第 32 卷 第 19 期 计 算 机 工 程 2006 年 10 月
Vol.32 № 19 Computer Engineering October 2006
·开发研究与设计技术· 文章编号：1000—3428(2006)19—0283—03 文献标识码：A 中图分类号：TP316

Linux 下 USB 设备驱动研究与开发

杨 伟，刘 强，顾 新
(西安电子科技大学计算机学院，西安 710071)

摘 要：开发 USB(Universal Serial Bus)设备驱动是一项比较繁琐的工作。Linux 中的 USB 核心子系统提供了大量的 API 以及相关的支持机制，保证了 USB 设备的即插即用，简化了驱动的编写。结合具体开发实例，介绍了 USB 的相关概念，分析 Linux 中 USB 核心子系统的框架构成以及重要的数据结构，剖析 Linux 内核对 USB 规范的支持，描述了驱动开发的一般方法和技巧。

关键词：USB；Linux；设备驱动程序

Research and Design of USB Device Driver in Linux

YANG Wei, LIU Qiang, GU Xin
(Department of Computer, Xidian University, Xi`an 710071)

【Abstract】 Programming for USB device drivers is a complex job. The Linux USB core subsystem has offered a large amount of relevant mechanisms and application programming interface(APIs) to support the USB device works Plug-and-Play(PNP) and relax the job developing of USB device drivers. With a representative example, the paper introduces some related concepts of USB, analyses the pattern, data structures related to Linux USB subsystem. It emphasizes on how Linux support the protocol of USB and describes the general method and skill in development of Linux USB drivers.

【Key words】 Universal serial bus(USB); Linux; Device driver program

通用串行总线(USB)是一种外部总线结构，特点是接口统一、易于使用、方便扩展、支持热插拔(hot plug)和 PNP(Plug-and-Play)，简化了计算机与不同类型外设间的连接，一经推出就得到计算机外设硬件制造商的广泛采用。Linux 作为一个占有相当市场份额的开源操作系统，自从 2.2.18 版内核开始，就加入了对 USB 的支持，2.4.X 版本的内核对 USB1.1 的支持已相对完善，新推出的 2.6 内核已经率先支持了 USB2.0 规范。本文结合开发一个特定 USB 设备驱动的经验，分析了 Linux 中 USB 核心子系统的具体实现机制。

1 USB 系统的体系结构

USB 是一种分层总线结构，USB 设备和主机之间的信息传输通过 USB 控制器实现。图 1 给出了 Linux 中 USB 驱动的体系结构。Linux USB 主机驱动由 3 部分组成：USB 主机控制器驱动，USB 驱动和不同的 USB 设备类驱动。

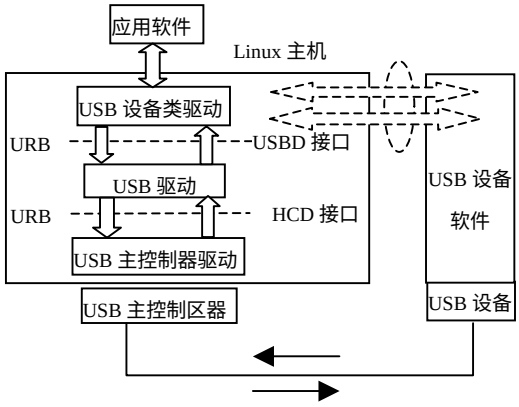

图 1 Linux USB 驱动体系结构

处于最底层 USB 主机控制器驱动(HCD)是 USB 主机直接与硬件交互的软件模块。Linux 2.4 内核中的 USB 支持 2 种主控制器接口：通用主控制器接口(UHCI)和开放控制器接口(OHCI)。主控制器驱动为上层提供统一的接口屏蔽掉硬件具体细节。具体实现的功能有主机控制器硬件初始化；为 USBD 层提供相应的接口函数；提供根集线器设备配置、控制功能；完成 4 种类型的传输。

USB 驱动(USBD)部分是整个 USB 主机驱动的核心，主要负责 USB 总线的管理、USB 总线设备、USB 总线带宽管理、为 USB 设备驱动提供相关的接口、提供应用程序访问的 USB 系统的文件接口。

USB 设备类驱动是最终与应用程序交互的软件模块，主要为访问特定的 USB 设备和应用程序提供接口。

2 Linux 中 USB 设备驱动的核心数据结构

在 Linux 的 USB 内核子系统中，提供了几个与设备驱动开发密切相关的数据结构，它们在实现 USB 的体系结构、完成 USB 设备的即插即用、构建驱动程序的框架方面都起着灵魂的作用。

2.1 struct usb_driver

struct usb_driver 是对 USB 设备的最高层次的抽象。这个结构用于向 USBD 层注册 USB 设备驱动程序。总线上有设备接入的时候，USBD 通过该结构查找相关的设备驱动程序，而对于断开操作，USBD 同样通过这个结构来断开设备与驱动的连接。与其它驱动结构最大的不同就在于它指向 probe 和 disconnect 的指针，当驱动正确安装以后，如果相应的 USB 设备接入主机，系统自动调用 probe 进行探测，设

作者简介：杨 伟(1983－)，女，硕士生，主研方向：嵌入式系统应用；刘 强，硕士生；顾 新，副教授

收稿日期：2005-10-16 **E-mail：**godliuqiang@etang.com

备卸载时 disconnect 函数被调用。这 2 个函数作为 USB 设备安装和卸载的入口点，保证了 USB 设备的即插即用。下面给出一个在系统中注册的 usb_driver 结构：

```
static struct usb_driver d12_usb_driver = {
  name : "d12usb",    //驱动名字
  probe : d12_usb_probe,    //探测函数进入指针
 disconnect    : d12_usb_disconnect,        //撤销函数进入指针
  fops : &d12_fops,      //设备的操作列表指针
  minor :192,    //次设备号必须为 16 的倍数
  id_table : &d12_usb_id,    //描述该驱动所适用设备信息的指针};
```

2.2 struct usb_device

struct usb_device 是对系统中所有 USB 设备的抽象，这个数据结构不仅包含了 USB 设备必要的硬件信息，而且据此可以了解到 USB 设备具体的配置信息。USB 标准中以层次化的结构定义设备的描述符。一个 USB 设备只能有一个设备描述符，内容直接由具体的硬件提供，但是却可以有多个配置描述符，每个配置描述符又由一个或多个接口描述符以及相应的端点描述符构成。Linux 中的 USB 核心子系统通过几个数据结构之间的对应关系充分体现了这种层次化的结构。具体如图 2 对比所示。

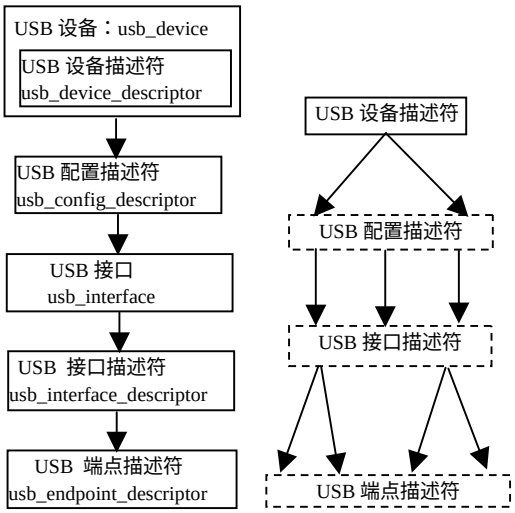

图 2 USB 描述符层次结构

2.3 struct urb USB 数据请求块

USB 信息传输包含 4 种传输类型：控制传输，等时传输，中断传输，成块传输。为了给设备驱动完成 USB 通信提供统一的接口，USB 设备驱动与 USBD，以及 USBD 与 HCD 之间的数据传输都是通过数据结构 URB 来实现的。因此 URB 中包含了数据传输在各个层次中的必要信息，下面列出结构中的部分定义，其中的 pipe 参数代表了进行传输的逻辑通道，直接反映了该 URB 所发生的数据端点，以及传输类型。

```
typedef struct urb{ …
struct usb_device *dev; //请求的设备
unsigned int pipe; //传输管道
int status; //返回状态
unsigned int transfer_flags; //传输标志
void *transfer_buffer; //传输数据
int transfer_buffer_length; //传输数据长度
int actual_length; //实际长度
int timeout; //超时期限
… }
```

对于这个结构的处理通常由 4 个函数来进行：

(1)purb_t usb_alloc_urb(int iso_packets) 为 USB 传输分配 URB 结构，参数 iso_packets 是用于等时传输的数据包数目，对于其他传输该参数为 0。

(2)void usb_free-urb(purb_t purb) 释放 URB 数据结构。

(3)int usb_submit_urb(purb_t purb) 用于向 USB 核心子系统提交一个异步请求。

(4)int usb_unlink_urb(purb_t purb) 在 URB 没有处理之前，通过这个函数取消相关的数据处理。

3 USB 内核子系统提供的接口 API

USBD 层为 USB 设备驱动提供了统一接口的 API，方便了用户自己编写适合的驱动程序。

3.1 注册和注销驱动程序

为了实现 USB 设备即插即用的特性，所有的 USB 设备驱动都要在 USBD 中注册或注销。具体由函数 usb_register 和 usb_deregister 来实现。当驱动程序调用 int usb_register(struct usb_driver *new_driver)时，系统将这个新的驱动程序加入驱动链表(usb_driver_list)的尾部完成注册，同时扫描总线上的所有设备，与驱动程序关联。注销函数的功能与此刚好相反，usb_deregister()关闭设备，撤销为设备在内核中分配的一切资源，然后把驱动结构从驱动链表中摘除。使用实例如下所示：

```
int init_module(void)
{ if(usb_register(&d12_usb_driver)<0){
    printk("unable to register device !\n");
    return -1; }
  return 0;    }
```

3.2 标准设备请求

USB 协议定义了一系列的标准设备请求用来完成 USB 设备的枚举、配置等操作。USB 设备以缺省的控制管道响应主机的请求。USBD 提供了部分接口函数支持 USB 标准请求，其余的控制传输由函数 usb_control_msg 完成。表 1 列出了 USB 设备请求与对应的接口函数。

表 1 标准设备请求接口

USB 标准设备请求	USBD 接口函数	说明
GET_STATUS	usb_get_status	读取设备状态
CLEAR_FEATURE	usb_clear_halt	
SET_FEATURE	-	
SET_ADDRESS	usb_set_address	设置设备地址
GET_DESCRIPTOR	usb_get_descriptor	读取描述字
	usb_get_string	读取字符串描述字
	usb_get_device_descriptor	读取设备描述字
SET_DESCRIPTOR	-	
GET_CONFIGURATION	usb_get_configuration	读取配置描述字
SET_CONFIGURATION	usb_set_configuration	USB 设备配置
GET_INTERFACE	-	
SET_INTERFACE	usb_set_interface	USB 接口设置
SYNCH_FRAME	-	

3.3 USB 数据传输

USBD 提供了 3 种用于数据传输的接口：usb_control_msg (用于控制传输)；usb_bulk_msg(用于批量传输)；直接提交 URB。但其实在 usb_control_msg()、usb_bulk_msg()内部，也是由提交 URB 来完成的，只不过填充请求的工作由系统来完成。以批量传输为例介绍 USB 设备读写模块的设计与具体实现。

```
static int usb_write(struct file * file, char * buf, size_t count, loff_t
*ppos)
    {    int this_write;
```

```
int partial;//实际写的长度
int result;//操作结果
if(device_plug_1==0){
  printk(“No device in!\n”);
  return -1;      }
while(count>0){
   this_write = (count >= OBUF_SIZE)?OBUFSIZE :count
   result=usb_bulk_msg(d12->dev, usb_sndbulkpipe(d12->
dev,2),d12_>obuf, this_write , HZ*TIMEOUT);
   …//根据 result 的结果进行相应处理
  if(partial ! = thiswrite) {…}
  //判断是否成功写入，并进行相应的处理
  …}//end while
return 1；  }
```

用户调用标准接口 usb_bulk_msg 完成批量传输。填充好相应的 URB 结构后，进入下面的调用序列——usb_bulk_msg()>usb_start_wait_urb()>usb_submit_urb()>uhci_submit_urb()>uhci_submit_bulk()——USB 控制器把具体的请求转化为交互队列(假设 USB 驱动采用 UHCI 层面)根据总线时间、带宽、传输类型等综合因素调度队列，执行相应的总线动作与设备进行交互，完成通信任务。图 3 给出了 USB 设备与主机的通信模型。

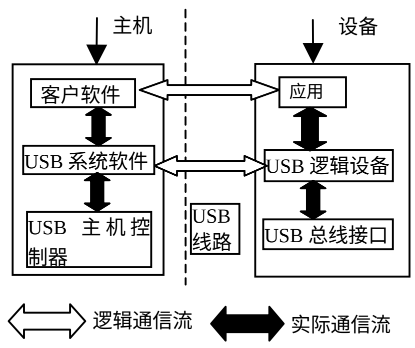

图 3 USB 设备通信模型

3.4 USB 设备的安装和卸载

USB 设备之所以可以即插即用，就是因为 USB 内核子系统即时对总线反馈的设备状态做出相应的处理。相应地在设备驱动方面也要提供 probe 和 disconnect 函数对设备进行初始化和断开操作。

USB 设备接入主机后，主机通过总线枚举来识别和管理必要的设备状态变化。首先检测设备在总线上的上拉电阻判断是否有新设备连接，并获悉该设备是全速设备还是低速设备，然后主机向该端口发送一个复位信号。设备只有在收到这个复位信号后，才能对总线的处理做出响应。此时设备采用的是默认地址(00H)，主机接收到设备对默认地址(00H)的响应，就为设备分配一个空闲的地址，以后设备就只对这个地址进行响应。接下来，主机读取 USB 设备描述符，确认 USB 设备的属性。根据读取的设备描述符对设备进行配置，设备如果可以达到要求就发送配置完毕的信号给主机。然后调用设备驱动的 probe 函数对设备进行初始化。下面给出一个简单的 probe 函数的写法。

```
static void *naked_usb_probe(struct usb_device *dev)
{//检查硬件定义与驱动是否相符
if(dev->descriptor.idVendor==0x0471 && dev->descriptor.
idProduct == 0x0222)    {    device_plug_in = 1;//设备插入标志
//初始化设备相关资源
if(d12->obuf=(char*)kmalloc(OBUF_SIZE,GFP_KERNEL)){…}
d12->dev=dev;
…
printk("My usb device pluged in.\n");
return ;    }    }
```

USB 设备卸载的时候，将释放 HCD 层为该设备分配的资源，调用设备驱动的 disconnect 函数断开与设备的连接。

3.5 USB 驱动的使用

在 2.4 内核中，虽然引进了 devfs 设备文件系统，但是对设备的管理主要还是采用主、次设备号的方法。为了使设备在系统中可以被应用程序访问，必须建立一个设备文件节点。可以通过“ mknod /dev/usb d12usb c 180 192 ”命令添加自己设备的节点，相对地在应用程序中通过系统调用 open(“/dev/usb/d12usb”,O_RDWR)打开设备，之后就可以根据需要对设备进行读写操作。

4 小结

新推出的 Linux 2.6 加入了对 USB2.0 的支持，重新定义了 struct usb_class_driver 结构取代了 2.4 中的 struct usb_driver 结构，同时对函数 probe 以及 usb_submit_urb 做了修订。包含了 ALSA(Advanced Linux Sound Architecture)，可以更加安全地使用 USB 设备。

参考文献

1 毛德操，胡希明. Linux 内核源代码情景分析[M]. 杭州: 浙江大学出版社, 2003.
2 Universal Serial Bus Specification Revision 1.1[Z]. Compaq, IBM, Intel, Microsoft, NEC, Northern Telecom, 1995.

~~~~~~

（上接第 274 页）

2.3 系统的安全技术

信息在网络上传输的时候为了防止信息被泄漏和修改，我们采用 DES 和 RSA 加密算法的混合加密算法对数据包进行加密(详细实现技术略)。

3 前景展望

当前的系统在青海省公安厅居民身份证制作中心进行了调试和试运行。系统体现出了良好的性能和优势。该系统提高了公安业务的处理速度，为公安工作的进一步扩大化奠定了基础，也为其他少数民族身份证制证工作提供了一个新的思路。

参考文献

1 周东平. 第二代居民身份证集成电路设计关键技术研究[J]. 信息技术与应用, 2003, (10): 35-37.
2 樊　胜. CS 与 BS 的结构比较及 Web 数据库的访问方式[J]. 情报科学, 2001, 19(4): 443-445.
3 肖宏伟. Visual C++ 6.0 实效编程百例[M]. 北京: 人民邮电出版社, 2002.
4 公安部科技局. GB/T2261—1980 公安信息化标准汇编──国家标准部分 GB1: 人的性别代码中国各民族名称的罗马字母拼写法和代码 GB/T3304—1991[S]. 北京: 中国标准出版社, 2001.
~~~~~~