Linux USB 驱动框架分析（一）

初次接触和 OS 相关的设备驱动编写，感觉还挺有意思的，为了不至于忘掉看过的东西，笔记跟总结当然不可缺，更何况我决定为嵌入式卖命了。好，言归正传，我说一说这段时间的收获，跟大家分享一下 Linux 的驱动研发。但这次只先针对 Linux 的 USB 子系统作分析，因为周五研讨老板催货。当然，还会顺带提一下其他的驱动程式写法。

事实上，Linux 的设备驱动都遵循一个惯例??表征驱动程式（用 driver 更贴切一些，应该称为驱动器比较好吧）的结构体，结构体里面应该包含了驱动程式所需要的所有资源。用术语来说，就是这个驱动器对象所拥有的属性及成员。由于 Linux 的内核用 c 来编写，所以我们也按照这种结构化的思想来分析代码，但我还是希望从 OO 的角度来阐述这些细节。这个结构体的名字有驱动研发人员决定，比如说，鼠标可能有一个叫做 mouse_dev 的 struct，键盘可能由一个 keyboard_dev 的 struct（dev for device，我们做的只是设备驱动）。而这次我们来分析一下 Linux 内核源码中的一个 usb-skeleton（就是 usb 驱动的骨架咯），自然，他定义的设备结构体就叫做 usb-skel：

```
struct usb_skel {
        struct usb_device *          udev;                    /* the usb device for this device */
        struct usb_interface *      interface;               /* the interface for this device */
        struct semaphore             limit_sem;              /* limiting the number of writes in progress
*/
        unsigned char *                bulk_in_buffer;        /* the buffer to receive data */
        size_t                bulk_in_size;                   /* the size of the receive buffer */
        __u8                  bulk_in_endpointAddr;          /* the address of the bulk in endpoint */
        __u8                  bulk_out_endpointAddr;         /* the address of the bulk out endpoint */
        struct kref      kref;
};
```

这里我们得补充说明一下一些 USB 的协议规范细节。USB 能够自动监测设备，并调用相应得驱动程式处理设备，所以其规范实际上是相当复杂的，幸好，我们不必理会大部分细节问题，因为 Linux 已提供相应的解决方案。就我目前的理解来说，USB 的驱动分为两块，一块是 USB 的 bus 驱动，这个东西，Linux 内核已做好了，我们能不管，但我们至少要了解他的功能。形象得说，USB 的 bus 驱动相当于铺出一条路来，让所有的信息都能通过这条 USB 通道到达该到的地方，这部分工作由 usb_core 来完成。当 USB 设备接到 USB 控制器接口时，usb_core 就检测该设备的一些信息，例如生产厂商 ID 和产品的 ID，或是设备所属的 class、subclass 跟 protocol，以便确定应该调用哪一个驱动处理该设备。里面复杂细节我们不用管，我们要做的是另一块工作??usb 的设备驱动。也就是说，我们就等着 usb_core 告诉我们要工作了，我们才工作。

从研发人员的角度看，每一个 usb 设备有若干个设置(configuration)组成，每个设置又能有多个接口(interface)，每个接口又有多个设置(setting 图中没有给出)，而接口本身可能没有端点或多个端点（end point）。USB 的数据交换通过端点来进行，主机和各个端点之间建立起单向的管道来传输数据。而这些接口能分为四类：

控制（control）

用于设置设备、获取设备信息、发送命令或获取设备的状态报告

中断（interrupt）

当 USB 宿主需求设备传输数据时，中断端点会以一个固定的速率传送少量数据，还用于发送数据到 USB 设备以控制设备，一般不用于传送大量数据。

批量（bulk）

用于大量数据的可靠传输，如果总线上的空间不足以发送整个批量包，他会被分割成多个包传输。

等时（isochronous）

大量数据的不可靠传输，不确保数据的到达，但确保恒定的数据流，多用于数据采集。

Linux 中用 struct usb_host_endpoint 来描述 USB 端点，每个 usb_host_endpoint 中包含一个 struct usb_endpoint_descriptor 结构体，当中包含该端点的信息及设备自定义的各种信息，这些信息包括：

bEndpointAddress（b for byte）

8 位端点地址，其地址还隐藏了端点方向的信息（之前说过，端点是单向的），能用掩码 USB_DIR_OUT 和 USB_DIR_IN 来确定。

bmAttributes

端点的类型，结合 USB_ENDPOINT_XFERTYPE_MASK 能确定端点是 USB_ENDPOINT_XFER_ISOC（等时）、USB_ENDPOINT_XFER_BULK（批量）还是 USB_ENDPOINT_XFER_INT（中断）。

wMaxPacketSize

端点一次处理的最大字节数。发送的 BULK 包能大于这个数值，但会被分割传送。

bInterval

如果端点是中断类型，该值是端点的间隔设置，以毫秒为单位。

在逻辑上，一个 USB 设备的功能划分是通过接口来完成的。比如说一个 USB 扬声器，可能会包括有两个接口：一个用于键盘控制，另外一个用于音频流传输。而事实上，这种设备需要用到不同的两个驱动程式来操作，一个控制键盘，一个控制音频流。但也有例外，比如蓝牙设备，需求有两个接口，第一用于 ACL 跟 EVENT 的传输，另外一个用于 SCO 链路，但两者通过一个驱动控制。在 Linux 上，接口使用 struct usb_interface 来描述，以下是该结构体中比较重要的字段：

struct usb_host_interface *altsetting（注意不是 usb_interface）

其实据我理解，他应该是每个接口的设置，虽然名字上有点奇怪。该字段是个设置的数组（一个接口能有多个设置），每个 usb_host_interface 都包含一套由 struct usb_host_endpoint 定义的端点设置。但这些设置次序是不定的。

unsigned num_altstting

可选设置的数量，即 altsetting 所指数组的元素个数。

struct usb_host_interface *cur_altsetting

当前活动的设置，指向 altsetting 数组中的一个。

int minor

当捆绑到该接口的 USB 驱动程式使用 USB 主设备号时，USB core 分配的次设备号。仅在成功调用 usb_register_dev 之后才有效。

除了他能用 struct usb_host_config 来描述之外，到目前为止，我对设置的了解不多。而整个 USB 设备则能用 struct usb_device 来描述，但基本上只会用他来初始化函数的接口，真正用到的应该是我们之前所提到的自定义的一个结构体。

Linux USB 驱动框架分析（二）

好，了解过 USB 一些规范细节之后，我们目前来看看 Linux 的驱动框架。事实上，Linux 的设备驱动，特别是这种 hotplug 的 USB 设备驱动，会被编译成模块，然后在需要时挂在到内核。要写一个 Linux 的模块并不复杂，以一个 helloworld 为例：

```
#include
#include
MODULE_LICENSE(“GPL”);

static int hello_init(void)
{
        printk(KERN_ALERT “Hello World! \ n”);
        return 0;
}
static int hello_exit(void)
{
        printk(KERN_ALERT “GOODBYE! \ n”);
}
module_init(hello_init);
module_exit(hello_exit);
```

这个简单的程式告诉大家应该怎么写一个模块，MODULE_LICENSE 告诉内核该模块的版权信息，非常多情况下，用 GPL 或 BSD，或两个，因为一个私有模块一般非常难得到社区的帮助。module_init 和 module_exit 用于向内核注册模块的初始化函数和模块推出函数。如程式所示，初始化函数是 hello_init，而退出函数是 hello_exit。

另外，要编译一个模块通常还需要用到内核源码树中的 makefile，所以模块的 Makefile 能写成：

```
ifneq ($(KERNELRELEASE),)
obj-m:= hello.o#usb-dongle.o

else
KDIR:= /usr/src/linux-headers-$(shell uname -r)
BDIR:= $(shell pwd)
default:
        $(MAKE) -C $(KDIR) M=$(PWD) modules
.PHONY: clean

clean:
        make -C $(KDIR) M=$(BDIR) clean
endif
```

能用 insmod 跟 rmmod 来验证模块的挂在跟卸载，但必须用 root 的身份登陆命令行，用普通用户加 su 或 sudo 在 Ubuntu 上的测试是不行的。

Linux USB 驱动框架分析（三）

下面分析一下 usb-skeleton 的源码。这个范例程式能在 linux-2.6.17/drivers/usb 下找到，其他版本的内核程式源码可能有所不同，但相差不大。大家能先找到源码看一看，先有个整体印象。

之前已提到，模块先要向内核注册初始化跟销毁函数：

```
static int __init usb_skel_init(void)
{
        int result;
```

```
	/* register this driver with the USB subsystem */
	result = usb_register(&skel_driver);
	if (result)
		err("usb_register failed. Error number %d", result);

	return result;
}

static void __exit usb_skel_exit(void)
{
	/* deregister this driver with the USB subsystem */
	usb_deregister(&skel_driver);
}

module_init (usb_skel_init);
module_exit (usb_skel_exit);
MODULE_LICENSE("GPL");
```

从代码开来，这个 init 跟 exit 函数的作用只是用来注册驱动程式，这个描述驱动程式的结构体是系统定义的标准结构 struct usb_driver, 注册和注销的方法非常简单，usb_register（struct *usb_driver）, usb_deregister（struct *usb_driver）。那这个结构体需要做些什么呢？他要向系统提供几个函数入口，跟驱动的名字：

```
static struct usb_driver skel_driver = {
	.name =		"skeleton",
	.probe =	skel_probe,
	.disconnect = skel_disconnect,
	.id_table =	skel_table,
};
```

从代码看来，usb_driver 需要初始化四个东西：模块的名字 skeleton，probe 函数 skel_probe,disconnect 函数 skel_disconnect, 及 id_table。

在解释 skel_driver 各个成员之前，我们先来看看另外一个结构体。这个结构体的名字有研发人员自定义，他描述的是该驱动拥有的所有资源及状态：

```
struct usb_skel {
	struct usb_device *	udev;			/* the usb device for this device */
	struct usb_interface *	interface;		/* the interface for this device */
	struct semaphore	limit_sem;		/* limiting the number of writes in progress
*/
	unsigned char *		bulk_in_buffer;		/* the buffer to receive data */
	size_t			bulk_in_size;		/* the size of the receive buffer */
	__u8			bulk_in_endpointAddr;	/* the address of the bulk in endpoint */
	__u8			bulk_out_endpointAddr;	/* the address of the bulk out endpoint */
	struct kref		kref;
};
```

我们先来对这个 usb_skel 作个简单分析，他拥有一个描述 usb 设备的结构体 udev，一

个接口 interface，用于并发访问控制的 semaphore(信号量) limit_sem，用于接收数据的缓冲 bulk_in_buffer 及其尺寸 bulk_in_size，然后是批量输入输出端口地址 bulk_in_endpointAddr、bulk_out_endpointAddr，最后是个内核使用的引用计数器。他们的作用我们将在后面的代码中看到。

我们再回过头来看看 skel_driver。

name 用来告诉内核模块的名字是什么，这个注册之后有系统来使用，跟我们关系不大。

id_table 用来告诉内核该模块支持的设备。usb 子系统通过设备的 production ID 和 vendor ID 的组合或设备的 class、subclass 跟 protocol 的组合来识别设备，并调用相关的驱动程式作处理。我们能看看这个 id_table 到底是什么东西：

```
/* Define these values to match your devices */
#define USB_SKEL_VENDOR_ID    0xfff0
#define USB_SKEL_PRODUCT_ID 0xfff0
/* table of devices that work with this driver */
static struct usb_device_id skel_table [] = {
        { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
        { }                                     /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, skel_table);
```

MODULE_DEVICE_TABLE 的第一个参数是设备的类型，如果是 USB 设备，那自然是 usb（如果是 PCI 设备，那将是 pci，这两个子系统用同一个宏来注册所支持的设备。这涉及 PCI 设备的驱动了，在此先不深究）。后面一个参数是设备表，这个设备表的最后一个元素是空的，用于标识结束。代码定义了 USB_SKEL_VENDOR_ID 是 0xfff0，USB_SKEL_PRODUCT_ID 是 0xfff0，也就是说，当有一个设备接到集线器时，usb 子系统就会检查这个设备的 vendor ID 和 product ID，如果他们的值是 0xfff0 时，那么子系统就会调用这个 skeleton 模块作为设备的驱动。

Linux USB 驱动框架分析（四）

probe 是 usb 子系统自动调用的一个函数，有 USB 设备接到硬件集线器时，usb 子系统会根据 production ID 和 vendor ID 的组合或设备的 class、subclass 跟 protocol 的组合来识别设备调用相应驱动程式的 probe（探测）函数，对于 skeleton 来说，就是 skel_probe。系统会传递给探测函数一个 usb_interface *跟一个 struct usb_device_id *作为参数。他们分别是该 USB 设备的接口描述（一般会是该设备的第 0 号接口，该接口的默认设置也是第 0 号设置）跟他的设备 ID 描述（包括 Vendor ID、Production ID 等）。probe 函数比较长，我们分段来分析这个函数：

```
dev->udev = usb_get_dev(interface_to_usbdev(interface));
dev->interface = interface;
```

在初始化了一些资源之后，能看到第一个关键的函数调用??interface_to_usbdev。他同 uo 一个 usb_interface 来得到该接口所在设备的设备描述结构。本来，要得到一个 usb_device 只要用 interface_to_usbdev 就够了，但因为要增加对该 usb_device 的引用计数，我们应该在做一个 usb_get_dev 的操作，来增加引用计数，并在释放设备时用 usb_put_dev 来减少引用计数。这里要解释的是，该引用计数值是对该 usb_device 的计数，并不是对本模块的计数，本模块的计数要由 kref 来维护。所以，probe 一开始就有初始化 kref。事实上，kref_init 操作不单只初始化 kref，还将其置设成 1。所以在出错处理代码中有 kref_put，他把 kref 的计数减 1，如果 kref 计数已为 0，那么 kref 会被释放。kref_put 的第二个参数是个函数指针，指向一个清理函数。注意，该指针不能为空，或 kfree。该函数会在最后一个对 kref 的引用释

放时被调用（如果我的理解不准确，请指正）。下面是内核源码中的一段注释及代码：

```
/**
* kref_put - decrement refcount for object.
* @kref: object.
* @release: pointer to the function that will clean up the object when the
*           last reference to the object is released.
*           This pointer is required, and it is not acceptable to pass kfree
*           in as this function.
*
* Decrement the refcount, and if 0, call release().
* Return 1 if the object was removed, otherwise return 0.  Beware, if this
* function returns 0, you still can not count on the kref from remaining in
* memory.  Only use the return value if you want to see if the kref is now
* gone, not present.
*/
int kref_put(struct kref *kref, void (*release)(struct kref *kref))
{
        WARN_ON(release == NULL);
        WARN_ON(release == (void (*)(struct kref *))kfree);
         /*
          * if current count is one, we are the last user and can release object
          * right now, avoiding an atomic operation on 'refcount'
          */
        if ((atomic_read(&kref->refcount) == 1) ||
            (atomic_dec_and_test(&kref->refcount))) {
            release(kref);
            return 1;
        }
        return 0;
}
```

当我们执行打开操作时，我们要增加 kref 的计数，我们能用 kref_get，来完成。所有对 struct kref 的操作都有内核代码确保其原子性。

得到了该 usb_device 之后，我们要对我们自定义的 usb_skel 各个状态跟资源作初始化。这部分工作的任务主要是向 usb_skel 注册该 usb 设备的端点。这里可能要补充以下一些关于 usb_interface_descriptor 的知识，但因为内核源码对该结构体的注释不多，所以只能靠个人猜测。在一个 usb_host_interface 结构里面有一个 usb_interface_descriptor 叫做 desc 的成员，他应该是用于描述该 interface 的一些属性，其中 bNumEndpoints 是个 8 位（b for byte）的数字，他代表了该接口的端点数。probe 然后遍历所有的端点，检查他们的类型跟方向，注册到 usb_skel 中。

```
        /* set up the endpoint information */
        /* use only the first bulk-in and bulk-out endpoints */
        iface_desc = interface->cur_altsetting;
        for (i = 0; i desc.bNumEndpoints; ++i) {
            endpoint = &iface_desc->endpoint.desc;
```

```
            if ( !dev->bulk_in_endpointAddr &&
                    ((endpoint->bEndpointAddress & USB_ENDPOINT_DIR_MASK) == =
USB_DIR_IN) &&
                ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) == =
USB_ENDPOINT_XFER_BULK)) {
                    /* we found a bulk in endpoint */
                    buffer_size = le16_to_cpu(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;
                    }
            }

            if (!dev->bulk_out_endpointAddr &&
                ((endpoint->bEndpointAddress & USB_ENDPOINT_DIR_MASK)=
=USB_DIR_OUT) &&
                    ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)= =
USB_ENDPOINT_XFER_BULK)) {
                    /* we found a bulk out endpoint */
                    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;
        }
```

Linux USB 驱动框架分析（五）

接下来的工作是向系统注册一些以后会用的的信息。首先我们来说明一下 usb_set_intfdata()，他向内核注册一个 data，这个 data 的结构能是任意的，这段程式向内核注册了一个 usb_skel 结构，就是我们刚刚看到的被初始化的那个，这个 data 能在以后用 usb_get_intfdata 来得到。

```
usb_set_intfdata(interface, dev);
retval = usb_register_dev(interface, &skel_class);
```

然后我们向这个 interface 注册一个 skel_class 结构。这个结构又是什么？我们就来看看这到底是个什么东西：

```
static struct usb_class_driver skel_class = {
        .name =          "skel%d",
        .fops =          &skel_fops,
        .minor_base = USB_SKEL_MINOR_BASE,
};
```

他其实是个系统定义的结构，里面包含了一名字、一个文件操作结构体更有一个次设备号的基准值。事实上他才是定义 真正完成对设备 IO 操作的函数。所以他的核心内容应该是 skel_fops。这里补充一些我个人的估计：因为 usb 设备能有多个 interface，每个 interface 所定义的 IO 操作可能不相同，所以向系统注册的 usb_class_driver 需求注册到某一个 interface，而不是 device，因此，usb_register_dev 的第一个参数才是 interface，而第二个参数就是某一个 usb_class_driver。通常情况下，linux 系统用主设备号来识别某类设备的驱动程式，用次设备号管理识别具体的设备，驱动程式能依照次设备号来区分不同的设备，所以，这里的次设备好其实是用来管理不同的 interface 的，但由于这个范例只有一个 interface，在代码上无法求证这个猜想。

```
static struct file_operations skel_fops = {
        .owner = THIS_MODULE,
        .read =             skel_read,
        .write =      skel_write,
        .open =             skel_open,
        .release =        skel_release,
};
```

这个文件操作结构中定义了对设备的读写、打开、释放（USB 设备通常使用这个术语 release）。他们都是函数指针，分别指向 skel_read、skel_write、skel_open、skel_release 这四个函数，这四个函数应该有研发人员自己实现。

当设备被拔出集线器时，usb 子系统会自动地调用 disconnect，他做的事情不多，最重要的是注销 class_driver（交还次设备号）和 interface 的 data:

```
dev = usb_get_intfdata(interface);
usb_set_intfdata(interface, NULL);

/* give back our minor */
usb_deregister_dev(interface, &skel_class);
```

然后他会用 kref_put(&dev->kref, skel_delete)进行清理，kref_put 的细节参见前文。

到目前为止，我们已分析完 usb 子系统需求的各个主要操作，下一部分我们在讨论一下对 USB 设备的 IO 操作。

Linux USB 驱动框架分析（六）

说到 usb 子系统的 IO 操作，不得不说 usb request block，简称 urb。事实上，能打一个这样的比喻，usb 总线就像一条高速公路，货物、人流之类的能看成是系统和设备交互的数据，而 urb 就能看成是汽车。在一开始对 USB 规范细节的介绍，我们就说过 USB 的 endpoint 有 4 种不同类型，也就是说能在这条高速公路上流动的数据就有四种。不过这对汽车是没有需求的，所以 urb 能运载四种数据，不过你要先告诉司机你要运什么，目的地是什么。我们目前就看看 struct urb 的具体内容。他的内容非常多，为了不让我的理解误导各位，大家最佳还是看一看内核源码的注释，具体内容参见源码树下 include/linux/usb.h。

在这里我们重点介绍程式中出现的几个关键字段：

```
struct usb_device    *dev
```

urb 所发送的目标设备。

```
unsigned int pipe
```

一个管道号码，该管道记录了目标设备的端点及管道的类型。每个管道只有一种类型和一个方向，他和他的目标设备的端点相对应，我们能通过以下几个函数来获得管道号并设置管道类型：

unsigned int usb_sndctrlpipe(struct usb_device *dev, unsigned int endpoint)

把指定 USB 设备的指定端点设置为一个控制 OUT 端点。

unsigned int usb_rcvctrlpipe(struct usb_device *dev, unsigned int endpoint)

把指定 USB 设备的指定端点设置为一个控制 IN 端点。

unsigned int usb_sndbulkpipe(struct usb_device *dev, unsigned int endpoint)

把指定 USB 设备的指定端点设置为一个批量 OUT 端点。

unsigned int usb_rcvbulkpipe(struct usb_device *dev, unsigned int endpoint)

把指定 USB 设备的指定端点设置为一个批量 OUT 端点。

unsigned int usb_sndintpipe(struct usb_device *dev, unsigned int endpoint)

把指定 USB 设备的指定端点设置为一个中断 OUT 端点。

unsigned int usb_rcvintpipe(struct usb_device *dev, unsigned int endpoint)

把指定 USB 设备的指定端点设置为一个中断 OUT 端点。

unsigned int usb_sndisocpipe(struct usb_device *dev, unsigned int endpoint)

把指定 USB 设备的指定端点设置为一个等时 OUT 端点。

unsigned int usb_rcvisocpipe(struct usb_device *dev, unsigned int endpoint)

把指定 USB 设备的指定端点设置为一个等时 OUT 端点。

unsigned int transfer_flags

当不使用 DMA 时，应该 transfer_flags |= URB_NO_TRANSFER_DMA_MAP（按照代码的理解，希望没有错）。

int status

当一个 urb 把数据送到设备时，这个 urb 会由系统返回给驱动程式，并调用驱动程式的 urb 完成回调函数处理。这时，status 记录了这次数据传输的有关状态，例如传送成功和否。成功的话会是 0。

要能够运货当然首先要有车，所以第一步当然要创建 urb：

struct urb *usb_alloc_urb(int isoc_packets, int mem_flags);

第一个参数是等时包的数量，如果不是乘载等时包，应该为 0，第二个参数和 kmalloc 的标志相同。

要释放一个 urb 能用：

void usb_free_urb(struct urb *urb);

要承载数据，还要告诉司机目的地信息跟要运的货物，对于不同的数据，系统提供了不同的函数，对于中断 urb，我们用

void usb_fill_int_urb(struct urb *urb, struct usb_device *dev, unsigned int pipe,
void *transfer_buffer, int buffer_length,
usb_complete_t complete, void *context, int interval);

这里要解释一下，transfer_buffer 是个要送/收的数据的缓冲，buffer_length 是他的长度，complete 是 urb 完成回调函数的入口，context 由用户定义，可能会在回调函数中使用的数据，interval 就是 urb 被调度的间隔。

对于批量 urb 和控制 urb，我们用:

void usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev, unsigned int pipe,
void *transfer_buffer, int buffer_length, usb_complete_t complete,
void *context);

void usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev, unsigned int pipe,
unsigned char* setup_packet,void *transfer_buffer,

```
                        int buffer_length, usb_complete_t complete,void *context);
```

控制包有一个特别参数 setup_packet，他指向即将被发送到端点的设置数据报的数据。

对于等时 urb，系统没有专门的 fill 函数，只能对各 urb 字段显示赋值。

有了汽车，有了司机，下一步就是要开始运货了，我们能用下面的函数来提交 urb

int usb_submit_urb(struct urb *urb, int mem_flags);

mem_flags 有几种：GFP_ATOMIC、GFP_NOIO、GFP_KERNEL，通常在中断上下文环境我们会用 GFP_ATOMIC。

当我们的卡车运货之后，系统会把他调回来，并调用 urb 完成回调函数，并把这辆车作为函数传递给驱动程式。我们应该在回调函数里面检查 status 字段，以确定数据的成功传输和否。下面是用 urb 来传送数据的细节。

```
/* initialize the urb properly */
usb_fill_bulk_urb(urb, dev->udev,
                        usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
                        buf, writesize, skel_write_bulk_callback, dev);
urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
/* send the data out the bulk port */
retval = usb_submit_urb(urb, GFP_KERNEL);
```

这里 skel_write_bulk_callback 就是个完成回调函数，而他做的主要事情就是检查数据传输状态和释放 urb：

```
dev = (struct usb_skel *)urb->context;
/* sync/async unlink faults aren't errors */
if (urb->status && !(urb->status = = -ENOENT || urb->status == -ECONNRESET || urb->status = = -ESHUTDOWN)) {
        dbg("%s - nonzero write bulk status received: %d", __FUNCTION__, urb->status);
}
/* free up our allocated buffer */
usb_buffer_free(urb->dev, urb->transfer_buffer_length,
                urb->transfer_buffer, urb->transfer_dma);
```

事实上，如果数据的量不大，那么能不一定用卡车来运货，系统还提供了一种不用 urb 的传输方式，而 usb-skeleton 的读操作正是采用这种方式实现：

```
/* do a blocking bulk read to get data from the device */
retval = usb_bulk_msg(dev->udev,
                        usb_rcvbulkpipe(dev->udev, dev->bulk_in_endpointAddr),
                        dev->bulk_in_buffer,
                        min(dev->bulk_in_size, count),
                        &bytes_read, 10000);
/* if the read was successful, copy the data to userspace */
if (!retval) {
        if (copy_to_user(buffer, dev->bulk_in_buffer, bytes_read))
                retval = -EFAULT;
        else
                retval = bytes_read;
}
```

程式使用了 usb_bulk_msg 来传送数据，他的原型如下：

int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,void *data,
int len, int *actual length, int timeout)

这个函数会阻塞等待数据传输完成或等到超时，data 是输入/输出缓冲，len 是他的大小，actual length 是实际传送的数据大小，timeout 是阻塞超时。

对于控制数据，系统提供了另外一个函数，他的原型是：

Int usb_contrl_msg(struct usb_device *dev, unsigned int pipe, __u8 request,
__u8 requesttype, __u16 value, __u16 index, void *data,
__u16 size, int timeout);

request 是控制消息的 USB 请求值、requesttype 是控制消息的 USB 请求类型，value 是控制消息的 USB 消息值，index 是控制消息的 USB 消息索引。具体是什么，暂时不是非常清晰，希望大家提供说明。

至此，Linux 下的 USB 驱动框架分析基本完成了。