers/usb/host)	3154
Ohci-s3c2410.c (drivers/usb/host)	1124
Ohci-salil.c (drivers/usb/host)	673

从上面看 2410 是使用的“微软规范”的 USB 主机控制器。

```
Ohci-q.c (linux-2.6.21.x/drivers/usb/host)
Ohci-rt3xxx.c (linux-2.6.21.x/drivers/usb/host)
Ohci-s3c2410.c (linux-2.6.21.x/drivers/usb/host)
```

可以看到“RT3xxx”就是我们的 3G 路由器的 USB 主机控制器是用的“OHCI”规范。

三、USB 总线驱动程序流程:

想大概看 USB 总线驱动程序的工作过程: (所做的事件)

USB 总线驱动程序的作用: (从代码里分析)

1. 识别 USB 设备。

1.1 分配地址。(刚上来时就用默认地址 0。)

1.2 并告诉 USB 设备(set address)。

1.3 发出命令获取描述符。

描述符的信息可以在 include/linux/usb/Ch9.h 看到。

2. 查找并安装对应的设备驱动程序

3. 提供 USB 读写函数

把 USB 设备接到开发板上，看输出信息：

```
usb 1-1: new full speed USB device using s3c2410-ohci and address 2
usb 1-1: configuration #1 chosen from 1 choice
scsi0 : SCSI emulation for USB Mass Storage devices (刚接上 USB 设备时弹出信息)
scsi 0:0:0:0: Direct-Access    HTC      Android Phone    0100 PQ: 0 ANSI: 2
sd 0:0:0:0: [sda] Attached SCSI removable disk (安装驱动程序)
```

接着拔掉：

```
usb 1-1: USB disconnect, address 2
```

再接上：

```
usb 1-1: new full speed USB device using s3c2410-ohci and address 3
usb 1-1: configuration #1 chosen from 1 choice
scsi1 : SCSI emulation for USB Mass Storage devices (刚接上 USB 设备时弹出信息)
scsi 1:0:0:0: Direct-Access    HTC      Android Phone    0100 PQ: 0 ANSI: 2
sd 1:0:0:0: [sda] Attached SCSI removable disk (这是在安装驱动程序)
```

分析这些信息：

```
usb 1-1: new full speed USB device using s3c2410-ohci and address 2
```

新的 全速 USB 设备 使用 “s3c2410-ohci” 和地址 2。可以到拔下后再插入时分配的是地址 3。

从源码中查到这些信息。搜索 Full speed 会不可靠，因为高速和低速，这里 speed 可能是 “%s”。

要搜索通用的部分。

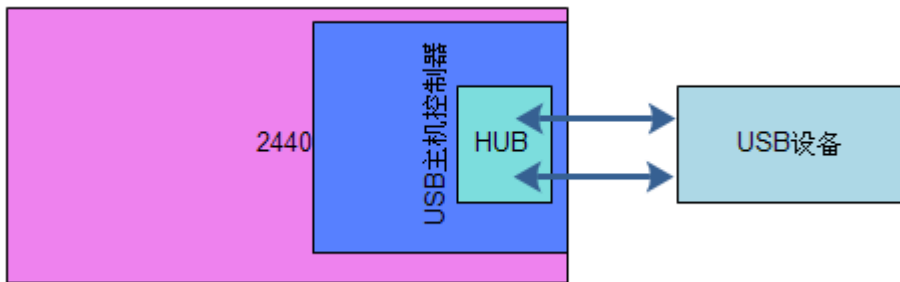
```
usb 1-1: configuration #1 chosen from 1 choice
```

```
scsi0 : SCSI emulation for USB Mass Storage devices
```

在内核驱动目录下搜：

```
grep "USB device using" * -nR
drivers/usb/core/hub.c:2186: "%s %s speed %sUSB device using %s and
address %d\n",
```

HUB 的概念:



每一个 USB 主机控制器，都自带了一个 USB HUB，HUB 上再接“USB 设备”。可以认为“HUB”是一个特殊的“USB 设备”。

可以发现打印是在如下:

```
int hub_port_init (struct usb_hub *hub, struct usb_device *udev, int port1,
int retry_counter) //端口初始化
{
    dev_info (&udev->dev,
              "%s %s speed %sUSB device using %s and address %d\n",
              (udev->config) ? "reset" : "new", speed, type,
              udev->bus->controller->driver->name, udev->devnum);
}
```

查这个“hub_port_init ()”在哪里被调用: hub_port_connect_change() 在 HUB 端口连接发生变化这个函数。

```
#ifdef CONFIG_USB_SUSPEND
```

电源管理方面的宏。

接上一个“USB 设备”后，由 USB 设备中的上拉电阻使得 USB 主机控制器一端的“D-”或“D+”有了电平，硬件方面就感知到了有 USB 设备 接入。USB 主机控制器中就会产生某个中断。

过程如下:

```
hub_irq()
```

```
-->kick_khubd() //其中有唤醒“休眠队列-khubd_wait”。
```

```
-->hub_thread() //HUB 线程，没事件时会休眠。
```

```
-->hub_events() //HUB 事件
```

```
-->hub_port_connect_change()
```

```
-->udev = usb_alloc_dev(hdev, hdev->bus, port1);
```

```
dev->dev.bus = &usb_bus_type;
```

```
choose_address(udev); // 给新设备分配编号(地址 1~127), 还没告诉设备。
```

```
hub_port_init // usb 1-1: new full speed USB device using s3c2410-ohci and
address 3
```

```
hub_set_address // 把编号(地址)告诉 USB 设备
```

```
usb_get_device_descriptor(udev, 8); // 获取设备描述符
retval = usb_get_device_descriptor(udev, USB_DT_DEVICE_SIZE);
```

```
usb_new_device(udev)
err = usb_get_configuration(udev); // 把所有的描述符都读出来，并解析
usb_parse_configuration
```

```
device_add // 把 device 放入 usb_bus_type 的 dev 链表，
// 从 usb_bus_type 的 driver 链表里取出 usb_driver，
// 把 usb_interface 和 usb_driver 的 id_table 比较
// 如果能匹配，调用 usb_driver 的 probe
```

int hub_thread(void *__unused)是一个内核线程，平时休眠：

```
do {
hub_events();
wait_event_interruptible(khubd_wait, //在 khubd_wait 队列里休眠.
!list_empty(&hub_event_list) ||
kthread_should_stop());
try_to_freeze();
} while (!kthread_should_stop() || !list_empty(&hub_event_list));
```

平时在 khubd_wait 队列里休眠.，当有事件发生时，把它唤醒。搜索这个“khubd_wait”队列：

```
static void kick_khubd(struct usb_hub *hub)
{
    unsigned long flags;

    /* Suppress autosuspend until khubd runs */
    to_usb_interface(hub->intfdev)->pm_usage_cnt = 1;

    spin_lock_irqsave(&hub_event_lock, flags);
    if (list_empty(&hub->event_list)) {
        list_add_tail(&hub->event_list, &hub_event_list);
        wake_up(&khubd_wait);
    }
    spin_unlock_irqrestore(&hub_event_lock, flags);
}
```

上面显示出“唤醒队列”。kick“踢”，相当于把这个“khubd”线程踢醒。

这个“kick_khubd()”又会被“hub_irq()”调用。这个 irq 是“USB 主机控制器”程序注册的中断，不是 USB 设备的中断，USB 设备按规范只能接收信息不能发信息。

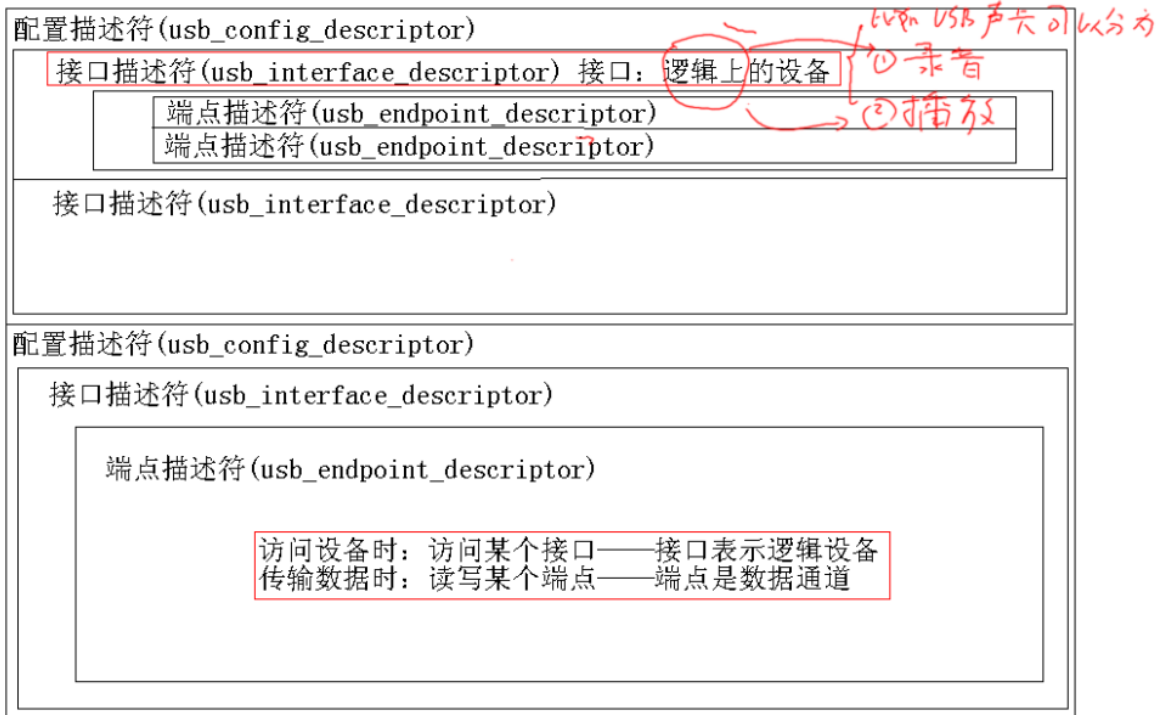
“描述符”：如下图。读出这些“描述符”后就对些 USB 设备清晰起来。根据这些信息找到它的驱动程序。

每一个硬件都有一个“设备描述符”。一个设备下可能会有多个“配置”，一个配置里可能会有多个“接口 — 逻辑设备”。

写驱动程序时，是给“接口 — 逻辑设备”写。所以说一个 USB 硬件可能安装多个驱动程序（因为可能有多个逻辑设备）。

一个“接口”里有多个“端点描述符”，“端点描述符”里说明一次性最多可以传输多大的数据。还有端上编号、方向、传输类型等信息。

include/linux/usb/ch9.h
设备描述符(usb_device_descriptor)



设备描述符:

```
/* USB_DT_DEVICE: Device descriptor */
struct usb_device_descriptor {
    __u8  bLength;
    __u8  bDescriptorType;

    __le16 bcdUSB;|
    __u8  bDeviceClass;
    __u8  bDeviceSubClass;
    __u8  bDeviceProtocol;
    __u8  bMaxPacketSize0;
    __le16 idVendor;
    __le16 idProduct;
    __le16 bcdDevice;
    __u8  iManufacturer;
    __u8  iProduct;
    __u8  iSerialNumber;
    __u8  bNumConfigurations;
} __attribute__ ((packed));
```

linux-2.6.22.6\include\linux\usb\Ch9.h 描述符的信息可以从 Ch9 是指 USB 协议第 9 章查到。

描述符是一些格式化的数据。第 9 章里有“usb_device_descriptor”USB 设备描述符。上面这个结构体就是“设备描述符”里有的内容:

bcdUSB -- USB 版本号
bDeviceClass -- 类
bDeviceSubClass -- 子类
bDeviceProtocol -- 协议
bMaxPacketSize0 -- 端点 0 的最大包大小

每个设备通过端口 0 识别出 USB 设备。把这个地址或命令信息发给端点 0，从端点 0 里读到那些描述符信息。

idVendor -- 厂家 ID。
idProduct -- 产品 ID。

bcdDevice

iManufacturer

iProduct -- 厂家

iSerialNumber

bNumConfigurations -- 配置的个数。是说这个 USB 设备有多少种配置。有配置自然有“配置描述符”。

把一个 USB 设备接到 PC 上，安装完驱动后在设备管理器中可以看到设备的详细信息（厂家 ID、产品 ID 等），这些信息就是 USB 主机控制器 通过发出命令 得到 USB 设备 里所谓的“usb_device_descriptor”设备描述符里得到的。

有“设备描述符”，还有“配置描述符”：


```

struct usb_config_descriptor {
    __u8  bLength;
    __u8  bDescriptorType;

    __le16 wTotalLength;
    __u8  bNumInterfaces;
    __u8  bConfigurationValue;
    __u8  iConfiguration;
    __u8  bmAttributes;
    __u8  bMaxPower;
} __attribute__((packed));

```

每一个 USB 硬件都有一个“USB 设备描述符”。可以发出某些命令得到这些“USB 设备描述符”。一个硬件有“USB 设备描述符”，硬件里还有“配置描述符”。一个设备可能有多种配置。

bLength：指这个“配置描述符”本身的长度。

bDescriptorType：类型。这里表示它是“配置描述符”。

wTotalLength：这个配置下所有其他信息的总长度。可以一次性全读出来。在配置下所有“接口描述

符”、“端点描述符”所有东西的大小。

bNumInterfaces：接口的个数。有“接口描述符”。每个配置下面可能有多个“接口”。

bConfigurationValue

iConfiguration

bmAttributes：属性

bMaxPower：消耗的电

“接口描述符”：逻辑上的设备。

每个配置下面可能多个“接口”，也可能只有一个。“接口描述符”接口就是“逻辑上的设备”。如一个 USB 声卡，它硬件上只有一个，但逻辑上可能会有两个功能（录音、播放）。这样就分成了两个逻辑设备，用两个“接口描述符”分别描述逻辑设备“录音”和“播放”。

```

/* USB_DT_INTERFACE: Interface descriptor */
struct usb_interface_descriptor {
    __u8  bLength;
    __u8  bDescriptorType;

    __u8  bInterfaceNumber;
    __u8  bAlternateSetting;
    __u8  bNumEndpoints;
    __u8  bInterfaceClass;
    __u8  bInterfaceSubClass;
    __u8  bInterfaceProtocol;
    __u8  iInterface;
} __attribute__((packed));

#define USB_DT_INTERFACE_SIZE 9

```

bLength : 此接口描述符的长度
bDescriptorType : 类型, 这里表示它是“接口描述符”。
bInterfaceNumber
bAlternateSetting
bNumEndpoints : 端点。表示设备下有多少个端点。
bInterfaceClass : 接口类
bInterfaceSubClass : 接口子类
bInterfaceProtocol : 接口协议
iInterface : 接口

bNumEndpoints : 端点。USB 传输的对象是端点。有端点就有“端点描述符”。表示设备下有多少个端点。

端点描述符:

```
/* USB_DT_ENDPOINT: Endpoint descriptor */
struct usb_endpoint_descriptor {
    __u8  bLength;
    __u8  bDescriptorType;

    __u8  bEndpointAddress;
    __u8  bmAttributes;
    __le16 wMaxPacketSize;
    __u8  bInterval;

    /* NOTE: these two are _only_ in audio endpoints. */
    /* use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof. */
    __u8  bRefresh;
    __u8  bSynchAddress;
} __attribute__((packed));
```



bEndpointAddress : 端点地址。如, 此设备是端点 1, 还是端点 2 或 3 等, 有个编号。
bmAttributes : 表示是哪种类型的端点。输入 或 输出, 是“实时端点”、“批量端点”或“中断端点”等。
就是说“传输的类型”是什么, 在“端点描述符”中有记录。
wMaxPacketSize : 最大包大小。在这个端点里面一次性最多可以给你多少数据(一次性读出多少数据)。
bInterval : 查询有频繁。如鼠标里有数据, PC 机应尽快的从里面得到数据, 里面的的传输类型是“中断传输端点”, 但并没有主动通知 PC 机的能力的。但给它一个中断的名字而实质是用“查询”方式来达到实时的查。查询有多频繁, 有个“bInterval”。

从“hub_port_connect_change()”开始分析如下的过程：

- 1.1 分配地址。（刚上来时就用默认地址 0。）
- 1.2 并告诉 USB 设备(set address)。
- 1.3 发出命令获取描述符。

```
hub_port_connect_change(struct usb_hub *hub, int port1, u16 portstatus, u16 portchange)
```

-->choose_address(udev); //选择地址。注释是说“设置地址”，实质是分配一个地址编号 1~127。但未告诉设备。

```
-->devnum = find_next_zero_bit(bus->devmap.devicemap, 128, bus->devnum_next);
```

查找下一个 0 位。显然是某地址用了，里面的某一位对应的就设置为 1。表示此地址正在使用。

```
-->if (devnum >= 128) devnum = find_next_zero_bit(bus->devmap.devicemap, 128, 1);
```

找到 128，若大于 128 时，就从头开始找。如：usb 1-1: new full speed USB device using s3c2410-ohci

```
and address 2
```

上面找到的是“address 2”，下一个将是“address 3”若找到最大值 128 时就返过来再找。

所以一个 USB 主机控制器里，最多可以接 1~127 一共 127 个 USB 设备。

```
-->hub_port_init(hub, udev, port1, i);
```

```
-->r = usb_control_msg(udev, usb_rcvaddr0pipe(), //USB 控制消息。用 USB 总线驱动程序提供的函数发送
```

一些控制消息。这是 4 种传输方式中的“控制

传输”。

```
-->USB_REQ_GET_DESCRIPTOR, USB_DIR_IN, //获得描述符。
```

分析到这里但没有遇到设置“USB 设备”的地址。看这个

“USB_REQ_GET_DESCRIPTOR”宏：

```
#define USB_REQ_SET_ADDRESS 0x05  
#define USB_REQ_GET_DESCRIPTOR 0x06
```

上面有一个“USB_REQ_SET_ADDRESS”设置 USB 设备地址的宏。搜索这个宏：

```
-->hub_set_address(struct usb_device *udev) //把地址(编号)告诉 USB 设备。
```

个函数也是在“hub_port_init()”中调用。这时以后 USB 设备就使用这个新地址了。

```
-->retval = usb_get_device_descriptor(udev, 8); 获取设备描述符。看上图描述符的概念。这里只获得 8 字节。
```

是因为还不知道你这个端点 0 一次性传输多少数据。看“设备描述符”结构前 8 字节的内容如下：

```

/* USB_DT_DEVICE: Device descriptor */
struct usb_device_descriptor {
    u8  bLength;
    u8  bDescriptorType;
    le16 bcdUSB;      前8字节
    u8  bDeviceClass;
    u8  bDeviceSubClass;
    u8  bDeviceProtocol;
    u8  bMaxPacketSize0;
    le16 idVendor;
    le16 idProduct;
    le16 bcdDevice;
    u8  iManufacturer;
    u8  iProduct;
    u8  iSerialNumber;
    u8  bNumConfigurations;
} __attribute__((packed));

```

从此 8 字节的“bMaxPacketSize0”中就可以知道一个端点 0 一次性能传输的最大数据大小。从这里得到大小后，以后就可以从这里更快的得到数据了。下面肯定还有 重读 的过程。

-->i = udev->descriptor.bMaxPacketSize0 == 0xff? 根据得到“设备描述符”中 bMaxPacketSize0 后设置它。然后再次获得“设备描述符”。

-->retval = usb_get_device_descriptor(udev, USB_DT_DEVICE_SIZE);

到这里“hub_port_init()”函数就结束了。接着再看“hub_port_connect_change()”还有哪些工作。

-->usb_new_device(udev); //新建一个 USB 设备。

-->usb_get_configuration(udev); //获得 USB 配置。读出所有描述符并解析。

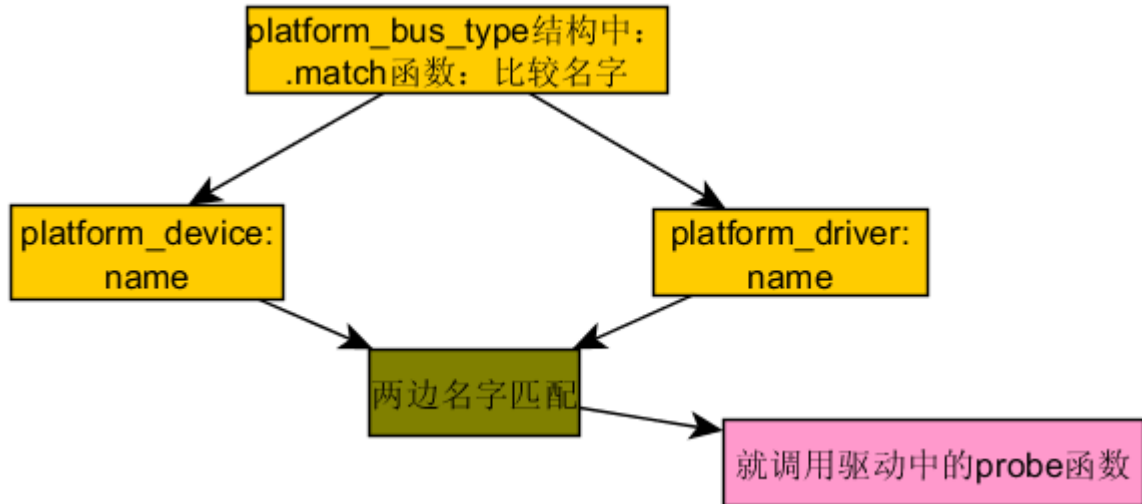
-->usb_get_descriptor(dev, USB_DT_CONFIG, cfgno, buffer, USB_DT_CONFIG_SIZE); //获得描述符。

-->length=max((int) le16_to_cpu(desc->wTotalLength), USB_DT_CONFIG_SIZE); 配置下接口、端点总大小。

-->usb_get_descriptor(dev, USB_DT_CONFIG, cfgno, bigbuffer, length); //读出全部(length). 再解析。

-->usb_parse_configuration(&dev->dev, cfgno, &dev->config[cfgno], bigbuffer, length); //解析描述符

-->device_add(&udev->dev); 回到总线设备驱动模型上。添加 USB 设备。把设备放到总线的 dev 链表。从总线的 driver 链表里取出 driver，一一比较。若能匹配上驱动，就调用 driver 中的“.probe”函数。



上面是在 hub.c 中的 “platform_bus_type”（平台设备驱动模型）。-- 比较名字
可以分析这个虚拟的 “总线 - 设备 - 驱动” 模型的框架：

搜索 “usb_new_device(udev)” 中的形参 “udev”：

最后跑到 usb.c 中 “usb_bus_type”（总线设备驱动模型）-- id_table 与 intf 比较
hub_port_connect_change() （hub.c 中）

```
-->struct usb_device *udev;
```

```
-->udev = usb_alloc_dev(hdev, hdev->bus, port1);
```

-->dev->dev.bus = &usb_bus_type; （usb.c 中）分配一个 udev 结构体。这个 udev 结构体中的总线为 “usb_bus_type”。这是另一条总线。

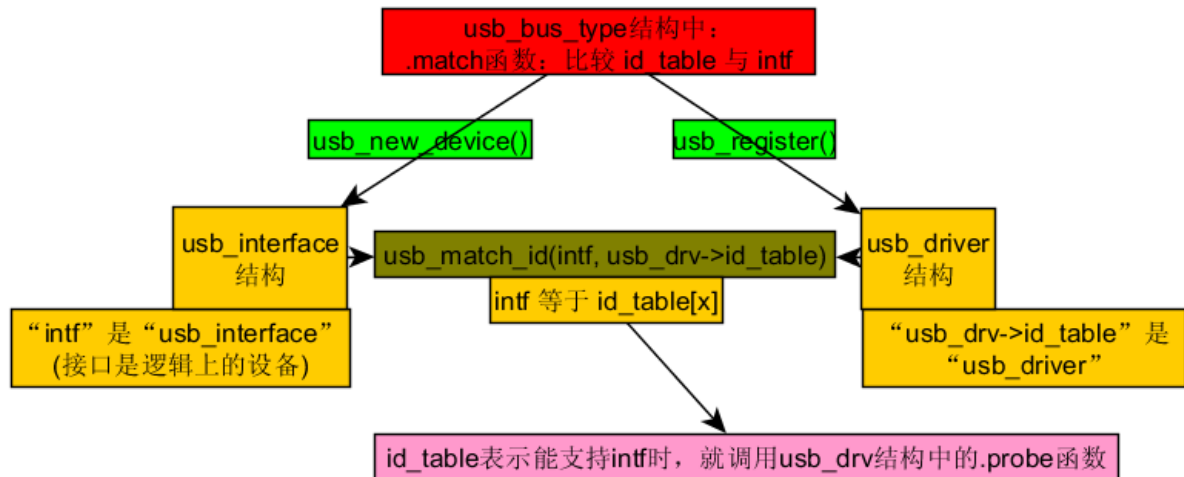
```

struct bus_type usb_bus_type = {
    .name = "usb",
    .match = usb_device_match,
    .uevent = usb_uevent,
    .suspend = usb_suspend,
    .resume = usb_resume,
};
  
```

这个 “usb_bus_type” 结构中有 “.match” 函数。int usb_device_match(struct device *dev, struct device_driver *drv)

-->id = usb_match_id(intf, usb_drv->id_table); 这条总线是与 “id_table” 比较。

一边 “usb_drv->id_table” 是 “usb_driver”，一边 “intf” 是 “usb_interface”（接口是逻辑上的设备）



看过“bus.c”中的“usb_bus_type”（总线设备驱动模型）后，再回到“hub.c”中的“platform_bus_type”（平台设备驱动模型）中的device_add(): Device_add()就会把自己注册进去。“usb_bus_type”（总线设备驱动模型）与“输入子系统框架”很像。

总结:

当接上一个USB设备时，就会产生一个中断(hub_irq())，在中断里会分配一个编号地址(choose_address(udev))。

再然后把这个地址告诉USB设备(hub_set_address())。

接着发出各种命令获取USB设备的“设备描述符”(usb_get_device_descriptor())。

再然后注册一个device(device_add())。这个被注册的device会被放到USB总线(usb_bus_type)的“设备链表”。并且会从总线(usb_bus_type)的“驱动链表”中取出“usb_driver”-USB设备驱动结构，把usb_interface和usb_driver的id_table比较，若能匹配就去调用“usb_driver”结构中的“.probe”函数。

要分析清楚“USB总线驱动程序”就看：<LINUX内核源代码情景分析>

三、USB驱动设备简单编写

USB总线驱动程序，在接入USB设备时，会帮我们构造一个新的usb_dev注册到“usb_bus_type”里去。这部分是内核做好的。我们要做的是，构造一个usb_driver结构体，注册到“usb_bus_type”中去。在“usb_driver”结构体中有“id_table”表示他能支持哪些设备，当USB设备能匹配id_table中某一个设备时，就会调用

“usb_driver”结构体中的“.probe”(自己确定在probe中做的事情)等函数，如当拔掉USB设备时，就会调用其中的“.disconnect”函数。

“usb_bus_type” USB 总线驱动设备模型只是提供了这一种框架而已。在“.probe”函数里面，注册“字符设备”也好，注册一个“input_dev”结构体也好，再或注册一个块设备也好。再或只是加了句打印也好，都可以由自己确定。

目标：

USB 鼠标用作按键：（相当于输入子系统）

左键 -- L

右键 -- S

中键 -- Enter

在“.probe”函数里做下面 4 件事情：

- 1, 分配一个 input_dev 结构体。
- 2, 设置，使其能产生按键类事件。
- 3, 注册这个 input_dev 结构。
- 4, 硬件相关操作：

之前的“按键”驱动程序，是注册某中断，在按键的中断里面读那些引脚，确定是按下还是松开，确定按键值。

触摸屏驱动程序里，是设置 ADC 控制器等。

现在 USB 鼠标里，是使用 USB 总线驱动程序提供的读写函数要收发数据。

所以，写 USB 设备驱动程序与以前的以前的驱动程序的差别就是 硬件的操作不一样了。

怎么写 USB 设备驱动程序？

1. 分配/设置 usb_driver 结构体

.id_table : 表示能支持的设备

.probe : 表示“USB 总线驱动程序”发现一个新设备后，就会与 driver 比较，若 id_table 表示能支持它，就调用.probe 函数。

.disconnect : 拔掉 USB 设备时调用这个函数。

2. 注册

例子: `linux-2.6.22.6\drivers\hid\usbhid\usbmouse.c`

*这是一个真正的 USB 鼠标驱动程序。

*/

从入口函数开始分析: `usbmouse.c`.

`usb_mouse_init(void)`:

`usb_register(&usb_mouse_driver)`; 注册一个 `usb_driver` 结构体 “`usb_moustr_driver`”.

```
static struct usb_driver usb_mouse_driver = {
    .name         = "usbmouse",
    .probe        = usb_mouse_probe,
    .disconnect   = usb_mouse_disconnect,
    .id_table     = usb_mouse_id_table,
};
```

假设有一个可以被 “.id_table”支持的 USB 设备出现了, 就会调用上面结构中的

“.probe = usb_mouse_probe,”

`int usb_mouse_probe(struct usb_interface *intf, const struct usb_device_id *id)`

-->`input_dev = input_allocate_device()`;分配一个 `input_dev` 结构体。

-->设置这个 `input_dev` 结构体。

`input_dev->evbit[0] = BIT(EV_KEY) | BIT(EV_REL)`; 能产生按键-EV_KEY 类事件或相对位移-EV_REL 类事件。

`input_dev->keybit[LONG(BTN_MOUSE)] = BIT(BTN_LEFT) | BIT(BTN_RIGHT) | BIT(BTN_MIDDLE)`;

支持按键类事件里的“左键”, “右键” “中键”

`input_dev->relbit[0] = BIT(REL_X) | BIT(REL_Y)`;

支持相对位移类事件的 X,Y 方向。

`input_dev->keybit[LONG(BTN_MOUSE)] |= BIT(BTN_SIDE) | BIT(BTN_EXTRA)`; //侧键, 额外按键。

`input_dev->relbit[0] |= BIT(REL_WHEEL)`; 能产生滚轮-REL_WHEEL 类事件。

-->`input_register_device(mouse->dev)`; 注册

之前的触摸屏驱动是从 2440 的 GPIO 管脚和 2440ADC 控制器里得到数据。现在的 USB 设备驱动程序要得到数据, 是发 USB 包。通过“USB 主机控制器”来得到那些数据。

代码框架：

```
: #include <linux/kernel.h>
: #include <linux/slab.h>
: #include <linux/module.h>
: #include <linux/init.h>
: #include <linux/usb/input.h>
: #include <linux/hid.h>
:
: //0, 分配一个 usb_driver 结构体, 设置并注册。
: //1, 分配一个 usb_driver 结构体。
: //1.1, 定义/设置 usb_driver结构变量 usbmouse_as_key_driver .
: static struct usb_driver usbmouse_as_key_driver = {
:     .name = "usbmouse_as_key",
:     .probe = usbmouse_as_key_probe,
:     .disconnect = usbmouse_as_key_disconnect,
:     .id_table = usbmouse_as_key_id_tebale,
: };
:
: static int usbmouse_as_key_init(void)
: {
:     //2,注册
:     usb_register(&usbmouse_as_key_driver);
:     return 0;
: }
:
: static void usbmouse_as_key_exit(void)
: {
:     //卸载。
:     usb_deregister(&usbmouse_as_key_driver);
:     return 0;
: }
:
: module_init(usbmouse_as_key_init);
: module_exit(usbmouse_as_key_exit);
: MODULE_LICENSE("GPL");
```



接着填充 usb_driver 结构中的三个函数。

一，id_table:

```
#define USB_INTERFACE_INFO(cl, sc, pr) \
    .match_flags = USB_DEVICE_ID_MATCH_INT_INFO, .bInterfaceClass = (cl), \
    .bInterfaceSubClass = (sc), .bInterfaceProtocol = (pr)
```

“.match_flags”：表示要匹配“设备描述符”中的哪一项。

“USB_DEVICE_ID_MATCH_INT_INFO”：INT 是接口意思，INT_INFO 就是指接口的信息。接口的信息就在“接口描述符”里。

“.bInterfaceClass”：只要“接口描述符”里的“类”是 (cl) 这个东西。

“.bInterfaceSubClass”：子类是 (sc) 这个东西。

“.bInterfaceProtocol”：协议是 (pr) 这个东西。

最后，只要这个设备的“接口描述符”里面匹配了上述的“类”“子类”和“协议”，这个USB设备，id_table就能支持。

```
static struct usb_device_id usbmouse_as_key_id_table [] = {
    { USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HID, USB_INTERFACE_SUBCLASS_BOOT,
        USB_INTERFACE_PROTOCOL_MOUSE) },
    { } /* Terminating entry */
};
```

只要这个USB设备里的“接口描述符”里面的“类”是“HID”类；“子类”是“BOOT”；“协议”是“MOUSE”，那么这个id_table就能支持。

若是想支持“某一款”设备，即“厂家ID”-VID，“设备ID”-PID这种情况：

```
/**
 * USB_DEVICE - macro used to describe a specific usb device
 * @vend: the 16 bit USB Vendor ID
 * @prod: the 16 bit USB Product ID
 *
 * This macro is used to create a struct usb_device_id that matches a
 * specific device.
 */
#define USB_DEVICE(vend, prod) \
    .match_flags = USB_DEVICE_ID_MATCH_DEVICE, .idVendor = (vend), \
    .idProduct = (prod)
```

假如这款设备的VID是“0x12d1”，PID是“0x1001”，则可以这样写：

```
static struct usb_device_id usbmouse_as_key_id_table [] = {
    { USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HID, USB_INTERFACE_SUBCLASS_BOOT,
        USB_INTERFACE_PROTOCOL_MOUSE) },
    { USB_DEVICE(0x12d1, 0x1001) },
    { } /* Terminating entry */
};
```

这样完全可以让此USB设备驱动程序只支持某个厂家的某款产品。

二，probe 函数：

“usb_driver”结构体是支持某个“usb_interface”结构体的某个接口。

一个USB硬件可能有多个逻辑上的设备，这些逻辑上的设备就是用“usb_interface”结构表示的。如一块“声卡”有“录音”和“播放”两个“逻辑接口”，则要两个驱动程序。

//1.3, 定义 probe 函数：

```
static int usbmouse_as_key_probe(struct usb_interface *intf, const struct
usb_device_id *id)
{
    printk("found usbmouse!\n");
    return 0;
}
```

暂时什么也不做。

三, disconnect 函数:

//1.4, disconnect 函数:

```
static void usbmouse_as_key_disconnect(struct usb_interface *intf)
{
    printk("disconnect usbmouse!\n");
}
```

暂时也什么也不做。

代码:

```
/*例子: inux-2.6.22.6\drivers\hid\usbhid\Usbmouse.c
 *这是一个真正的 USB 鼠标驱动程序。从入
 */
```

```
#include <linux/kernel.h>
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/usb/input.h>
#include <linux/hid.h>
```

//1.2, 定义 id_table :支持哪些 USB 设备。

```
static struct usb_device_id usbmouse_as_key_id_tebble [] = {
    { USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HID, USB_INTERFACE_SUBCLASS_BOOT,
    USB_INTERFACE_PROTOCOL_MOUSE) },
    { } /* Terminating entry */
};
```

//1.3, 定义 probe 函数:

```
static int usbmouse_as_key_probe(struct usb_interface *intf, const struct
usb_device_id *id)
{
    printk("found usbmouse!\n");
    return 0;
}
```

//1.4, disconnect 函数:

```
static void usbmouse_as_key_disconnect(struct usb_interface *intf)
{
    printk("disconnect usbmouse!\n");
}
```

//0, 分配一个 usb_driver 结构体, 设置并注册。

```

//1, 分配一个 usb_driver 结构体。
        //1.1, 定义/设置 usb_driver 结构变量 usbmouse_as_key_driver .
    static struct usb_driver usbmouse_as_key_driver = {
        .name          = "usbmouse_as_key",
        .probe         = usbmouse_as_key_probe,
        .disconnect    = usbmouse_as_key_disconnect,
        .id_table      = usbmouse_as_key_id_tebale,
    };

static int usbmouse_as_key_init(void)
{
//2,注册
usb_register(&usbmouse_as_key_driver);
return 0;
}

static void usbmouse_as_key_exit(void)
{
//卸载。
usb_deregister(&usbmouse_as_key_driver);
return 0;
}

module_init(usbmouse_as_key_init);
module_exit(usbmouse_as_key_exit);
MODULE_LICENSE("GPL");

```

四，测试：

测试 1th/2th:

make menuconfig 去掉原来的 USB 鼠标驱动

，不然一接 USB 鼠标内核会找到原来的驱动。新注册上的驱动还轮不上使用。

-> Device Drivers

 -> HID Devices

 <> USB Human Interface Device (full HID) support 此项不选中先不要内核中的驱动。

2. make ulmage 并使用新的内核启动

```

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

```

```
"Makefile" 10L, 184C written
book@book-desktop:/work/drivers_and_test/12th_usb/1th$ make
make -C /work/system/linux-2.6.22.6 M=`pwd` modules
make[1]: Entering directory `/work/system/linux-2.6.22.6`
CC [M] /work/drivers_and_test/12th_usb/1th/usbmouse_as_key.o
Building modules, stage 2.
MODPOST 1 modules
LD [M] /work/drivers_and_test/12th_usb/1th/usbmouse_as_key.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6`
book@book-desktop:/work/drivers_and_test/12th_usb/1th$ cp usbmouse_as_key.ko /work/nfs_root/first_fs
book@book-desktop:/work/drivers_and_test/12th_usb/1th$
```

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

下载新内核并在内存中启动

3. insmod usbmouse_as_key.ko

```
# mount -t nfs -o nolock,vers=2 192.168.1.5:/work/nfs_root/first_fs /mnt
# cd /mnt/
# insmod usbmouse_as_key.ko
usbcore: registered new interface driver usbmouse_as_key_
#
```

4. 在开发板上接入、拔出 USB 鼠标。

```
# usb 1-1: new full speed USB device using s3c2410-ohci and address 2
usb 1-1: configuration #1 chosen from 1 choice
found usbmouse!
usb 1-1: USB disconnect, address 2
disconnect usbmouse!
```

这就是最简单的 USB 鼠标驱动程序。

六，在 probe 函数中打印 厂家 ID 和 设备 ID：

VID 和 PID 都是在 USB “设备描述符”中的。当接入一个 USB 设备后，USB 总线驱动程序已经把这些“设备描述符”全部读了出来。直接使用即可。

1, 通过 “usb_interface” USB 接口得到 “usb_device” (USB 设备)结构体:

```
int usb_mouse_probe(struct usb_interface *intf, const struct usb_device_id *id)
```

```
→struct usb_device *dev = interface_to_usbdev(intf);
```

在 “usb_device” 结构定义中有 “设备描述符” -- VID,PID:

```
struct usb_device {
```

```
    . . . . .
```

```
    struct usb_device_descriptor descriptor; 设备描述符。
```

```
    . . . . .
```

```
}
```

```
/* USB_DT_DEVICE: Device descriptor */
struct usb_device_descriptor {
    __u8  bLength;
    __u8  bDescriptorType;

    __le16 bcdUSB;
    __u8  bDeviceClass;
    __u8  bDeviceSubClass;
    __u8  bDeviceProtocol;
    __u8  bMaxPacketSize0;
    __le16 idVendor;      厂家ID
    __le16 idProduct;   设备ID
    __le16 bcdDevice;
    __u8  iManufacturer;
    __u8  iProduct;
    __u8  iSerialNumber;
    __u8  bNumConfigurations;
} __attribute__ ((packed));
```



```
//1.3,定义 probe 函数:
```

```
static int usbmouse_as_key_probe(struct usb_interface *intf, const struct usb_device_id *id)
{
    struct usb_device *dev = interface_to_usbdev(intf);
    printk("found usbmouse!\n");
    printk("bcdUSB = %x\n", dev->descriptor.bcdUSB); //打印USB设备版本
    printk("VID = 0x%x\n", dev->descriptor.idVendor); //打印USB设备厂家ID
    printk("PID = 0x%x\n", dev->descriptor.idProduct); //打印USB设备产口ID
    return 0;
}
```

这些信息与在 WINDOWS 上看到的应该是一样的。（设备管理-驱动详细信息）

重新编译这个驱动代码：

```
#
# rmmod usbmouse_as_key
usbcore: deregistering interface driver usbmouse_as_key_
# insmod usbmouse_as_key.ko
usbcore: registered new interface driver usbmouse_as_key_
# usb 1-1: new full speed USB device using s3c2410-ohci and address 3
usb 1-1: configuration #1 chosen from 1 choice
found usbmouse!
bcdUSB = 200
VID = 0x46d
PID = 0xc52f
```

四、USB 鼠标驱动

在内核自带的实例“usbmouse.c”的“.probe”中会进一步判断设备是否为鼠标。

int usb_mouse_probe(struct usb_interface *intf, const struct usb_device_id *id)
—>得到设备接口

struct usb_host_interface *interface;

interface = intf->cur_altsetting; 当前设置

if (interface->desc.bNumEndpoints != 1) return -ENODEV; “bNumEndpoints--端点个数(除端点0外)”

```
/* USB_DT_INTERFACE: Interface descriptor */
struct usb_interface_descriptor {
    __u8  bLength;
    __u8  bDescriptorType;

    __u8  bInterfaceNumber;
    __u8  bAlternateSetting;
    __u8  bNumEndpoints;
    __u8  bInterfaceClass;
    __u8  bInterfaceSubClass;
    __u8  bInterfaceProtocol;
    __u8  iInterface;
} __attribute__((packed));
```

端点是USB传输的对象。收发数据是与某个端点来传输。端点0是每个USB设备都有的，bNumEndpoints是除端点0外的其他端点的个数。这里是说若除了端点0外，它的端点个数不是1时，就返回一个错误。说这个设备不支持。

endpoint = &interface->endpoint[0].desc; 若只有一个端点，就放到endpoint[]数组中，endpoint[0]是除了端点0外的第一个端点。这时是得到它的“端点描述符”。

if (!usb_endpoint_is_int_in(endpoint)) return -ENODEV; 若这不是中断类型的‘输入’端点。输入输出是主机控制器的概念。这里是指数据‘输入’给主机。端点描述符中有个属性。

```
/* USB_DT_ENDPOINT: Endpoint descriptor */
struct usb_endpoint_descriptor {
    __u8  bLength;
    __u8  bDescriptorType;

    u8    bEndpointAddress;
    __u8  bmAttributes;
    __le16 wMaxPacketSize;
    __u8  bInterval;

    /* NOTE: these two are _only_ in audio endpoints. */
    /* use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof. */
    __u8  bRefresh;
    __u8  bSynchAddress;
} __attribute__((packed));
```

我们在自己的代码中这个判断过程就不加上了。

下面直接开始：在 probe 函数中。

在我们的 usbmouse_as_key_probe() 中完美代码：将鼠标当键盘用。

一，分配一个 input_dev 结构。

```
//3, 分配一个 input_dev 结构体
//3.2, 分配这个 input_dev 结构体
uk_dev = input_allocate_device ();
```

二，设置这个 input_dev 结构

```
//4, 设置
//4.1, 能产生哪类事件. 这里把 USB 鼠标当键盘来用.
set_bit(EV_KEY, uk_dev->evbit); //能产生按键类事件. evbit 数组表示能产生哪一类.
set_bit(EV_REL, uk_dev->evbit); //能产生重复类事件--按下字母不动时会重复输入.
```



```

unsigned long evbit[NBITS(EV_MAX)]; //表示能产生哪类事件。 哪类事件 ← 哪些事件
unsigned long keybit[NBITS(KEY_MAX)]; //表示能产生哪些事件。这是“哪些”不同于“哪类”。
unsigned long relbit[NBITS(REL_MAX)]; //表示能产生哪些相对位移事件（如鼠标x, y和滚轮）
unsigned long absbit[NBITS(ABS_MAX)]; //表示能产生哪些绝对位移事件。
unsigned long mscbit[NBITS(MSC_MAX)];
unsigned long ledbit[NBITS(LED_MAX)];
unsigned long sndbit[NBITS(SND_MAX)];
unsigned long ffbbit[NBITS(FF_MAX)];
unsigned long swbit[NBITS(SW_MAX)];

```

//4.2, 能产生哪些事件。目的是想产生: 左键 -- L 右键 -- S 中键 -- Enter

```

set_bit(KEY_L, uk_dev->keybit); //L
set_bit(KEY_S, uk_dev->keybit); //S
set_bit(KEY_ENTER, uk_dev->keybit); //Enter

```

三, 注册

```

//5, 注册
input_register_device(uk_dev);

```

四, 硬件相关操作

以前的驱动程序, 数据是从中断里来, 或读寄存器, 引脚状态而来, 确定是什么数据。现在这里数据要用底层“USB 总线驱动程序”提供的函数来收发 USB 数据。

A, 数据传输 3 要素: 源, 目的, 长度。

1, 源: USB 设备的某个端点。

每个 USB 设备接上后都有个编号地址:

```

# usb 1-1: new full speed USB device using s3c2410-ohci and address 3
usb 1-1: configuration #1 chosen from 1 choice
found usbmouse!

```

```

pipe = usb_rcvintpipe (dev, endpoint->bEndpointAddress);
#define usb_rcvintpipe(dev, endpoint) \
    ((PIPE_INTERRUPT << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)

```

这个宏“usb_rcvintpipe(dev, endpoint)”就包含了 USB 设备的地址, 和哪一号端点(端点地址)。

“PIPE_INTERRUPT” 中断类型端点。

“USB_DIR_IN” 端点的方向。

“源-pipe”是个整数, 这个整数里含有端点的类型--PIPE_INTERRUPT, 和端点的方向--USB_DIR_IN。

“__create_pipe(dev, endpoint)” 这里既含有设备地址，也含有端点地址。

```
static inline unsigned int __create_pipe(struct usb_device *dev,
                                         unsigned int endpoint)
{
    return (dev->devnum << 8) | (endpoint << 15);
}
```

“devnum”：就是 USB 设备的地址（USB 设备编号）。

“endpoint”：就是端点的地址（bEndpointAddress）也是端点的一个编号（也是整数）。

2, 目的：从 USB 设备读数据，读到一个缓冲区，

分配一个缓冲区，不能用 kalloc 等，是用 “usb_buffer_alloc()”

```
void *usb_buffer_alloc(
struct usb_device *dev,
size_t size, //参 2, 分配多大的长度, 是
“usb_endpoint_descriptor.wMaxPacketSize”
gfp_t mem_flags,
dma_addr_t *dma //参 4, 这是物理地址的意思
)
```

3, 长度:

端点描述符中有长度.usb_endpoint_descriptor.wMaxPacketSize(最大包大小), 要作为
“usb_buffer_alloc()” 的形参 2.

//6, 硬件相关操作--三要素: 源

//6.1, 源:USB 设备某个端点。每个 USB 设备都有编号地址

```
pipe = usb_rcvintpipe (dev, endpoint->bEndpointAddress);
```

//6.3, 长度:端点描述符中有长度.usb_endpoint_descriptor.wMaxPacketSize(最大包大小)

```
len = endpoint->wMaxPacketSize;
```

//6.2, 目的:

```
usb_buf = usb_buffer_alloc(dev, len, GFP_ATOMIC, usb_buf_phys);
```

B, 使用三要素:

1, 要分配一个 “urb” : usb 请求块 (usb request block)

2,使用三要素前要设置 urb 。

a, Usb_fill_int_urb():fill(填充)中断类型的 urb。

```
inline void usb_fill_int_urb (struct urb *urb,
struct usb_device *dev,
unsigned int pipe, //三要素: 源
```

```

void *transfer_buffer,      //三要素：目的
int buffer_length,        //三要素：长度
usb_complete_t complete_fn, //complete_fn 完成函数。
void *context,            //给 complete_fn 使用
int interval)             //中断方式是通过查询，查询有多频繁(interval)。

```

鼠标是中断传输，实际上 USB 设备（鼠标）并没有主动通知“USB 主机控制器”的能力，即没有打断“USB 主机控制器”的能力。如何保证数据是“及时”的——用不断的查询。这个查询不是由 CPU 来查询，是由“USB 主机控制器”来不断查询，查询到数据后，USB 主机控制器发出中断（USB 主机控制器有中断 CPU 的能力）。USB 设备却没有中断“USB 主机控制器”的能力。USB 设备（USB 鼠标）所谓的“中断传输”是指“USB 主机控制器”不断的查询“USB 设备”是否有数据过来。

当“USB 主机控制器”得到数据后，“USB 总线驱动程序”就会调用“complete_fn——完成函数”

“查询频率”：中断方式是通过查询，查询有多频繁(interval)。端点描述符中有个“interval——间隔、间隙”。

b, USB 主机控制器得到“USB 设备”的数据后，要往某个内存去写。它需要的是物理地址。所以要告诉“USB 主机控制器”某个内存的物理地址。

c, 设置某些标记（不知道其意思）。

//使用三要素：

//6.4, 分配一个 usb request block(USB 请求块)

```
uk_urb = usb_alloc_urb(0, GFP_KERNEL);
```

//6.5, 使用三要素设置 urb

//6.5.1, 填充中断类型的 urb。

```
usb_fill_int_urb(uk_urb, dev, pipe, usb_buf, len, usbmouse_as_key_irq, NULL,
endpoint->bInterval);
```

//6.5.2, USB 主机控制收到 USB 设备数据要写到一个内存的物理地址.

```
uk_urb->transfer_dma = usb_buf_phys;
```

//6.5.3, 设置某些标记

```
uk_urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
```

3, 使用“urb”：提交 urb.usb_submit_urb()

C,usbmouse_as_key_irq 函数：

当“USB 主机控制器”来不断的查询“USB 设备 - 鼠标”，有数据后“USB 主机控制器”会存在 buf 中，然后“USB 主机控制器”产生一个中断。这时“usbmouse_as_key_irq”函数会调用。

内核示例 usbmouse.c 中“usbmouse_as_key_irq ()”根据数据判断是哪个按键或左右方向位移。再重新提交“urb”。

自己的代码也按这个思路写:

```
static void usbmouse_as_key_irq(struct urb *urb)
{
    int i;
    static int cnt = 0;
    //打印数据。用 hexdump 查含意。
    printk("data cnt %d: ", ++cnt);
    for(i = 0; i < len; i++)
    {
        printk("%02x", usb_buf[i]); //以后把打印改成 input_event 上报事件后就是
完整的驱动程序了。
    }
    printk("\n");

    //然后重新提交 urb。
    usb_submit_urb (uk_urb, GFP_KERNEL);
}
```

以上便结束了 probe 函数。

usbmouse_as_key_disconnect 函数:

//1.4, disconnect 函数:

```
static void usbmouse_as_key_disconnect(struct usb_interface *intf)
{
    struct usb_device *dev = interface_to_usbdev (intf); //从形参 intf 接口到
usb_device USB 设备结构体。
    //printk("disconnect usbmouse!\n");
    //1.4.1,提交过 urb ,这里杀掉 urb
    usb_kill_urb(uk_urb);
    //1.4.2,分配过 urb , 这里释放 urb
    usb_free_urb(uk_urb);
    //1.4.3, 分配过主机控制器从 USB 设备读数据后存放的 buffer,这里释放
    usb_buffer_free(dev, len, usb_buf, usb_buf_phys);
    //1.4.4,注册过 uk_dev ,这里卸载掉。
    input_unregister_device (uk_dev);
    //1.4.5,为 uk_dev 结构分配过空间,这里释放
    input_free_device (uk_dev);
}
```

编译:

```
book@book-desktop:/work/drivers_and_test/12th_usb/3th$ make
make -C /work/system/linux-2.6.22.6 M=`pwd` modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
  CC [M] /work/drivers_and_test/12th_usb/3th/usbmouse_as_key.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /work/drivers_and_test/12th_usb/3th/usbmouse_as_key.mod.o
  LD [M] /work/drivers_and_test/12th_usb/3th/usbmouse_as_key.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
book@book-desktop:/work/drivers_and_test/12th_usb/3th$ cp usbmouse_as_key.ko /work/nfs_root/first_fs
book@book-desktop:/work/drivers_and_test/12th_usb/3th$
```

测试 3th:

1. insmod usbmouse_as_key.ko

```
# rmmod usbmouse_as_key
usbcore: deregistering interface driver usbmouse_as_key_
# insmod usbmouse_as_key.ko
usbcore: registered new interface driver usbmouse_as_key_
# ls /dev/event*
```

2. ls /dev/event*

```
# ls /dev/event*
ls: /dev/event*: No such file or directory
#
```

3. 接上 USB 鼠标。

4. ls /dev/event*

```
# usb 1-1: new full speed USB device using s3c2410-ohci and address 4
usb 1-1: configuration #1 chosen from 1 choice
input: Unspecified device as /class/input/input0
#
# ls /dev/event*
/dev/event0
#
```

/dev/event0 就对应了鼠标。

5. 操作鼠标观察数据

```
# data cnt 1: 00 00 ff ff fb ff 00 00
data cnt 2: 00 00 fe ff fc ff 00 00
data cnt 3: 00 00 fc ff fb ff 00 00
data cnt 4: 00 00 ff ff fc ff 00 00
data cnt 5: 00 00 fe ff fb ff 00 00
data cnt 6: 00 00 fe ff fa ff 00 00
data cnt 7: 00 00 fc ff f8 ff 00 00
data cnt 8: 00 00 fd ff f9 ff 00 00
data cnt 9: 00 00 00 00 fe ff 00 00
data cnt 10: 00 00 02 00 ff ff 00 00
data cnt 11: 00 00 00 00 ff ff 00 00
data cnt 12: 00 00 00 00 ff ff 00 00
data cnt 13: 00 00 00 00 ff ff 00 00
data cnt 14: 00 00 00 00 ff ff 00 00
data cnt 15: 00 00 00 00 ff ff 00 00
```

上面是 16 进制。有 8 个字节。

```
00 00 00 00 00 00 00 00
按键 X 方向位移 Y 方向位移 滚轮
```

不同的鼠标可能不一样，有的是 4 字节。

```
00 00 00 00
按键 X 方向位移 Y 方向位移 滚轮
```

X 方向：往左移是负数，往右移是正数。

Y 方向：往上移是负数，往下移是正数。

滚 轮：往前滑是正数，往后滑是负数。

左键：

```
# data cnt 27: 01 00 00 00 00 00 00 00
```

第一个字节的数据“01”里的“1”是 bit0，表示“左键”。

松开：

```
# data cnt 28: 00 00 00 00 00 00 00 00
```

右键：

```
# data cnt 31: 02 00 00 00 00 00 00 00
```

“02”中的“2”就是 bit1。

中键：

```
# data cnt 47: 04 00 00 00 00 00 00 00
```

“04”中的“4”就是 bit2。

左、右键同时按下：

```
data cnt 52: 03 00 00 00 00 00 00 00
```

往右边移：

```
data cnt 86: 00 00 01 00 00 00 00 00
```

正值表示往右边移。

```
data cnt 87: 00 00 fe ff 00 00 00 00
```

这是个负值，表示往左边移。

明白鼠标数据的含义后，在 `usbmouse_as_key_irq` 函数中，将打印鼠标数据的过程改成上报事件。

USB 总线驱动程序提供“识别设备”，“给设备找到驱动”，提供“读写数据”，只收发数据却并不知道数据的含意。

USB 设备驱动程序知道数据的含意。数据的含意需要“USB 设备”驱动程序来解析。从上面打印的鼠标数据分析知道了数据的含意。

```
static void usbmouse_as_key_irq(struct urb *urb)
```

```

{
    static unsigned char pre_val; //上一次数据
#ifdef 0
    int i;
    static int cnt = 0;
    //打印数据。用 hexdump 查含意。
    printk("data cnt %d: ", ++cnt);
    for(i = 0; i < len; i++)
    {
        printk("%02x", usb_buf[i]); //以后把打印改成 input_event 上报事件后就是
完整的驱动程序了。
    }
    printk("\n");
#endif
    /* USB 鼠标数据含意:
    * data[0]: bit0-左键, 1-按下, 0-松开
    *          bit1-右键, 1-按下, 0-松开
    *          bit2-中键, 1-按下, 0-松开
    */
    //如果上一次数据的 bit0 不等于这一次数据的 bit0, 那么就是左键发生了变化
    if((pre_val & (1<<0)) != (usb_buf[0] & (1<<0)))
    { //左键--L 发生了变化 (usb_buf & (1<<0)) ? 1 : 0, 即(usb_buf & (1<<0))有值
    时, 就是表示 1, 是按下。
        input_event(uk_dev, EV_KEY, KEY_L, (usb_buf[0] & (1<<0)) ? 1 : 0);
        input_sync (uk_dev);
    }

    if((pre_val & (1<<1)) != (usb_buf[0] & (1<<1)))
    { //右键--S 发生了变化 bit1
        input_event(uk_dev, EV_KEY, KEY_S, (usb_buf[0] & (1<<1)) ? 1 : 0);
        input_sync (uk_dev);
    }

    if((pre_val & (1<<2)) != (usb_buf[0] & (1<<2)))
    { //中键--回车发生了变化 bit2
        input_event(uk_dev, EV_KEY, KEY_ENTER, (usb_buf[0] & (1<<2)) ? 1 : 0);
        input_sync (uk_dev);
    }
    pre_val = usb_buf[0]; //保存当前值
    //然后重新提交 urb。
    usb_submit_urb (uk_urb, GFP_KERNEL);
}

```

```
book@book-desktop:/work/drivers_and_test/12th_usb/4th$ make
make -C /work/system/linux-2.6.22.6 M=`pwd` modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
  CC [M] /work/drivers_and_test/12th_usb/4th/usbmouse_as_key.o
Building modules, stage 2.
MODPOST 1 modules
  CC      /work/drivers_and_test/12th_usb/4th/usbmouse_as_key.mod.o
  LD [M] /work/drivers_and_test/12th_usb/4th/usbmouse_as_key.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
book@book-desktop:/work/drivers_and_test/12th_usb/4th$ cd ..
```

测试 4th:

1. insmod usbmouse_as_key.ko

```
# rmmod usbmouse_as_key
usbcore: deregistering interface driver usbmouse_as_key_
data cnt 330: 00 00 00 00 00 00 00 00
# insmod usbmouse_as_key.ko
input: Unspecified device as /class/input/input1
usbcore: registered new interface driver usbmouse_as_key_
#
```

不需要再拔插。直接卸载旧驱动加载新驱动。

2. ls /dev/event*

```
# ls /dev/event0
/dev/event0
#
```



3. cat /dev/tty1 然后按鼠标键

```
# cat /dev/tty1
ls
```

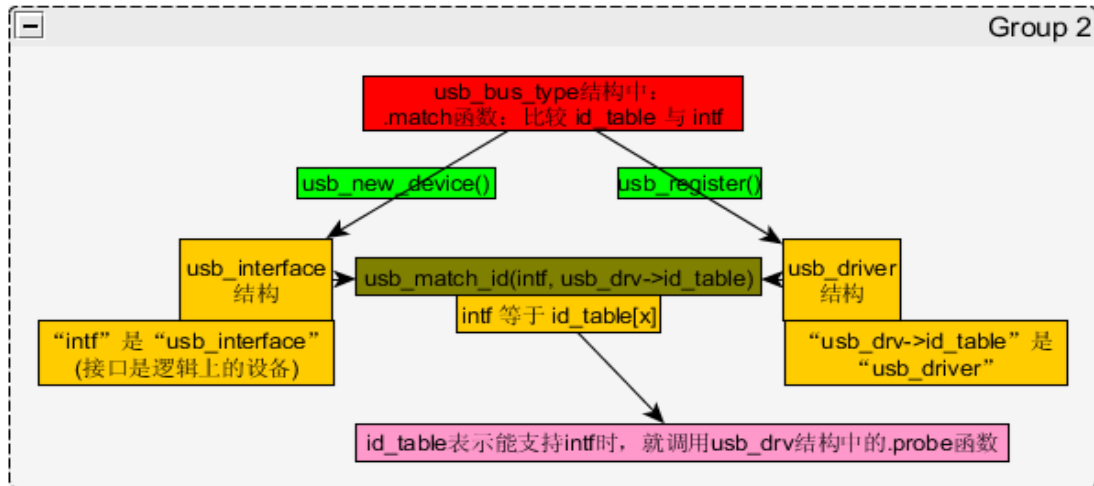
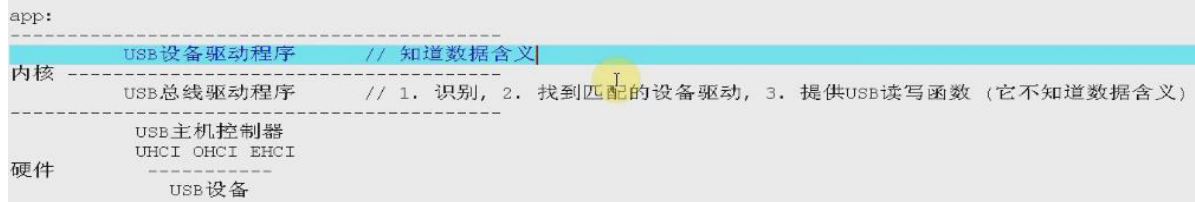
左键+右键后回车

4. hexdump /dev/event0 要最后那个 event (并不一定是 event0)

```
# hexdump /dev/event0
00000000 099a 0000 8eea 000e 0001 0026 0001 0000
00000010 099a 0000 8ef3 000e 0000 0000 0000 0000
00000020 099b 0000 db03 0000 0001 0026 0000 0000
00000030 099b 0000 db0b 0000 0000 0000 0000 0000
00000040 099b 0000 b2e4 0008 0001 001f 0001 0000
```

秒	微秒	按键类	哪个按键	按下
099a 0000	8eea 000e	0001	0026	0001 0000

USB驱动程序框架:



总结:



根据“usb_bus_type”总线驱动设备驱动模型，里面有个“.match”函数，左边是由“USB 总线驱动程序”帮我们来发现这个新“USB 设备”，会注册“usb_new_device()”一个“USB 设备”，并且从右边“driver”链表里找了一个个 USB 驱动程序和左边注册进来的“USB 设备”比较。所谓的比较，是“usb_bus_type”中的“.match”函数，把左边“usb_interface”结构中的“接口”与右边“usb_driver”结构中的“id_table”比较。若能吻合，则调用“usb_driver”中的“.probe”函数。“usb_bus_type”提供了这套机制。

在“.probe”函数中，可以只是打印，也可以注册字符设备，或注册“input_dev”结构。完全由自己确定。

以前的驱动程序，数据是从“中断”（按键中断，ADC 中断）里面读寄存器得到。现在“USB 设备驱动程序”中的数据从“USB 总线”来，是 USB 总线驱动程序提供的函数（读写等）发起 USB 传输，从 USB 传输里得到那些数据。（数据传输三要素：源，目的，长度。再构造一个“usb_urb = usb_alloc_urb(0, GFP_KERNEL)”后，接着把“源，目的，长度”填充到“usb_urb”中，使用就是提交“usb_urb”，提交 usb_submit_urb 函数是 USB 总线驱动程序提供的）。

当“USB 主机控制器”接收到数据后，“usb_as_key_irq”函数（complete_fn 完成函数）被调用。在这个函数里根据“USB 设备”数据的含义去上报(input_event())