Programming Guide for Linux USB Device Drivers

(c) 2000 by Detlef Fliegl, deti@fliegl.de

http://usb.in.tum.de

$Id: usbdoc.tex,v 1.28 2000/12/06 18:46:09 fliegl Exp $

Preface

The development of the Linux USB subsystem started in 1997 and in the meantime it was redesigned many times. This implied various changes of its internal structure and its API too. So it is even hard for experienced device driver developers to keep up to date with all ongoing discussions and current changes.

This document should give detailed information about the current state of the USB subsystem and its API for USB device drivers. The first section will deal with the basics of USB devices. You will learn about different types of devices and their properties. Going into detail you will see how USB devices communicate on the bus. The second section gives an overview of the Linux USB subsystem [2] and the device driver framework. Then the API and its data structures will be explained step by step. The last section of this document contains a reference of all API calls and their return codes.

Contents

List of Figures

1 The Universal Serial Bus

In 1994 an alliance of four industrial partners (Compaq, Intel, Microsoft and NEC) started to specify the Universal Serial Bus (USB). The bus was originally designed with these intentions:

- Connection of the PC to the telephone
- Ease-of-use
- Port expansion

The specification (version 1.0) was first released in january 1996 and the latest official version 1.1 was released in september 1998 [4]. The document is still under development and a version 2.0 was announced in 1999. More information and all specification papers can be found in [1]. The USB is strictly hierarchical and it is controlled by one host. The host uses a master / slave protocol to communicate with attached USB devices. This means that every kind of communication is initiated by the host and devices cannot establish any direct connection to other devices. This seems to be a drawback in comparison to other bus architectures but it is not because the USB was designed as a compromise of costs and performance. The master / slave protocol solves implicitly problems like collision avoidance or distributed bus arbitration. The current implementation of the USB allows 127 devices to be connected at the same time and the communication bandwidth is limited to 12Mbit/s.

1.1 Host Controllers

Today the USB host controller is integrated on most motherboard chipsets. Older boards which are not equipped with such a controller can be upgraded by PCI cards with such host controllers. All these controllers are compatible with either the Open Host Controller Interface (OHCI by Compaq) or the Universal Host Controller Interface (UHCI by Intel [7]) standard. Both types have the same capabilities and USB devices do not have to care about the host controller. Basically the hardware of UHCI is simpler and therefor it needs a more complex device driver, which could cause slightly more CPU load.

1.2 USB Devices and Transfer Characteristics

There are different types of USB devices as they can be used for different purposes. First a device can be self powered, bus powered or both. The USB can provide a power supply up to 500mA for its devices. If there are only bus powered devices on the bus the maximum power dissipation could be exceeded and therefor self powered devices exist. They need to have their own power supply. Devices that support both power types can switch to self powered mode when attaching an external power supply.

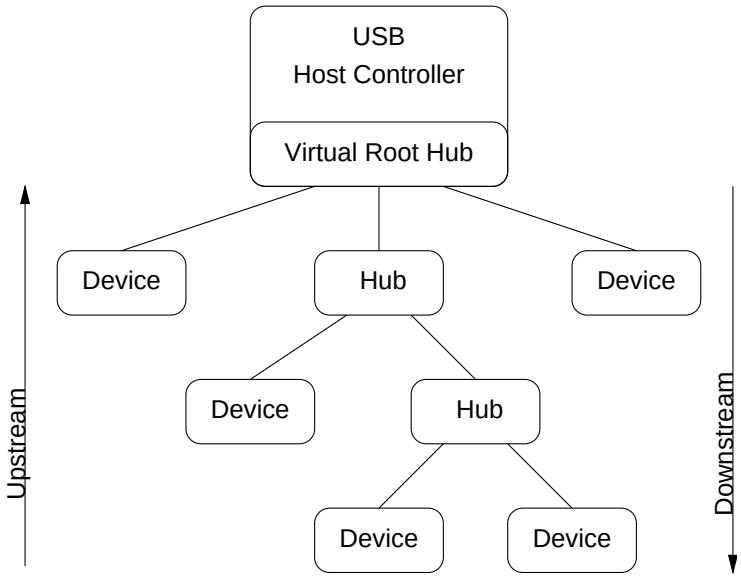

Figure 1: USB Topology

Even the maximum communication speed can differ for particular USB devices. The USB specification decides between low speed and full speed devices. Low speed devices (such as mice, keyboards, joysticks etc.) communicate at 1.5MBit/s and have only limited capabilities. Full speed devices (such as audio and video systems) can use up to 90% of the 12Mbit/s which is about 10Mbit/s including the protocol overhead.

1.2.1 Hubs

Physically there exist a number of USB ports at the rear panel of a computer. These ports can be used to attach normal devices or a hub. A hub is a USB device which extends the number of ports (i.e. 2-8) to connect other USB devices. The maximum number of attachable devices is reduced by the number of hubs on the bus. Hubs are self- and/or bus powered full speed devices.

Normally the physical ports of the host controller are handled by a virtual root hub. This hub is simulated by the host controllers device driver and helps to unify the bus topology. So every port can be handled in the same way by the USB subsystem's hub driver (see figure 1).

1.2.2 Data Flow Types

The communication on the USB is done in two directions and uses 3 different transfer types. Data directed from the host to a device is called downstream or

OUT transfer. The other direction is called upstream or IN transfer. Depending on the device type different transfer variants are used:

- **Control transfers** are used to request and send reliable short data packets. It is used to configure devices and every one is required to support a minimum set of control commands. Here is a list of standard commands:
  - GET_STATUS
  - CLEAR_FEATURE
  - SET_FEATURE
  - SET_ADDRESS
  - GET_DESCRIPTOR
  - SET_DESCRIPTOR
  - GET_CONFIGURATION
  - SET_CONFIGURATION
  - GET_INTERFACE
  - SET_INTERFACE
  - SYNCH_FRAME

  Further control commands can be used to transfer vendor specific data.
- **Bulk transfers** are used to request or send reliable data packets up to the full bus bandwidth. Devices like scanners or scsi adapters use this transfer type.
- **Interrupt transfers** are similar to bulk transfers which are polled periodically. If an interrupt transfer was submitted the host controller driver will automatically repeat this request in a specified interval (1ms - 255ms).
- **Isochronous transfers** send or receive data streams in realtime with guaranteed bus bandwidth but without any reliability. In general these transfer types are used for audio and video devices.

1.3 Enumeration and Device Descriptors

Whenever a USB device is attached to the bus it will be enumerated by the USB subsystem - i.e an unique device number (1-127) is assigned and then the device descriptor is being read. Such a desciptor is a data structure which contains information about the device and its properties. The USB standard defines a hierarchy of descriptors (see figure 2).

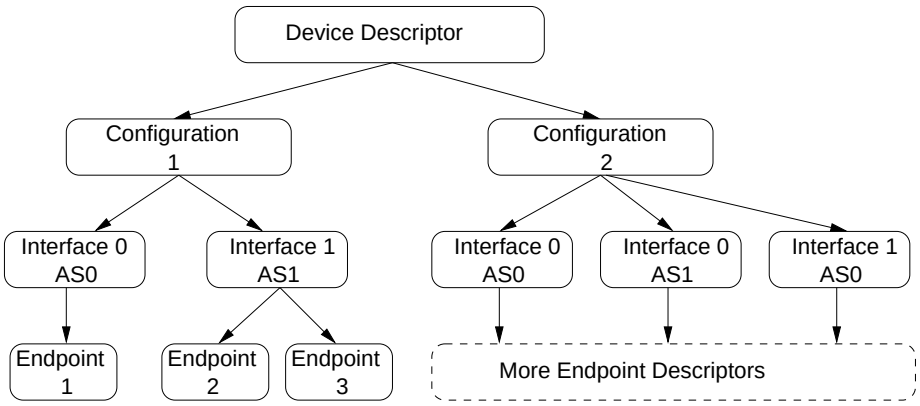

Figure 2: USB Descriptor Hierarchy

1.3.1 Standard Descriptors

- A **Device Descriptor** describes general information about a USB device. It includes information that applies globally to the device and all of the device s configurations. A USB device has only one device descriptor.
- The **Configuration Descriptor** gives information about a specific device configuration. A USB device has one or more configuration descriptors. Each configuration has one or more interfaces and each interface has zero or more endpoints. An endpoint is not shared among interfaces within a single configuration unless the endpoint is used by alternate settings of the same interface. Endpoints may be shared among interfaces that are part of different configurations without this restriction. Configurations can be activated exclusively by the standard control transfer `set_configuration`. Different configurations can be used to change global device settings like power consumption.
- An **Interface Descriptor** describes a specific interface within a configuration. A configuration provides one or more interfaces, each with zero or more endpoint descriptors describing a unique set of endpoints within the configuration. An interface may include alternate settings that allow the endpoints and/or their characteristics to be varied after the device has been configured. The default setting for an interface is always alternate setting zero. Alternate settings can be selected exclusively by the standard control transfer `set_interface`. For example a multifunctional device like a video camera with internal microphone could have three alternate settings to change the bandwidth allocation on the bus.

  1. Camera activated
  2. Microphone activated

Device Class	Example Device
Display	Monitor
Communication	Modem
Audio	Speakers
Mass storage	Hard drive
Human interface	Data glove

Table 1: USB Device Classes

3. Camera and microphone activated

- An **Endpoint Descriptor** contains information required by the host to determine the bandwidth requirements of each endpoint. An endpoint represents a logical data source or sink of a USB device. The endpoint zero is used for all control transfers and there is never a descriptor for this endpoint. The USB specification
citeusbspec11 uses the term pipe for an endpoint too.
- **String Descriptors** are optional and provide additional information in human readable unicode format. They can be used for vendor and device names or serial numbers.

1.3.2 Device Classes

The standard device and interface descriptors contain fields that are related to classification: class, sub-class and protocol. These fields may be used by a host system to associate a device or interface to a driver, depending on how they are specified by the class specification [5]. Valid values for the class fields of the device and interface descriptors are defined by the USB Device Working Group (see also Figure 1).

Grouping devices or interfaces together in classes and then specifying the characteristics in a Class Specification allows the development of host software which can manage multiple implementations based on that class. Such host software adapts its operation to a specific device or interface using descriptive information presented by the device. A class specification serves as a framework defining the minimum operation of all devices or interfaces which identify themselves as members of the class.

1.3.3 Human Interface Devices (HID)

The HID class [6] consists primarily of devices that are used by humans to control the operation of computer systems. Typical examples of HID class devices include:

- Keyboards and pointing devices for example, standard mouse devices, trackballs, and joysticks.

- Front-panel controls for example: knobs, switches, buttons, and sliders.
- Controls that might be found on devices such as telephones, VCR remote controls, games or simulation devices for example: data gloves, throttles, steering wheels, and rudder pedals.

1.4 USB Device Drivers

Finding device drivers for USB devices presents some interesting situations. In some cases the whole USB device is handled by a single device driver. In other cases, each interface of the device has a separate device driver.

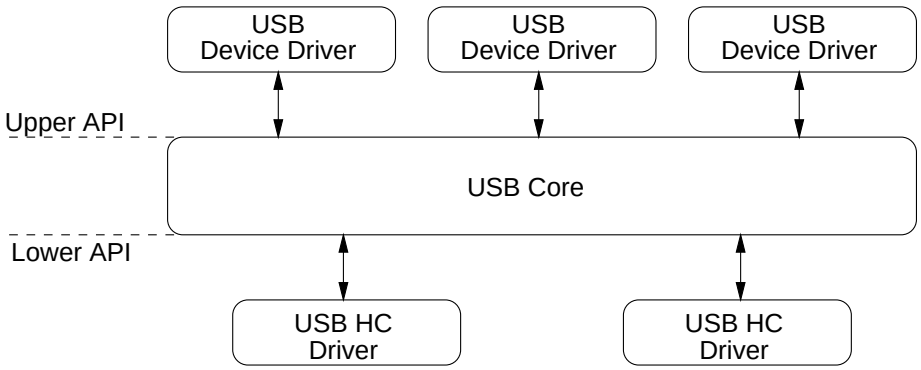

Figure 3: USB Core API Layers

2 The Linux USB Subsystem

In Linux there exists a subsystem called “The USB Core” with a specific API to support USB devices and host controllers. Its purpose is to abstract all hardware or device dependent parts by defining a set of data structures, macros and functions.

The USB core contains routines common to all USB device drivers and host controller drivers. These functions can be grouped into an upper and a lower API layer. As shown in figure 3 there exists an API for USB device drivers and another one for host controllers. The following section concentrates on the USB device driver layer, because the development for host controller drivers is already finished.

This section will give an overview of the USB framework by explaining entry points and the usage of API functions. If you are not familar with linux device drivers the following section might not be very useful. Appropriate literature can be found here [8], [9].

2.1 The USB Device Driver Framework

USB devices drivers are registered and deregistered at the subsystem. A driver must register 2 entry points and its name. For specific USB devices (which are not suitable to be registered at any other subsystem) a driver may register a couple of file operations and a minor number. In this case the specified minor number and the 15 following numbers are assigned to the driver. This makes it possible to serve up to 16 similar USB devices by one driver. The major number of all USB devices is 180.

```
struct usb_driver {
const char *name;

void * (*probe)(struct usb_device *, unsigned int);
void (*disconnect)(struct usb_device *, void *);

struct list_head driver_list;

struct file_operations *fops;
int minor;
};
```

Figure 4: usb_driver structure

2.1.1 Framework Data Structures

All USB related functions or data structures follow the same naming convention and start with `usb_`. Figure 4 shows the structure needed to register a USB device driver at the subsystem.

- `name`: Usually the name of the module.
- `probe`: The entry point of the probe function.
- `disconnect`: The entry point of the disconnect function.
- `driver_list`: For internal use of the subsystem - initialize to {NULL,NULL}
- `fops`: The usual list of file operations for a driver
- `minor`: The base minor number assigned to this device (the value has to be a multiple of 16)

2.1.2 Framework Entry Points

The USB driver framework adds two entry points to normal device drivers:

- `void *probe(struct usb_device *dev, unsigned int interface);` This entry point is called whenever a new device is attached to the bus. Then the device driver has to create a new instance of its internal data structures for the new device.

  The `dev` argument specifies the device context, which contains pointers to all USB descriptors. The `interface` argument specifies the interface number. If a USB driver wants to bind itself to a particular device and interface it has to return a pointer. This pointer normally references the device driver's context structure.

```
void *probe(struct usb_device *dev, unsigned int interface)
{
  struct driver_context *context;

  if (dev->descriptor.idVendor == 0x0547 &&
      dev->descriptor.idProduct == 0x2131 &&
      interface == 1 ) {
      MOD_INC_USE_COUNT;

      /* allocate resources for this instance */
      context=allocate_driver_resources();

      /* return pointer to instance context */
      return context;
  }

  return NULL;
}
```

Figure 5: A simple probe function

Probing normally is done by checking the vendor and product identifications or the class and subclass definitions. If they match the interface number is compared with the ones supported by the driver. When probing is done class based it might be necessary to parse some more USB descriptors because the device properties can differ in a wide range.

A simple probe routine is shown in figure 5.

- `void disconnect(struct usb_device *dev, void *drv_context);`
  This function is called whenever a device which was served by this driver is disconnected.

  The argument `dev` specifies the device context and the `driver_context` returns a pointer to the previously registered `driver_context` of the probe function. After returning from the disconnect function the USB framework completly deallocates all data structures associated with this device. So especially the `usb_device` structure must not be used any longer by the usb driver.

  A simple disconnect function is shown in figure 6.

2.1.3 Framework Functions

- `int usb_register(struct usb_driver *drv);`

  This function is used to register a new USB device driver at the subsystem. The argument `drv` points to a completely initialized `usb_driver` (see

```
static void dabusb_disconnect (struct usb_device *usbdev, void *drv_context)
{
  /* get a pointer to our driver_context */
  struct driver_context *s = drv_context;

  /* set remove pending flag */
  s->remove_pending = 1;

  /* wake up all sleeping parts of the driver */
  wake_up (&s->wait);

  /* wait until driver is ready to release device associated structures */
  sleep_on (&s->remove_ok);

  /* deallocate resources used by this instance */
  free_driver_resources(s);

  MOD_DEC_USE_COUNT;
}
```

Figure 6: A simple disconnect function

figure 4) structure. On success 0 is returned otherwise an error value is returned.

- `void usb_deregister(struct usb_driver *drv);`

  This function deregisters a formerly registerd USB device driver at the subsystem.

- `void usb_driver_claim_interface(struct usb_driver *driver, struct usb_interface *iface, void *drv_context);`

  This function is intended to be used by USB device drivers that need to claim more than one interface on a device at once when probing. The argument `driver` points to a completely initialized `usb_driver` structure. The `iface` argument points to a usb_interface structure which is part of the `usb_config_descriptor` which is accesible from the `usb_device structure` (given in the `probe` function). The `drv_context` pointer normally references the device driver's context structure (see return value of the probe function).

- `int usb_interface_claimed(struct usb_interface *iface);`

  This function is used to check if another device driver already has claimed the specified interface. The return value is 0 if the interface was not claimed by any driver.

- `void usb_driver_release_interface(struct usb_driver *driver, struct usb_interface *iface);`

  If a driver wants to release a previously claimed interface it has to call this function. In the `disconnect` function you do not have to release any interfaces that were additionally claimed in the `probe` function.

2.2 Configuring USB Devices

The API includes a set of functions to select or query descriptors, configurations and alternate settings of devices. All these standard operations are done via control transfers to the device.

2.2.1 Descriptor Data Structures

The Linux USB subsystem describes the hierarchical structure of descriptors by extending or embedding the standard USB descriptors with or in a subsystem specific structure. This structure helps storing pointers to the selected configuration and interfaces.

The elements of these structures are only explained in detail as far as they are necessary for subsequent API calls. Detailed information about the descriptors can be found in `usb.h` and [4] section 9.5.

```
struct usb_device{
  ...
  struct usb_config_descriptor *actconfig;/* the active configuration */
  ...
  struct usb_device_descriptor descriptor;/* Descriptor */
  struct usb_config_descriptor *config; /* All of the configs */
}
```

The `usb_device` structure is the root of all USB specific descriptors. Sometimes it is necessary to parse the descriptors within a driver to configure the device or to setup transfer requests properly.

- Accessing all available configuration descriptors can be done like this:

  ```
  for (i = 0; i < dev->descriptor.bNumConfigurations; i++) {
    struct usb_config_descriptor *cfg = &dev->config[i];
    ...
  }
  ```

- Accessing all available interface descriptors of a particular configuration is done like this:

  ```
  for (j = 0; j < cfg->bNumInterfaces; j++) {
    struct usb_interface *ifp = &cfg->interface[j];
    ...
  }
  ```

To start the parsing of the active configuration simply use the `dev->actconfig` pointer.

- Accessing all alternate settings of a particular interface can be done like this:

```
for (k = 0; k < ifp->num_altsetting; k++) {
  struct usb_interface_descriptor *as = &ifp->altsetting[k];
  ...
}
```

The active alternate setting can be accessed via
`*as = &ifp->altsetting[ifp->act_altsetting]`

- Accessing all endpoint descriptors of a particular alternate setting can done like this:

```
for(l = 0; l < as->bNumEndpoints; l++) {
  struct usb_endpoint_descriptor *ep=&as->endpoint[k];
  ...
}
```

2.2.2 Standard Device Requests

To query or set a particular configuration or alternate setting there exist a number functions. These commonly used functions setup standard device requests (control transfers for a specified device:

- `int usb_set_configuration(struct usb_device *dev, int configuration);`

  To activate a particular configuration use this function.
  The argument is of
  `0 <= configuration < dev->descriptor.bNumConfigurations`.
  Setting the configuration to 0 brings the device into an unaddressed state. This means the device drops its device address and is ready to accept a new one. Please do not set the 0 configuration, because you will not be able to access the device until it is reconnected physically to the bus.

- `int usb_set_interface(struct usb_device *dev, int interface, int alternate);`

  This function activates an alternate setting of a specified interface. The argument `interface` is of
  `0 <= interface < dev->actconfig->bNumInterfaces`.
  The argument `alternate` is of
  `0 <= alternate < dev->actconfig->interface[interface].num_altsetting`

- `int usb_get_device_descriptor(struct usb_device *dev);`

  This function rereads the complete descriptor tree from a particular device. It is called automatically whenever a device is attached to the bus or it may be called whenever a USB descriptor has changed.

- `int usb_get_descriptor(struct usb_device *dev, unsigned char desctype, unsigned char descindex, void *buf, int size);`

  Single USB descriptors can be read as raw data from a device. This function can be used to parse extended or vendor specific descriptors. The arguments `desctype` and `descindex` are documented in [4] section 9.4.3 and 9.5.

- `int usb_get_string(struct usb_device *dev, unsigned short langid, unsigned char index, void *buf, int size);`

  If a device, configuration or interface descriptor references a string index value (see [4] section 9.6.5) this function can be used to retrieve the string descriptor. According to the specification USB strings are coded as unicode. If successful the function returns 0 otherwise an error code is returned.

- `int usb_string(struct usb_device *dev, int index, char *buf, size_t size);`

  This function simplifies `usb_get_string` by converting unicode strings into ASCII strings.

- `int usb_get_status(struct usb_device *dev, int type, int target, void *data);`

  This USB control request is documented in [4] section 9.4.5.

- `int usb_clear_halt(struct usb_device *dev, int pipe);`

  If an endpoint is stalled (see [4] chapter 8.4.4) call this function to clear the STALL condition. STALL indicates that a function is unable to transmit or receive data, or that a control pipe request is not supported. The argument `endpoint` defines a pipe handle.

- `int usb_get_protocol(struct usb_device *dev);`

  This HID USB control request is documented in [6] section 7.2.5.

- `int usb_set_protocol(struct usb_device *dev, int protocol);`

  This HID USB control request is documented in [6] section 7.2.6.

- `int usb_get_report(struct usb_device *dev, unsigned char type, unsigned char id, unsigned char index, void *buf, int size);`

  This HID USB control request is documented in [6] section 7.2.1

- `int usb_set_idle(struct usb_device *dev, int duration, int report_id);`

  This HID USB control request is documented in [6] section 7.2.4

2.3 USB Transfers

This section will give an overview of all data structures, macros and functions related to data transfers on the bus. Further it will be explained how to actually set up, submit and process transfer requests.

2.3.1 Transfer Data Structures & Macros

The Linux USB subsystem uses only one data structure called USB Request Block (URB). This structure contains all parameters to setup any USB transfer type. All transfer requests are sent asynchronously to the USB core and the completion of the request is signalled via a callback function.

As shown in figure 7 the URB structure contains elements common to all transfer types (marked with C). Elements marked with $>$ are input parameters, M means mandatory and O means optional. Elements marked with $<$ are return values. Elements marked with T are transient parameters (input and output). All non common elements are marked on three columns which represent control, interrupt and isochronous transfers. A X marks this element to be used with the associated transfer type.

The URB structure might look confusing but this is just an overview of its versatility. There are several helping macros to setup the right parameters but first the common elements will be explained as they are very important.

- `dev` [mandatory input parameter]

  This element is a pointer to the usb_device structure (introduced in the framework function `probe` section 2.1.2).

- `pipe` [mandatory input parameter]

  The pipe element is used to encode the endpoint number and properties. There exist several macros to create an appropriate pipe value:

  - `pipe=usb_sndctrlpipe(dev,endpoint)`
    `pipe=usb_rcvctrlpipe(dev,endpoint)`
    Creates a pipe for downstream (snd) or upstream (rcv) control transfers to a given endpoint. The argument `dev` is a pointer to a usb_device structure. The argument `endpoint` is usually 0.
  - `pipe=usb_sndbulkpipe(dev,endpoint)`
    `pipe=usb_rcvbulkpipe(dev,endpoint)`
    Creates a pipe for downstream (snd) or upstream (rcv) bulk transfers to a given endpoint. The endpoint is of `1 <= endpoint <= 15` (depending on active endpoint descriptors)

```
typedef struct
{
  unsigned int offset;       // offset to the transfer_buffer
  unsigned int length;       // expected length
  unsigned int actual_length;// actual length after processing
  unsigned int status;       // status after processing
} iso_packet_descriptor_t, *piso_packet_descriptor_t;

struct urb;
typedef void (*usb_complete_t)(struct urb *);

typedef struct urb
{
     void *hcpriv;               // private data for host controller (don't care)
     struct list_head urb_list;  // list pointer to all active urbs (don't care)
>CO  struct urb* next;           // pointer to next URB
>CM  struct usb_device *dev;     // pointer to associated USB device
>CM  unsigned int pipe;          // pipe information
<C   int status;                // returned status
TCO  unsigned int transfer_flags;//USB_DISABLE_SPD|USB_ISO_ASAP|USB_URB_EARLY_COMPLETE
>CM  void *transfer_buffer;      // associated data buffer
>CM  int transfer_buffer_length; // data buffer length
<C   int actual_length;          // actual data buffer length
<X-- unsigned char *setup_packet;// setup packet (control only)
T-XX int start_frame;            // start frame (iso/irq only)
>--X int number_of_packets;      // number of packets in this request (iso only)
>-X- int interval;               // polling interval (irq only)
<--X int error_count;            // number of errors in this transfer (iso only)
>XXX int timeout;                // timeout in jiffies

>CO  void *context;              // context for completion routine
>CO  usb_complete_t complete;    // pointer to completion routine

>--X iso_packet_descriptor_t  iso_frame_desc[0]; // optional iso descriptors
} urb_t, *purb_t;
```

Figure 7: URB Structure

  - `pipe=usb_sndintpipe(dev,endpoint)`
    `pipe=usb_rcvintpipe(dev,endpoint)`
    Creates a pipe for downstream (snd) or upstream (rcv) interrupt transfers to a given endpoint. The endpoint is of `1 <= endpoint <= 15` (depending on active endpoint descriptors)
  - `pipe=usb_sndisopipe(dev,endpoint)`
    `pipe=usb_rcvisopipe(dev,endpoint)`
    Creates a pipe for downstream (snd) or upstream (rcv) isochronous transfers to a given endpoint. The endpoint is of 1 <= endpoint <= 15 (depending on active endpoint descriptors)

- `transfer_buffer` [mandatory input parameter]

  This element is a pointer to the associated transfer buffer which contains data transferred from or to a device. This buffer has to be allocated as a non-pageable contiguous physical memory block (simply use `void *kmalloc(size_t, GFP_KERNEL);`).

- `transfer_buffer_length` [mandatory input parameter]

  This element specifies the size of the transfer buffer in bytes. For interrupt and control transfers the value has to be less or equal the maximum packet size of the associated endpoint. The maximum packet size can be found as element `wMaxPacketSize` of an endpoint descriptor. Because there is no endpoint descriptor for the default endpoint 0 which is used for all control transfers the maximum packet size can be found as element `maxpacketsize` of the `usb_device` structure.

  Bulk transfers which are bigger than `wMaxPacketSize` are automatically split into smaller portions.

- `complete` [optional input parameter]

  As noted above the USB subsystem processes requests asynchronously. This element allows to specify a pointer to a caller supplied handler function which is called after the request is completed. The purpose of this handler is to finish the caller specific part of the request as fast as possible because it is called out of the host controller's hardware interrupt handler. This even implies all other restrictions that apply for code which is written for interrupt handlers.

- `context` [optional input parameter]

  Optionally a pointer to a request related context structure can be given. Figure 8 shows a simple completion handler.

- `transfer_flags` [optional input parameter and return value]

  A number of transfer flags may be specified to change the behaviour when processing the transfer request.

```
void complete( struct urb *purb )
{
   struct device_context *s = purb->context;
  /* wake up sleeping requester */
  wake_up (&s->wait);
}
```

Figure 8: A simple completion handler

  - USB_DISABLE_SPD

    This flag disables short packets. A short packet condition occures if an upstream request transfers less data than maximum packet size of the associated endpoint.

  - USB_URB_EARLY_COMPLETE

    As described above a completion handler is called after the request was processed. Use this flag to have the handler called before other (linked) URBs are resubmitted.

  - USB_ISO_ASAP

    When scheduling isochronous requests this flag tells the host controller to start the transfer as soon as possible. If USB_ISO_ASAP is not specified a start frame has to be given. It is recommended to use this flag if isochronous transfers do not have to be synchronized with the current frame number. The current frame number is a 11 bit counter that increments every millisecond (which is the duration of 1 frame on the bus). Further documentation can be found in [4] sections 5.10.6 and 5.10.8.

  - USB_ASYNC_UNLINK

    When a URB has to be cancelled (see 2.3.2) it can be done synchronously or asynchronously. Use this flag to switch on asynchronous URB unlinking.

  - USB_TIMEOUT_KILLED

    This flag is only set by the host controller to mark the URB as killed by timeout. The URB status carries the actual error which caused the timeout.

  - USB_QUEUE_BULK

    This flag is used to allow queueing for bulk transfers. Normally only one bulk transfer can be queued for an endpoint of a particular device.

- next [optional input parameter]

  It is possible to link several URBs in a chain by using the next pointer. This allows you to send a sequence of USB transfer requests to the USB

core. The chain has to be terminated by a NULL pointer or the last URB has to be linked with the first. This allows to automatically reschedule a number of URBs to transfer a continous data stream.

- `status` [return value]

  This element carries the status of an ongoing or already finished request. After successfully sending a request to the USB core the status is `-EINPROGRESS`. The successful completion of a request is indicated by 0. There exist a number of error conditions which are documented in section 3.1.

- `actual_length` [return value]

  After a request has completed this element counts the number of bytes transferred.

The remaining elements of the URB are specific to the transfer type.

- Bulk Transfers

  No additional parameters have to be specified.

- Control Transfers

  - `setup_packet` [mandatory input parameter]

    Control transfers consist of 2 or 3 stages (see [4] sections 5.5, 8.5.2) the first stage is the downstream transfer of the setup packet. This element takes the pointer to a buffer containing the setup data. This buffer has to be allocated as a non-pageable contiguous physical memory block (simply use `void *kmalloc(size_t, GFP_KERNEL);`).

- Interrupt Transfers

  - `start_frame` [return value]

    This element is returned to indicate the first frame number the interrupt is scheduled. Setting this value to -1 starts the interrupt transfers as soon as possible.

  - `interval` [mandatory input parameter] This element specifies the interval in milliseconds for the interrupt transfer. Allowed values are `1 <= interval <= 255`. Specifying an interval of 0ms causes an one shot interrupt (no automatic rescheduling is done). You can find the interrupt interval as element `bInterval` of an endpoint descriptor for interrupt endpoints.

- Isochronous Transfers

  - `start_frame` [input parameter or return value]

    This element specifies the first frame number the isochronous transfer is scheduled. Setting the `start_frame` allows to synchronize

transfers to or from a endpoint. If the USB_ISO_ASAP flag is specified this element is returned to indicate the first frame number the isochonous transfer is scheduled.

- number_of_packets [mandatory input parameter]
  Isochronous transfer requests are sent to the USB core as a set of single requests. A single requests transfers a data packet up to the maximum packet size of the specified endpoint (pipe). This element sets the number of packets for the transfer.
- error_count [return value]
  After the request is completed (URB status is != -EINPROGRESS) this element counts the number of errornous packets. Detailed information about the single transfer requests can be found in the iso_frame_desc structure.
- timeout [input parameter] A timeout in jiffies can be specified to automatically remove a URB from the host controller schedule. If a timeout happens the transfer flag USB_TIMEOUT_KILLED is set. The actual transfer status carries the USB status which caused the timeout.
- iso_frame_desc [mandatory input parameter]
  This additional array of structures at the end of every isochronous URB sets up the transfer parameters for every single request packet.
  - offset [mandatory input parameter]
    Specifies the offsetaddress to the transfer_buffer for a single request.
  - length [mandatory input parameter]
    Specifies the length of the data buffer for a single packet. If length is set to 0 for a single request the USB frame is skipped and no transfer will be initiated. This option can be used to synchronize isochronous data streams (specified in [4] section 5.6).
  - actual_length [return value]
    Returns the actual number of bytes transferred by this request.
  - status [return value]
    Returns the status of this request. Further documentation can be found in section 3.1.

2.3.2 URB Functions

There are four functions of the USB core that handle URBs.

- purb_t usb_alloc_urb(int iso_packets);
  Whenever a URB structure is needed this function has to be called. The argument iso_packets is used to specify the number of iso_frame_desc structures at the end of the URB structure when setting up isochronous

transfers. If successful the return value is a pointer to a URB structure preset to zero otherwise a NULL pointer is returned.

- `void usb_free_urb (purb_t purb);`

  To free memory allocated by `usb_alloc_urb` simply call this function.

- `int usb_submit_urb(purb_t purb);`

  This function sends a transfer request asynchronously to the USB core. The argument `purb` is a pointer to a previously allocated and initialized URB structure. If successful the return value is 0 otherwise an appropriate error code is returned (see section 3.1). *The function returns always non-blocking and it is possible to schedule several URBs for different endpoints without waiting. On isochronous endpoints it is even possible to schedule more URBs for one endpoint.* This limitation is caused due to error handling and retry mechanisms of the USB protocol (see [4] section 8.5)

- `int usb_unlink_urb(purb_t purb);`

  This function cancels a scheduled request before it is completed. The argument `purb` is a pointer to a previously submitted URB structure. The function can be called synchronously or asynchronously depending on the transfer_flag `USB_ASYNC_UNLINK` (see 2.3.1). Synchronously called the function waits for 1ms and must not be called from an interrupt or completion handler. The return value is 0 if the function succeeds. Asynchronously called the function returns immediately. The return value is -EINPROGRESS if the function was successfully started. When calling `usb_unlink_urb` the completion handler is called after the function completed. The URB status is marked with -ENOENT (synchronously called) or -ECONNRESET (asynchronously called).

  `usb_unlink_urb` is also used to stop an interrupt transfer URB. As documented in sections 1.2.2, 2.3.1 interrupt transfers are automatically rescheduled. Call `usb_unlink_urb` even for “one shot interrupts”.

2.3.3 URB Macros

To initialize URB structures for different transfer types there exist some macros:

- `FILL_CONTROL_URB(purb, dev, pipe, setup_packet, transfer_buffer, transfer_buffer_length, complete, context);`
- `FILL_BULK_URB(purb, dev, pipe, transfer_buffer, transfer_buffer_length, complete, context);`
- `FILL_INT_URB(purb, dev, pipe, transfer_buffer, transfer_buffer_length, complete, context, interval);`

The macros are self explaining - more documentation can be found in the include file usb.h.

2.3.4 Compatibility Wrappers

The USB core contains a number of higher level functions which were introduced as compatibility wrappers for the older APIs. Some of these functions can still be used to issue blocking control or bulk transfers.

- `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);`

  Issues a blocking standard control request. The arguments are according to [4] section 9.3. A timeout in milliseconds has to be specified. If successful the return value is a positive number which represents the bytes transferred otherwise an error code is returned.

- `int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe, void *data, int len, unsigned long *actual_length, int timeout);`

  Issues a blocking bulk transfer. The standard arguments should be self explaining. `actual_length` is an optional pointer to a variable which carries the actual number of bytes transferred by this request. A `timeout` in milliseconds has to be specified.

The following functions are deprecated and should no longer be used:

- `void *usb_request_bulk(struct usb_device *dev, unsigned int pipe, usb_device_irq handler, void *data, int len, void *dev_id);`
- `int usb_terminate_bulk(struct usb_device *dev, void *first);`
- `int usb_request_irq(struct usb_device *dev, unsigned int pipe, usb_device_irq handler, int period, void *dev_id, void **handle);`
- `int usb_release_irq(struct usb_device *dev, void *handle, unsigned int pipe);`

2.4 Examples

A sample device driver is the dabusb driver which is part of the latest kernel tree. The driver covers these topics:

- Supporting multiple devices
- Claiming an interface
- Setting configuration and alternate settings

- Submitting control and bulk URBs
- Reading an isochronous data stream
- Allowing hot unplug

You can find further information and updates on [3], [2]

Now some code fragments will follow to show how to actually program different transfers.

3 Reference

3.1 Errorcodes

This is the documentation of (hopefully) all possible error codes (and their interpretation) that can be returned from the hostcontroller driver and from usbcore.

3.1.1 Error codes returned by usb_submit_urb

- Non USB specific

0	URB submission successful
-ENOMEM	No memory for allocation of internal structures

- USB specific

-ENODEV	Specified USB-device or bus doesn't exist
-ENXIO	URB already queued
-EINVAL	a) Invalid transfer type specified (or not supported) b) Invalid interrupt interval (0<=n<256) c) More than one interrupt packet requested
-EAGAIN	a) Specified ISO start frame too early b) (using ISO-ASAP) Too much scheduled for the future wait some time and try again.
-EFBIG	Too much ISO frames requested (currently uhci>900)
-EPIPE	Specified pipe-handle is already stalled
-EMSGSIZE	Endpoint message size is zero, do interface/alternate setting

3.1.2 URB Error Codes

- These error codes are returned in `urb->status` or `iso_frame_desc[n].status`:

0	Transfer completed successfully
-ENOENT	URB was canceled by unlink_urb
-EINPROGRESS	URB still pending, no results yet (actually no error until now)
-EPROTO	a) Bitstuff error b) Unknown USB error
-EILSEQ	CRC mismatch
-EPIPE	a) Babble detect b) Endpoint stalled
-ENOST	Buffer error
-ETIMEDOUT	Transfer timed out, NAK
-ENODEV	Device was removed
-EREMOTEIO	Short packet detected
-EXDEV	ISO transfer only partially completed look at individual frame status for details
-EINVAL	ISO madness, if this happens: Log off and go home

3.1.3 Error Codes returned by USB Core Functions

- `usb_register()`:

  `-EINVAL` Error during registering new driver.

- `usb_terminate_bulk()`:

  `-ENODEV` URB already removed.

- `usb_get_*/usb_set_*()`:

  All USB errors (submit/status) can occur.

References

[1] http://www.usb.org, Universal Serial Bus Implementers Forum

[2] http://www.linux-usb.org, Linux USB Developer and Support information.

[3] http://usb.in.tum.de, Linux USB Developer Pages

[4] Universal Serial Bus Specification Compaq, Intel, Microsoft, NEC, Revision 1.1, September 23, 1998

[5] Universal Serial Bus Common Class Specification Systemsoft Corporation, Intel Corporation, Revision 1.0 December 16, 1997

[6] Device Class Definition for Human Interface Devices (HID) Firmware Specification, Version 1.1, Universal Serial Bus (USB),July 4, 1999

[7] Intel Universal Host Controller Interface (UHCI) Design Guide, Revision 1.1, March 1996

[8] Linux Device Drivers, 1st Edition, Alessandro Rubini, February 1998

[9] http://selva.dit.upm.es/~jmseyas/linux/kernel/hackers-docs.html, Index of Documentation for People Interested in Writing and/or Understanding the Linux Kernel, Juan-Mariano de Goyeneche

Index