

# Linux Kernel Module Programming

Anandkumar

July 11, 2021



# **USB Driver**

# What is USB?

- ④ USB stands for Universal Serial Bus
- ④ Provides an expandable, fast, bi-directional, low cost, hot pluggable Plug and Play serial hardware interface
- ④ Allows users to connect a wide variety of peripherals to a computer and have them automatically configured and ready to use
- ④ Implemented to provide a replacement for legacy ports to make the addition of peripheral devices quick and easy for the end user

# Pre-Releases of USB

- ◎ **USB 0.7:** Released in November 1994.
- ◎ **USB 0.8:** Released in December 1994.
- ◎ **USB 0.9:** Released in April 1995.
- ◎ **USB 0.99:** Released in August 1995.
- ◎ **USB 1.0:** Released in November 1995

# History of USB

- ⦿ There have been three versions released prior to 3.0
  - USB 1.0 in January 1996 – data rates of 1.5 Mbps up to 12 Mbps
  - USB 1.1 in September 1998 – first widely used version of USB
  - USB 2.0 in April 2000  
Major feature revision was the addition of a high speed transfer rate of 480 Mbps

# USB 3.0 Now

- On Nov 17, 2008 It was Developed
- It is called as “SUPER SPEED” Technology
- Transfer Mode of Up to 4.8 Gbps



# Key Features

- ④ Single connector type
  - Replaces all different legacy connectors with one well-defined standardized USB connector for all USB peripheral devices
- ④ Hot swappable
  - Devices can be safely plugged and unplugged as needed while the computer is running (no need to reboot)
- ④ Plug and Play
  - OS software automatically identifies, configures, and loads the appropriate driver when connection is made

# Key Features

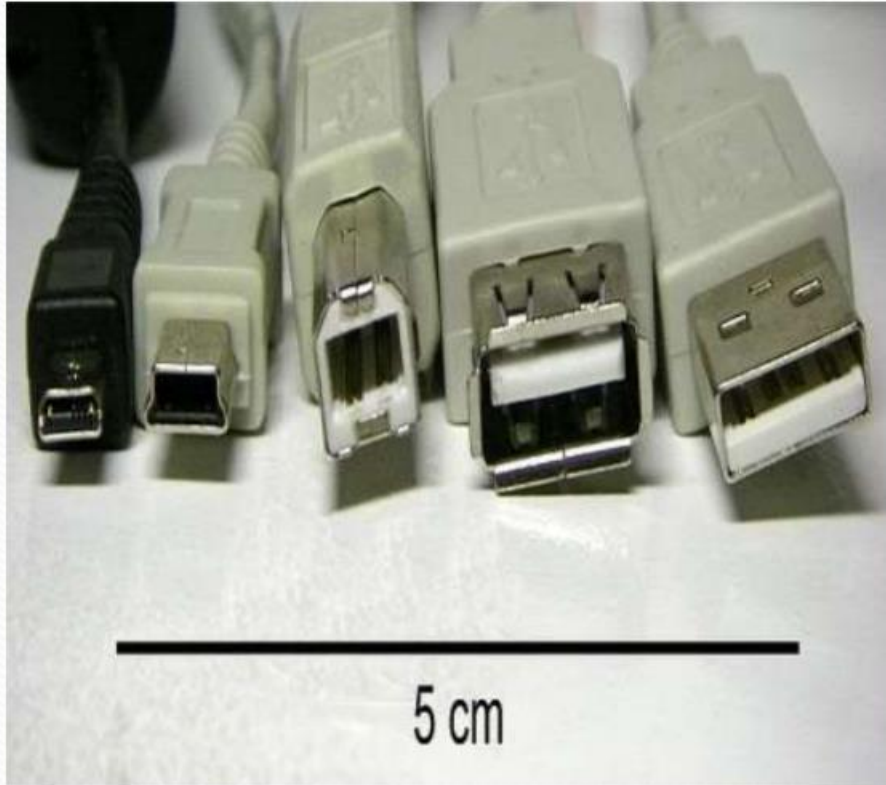
- ④ High performance
  - USB offers data transfer speeds at up to 4.8 Gbps
- ④ Expandability
  - Up to 127 different peripheral devices may theoretically be connected to a single bus at one time
- ④ Bus-supplied power
  - USB distributes the power to all connected devices, eliminating the need for an external power source for low power devices (flash drives, memory cards, Bluetooth)



# Connector properties

- Availability
  - Consumer Products are expected to become available in 2010
- Usability
  - Most connectors cannot be plugged in upside down
- Durability
  - The standard connectors were designed to be robust
- Compatibility
  - Two-way communication is also possible. In USB 3.0, full-duplex communications are done when using SuperSpeed (USB 3.0) transfer

# Connector Types



- male micro USB
- male mini USB B-type
- male B-type
- female A-type
- male A-type

Pin	Name	Cable Color	Description
1	<u>VCC</u>	Red	+5 V
2	D-	White	Data -
3	D+	Green	Data +
4	<u>GND</u>	Black	Ground

# Maximum Useful Distance

- **USB 1.1** maximum cable length is 3 metres (9.8 ft)
- **USB 2.0** maximum cable length is 5 metres (16 ft)
- **USB 3.0** cable assembly may be of any length



# USB 2.0 & USB 3.0





# APPLICATIONS

- USB implements connections to storage devices using a set of standards called the USB mass storage device class.
- USB 3.0 can also support portable hard disk drives. The earlier versions of USBs were not supporting the 3.5 inch hard disk drives.
- These external drives usually contain a translating device that interfaces a drive of conventional technology (IDE, PATA, SATA, ATAPI, or even SCSI) to a USB port.

# Linux USB drivers

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

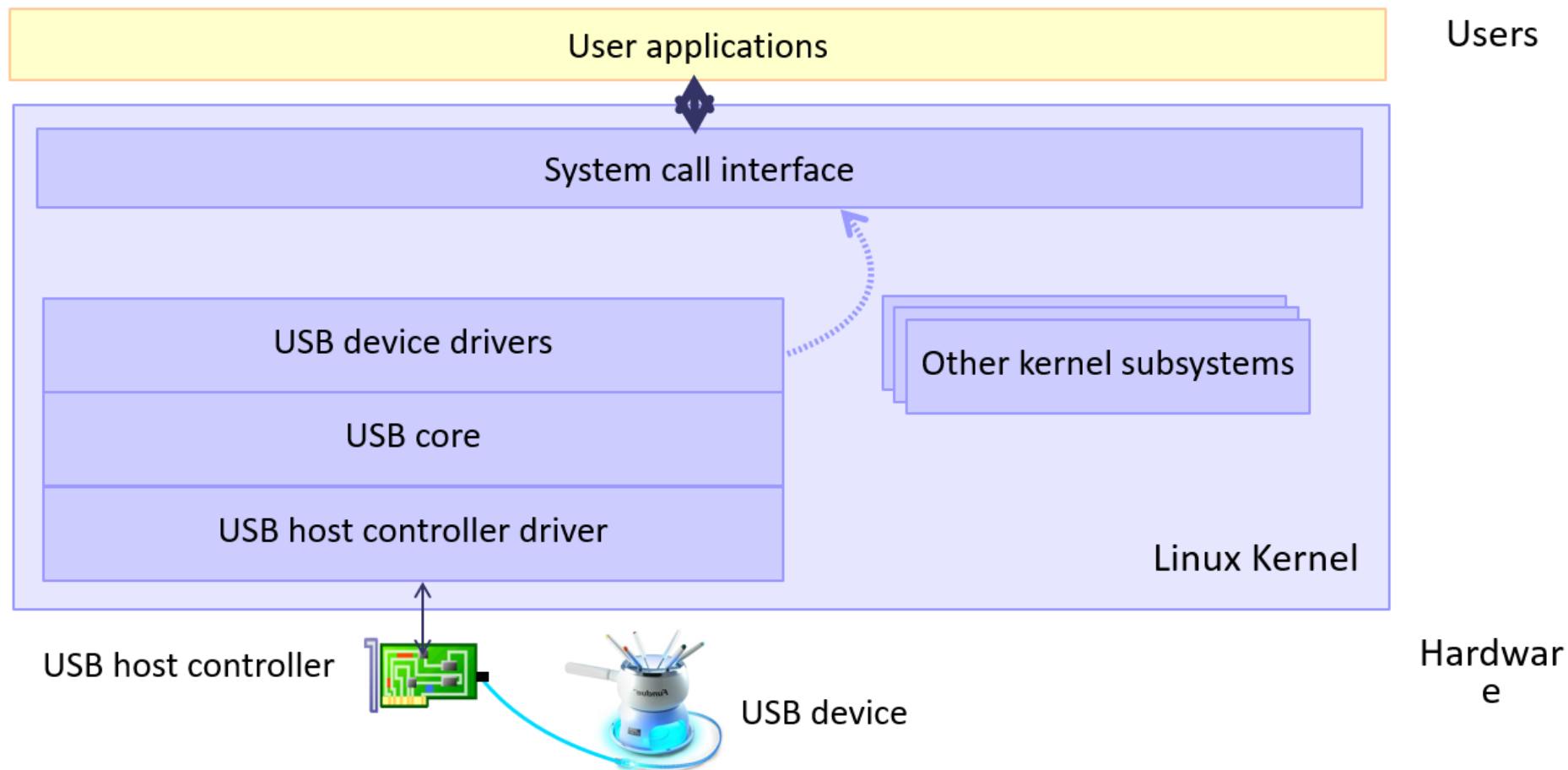
Drivers for Linux systems with a USB device controller

- ▶ 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 drivers

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->altsetting[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.

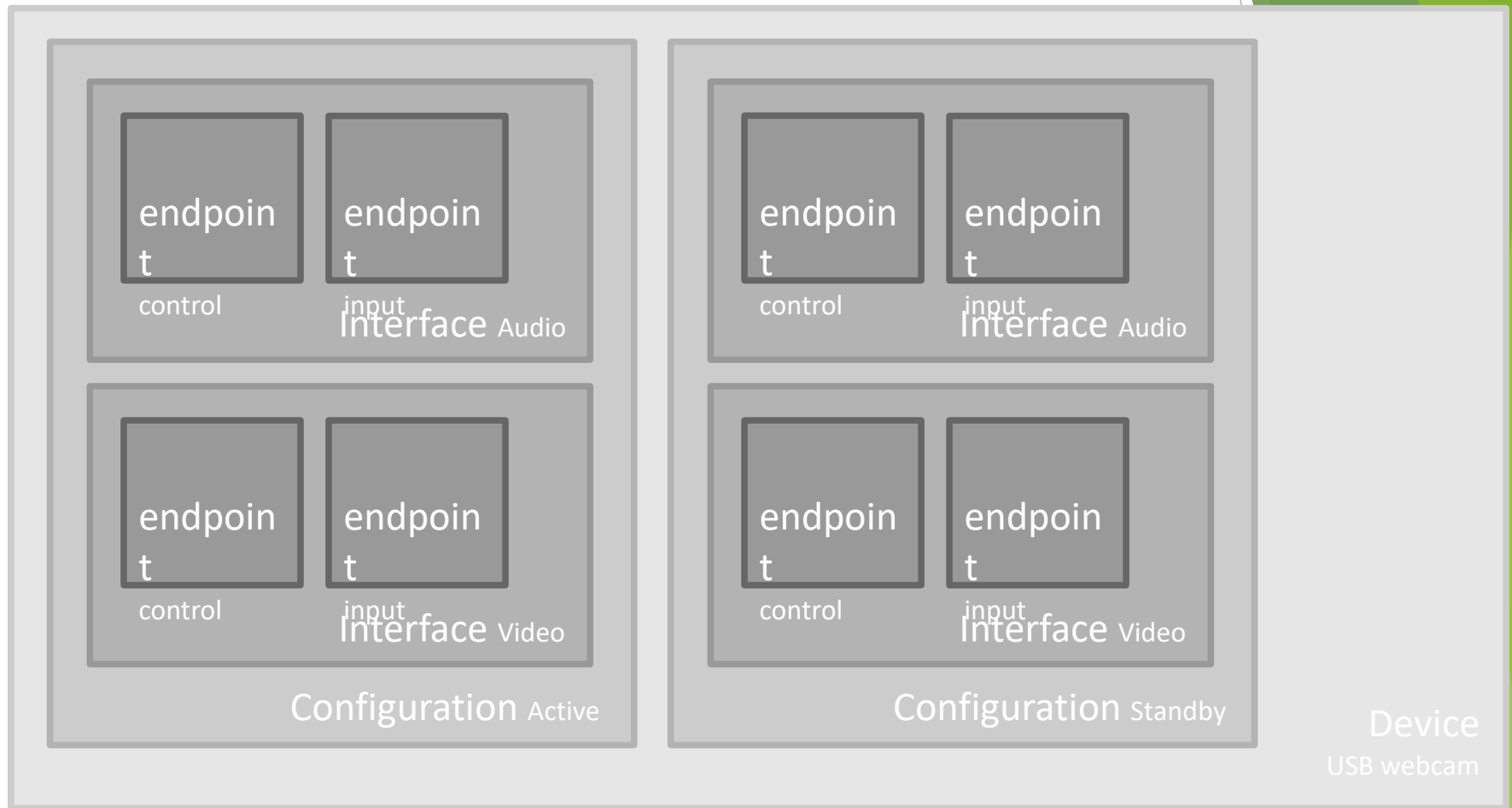
- ▶ Configurations represent the state of the device.  
Examples: Active, Standby, Initialization
- ▶ Configurations are described with the `usb_host_config` structure.
- ▶ However, drivers do not need to access this structure.



# 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.

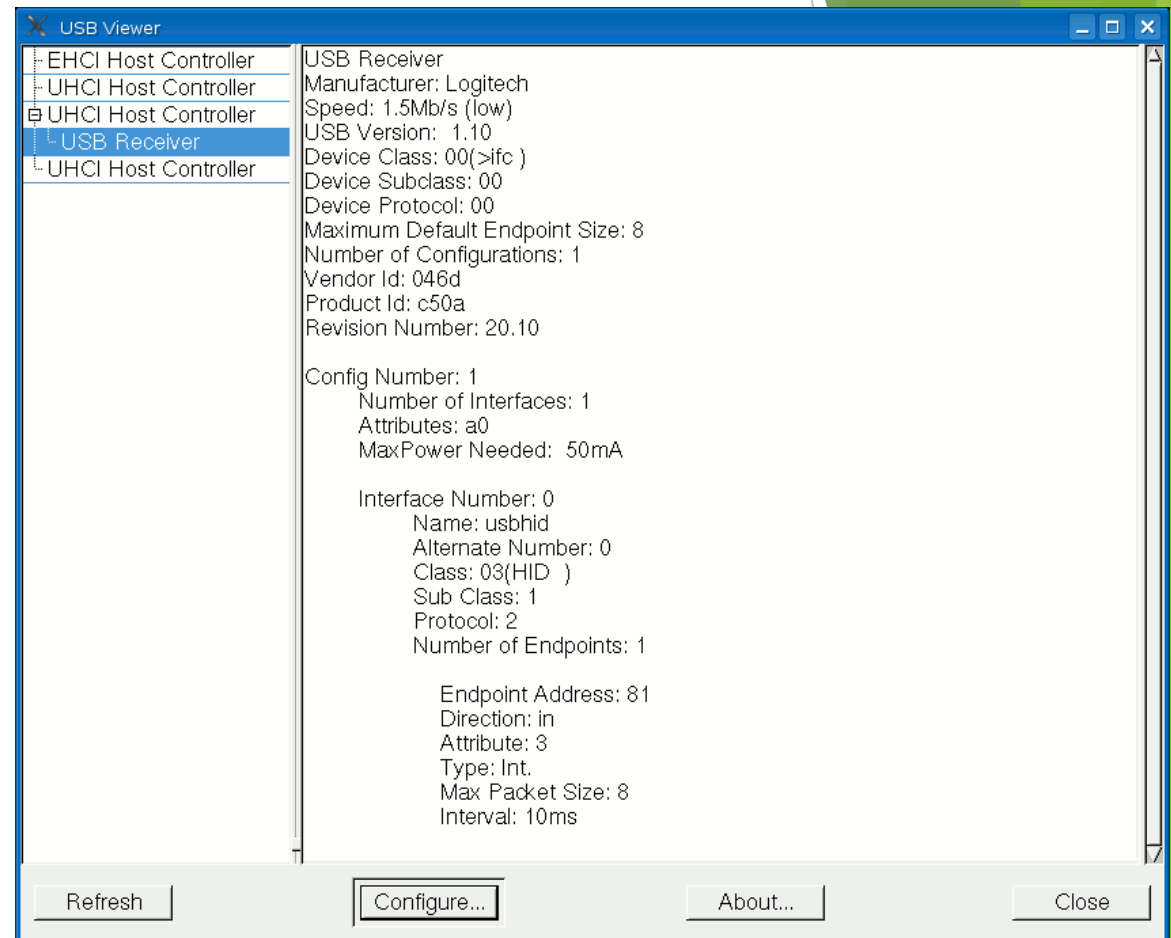
# Linux USB drivers

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 drivers

Linux USB communication  
USB Request Blocks



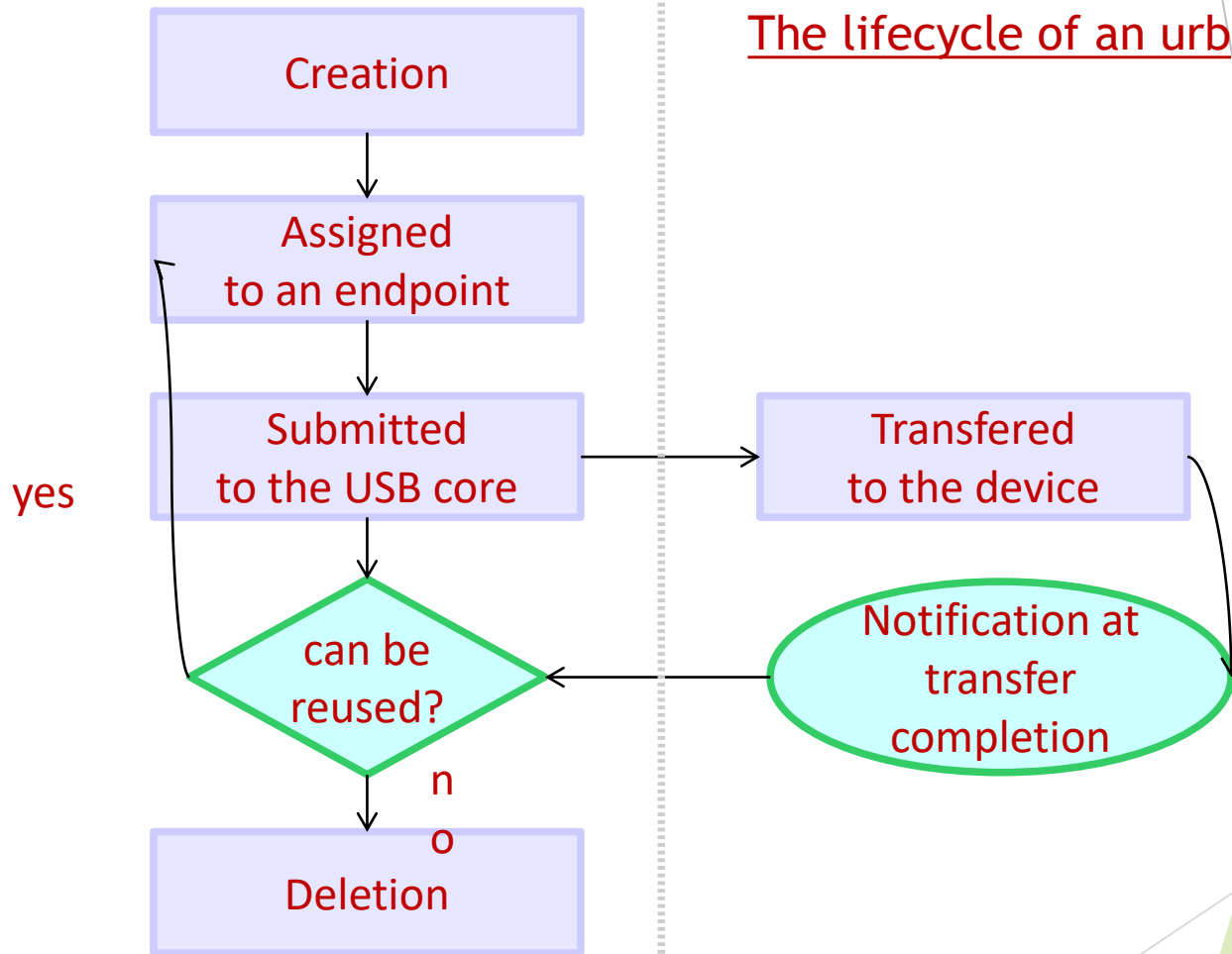
# USB Request Blocks

- ▶ 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

Device  
driver



USB core  
(controller  
driver)

# 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 drivers

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](#)

# 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 drivers

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](#).

# Linux USB drivers

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/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).

# Linux USB drivers

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.

# Linux USB drivers

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

## Module loading

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

## Supported devices are found

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

## Devices are opened

- ▶ 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.

## Devices are removed

- ▶ 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.