# 硬件相关

## usb识别过程

1. 硬件检测

usb接口有vbus、GND、D+、D-四根线，主机端会将D+或D-下拉，也就是空载时为低电平，设备端则会将D+或D-上拉，则在设备接入主机时会将主机端的将D+或D-拉高，主机也就检测到了usb设备的接入。

1. 握手匹配

主机检测到usb设备接入后，主机端会发起获取设备信息的描述符，设备则需要按照固定格式返回描述符。

1. 分配地址

一个主机可以接入多个usb设备，为了区分这些设备，则会在握手成功后，主机会给设备分配设备地址，主机发送的命令中就会包含地址信息。在握手之前，主机会用编号0来和设备交互。

## usb数据传输

在usb上的数据信息的一次传输就是一个事务，事务包括输入事务、输出事务、设置事务等。每个事务又由令牌包、数据包(可选)、握手包组成：

* 令牌包：用于启动一个事务，总是由主机发送
* 数据包：可以从主机到设备，也可以从设备到主机，方向由令牌包决定
* 握手包：通常由数据的接收者发送握手包(ACK或者NAK)

usb协议中规定了4中传输类型：控制传输、批量传输、同步传输、中断传输。其中批量传输、同步传输、中断传输的每次传输都是一个事务，控制传输则包含三个过程，建立过程和状态过程分别是一个事务，数据过程则包含多个事务。

# usb驱动架构

usb控制器根据其作为主机端还是设备端分别为usb主机控制器(host controller)和usb设备控制器(UDC)。

usb设备驱动

Mass storage/HID

usb主机控

制器驱动

usb核心

主机

设备功能驱动

Mass storage/serial

usb设备控

制器驱动

Gadget驱动

设备

主机端驱动可以分为三层：

* usb设备驱动：用于控制设备与主机的通信方法，完成指定功能，包括mass storage、HID等
* usb核心：负责usb驱动的管理与协议的处理
* usb控制器驱动：用于完成与插入其中的usb设备的通信

设备端驱动同样分为三层：

* 功能驱动：实现了作为usb设备时的具体功能，如mass storage、HID等
* gadget驱动：用于隔离UDC驱动和功能驱动
* UDC驱动：用于硬件的控制以完成与主机的通信

注意usb设备驱动和设备端驱动的区别，设备驱动是为了使用插到主机上的从设备的功能，而设备端驱动则是要利用UDC将自己化身为一个usb设备，这个设备插到主机上后还需要一个与此设备匹配的设备驱动。

# usb控制器驱动

# usb设备驱动

## usb设备基础

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，每个设备可以包含一个或多个配置，不同的配置是不同功能即接口的组合，接口又由多个端点组成，端点就是usb通信的最基本单元。下图即为一个usb设备的基本组成，其中每个接口都对应着一个设备驱动。

usb driver

usb driver

usb driver

usb driver

endpoint

endpoint

endpoint

interface

endpoint

endpoint

endpoint

interface

config

endpoint

endpoint

endpoint

interface

endpoint

endpoint

endpoint

interface

config

usb

device

每个层次都通过描述符进行描述。一个usb设备只有一个设备描述符，设备描述符中定义了该设备有多少种配置，每种配置都对应着配置描述符，配置描述符中又定义了该配置里有多少个接口，每个接口都对应着接口描述符，接口描述符中又定义了该接口有多少个端点，每个端点都对应着端点描述符。

设备描述符：

struct usb\_device\_descriptor {

\_\_u8 bLength;

\_\_u8 bDescriptorType;

\_\_le16 bcdUSB;

\_\_u8 bDeviceClass;

\_\_u8 bDeviceSubClass;

\_\_u8 bDeviceProtocol;

\_\_u8 bMaxPacketSize0;

\_\_le16 idVendor;

\_\_le16 idProduct;

\_\_le16 bcdDevice;

\_\_u8 iManufacturer;

\_\_u8 iProduct;

\_\_u8 iSerialNumber;

\_\_u8 bNumConfigurations;

};

.bLength：描述符长度，即整个结构大小，可使用sizeof(struct usb\_device\_descriptor )或者使用USB\_DT\_DEVICE\_SIZE宏。

.bDescriptorType：描述符类型，这个描述符为设备描述符，应该为USB\_DT\_DEVICE。

.bcdUSB：usb协议版本号，0x0200为usb2.0，0x0110为usb1.1。

.bDeviceClass：设备类型编码，可用类型编码由USB\_CLASS\_\*一系列宏定义，包括人机交互类、音频类等，如果为0xff则表示该设备类型是厂家自定义的，为0则表示设备类型在接口描述符中定义。

.bDeviceSubClass：设备子类型编码，bDeviceClass为0时，此成员也应该为0。

.bDeviceProtocol：协议编码。

.bMaxPacketSize0：endpoint0最大包字节数。端点0没有专门的端点描述符，所以这个属性保存在设备描述符中，这个值的可选8/16/32/64之一，高速模式下为64，低速模式下为8。

.idVendor：厂商id。

.idProduct：设备id。

.bcdDevice：设备版本号。

.iManufacturer：厂商的字符串描述符的索引值。

.iProduct：产品的字符串描述符的索引值。

.iSerialNumber：产品序列号的字符串描述符的索引值。

.bNumConfigurations：设备的配置数量。

配置描述符：

struct usb\_config\_descriptor {

\_\_u8 bLength;//描述符长度，应该为USB\_DT\_CONFIG\_SIZE

\_\_u8 bDescriptorType;//描述符类型，应该为USB\_DT\_CONFIG

\_\_le16 wTotalLength;//使用GET\_DESCRIPTOR请求从设备中得到的包括配置、接口、端点、设备类型和厂家定义的描述的总厂

\_\_u8 bNumInterfaces;//此配置包含的接口数目

\_\_u8 bConfigurationValue;//对于含有多个配置的设备，作为SET\_CONFIGURATION请求的参数更改需要激活的配置，设备虽然可以有多个配置，但是同一时间只能有一个配置被激活

\_\_u8 iConfiguration;//描述配置信息的字符串描述符的索引值

\_\_u8 bmAttributes;//电源配置，使用USB\_CONFIG\_ATT\_\*系列宏定义，bit7必须为1，bit6为1代表self-powered，bit5为1代表支持远程唤醒

\_\_u8 bMaxPower;//设备从总线分得的最大电流，以2mA为单位，如hub上包含多个设备超过了hub所能提供的最大电流，此时hub就会拒绝设备

};

接口描述符：

struct usb\_interface\_descriptor {

\_\_u8 bLength;//描述符长度，应该为USB\_DT\_INTERFACE\_SIZE

\_\_u8 bDescriptorType;//描述符类型，应该为USB\_DT\_INTERFACE

\_\_u8 bInterfaceNumber;//接口编号

\_\_u8 bAlternateSetting;//备用接口描述符编号

\_\_u8 bNumEndpoints;//该接口包含的端点数量，不包含端点0

\_\_u8 bInterfaceClass;//类型编码

\_\_u8 bInterfaceSubClass;//子类型编码

\_\_u8 bInterfaceProtocol;//协议编码

\_\_u8 iInterface;//接口对应的字符串描述符索引

};

端点描述符：

struct usb\_endpoint\_descriptor {

\_\_u8 bLength;//描述符长度，非音频设备应该为USB\_DT\_ENDPOINT\_SIZE，音频设备为USB\_DT\_ENDPOINT\_AUDIO\_SIZE

\_\_u8 bDescriptorType;//描述符类型，应该为USB\_DT\_ENDPOINT

\_\_u8 bEndpointAddress;//bit0-3表示端点号，bit7表示方向（0-输出 1-输入）

\_\_u8 bmAttributes;//端点属性，bit0-1表示传输类型（00-控制传输 01-同步传输 10-批量传输 11-中断传输）

\_\_le16 wMaxPacketSize;//端点一次可处理的最大字节数

\_\_u8 bInterval;//轮询数据传输端点的时间间隔；批量传输端点以及控制传输端点，忽略此成员；同步传输端点，此成员必须为1；中断传输端点，此成员范围1-255

\_\_u8 bRefresh;

\_\_u8 bSynchAddress;

};

字符串描述符：

struct usb\_string\_descriptor {

\_\_u8 bLength;//描述符长度，由整个字符串+bLength和bDescriptorType长度决定

\_\_u8 bDescriptorType;//描述符类型，应该为USB\_DT\_STRING

\_\_le16 wData[1];//Unicode编码字符串

};

## 驱动注册

usb设备可以包含多个接口，每个接口代表一个功能，每个接口就对应着一个驱动：

struct usb\_interface {

struct usb\_host\_interface \*altsetting;//接口的可选设置

struct usb\_host\_interface \*cur\_altsetting; //接口当前使用的设置

unsigned num\_altsetting; //接口可选设置数目

struct usb\_interface\_assoc\_descriptor \*intf\_assoc;//与接口相关的描述符

int minor; //接口的次设备号

enum usb\_interface\_condition condition; //接口和驱动的绑定状态

unsigned sysfs\_files\_created:1; /\* the sysfs attributes exist \*/

unsigned ep\_devs\_created:1; /\* endpoint "devices" exist \*/

unsigned unregistering:1; /\* unregistration is in progress \*/

unsigned needs\_remote\_wakeup:1; /\* driver requires remote wakeup \*/

unsigned needs\_altsetting0:1; /\* switch to altsetting 0 is pending \*/

unsigned needs\_binding:1; /\* needs delayed unbind/rebind \*/

unsigned resetting\_device:1; /\* true: bandwidth alloc after reset \*/

unsigned authorized:1; /\* used for interface authorization \*/

struct device dev; /\* interface specific device info \*/

struct device \*usb\_dev;

atomic\_t pm\_usage\_cnt; //引用计数，计数为0时就会autosuspend

struct work\_struct reset\_ws; /\* for resets in atomic context \*/

};

接口的设置由如下结构表示：

struct usb\_host\_interface {

struct usb\_interface\_descriptor desc;//接口描述符

int extralen;//额外描述符长度

unsigned char \*extra; //额外描述符，厂家为设备特别定义的描述符

struct usb\_host\_endpoint \*endpoint;//此设置所使用的端点数组，数组大小为desc.bNumEndpoints

char \*string; //用来保存从设备中获取的字符串描述符信息

};

usb\_host\_endpoint结构描述了端点信息：

struct usb\_host\_endpoint {

struct usb\_endpoint\_descriptor desc;//端点描述符

struct usb\_ss\_ep\_comp\_descriptor ss\_ep\_comp;

struct usb\_ssp\_isoc\_ep\_comp\_descriptor ssp\_isoc\_ep\_comp;

struct list\_head urb\_list;//端点要处理的urb队列

void \*hcpriv;//提供给HCD（host controller driver）用的

struct ep\_device \*ep\_dev; //供sysfs使用

unsigned char \*extra;

int extralen;

int enabled;

int streams;

};

usb设备驱动通过如下结构描述：

struct usb\_driver {

const char \*name;

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

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

int (\*unlocked\_ioctl) (struct usb\_interface \*intf, unsigned int code, void \*buf);

int (\*suspend) (struct usb\_interface \*intf, pm\_message\_t message);

int (\*resume) (struct usb\_interface \*intf);

int (\*reset\_resume)(struct usb\_interface \*intf);

int (\*pre\_reset)(struct usb\_interface \*intf);

int (\*post\_reset)(struct usb\_interface \*intf);

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

struct usb\_dynids dynids;

struct usbdrv\_wrap drvwrap;

unsigned int no\_dynamic\_id:1;

unsigned int supports\_autosuspend:1;

unsigned int disable\_hub\_initiated\_lpm:1;

unsigned int soft\_unbind:1;

};

.name：驱动名称。

.probe：当设备和驱动匹配成功后执行。如果其可管理指定设备接口usb\_interface，则使用usb\_set\_intfdata()可将私有数据与接口联系起来。

.disconnect：当设备不可用如被拔出时执行。

.id\_table：用于匹配设备的列表，包含了此驱动可接受的usb设备，下面会详细介绍。

struct usb\_device\_id描述了usb设备的信息：

struct usb\_device\_id {

\_\_u16 match\_flags;

\_\_u16 idVendor;

\_\_u16 idProduct;

\_\_u16 bcdDevice\_lo;

\_\_u16 bcdDevice\_hi;

\_\_u8 bDeviceClass;

\_\_u8 bDeviceSubClass;

\_\_u8 bDeviceProtocol;

\_\_u8 bInterfaceClass;

\_\_u8 bInterfaceSubClass;

\_\_u8 bInterfaceProtocol;

\_\_u8 bInterfaceNumber;

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

};

.match\_flags：决定匹配设备时使用如下哪些成员，内核定义了USB\_DEVICE\_ID\_MATCH\_\*一系列宏来定义此标志，此成员无需直接设置，可利用USB\_DEVICE等一系列宏来统一定义此标志和所用成员。

.idVendor：设备的厂商id。供应商id是供应商向usb执行论坛申请的，每个供应商的vid是唯一的。

.idProduct：产品id，由供应商自己定义。

.bcdDevice\_lo/bcdDevice\_hi：由供应商提供的产品版本号，用于区分单个产品的不同版本。

.bDeviceClass/bDeviceSubClass/bDeviceProtocol：设备的类、子类和协议。

.bInterfaceClass/bInterfaceSubClass/bInterfaceProtocol/bInterfaceNumber：接口的类、子类和协议。

.driver\_info：用于保存驱动程序使用的信息。

对于usb\_device\_id的初始化内核已提供了很多宏来帮忙完成，包括USB\_DEVICE\_\*()/USB\_INTERFACE\_\*()等，都是用来初始化匹配标志以及对应的成员，如：

USB\_DEVICE(vend, prod) //用于匹配指定vid和pid的usb设备。

usb\_driver初始化完成后，可通过usb\_register()/usb\_deregister()完成驱动的注册与注销，当然也可使用如下宏统一完成驱动的注册与注销：

module\_usb\_driver();

## 设备通信

### 有urb

usb设备通信通过urb(USB Request Block)结构描述，urb是usb端点的处理对象：

struct urb {

//私有字段：只能由usb核心和主机控制器访问的字段

struct kref kref;

void \*hcpriv;

atomic\_t use\_count;

atomic\_t reject;

int unlinked;

//公共字段：可以被驱动使用的字段

struct list\_head urb\_list;

struct list\_head anchor\_list;

struct usb\_anchor \*anchor;

struct usb\_device \*dev;

struct usb\_host\_endpoint \*ep;

unsigned int pipe;

unsigned int stream\_id;

int status;

unsigned int transfer\_flags;

void \*transfer\_buffer;

dma\_addr\_t transfer\_dma;

struct scatterlist \*sg;

int num\_mapped\_sgs;

int num\_sgs;

u32 transfer\_buffer\_length;

u32 actual\_length;

unsigned char \*setup\_packet;

dma\_addr\_t setup\_dma;

int start\_frame;

int number\_of\_packets;

int interval;

int error\_count;

void \*context;

usb\_complete\_t complete;

struct usb\_iso\_packet\_descriptor iso\_frame\_desc[0];

};

.dev：指向与此urb关联的usb设备，urb被提交到usb核心前必须被usb设备驱动初始化。

.pipe：管道信息，包含了目标设备的端点以及管道类型和方向，urb被提交到usb核心前必须被usb设备驱动初始化，可借用如下函数进行初始化：

usb\_sndctrlpipe(dev, endpoint)//根据usb设备与指定端点创建一个发送控制类型的端点管道

usb\_rcvctrlpipe(dev, endpoint)//根据usb设备与指定端点创建一个接收控制类型的端点管道

usb\_sndisocpipe(dev, endpoint)//根据usb设备与指定端点创建一个发送同步类型的端点管道

usb\_rcvisocpipe(dev, endpoint)//根据usb设备与指定端点创建一个接收同步类型的端点管道

usb\_sndbulkpipe(dev, endpoint)//根据usb设备与指定端点创建一个发送批量类型的端点管道

usb\_rcvbulkpipe(dev, endpoint)//根据usb设备与指定端点创建一个接收批量类型的端点管道

usb\_sndintpipe(dev, endpoint)//根据usb设备与指定端点创建一个发送中断类型的端点管道

usb\_rcvintpipe(dev, endpoint)//根据usb设备与指定端点创建一个接收中断类型的端点管道

.transfer\_flags：传输标志，用于影响此urb如何被处理：

URB\_SHORT\_NOT\_OK：只对读urb起作用，表示IN端点上发生了短读，应该被usb核心当做错误

URB\_ISO\_ASAP：同步传输时使用

URB\_NO\_TRANSFER\_DMA\_MAP：表示urb包含dma传输缓冲，usb核心会使用transfer\_dma而不是transfer\_buffer指向的缓冲

URB\_NO\_SETUP\_DMA\_MAP：表示urb包含dma控制传输缓冲，usb核心会使用setup\_dma而不是setup\_packet指向的缓冲

URB\_ASYNC\_UNLINK：

URB\_NO\_FSBR：

URB\_ZERO\_PACKET：批量发送时必须使用short packet结束

URB\_NO\_INTERRUPT：

.transfer\_buffer：kmalloc分配的缓冲区

.transfer\_dma：dma缓冲区

.transfer\_buffer\_length：transfer\_buffer/transfer\_dma的长度

.setup\_packet：kmalloc分配的控制传输专用缓冲区

.setup\_dma：控制传输的dma缓冲

.actual\_length：实际发送或接收的字节数

.status：在urb回调函数中通过判断此成员获取urb执行状态。0-执行成功，-ENOENT-被usb\_kill\_urb()取消，-ECONNRESET-被usb\_unlink\_urb()取消，-EPROTO-传输过程中发生bitstuff错误或者硬件未能及时收到响应帧，-ENODEV-设备被移除。

urb的使用流程如下：

1. 创建urb

urb不能被静态创建，因为可能会破坏usb核心给urb使用的引用计数方法，必须使用如下函数创建：

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

.iso\_packets：同步数据包数目，如果不想创建同步urb则为0。

.mem\_flags：分配内存标志，和kmalloc()的分配标志含义相同。

urb使用完毕后需要释放：

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

1. 初始化urb，根据urb类型的不同使用不同的函数初始化
2. 中断urb

static inline 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\_fn, void \*context, int interval);

.urb：指向需要被初始化的urb。

.dev：urb要被发送到的usb设备。

.pipe：该urb要被发送到usb设备的特定端点，使用上面已经提到的usb\_sndintpipe()/usb\_rcvintpipe()创建相应管道。

.transfer\_buffer：指向缓冲区，不能是静态缓冲区，必须使用kmalloc()分配。

.buffer\_length：缓冲区的大小。

.complete\_fn：该urb完成时的回调函数。

.context：指向数据区，可在回调函数中获取。

.interval：这个urb应该被调度的间隔。

1. 批量urb

static inline 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\_fn, void \*context);

pipe需要使用usb\_sndbulkpipe()/usb\_rcvbulkpipe()来创建，其他参数与usb\_fill\_int\_urb()参数相同。

1. 控制urb

static inline void usb\_fill\_control\_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\_fn, void \*context);

.setup\_packet：即将被发送到端点的设置数据包。

pipe需要使用usb\_sndctrlpipe()/usb\_rcvictrlpipe()来创建，其他参数与usb\_fill\_int\_urb()参数相同。

1. 同步urb

同步urb没有初始化函数，需要自己完成各成员的赋值，可在内核代码中搜索usb\_rcvisocpipe()或usb\_sndisocpipe()就能找到初始化同步urb的例子。

1. 提交urb

urb创建和初始化完成后便可被提交到usb核心：

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

.mem\_flags：与kmalloc()参数意义相同，用于告诉usb核心此时如何分配内存缓冲区。因为

usb\_submit\_urb()可被在任何时候包括中断上下文中被调用，所以mem\_flags必须根据当前调用环境正 确设置，不同值的调用环境如下：

GFP\_ATOMIC：当调用者处于中断处理函数、底半部、tasklet、定时器处理函数或者urb完成函数中，或者调用者持有自旋锁或者读写锁以及当驱动将current->state改为非TASK\_RUNNING时，应该使用此标志。

GFP\_NOIO：在存储设备的块I/O或者错误处理路径中，应使用此标志。

GFP\_KERNEL：其他情况使用此标志。

1. 完成urb

如下三种情况下，urb将结束并且完成函数被调用，我们需要判断urb->status已确认urb是否执行成功：

* urb被成功发送给设备并且设备返回正确的确认，对于输出urb则数据成功发送，对于输入urb则数据成功接收，此时urb->status为0。
* 在发送或者接收数据时发生错误，urb->status将记录错误原因。
* urb被从usb核心中去链，这可能发生在驱动调用usb\_unlink\_urb()/usb\_kill\_urb()或者usb设备被移除时。

1. 取消urb

为停止一个已经提交到usb核心的urb，需要调用如下两个函数：

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

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

usb\_unlink\_urb()是异步的，而usb\_kill\_urb()则会等待其完成，其通常在usb设备的disconnect()中被调用。

### 无urb

当我们想要对设备完成单次的读写，没有必要再完成上述对urb的所有过程，如下两个函数可以帮助我们完成所有工作，当然其内部也就是对上述urb过程的封装。

创建一个批量urb发送到设备，然后等待urb完成：

int usb\_bulk\_msg(struct usb\_device \*usb\_dev, unsigned int pipe, void \*data, int len, int \*actual\_length, int timeout);

.usb\_dev：接收此消息的usb设备

.pipe：使用usb\_sndbulkpipe()/usb\_rcvbulkpipe()创建的管道

.data：发送或接收数据的缓冲区

.len：缓冲区长度

.actual\_length：实际发送或接收的字节数

.timeout：发送超时，以ms为单位，为0则永远等待

返回值：成功返回0，否则返回错误号，含义与urb结构的status成员含义相同

创建一个控制urb发送到设备，然后等待urb完成：

int usb\_control\_msg(struct usb\_device \*dev, unsigned int pipe, \_\_u8 request, \_\_u8

requesttype, \_\_u16 value, \_\_u16 index, void \*data, \_\_u16 size, int timeout);

.pipe：使用usb\_sndctrlpipe()/usb\_rcvictrlpipe()创建的管道

.request：此控制消息的请求值

.requesttype：此控制消息的请求类型

.value：此控制消息的消息值

.index：此控制消息的消息索引值，关于request/requesttype/value/index可参照usb规范的第九章中定义的主机请求定义

.data：发送或接收数据的缓冲区

.size：缓冲区长度

.timeout：发送超时，以ms为单位，为0则永远等待

注意因为以上两个函数是同步的，所以不能在中断上下文以及持有自旋锁的情况下使用。并且这两个函数也不能像取消urb那样取消掉。

## usb skeleton

usb-skeleton.c为我们提供了一个最简单的usb设备驱动，我们在编写usb设备驱动时可以参照此驱动完成。usb-skeleton.c为应用层提供了字符设备接口可用于设备的读写操作，想使用此驱动需要将驱动编入内核或者编成模块，drivers/usb/目录下的Makefile和Kconfig添加如下：

diff --git a/drivers/usb/Kconfig b/drivers/usb/Kconfig

index 644e978c..535c7ac6 100644

--- a/drivers/usb/Kconfig

+++ b/drivers/usb/Kconfig

+config USB\_SKELETON

+ tristate "USB skeleton driver"

+ default m

diff --git a/drivers/usb/Makefile b/drivers/usb/Makefile

index dca78565..6b96545b 100644

--- a/drivers/usb/Makefile

+++ b/drivers/usb/Makefile

+obj-$(CONFIG\_USB\_SKELETON) += usb-skeleton.o

usb-skeleton.c中的设备列表skel\_table[]也需要更改，需要将想要加载此驱动的设备信息加入此列表。此后如果

设备插入，驱动就会创建/dev/skelX设备，我们就可以利用此字符设备完成对usb设备的读写操作。

# usb设备端驱动

## 驱动架构

设备端驱动可以进一步细化如下，其中compsite framwork提供了通用的gadget驱动，lagacy则是具体的复合设备驱动，复合设备可以包含一个或多个function驱动提供的功能。

UDC驱动

gadget API

compsite framwork

legacy

functions

hid

serial

ether

所以想要完成一个usb设备端驱动只需要关心legacy和function，我们可以仿照已有的legacy将功能进行组合，当然如果function中不具有我们需要的功能，则需要实现function驱动。

## udc驱动

### 设备树配置

需要将usb设置为设备模式：

&usb1 {

status = "okay";

dr\_mode = "peripheral";

};

### 驱动注册

usb设备控制器(UDC)驱动指的是作为其他USB主机控制器的USB硬件设备上的底层控制器驱动，该硬件和驱动负责将一个usb设备依附于一个usb主机控制器上。usb设备控制器通过如下结构描述：

struct usb\_udc {

struct usb\_gadget\_driver \*driver;//udc设备驱动，指向Composite驱动中的usb\_gadget\_driver

struct usb\_gadget \*gadget;//udc设备

struct device dev;

struct list\_head list;

bool vbus;

};

usb\_gadget结构描述了一个udc设备：

struct usb\_gadget {

struct work\_struct work;//sysfs\_notify()使用的工作队列

struct usb\_udc \*udc;//指向usb设备控制器

const struct usb\_gadget\_ops \*ops;//udc设备操作函数集

struct usb\_ep \*ep0;//端点0，用于处理setup()请求

struct list\_head ep\_list; //除端点0外的端点链表

enum usb\_device\_speed speed;//当前连接的usb主机速度

enum usb\_device\_speed max\_speed;//udc能处理的最大速度

enum usb\_device\_state state;//udc状态

const char \*name;

struct device dev;

unsigned out\_epnum;

unsigned in\_epnum;

unsigned mA;

struct usb\_otg\_caps \*otg\_caps;

unsigned sg\_supported:1;

unsigned is\_otg:1;

unsigned is\_a\_peripheral:1;

unsigned b\_hnp\_enable:1;

unsigned a\_hnp\_support:1;

unsigned a\_alt\_hnp\_support:1;

unsigned hnp\_polling\_support:1;

unsigned host\_request\_flag:1;

unsigned quirk\_ep\_out\_aligned\_size:1;

unsigned quirk\_altset\_not\_supp:1;

unsigned quirk\_stall\_not\_supp:1;

unsigned quirk\_zlp\_not\_supp:1;

unsigned quirk\_avoids\_skb\_reserve:1;

unsigned is\_selfpowered:1;

unsigned deactivated:1;

unsigned connected:1;

};

udc设备操作函数集如下：

struct usb\_gadget\_ops {

int (\*get\_frame)(struct usb\_gadget \*);

int (\*wakeup)(struct usb\_gadget \*);

int (\*set\_selfpowered) (struct usb\_gadget \*, int is\_selfpowered);

int (\*vbus\_session) (struct usb\_gadget \*, int is\_active);

int (\*vbus\_draw) (struct usb\_gadget \*, unsigned mA);

int (\*pullup) (struct usb\_gadget \*, int is\_on);

int (\*ioctl)(struct usb\_gadget \*, unsigned code, unsigned long param);

void (\*get\_config\_params)(struct usb\_dcd\_config\_params \*);

int (\*udc\_start)(struct usb\_gadget \*, struct usb\_gadget\_driver \*);

int (\*udc\_stop)(struct usb\_gadget \*);

struct usb\_ep \*(\*match\_ep)(struct usb\_gadget \*, struct usb\_endpoint\_descriptor \*, struct usb\_ss\_ep\_comp\_descriptor \*);

};

设备端的usb端点由如下结构描述：

struct usb\_ep {

void \*driver\_data;

const char \*name;

const struct usb\_ep\_ops \*ops;//端点的操作函数集

struct list\_head ep\_list;

struct usb\_ep\_caps caps;

bool claimed;

bool enabled;

unsigned maxpacket:16;

unsigned maxpacket\_limit:16;

unsigned max\_streams:16;

unsigned mult:2;

unsigned maxburst:5;

u8 address;

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

const struct usb\_ss\_ep\_comp\_descriptor \*comp\_desc;

};

端点的操作函数集如下：

struct usb\_ep\_ops {

int (\*enable) (struct usb\_ep \*ep, const struct usb\_endpoint\_descriptor \*desc);

int (\*disable) (struct usb\_ep \*ep);

struct usb\_request \*(\*alloc\_request) (struct usb\_ep \*ep, gfp\_t gfp\_flags);

void (\*free\_request) (struct usb\_ep \*ep, struct usb\_request \*req);

int (\*queue) (struct usb\_ep \*ep, struct usb\_request \*req, gfp\_t gfp\_flags);

int (\*dequeue) (struct usb\_ep \*ep, struct usb\_request \*req);

int (\*set\_halt) (struct usb\_ep \*ep, int value);

int (\*set\_wedge) (struct usb\_ep \*ep);

int (\*fifo\_status) (struct usb\_ep \*ep);

void (\*fifo\_flush) (struct usb\_ep \*ep);

};

udc设备驱动由如下结构描述：

struct usb\_gadget\_driver {

char \*function;//描述此驱动的字符串

enum usb\_device\_speed max\_speed;//驱动可处理的最大速度

int (\*bind)(struct usb\_gadget \*gadget,struct usb\_gadget\_driver \*driver);

void (\*unbind)(struct usb\_gadget \*);

int (\*setup)(struct usb\_gadget \*, const struct usb\_ctrlrequest \*);

void (\*disconnect)(struct usb\_gadget \*);

void (\*suspend)(struct usb\_gadget \*);

void (\*resume)(struct usb\_gadget \*);

void (\*reset)(struct usb\_gadget \*);

struct device\_driver driver;

char \*udc\_name;

struct list\_head pending;

unsigned match\_existing\_only:1;

};

usb\_gadget结构初始化后就可注册udc驱动，其会分配usb\_udc结构并初始化相关成员，最后会将usb\_udc挂

到全局的udc\_list上：

int usb\_add\_gadget\_udc(struct device \*parent, struct usb\_gadget \*gadget);

对应的释放函数为：

void usb\_del\_gadget\_udc(struct usb\_gadget \*gadget);

Composite驱动调用usb\_gadget\_probe\_driver和UDC驱动匹配，首先遍历udc\_list链表，若有usb\_udc的driver成员为空，则表示匹配成功，接着Composite驱动和UDC驱动绑定，通过将Composite驱动的usb\_composite\_driver.gadget\_driver的地址设置到usb\_udc.driver成员中完成绑定，最后回调udc\_start启动USB设备控制器。调用usb\_gadget\_unregister\_driver解除Composite驱动和UDC驱动的绑定关系。

## Composite驱动

### 驱动注册

usb composite（复合）设备指的是支持多个function的gadget设备，使用usb\_composite\_dev结构表示，该结构会在composite驱动注册时在进行驱动绑定时自动创建（composite\_driver\_template.composite\_bind）：

struct usb\_composite\_dev {

struct usb\_gadget \*gadget;

struct usb\_request \*req;

struct usb\_request \*os\_desc\_req;

struct usb\_configuration \*config;

u8 qw\_sign[OS\_STRING\_QW\_SIGN\_LEN];

u8 b\_vendor\_code;

struct usb\_configuration \*os\_desc\_config;

unsigned int use\_os\_string:1;

unsigned int suspended:1;

struct usb\_device\_descriptor desc;

struct list\_head configs;

struct list\_head gstrings;

struct usb\_composite\_driver \*driver;

u8 next\_string\_id;

char \*def\_manufacturer;

unsigned deactivations;

int delayed\_status;

spinlock\_t lock;

unsigned setup\_pending:1;

unsigned os\_desc\_pending:1;

};

usb\_composite\_driver结构则描述了composite驱动：

struct usb\_composite\_driver {

const char \*name;//驱动名称

const struct usb\_device\_descriptor \*dev;//设备描述符，必须定义

struct usb\_gadget\_strings \*\*strings;//字符串列表，bind中会为设备、功能等分配id，即字符串的索引值，如将设备描述符的iProduct设置为1，则字符串即为id为1的对应字符串

enum usb\_device\_speed max\_speed;//设备支持的最大速度

unsigned needs\_serial:1;

int (\*bind)(struct usb\_composite\_dev \*cdev);//用于分配整个设备共享的资源，使用usb\_add\_config添加配置

int (\*unbind)(struct usb\_composite\_dev \*);//用于释放资源

void (\*disconnect)(struct usb\_composite\_dev \*);

void (\*suspend)(struct usb\_composite\_dev \*);

void (\*resume)(struct usb\_composite\_dev \*);

struct usb\_gadget\_driver gadget\_driver;//内核已经实现，驱动注册时会被赋值为composite\_driver\_template

};

usb\_composite\_driver是我们在完成设备驱动时需要实现的重点，实现完成后就可以进行驱动的注册：

module\_usb\_composite\_driver(\_\_usb\_composite\_driver);

### legacy

drivers/usb/gadget/legacy/目录下已经实现了很多的composite驱动，完成了常用的usb设备，如存储、串口等。如果我们需要实现一个内核未能提供的设备驱动，其实我们可以参照此目录下的驱动添加usb设备描述符信息以及若干配置实现自己的驱动。

#### zero

与zero相关的有两个功能文件：f\_loopback.c f\_sourcesink.c。在这两个文件中，各实现了一个configuration，因此，g\_zero设备就有两个configuration。其中，loopback用于将host发送过来的数据返给host，sourcesink则仅仅读取host发过来的数据，不作保存。

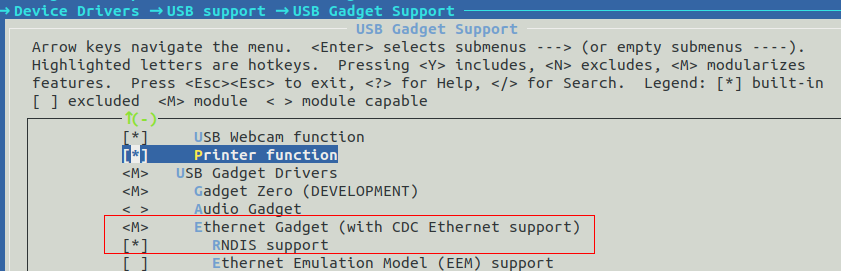
#### mass storage

insmod g\_mass\_storage.ko file=/dev/mmcblk0p1 removable=1

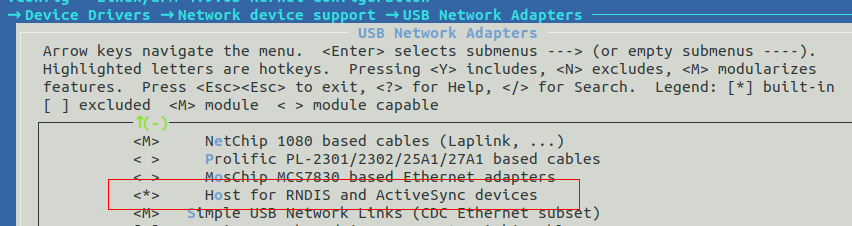
此时插到window上，就会被识别出一个存储设备。

#### ethernet

设备端内核配置：

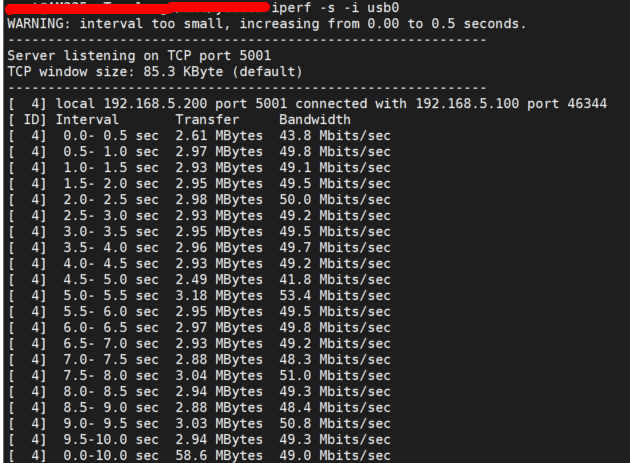


主机端内核配置：



此时insmod g\_ether.ko，无论是主机端还是设备端都会虚拟出usb0网络设备，设置ip地址后可通过iperf测试，

但是速度只能到50Mbits/sec，远远达不到高速usb的速度，我是用两个am3352板卡一主一从测试，不知道速度受限的原因是什么。



如果插到windows下，pc会识别出COM口，此时需要更改驱动，解决可参考：

<https://www.mobileread.com/forums/showthread.php?p=3283986>

解决过程就是：

* Download & Unzip attachment kindle\_rndis.inf\_amd64-v1.0.0.1.zip
* R-click "5-runasadmin\_register-CA-cer.cmd" and "Run as administrator"\*
* In Device Manager, expand "Ports (COM & LPT)", R-click "Serial USB device (COM3)" > Update Driver Software...
* Browse for my computer for driver software > Select extract folder

此后设备管理器下可以看到此设备：



同时网络连接下也能看到新的网卡：



此时将网卡设置与嵌入式中usb0的ip相同网段的ip就可实现PC和嵌入式之间的网络通信。

## Function驱动

### 驱动注册

drivers/usb/gadget/function/目录下提供了各种usb子类设备的功能驱动，Function驱动通过如下结构表示：

struct usb\_function\_driver {

const char \*name;

struct module \*mod;

struct list\_head list;

struct usb\_function\_instance \*(\*alloc\_inst)(void);

struct usb\_function \*(\*alloc\_func)(struct usb\_function\_instance \*inst);

};

驱动中需要实现alloc\_func和alloc\_inst，alloc\_inst创建usb\_function\_instance数据结构并初始化，alloc\_func创建usb\_function并初始化，重点是设置里面的回调函数。

usb\_function\_driver通常使用DECLARE\_USB\_FUNCTION(\_name, \_inst\_alloc, \_func\_alloc)宏定义并初始化，其会定义一个名为\_name+usb\_func的usb\_function\_driver结构，此时可通过如下两个函数完成Function驱动的注册与注销：

int usb\_function\_register(struct usb\_function\_driver \*newf);

void usb\_function\_unregister(struct usb\_function\_driver \*fd);

实现usb\_function中的回调函数才是实现Function驱动的重点，此结构描述了一个配置的一个功能：

struct usb\_function {

const char \*name;//功能名称

struct usb\_gadget\_strings \*\*strings;

struct usb\_descriptor\_header \*\*fs\_descriptors;

struct usb\_descriptor\_header \*\*hs\_descriptors;

struct usb\_descriptor\_header \*\*ss\_descriptors;

struct usb\_descriptor\_header \*\*ssp\_descriptors;

struct usb\_configuration \*config;

struct usb\_os\_desc\_table \*os\_desc\_table;

unsigned os\_desc\_n;

int (\*bind)(struct usb\_configuration \*, struct usb\_function \*);

void (\*unbind)(struct usb\_configuration \*, struct usb\_function \*);

void (\*free\_func)(struct usb\_function \*f);

struct module \*mod;

int (\*set\_alt)(struct usb\_function \*, unsigned interface, unsigned alt);

int (\*get\_alt)(struct usb\_function \*, unsigned interface);

void (\*disable)(struct usb\_function \*);

int (\*setup)(struct usb\_function \*, const struct usb\_ctrlrequest \*);

bool (\*req\_match)(struct usb\_function \*, const struct usb\_ctrlrequest \*, bool config0);

void (\*suspend)(struct usb\_function \*);

void (\*resume)(struct usb\_function \*);

int (\*get\_status)(struct usb\_function \*);

int (\*func\_suspend)(struct usb\_function \*, u8 suspend\_opt);

struct list\_head list;

DECLARE\_BITMAP(endpoints, 32);

const struct usb\_function\_instance \*fi;

unsigned int bind\_deactivated:1;

};

## API说明

设备相关：

向设备添加配置：

int usb\_add\_config\_only(struct usb\_composite\_dev \*cdev, struct usb\_configuration

\*config);

向配置添加功能：

int usb\_add\_function(struct usb\_configuration \*config, struct usb\_function

\*function);

重置所有端点自动配置状态：

void usb\_ep\_autoconfig\_reset (struct usb\_gadget \*gadget);

下面这个函数可以同时完成上面三个函数的功能，bind为向设备添加完配置后执行的函数，一般在此函数中完成usb\_add\_function()：

int usb\_add\_config(struct usb\_composite\_dev \*cdev, struct usb\_configuration

\*config, int (\*bind)(struct usb\_configuration \*));

端点控制：

根据端点描述符分配端点：

struct usb\_ep \*usb\_ep\_autoconfig(struct usb\_gadget \*gadget, struct

usb\_endpoint\_descriptor \*desc);

使能端点：

int usb\_ep\_enable(struct usb\_ep \*ep);

禁用端点：

int usb\_ep\_disable(struct usb\_ep \*ep);

向端点提交一个I/O请求：

int usb\_ep\_queue(struct usb\_ep \*ep, struct usb\_request \*req, gfp\_t gfp\_flags);

从端点取消一个I/O请求：

int usb\_ep\_dequeue(struct usb\_ep \*ep, struct usb\_request \*req);

分配usb\_request：

struct usb\_request \*usb\_ep\_alloc\_request(struct usb\_ep \*ep, gfp\_t gfp\_flags);

释放usb\_request：

void usb\_ep\_free\_request(struct usb\_ep \*ep, struct usb\_request \*req);

根据端点描述符，匹配要使用的端点

int usb\_gadget\_ep\_match\_desc(struct usb\_gadget \*gadget, struct usb\_ep \*ep, struct

usb\_endpoint\_descriptor \*desc, struct usb\_ss\_ep\_comp\_descriptor \*ep\_comp);

功能相关：

为功能赋值不同速度下的描述符：

int usb\_assign\_descriptors(struct usb\_function \*f, struct usb\_descriptor\_header

\*\*fs, struct usb\_descriptor\_header \*\*hs, struct usb\_descriptor\_header \*\*ss, struct usb\_descriptor\_header \*\*ssp);

## 数据传输

设备端的数据传输也有类似urb的数据结构urt--usb\_request：

struct usb\_request {

void \*buf;

unsigned length;

dma\_addr\_t dma;

struct scatterlist \*sg;

unsigned num\_sgs;

unsigned num\_mapped\_sgs;

unsigned stream\_id:16;

unsigned no\_interrupt:1;

unsigned zero:1;

unsigned short\_not\_ok:1;

void (\*complete)(struct usb\_ep \*ep, struct usb\_request \*req);

void \*context;

struct list\_head list;

int status;

unsigned actual;

};

.buf：数据缓冲区，用于存储接收或者发送的数据。

.length：数据长度。

.dma：buf对应的dma地址。

.zero：为true时在写入数据时最后会发送一个short packet。

.short\_not\_ok：读取数据时，将短数据包视为错误。

.complete：请求完成后执行的回调，回调中不能睡眠。

.context：供回调函数使用的参数。

.status：在回调函数中通过判断此成员获取请求执行状态。0-执行成功，-ECONNRESET-请求被取消，-ESHUTDOWN-设备被移除。

.actual：实际传输的字节数。

urt的使用流程如下：

1. 创建urt

urt的创建使用如下函数完成：

struct usb\_request \*usb\_ep\_alloc\_request(struct usb\_ep \*ep, gfp\_t gfp\_flags);

.ep：此请求使用的端点

.gfp\_flags：分配内存标志，和kmalloc()的分配标志含义相同。

相应的释放函数为：

void usb\_ep\_free\_request(struct usb\_ep \*ep, struct usb\_request \*req);

1. 初始化urt

需要初始化urt的buf和length，buf需要通过kmalloc申请。

当然内核也提供了函数能同时完成urt的创建于初始化：

struct usb\_request \*alloc\_ep\_req(struct usb\_ep \*ep, size\_t len);

ep就是请求使用的端点，len就是传输长度。

1. 提交urt

提交urt其实就是将请求添加到端点的队列中：

int usb\_ep\_queue(struct usb\_ep \*ep, struct usb\_request \*req, gfp\_t gfp\_flags);

1. 完成urt

当urt执行成功、被取消、设备被移除、执行失败都会导致回调函数被执行，需要在回调函数中通过urt->status判断urt的执行状态。

1. 取消urt

可以将已经加入端点队列的urt取消掉：

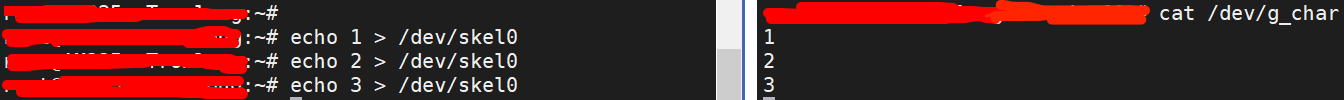
int usb\_ep\_dequeue(struct usb\_ep \*ep, struct usb\_request \*req);

# 示例说明

fake\_mouse.c实现了一个假的鼠标驱动，并未注册input设备，也并未上报鼠标按键、坐标和滚轮的变化信息，只是在点击鼠标左右键或者滚轮键时打印了信息，目的只是演示urb的使用过程。

g\_char.c则实现了一个usb gadget，用户层可利用/dev/g\_char字符设备在用户层完成usb的读写操作。我使用两块嵌入式板卡，一个作为host，一个作为peripheral，peripheral端insmod g\_char.ko，host则借助usb skeleton驱动 insmod usb-skeleton.ko，主从直接可完成读写操作：

主写从读：



主读从写：

