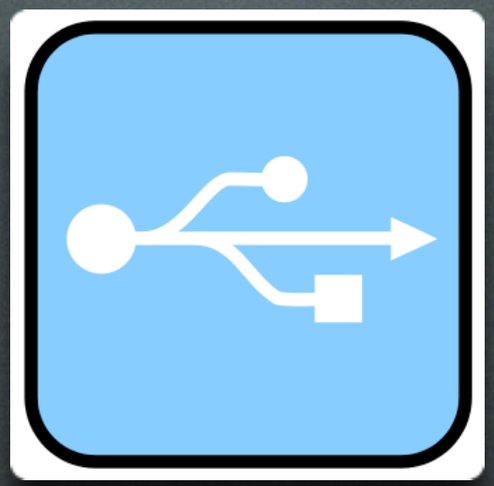
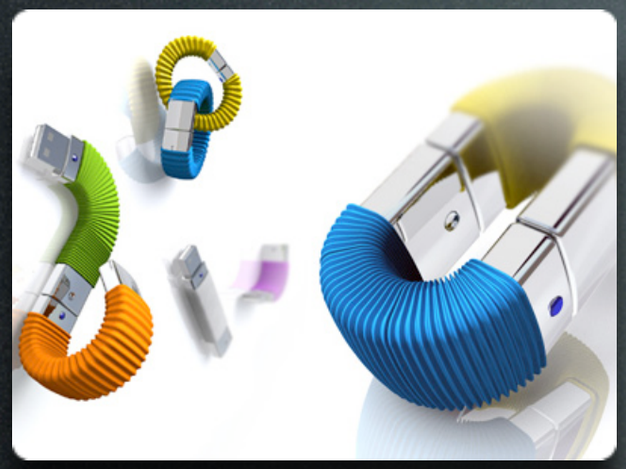
USB



Funcionamiento en Linux

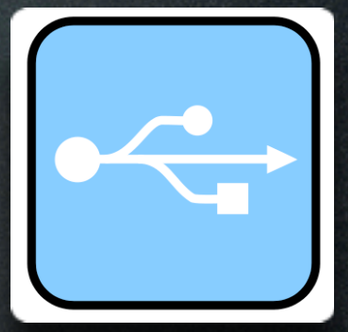
Indice

- ¿Qué es un dispositivo USB?
- Fundamentos de los dispositivos USB
- USB y Sysfs(sistema de archivos virtual)
- USB urbs

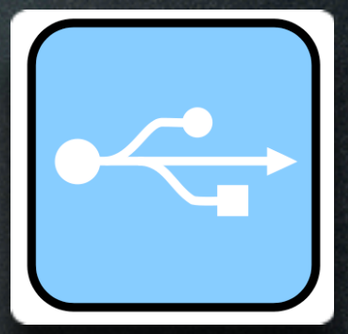


Dispositivos USB

- Conexión más sencilla
- Plug and Play
- Hot Pluggable



- Mayor rendimiento
- Soporte multiplataforma
- Simultaneidad de dispositivos conectados



- Un dispositivo USB es algo muy complejo (http://www.usb.org).
- Linux nos provee del USB core, es una API para dispositivos USB y controladores

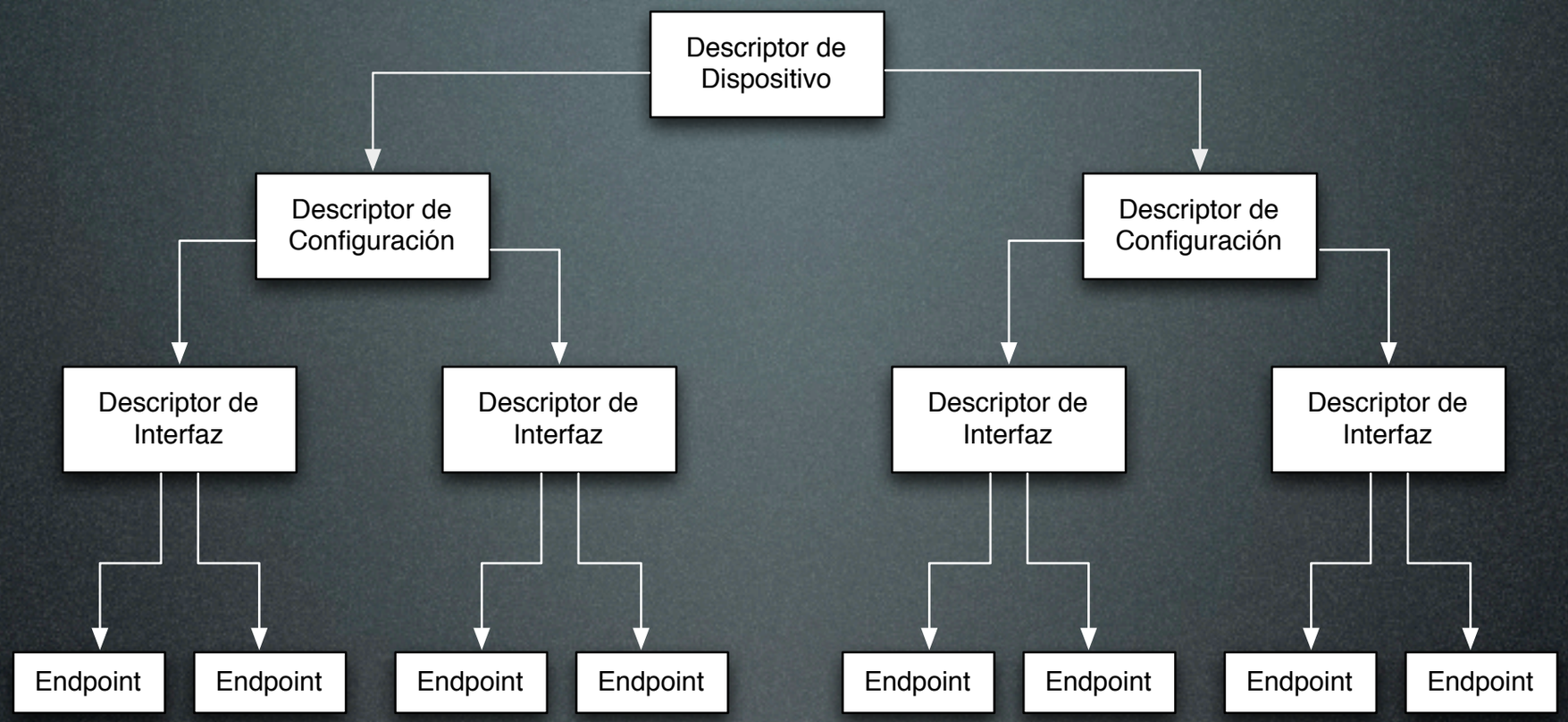
Fundamentos de los dispositivos USB

Fundamentos de los dispositivos USB

- Endpoints
  - ¿Qué son y para que se utilizan?
- Interfaz
  - ¿Qué es y para que se utilizan?
- Configuración
  - ¿cómo configuramos un USB?

Endpoints

- Un Endpoint puede verse como la interfaz entre el hardware y el firmware del dispositivo.
- Los endpoints pueden ser descritos como fuentes o sumideros, de ellos salen y entran datos.
- Todos los dispositivos poseen un endpoint 0. Este es el que recibe el control y las peticiones de estado del dispositivo durante la enumeración.



- Un dispositivo USB hay que entenderlo como una serie de endpoints, a su vez los endpoints se agrupan en conjuntos que dan lugar a interfaces, las cuales permiten controlar la función del dispositivo.

- Las transferencias de datos que salen y entran de los endpoints pueden ser de 4 tipos:
  - Transferencias de control
  - Transferencias síncronas
  - Transferencias de interrupción
  - Transferencias de bultos ("Bulk")

- Transferencias de control:
  - configurar
  - obtener información
  - manipular el estado de los dispositivos.

- Transferencias síncronas:
  - Dispositivos con velocidad alta o media
  - Grandes cantidades de datos
  - Aseguran flujo de datos constante

- Transferencias de interrupción:
  - Pequeñas cantidades de datos
  - Aseguran un flujo de datos costante
  - Si falla el envio de un dato, se reenvia de nuevo

- Transferencias de bultos ("Bulk")
  - Envío de grandes cantidades de datos
  - No importa que no lleguen los datos
  - Usa todo el ancho de banda

- Los endpoints están descritos en: struct usb_host_endpoint

```
struct usb_host_endpoint {
        struct usb_endpoint_descriptor  desc;
        struct list_head                urb_list;
        void                           *hcpriv;
        struct ep_device                *ep_dev;        /* For sysfs info */

        unsigned char *extra;   /* Extra descriptors */
        int extralen;
};
```

- La información del endpoint la contiene: struct usb_endpoint_descriptor.

```
struct usb_endpoint_descriptor {
        __u8  bLength;
        __u8  bDescriptorType;

        __u8  bEndpointAddress;
        __u8  bmAttributes;
        __le16 wMaxPacketSize;
        __u8  bInterval;

        /* NOTE:  these two are _only_ in audio endpoints. */
        /* use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof. */
        __u8  bRefresh;
        __u8  bSynchAddress;
}
```

- Los campos de usb_endpoint_descriptor:
  - **bEndpointAddress** -> Dirección USB del endpoint
  - **bmAttributes** -> El tipo del endpoint
  - **wMaxPacketSiz**e -> Tamaño máximo del endpoint
  - **bInterva**l -> El tiempo transcurrido entre las solicitudes de interrumpir el endpoint

Interfaz

- Manejan sólo un tipo de conexión lógica USB.
- Algunos dispositivos tienen múltiples interfaces.
- Pueden tener configuraciones alternativas.
- El estado inicial es la configuración 0.

- se describen en el kernel con: struct usb_interface

```
struct usb_interface {
        /* array of alternate settings for this interface,
         * stored in no particular order */
        struct usb_host_interface *altsetting;

        struct usb_host_interface *cur_altsetting;      /* the currently
                                         * active alternate setting */
        unsigned num_altsetting;        /* number of alternate settings */

        /* If there is an interface association descriptor then it will list
         * the associated interfaces */
        struct usb_interface_assoc_descriptor *intf_assoc;

        int minor;                      /* minor number this interface is
                                         * bound to */
        enum usb_interface_condition condition;         /* state of binding */
        unsigned is_active:1;           /* the interface is not suspended */
        unsigned needs_remote_wakeup:1; /* driver requires remote wakeup */

        struct device dev;              /* interface specific device info */
        struct device *usb_dev;         /* pointer to the usb class's device, if any */
        int pm_usage_cnt;               /* usage counter for autosuspend */
};
```

- Los campos de usb_interface:
  - **struct usb_host_interface *altsetting** -> Estructuras que contienen las configuraciones alternativas de la interfaz.
  - **unsigned num_altsetting** -> El número de configuraciones alternativas.
  - **struct usb_host_interface *cur_altsetting** -> Apunta la configuración activa de la interfaz
  - **int minor** -> Número asignado por el kernel a la interfaz.

Configuraciones

- Un dispositivo puede tener varias configuraciones.
- Puede cambiar entre ellas con el fin de cambiar el estado del dispositivo.
- Sólo una configuración puede estar activa a la vez.

- Se describen las configuraciones con: struct usb_host_config.
- Se describen los dispositivos con: struct usb_device.

- Un driver de dispositivo Usb tiene que hacer una conversión de estructuras. usb_interface --> usb_device
- Para ello disponemos de la función interface_to_usbdev

Para resumir

- Los dispositivos suelen tener una o más configuraciones.
- Las configuraciones suelen tener una o más interfaces.
- Las interfaces generalmente, presentan una o más configuraciones.
- Las interfaces pueden tener varios endpoints o ninguno.

USB y Sysfs

- Sysfs es un sistema de archivos virtual que proporciona el núcleo Linux v2.6.
- Sysfs exporta información sobre los dispositivos y sus controladores desde el modelo de dispositivos del núcleo hacia el espacio del usuario, también permite configurar parámetros.

Arbol Sysfs para un dispositivo

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1
|-- 2-1:1.0
|   |-- bAlternateSetting
|   |-- bInterfaceClass
|   |-- bInterfaceNumber
|   |-- bInterfaceProtocol
|   |-- bInterfaceSubClass
|   |-- bNumEndpoints
|   |-- detach_state
|   |-- iInterface
|   `-- power
|       `-- state
|-- bConfigurationValue
|-- bDeviceClass
|-- bDeviceProtocol
|-- bDeviceSubClass
|-- bMaxPower
|-- bNumConfigurations
|-- bNumInterfaces
|-- bcdDevice
|-- bmAttributes
|-- detach_state
|-- devnum
|-- idProduct
|-- idVendor
|-- maxchild
|-- power
|   `-- state
|-- speed
`-- version
```

- La estructura usb_device en el árbol:

  sys/devices/pci0000:00/0000:00:09.0/usb2/2-1

- Si conectamos un nuevo dispositivo:

  sys/devices/pci0000:00/0000:00:09.0/usb2/2-1/2-1:1.0

- La forma de etiquetar los dispositivos es:

  root_hub-hub_port : config.interface

  2 - 1 : 1 . 0

USB Urbs

- Petición de bloque USB / USB request block
- Es un procedimiento de comunicación entre el kernel y los dispositivos USB.
- Se usa para enviar y recibir datos desde y hacia un determinado endpoint en un dispositivo.

- El típico ciclo de vida de un urb es el siguiente:
  - Un driver de dispositivo USB lo crea.
  - Es asignado a un determinado endpoint de un determinado dispositivo USB.
  - Es enviado al USB core, por el driver de dispositivo USB.
  - Es enviado al driver del host USB especificado para el dispositivo especificado por el USB kernel.
  - Es procesado por el driver del host USB que hace una transferencia al dispositivo USB.
  - Cuando la urb se ha completado, el driver del USB notifica al driver del dispositivo USB.

- Un urb puede ser cancelado en cualquier momento, por el driver que envió el urb o el USB core.
- Son creados dinámicamente

Struct urb

```
struct urb
{
        /* private: usb core and host controller only fields in the urb */
        struct kref kref;               /* reference count of the URB */
        spinlock_t lock;                /* lock for the URB */
        void *hcpriv;                   /* private data for host controller */
        atomic_t use_count;             /* concurrent submissions counter */
        u8 reject;                      /* submissions will fail */

        /* public: documented fields in the urb that can be used by drivers */
        struct list_head urb_list;      /* list head for use by the urb's
                                         * current owner */
        struct list_head anchor_list;   /* the URB may be anchored by the driver */
        struct usb_anchor *anchor;
        struct usb_device *dev;         /* (in) pointer to associated device */
        unsigned int pipe;              /* (in) pipe information */
        int status;                     /* (return) non-ISO status */
        unsigned int transfer_flags;    /* (in) URB_SHORT_NOT_OK | ...*/
        void *transfer_buffer;          /* (in) associated data buffer */
        dma_addr_t transfer_dma;        /* (in) dma addr for transfer_buffer */
        int transfer_buffer_length;     /* (in) data buffer length */
        int actual_length;              /* (return) actual transfer length */
        unsigned char *setup_packet;    /* (in) setup packet (control only) */
        dma_addr_t setup_dma;           /* (in) dma addr for setup_packet */
        int start_frame;                /* (modify) start frame (ISO) */
        int number_of_packets;          /* (in) number of ISO packets */
        int interval;                   /* (modify) transfer interval
                                         * (INT/ISO) */
        int error_count;                /* (return) number of ISO errors */
        void *context;                  /* (in) context for completion */
        usb_complete_t complete;        /* (in) completion routine */
        struct usb_iso_packet_descriptor iso_frame_desc[0];
                                        /* (in) ISO ONLY */
};
```

- Los campos de los urbs son los siguientes:

- **struct usb_device *dev** -> Puntero al dispositivo

- **unsigned int pipe** -> Información del endpoint para la estructura struct usb_device

- **void *transfer_buffer** -> Puntero al buffer usado

- **dma_addr_t transfer_dma** -> Buffer usado para transferir datos por DMA

- **int transfer_buffer_lenght** -> Longitud del buffer

- **unsigned char *setup_packet** -> Puntero a la configuración de paquetes

- **dma_addr_t setup_dma** -> Dirección del buffer para la transferencia

- **int status** -> Indica el estado actual del urb.

- **usb_complete_t complete** -> Rutina de finalización transferencia
- **int start_frame** -> Devuelve el número del marco inicial de transferencia.
- **int interval** -> Tiempo de interrupción en transmisiones síncronas.
- **int number_of_packets** -> Número de buffers de transferencia.
- **int error_count** -> Número de transferencias con algún incidente.
- **struct usb_iso_packet_descriptor iso_frame_desc[0]** -> Conjunto de estructuras que recogen el estado de cada transferencia.

Creación y destrucción de Urbs

- La estructura urb nunca debe ser creada estáticamente.
- Debe de ser creada mediante una llamada a **usb_alloc_urb**

```
struct urb *usb_alloc_urb(int iso_packets, int mem_flags)
{
        struct urb *urb;

        urb = kmalloc(sizeof(struct urb) +
                iso_packets * sizeof(struct usb_iso_packet_descriptor),
                mem_flags);
        if (!urb) {
                err("alloc_urb: kmalloc failed");
                return NULL;
        }
        usb_init_urb(urb);
        return urb;
}
```

- El parámetro **iso_packets** es el número de paquetes síncrono que este urb puede contener.
- El parámetro **mem_flags** se usa para la asignación de memoria por el kernel.
- Si tiene éxito devuelve el puntero al urb creado.
- Si no tiene éxito el valor de retorno es NULL
- Después de crear el urb, debe ser inicializada correctamente.

- Cuando el driver deja de usar el urb, se hace la llamada a la funcion:void **usb_free_urb(struct urb *urb)**
- El parámetro urb es un puntero a la struct urb que se desea eliminar.

urbs de Interrupción

- La función usb_fill_int_urb inicializa correctamente una 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)
{
        spin_lock_init(&urb->lock);
        urb->dev = dev;
        urb->pipe = pipe;
        urb->transfer_buffer = transfer_buffer;
        urb->transfer_buffer_length = buffer_length;
        urb->complete = complete_fn;
        urb->context = context;
        if (dev->speed == USB_SPEED_HIGH)
                urb->interval = 1 << (interval - 1);
        else
                urb->interval = interval;
        urb->start_frame = -1;
}
```

- struct urb *urb -> Un puntero a la urb a ser inicializado.
- struct usb_device *dev -> El dispositivo USB para el que se crea este urb.
- unsigned int pipe -> El endpoint específico de los dispositivo USB para el que se crea este urb.
- void *transfer_buffer -> Un puntero a un buffer para la transferencia de datos.
- int buffer_length -> La longitud del buffer.
- usb_complete_t complete -> Puntero al manejador que se llama cuando se completa un urb.
- void *context -> Puntero que se añade a la estructura urb para su posterior recuperación del manejador de terminación.

Bulk urbs

```
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)
{
        spin_lock_init(&urb->lock);
        urb->dev = dev;
        urb->pipe = pipe;
        urb->transfer_buffer = transfer_buffer;
        urb->transfer_buffer_length = buffer_length;
        urb->complete = complete_fn;
        urb->context = context;
}
```

- Los parámetros son los mismos que la funcion usb_fill_int_urb, sin parámetro de intervalo.

Control urbs

```
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)
{
        spin_lock_init(&urb->lock);
        urb->dev = dev;
        urb->pipe = pipe;
        urb->setup_packet = setup_packet;
        urb->transfer_buffer = transfer_buffer;
        urb->transfer_buffer_length = buffer_length;
        urb->complete = complete_fn;
        urb->context = context;
    }
```

- Los parámetros son todos los mismos que en el usb_fill_bulk_urb, excepto que hay un nuevo parámetro, unsigned char * setup_packet, que debe apuntar a la configuración de paquetes de datos que se enviará a la final.

urbs síncrono

- No tiene una función de inicialización, debe de ser inicializada a “mano”.
- Este ejemplo ha sido tomado de konicawc.c situado en driver/usb/media:

```
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;
}
```

Envío de urbs

- Una vez creada e inicializada la urb, se envía al USB core para ser enviado al dispositivo USB.
- La funcion responsable es:
  int usb_submit_urb(struct urb *urb, int mem_flags);
- El parámetro urb es un puntero a la urb que se ha enviado.
- El parámetro mem_flags se usa para pedirle al kernel USB la manera de asignar los buffers.

- Cuando el envio al USB core se realice con éxito, no se debe de acceder a cualquier campo de la urb hasta que la llamada a la función termine.

Finalizar urbs

- Si la llamada a usb_submit_urb fue un éxito, la función devuelve 0.
- Si la función falla devolverá un numero negativo.
- Si la función tiene exito se llama al manejador de terminación del urb.

- Sólo hay tres maneras de terminar una urb:
  - La urb se ha enviado al dispositivo, y el dispositivo devuelve la señal adecuada. Si esto ha ocurrido, la variable de estado de la urb estara a 0.
  - Algún tipo de error ocurre cuando se realiza un envío o recepción de datos desde el dispositivo.
  - La urb fue "disociada" del USB core.

Cancelar urbs

- Para detener una urb que se ha enviado al USB core, debería llamarse a las funciones **usb_kill_urb** o **usb_unlink_urb.**

```
int usb_kill_urb(struct urb *urb);

int usb_unlink_urb(struct urb *urb);
```

- El parametro urb para estas dos funciones es un puntero al urb que se ha cancelado.

- Cuando la función es **usb_kill_urb**, el ciclo de vida del urb se detiene. Esta función se usa generalmente cuando el dispositivo está desconectado del sistema, en la llamada de desconexión.

lsusb -> Mouse USB

```
Bus 004 Device 003: ID 15ca:00c3 Textech International Ltd.
Mini Optical Mouse
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass            0 (Defined at Interface level)
  bDeviceSubClass         0
  bDeviceProtocol         0
  bMaxPacketSize0         8
  idVendor           0x15ca Textech International Ltd.
  idProduct          0x00c3 Mini Optical Mouse
  bcdDevice            5.12
  iManufacturer           0
  iProduct                2 USB Optical Mouse
  iSerial                 0
  bNumConfigurations      1
  Configuration Descriptor:
    bLength               9
    bDescriptorType       2
    wTotalLength         34
    bNumInterfaces        1
    bConfigurationValue   1
    iConfiguration        0
    bmAttributes       0xa0
      (Bus Powered)
      Remote Wakeup
    MaxPower             98mA
    Interface Descriptor:
      bLength             9
      bDescriptorType     4
      bInterfaceNumber    0
      bAlternateSetting   0
      bNumEndpoints       1
      bInterfaceClass     3 Human Interface Device
      bInterfaceSubClass  1 Boot Interface Subclass
      bInterfaceProtocol  2 Mouse
      iInterface          0
        HID Device Descriptor:
          bLength             9
          bDescriptorType    33
          bcdHID           1.10
          bCountryCode        0 Not supported
          bNumDescriptors     1
          bDescriptorType    34 Report
          wDescriptorLength  72
          Report Descriptors:
            ** UNAVAILABLE **
        Endpoint Descriptor:
          bLength             7
          bDescriptorType     5
          bEndpointAddress 0x81  EP 1 IN
          bmAttributes        3
            Transfer Type            Interrupt
            Synch Type               None
            Usage Type               Data
          wMaxPacketSize  0x0004  1x 4 bytes
          bInterval          10
Device Status:     0x0000
  (Bus Powered)
```

Referencias

- http://www.lrr.in.tum.de/Par/arch/usb/usbdoc/
- http://marc.info/?l=linux-usb-announce&m=92328122882688 9&w=2
- http://tali.admingilde.org/linux-docbook/usb/index.html
- http://www.linuxjournal.com/article/4786
- http://book.chinaunix.net/special//ebook/oreilly/ldd3/0596005903/linuxdrive3-CHP-13-SECT-3.html

Fin