Linux 环境下 USB 的原理、驱动和配置

作者：梁国军

关键词：USB 设备，Linux，驱动

摘要：本文对 Linux 环境下 USB 的原理、驱动和配置进行详细介绍。

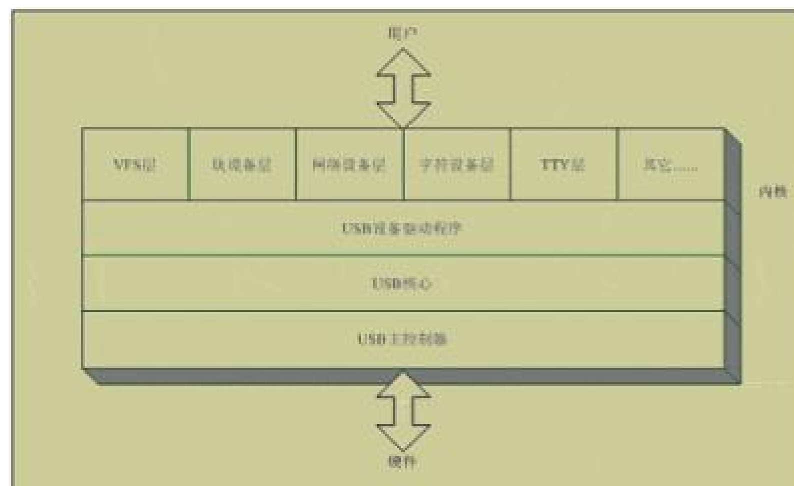
随着生活水平的提高，人们对 USB 设备的使用也越来越多，鉴于 Linux 在硬件配置上尚不能全部即插即用，因此关于 Linux 如何配置和使用，成为困扰我们的一大问题。

什么是 USB？

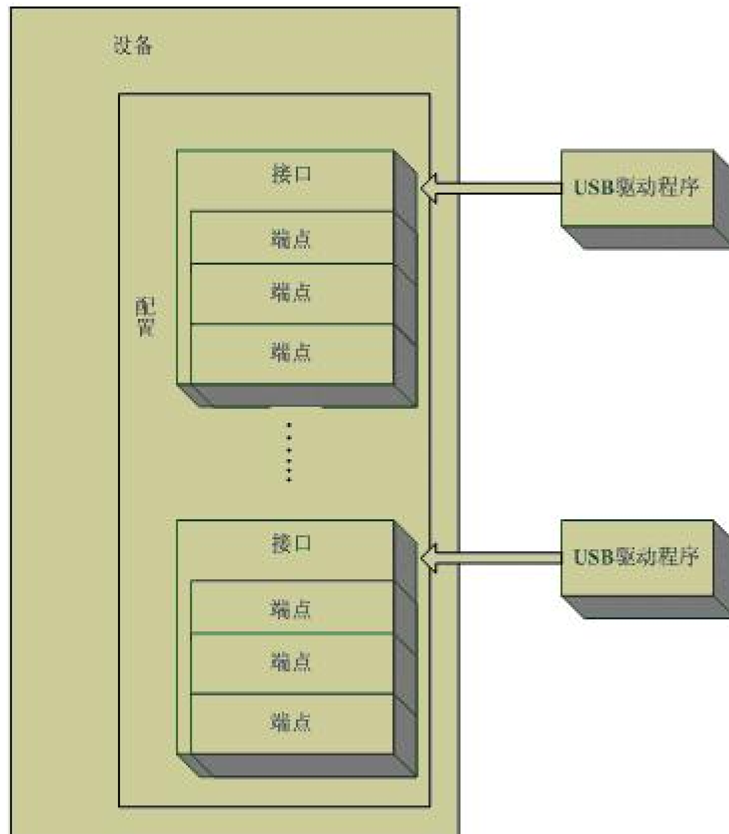
USB 是英文 Universal Serial Bus 的缩写，意为通用串行总线。USB 最初是为了替代许多不同的低速总线（包括并行、串行和键盘连接）而设计的，它以单一类型的总线连接各种不同的类型的设备。USB 的发展已经超越了这些低速的连接方式，它现在可以支持几乎所有可以连接到 PC 上的设备。最新的 USB 规范修订了理论上高达 480Mbps 的高速连接。Linux 内核支持两种主要类型的 USB 驱动程序：宿主系统上的驱动程序和设备上的驱动程序，从宿主的观点来看（一个普通的宿主也就是一个 PC 机），宿主系统的 USB 设备驱动程序控制插入其中的 USB 设备，而 USB 设备的驱动程序控制该设备如何作为一个 USB 设备和主机通信。

USB 的具体构成

在动手写 USB 驱动程序这前，让我们先看看写的 USB 驱动程序在内核中的结构，如下图：



USB 驱动程序存在于不同的内核子系统和 USB 硬件控制器之间，USB 核心为 USB 驱动程序提供了一个用于访问和控制 USB 硬件的接口，而不必考虑系统当前存在的各种不同类型的 USB 硬件控制器。USB 是一个非常复杂的设备，linux 内核为我们提供了一个称为 USB 的核心的子系统来处理大部分的复杂性，USB 设备包括配置(configuration)、接口（interface）和端点(endpoint)，USB 设备绑定到接口上，而不是整个 USB 设备。如下图所示：



USB 通信最基本的形式是通过端点（USB 端点分中断、批量、等时、控制四种，每种用途不同），USB 端点只能往一 个方向传送数据，从主机到设备或者从设备到主机，端点可以看作是单向的管道（pipe）。所以我们可以这样认为：设备通常具有一个或者更多的配置，配置经常具有一个或者更多的接口，接口通常具有一个或者更多的设置，接口没有或具有一个以上的端点。驱动程序把驱动程序对象注册到 USB 子系统中，稍后再使用制造商和设备标识来判断是否已经安装了硬件。USB 核心使用一个列表（是一个包含制造商 ID 和设备号 ID 的一个结构体）来判断对于一个设备该使用哪一个驱动程序，热插拨脚本使用它来确定当一个特定的设备插入到系统时该自动装载哪一个驱动程序。

上面我们简要说明了驱动程序的基本理论，在写一个设备驱动程序之前，我们还要了解以下两个概念：模块和设备文件。

模块：是在内核空间运行的程序，实际上是一种目标对象文件，没有链接，不能独立运行，但是可以装载到系统中作为内核的一部分运行，从而可以动态扩充内核的功能。模块最主要的用处就是用来实现设备驱动程序。Linux 下对于一个硬件的驱动，可以有两种方式：直接加载到内核代码中，启动内核时就会驱动此硬件设备。另一种就是以模块方式，编译生成一个.ko 文件(在 2.4 以下内核中是用.o 作模块文件，我们以 2.6 的内核为准，以下同)。当应用程序需要时再加载到内核空间运行。所以我们所说的一个硬件的驱动程序，通常指的就是一个驱动模块。

设备文件：对于一个设备，它可以在/dev 下面存在一个对应的逻辑设备节点，这个节点以文件的形式存在，但它不是普通意义上的文件，它是设备文件，更确切的说，它是设备节点。这个节点是通过 mknod 命令建立的，其中指定了主设备号和次设备号。主设备号表明了某一类设备，一般对应着确定的驱动程序；次设备号一般是区分不同属性，例如不同的使用方法，不同的位置，不同的操作。这个设备号是从/proc/devices 文件中获得的，所以一般是先有驱动程序在内核中，才有设备节点在目录中。这个设备号（特指主设备号）的主要作用，就是声明设备所使用的驱动程序。驱动程序和设备号是一一对应的，当你打开一个设备文件时，操作系统就已经知道这个设备所对应的驱动程序。对于一个硬件，Linux 是这样来进行驱动的：首先，我们必须提供一个.ko 的驱动模块文件。我们要使用这个驱动程序，首先要加载它，我们可以用 insmod xxx.ko，这样驱动就会根据自己的类型（字符设备类型或块设备类型，例如鼠标就是字符设备而硬盘就是块设备）向系统注册，注册成功系统会反馈一个主设备号，这个主设备号就是系统对它的唯一标识。驱动就是根据此主设备号来创建一个一般放置在/dev 目录下的设备文件。在我们要访问此硬件时，就可以对设备文件通过 open、read、write、close 等命令进行。而驱动就会接收到相应的 read、write 操作而根据自己的模块中的相应函数进行操作了。

USB 驱动程序如何应用

了解了上述理论后，我们就可以动手写驱动程序，如果你基本功好，而且写过 Linux 下的硬件驱动，USB 的硬件驱动和 pci_driver 很类似，那么写 USB 的驱动就比较简单了，如果你只是大体了解了 linux 的硬件驱动，那也不要紧，因为在 linux 的内核源码中有一个框架程序可以拿来借用一下，这个框架程序在/usr/src/~（你的内核版本，以下同）/drivers/usb 下，文件名为 usb-skeleton.c。写一个 USB 的驱动程序最基本的要做四件事：驱动程序要支持的设备、注册 USB 驱动程序、探测和断开、提交和控制 urb（USB 请求块）（当然也可以不用 urb 来传输数据，下文我们会说到）。

驱动程序支持的设备：有一个结构体 struct usb_device_id，这个结构体提供了一列不同类型的该驱动程序支持的 USB 设备，对于一个只控制一个特定的 USB 设备的驱动程序来说，struct usb_device_id 表被定义为：

```
/*  驱动程序支持的设备列表  */

static struct usb_device_id skel_table [] = {

      { USB_DEVICE(USB_SKEL_VEN DOR_ID, USB_SKEL_PRODUCT_ID) },

      { }                       /*  终止入口  */

};

       MODULE_DEVICE_TABLE (usb, skel_table);
```

对于 PC 驱动程序，MODULE_DEVICE_TABLE 是必需的，而且 usb 必需为该宏的第一个值，而 USB_SKEL_VENDOR_ID 和 USB_SKEL_PRODUCT_ID 就是这个特殊设备的制造商和产品的 ID 了，我们在程序中把定义的值改为我们这款 USB 的，如：

```
/*  定义制造商和产品的 ID 号  */

#define USB_SKEL_VENDOR_ID       0x1234

#define USB_SKEL_PRODUCT_ID     0x2345
```

这两个值可以通过命令 lsusb，当然你得先把 USB 设备先插到主机上了。或者查看厂商的 USB 设备的手册也能得到，在我机器上运行 lsusb 是这样的结果：

```
Bus 004 Device 001: ID 0000:0000

Bus 003 Device 002: ID 1234:2345   Abc   Corp.

Bus 002 Device 001: ID 0000:0000

Bus 001 Device 001: ID 0000:0000
```

得到这两个值后把它定义到程序里就可以了。

注册 USB 驱动程序：所有的 USB 驱动程序都必须创建的结构体是 struct usb_driver。这个结构体必须由 USB 驱动程序来填写，包括许多回调函数和变量，它们向 USB 核心代码描述 USB 驱动程序。创建一个有效的 struct usb_driver 结构体，只须要初始化五个字段就可以了，在框架程序中是这样的：

```
static struct usb_driver skel_driver = {

        .owner = THIS_MODULE,

        .name =          "skeleton",

        .probe =   skel_probe,

        .disconnect =   skel_disconnect,

        .id_table =      skel_table,
```

};

struct module *owner ：指向该驱动程序的模块所有者的批针。USB 核心使用它来正确地对该 USB 驱动程序进行引用计数，使它不会在不合适的时刻被卸载掉，这个变量应该被设置为 THIS_MODULE 宏。

const char *name：指向驱动程序名字的指针，在内核的所有 USB 驱动程序中它必须是唯一的，通常被设置为和驱动程序模块名相同的名字。

int (*probe) (struct usb_interface *intf,const struct usb_device_id *id)：这个是指向 USB 驱动程序中的探测函数的指针。当 USB 核心认为它有一个接口（usb_interface）可以由该驱动程序处理时，这个函数被调用。

void (disconnect)(struct usb_interface *intf)：指向 USB 驱动程序中的断开函数的指针，当一个 USB 接口（usb_interface）被从系统中移除或者驱动程序正在从 USB 核心中卸载时，USB 核心将调用这个函数。

const struct usb_device_id *id_table：指向 ID 设备表的指针，这个表包含了一列该驱动程序可以支持的 USB 设备，如果没有设置这个变量，USB 驱动程序中的探测回调函数就不会被调用。

在这个结构体中还有其它的几个回调函数不是很常用，这里就不一一说明了。以 struct usb_driver 指针为参数的 usb_register_driver 函数调用把 struct usb_driver 注册到 USB 核心。一般是在 USB 驱动程序的模块初始化代码中完成这个工作的：

```
static int __init usb_skel_init(void)

{

    int result;



    /*  驱动程序注册到 USB 子系统中*/

    result = usb_register(&skel_driver);

    if (result)

        err("usb_register failed. Error number %d", result);
```

```
        return result;
    }
```

当 USB 驱动程序将要被卸开时，需要把 struct usb_driver 从内核中注销。通过调用 usb_deregister_driver 来完成这个工作，当调用发生时，当前绑定到该驱动程序上的任何 USB 接口都被断开，断开函数将被调用：

```
    static void __exit usb_skel_exit(void)
    {
        /* 从子系统注销驱动程序 */
        usb_deregister(&skel_driver);
    }
```

探测和断开：当一个设备被安装而 USB 核心认为该驱动程序应该处理时，探测函数被调用，探测函数检查传递给它的设备信息，确定驱动程序是否真的适合该设备。当驱动程序因为某种原因不应该控制设备时，断开函数被调用，它可以做一些清理工作。探测回调函数中，USB 驱动程序初始化任何可能用于控制 USB 设备的局部结构体，它还把所需的任何设备相关信息保存到一个局部结构体中，下面是探测函数的部分源码，我们加以分析。

```
        /* 设置端点信息 */

        /* 只使用第一个批量 IN 和批量 OUT 端点 */
        iface_desc = interface->cur_altsetting;
        for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
            endpoint = &iface_desc->endpoint[i].desc;


            if (!dev->bulk_in_endpointAddr &&
```

```
   (endpoint->bEndpointAddress & USB_DIR_IN) &&
   ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
               == USB_ENDPOINT_XFER_BULK)) {
     /* 找到一个批量 IN 端点 */
     buffer_size = endpoint->wMaxPacketSize;
     dev->bulk_in_size = buffer_size;
     dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
     dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
    if (!dev->bulk_in_buffer) {
          err("Could not allocate bulk_in_buffer");
        & nbsp; goto error;
     }
}

if (!dev->bulk_out_endpointAddr &&
   !(endpoint->bEndpointAddress & USB_DIR_IN) &&
   ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
               == USB_ENDPOINT_XFER_BULK)) {
     /* 找到一个批量 OUT 端点 */

     dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
```

```
        }
    }
    if (!(dev->bulk_in_endpointAddr && dev->bulk_out_endp ointAddr)) {
        err("Could not find both bulk-in and bulk-out endpoints");
        goto error;
    }
```

在探测函数里，这个循环首先访问该接口中存在的每一个端点，给该端点一个局部指针以便以后访问：

```
for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
        endpoint = &amp;iface_desc->endpoint[i].desc;
```

在一轮探测过后，我们就有了一个端点，在还没有发现批量 IN 类型的端点时，探测该端点方向是否为 IN，这可以通过检查 USB_DIR_IN 是否包含在 bEndpointAddress 端点变量有确定，如果是的话，我们在探测该端点类型是否为批量，先用 USB_ENDPOINT_XFERTYPE_MASK 位掩来取 bmAttributes 变量的值，然后探测它是否和 USB_ENDPOINT_XFER_BULK 值匹配：

```
        if (!dev->bulk_out_endpointAddr &&
          !(endpoint->bEndpointAddress & USB_DIR_IN) &&
          ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
                  == USB_ENDPOINT_XFER_BULK))
```

如果所有这些探测都通过了，驱动程序就知道它已经发现了正确的端点类型，可以把该端点的相关信息保存到一个局部结构体中以便稍后用它来和端点进行通信：

```
            /*  找到一个批量 IN 类型的端点  */
            buffer_size = endpoint->wMaxPacketSize;
            dev->bulk_in_size = buffer_size;
```

```
                    dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;

                dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);

                if (!dev->bulk_in_buffer) {

                    err("Could not allocate bulk_in_buffer");

                    goto error;

                }
```

因为 USB 驱动程序要在设备的生命周期的稍后时间获取和接口相关联的局部数据结构体，所以调用了 usb_set_intfdata 函数，把它保存到 struct usb_interface 结构体中以便后面的访问

```
        /* 把数据指针保存到这个接口设备中 */

        usb_set_intfdata(interface, dev);
```

我们以后调用 usb_set_intfdata 函数来获取数据。当这一切都完成后，USB 驱动程序必须在探测函数中调用 usb_register_dev 函数来把该设备注册到 USB 核心里：

```
        /* 注册设备到 USB 核心 */

        retval = usb_register_dev(interface, &skel_class);

        if (retval) {

            /* 有些情况下是不允许注册驱动程序的 */

            err("Not able to get a minor for this device.");

            usb_set_intfdata(interface, NULL);

            goto error;

        }
```

当一个 USB 设备被断开时，和该设备相关联的所有资源都应该被尽可能的清理掉，在此时，如果已在在探测函数中调 用了注册函数来为该 USB 设备分配了一个次设备号话，必须调用 usb_deregister_dev 函数来把次设备号交还给 USB 核心。在断开函数中，从接口

获取之前调用 usb_set_intfdata 设置的任何数据也是很重要的。然后设置 struct usb_interface 结构体中的数据指针为 NULL，以防任何不适当的对该数据的错误访问。

在探测函数中会对每一个接口进行一次探测，所以我们在写 USB 驱动程序的时候，只要做好第一个端点，其它的端点就会自动完成探测。在探测函数中我们要注意的是在内核中用结构体 struct usb_host_endpoint 来描述 USB 端点，这个结构体在另一个名为 struct usb_endpoint_descriptor 的结构体中包含了真正的端点信息，struct usb_endpoint_descriptor 结构体包含了所有的 USB 特定的数据，该结构体中我们要关心的几个字段是：

bEndpointAddress：这个是特定的 USB 地址，可以结合 USB_DIR_IN 和 USB_DIR_OUT 来使用，以确定该端点的数据是传向设备还是主机。

bmAttributes：这个是端点的类型，这个值可以结合位掩码 USB_ENDPOINT_XFERTYPE_MASK 来使用，以确定此端点的类型是 USB_ENDPOINT_XFER_ISOC（等时）、USB_ENDPOINT_XFER_BULK（批量）、USB_ENDPOINT_XFER_INT 的哪一种。

wMaxPacketSize：这个是端点一次可以处理的最大字节数，驱动程序可以发送数量大于此值的数据到端点，在实际传输中，数据量如果大于此值会被分割。

bInterval：这个值只有在端点类型是中断类型时才起作用，它是端点中断请求的间隔时间，以毫秒为单位。

提交和控制 urb：当驱动程序有数据要发送到 USB 设备时（大多数情况是在驱动程序的写函数中），要分配一个 urb 来把数据传输给设备：

```
/*  创建一个 urb,并且给它分配一个缓存*/

 urb = usb_alloc_urb(0, GFP_KERNEL);

 if (!urb) {

       retval = -ENOMEM;

       goto error;

 }
```

当 urb 被成功分配后，还要创建一个 DMA 缓冲区来以高效的方式发送数据到设备，传递给驱动程序的数据要复制到这块缓冲中去：

```
buf = usb_buffer_alloc(dev->udev, count, GFP_KERNEL, &urb->transfer_dma);
if (!buf) {
    retval = -ENOMEM;
    goto error;
}

if (copy_from_user(buf, user_buffer, count)) {
    retval = -EFAULT;
    goto error;
}
```

当数据从用户空间正确复制到局部缓冲区后，urb 必须在可以被提交给 USB 核心之前被正确初始化：

```
/* 初始化 urb */
usb_fill_bulk_urb(urb, dev->udev,
          usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
          buf, count, skel_write_bulk_callback, dev);
urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
```

然后 urb 就可以被提交给 USB 核心以传输到设备了：

```
/* 把数据从批量 OUT 端口发出 */
retval = usb_submit_urb(urb, GFP_KERNEL);
if (retval) {
    err("%s - failed submitting write urb, error %d", __FUNCTION__, retval);
```

```
            goto error;
        }
```

当 urb 被成功传输到 USB 设备之后，urb 回调函数将被 USB 核心调用，在我们的例子中，我们初始化 urb，使它指向 skel_write_bulk_callback 函数，以下就是该函数：

```
 static void skel_write_bulk_callback(struct urb *urb, struct pt_regs *regs)
    {
        struct usb_skel *dev;


        dev = (struct usb_skel *)urb->context;


        if (urb->status &&
          !(urb->status == -ENOENT ||
            urb->status == -ECONNRESET ||
          urb->status == -ESHUTDOWN)) {
            dbg("%s - nonzero write bulk status received: %d",
              __FUNCTION__, urb->status);
        }
            /* 释放已分配的缓冲区 */
        usb_buffer_free(urb->dev, urb->transfer_buffer_length,
                urb->transfer_buffer, urb->transfer_dma);
    }
```

来源：http://www.icembed.com/info-20574.htm