Linux drivers for USB devices

Purpose is to
implement
Linux drivers
for USB devices

USB

- Universal Serial Bus (USB) is an industry standard developed in the mid-1990s that defines the cables, connectors and communications protocols used in a bus for connection, communication and power supply between computers and electronic devices
- Universal Serial Bus Specification provides the technical details to understand USB requirements and design USB compatible products.
- This chapter details how the usb system that runs on desktop computer works

Contents

Linux USB basics

- USB Devices
- User Space representation
- USB Drivers

Writing USB Drivers

- List Supported devices
- Registering a usb driver
- Probe() and disconnect() functions
- Deregistering usb driver

Linux USB basics

USB devices

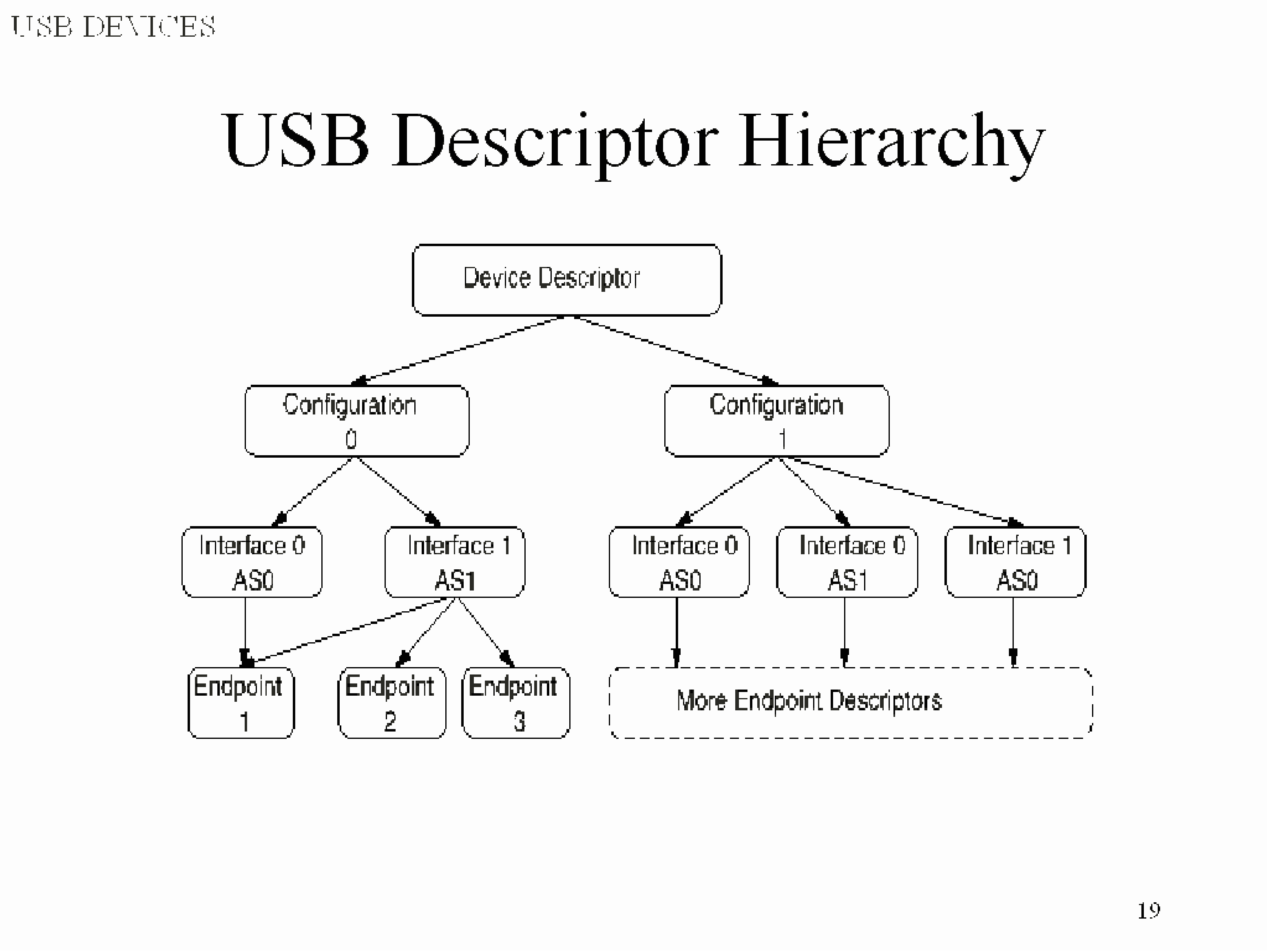
USB Device classes

- USB defines a set of standard device classes to enable interoperability across multiple platforms
  - HID – Human Interface Device
    - Keyboards, mice, controls, thermometers,
  - Mass Storage
    - Removable and non-removable storage: floppy, hard, optical, and Flash drives
  - Audio
    - Speaker, microphone, audio processor
  - Communications Device Class
    - Analog and digital modems, analog and digital telephones, ADSL and cable modems, ethernet adapters and hubs

Device identifier

- The Host machine distinguishes between devices by looking at their unique identifiers
  - VID – Vendor ID
    - Assigned by the USB Implementer’s Forum
  - PID – Product ID
    - Assigned by the vendor
  - Serial Number
    - Assigned by the developer/manufacturer
    - Unique for every USB device

USB device internally



USB DEVICES

USB Descriptor Hierarchy

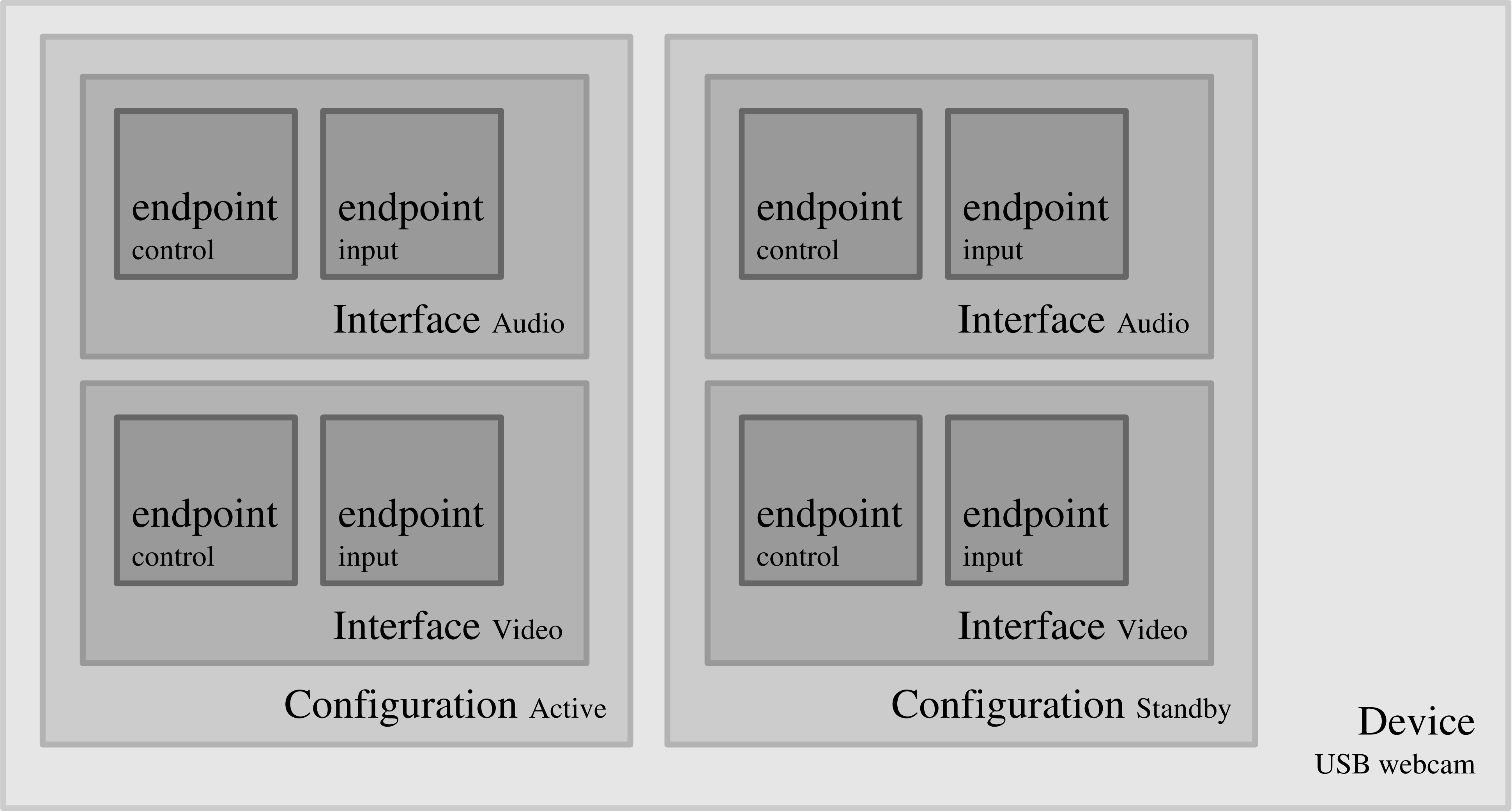
19

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).

USB device overview



Endpoint

- The most basic form of USB communication
- Can carry data in only one direction
- Either from the host computer to device(**OUT endpoint**)
- Or from the device to the host computer(**IN endpoint**)

Endpoint types

- 4 different types of endpoints
  - control: device control, accessing information, small transfers. **Every device has a control endpoint (endpoint 0), used to configure the device at insertion time.**
  - interrupt : Data transfer at a fixed rate. For devices requiring guaranteed response time, such as USB mice and keyboards.
  - bulk : Fastest transfer type. Typically used for printers, storage or network devices.
  - isochronous : Used by real-time data transfers, like audio, video.

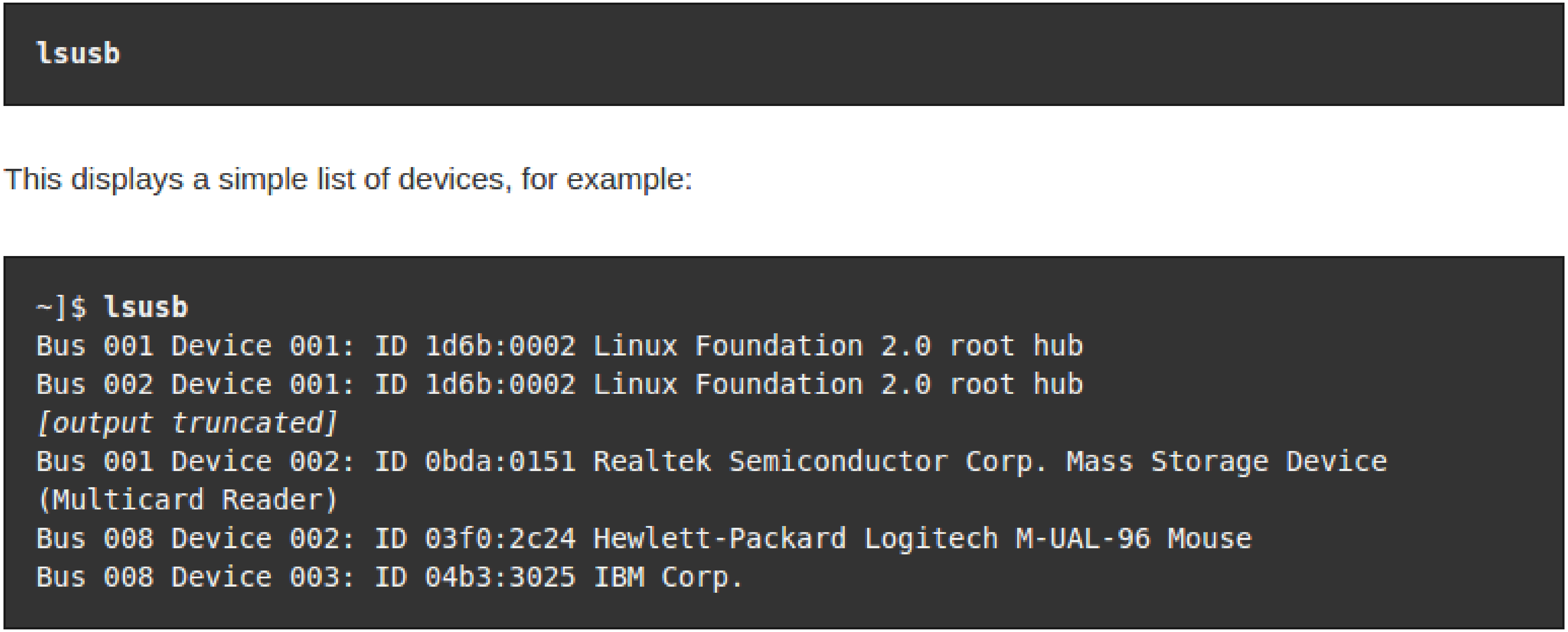
USB devices - Summary

- Hierarchy: device → configurations → interfaces → endpoints
- 4 different types of communication methods
  - control transfer using control endpoint
  - interrupt transfer using interrupt endpoint
  - bulk transfer using bulk endpoint
  - Isochronous transfer using isochronous endpoint

Linux usb basics

User Space representation

User space usb view

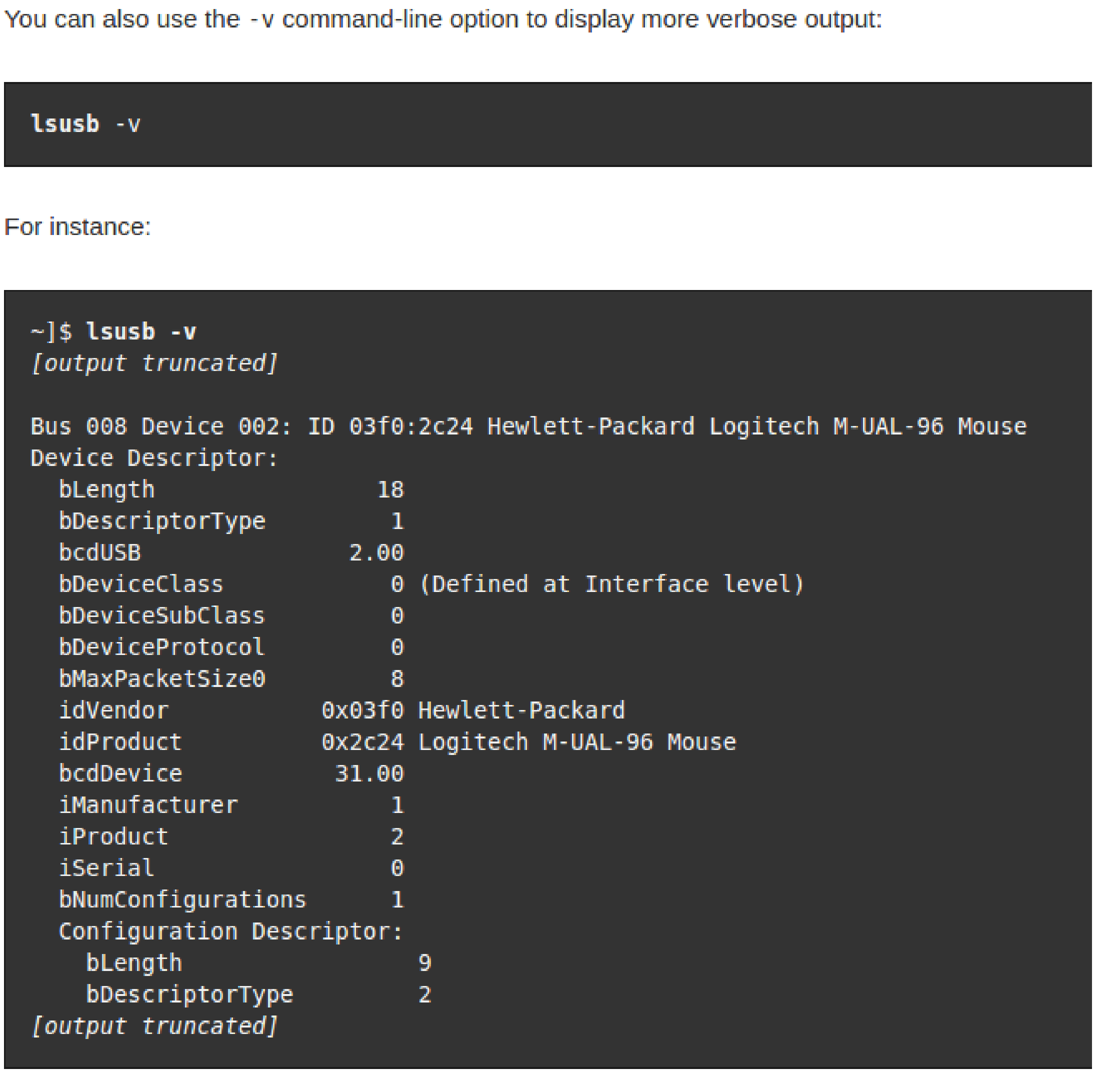


```
lsusb
```

This displays a simple list of devices, for example:

```
~]$ lsusb
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
[output truncated]
Bus 001 Device 002: ID 0bda:0151 Realtek Semiconductor Corp. Mass Storage Device
(Multicard Reader)
Bus 008 Device 002: ID 03f0:2c24 Hewlett-Packard Logitech M-UAL-96 Mouse
Bus 008 Device 003: ID 04b3:3025 IBM Corp.
```

Usb view in details (1)



You can also use the -v command-line option to display more verbose output:

```
lsusb -v
```

For instance:

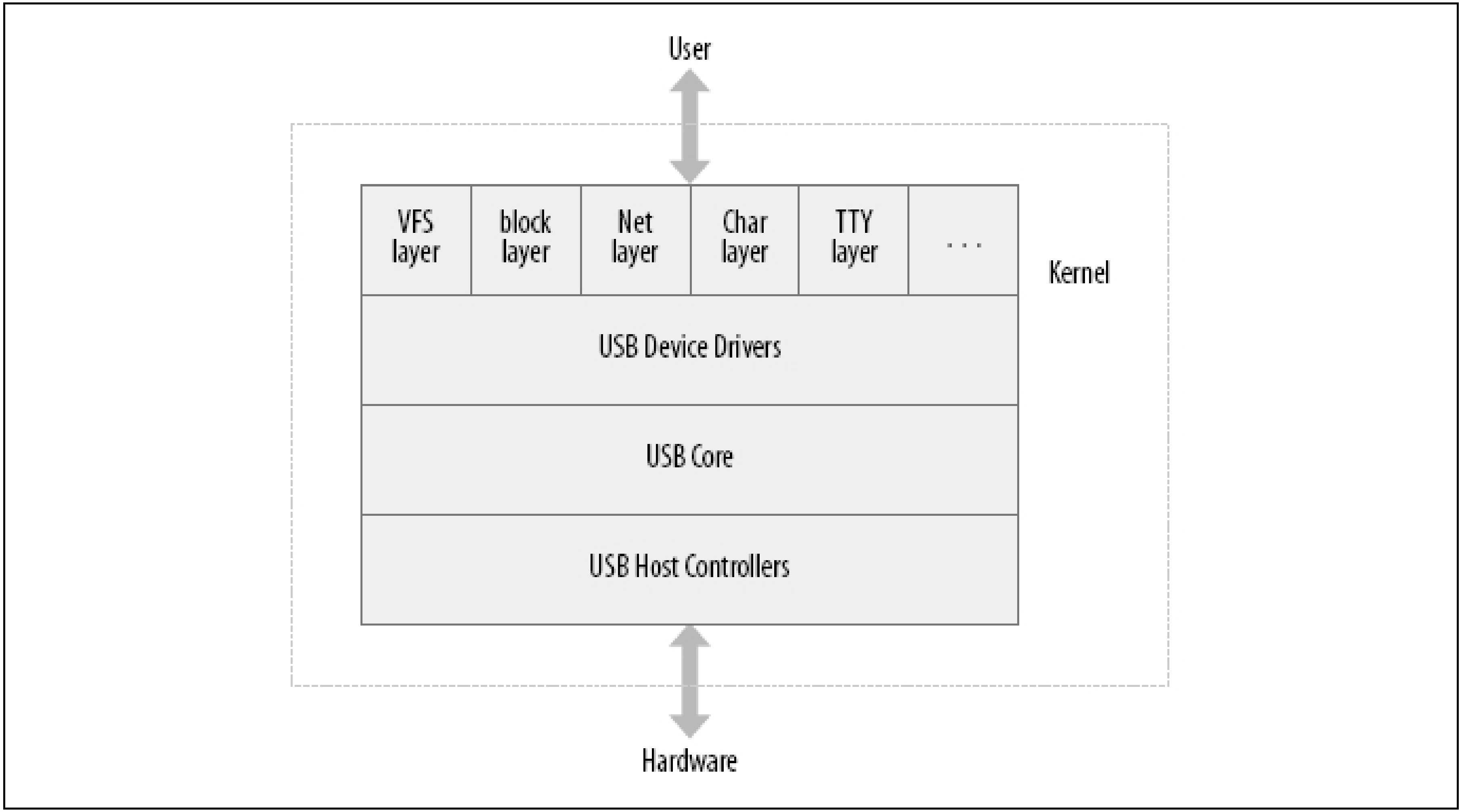
```
~]$ lsusb -v
[output truncated]

Bus 008 Device 002: ID 03f0:2c24 Hewlett-Packard Logitech M-UAL-96 Mouse
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass            0 (Defined at Interface level)
  bDeviceSubClass         0
  bDeviceProtocol         0
  bMaxPacketSize0         8
  idVendor           0x03f0 Hewlett-Packard
  idProduct          0x2c24 Logitech M-UAL-96 Mouse
  bcdDevice           31.00
  iManufacturer           1
  iProduct                2
  iSerial                 0
  bNumConfigurations      1
  Configuration Descriptor:
    bLength                 9
    bDescriptorType         2
[output truncated]
```

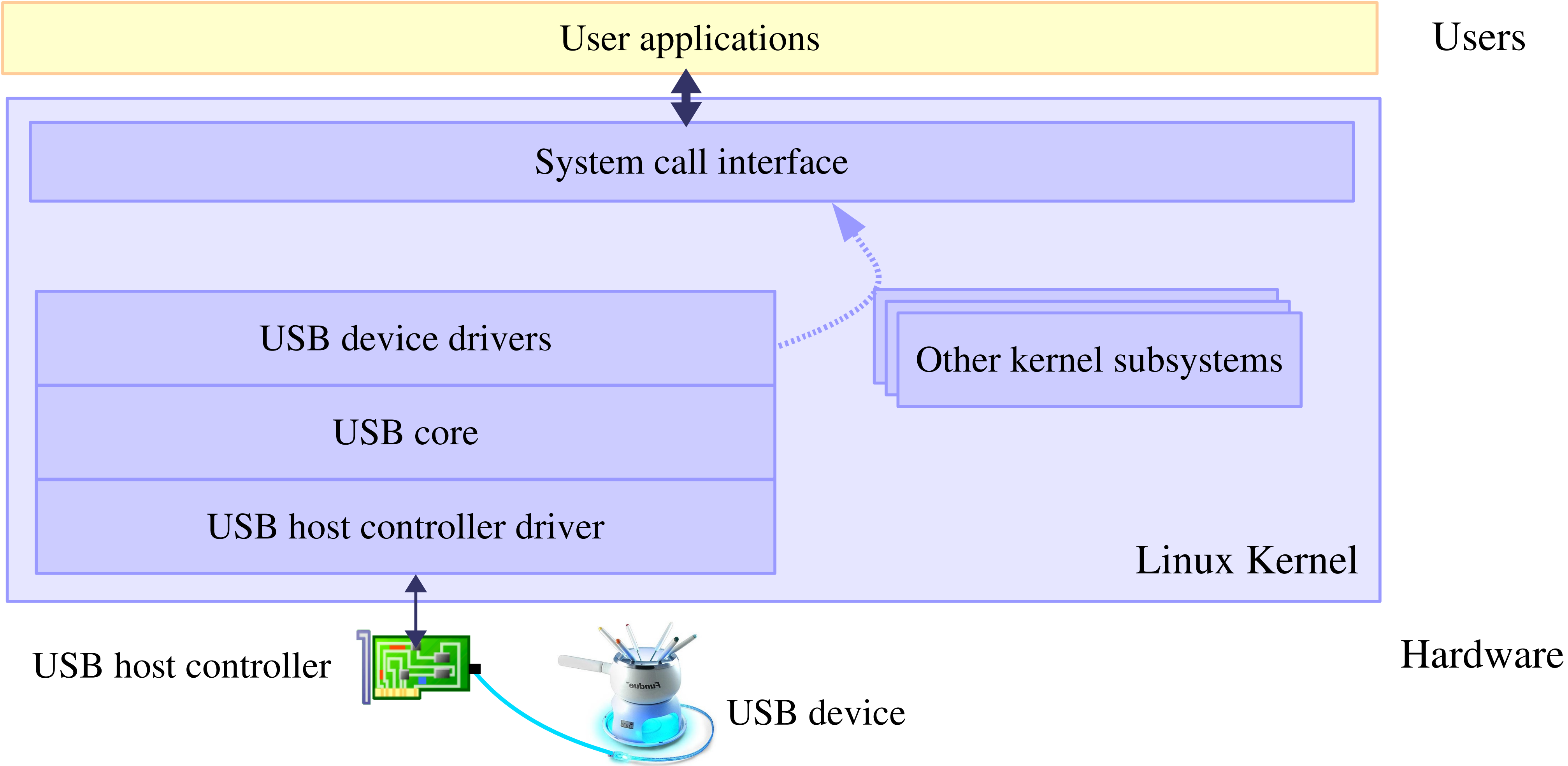
Linux USB basics

USB drivers

USB Driver overview



Linux USB support overview



User applications

Users

System call interface

USB device drivers

Other kernel subsystems

USB core

USB host controller driver

Linux Kernel

USB host controller

USB device

Hardware

USB drivers

USB core drivers

- Provides **Information about which devices the driver supports**.
- Architecture independent kernel subsystem.

USB host controller drivers

- USB Hardware controllers (Take care of connection)
- 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 device driver

- Drivers for devices on the USB bus.
- 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 host controllers – OHCI to xHCI

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).
- 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.
- EHCI - Extended Host Controller Interface.
  For USB 2.0. To support high-speed transfers.
- xHCI - eXtensible Host Controller Interface.
  Created by Intel and it is capable of interfacing with USB 1.x, 2.0, and 3.x compatible devices

Writing USB drivers

Linux USB drivers

Writing basic usb drivers

- USB Driver registration and un-registration
- Probe, disconnect functions
- Listing supported Devices the driver can support

Writing USB drivers

Registering a USB driver and de-registering USB driver

Driver registration and deregistartion

- int usb_register(struct usb_driver *driver);
- void usb_deregister(struct usb_driver *driver);

The `usb_driver` structure

- As part of the **usb_driver structure**, the fields to be provided are
  - the driver's name,
  - ID table for auto-detecting the particular device, and
  - the two callback functions to be invoked by the USB core during a hot plugging and a hot removal of the device, respectively.
  - etc..

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.

-
  ```
  int (*ioctl) (struct usb_interface, unsigned int
  code,void *buf)
  ```

Driver registration

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

```
static struct usb_driver myusb_driver = {
        .name           = "myusb",
        .probe          = myusb_probe,
        .disconnect     = myusb_disconnect,
        .id_table       = myusb_devices,
};

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

Driver unregistration

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

```
static void __exit myusb_cleanup(void)
{
        dbg("%s - called", __FUNCTION__);
        usb_deregister(&myusb_driver);
}
```

Writing USB drivers

Probe() and disconnect() functions

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.

Probe and disconnect function

- Pointer to a probing function which gets called for all usb devices which match the id table and are not handled by the other drivers yet.
- This function gets passed a pointer to the usb_interface structure representing the device interface info and also usb-device_id structure pointer.
- int (*probe) (struct usb_interface *intf, const struct usb_device_id *id);
- void (*disconnect) (struct usb_interface *intf);

Writing USB drivers

Supported devices

What devices does the driver support?

Or what driver supports a given device?

- Information needed by core, 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.

The struct usb_device_id structure provides a list of different types of USB devices that this driver supports.

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 myusb_devices [] = {
   { 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, myusb_devices);
```

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)`.

Acess USB device from user space

- Libusb library
- References
  - http://www.opensourceforu.com/2011/10/usb-drivers-in-linux-1/
  - http://libusb.info/
  -