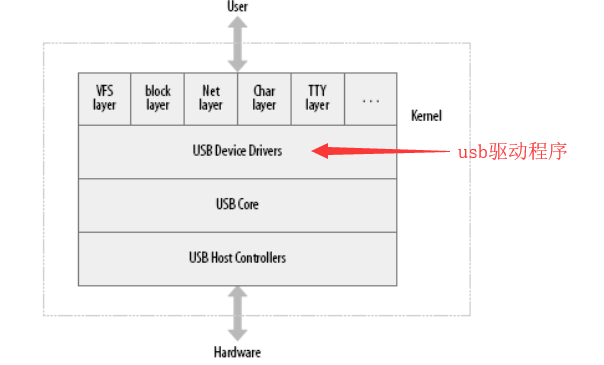
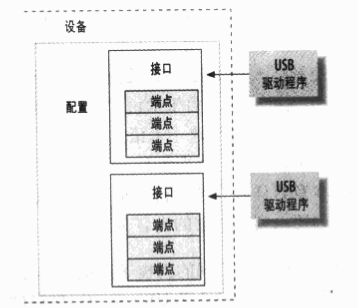
USB协议定义了一套标准，如果设备遵循这套标准，那么我们无需编写特殊的驱动程序



我们接下来介绍的概念的关系如下



1. **端点**

USB通过端点与设备进行通信，端点只能单向传输数据

4中不同类型的端点

CONTROL（控制）

通常用于配置设备，获取设备信息

INTERRUPT （中断）

用于传输少量数据（如usb键盘，鼠标）

BULK （批量）

用于传输大量数据（如打印机，存储设备通常会有BULK端口）

ISOCHRONOUS

用于传输大量数据，但不保证数据到达（如音频视频设备）

**usb\_host\_endpoint结构**

内核中使用结构 struct usb\_host\_endpoint 来描述来描述端点，我们关心如下字段：

bEndpointAddress

USB端点的地址，其中包含了数据传输方向，使用为掩码来USB\_DIR\_OUT 和 USB\_DIR\_IN进行判断

bmAttributes

这是端点的类型. 通过和位掩码 USB\_ENDPOINT\_XFERTYPE\_MASK 的 & 值然后和USB\_ENDPOINT\_XFER\_ISOC（等时）、USB\_ENDPOINT\_XFER\_BULK（批量）、USB\_ENDPOINT\_XFER\_INT（中断）比较判断，如

if((endpoint->bmAttributes & USB\_ENDPOINT\_XFERTYPE\_MASK)==USB\_ENDPOINT\_XFER\_BULK)

wMaxPacketSize

此端口一次可处理的最大字节数

bInterval

如果这个端点是中断类型的, 这个值是端点的间隔设置

1. **接口**

USB接口代表一个基本的功能，一个接口对应一个驱动程序

内核使用struct usb\_interface结构来表示一个接口，在驱动程序的探测函数中，内核会将usb\_interface作为参数传进来

每个usb\_interface可以有多种设置，每种设置会影响端口的行为

**usb\_interface结构**

struct usb\_host\_interface \*altsetting

一个包含所有可选设置的接口结构的数组，每struct usb\_host\_interface 包含一套端点的配置

unsigned num\_altsetting

由 altsetting 指针指向的可选设置的数目.

struct usb\_host\_interface \*cur\_altsetting

指向数组 altsetting 内部的一个指针, 表示这个接口当前的激活的设置.

int minor

如果绑定到这个接口的 USB 驱动使用 USB 主编号, 这个变量包含由 USB 核心安排给接口的次编号.

1. **配置与USB设备**

linux 描述 USB 配置使用结构 struct usb\_host\_config

Linux描述 USB 设备使用结构 struct usb\_device.

一个 USB 设备驱动通常需要从给定的 struct usb\_interface 结构转到 struct usb\_device 结构，可以使用函数interface\_to\_usbdev

USB 设备是非常复杂的，上述介绍的关系如下

1. 设备通常具有一个或者更多的配置
2. 配置经常具有一个或者更多的接口
3. 接口通常具有一个或者更多的设置
4. 接口没有或者具有一个以上的端点
5. **USB urb**

和USB设备通信是通过传送urb（USB请求块）完成的

**urb结构体**

struct usb\_device \*dev

目标usb设备

unsigned int pipe

要使用的端点的信息

驱动程序必须使用如下函数的返回值来设置该字段的信息

unsigned int usb\_sndctrlpipe(struct usb\_device \*dev, unsigned int endpoint)

把指定设备的指定端口设为 控制 OUT 端口

unsigned int usb\_rcvctrlpipe(struct usb\_device \*dev, unsigned int endpoint)

把指定设备的指定端口设为 控制 IN 端口

unsigned int usb\_sndbulkpipe(struct usb\_device \*dev, unsigned int endpoint)

把指定设备的指定端口设为 批量 OUT 端口

unsigned int usb\_rcvbulkpipe(struct usb\_device \*dev, unsigned int endpoint)

把指定设备的指定端口设为 批量 IN 端口

unsigned int usb\_sndintpipe(struct usb\_device \*dev, unsigned int endpoint)

把指定设备的指定端口设为 中断 OUT 端口

unsigned int usb\_rcvintpipe(struct usb\_device \*dev, unsigned int endpoint)

把指定设备的指定端口设为 中断 IN 端口

unsigned int usb\_sndisocpipe(struct usb\_device \*dev, unsigned int endpoint)

把指定设备的指定端口设为 等时 OUT 端口

unsigned int usb\_rcvisocpipe(struct usb\_device \*dev, unsigned int endpoint)

把指定设备的指定端口设为 等时 IN 端口

void \*transfer\_buffer

指向用在发送数据到设备(对一个 OUT urb)或者从设备中获取数据(对于一个 IN urb)的缓冲的指针.

dma\_addr\_t transfer\_dma

用来使用 DMA 传送数据到 USB 设备的缓冲.

int transfer\_buffer\_length

transfer\_buffer 或 transfer\_dma 缓冲区的长度

int actual\_length

Urb发送或接收结束后，这个变量被设置为发送或接收数据的真实长度

usb\_complete\_t complete

urb完成的回调函数的指针

usb\_complete\_t 类型定义如此:

typedef void (\*usb\_complete\_t)(struct urb \*, struct pt\_regs \*);

void \*context

保存驱动程序数据的字段

**创建urb**

urb由USB核心维护其引用计数，因此urb不能静态创建

使用如下函数创建urb

struct urb \*usb\_alloc\_urb(int iso\_packets, int mem\_flags);

iso\_packet：

Urb包含等时数据包的数量，如果不创建等时urb，这个变量应当被设置为 0.

mem\_flags：

和传递给 kmalloc 函数从内核分配内存的标志相同

**销毁urb**

void usb\_free\_urb(struct urb \*urb);

**urb初始化**

中断urb

void usb\_fill\_int\_urb(

    // urb 指针

    struct urb \*urb,

    // 目标 usb 设备

    struct usb\_device \*dev,

    // 目标端口的信息

    unsigned int pipe,

    // urb 的缓存

    void \*transfer\_buffer,

    // 缓存的大小

    int buffer\_length,

    // urb 结束之后调用的结束例程的指针

    usb\_complete\_t complete,

    // urb 的 context，保存驱动程序的数据

    void \*context,

    // 被调度的时间间隔

    int interval);

批量urb

void usb\_fill\_bulk\_urb(

    // urb 指针

    struct urb \*urb,

    // 目标 usb 设备

    struct usb\_device \*dev,

    // 目标端口的信息

    unsigned int pipe,

    // urb 的缓存

    void \*transfer\_buffer,

    // 缓存的大小

    int buffer\_length,

    // urb 结束之后调用的结束例程的指针

    usb\_complete\_t complete,

    // urb 的 context，保存驱动程序的数据

    void \*context);

控制urb

void usb\_fill\_control\_urb(

    // urb 指针

    struct urb \*urb,

    // 目标 usb 设备

    struct usb\_device \*dev,

    // 目标端口的信息

    unsigned int pipe,

    // 指向即将被发送至端点的设置数据包的数据

    unsigned char \*setup\_packet,

    // urb 的缓存

    void \*transfer\_buffer,

    // 缓存的大小

    int buffer\_length,

    // urb 结束之后调用的结束例程的指针

    usb\_complete\_t complete,

    // urb 的 context，保存驱动程序的数据

    void \*context);

等时urb

等时urb没有初始化函数，需要手工初始化

urb->dev = dev;

urb->context = uvd;

urb->pipe = usb\_rcvisocpipe(dev, uvd->video\_endp - 1);

urb->interval = 1;

urb->transfer\_flags = URB\_ISO\_ASAP;

urb->transfer\_buffer = cam->sts\_buf[i];

urb->complete = konicawc\_isoc\_irq;

urb->number\_of\_packets = FRAMES\_PER\_DESC;

urb->transfer\_buffer\_length = FRAMES\_PER\_DESC;

for (j = 0; j < FRAMES\_PER\_DESC; j++)

{

    urb->iso\_frame\_desc[j].offset = j;

    urb->iso\_frame\_desc[j].length = 1;

}

**提交urb**

int usb\_submit\_urb(struct urb \*urb, int mem\_flags);

Urb：以初始化的urb

Mem\_flags：与kmalloc标志相同

返回值：成功返回0，否则返回负值

**取消urb**

如果urb以提交到USB核心，可以调用如下函数来取消

int usb\_kill\_urb(struct urb \*urb);

int usb\_unlink\_urb(struct urb \*urb);

1. **不使用urb的数据传输**

有时我们只想发送或接收一些简单的数据而不想像上述繁琐，使用如下函数一步完成

int usb\_bulk\_msg(struct usb\_device \*usb\_dev,

                 unsigned int pipe,

                 void \*data,

                 int len,

                 int \*actual\_length,

                 int timeout);

这个函数的参数是:

struct usb\_device \*usb\_dev

目标usb

unsigned int pipe

端口信息

void \*data

指向数据的缓存区

int len

被 data 参数指向的缓冲的长度

int \*actual\_length

指向函数放置真实数据字节数的指针

int timeout

超时时间，以滴答计算. 如果这个值是 0, 函数永远等待消息完成.

1. **如何编写USB驱动程序**

**声明驱动支持的设备列表并导出**

/\* 声明驱动程序支持的设备列表 \*/

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

    {USB\_DEVICE(USB\_SKEL\_VENDOR\_ID, USB\_SKEL\_PRODUCT\_ID)},

    {} /\* 以空结尾 \*/

};

// 导出设备列表

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

**编写usb\_driver结构**

static struct usb\_driver skel\_driver = {

    .owner = THIS\_MODULE,

    .name = "skeleton",

    .id\_table = skel\_table,

    .probe = skel\_probe,

    .disconnect = skel\_disconnect,

};

此结构与PCI驱动的pci\_driver结构相似，这里不做详细介绍

struct module \*owner

指向这个驱动的模块拥有者的指针.

const char \*name

驱动名字的指针.

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

驱动支持的设备列表

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

指向 USB 驱动中探测函数的指针.

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

指向 USB 驱动的断开连接函数的指针.

**注册USB驱动程序**

static int \_\_init usb\_skel\_init(void)

{

    int result;

    // 模块初始化时注册

    result = usb\_register(&skel\_driver);

    if (result)

        err("usb\_register failed. Error number %d", result);

    return result;

}

static void \_\_exit usb\_skel\_exit(void)

{

    // 模块卸载时注销

    usb\_deregister(&skel\_driver);

}

**探测函数的细节**

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

{

    struct usb\_skel \*dev;                     //特定设备结构体

    struct usb\_host\_interface \*iface\_desc;    //设置结构体

    struct usb\_endpoint\_descriptor \*endpoint; //端点描述符

    size\_t buffer\_size;

    int i;

    int retval = -ENOMEM;

    /\* allocate memory for our device state and initialize it \*/

    dev = kzalloc(sizeof(\*dev), GFP\_KERNEL);

    if (!dev)

    {

        err("Out of memory");

        goto error;

    }

    kref\_init(&dev->kref);                        ////初始化内核引用计数

    sema\_init(&dev->limit\_sem, WRITES\_IN\_FLIGHT); //初始化信号量

    mutex\_init(&dev->io\_mutex);                   //初始化互斥锁

    spin\_lock\_init(&dev->err\_lock);               //初始化自旋锁

    init\_usb\_anchor(&dev->submitted);

init\_completion(&dev->bulk\_in\_completion); //初始化完成量

    dev->udev = usb\_get\_dev(interface\_to\_usbdev(interface)); //获取usb\_device结构体

dev->interface = interface;                              //获取usb\_ interface结构体

    // 获取接口激活的设置

iface\_desc = interface->cur\_altsetting;

    // 循环接口设置中断端口

    for (i = 0; i < iface\_desc->desc.bNumEndpoints; ++i)

    {

        // 获取端口

        endpoint = &iface\_desc->endpoint[i].desc;

        // 查找用于读取的端口

        if (!dev->bulk\_in\_endpointAddr &&                                                      // 当前dev结构的读取端口地址为空

            (endpoint->bEndpointAddress & USB\_DIR\_IN) &&                                       // 端口的数据方向为IN（读取）

            ((endpoint->bmAttributes & USB\_ENDPOINT\_XFERTYPE\_MASK) == USB\_ENDPOINT\_XFER\_BULK)) // 端口为批量类型

        {

            // 获取端口一次可处理的最多字节

            buffer\_size = endpoint->wMaxPacketSize;

            // 记录端口一次可处理的最多字节

            dev->bulk\_in\_size = buffer\_size;

            // 记录端口的地址

            dev->bulk\_in\_endpointAddr = endpoint->bEndpointAddress;

            // 为 dev 结构分配一个缓存区

            dev->bulk\_in\_buffer = kmalloc(buffer\_size, GFP\_KERNEL);

            if (!dev->bulk\_in\_buffer)

            {

                err("Could not allocate bulk\_in\_buffer");

                goto error;

            }

        }

        // 查找用于写入的端口

        if (!dev->bulk\_out\_endpointAddr &&                                                     // 当前dev写入的端口地址为空

            !(endpoint->bEndpointAddress & USB\_DIR\_IN) &&                                      // 当前端口的方向为OUT（写入）

            ((endpoint->bmAttributes & USB\_ENDPOINT\_XFERTYPE\_MASK) == USB\_ENDPOINT\_XFER\_BULK)) // 当前端口为批量类型

        {

            // 记录端口的写入地址

            dev->bulk\_out\_endpointAddr = endpoint->bEndpointAddress;

        }

}

    if (!(dev->bulk\_in\_endpointAddr && dev->bulk\_out\_endpointAddr))

    {

        err("Could not find both bulk-in and bulk-out endpoints");

        goto error;

}

    // 将dev结构保存到接口中，以后可以使用usb\_get\_intfdata获取接口保存的数据

    // 为什么不在函数外使用静态变量保存dev的指针？

    // 因为我们驱动程序可能会支持多个usb设备，如果使用静态变量，我们需要自己维护接口到dev结构的映射

usb\_set\_intfdata(interface, dev);

    // skel\_class为 struct usb\_class\_driver 的指针，其结构如下

    // char \*name

    //      sysfs 用来描述设备的名字

    // struct file\_operations \*fops;

    //      指向 struct file\_operations 的结构的指针, 这个结构的更多信息见第 3 章.

    // mode\_t mode;

    //      给这个驱动的要被创建的 devfs 文件的模式

    // int minor\_base;

    //      这是给这个驱动安排的次编号的开始

    retval = usb\_register\_dev(interface, &skel\_class); //注册USB设备

    if (retval)

    {

        /\* something prevented us from registering this driver \*/

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

        usb\_set\_intfdata(interface, NULL);

        goto error;

}

    dev\_info(&interface->dev,

             "USB Skeleton device now attached to USBSkel-%d",

             interface->minor);

return 0;

error:

    if (dev)

        kref\_put(&dev->kref, skel\_delete);

    return retval;

}

**断开连接函数的细节**

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

{

    struct usb\_skel \*dev;

    int minor = interface->minor;

    lock\_kernel();

    // 从接口中获取保存的数据

    dev = usb\_get\_intfdata(interface);

    // 设置接口保存的数据为空

    usb\_set\_intfdata(interface, NULL);

    // 注销设备

    usb\_deregister\_dev(interface, &skel\_class);

    unlock\_kernel(); /\* decrement our usage count \*/

    kref\_put(&dev->kref, skel\_delete);

    info("USB Skeleton #%d now disconnected", minor);

}

**提交和控制urb**

分配urb

urb = usb\_alloc\_urb(0, GFP\_KERNEL);

if (!urb)

{

    retval = -ENOMEM;

    goto error;

}

这里使用DMA缓存来高效的发送数据到设备，下面申请DMA缓存

// dev->udev保存usb\_driver结构体

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;

}

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;

提交urb

retval = usb\_submit\_urb(urb, GFP\_KERNEL);

if (retval)

{

    err("%s - failed submitting write urb, error %d", \_\_FUNCTION\_\_, retval);

    goto error;

}

urb成功后回调函数

static void skel\_write\_bulk\_callback(struct urb \*urb, struct pt\_regs \*regs)

{

    /\* 判断是否有错误 \*/

    if (urb->status &&

        !(urb->status == -ENOENT ||

          urb->status == -ECONNRESET ||

          urb->status == -ESHUTDOWN))

    {

        dbg("%s - nonzero write bulk status received: %d", \_\_FUNCTION\_\_, urb->status);

    }

    /\* 释放我们分配的缓存 \*/

    usb\_buffer\_free(urb->dev, urb->transfer\_buffer\_length, urb->transfer\_buffer, urb->transfer\_dma);

}