Linux USB drivers

Linux USB drivers

Michael Opdenacker

Free Electrons

http://free-electrons.com/

Created with OpenOffice.org 2.x

Purpose of this course

Learn how to implement
Linux drivers
for some of the most
complex USB devices!

Buy yours on http://www.thinkgeek.com/stuff/41/fundue.shtml!

Rights to copy

Document sources, updates and translations:
http://free-electrons.com/articles/linux-usb

Corrections, suggestions, contributions and translations are welcome!

Best viewed with...

This document is best viewed with a recent PDF reader or with OpenOffice.org itself!

- Take advantage of internal or external hyperlinks. So, don't hesitate to click on them!
- Find pages quickly thanks to automatic search
- Use thumbnails to navigate in the document in a quick way

If you're reading a paper or HTML copy, you should get your copy in PDF or OpenOffice.org format on http://free-electrons.com/articles/linux-usb!

Course prerequisites

- Fondue cheese
- Good knowledge about Linux device driver development. Most notions which are not USB specific are covered in our http://free-electrons.com/training/drivers course.
- To create real, working drivers: a good knowledge about the USB devices you want to write drivers for. A good knowledge about USB specifications too.

Contents

Linux USB basics

Linux USB drivers

USB drivers (1)

USB core drivers

- Architecture independent kernel subsystem.
  Implements the USB bus specification.
  Outside the scope of this training.

USB host drivers

- Different drivers for each USB control hardware.
  Usually available in the Board Support Package.
  Architecture and platform dependent.
  Not covered yet by this training.

USB drivers (2)

USB device drivers

- Drivers for devices on the USB bus.
  The main focus of this course!
- Platform independent: when you use Linux on an embedded platform, you can use any USB device supported by Linux (cameras, keyboards, video capture, wi-fi dongles...).

USB device controller drivers

- For Linux systems with just a USB device controller (frequent in embedded systems).
  Not covered yet by this course.

USB gadget drivers

Drivers for Linux systems with a USB device controller

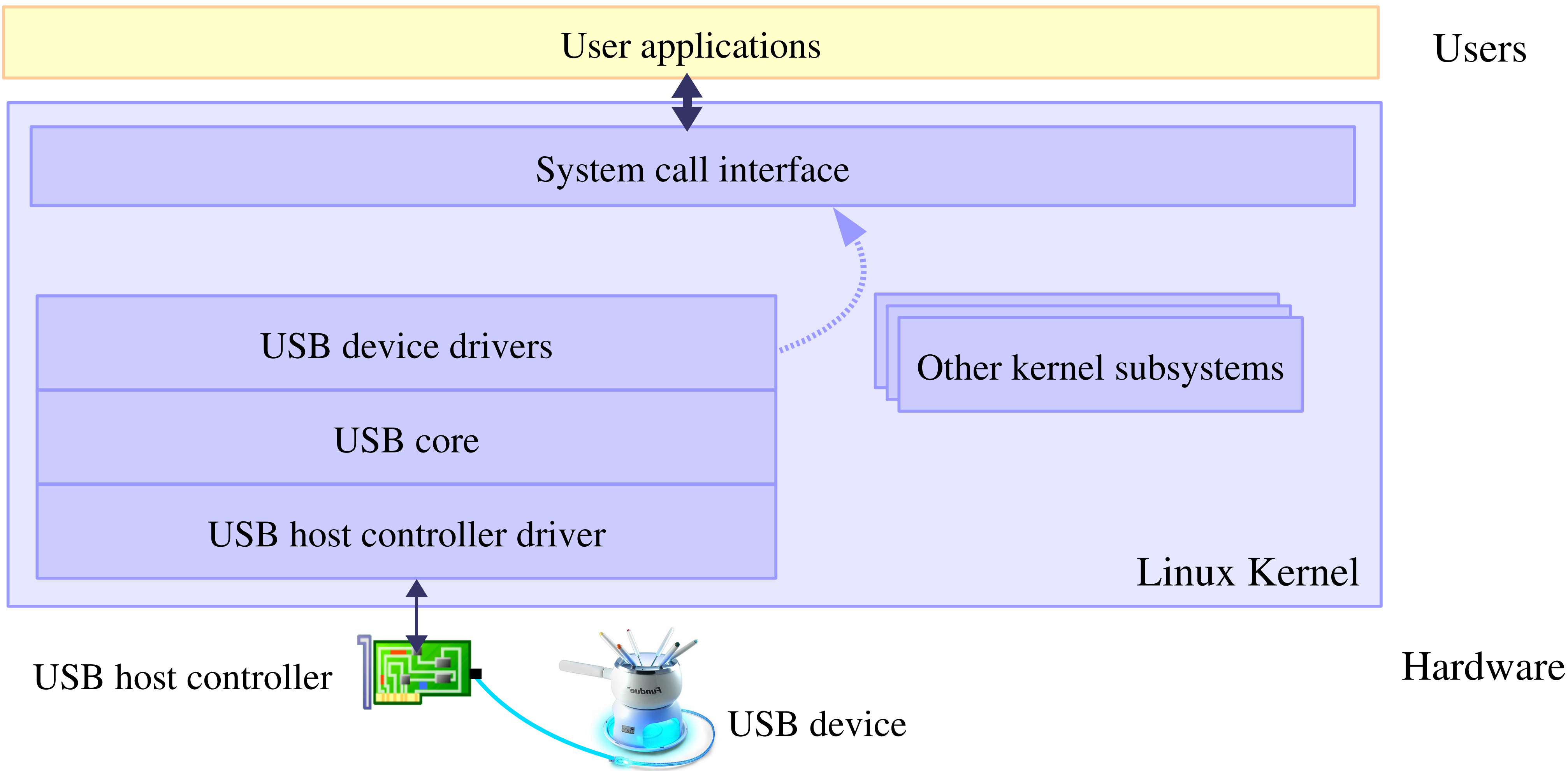
- Typical example: digital cameras.
  You connect the device to a PC and see the camera
  as a USB storage device.

- USB device controller driver:
  Platform dependent. Supports the chip connecting to the USB bus.

- USB gadget drivers, platform independent. Examples:
  Ethernet gadget: implements networking through USB
  Storage gadget: makes the host see a USB storage device
  Serial gadget: for terminal-type of communication.

See Documentation/DocBook/gadget/ in kernel sources.

Linux USB support overview



USB host controllers - OHCI and UHCI

2 competing Host Control Device (HCD) interfaces

- OHCI - Open Host Controller Interface
  Compaq's implementation adopted as a standard for USB 1.0 and 1.1 by the USB Implementers Forum (USB-IF).
  Also used for Firewire devices.

- UHCI - Universal Host Controller Interface.
  Created by Intel, insisting that other implementers use it and pay royalties for it. Only VIA licensed UHCI, and others stuck to OHCI.

This competition required to test devices for both host controller standards!

For USB 2.0, the USB-IF insisted on having only one standard.

USB host controllers - EHCI

EHCI - Extended Host Controller Interface.

- For USB 2.0. The only one to support high-speed transfers.
- Each EHCI controller contains four virtual HCD implementations to support Full Speed and Low Speed devices.
- On Intel and VIA chipsets, virtual HCDs are UHCI.
  Other chipset makers have OHCI virtual HCDs.

USB transfer speed

- Low-Speed: up to 1.5 Mbps
  Since USB 1.0
- Full-Speed: up to 12 Mbps
  Since USB 1.1
- Hi-Speed: up to 480 Mbps
  Since USB 2.0

Linux USB basics

USB devices

USB descriptors

Operating system independent. Described in the USB specification

- Device - Represent the devices connected to the USB bus.
  Example: USB speaker with volume control buttons.
- Configurations - Represent the state of the device.
  Examples: Active, Standby, Initialization
- Interfaces - Logical devices.
  Examples: speaker, volume control buttons.
- Endpoints - Unidirectional communication pipes.
  Either `IN` (device to computer) or `OUT` (computer to device).

Control endpoints

- Used to configure the device, get information about it, send commands to it, retrieve status information.
- Simple, small data transfers.
- Every device has a control endpoint (endpoint 0), used to configure the device at insertion time.
- The USB protocol guarantees that the corresponding data transfers will always have enough (reserved) bandwidth.

Interrupt endpoints

- Transfer small amounts of data at a fixed rate each time the hosts asks the device for data.
- Guaranteed, reserved bandwidth.
- For devices requiring guaranteed response time, such as USB mice and keyboards.
- Note: different than hardware interrupts. Require constant polling from the host.

Bulk endpoints

- Large sporadic data transfers
  using all remaining available bandwidth.
- No guarantee on bandwidth or latency.
- Guarantee that no data is lost.
- Typically used for printers, storage or network devices.

Isochronous endpoints

- Also for large amounts of data.
- Guaranteed speed
  (often but not necessarily as fast as possible).
- No guarantee that all data makes it through.
- Used by real-time data transfers (typically audio and video).

The `usb_endpoint_descriptor` structure (1)

The `usb_endpoint_descriptor` structure contains all the USB-specific data announced by the device itself.

Here are useful fields for driver writers:

- `__u8 bEndpointAddress`:
  USB address of the endpoint.
  It also includes the direction of the endpoint. You can use the `USB_ENDPOINT_DIR_MASK` bitmask to tell whether this is a `USB_DIR_IN` or `USB_DIR_OUT` endpoint. Example:

```
if ((endpoint->desc.bEndpointAddress &
USB_ENDPOINT_DIR_MASK) == USB_DIR_IN)
```

The `usb_endpoint_descriptor` structure (2)

- `__u8 bmAttributes`:
  The type of the endpoint. You can use the `USB_ENDPOINT_XFERTYPE_MASK` bitmask to tell whether the type is `USB_ENDPOINT_XFER_ISOC`, `USB_ENDPOINT_XFER_BULK`, `USB_ENDPOINT_XFER_INT` or `USB_ENDPOINT_XFER_CONTROL`.

- `__u8 wMaxPacketSize`:
  Maximum size in bytes that the endpoint can handle. Note that if greater sizes are used, data will be split in `wMaxPacketSize` chunks.

- `__u8 bInterval`:
  For interrupt endpoints, device polling interval (in milliseconds).

Note that the above names do not follow Linux coding standards.
The Linux USB implementation kept the original name from the USB specification (http://www.usb.org/developers/docs/).

Interfaces

- Each interface encapsulates a single high-level function (USB logical connection). Example (USB webcam): video stream, audio stream, keyboard (control buttons).
- One driver is needed for each interface!
- Alternate settings: each USB interface may have different parameter settings. Example: different bandwidth settings for an audio interface. The initial state is in the first setting, (number `0`).
- Alternate settings are often used to control the use of periodic endpoints, such as by having different endpoints use different amounts of reserved USB bandwidth. All standards-compliant USB devices that use isochronous endpoints will use them in non-default settings.

The `usb_interface` structure (1)

USB interfaces are represented by the `usb_interface` structure. It is what the USB core passes to USB drivers.

- `struct usb_host_interface *altsetting;`
  List of alternate settings that may be selected for this interface, in no particular order.
  The `usb_host_interface` structure for each alternate setting allows to access the `usb_endpoint_descriptor` structure for each of its endpoints:
  `interface->alsetting[i]->endpoint[j]->desc`
- `unsigned int num_altsetting;`
  The number of alternate settings.

The `usb_interface` structure (2)

- `struct usb_host_interface *cur_altsetting;`
  The currently active alternate setting.
- `int minor;`
  Minor number this interface is bound to.
  (for drivers using `usb_register_dev()`, described later).

Other fields in the structure shouldn't be needed by USB drivers.

Configurations

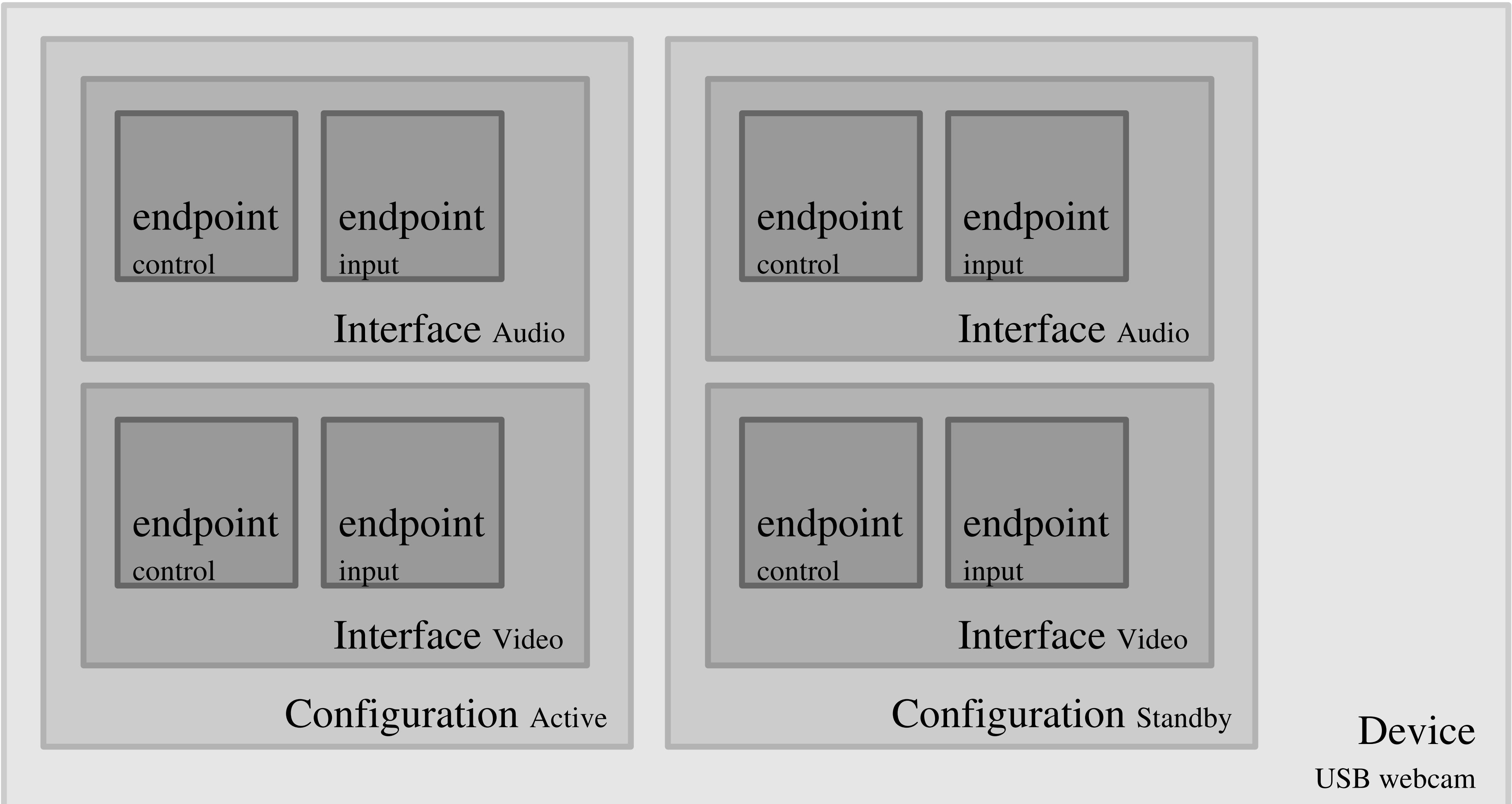
Interfaces are bundled into configurations.

- Configurations represent the state of the device.
  Examples: Active, Standby, Initialization
- Configurations are described
  with the `usb_host_config` structure.
- However, drivers do not need to access this structure.

Devices

- Devices are represented by the `usb_device` structure.
- We will see later that several USB API functions need such a structure.
- Many drivers use the `interface_to_usbdev()` function to access their `usb_device` structure from the `usb_interface` structure they are given by the USB core.

USB device overview



USB devices - Summary

- Hierarchy: device → configurations → interfaces → endpoints
- 4 different types of endpoints
  - control: device control, accessing information, small transfers. Guaranteed bandwidth.
  - interrupt (keyboards, mice...): data transfer at a fixed rate. Guaranteed bandwidth.
  - bulk (storage, network, printers...): use all remaining bandwidth. No bandwidth or latency guarantee.
  - isochronous (audio, video...): guaranteed speed. Possible data loss.

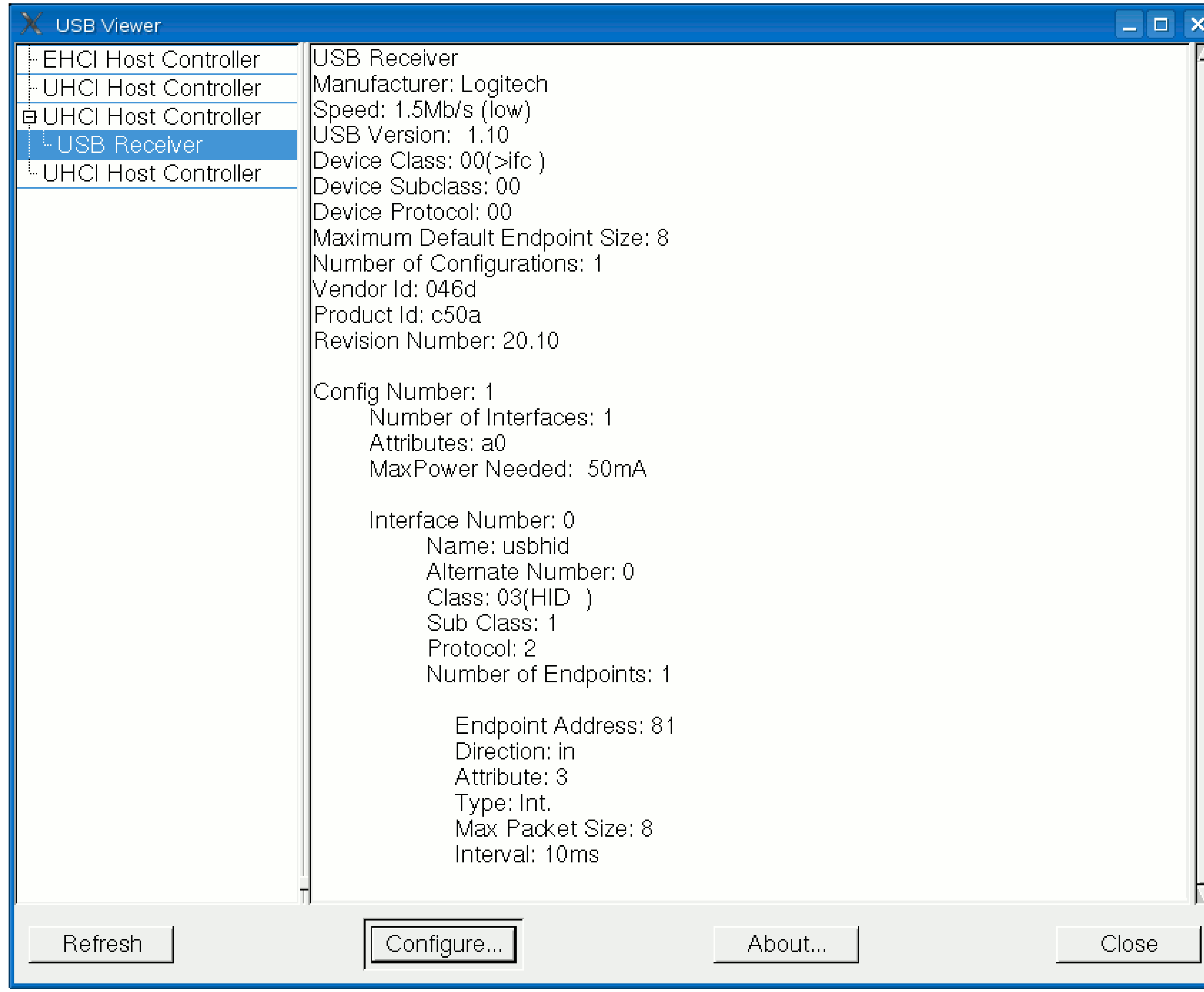
Linux USB basics

User-space representation

usbview

http://usbview.sourceforge.net

Graphical display of the contents of `/proc/bus/usb/devices`.



usbtree

http://www.linux-usb.org/usbtree

Also displays information from `/proc/bus/usb/devices`:

```
> usbtree
\/:  Bus 04.Port 1: Dev 1, Class=root_hub, Driver=ehci_hcd/6p, 480M
/:  Bus 03.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p, 12M
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p, 12M
    |__ Port 1: Dev 7, If 0, Class=HID, Driver=usbhid, 1.5M
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p, 12M
```

Linux USB communication

USB Request Blocks

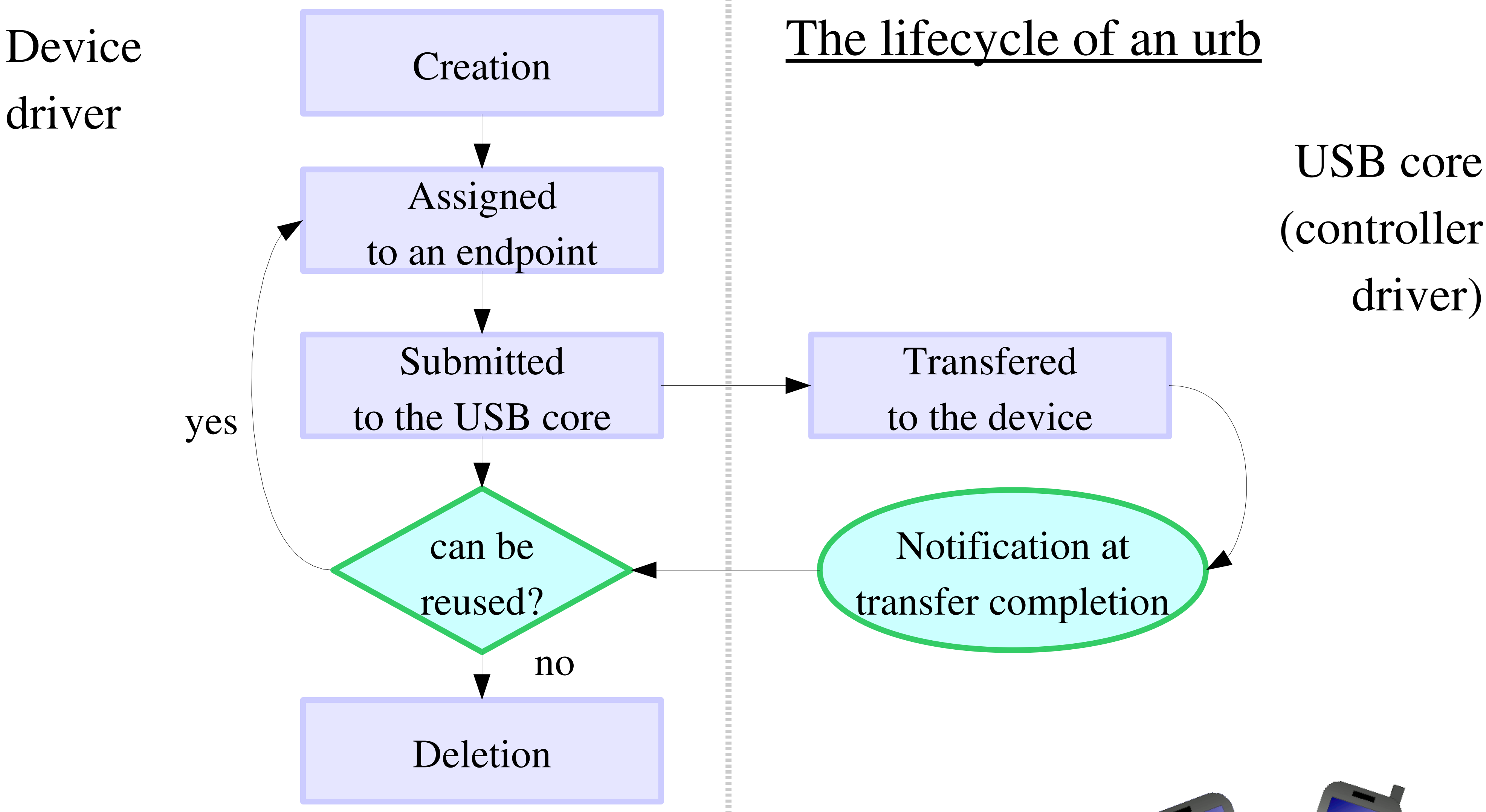
USB Request Blocks

- Any communication between the host and device is done asynchronously using USB Request Blocks (urbs).
- They are similar to packets in network communications.
- Every endpoint can handle a queue of urbs.
- Every urb has a completion handler.
- A driver may allocate many urbs for a single endpoint, or reuse the same urb for different endpoints.

See `Documentation/usb/URB.txt` in kernel sources.

Urban life

The lifecycle of an urb



The `urb` structure (1)

Fields of the `urb` structure useful to USB device drivers:

- `struct usb_device *dev;`
  Device the urb is sent to.
- `unsigned int pipe;`
  Information about the endpoint in the target device.
- `int status;`
  Transfer status.
- `unsigned int transfer_flags;`
  Instructions for handling the urb.

The `urb` structure (2)

- `void * transfer_buffer;`
  Buffer storing transferred data.
  Must be created with `kmalloc()`!
- `dma_addr_t transfer_dma;`
  Data transfer buffer when DMA is used.
- `int transfer_buffer_length;`
  Transfer buffer length.
- `int actual_length;`
  Actual length of data received or sent by the urb.
- `usb_complete_t complete;`
  Completion handler called when the transfer is complete.

The `urb` structure (3)

- `void *context;`
  Data blob which can be used in the completion handler.
- `unsigned char *setup_packet;` (control urbs)
  Setup packet transferred before the data in the transfer buffer.
- `dma_addr_t setup_dma;` (control urbs)
  Same, but when the setup packet is transferred with DMA.
- `int interval;` (isochronous and interrupt urbs)
  Urb polling interval.
- `int error_count;` (isochronous urbs)
  Number of isochronous transfers which reported an error.

The `urb` structure (4)

- `int start_frame;` (isochronous urbs)
  Sets or returns the initial frame number to use.
- `int number_of_packets;` (isochronous urbs)
  Number of isochronous transfer buffers to use.
- `struct usb_iso_packet_descriptor` (isochronous urbs)
  `iso_frame_desc[0];`
  Allows a single urb to define multiple isochronous transfers at once.

Creating pipes

Functions used to initialize the `pipe` field of the `urb` structure:

- Control pipes
  `usb_sndctrlpipe()`, `usb_rcvctrlpipe()`
- Bulk pipes
  `usb_sndbulkpipe()`, `usb_rcvbulkpipe()`
- Interrupt pipes
  `usb_sndintpipe()`, `usb_rcvintpipe()`
- Isochronous pipes
  `usb_sndisocpipe()`, `usb_rcvisocpipe()`

Prototype

send (out) receive (in)

```
unsigned int usb_[snd/rcv][ctrl/bulk/int/isoc]pipe(
      struct usb_device *dev, unsigned int endpoint);
```

Creating urbs

- `urb` structures must always be allocated with the `usb_alloc_urb()` function.
  That's needed for reference counting used by the USB core.

```
#include <linux/usb.h>

struct urb *usb_alloc_urb(
     int iso_packets,    // Number of isochronous
                         // packets the urb should contain.
                         // 0 for other transfer types
     gfp_t mem_flags);   // Standard kmalloc() flags
```

- Check that it didn't return `NULL` (allocation failed)!
- Typical example:
  `urb = usb_alloc_urb(0, GFP_KERNEL);`

Freeing urbs

- Similarly, you have to use a dedicated function to release urbs:

```
void usb_free_urb(struct urb *urb);
```

USB Request Blocks - Summary

- Basic data structure used in any USB communication.
- Implemented by the `struct urb` type.
- Must be created with the `usb_alloc_urb()` function. Shouldn't be allocated statically or with `kmalloc()`.
- Must be deleted with `usb_free_urb()`.

Linux USB communication

Initializing and submitting urbs

Initializing interrupt urbs

```
void usb_fill_int_urb (
  struct urb *urb,                 // urb to be initialized
  struct usb_device *dev,       // device to send the urb to
  unsigned int pipe,               // pipe (endpoint and device specific)
  void *transfer_buffer,        // transfer buffer
  int buffer_length,               // transfer buffer size
  usb_complete_t complete,       // completion handler
  void *context,                 // context (for handler)
  int interval                     // Scheduling interval (see next page)
  );
```

- This doesn't prevent you from making more changes to the urb fields before urb submission.
- The `transfer_flags` field needs to be set by the driver.

urb scheduling interval

For interrupt and isochronous transfers

- Low-Speed and Full-Speed devices:
  the `interval` unit is frames (ms)
- Hi-Speed devices:
  the `interval` unit is microframes (1/8 ms)

Initializing bulk urbs

Same parameters as in `usb_fill_int_urb()`, except that there is no `interval` parameter.

```
void usb_fill_bulk_urb (
  struct urb *urb,              // urb to be initialized
  struct usb_device *dev,       // device to send the urb to
  unsigned int pipe,            // pipe (endpoint and device specific)
  void *transfer_buffer,        // transfer buffer
  int buffer_length,            // transfer buffer size
  usb_complete_t complete,      // completion handler
  void *context,                // context (for handler)
  );
```

Initializing control urbs

Same parameters as in `usb_fill_bulk_urb()`, except that there is a `setup_packet` parameter.

```
void usb_fill_control_urb (
  struct urb *urb,                // urb to be initialized
  struct usb_device *dev,  // device to send the urb to
  unsigned int pipe,              // pipe (endpoint and device specific)
  unsigned char *setup_packet, // setup packet data
  void *transfer_buffer,   // transfer buffer
  int buffer_length,              // transfer buffer size
  usb_complete_t complete,     // completion handler
  void *context,               // context (for handler)
  );
```

Note that many drivers use the `usb_control_msg()` function instead (explained later).

Initializing isochronous urbs

No helper function. Has to be done manually by the driver.

```
for (i=0; i < USBVIDEO_NUMSBUF; i++) {
    int j, k;
    struct urb *urb = uvd->sbuf[i].urb;
    urb->dev = dev;
    urb->context = uvd;
    urb->pipe = usb_rcvisocpipe(dev, uvd->video_endp);
    urb->interval = 1;
    urb->transfer_flags = URB_ISO_ASAP;
    urb->transfer_buffer = uvd->sbuf[i].data;
    urb->complete = usbvideo_IsocIrq;
    urb->number_of_packets = FRAMES_PER_DESC;
    urb->transfer_buffer_length = uvd->iso_packet_len * FRAMES_PER_DESC;
    for (j=k=0; j < FRAMES_PER_DESC; j++, k += uvd->iso_packet_len) {
        urb->iso_frame_desc[j].offset = k;
        urb->iso_frame_desc[j].length = uvd->iso_packet_len;
    }
}
```

`drivers/media/video/usbvideo/usbvideo.c` example

Allocating DMA buffers (1)

You can use the `usb_buffer_alloc()` function to allocate a DMA consistent buffer:

```
void *usb_buffer_alloc (
   struct usb_device *dev, // device
   size_t size,            // buffer size
   gfp_t mem_flags,        // kmalloc() flags
   dma_addr_t *dma         // (output) DMA address
);                         // of the buffer.
```

Example:

```
buf = usb_buffer_alloc(dev->udev,
    count, GFP_KERNEL, &urb->transfer_dma);
```

Allocating DMA buffers (2)

- To use these buffers, use the `URB_NO_TRANSFER_DMA_MAP` or `URB_NO_SETUP_DMA_MAP` settings for `urb->transfer_flags` to indicate that `urb->transfer_dma` or `urb->setup_dma` are valid on submit.
- Examples:

```
urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
u->transfer_flags |= URB_NO_SETUP_DMA_MAP;
```

- Freeing these buffers:

```
void usb_buffer_free (
    struct usb_device *dev, // device
    size_t size,                    // buffer size
    void *addr,                     // CPU address of buffer
    dma_addr_t dma                  // DMA address of buffer
);
```

Submitting urbs

After creating and initializing the urb

```
int usb_submit_urb(
  struct urb *urb,     // urb to submit
  int mem_flags);      // kmalloc() flags
```

`mem_flags` is used for internal allocations performed by `usb_submit_urb()`. Settings that should be used:

- `GFP_ATOMIC`: called from code which cannot sleep: a urb completion handler, hard or soft interrupts. Or called when the caller holds a spinlock.
- `GPF_NOIO`: in some cases when block storage is used.
- `GFP_KERNEL`: in other cases.

usb_submit_urb return values

usb_submit_urb() immediately returns:

- 0: Request queued
- -ENOMEM: Out of memory
- -ENODEV: Unplugged device
- -EPIPE: Stalled endpoint
- -EAGAIN: Too many queued ISO transfers
- -EFBIG: Too many requested ISO frames
- -EINVAL: Invalid INT interval
  More than one packet for INT

Canceling urbs asynchronously

To cancel a submitted urb without waiting

- `int usb_unlink_urb(struct urb *urb);`
- Success: returns `-EINPROGRESS`
- Failure: any other return value. It can happen:
  - When the urb was never submitted
  - When the has already been unlinked
  - When the hardware is done with the urb,
    even if the completion handler hasn't been called yet.
- The corresponding completion handlers will still be run and will see `urb->status == -ECONNRESET`.

Canceling urbs synchronously

To cancel an urb and wait for all completion handlers to complete

- This guarantees that the urb is totally idle and can be reused.
- `void usb_kill_urb(struct urb *urb);`
- Typically used in a `disconnect()` callback or `close()` function.
- Caution: this routine mustn't be called in situations which can not sleep: in interrupt context, in a completion handler, or when holding a spinlock.

See comments in `drivers/usb/core/urb.c` in kernel sources for useful details.

Initializing and submitting urbs - Summary

- `urb` structure fields can be initialized with helper functions `usb_fill_int_urb()`, `usb_fill_bulk_urb()` , `usb_fill_control_urb()`
- Isochronous urbs have to be initialized by hand.
- The `transfer_flags` field must be initialized manually by each driver.
- Use the `usb_submit_urb()` function to queue urbs.
- Submitted urbs can be canceled using `usb_unlink_urb()` (asynchronous) or `usb_kill_urb()` (synchronous).

Linux USB communication

Completion handlers

When is the completion handler called?

The completion handler is called in interrupt context, in only 3 situations. Check the error value in `urb->status`.

- After the data transfer successfully completed.
  `urb->status == 0`
- Error(s) happened during the transfer.
- The urb was unlinked by the USB core.

`urb->status` should only be checked from the completion handler!

Transfer status (1)

Described in `Documentation/usb/error-codes.txt`

The urb is no longer “linked” in the system

- `-ECONNRESET`
  The urb was unlinked by `usb_unlink_urb()`.
- `-ENOENT`
  The urb was stopped by `usb_kill_urb()`.
- `-ESHUTDOWN`
  Error in from the host controller driver. The device was disconnected from the system, the controller was disabled, or the configuration was changed while the urb was sent.
- `-ENODEV`
  Device removed. Often preceded by a burst of other errors, since the hub driver doesn't detect device removal events immediately.

Transfer status (2)

Typical hardware problems with the cable or the device (including its firmware)

- `-EPROTO`
  Bitstuff error, no response packet received in time by the hardware, or unknown USB error.
- `-EILSEQ`
  CRC error, no response packet received in time, or unknown USB error.
- `-EOVERFLOW`
  The amount of data returned by the endpoint was greater than either the max packet size of the endpoint or the remaining buffer size. "Babble".

Transfer status (3)

Other error status values

- `-EINPROGRESS`
  Urb not completed yet. Your driver should never get this value.
- `-ETIMEDOUT`
  Usually reported by synchronous USB message functions
  when the specified timeout was exceed.
- `-EPIPE`
  Endpoint stalled. For non-control endpoints,
  reset this status with `usb_clear_halt()`.
- `-ECOMM`
  During an IN transfer, the host controller received data from an endpoint faster
  than it could be written to system memory.

Transfer status (4)

- `-ENOSR`
  During an OUT transfer, the host controller could not retrieve data from system memory fast enough to keep up with the USB data rate.
- `-EREMOTEIO`
  The data read from the endpoint did not fill the specified buffer, and `URB_SHORT_NOT_OK` was set in `urb->transfer_flags`.
- `-EXDEV`
  Isochronous transfer only partially completed.
  Look at individual frame status for details.
- `-EINVAL`
  Typically happens with an incorrect urb structure field or `usb_submit_urb()` function parameter.

Completion handler implementation

- Prototype:
```
void (*usb_complete_t)(
    struct urb *,        // The completed urb
    struct pt_regs *     // Register values at the time
                         // of the corresponding interrupt (if any)
);
```
- Remember you are in interrupt context:
  - Do not execute call which may sleep (use `GFP_ATOMIC`, etc.).
  - Complete as quickly as possible.
    Schedule remaining work in a tasklet if needed.

Completion handler - Summary

- The completion handler is called in interrupt context. Don't run any code which could sleep!
- Check the `urb->status` value in this handler, and not before.
- Success: `urb->status == 0`
- Otherwise, error status described in `Documentation/usb/error-codes.txt`.

Writing USB drivers

Supported devices

What devices does the driver support?

Or what driver supports a given device?

- Information needed by user-space, to find the right driver to load or remove after a USB hotplug event.
- Information needed by the driver, to call the right `probe()` and `disconnect()` driver functions (see later).

Such information is declared in a `usb_device_id` structure by the driver `init()` function.

The `usb_device_id` structure (1)

Defined according to USB specifications and described in `include/linux/mod_devicetable.h`.

- `__u16 match_flags`
  Bitmask defining which fields in the structure are to be matched against. Usually set with helper functions described later.
- `__u16 idVendor`, `idProduct`
  USB vendor and product id, assigned by the USB-IF.
- `__u16 bcdDevice_lo`, `bcdDevice_hi`
  Product version range supported by the driver, expressed in binary-coded decimal (BCD) form.

The `usb_device_id` structure (2)

- `__u8 bDeviceClass`, `bDeviceSubClass`, `bDeviceProtocol`
  Class, subclass and protocol of the device.
  Numbers assigned by the USB-IF.
  Products may choose to implement classes, or be vendor-specific. Device classes specify the behavior of all the interfaces on a device.
- `__u8 bInterfaceClass`, `bInterfaceSubclass`, `bInterfaceProtocol`
  Class, subclass and protocol of the individual interface.
  Numbers assigned by the USB-IF.
  Interface classes only specify the behavior of a given interface.
  Other interfaces may support other classes.
- `kernel_ulong_t  driver_info`

The `usb_device_id` structure (3)

- `kernel_ulong_t  driver_info`
  Holds information used by the driver. Usually it holds a pointer to a descriptor understood by the driver, or perhaps device flags.
  This field is useful to differentiate different devices from each other in the `probe()` function.

Declaring supported devices (1)

`USB_DEVICE(vendor, product)`

- Creates a `usb_device_id` structure which can be used to match only the specified vendor and product ids.
- Used by most drivers for non-standard devices.

`USB_DEVICE_VER(vendor, product, lo, hi)`

- Similar, but only for a given version range.
- Only used 11 times throughout Linux 2.6.18!

Declaring supported devices (2)

`USB_DEVICE_INFO (class, subclass, protocol)`

- Matches a specific class of USB devices.

`USB_INTERFACE_INFO (class, subclass, protocol)`

- Matches a specific class of USB interfaces.

The above 2 macros are only used in the implementations of standard device and interface classes.

Declaring supported devices (3)

Created `usb_device_id` structures are declared
with the `MODULE_DEVICE_TABLE()` macro as in the below example:

```
/* Example from drivers/usb/net/catc.c */
static struct usb_device_id catc_id_table [] = {
   { USB_DEVICE(0x0423, 0xa) }, /* CATC Netmate, Belkin F5U011 */
   { USB_DEVICE(0x0423, 0xc) }, /* CATC Netmate II, Belkin F5U111 */
   { USB_DEVICE(0x08d1, 0x1) }, /* smartBridges smartNIC */
   { }                          /* Terminating entry */
};

MODULE_DEVICE_TABLE(usb, catc_id_table);
```

Note that `MODULE_DEVICE_TABLE()` is also used
with other subsystems: `pci`, `pcmcia`, `serio`, `isapnp`, `input`...

Supported devices - Summary

- Drivers need to announce the devices they support in `usb_device_id` structures.
- Needed for user space to know which module to (un)load, and for the kernel which driver code to execute, when a device is inserted or removed.
- Most drivers use `USB_DEVICE()` to create the structures.
- These structures are then registered with `MODULE_DEVICE_TABLE(usb, xxx)`.

Writing USB drivers

Registering a USB driver

The `usb_driver` structure

USB drivers must define a `usb_driver` structure:

- `const char *name`

  Unique driver name. Usually be set to the module name.

- `const struct usb_device_id *id_table;`

  The table already declared with `MODULE_DEVICE_TABLE()`.

- ```
  int (*probe) (struct usb_interface *intf,
                const struct usb_device_id *id);
  ```

  Probe callback (detailed later).

- `void (*disconnect) (struct usb_interface *intf);`

  Disconnect callback (detailed later).
  ```

Optional `usb_driver` structure fields

- ```
  int (*suspend) (struct usb_interface *intf,
                  pm_message_t message);
  int (*resume) (struct usb_interface *intf);
  ```
  Power management: callbacks called before and after the USB core suspends and resumes the device.

- ```
  void (*pre_reset) (struct usb_interface *intf);
  void (*post_reset) (struct usb_interface *intf);
  ```
  Called by `usb_reset_composite_device()` before and after it performs a USB port reset.

Driver registration

Use `usb_register()` to register your driver. Example:

```
/* Example from drivers/usb/input/mtouchusb.c */

static struct usb_driver mtouchusb_driver = {
        .name           = "mtouchusb",
        .probe          = mtouchusb_probe,
        .disconnect     = mtouchusb_disconnect,
        .id_table       = mtouchusb_devices,
};

static int __init mtouchusb_init(void)
{
        dbg("%s - called", __FUNCTION__);
        return usb_register(&mtouchusb_driver);
}
```

Driver unregistration

Use `usb_deregister()` to register your driver. Example:

```
/* Example from drivers/usb/input/mtouchusb.c */

static void __exit mtouchusb_cleanup(void)

{
        dbg("%s - called", __FUNCTION__);
        usb_deregister(&mtouchusb_driver);
}
```

probe() and disconnect() functions

- The `probe()` function is called by the USB core to see if the driver is willing to manage a particular interface on a device.
- The driver should then make checks on the information passed to it about the device.
- If it decides to manage the interface, the `probe()` function will return `0`. Otherwise, it will return a negative value.
- The `disconnect()` function is called by the USB core when a driver should no longer control the device (even if the driver is still loaded), and should do some clean-up.

Context: USB hub kernel thread

- The `probe()` and `disconnect()` callbacks are called in the context of the USB hub kernel thread.
- So, it is legal to call functions which may sleep in these functions.
- However, all addition and removal of devices is managed by this single thread.
- Most of the probe function work should indeed be done when the device is actually opened by a user. This way, this doesn't impact the performance of the kernel thread in managing other devices.

probe() function work

- In this function the driver should initialize local structures which it may need to manage the device.
- In particular, it can take advantage of information it is given about the device.
- For example, drivers usually need to detect endpoint addresses and buffer sizes.

Time to show and explain examples in detail!

usb_set_intfdata() / usb_get_intfdata()

```
static inline void usb_set_intfdata (
     struct usb_interface *intf,
     void *data);
```

- Function used in `probe()` functions to attach collected device data to an interface. Any pointer will do!
- Useful to store information for each device supported by a driver, without having to keep a static data array.
- The `usb_get_intfdata()` function is typically used in the device open functions to retrieve the data.
- Stored data need to be freed in `disconnect()` functions:
  `usb_set_intfdata(interface, NULL);`

Plenty of examples are available in the kernel sources.

Writing USB drivers

USB transfers without URBs

Transfers without URBs

The kernel provides two `usb_bulk_msg()` and `usb_control_msg()` helper functions that make it possible to transfer simple bulk and control messages, without having to:

- Create or reuse an urb structure,
- Initialize it,
- Submit it,
- And wait for its completion handler.

Transfers without URBs - constraints

- These functions are synchronous and will make your code sleep. You must not call them from interrupt context or with a spinlock held.
- You cannot cancel your requests, as you have no handle on the URB used internally. Make sure your `disconnect()` function can wait for these functions to complete.

See the kernel sources for examples using these functions!

USB device drivers - Summary

<u>Module loading</u>

- Declare supported devices (interfaces).
- Bind them to `probe()` and `disconnect()` functions.

<u>Supported devices are found</u>

- probe() functions for matching interface drivers are called.
- They record interface information and register resources or services.

<u>Devices are opened</u>

- This calls data access functions registered by the driver.
- URBs are initialized.
- Once the transfers are over, completion functions are called.
  Data are copied from/to user-space.

<u>Devices are removed</u>

- The `disconnect()` functions are called.
- The drivers may be unloaded.

Advice for embedded system developers

If you need to develop a USB device driver for an embedded Linux system.

- Develop your driver on your GNU/Linux development host!
- The driver will run with no change on the target Linux system (provided you wrote portable code!): all USB device drivers are platform independent.
- Your driver will be much easier to develop on the host, because of its flexibility and the availability of debugging and development tools.

References

- Wikipedia's article on USB
  http://en.wikipedia.org/wiki/Universal_Serial_Bus
- The USB drivers chapter in the Linux Device Drivers book:
  http://lwn.net/Kernel/LDD3/ (Free License!)
- The Linux kernel sources (hundreds of examples, “Use the Source!”)
  Browse them with http://lxr.free-electrons.com.
- Linux USB project
  http://www.linux-usb.org/
- Linux kernel documentation:
  `Documentation/usb/`
  Linux USB API (generated from kernel sources):
  http://free-electrons.com/kerneldoc/latest/DocBook/usb/
- USB specifications:
  http://www.usb.org/developers/docs/

Annex

Ethernet over USB

Ethernet over USB (1)

If your device doesn't have Ethernet connectivity, but has a USB device controller

- You can use Ethernet over USB through the `g_ether` USB device ("gadget") driver (`CONFIG_USB_GADGET`)
- Of course, you need a working USB device driver. Generally available as more and more embedded processors (well supported by Linux) have a built-in USB device controller
- Plug-in both ends of the USB cable

Ethernet over USB (2)

- On the host, you need to have the `usbnet` module (`CONFIG_USB_USBNET`)
- Plug-in both ends of the USB cable. Configure both ends as regular networking devices. Example:
  - On the target device

    ```
    modprobe g_ether
    ifconfig usb0 192.168.0.202
    route add 192.168.0.200 dev usb0
    ```

  - On the host

    ```
    modprobe usbnet
    ifconfig usb0 192.168.0.200
    route add 192.168.0.202 dev usb0
    ```

- Works great on iPAQ PDAs!

How to help

If you support this work, you can help ...

- By sending ideas, corrections, suggestions, contributions and translations for this document.
- By speaking about it to your friends, colleagues and local Free Software community.
- By adding links to our on-line materials on your website, to increase their visibility in search engine results.
- And of course, but writing your own drivers!

Thanks

- To the OpenOffice.org project, for their presentation and word processor tools which satisfied all my needs
- To http://openclipart.org project contributors for their nice public domain clipart.
- To the members of the whole Free Software and Open Source community, for sharing the best of themselves: their work, their knowledge, their friendship.

To people who helped, sent corrections or suggestions:

Manish Katiyar

Related documents

All the technical presentations and training materials created and used by Free Electrons, available under a free documentation license (more than 1500 pages!).

http://free-electrons.com/training

- Introduction to Unix and GNU/Linux
- Embedded Linux kernel and driver development
- Free Software tools for embedded Linux systems
- Audio in embedded Linux systems
- Multimedia in embedded Linux systems

http://free-electrons.com/articles

- Advantages of Free Software in embedded systems
- Embedded Linux optimizations
- Embedded Linux from Scratch... in 40 min!
- Linux USB drivers
- Real-time in embedded Linux systems
- Introduction to uClinux
- Linux on TI OMAP processors
- Free Software development tools
- Java in embedded Linux systems
- Introduction to GNU/Linux and Free Software
- Linux and ecology
- What's new in Linux 2.6?
- How to port Linux on a new PDA

Free Electrons

Free Software for embedded systems

Embedded Linux Training

All materials released with a free license!

Unix and GNU/Linux basics
Linux kernel and drivers development
Real-time Linux, uClinux
Development and profiling tools
Lightweight tools for embedded systems
Root filesystem creation
Audio and multimedia
System optimization

Custom Development

System integration
Embedded Linux demos and prototypes
System optimization
Linux kernel drivers
Application and interface development

Consulting

Help in decision making
System architecture
Identification of suitable technologies
Managing licensing requirements
System design and performance review

Technical Support

Development tool and application support
Issue investigation and solution follow-up with mainstream developers
Help getting started

http://free-electrons.com

Free Electrons
Free Software for Embedded Systems