USB drivers

Jernej Vičič

Overview

Introduction

- universal serial bus (USB),
- connection between the computer and peripherals,
- primarily intended for slow connections:
  - parallel port,
  - serial port,
  - keyboard,
  - ...
- new standards enable higher speeds and throughput (USB2/USB3/USB3.1,...).
- USB2: 480 MBps,
- USB3: 5 GBps,
- USB3.2: 20 GBps,
- power: 5 V 1A – 20V 5A.

Introduction

- topologicaly: USB is not bus,
- it is a point-to-point connection tree,
- cables are 4-core cables:
  - ground,
  - power,
  - two data wires.
- technologically, the implementation is simple.
- *single-master* implementation,
- the computer polls peripheral devices.

USB

- the USB device is complicated,
- documentation is at: *http://www.usb.org*,
- the Linux kernel uses the *USB core* subsystem,
- deals with complex issues (facilitates work by programmers),
- interaction between *USB core* and the driver,
- *endpoint* - a communication point.

USB driver architecture

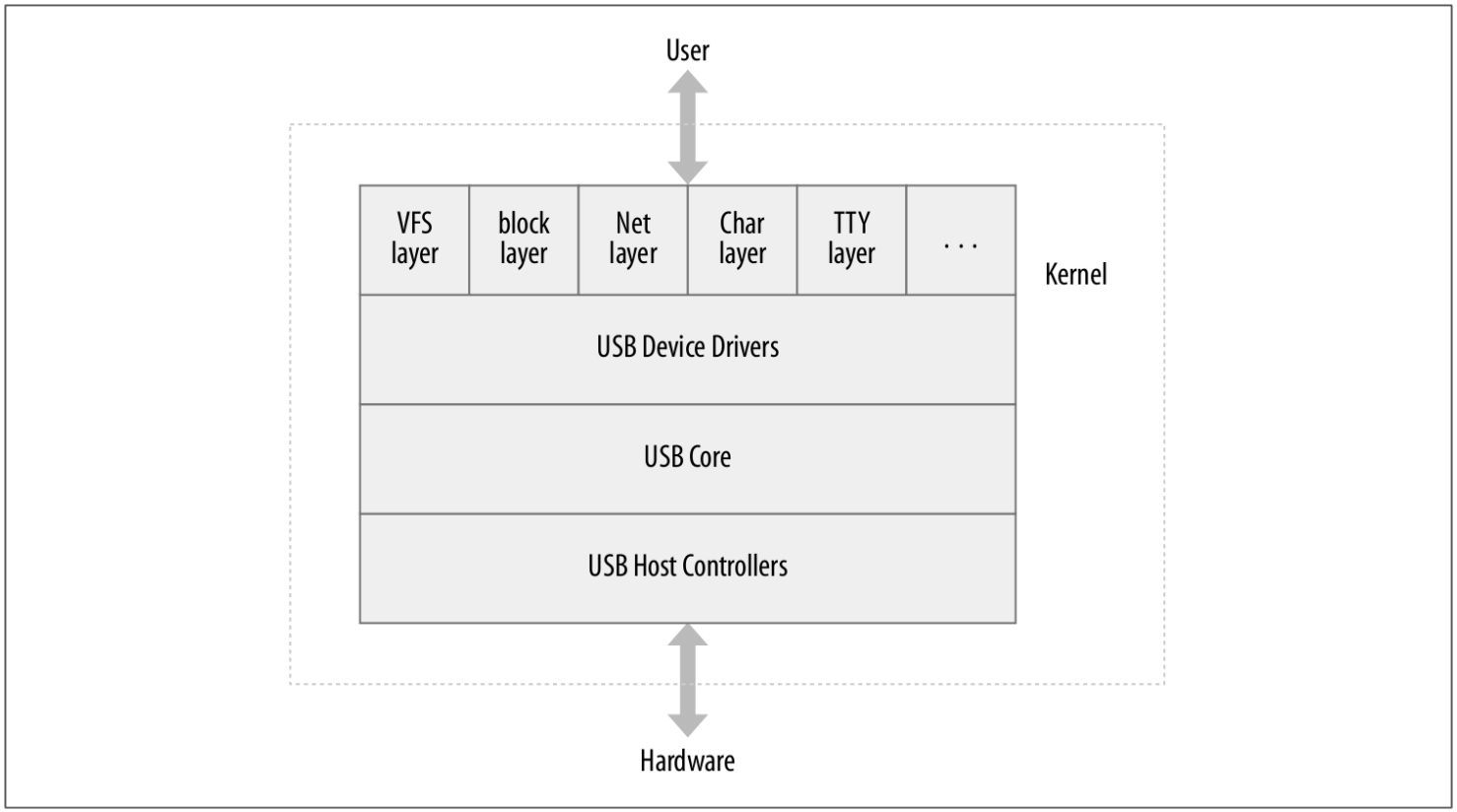

Figure: USB driver architecture.

USB device architecture

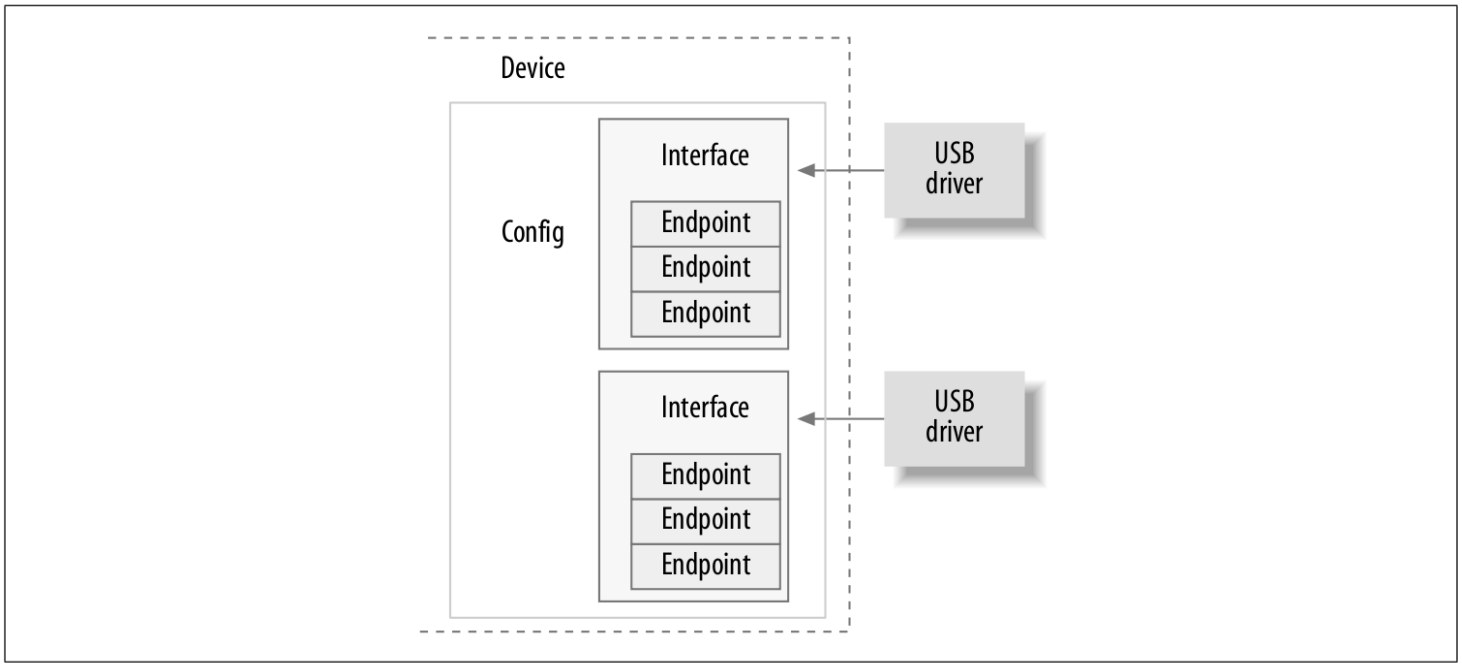

Figure: USB device architecture.

Endpoints

- the simplest way of communication,
- documentation is at: *http://www.usb.org*,
- endpoint can only communicate one-way,
- *OUT endpoint* - a computer against the device,
- *IN endpoint* - Device against the computer.

Endpoints

- 4 possible types:
  - *CONTROL*:
    - allow access to various parts of the USB device.
    - usually for configuration,
    - collecting device information.
  - *INTERRUPT*:
    - for small amounts of fixed length data,
    - the device sends the data when the computer asks,
    - most often for keyboard and mouse.

Endpoints

- 4 possible types:
  - *BULK*:
    - for large amounts of data,
    - most often for printers, storage, network devices.
  - *ISOCHRONOUS*:
    - for large amounts of data,
    - there is no guarantee that all data will arrive to the computer,
    - for devices that can suffer loss of data,
    - audio, video, real-time connections.

Interfaces

- endpoints are merged into interfaces,
- USB interface manages one logical connection,
- USB Speaker:
  - one logical connection for the audio signal,
  - another logical connection (keyboard) for buttons on the device,
- for each interface we need a separate driver,
- for our speaker two drivers.

Configurations

- interfaces are merged into configurations,
- USB device can have several configurations,
- it can only use one configuration at a time.

Conclusion

- USB devices are complex,
- they consists of several logical units:
  - one or more configurations,
  - configurations have one or more interfaces,
  - interfaces have one or more sets of settings,
  - interfaces have zero or more endpoints.

USB and sysfs

- example entry in sysfs for a device,
- each device is presented with a structure *struct usb_device*,
- example devise (mouse)is in file: */sys/devices/pci0000:00/0000:00:09.0/usb2/2-1*,
- example interface for the same device is in file: */sys/devices/pci0000:00/0000:00:09.0/usb2/2-1/2-1:1.0*,
- example entry is presented on next slide:

USB and sysfs

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1
|-- 2-1:1.0
|  |-- bAlternateSetting
|  |-- bInterfaceClass
|  |-- bInterfaceNumber
|  |-- bInterfaceProtocol
|  |-- bInterfaceSubClass
|  |-- bNumEndpoints
|  |-- detach_state
|  |-- iInterface
|  `-- power
|  `-- state
|-- bConfigurationValue
```

USB and sysfs

- naming convention:
- the first USB device is *root hub*,
- this is the USB controller (controller),
- it is a bridge between the PCI bus and USB bus,
- each root hub has a unique number,
- in our example *usb2*,
- each device assumes the hub number as its initial number,
- follows the - character,

USB and sysfs

- followed by the port number where the device is connected,
- in our example *1*,
- device name is: *2-1*,
- interface requires its input into sysfs,
- name for our interface is *:1.0*,
- represents the first configuration on the *0* interface,

USB and sysfs

- *root_hub-hub_port:config.interface*,
- *root_hub-hub_port-hub_port:config.interface*.

USB Urbs

- *urb* – USB request block,
- the code communicates with all devices via the cables,
- urb is described in the *struct urb* structure,
- communication takes place asynchronously,
- communication is between the driver and the USB endpoint.

USB Urbs, life cycle

- it is made by a USB driver.
- urb is assigned a specific endpoint of the USB device,
- urb is sent to the USB core,
- urb is sent to the specified USB driver controller in the USB core,
- it is processed by the USB controller driver that sends forward a USB transfer device,
- when the urb is finished, the USB controller driver informs the USB device driver.

Create and destroy urb

- function to create an urb is called: *usb_alloc_urb*.

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

- *iso_packets* – the number of isochronous packets that this hub should have,
- if we do not use isochronous urb, set this value to 0,
- *mem_flags* - similar to kmalloc.

Create and destroy urb

- initialise urb,
- depends on urb type,
- free with function:

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

- *urb* – pointer to urb structure.

Send urb

```
int usb_submit_urb(struct urb *urb, int mem_flags);
```

- *urb* – pointer to urb structure,

Urb ends

- function call *usb_submit_urb* ends:
  - urb has been successfully sent to device,
    - for OUT urb the data was sucessfully sent,
    - for IN urb the data was sucessfully received,
    - variable status is set to 0,
  - an error occurred, the variable status is set to the value of the error,
  - urb has been unlinked from the USB core:
    - driver cancels the urb,
    - device was removed from the system.
    - naprava je bila umaknjena iz sistema.

USB driver

- similar to a PCI driver,
- driver registers its driver object to the USB system,
- which device supports the driver,
- defined in the structure *struct usb _device_id*.

Register USB driver

- multiple fields, needed only 5:

```
static struct usb_driver skel_driver = {
.owner = THIS_MODULE,
.name = "skeleton",
.id_table = skel_table,
.probe = skel_probe,
.disconnect = skel_disconnect,
}
```

Register USB driver

- structure *struct usb_driver* is registered with function *usb_register_driver*,
- usually in the init phase,

```
static int __init usb_skel_init(void)
{
  int result;
  /* register this driver with the USB subsystem */
  result = usb_register(&skel_driver);
  if (result)
    err("usb_register failed. Error number %d", result);
  return result;
}
```

Register USB driver

- cleanup at the end:

```
static void __exit usb_skel_exit(void)
{
  /* deregister this driver with the USB subsystem */
  usb_deregister(&skel_driver);
}
```

Sending and controlling the urb

- send data example (write function),
- allocate urb:

```
urb = usb_alloc_urb(0, GFP_KERNEL);
if (!urb) {
  retval = -ENOMEM;
  goto error;
}
```

Sending and controlling the urb

- allocate DMA buffer,
- copy content from user space

```
buf = usb_buffer_alloc(dev->udev, count, GFP_KERNEL, &urb->
if (!buf) {
  retval = -ENOMEM;
  goto error;
}
if (copy_from_user(buf, user_buffer, count)) {
  retval = -EFAULT;
  goto error;
}
```

Sending and controlling the urb

- initialize urb:

```
/* initialize the urb properly */
usb_fill_bulk_urb(urb, dev->udev,
  usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
  buf, count, skel_write_bulk_callback, dev);
urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
```

Sending and controlling the urb

- send urb (to USB core):

```
/* send the data out the bulk port */
retval = usb_submit_urb(urb, GFP_KERNEL);
if (retval) {
  err("%s - failed submitting write urb, error %d", __FUNC
  goto error;
}
```

Sending and controlling the urb

- urb callback function,
- called ba USB core,

```
static void skel_write_bulk_callback(struct urb *urb, struc
{
/* sync/async unlink faults aren't errors */
  if (urb->status &&
    !(urb->status = = -ENOENT ||
    urb->status = = -ECONNRESET ||
    urb->status = = -ESHUTDOWN)) {
    dbg("%s - nonzero write bulk status received: %d",
      __FUNCTION__, urb->status);
  }
/* free up our allocated buffer */
```