Lsusb

/sys/device/usbmon/

/proc/bus/usb/devices

<< Linux那些事儿之我是 USB >>

### USB 总线简介

USB 是一种“总线”，它与传统的外部设备与主机之间的连接方式不同，它允许将不同种类的外部设备混合连接到同一个接口上(这体现它的通用性)，但是，它又和计算机内部的总线( 如 PCI 总线)不同，CPU不能通过访问内指令或 I/O指令直接访问连接在 USB 上的设备，而要通过一个“USB 控制器”，间接地与连接在 USB 上的设备打交道， USB 总线存在于计算机的外部，所以说 USB 总线是外部总线。

<http://blog.csdn.net/alan445947767/article/details/41170367>

#### USB 传输类型

USB 设备的数据传输在 USB 协议里分为 4 大传输类型：控制传输、中断传输、批量传输、实时传输。

##### 控制传输

控制传输是每一个 USB 设备都必须支持的，它通常用来获取设备描述符、设置设备的状态等。 控制传输可靠、实时， USB 设备的识别过程就是控制传输。

##### 中断传输

注意了，这里所谓的中断传输，并不是真正的“中断”， USB 设备并没有中断 USB 主机的本事，这里只是借用一下中断的概念，它其实是一种轮询的方式来完成数据的传输。 支持中断传输的 USB 设备多了去了，比如 USB 鼠标、 USB键盘等等。

##### 批量传输

批量，顾名思义就是大量的数据拷贝，批量传输它能够保证数据的准确性，但是不能保证时间的实时性。支持批量传输的 USB 设备，最常见的莫过于 U 盘，还有移动硬盘。

##### 实时传输

实时，顾名思义就是时间要求严格，对传输延时敏感。换句话说，对实时性要求比较高。 USB 摄像头是一个典型的代表，想到这个你就很好理解为什么实时性要求比较高了。 想象一下，某天你和某个 MM 进行 QQ 视频聊天，传过来的视频要是一卡一卡的，你不崩溃？ 当然了，实时传输是一种不可靠的传输，还记得以前学网络编程说过的？ QQ 视频传输是一种以 UDP 协议来传输数据的。

#### USB 的主从结构

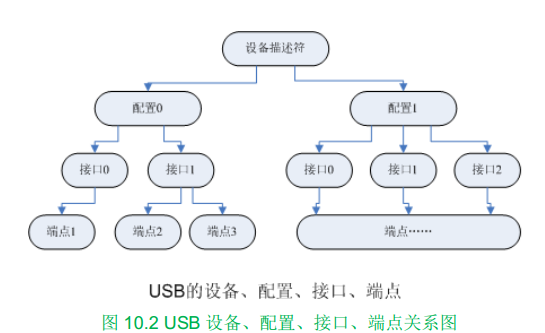
所有的 USB 传输，**都是从 USB 主机这方发起； USB 设备没有"主动"通知USB 主机的能力。** 比如： USB 鼠标滑动一下立刻产生数据，但是它没有能力通知 PC 机来读数据，只能被动地等得 PC 机来读。 这就是为什么前面我们讲过的中断传输，并不是实际意义上的中断传输，因为它还没那个本事呢。

#### USB 传输的对象

说了那么多传输，到底传输的对象是什么？ U 盘？鼠标？键盘？不，这是和非专业型人说的。专业点呢？ 端点(endpoint)！比如： 我们说"读 U 盘"、 "写 U 盘"， 换句话说：**把数据写到 U 盘的端点 1 ，从 U 盘的端点 2 里读出数据。注意了，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。 端点 1 、 2 等一般用作数据端点，存放主机与设备间往来的数据。**

### 设备、配置、接口、端点

在 USB 设备的逻辑组织中，包含设备、配置、接口和端点 4 个层次。 这 4 者之间的关系不用多介绍，只需看图 10.2 即可明白。



从上图可知，一个 USB 设备通常有一个或多个配置；配置通常又有一个或多个接口；接口通常又还有零个或多个端点。

#### 设备描述符

在 Linux 系统中，这种层次化配置信息用一组标准的描述符来描述。 当你将一个 USB 设备插入电脑的时候，电脑很快就知道这个 USB 设备是什么东西了。有没有想过为什么它会那么聪明？ 其实， USB 协议里规定了， PC 和 USB 设备都得遵守一定的规范。 PC 里事先安装有 **USB 总线驱动程序**，当 USB 插入电脑时， USB 总线驱动程序就会发出某些命令去获取设备的信息(描述符)， USB 设备收到这个命令后，将设备描述符返回给 PC。

/\* 参考 include\uapi\linux\usb\ch9.h \*/

struct usb\_device\_descriptor {

\_\_u8 bLength; /\* 描述符长度 \*/

\_\_u8 bDescriptorType; /\* 描述符类型编号 \*/

\_\_le16 bcdUSB; /\* USB 版本号 \*/

\_\_u8 bDeviceClass; /\* USB 分配的设备类 code \*/

\_\_u8 bDeviceSubClass; /\* USB 分配的子类 code \*/

\_\_u8 bDeviceProtocol; /\* USB 分配的协议 code \*/

\_\_u8 bMaxPacketSize0; /\* endpoint0 最大包大小 \*/

\_\_le16 idVendor; /\* 厂商编号 \*/

\_\_le16 idProduct; /\* 产品编号 \*/

\_\_le16 bcdDevice; /\* 设备出厂编号 \*/

\_\_u8 iManufacturer; /\* 描述厂商字符串的索引 \*/

\_\_u8 iProduct; /\* 描述产品字符串的索引 \*/

\_\_u8 iSerialNumber; /\* 描述设备序列号字符串的索引 \*/

\_\_u8 bNumConfigurations; /\* 配置的数量 \*/

} \_\_attribute\_\_ ((packed));

#define USB\_DT\_DEVICE\_SIZE 18

#### 配置描述符

每个 USB 设备都提供了不同级别的配置信息，可以包含一个或多个配置，不同的配置使设备表现出不同的功能组合（在探测/连接期间需从其中选定一个），配置由多个接口组成。

/\* 参考 include\uapi\linux\usb\ch9.h \*/

struct usb\_config\_descriptor {

\_\_u8 bLength; /\* 描述符长度 \*/

\_\_u8 bDescriptorType; /\* 描述符类型编号 \*/

\_\_le16 wTotalLength; /\* 配置所返回的所有数据的大小 \*/

\_\_u8 bNumInterfaces; /\* 配置所支持的接口数 \*/

\_\_u8 bConfigurationValue; /\* Set\_Configuration 命令需要的参数值 \*/

\_\_u8 iConfiguration; /\* 描述该配置的字符串的索引值 \*/

\_\_u8 bmAttributes;

\_\_u8 bMaxPower;

} \_\_attribute\_\_ ((packed));

#define USB\_DT\_CONFIG\_SIZE 9

#### 接口描述符

接口是逻辑上的设备， 一个 USB 设备可以有多个接口，比如一个 USB 声卡设备， 它有录音接口，也有播放接口。

/\* 参考 include\uapi\linux\usb\ch9.h \*/

struct usb\_interface\_descriptor {

\_\_u8 bLength; /\* 描述符长度 \*/

\_\_u8 bDescriptorType; /\* 描述符类型编号 \*/

\_\_u8 bInterfaceNumber; /\* 接口的编号 \*/

\_\_u8 bAlternateSetting; /\* 备用的接口描述符编号 \*/

\_\_u8 bNumEndpoints; /\* 端点数，不包括端点 0 \*/

\_\_u8 bInterfaceClass; /\* 接口类 \*/

\_\_u8 bInterfaceSubClass; /\* 接口子类 \*/

\_\_u8 bInterfaceProtocol; /\* 接口所遵循的协议 \*/

\_\_u8 iInterface; /\* 描述该接口的字符串索引值 \*/

} \_\_attribute\_\_ ((packed));

#define USB\_DT\_INTERFACE\_SIZE 9

#### 端点描述符

端点是 USB 通信的最基本形式，每一个 USB 设备接口在主机看来就是一个端点的集合。主机只能通过端点与设备进行通信，以使用设备的功能。在 USB系统中每一个端点都有惟一的地址，这是由设备地址和端点号给出的。每个端点都有一定的属性，其中包括传输方式、总线访问频率、带宽、端点号和数据包的最大容量等。

struct usb\_endpoint\_descriptor {

\_\_u8 bLength; /\* 描述符长度 \*/

\_\_u8 bDescriptorType; /\* 描述符类型编号 \*/

/\* 端点地址： 0～3 位是端点号，第 7 位是方向(0-OUT,1 -IN) \*/

\_\_u8 bEndpointAddress;

/\* 端点属性： bit[0:1] 的值为 00 表示控制，

\* 为 01 表示实时，为 02 表示批量，为 03 中断

\*/

\_\_u8 bmAttributes;

\_\_le16 wMaxPacketSize; /\* 本端点接收或发送的最大信息包的大小 \*/

\_\_u8 bInterval; /\* 轮询数据传送端点的时间间隔 \*/

/\* NOTE: these two are \_only\_ in audio endpoints. \*/

/\* use USB\_DT\_ENDPOINT\*\_SIZE in bLength, not sizeof. \*/

\_\_u8 bRefresh;

\_\_u8 bSynchAddress;

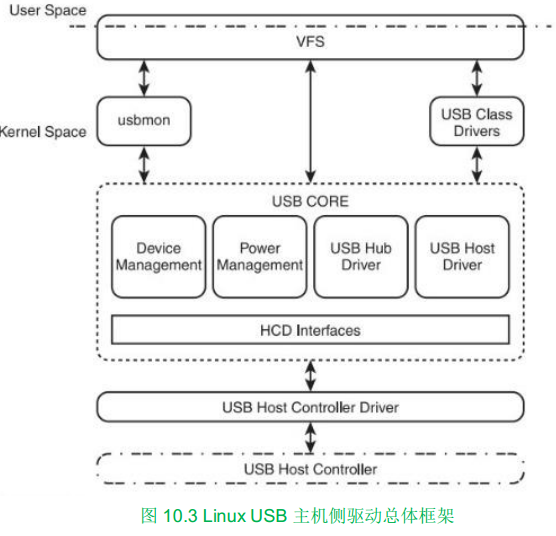
} \_\_attribute\_\_ ((packed));

#define USB\_DT\_ENDPOINT\_SIZE 7

### USB 驱动程序

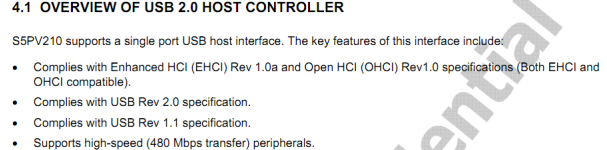
在 Linux 系统中，提供了主机侧和设备侧视角的 USB 驱动框架，这里，仅仅讲解主机侧角度看到的 USB 驱动框架。

从主机侧的角度而言，需要编写的 USB 驱动程序包括**主机控制器驱动和设备驱动**两类。 USB 主机控制器驱动程序控制插入其中的 USB 设备，而 USB 设备驱动程序控制该设备如何作为设备与主机通信。 在 USB 主机控制器驱动和USB 设备驱动之间还有一层叫 USB 核心层。 **USB 核心负责 USB 驱动管理和协议处理工作，它通过定义一些数据结构、宏和功能函数，向上为 USB 设备驱动提供编程接口，向下为 USB 主机控制器驱动提供编程接口；通过全局变量维护整个系统的 USB 设备信息，完成设备热插拔控制、总线数据传输控制等。**



### S5PV210 主机控制器驱动的移植

USB 主机控制器有 3 种规范， UHCI(Universal Host Controller Interface)，这种规范主要是 Intel、 Via 芯片公司提供支持 PC 主板的； OHCI(Open HostController Interface)，这种规范是微软提出来的，主要应用在非 PC 系统上的嵌入式领域上的 USB 芯片； EHCI(Enhanced Host Controller Interface)，这种后来为提高 USB 速度而提出的规范，它支持最高速度为 480Mbps。在《S5PV210\_UM\_REV1.1 》手册上搜索 OHCI 关键词，会发现下面一段话



这表明 S5PV210 这款 CPU 支持一个 USB 主机接口，同时支持 EHCI 和OHCI 这两种规范，支持 USB1.1 和 USB2.0 规范，支持最高的外设传输速率为480Mbps。 注意了，它并不支持 USB3.0 规范的 USB 设备，所以做测试的时候，千万不要拿 USB3.0 规范的 USB 设备去测试。

#### 移植

ohci-s5p 驱动打开内核目录:drivers\usb\host\，发现 Linux 系统提供了大量的主机控制器驱动，找遍所有平台，都没有找到 ohci-s5p.c 源码。 很遗憾， 3.8 的内核没有提供 S5PV210 的 USB HOST 控制器驱动程序。最好验证有没有提供的办法就是，烧写网蜂提供的第一版的 uImage进去，然后找个 U 盘、或者鼠标插入 Webee210开发板的 USB HOST 接口，看看串口有没有打印什么信息，结果是不会有任何反应的。 既然没有提供， 这就需要我们自己来编写了，这下不好办了吧？

不用紧张，仔细再找找，还是能发现一些类似的源码，可供我们移植的。 我们发现，内核虽然没有提供 ohci-s5p.c 源码，但是有提供 ehci-s5p.c 源码，还有 ohci 相关的其他平台的源码，比如 ohci-s3c2410.c、 ohci-exynos.c 供我们移植参考。

##### ohci-s5p.c 的移花接木

内核既然没有 ohci-s5p.c，那我们使用其他平台的ohci 源码，这里我们拷贝drivers\usb\host\目录下的 ohci-exynos.c 为 ohci-s5p.c。 然后将所有 exynos 字符串替换成 s5p， 由于有些地方是 exynos4，所以还需要将 s5p4 替换为 s5p。最后还需要修改一下头文件，将

#include <linux/platform\_data/usb-exynos.h>

改为：

#include <linux/platform\_data/usb-ohci-s5p.h>

##### **usb-ohci-s5p.h** 的移花接木

打开内核目录 include\linux\platform\_data\，然后 拷贝 usb-exynos.h 为usb-ohci-s5p.h。将所有的 exynos4 字符串替换为 s5p， 将 EXYNOS 替换为 S5P。最后添加平台数据：

static struct s5p\_ohci\_platdata s5p\_ohci\_platdata;

为了以后支持 EHCI 还添加 echi 的平台数据，最后 usb-ohci-s5p.h 修改为：

#ifndef \_\_MACH\_S5P\_OHCI\_H

#define \_\_MACH\_S5P\_OHCI\_H

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*Add by Webee\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifdef CONFIG\_S5P\_DEV\_USB\_EHCI

static struct s5p\_ehci\_platdata s5p\_ehci\_platdata;

#endif

static struct s5p\_ohci\_platdata s5p\_ohci\_platdata;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*Add by Webee\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

struct s5p\_ohci\_platdata {

int (\*phy\_init)(struct platform\_device \*pdev, int type);

int (\*phy\_exit)(struct platform\_device \*pdev, int type);

};

extern void s5p\_ohci\_set\_platdata(struct s5p\_ohci\_platdata \*pd);

#endif /\* \_\_MACH\_S5P\_OHCI\_H \*/

##### 添加 s5p\_ohci\_driver 到 ohci-hcd.c

打开 drivers\usb\host\ ohci-hcd.c，在 CONFIG\_USB\_OHCI\_EXYNOS 前面添加如下代码：

#ifdef CONFIG\_USB\_OHCI\_S5P

#include "ohci-s5p.c"

#define PLATFORM\_DRIVER s5p\_ohci\_driver

#endif

因为 S5PV210 USB HOST 控制器驱动由 drivers\usb\host\ ohci-hcd.c(支持各种 SoC 下的主机控制器驱动的通用部分)和 drivers\usb\host\ohci-s5p.c 共同完成。

##### 添加平台设备

前面我们移植 ohci-s5p.c 主要是围绕 platform\_driver 来编程的，这又回到了平台驱动设备模型了。还记得我们移植 gpio-key 驱动了吗？里面就添加了平台设备来支持平台驱动。今天，我们同样需要添加平台设备来支持 s5p\_ohci\_driver这个平台驱动。 怎么添加呢？参考别人怎么写！

打开 **arch\arm\plat-samsung\devs.c，**找到 s5p\_device\_ehci 这个平台设备，模仿它来修改。

打开 **arch\arm\mach-s5pv210\mach-smdkv210.c**，在 smdkv210\_devices[ ]前，添加如下代码：

**#include <linux/platform\_data/usb-ohci-s5p.h>**

static struct resource s5p\_ohci\_resource[] = {

[0] = DEFINE\_RES\_MEM(0xEC300000, SZ\_256),

[1] = DEFINE\_RES\_IRQ(S5P\_IRQ\_VIC1(23)),

};

static u64 samsung\_device\_dma\_mask = DMA\_BIT\_MASK(32);

struct platform\_device s5p\_device\_ohci = {

.name = "s5p-ohci",

.id = -1,

.num\_resources = ARRAY\_SIZE(s5p\_ohci\_resource),

.resource = s5p\_ohci\_resource,

.dev = {

.dma\_mask = &samsung\_device\_dma\_mask,

.coherent\_dma\_mask = DMA\_BIT\_MASK(32),

}

};

void \_\_init s5p\_ohci\_set\_platdata(struct s5p\_ohci\_platdata \*pd)

{

struct s5p\_ohci\_platdata \*npd;

npd = s3c\_set\_platdata(pd, sizeof(struct s5p\_ohci\_platdata),&s5p\_device\_ohci);

if (!npd->phy\_init)

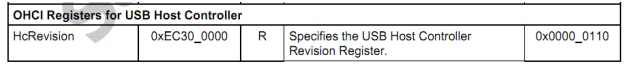
npd->phy\_init = s5p\_usb\_phy\_init;

if (!npd->phy\_exit)

npd->phy\_exit = s5p\_usb\_phy\_exit;

}

怎 么 确 定 s5p\_ohci\_resource 里 面 的 内 存 地 址 呢 ？ 这 自 然 要 回 到《S5PV210\_UM\_REV1.1 》手册了，在 USB HOST 这章的寄存器介绍里面有这么一段描述：



而 IRQ 的确定，则是找到下面这段话。

**/\* 参考 arch\arm\mach-s5pc100\include\mach\irqs.h \*/**

#define IRQ\_UHOST S5P\_IRQ\_VIC1(23)

然后将定义设置好的 s5p\_device\_ohci 添加到 smdkv210\_devices[ ]，如：

static struct platform\_device \*smdkv210\_devices[] \_\_initdata = {

&s5p\_device\_ohci, /\* Add by Webee \*/

……

&webee210\_button\_device, /\* Add by Webee \*/

};

最后，在 smdkv210\_machine\_init 函数中添加平台数据的设置函数。

#ifdef CONFIG\_S5P\_DEV\_USB\_OHCI

s5p\_ohci\_set\_platdata(&s5p\_ohci\_platdata);

#endif

##### 移植 **Kconfig**

###### 一、移植 drivers\usb\host\目录下的 Kconfig

打开 drivers\usb\host\目录下的 Kconfig，在 USB\_OHCI\_EXYNOS 前面添加 USB\_OHCI\_S5P 的配置支持。修改后如下：

# Add by Webee

config USB\_OHCI\_S5P

boolean "OHCI support for Samsung S5P SoC Series"

depends on USB\_OHCI\_HCD && PLAT\_S5P

select S5P\_DEV\_USB\_OHCI

help

Enable support for the Samsung S5P SOC's on-chip OHCI controller.

#Add by Webee

config USB\_OHCI\_EXYNOS

boolean "OHCI support for Samsung EXYNOS SoC Series"

depends on USB\_OHCI\_HCD && ARCH\_EXYNOS

help

Enable support for the Samsung Exynos SOC's on-chip OHCI controller.

###### 二、移植 arch\arm\plat-samsung 目录下的 Kconfig

打开 arch\arm\plat-samsung 目录下的 Kconfig，在 S5P\_DEV\_USB\_EHCI后面添加 S5P\_DEV\_USB\_OHCI 的配置支持，修改后如下：

config S5P\_DEV\_USB\_EHCI

bool

help

Compile in platform device definition for USB EHCI

#Add by Webee

config S5P\_DEV\_USB\_OHCI

bool

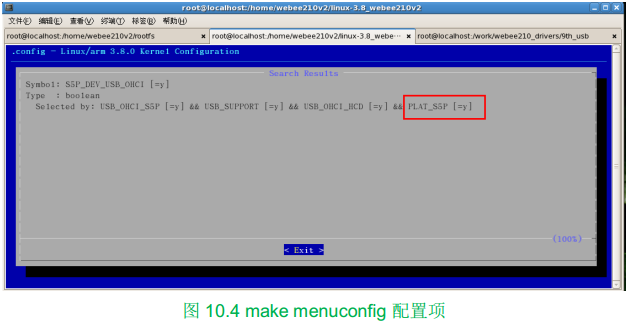
help

Compile in platform device definition for USB OHCI

#Add by Webee

###### 三、移植 drivers\usb\目录下的 Kconfig

在 内 核 目 录 下 输 入 make menuconfig 配 置 内 核 时 ， 搜 索 S5P\_DEV\_USB\_OHCI 发现如下现象，它表明 S5P\_DEV\_USB\_OHCI 的配置需要先将 PLAT\_S5P 配置上。



打开 drivers\usb\目录下的 Kconfig，在 USB\_ARCH\_HAS\_OHCI 模块下添加如下内容：

default y if PLAT\_S5P

##### 创建 setup-usb-phy.c

文件在 arch\arm\mach-s5pv210\目录下创建 setup-usb-phy.c 文件，为什么要创建这么一个文件呢？还记得前面在 smdkv210\_machine\_init()函数里添加过s5p\_ohci\_set\_platdata(&s5p\_ohci\_platdata);这个函数吗？

void \_\_init s5p\_ohci\_set\_platdata(struct s5p\_ohci\_platdata \*pd)

{

struct s5p\_ohci\_platdata \*npd;

npd = s3c\_set\_platdata(pd, sizeof(struct s5p\_ohci\_platdata),

&s5p\_device\_ohci);

if (!npd->phy\_init)

npd->phy\_init = s5p\_usb\_phy\_init;

if (!npd->phy\_exit)

npd->phy\_exit = s5p\_usb\_phy\_exit;

}

其中就会去设置 s5p\_ohci\_platdata 里的 phy\_init、 phy\_exit这两个成员函数。那么就需要实现， s5p\_usb\_phy\_init 函数和 s5p\_usb\_phy\_exit 函数。 最后将setup-usb-phy.c 文件添加如下代码：

#include <linux/platform\_device.h>

#include <linux/clk.h>

#include <linux/err.h>

#include <linux/io.h>

#include <linux/delay.h>

#include <mach/regs-clock.h>

#include <mach/gpio.h>

#include <mach/regs-sys.h>

#include <plat/regs-usb-hsotg-phy.h>

#include <plat/usb-phy.h>

#include <plat/clock.h>

#include <plat/gpio-cfg.h>

int s5p\_usb\_phy\_init(struct platform\_device \*pdev, int type)

{

int err;

struct clk \*otg\_clk;

if (type != S5P\_USB\_PHY\_HOST)

return -EINVAL;

otg\_clk = clk\_get(&pdev->dev, "otg");

if (IS\_ERR(otg\_clk)) {

dev\_err(&pdev->dev, "Failed to get otg clock\n");

return PTR\_ERR(otg\_clk);

}

err = clk\_enable(otg\_clk);

if (err) {

clk\_put(otg\_clk);

return err;

}

if (readl(S5PV210\_USB\_PHY\_CON) & (0x1<<1)) {

clk\_disable(otg\_clk);

clk\_put(otg\_clk);

return 0;

}

\_\_raw\_writel(\_\_raw\_readl(S5PV210\_USB\_PHY\_CON) | (0x1<<1),S5PV210\_USB\_PHY\_CON);

\_\_raw\_writel((\_\_raw\_readl(S3C\_PHYPWR)& ~(0x1<<7) & ~(0x1<<6)) | (0x1<<8) | (0x1<<5) | (0x1<<4),S3C\_PHYPWR);

\_\_raw\_writel((\_\_raw\_readl(S3C\_PHYCLK) & ~(0x1<<7)) | (0x3<<0),S3C\_PHYCLK);

\_\_raw\_writel((\_\_raw\_readl(S3C\_RSTCON)) | (0x1<<4) | (0x1<<3),S3C\_RSTCON);

\_\_raw\_writel(\_\_raw\_readl(S3C\_RSTCON) & ~(0x1<<4) & ~(0x1<<3),S3C\_RSTCON);

/\* "at least 10uS" for PHY reset elsewhere, 20 not enough here... \*/

udelay(50);

clk\_disable(otg\_clk);

clk\_put(otg\_clk);

return 0;

}

int s5p\_usb\_phy\_exit(struct platform\_device \*pdev, int type)

{

if (type != S5P\_USB\_PHY\_HOST)

return -EINVAL;

\_\_raw\_writel(\_\_raw\_readl(S3C\_PHYPWR) | (0x1<<7)|(0x1<<6),S3C\_PHYPWR);

\_\_raw\_writel(\_\_raw\_readl(S5PV210\_USB\_PHY\_CON) & ~(1<<1),S5PV210\_USB\_PHY\_CON);

return 0;

}

主要工作是获取 otg 时钟，设置 USB 相关的寄存器，具体为什么这么设置，Webee 也没有深入了解过，这是在外老网站上搜索到的代码。 有兴趣的小伙伴们，自己打开手册研究研究。

##### make menuconfig

配置内核选项前面做了那么多工作，为谁而做？为配置选项而做！就好比，你读了那么多书，为谁而读？你别跟我扯，说什么为祖国而读，为中华的崛起而读书？在内核目录下输入 make menuconfig，进入内核配置菜单栏，配置如下：

Device Drivers ——>

SCSI device support ——>

<\*> SCSI device support

[ \* ] legacy /proc/scsi/ support

<\*> SCSI disk support

<\*> SCSI CDROM support

<\*> SCSI generic support

HID support ——>

<\*> Generic HID driver

[ \* ] USB support ——>

<\*> Support for Host-side USB

<\*> OHCI HCD support

[ \* ] OHCI support for Samsung S5P SoC Series

[ \* ] Generic OHCI driver for a platform device

<\*> USB mass Storage support

[ \* ] USB mass Storage verbose debug

完成以上所有移植后，就可以编译内核了，在内核目录下使用 make uImage命令编译内核。如果编译过程中出现说获取不到 usbhost 时钟，那么将 ohci-s5p.c里的 s5p\_ohci\_probe()函数里的：

s5p\_ohci->clk = devm\_clk\_get(&pdev->dev, "usbhost");

改为：

s5p\_ohci->clk = devm\_clk\_get(&pdev->dev, "usb-host");

#### S5PV210 主机控制器驱动的测试

ohci\_hcd: USB 1.1 'Open' Host Controller (OHCI) Driver

s5p-ohci s5p-ohci: S5PV210 OHCI Host Controller

s5p-ohci s5p-ohci: new USB bus registered, assigned bus number 1

s5p-ohci s5p-ohci: irq 87, io mem 0xec300000

hub 1 -0:1.0: USB hub found

hub 1 -0:1.0: 1 port detected

Initializing USB Mass Storage driver...

usbcore: registered new interface driver usb-storage

USB Mass Storage support registered.

mousedev: PS/2 mouse device common for all mice

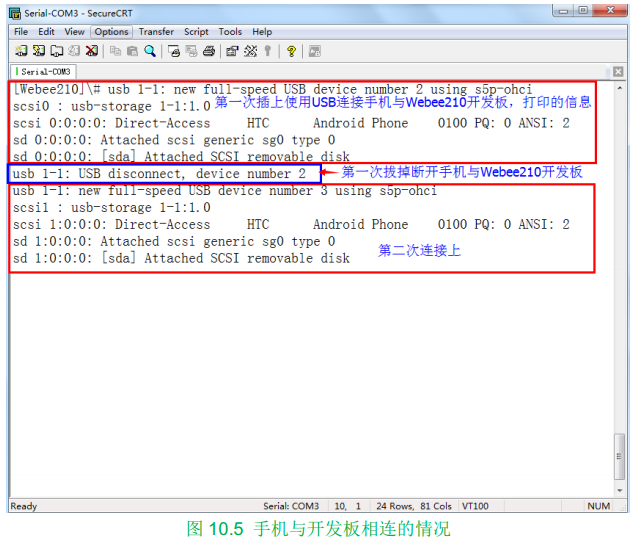
i2c /dev entries driver

usbcore: registered new interface driver usbhid

usbhid: USB HID core driver

#### 图说 OHCI 驱动之手机测试

使用 USB 线将手机与 webee210 开发板连接起来，观察打印信息。

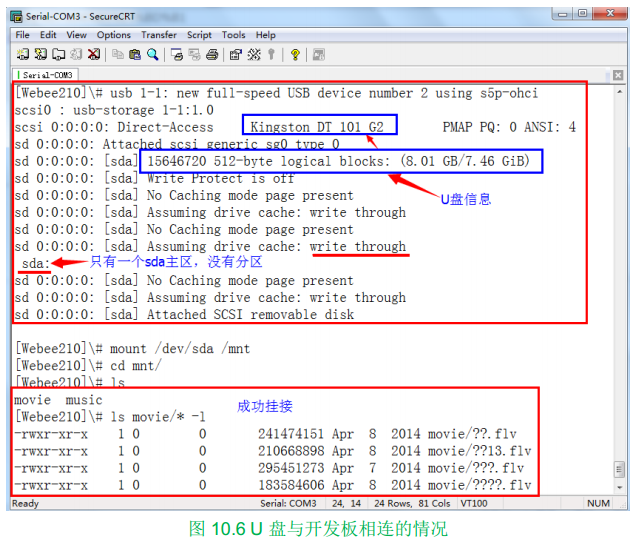


移植 OHCI 驱动前，无论你插入神马东西，它都不会吐任何一个字，没有一点点反应，这是因为之前的内核还不支持 USB HOST 驱动。

移植 OHCI 驱动后，插入手机，立马弹出图 10.5 的信息。 里面信息告诉我们， webee210 开发板知道你插入的是什么 USB 设备，并且帮你找到相应的 USB  
设备驱动程序并安装上。当你拔掉手机后，它弹出 disconnect 的信息，知道你已经将手机拔掉了。这就是牛逼哄哄的热插拔功能。 有些“后生仔”可能不认为这是一件很牛逼的事情，因为它没见识过，在热插拔技术未诞生之前，不能热插拔是多么痛苦的一件事。 我就不扯开了，自己去想象吧。

#### 图说 OHCI 驱动之 U 盘测试

将 U 盘(注意，不要插 USB3.0 的 U 盘)插入 webee210 开发板的 USB HOST接口，观察打印信息(不同 U 盘信息可能不完全一致)：



插入 U 盘的瞬间，就输出那么多的信息。从信息里，我们可以知道， webee210开发板已经识别出你插入的是 U 盘，并且知道你的 U 盘的容量有多大等等信息。那我们接下来测试一下 U 盘能不能读写呗~~先挂接 U 盘，然后进入 mnt 目录  
[Webee210]\# mount /dev/sda /mnt  
[Webee210]\# cd /mnt

然后就可以查看到 U 盘里面的文件数据了，我的 U 盘里只有二个文件夹，movie 和 music。至于是什么 movie，尽情的发挥你们的想象吧，骚年。  
接下来我们来验证一下 U 盘能不能被读写。先创建一个 temp.txt 文件:  
[Webee210]\# touch temp.txt

然后往 temp.txt 里写数据:  
/\* 注意了，使用 echo 命令，输入的内容不能有空格 \*/  
[Webee210]\# echo hello,USB > temp.txt

最后检查一下内容写进去没，即检查可不可以被读：

### USB 核心

USB 核心负责 USB 驱动管理和协议处理工作，它通过定义一些数据结构、宏和功能函数，向上为 USB 设备驱动提供编程接口， 向下为 USB 主机控制器驱动提供编程接口；通过全局变量维护整个系统的 USB 设备信息，完成设备热插拔控制、总线数据传输控制等。

drivers\usb\目录下，有个目录叫 core 目录， 进去之后，发现里面有 22 个文件(如图 10.11 )，

#### hub.c

细心的读者可能发现了 10.3.2 节中，手机、 U 盘、鼠标的测试的第一句打印信息都是相似的。如：

/\* 手机 \*/

usb 1 -1:new full-speed USB device number 2 using s5p-ohci  
/\* U 盘 \*/

usb 1 -1:new full-speed USB device number 2 using s5p-ohci  
/\* 鼠标 \*/

usb 1 -1:new low-speed USB device number 2 using s5p-ohci

发现 3 种不同的 USB 设备，打印的信息都是极其类似的。那么，我们推测，这句应该是通用的。既然是通用的，应该是核心做的事情，核心嘛，就是想多管闲事，管管别人。 在 Source Insight 里搜索” USB device number”就会找到 hub.c文件里的 hub\_port\_init 函数里有那么一句：

if (udev->speed != USB\_SPEED\_SUPER)  
dev\_info(&udev->dev,"%s %s USB device number %d using %s\n",(udev->config) ? "reset" : "new", speed,devnum, udev->bus->controller->driver->name);

#### USB 核心的事件处理流程

假如我们想弄清楚， USB 核心是怎么处理 USB 设备从插入到拔出， USB 核心做了什么工作，工作流程是怎样的？USB 设备都是通过插入上层 HUB 的一个 Port来连入系统并进而被系统发现的, 当 USB 设备插入一个 HUB 时,该 HUB 的那个 port 的状态就会改变, 从而系统就会知道这个改变, 此时会调用 **hub\_port\_connect\_change**()函数。 这个函数在 drivers/usb/core/hub.c

subsys\_initcall(usb\_init); //drivers\usb\core\usb.c

usb\_init //drivers\usb\core\usb.c

usb\_hub\_init

kthread\_run(hub\_thread, NULL, "khubd");

hub\_thread

hub\_events

hub\_port\_connect\_change

hub\_port\_connect\_change 函数主要做了二件事：

hub\_port\_connect\_change

hub\_port\_init //复位设备，分配地址，获取设备描述符

usb\_new\_device(udev);

我们再来看看 **hub\_port\_init** 函数主要做了什么事？

hub\_port\_init

/\* 给 USB 设备分配地址 \*/

hub\_set\_address(udev, devnum);

/\* 获取 USB 设备描述符 \*/

usb\_get\_device\_descriptor(udev, 8);

usb\_get\_device\_descriptor(udev, USB\_DT\_DEVICE\_SIZE);

继续看看 **usb\_new\_device** 函数主要做了什么？

usb\_new\_device(udev);

usb\_enumerate\_device(udev); /\* Read descriptors \*/

usb\_get\_configuration(udev); /\* 获取设备描述符 \*/

usb\_parse\_configuration /\* 解析设备描述符 \*/

device\_add(&udev->dev);

**device\_add 函**数在 drivers/base/core.c 里实现。

device\_add(&udev->dev);

get\_device(dev);

bus\_add\_device(dev);

bus\_probe\_device(dev);

device\_attach(dev);

/\*从总线上已注册的所有驱动中找出匹配的驱动程序.\*/

bus\_for\_each\_drv(dev->bus, NULL, dev, \_\_device\_attach);

while ((drv = next\_driver(&i)) && !error)

error = fn(drv, data);

**bus\_for\_each\_drv(dev->bus, NULL, dev, \_\_device\_attach)函数遍历 bus 上的所有驱动程序 , 并为每个驱动调用 fn()来查看是否匹配 . 这里的 fn 就是\_\_device\_attach.**

\_\_device\_attach(struct device\_driver \*drv, void \*data)

driver\_probe\_device(drv, dev);

really\_probe(dev, drv);

dev->driver = drv;

else if (drv->probe)

ret = drv->probe(dev);

**总结一下， device\_add 函数把 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。**

#### 设备驱动之鼠标驱动

USB 设备驱动指的是从主机角度看到的，怎样访问被插入的 USB 设备。Linux 发展到今天， USB 设备驱动已经相当丰富，像 U 盘、 USB 鼠标、 USB 键盘几乎都不用驱动工程师再编写了，一般由芯片厂家提供。并且 USB 设备驱动在 Linux 里有大量的参考例子，只需要稍微修改一下就可以用了。 Linux3.8 下的usb 鼠标驱动在/drivers/hid/usbhid/usbmouse.c 中实现，这应该是一个通用的USB 鼠标驱动。

##### 四个重要的结构体

我们说过了 Linux 内核里，驱动程序几乎都是面向对象的编程。既然是面向对象，那么肯定有需要定义、设置、注册一个结构体。 USB 设备驱动也不例外，在 Linux 内核里，使用 usb\_driver 结构体描述一个 USB 设备驱动。

###### USB 设备驱动(usb\_driver)

usb\_driver 结构体主要的成员“菜单”， 客官， 请慢用^\_^……

struct usb\_driver {

/\* 驱动名字 \*/

const char \*name;

/\* 探测函数， USB 设备被插入的时候被调用 \*/

int (\*probe) (struct usb\_interface \*intf,const struct usb\_device\_id \*id);

/\* 断开函数， USB 设备被拔出的时候被调用 \*/

void (\*disconnect) (struct usb\_interface \*intf);

/\* id\_table，描述了这个 USB 驱动所支持的 USB 设备列表 \*/

const struct usb\_device\_id \*id\_table;

......

};

###### USB 设备匹配列表(usb\_device\_id)

/\* 参考 include\linux\Mod\_devicetable.h \*/

struct usb\_device\_id {

\_\_u16 match\_flags; /\* 标明要与哪些成员匹配 \*/

/\* Used for product specific matches; range is inclusive \*/

\_\_u16 idVendor;

\_\_u16 idProduct;

\_\_u16 bcdDevice\_lo;

\_\_u16 bcdDevice\_hi;

/\* Used for device class matches \*/

\_\_u8 bDeviceClass;

\_\_u8 bDeviceSubClass;

\_\_u8 bDeviceProtocol;

/\* Used for interface class matches \*/

\_\_u8 bInterfaceClass;

\_\_u8 bInterfaceSubClass;

\_\_u8 bInterfaceProtocol;

/\* Used for vendor-specific interface matches \*/

\_\_u8 bInterfaceNumber;

/\* not matched against \*/

kernel\_ulong\_t driver\_info

\_\_attribute\_\_((aligned(sizeof(kernel\_ulong\_t))));

};

一般在 USB 设备驱动程序里，使用下面几个宏来生成 usb\_device\_id 结构体的实例。

/\* 参考 include\linux\usb.h \*/

USB\_DEVICE(vend, prod)

USB\_DEVICE\_VER(vend, prod, lo, hi)

USB\_DEVICE\_INTERFACE\_CLASS(vend, prod, cl)

USB\_DEVICE\_INTERFACE\_PROTOCOL(vend, prod, pr)

USB\_DEVICE\_INTERFACE\_NUMBER(vend, prod, num)

USB\_DEVICE\_INFO(cl, sc, pr)

USB\_INTERFACE\_INFO(cl, sc, pr)

USB\_DEVICE\_AND\_INTERFACE\_INFO(vend, prod, cl, sc, pr)

USB\_VENDOR\_AND\_INTERFACE\_INFO(vend, cl, sc, pr)

关于这 9 个宏如何使用，请参考 include\linux\usb.h 源码。这里仅仅举个例子，比如在 usb\_device\_id 数组实例中，使用 USB\_DEVICE(vend, prod)这个宏。表示，这个 USB 设备驱动所支持的制造商 ID、产品 ID 应该与 usb\_device\_id数组实例中的 idVendor、 idProduct 匹配。

当 USB 核心检测到插入的 USB 设备的属性与某个 USB 设备驱动的usb\_device\_id 结构体实例所携带的信息一致时，这个驱动程序的 probe()函数就会被调用。同理，当 USB 核心检测到 USB 设备被拔出时， disconnect()函数就来响应这个动作。所以， USB 设备驱动其实主要就是完成 probe() 和disconnect()函数，至于你在这两个函数里做什么事情，这个你决定。只打印一句话，都可以。 USB 总线已经完成它的本分工作了，剩下的就是 USB 本身所属类型的本分工作了。比如，你是一个 USB 串口，那么你就完成 tty 设备该完成的工作；你是一个字符设备，那么就完成字符设备该完成的工作；而 USB 鼠标是一个输入设备，那么很自然的，它就应该完成输入子系统那套工作。

###### USB 请求块(urb)

USB 请求块(USB request block,urb)是 USB 设备驱动中用来描述 USB 设备通信所用的核心数据结构。 urb 结构体的成员实在是太多了，但是 webee 发现在很多 USB 设备驱动里，很多成员都是没有使用到的，也就是说很多成员使用率并不高。 urb 的成员，大多数使用 urb 封装函数来设置的。后面程序里会提及，这里暂且先跟大家说一下。

struct urb {

/\* private: usb core and host controller only fields in the urb \*/

struct kref kref; /\* URB 引用计数 \*/

void \*hcpriv; /\* host 控制器的私有数据 \*/

atomic\_t use\_count; /\* 原子变量，用于并发传输计数 \*/

atomic\_t reject; /\* 原子变量，用于并发传输失败 \*/

int unlinked; /\* unlink 错误码 \*/

/\* public: documented fields in the urb that can be used by drivers \*/

struct list\_head urb\_list; /\* URB 链表头 \*/

struct list\_head anchor\_list;

struct usb\_device \*dev; /\* 内嵌的 USB 设备结构体 \*/

struct usb\_host\_endpoint \*ep; /\* (internal) pointer to endpoint \*/

unsigned int pipe; /\* 管道信息 \*/

int status; /\* URB 的当前状态 \*/

unsigned int transfer\_flags; /\* URB\_SHORT\_NOT\_OK | ...\*/

/\* 发送数据到 USB 设备，或从 USB 设备里接收数据到缓冲区 \*/

void \*transfer\_buffer;

dma\_addr\_t transfer\_dma; /\* 用 DMA 方式向设备传输数据到缓冲区 \*/

......

/\* transfer\_buffer 或 transfer\_dma 指向缓冲区的大小 \*/

u32 transfer\_buffer\_length;

u32 actual\_length; /\* 实际发送或接收的长度 \*/

......

};

###### USB 鼠标(usb\_mouse)

usb\_mouse 结构体代表一个 USB 鼠标设备，里面封装了 usb\_device，input\_dev 和 urb 三个重要的结构体。整个 USB 鼠标驱动就是围绕这个结构体来展开工作的。

struct usb\_mouse {

char name[128]; /\* 驱动名字 \*/

char phys[64]; /\* 设备节点 \*/

struct usb\_device \*usbdev; /\* 内嵌 USB 设备结构体 \*/

struct input\_dev \*dev; /\* 内嵌 INPUT 设备结构体 \*/

struct urb \*irq; /\* USB 请求块 \*/

signed char \*data; /\* transfer\_buffer 缓冲区 \*/

dma\_addr\_t data\_dma; /\* transfer \_dma 缓冲区 \*/

};

###### 模块加载宏

打开 drivers/hid/usbhid/usbmouse.c 后，发现并没有模块加载函数了，很奇怪。也没有看到类似于 module\_init(xxx\_init)、 module\_exit(xxx\_exit)的修饰语句了，哪去了呢？只看到下面这句：

module\_usb\_driver(usb\_mouse\_driver);

不知道从 Linux 的 3.0 以后的哪个版本开始，就喜欢用类似于上面这种方法来代替以前的 module\_init(xxx\_init)、 module\_exit(xxx\_exit)了。深入一点，你会发现它与以前的入口函数、出口函数的功能是一样的，只不过是封装了一下。

/\* 参考:include\linux\usb.h \*/

#define module\_usb\_driver(\_\_usb\_driver) \

module\_driver(\_\_usb\_driver, usb\_register, \

usb\_deregister)

#define module\_driver(\_\_driver, \_\_register, \_\_unregister, ...) \

static int \_\_init \_\_driver##\_init(void) \

{ \

return \_\_register(&(\_\_driver) , ##\_\_VA\_ARGS\_\_); \

} \

module\_init(\_\_driver##\_init); \

static void \_\_exit \_\_driver##\_exit(void) \

{ \

\_\_unregister(&(\_\_driver) , ##\_\_VA\_ARGS\_\_); \

} \

module\_exit(\_\_driver##\_exit);

有兴趣的读者，自行展开宏，这有点考验你 C 语言的功底囖。

###### usb\_mouse\_driver

static struct usb\_driver usb\_mouse\_driver = {

.name = "usbmouse", /\* 驱动名 \*/

.probe = usb\_mouse\_probe, /\* 匹配方法 \*/

.disconnect = usb\_mouse\_disconnect, /\* 拔出方法 \*/

.id\_table = usb\_mouse\_id\_table, /\* 支持设备 ID 列表 \*/

};

###### usb\_mouse\_id\_table

当插入鼠标时会根据 usb\_mouse\_id\_table 去匹配创建 usb 设备。

static struct usb\_device\_id usb\_mouse\_id\_table [] = {

{

USB\_INTERFACE\_INFO(USB\_INTERFACE\_CLASS\_HID,

USB\_INTERFACE\_SUBCLASS\_BOOT,

USB\_INTERFACE\_PROTOCOL\_MOUSE)

},

{ } /\* Terminating entry \*/

};

宏展开之后：

static struct usb\_device\_id usb\_mouse\_id\_table [] = {

{

.match\_flags = USB\_DEVICE\_ID\_MATCH\_INT\_INFO,

.bInterfaceClass = USB\_INTERFACE\_CLASS\_HID,

.bInterfaceSubClass = USB\_INTERFACE\_SUBCLASS\_BOOT,

.bInterfaceProtocol = USB\_INTERFACE\_PROTOCOL\_MOUSE

},

{ } /\* Terminating entry \*/

};

它表示这个 USB 鼠标驱动在判断支不支持 USB 鼠标时是根据接口信息来判断 的 。 如 果 USB 鼠 标 的 接 口 类 、 子 类 、 协 议 分 别 是USB\_INTERFACE\_CLASS\_HID、 USB\_INTERFACE\_SUBCLASS\_BOOTUSB\_INTERFACE\_PROTOCOL\_MOUSE 的话，那么这个 USB 鼠标驱动就支持这个 USB 鼠标。 usb 插入枚举时候会获取 usb 鼠标的接口类型,获取其接口类信息,匹配成功的话会动态创建一个 usb\_device.

###### usb\_mouse\_probe

匹配成功了就会调用 probe 方法，它是整个 USB 鼠标驱动的主体函数，主要围绕 usb\_mouse 结构体来展开工作。

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

{

/\* 由接口获取 usb\_device \*/

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

struct usb\_host\_interface \*interface;

struct usb\_endpoint\_descriptor \*endpoint;

struct usb\_mouse \*mouse;

struct input\_dev \*input\_dev;

int pipe, maxp;

int error = -ENOMEM;

/\* 由 usb\_interface 实例获取 usb\_host\_interface 实例 \*/

interface = intf->cur\_altsetting;

/\* 鼠标端点(Endpoint)只有 1 个 \*/

if (interface->desc.bNumEndpoints != 1)

return -ENODEV;

/\* 获取端点描述符，这里并不是获取端点 0 的描述符 \*/

endpoint = &interface->endpoint[0].desc;

/\* 检查该端点是否是中断输入端点,根据 HID 规范，

\* 鼠标唯一的端点应为中断端点

\*/

if (!usb\_endpoint\_is\_int\_in(endpoint))

return -ENODEV;

/\* 产生中断管道，驱动程序 buffer 和端点之间虚拟通道 \*/

pipe = usb\_rcvintpipe(dev, endpoint->bEndpointAddress);

/\* 返回该端点能够传输的最大的包长度，

\* 鼠标的返回的最大数据包为 4 个字节

\*/

maxp = usb\_maxpacket(dev, pipe, usb\_pipeout(pipe));

/\* 分配 usb\_mouse 结构体大小的内存 \*/

mouse = kzalloc(sizeof(struct usb\_mouse), GFP\_KERNEL);

/\* 鼠标是输入设备，所以要创建输入型设备 \*/

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

if (!mouse || !input\_dev)

goto fail1;

/\* 分配初始化 usb 鼠标数据缓冲区内存(默认 8 位数据) \*/

mouse->data = usb\_alloc\_coherent(dev, 8, GFP\_ATOMIC,&mouse->data\_dma);

if (!mouse->data)

goto fail1;

/\* 分配 URB，第一个参数为 0，表明中断传输，不设置等时包 \*/

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

if (!mouse->irq)

goto fail2;

mouse->usbdev = dev; /\* 填充 mouse 的 usb\_device 结构体 \*/

mouse->dev = input\_dev; /\* 填充 mouse 的 input 结构体 \*/

if (dev->manufacturer)

strlcpy(mouse->name, dev->manufacturer, sizeof(mouse->name));

if (dev->product) {

if (dev->manufacturer)

strlcat(mouse->name, " ", sizeof(mouse->name));

strlcat(mouse->name, dev->product, sizeof(mouse->name));

}

if (!strlen(mouse->name))

snprintf(mouse->name, sizeof(mouse->name),"USB HIDBP Mouse %04x:%04x",

le16\_to\_cpu(dev->descriptor.idVendor),

le16\_to\_cpu(dev->descriptor.idProduct));

/\* 设置设备节点 \*/

usb\_make\_path(dev, mouse->phys, sizeof(mouse->phys));

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

/\* 输入设备的名字设置成 usb 鼠标的名字 \*/

input\_dev->name = mouse->name;

/\* 输入设备的路径设置成 usb 鼠标的路径 \*/

input\_dev->phys = mouse->phys;

/\* 设置输入设备的 bustype,vendor,product,version \*/

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

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

/\* 设置鼠标能产生按键类和相对位移类事件 \*/

input\_dev->evbit[0] = BIT\_MASK(EV\_KEY) | BIT\_MASK(EV\_REL);

/\* 设置鼠标能产生按键类下的鼠标、左键、右键、中键 \*/

input\_dev->keybit[BIT\_WORD(BTN\_MOUSE)] = BIT\_MASK(BTN\_LEFT) | BIT\_MASK(BTN\_RIGHT) | BIT\_MASK(BTN\_MIDDLE);

/\* 设置鼠标能产生 X、 Y 方向的相对位移类事件 \*/

input\_dev->relbit[0] = BIT\_MASK(REL\_X) | BIT\_MASK(REL\_Y);

/\* 设置鼠标能产生相对位移类下的鼠标、旁键、外部键\*/

input\_dev->keybit[BIT\_WORD(BTN\_MOUSE)] |= BIT\_MASK(BTN\_SIDE) | BIT\_MASK(BTN\_EXTRA);

/\* 设置鼠标能产生滚轮的相对位移类事件 \*/

input\_dev->relbit[0] |= BIT\_MASK(REL\_WHEEL);

/\* usb 鼠标驱动文件作为输入设备的设备文件的驱动数据 \*/

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

/\* 设置输入事件的打开、关闭方法 \*/

input\_dev->open = usb\_mouse\_open;

input\_dev->close = usb\_mouse\_close;

/\* 填充中断类型 urb 指定了 urb 的回调函数是 usb\_mouse\_irq \*/

usb\_fill\_int\_urb(mouse->irq, dev, pipe, mouse->data,(maxp > 8 ? 8 : maxp),usb\_mouse\_irq, mouse, endpoint->bInterval);

/\* dma 数据缓冲区指向 usb 鼠标设备的 data\_dma 成员 \*/

mouse->irq->transfer\_dma = mouse->data\_dma;

/\* 无 DMA 映射 \*/

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

/\* 注册鼠标输入设备 \*/

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

if (error)

goto fail3;

/\* 相当于 intf->dev->p->driver\_data = mouse;

\* 将 usb\_mouse 实例放到 driver\_data,供其他函数使用,

\* 如:可通过 usb\_get\_intfdata (intf);取出 usb\_mouse 实例

\*/

usb\_set\_intfdata(intf, mouse);

return 0;

fail3:

usb\_free\_urb(mouse->irq);

fail2:

usb\_free\_coherent(dev, 8, mouse->data, mouse->data\_dma);

fail1:

input\_free\_device(input\_dev);

kfree(mouse);

return error;

}

###### usb\_mouse\_disconnect

拔掉 usb 鼠标就会调用 disconnect 方法，其实 disconnect 只是 probe 的一个逆操作而已。

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

{

/\* 通过接口获得 usb 鼠标设备 \*/

struct usb\_mouse \*mouse = usb\_get\_intfdata (intf);

/\* intf->dev = NULL \*/

usb\_set\_intfdata(intf, NULL);

if (mouse) {

usb\_kill\_urb(mouse->irq); /\* 取消已经提交的 urb \*/

input\_unregister\_device(mouse->dev); /\* 注销输入设备 \*/

usb\_free\_urb(mouse->irq); /\* 释放已经分配的 urb \*/

/\* 清除传输数据缓冲区 \*/

usb\_free\_coherent(interface\_to\_usbdev(intf), 8, mouse->data,

mouse->data\_dma);

kfree(mouse); /\* 释放 usb 鼠标设备 \*/

}

}

###### usb\_mouse\_open

经过 probe 过程,注册了输入设备则会在/dev/input/目录下会产生对应的鼠标设备节点 , 应用程序可以打开该节点来控制 usb 鼠标设备，此时会调用usb\_mouse\_open 方法

static int usb\_mouse\_open(struct input\_dev \*dev)

{

/\* 通过输入设备获取 usb 鼠标设备 \*/

struct usb\_mouse \*mouse = input\_get\_drvdata(dev);

/\* 设置 urb 设备对应的 usb 设备 \*/

mouse->irq->dev = mouse->usbdev;

/\* 提交 urb,通过 urb 提交之后,鼠标动作

\* 通过 usb 传输数据就会交由 urb 去处理了

\*/

if (usb\_submit\_urb(mouse->irq, GFP\_KERNEL))

return -EIO;

return 0;

}

###### usb\_mouse\_irq

当 操 作 鼠 标 的 时 候 , 会 引 起 urb 数 据 传 输 在 数 据 传 输 之 后 会 调 用usb\_mouse\_irq，注意了，它并不是一个真正的中断处理函数。我们说过了， USB设备没有中断主机控制器的能力，它只能轮询。

static void usb\_mouse\_irq(struct urb \*urb)

{

/\* 获取 usb 鼠标设备 \*/

struct usb\_mouse \*mouse = urb->context;

/\* transfer\_buffer 缓冲区 \*/

signed char \*data = mouse->data;

struct input\_dev \*dev = mouse->dev;

int status;

/\* 通过 urb->status 判断 URB 传输是否成功，

\* 如果 urb->status = 0，则正确继续执行，错误时不会执行

\*/

switch (urb->status) {

case 0: /\* success \*/

break;

case -ECONNRESET: /\* unlink \*/

case -ENOENT:

case -ESHUTDOWN:

return;

/\* -EPIPE: should clear the halt \*/

default: /\* error \*/

goto resubmit;

}

/\* 正确时报告鼠标按键情况，使用 input\_report\_key()

\* 报告信息，使用 input\_sync 表明报告完毕。

\*/

input\_report\_key(dev, BTN\_LEFT, data[0] & 0x01);

input\_report\_key(dev, BTN\_RIGHT, data[0] & 0x02);

input\_report\_key(dev, BTN\_MIDDLE, data[0] & 0x04);

input\_report\_key(dev, BTN\_SIDE, data[0] & 0x08);

input\_report\_key(dev, BTN\_EXTRA, data[0] & 0x10);

input\_report\_rel(dev, REL\_X, data[1]);

input\_report\_rel(dev, REL\_Y, data[2]);

input\_report\_rel(dev, REL\_WHEEL, data[3]);

input\_sync(dev);

resubmit:

/\* 继续提交 urb \*/

status = usb\_submit\_urb (urb, GFP\_ATOMIC);

if (status)

dev\_err(&mouse->usbdev->dev,

"can't resubmit intr, %s-%s/input0, status %d\n",

mouse->usbdev->bus->bus\_name,

mouse->usbdev->devpath, status);

}

usb 接口传来的数据会保存在 usb 鼠标 data 指针成员指向的缓冲区中， 这里可以看出 usb 鼠标传输的每次数据基本是 4 个字节。

第 0 个字节的第 1 位表示右键,第 2 位表示左键,第 3 位表示中键,第 4 位表示边键,第 5 为表示外部键；而第 1 个字节表示相对 x 坐标的位移,第 2 个字节表示相对 y 坐标的位移,第 3 个字节表示相对滚轮的位移。

当输入设备上报完 usb 接口接收来的数据后,需要调用 input\_sync 同步事件消息,并调用 usb\_submit\_urb 提交 urb，使其继续监视处理 usb 鼠标设备传递的新数据。