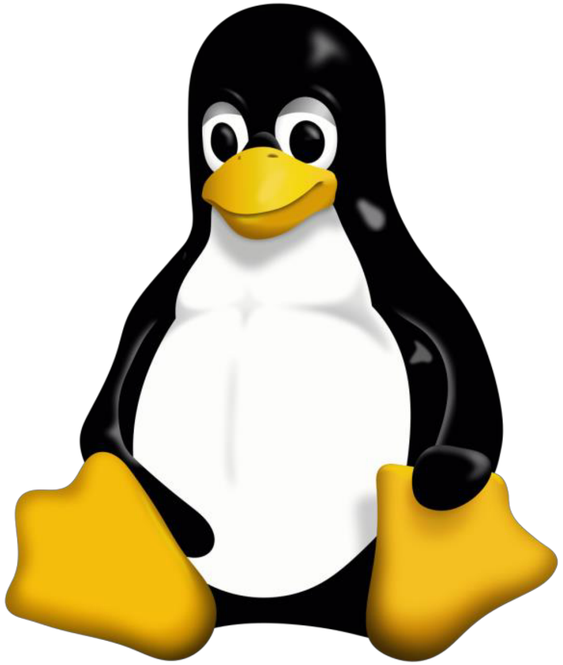


Linux Character Device Driver

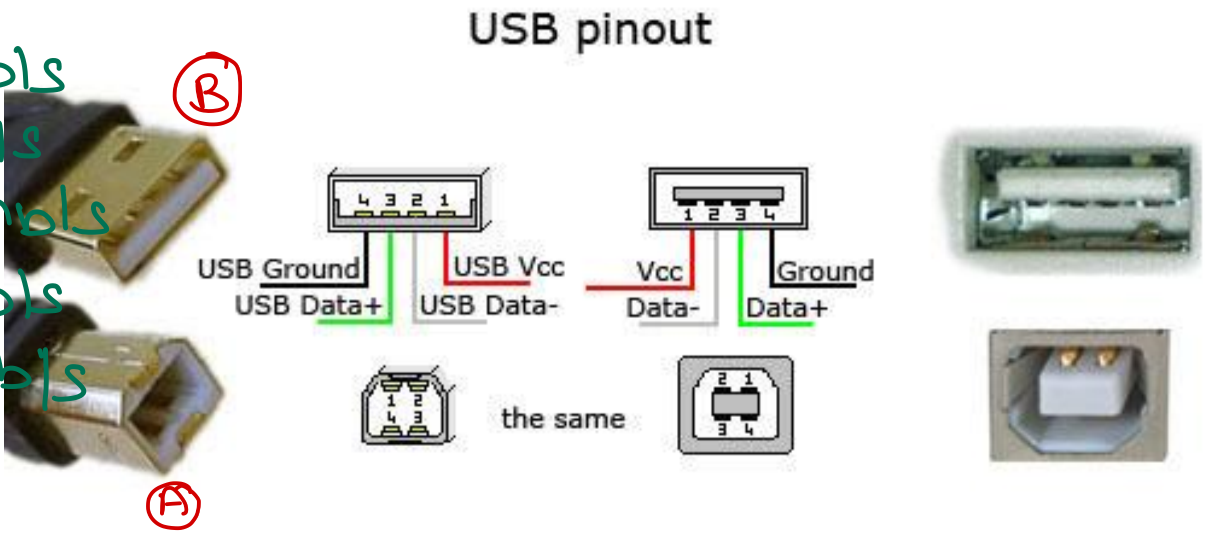
Sunbeam Infotech

Universal Serial Bus

- USB is a bus specification/standard.
- USB was invented to replace many other different types of buses like PS/2, Audio, Network, Serial/Parallel port, ...
- USB bus is 4-wire bus:
  - Vcc: +5V
  - Gnd: 0
  - Data+ : Data +ve
  - Data- : Data -ve
- USB is differential bus & hence immune to noise.
- Since bus has only wires, we can send any type of data including files, audio, video, control signals, ...
- USB is supported on many architectures including embedded (e.g. ARM, AVR, ...)
- Typically USB is connected to PC via PCI bus.

} TTL

USB 1.0 → LS → 1.5 Mb/s
USB 1.1 → FS → 12 Mb/s
USB 2.0 → HS → 480 Mb/s
USB 3.0 → SS → 5 Gb/s
USB 3.1 → SS → 10 Gb/s



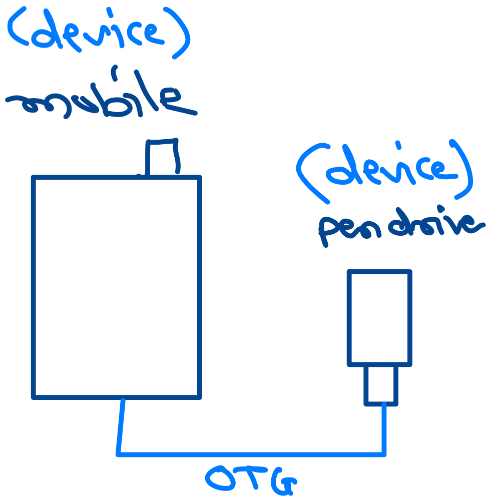
USB is a serial bus. It uses 4 shielded wires: two for power (+5v & GND) and two for differential data signals (labelled as D+ and D- in pinout)

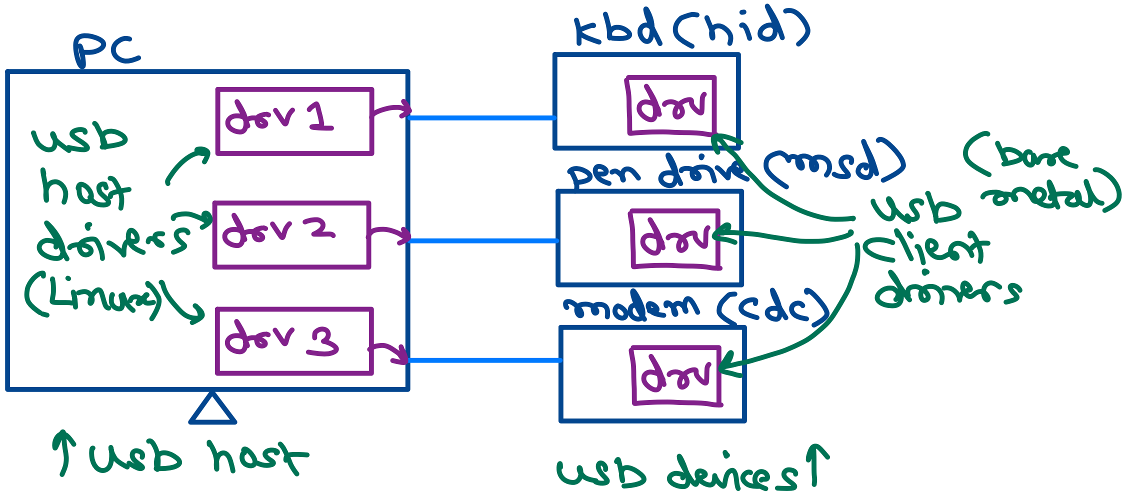
http://pinouts.ru/Slots/USB_pinout.shtml

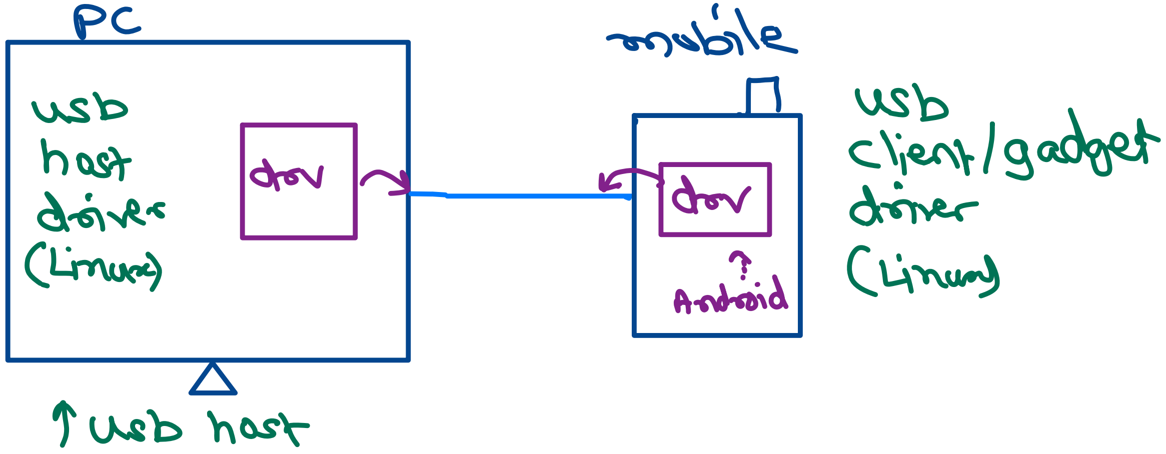
USB controller is implemented in hardware for most of microcontroller e.g. STM32, AM335x, ...

USB Drivers

- USB Host driver:
  - The driver runs on host machine (in Linux system).
  - Responsible for giving commands to the device and retrieving data from device.
  - Majority of drivers fall in this type.
  - e.g. Pen drive driver, Keyboard driver, Mouse driver, ...
- USB Client driver:
  - The driver runs in USB device (in Linux system).
  - Responsible for projecting the device as USB device to the host. Take commands from host & execute them.
  - Such drivers are also called as "USB Gadget driver".





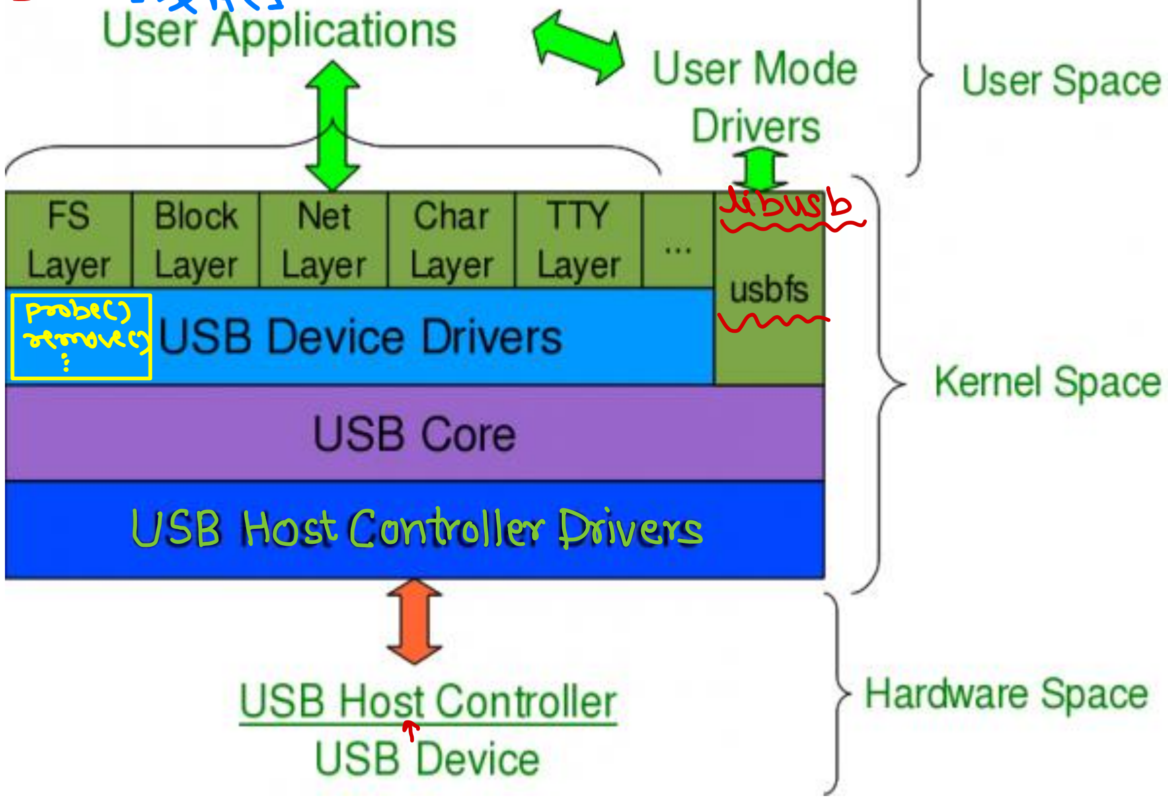


USB subsystem

- USB Host Controller Driver
  - HAL communicating with USB device, as per HCI.

USB 1.0 - UHCI
USB 1.1 - OHCI
USB 2.0 - EHCI
USB 3.0 - xHCI

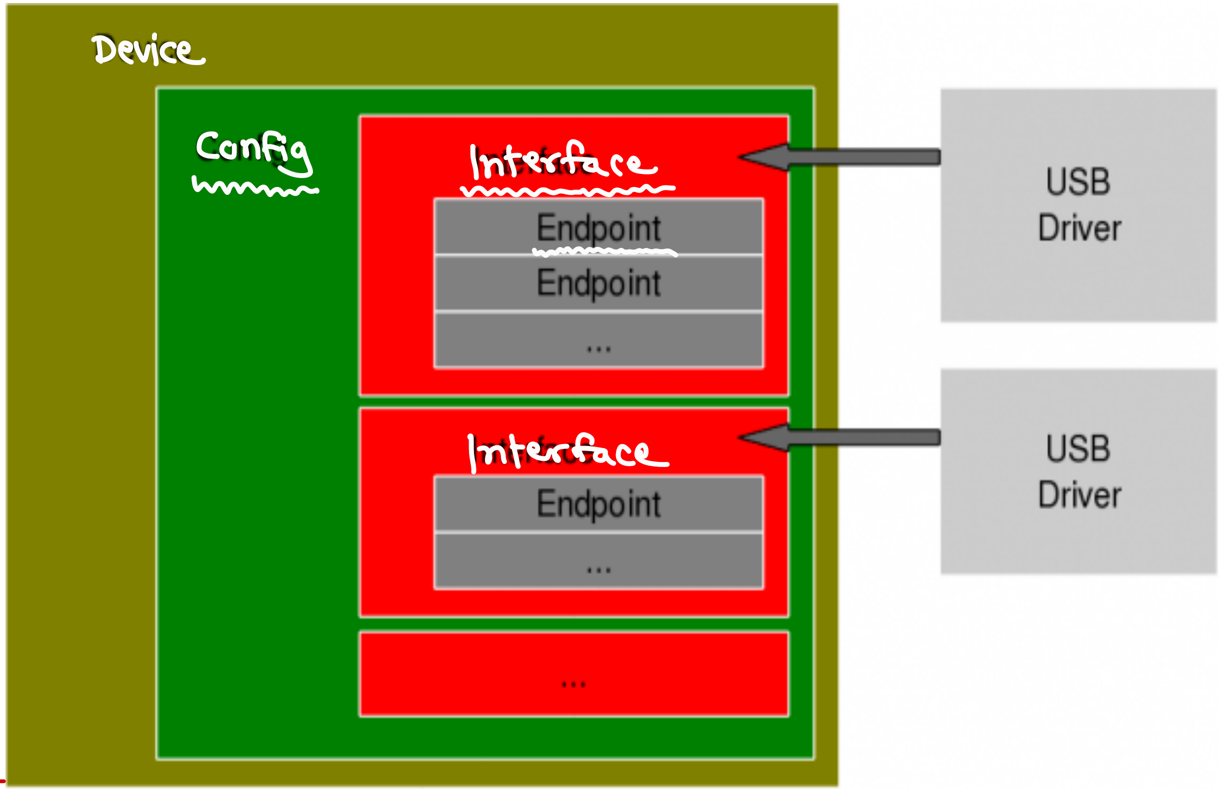
- USB Core
  - Core component for functioning of USB devices.
  - Responsible for giving commands to the Host Controller Driver & provide framework for USB drivers.
  - Invokes probe() and remove() functions of USB driver
  - Make detected USB device information available to them as "struct usb_device".
- USB Device Driver
  - USB Host device driver implementation.
- Rest of system can access USB driver.
- "usbfs" component
  - makes USB device info & communication available directly to user space under "/sys".
  - Any user space application can directly communicate with USB devices typically using libusb.
  - Such user space programs are referred as "user-space USB drivers".



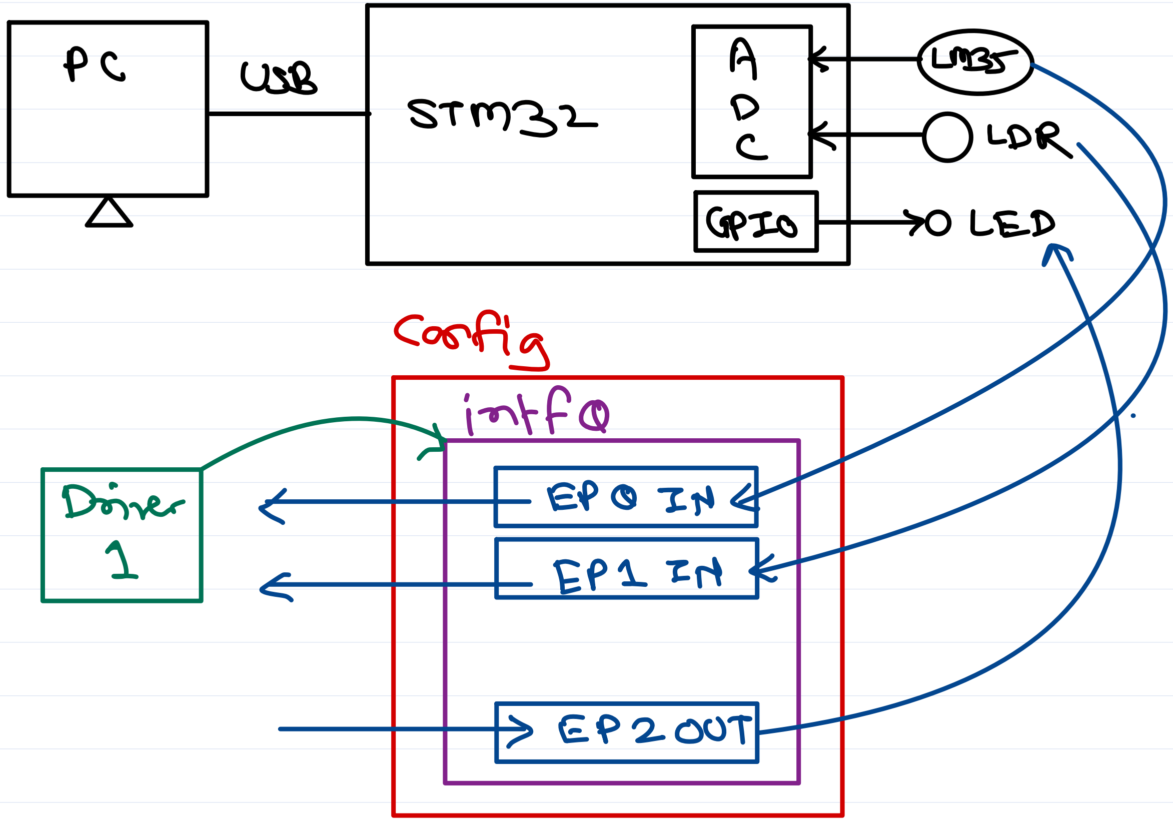
USB Device structure

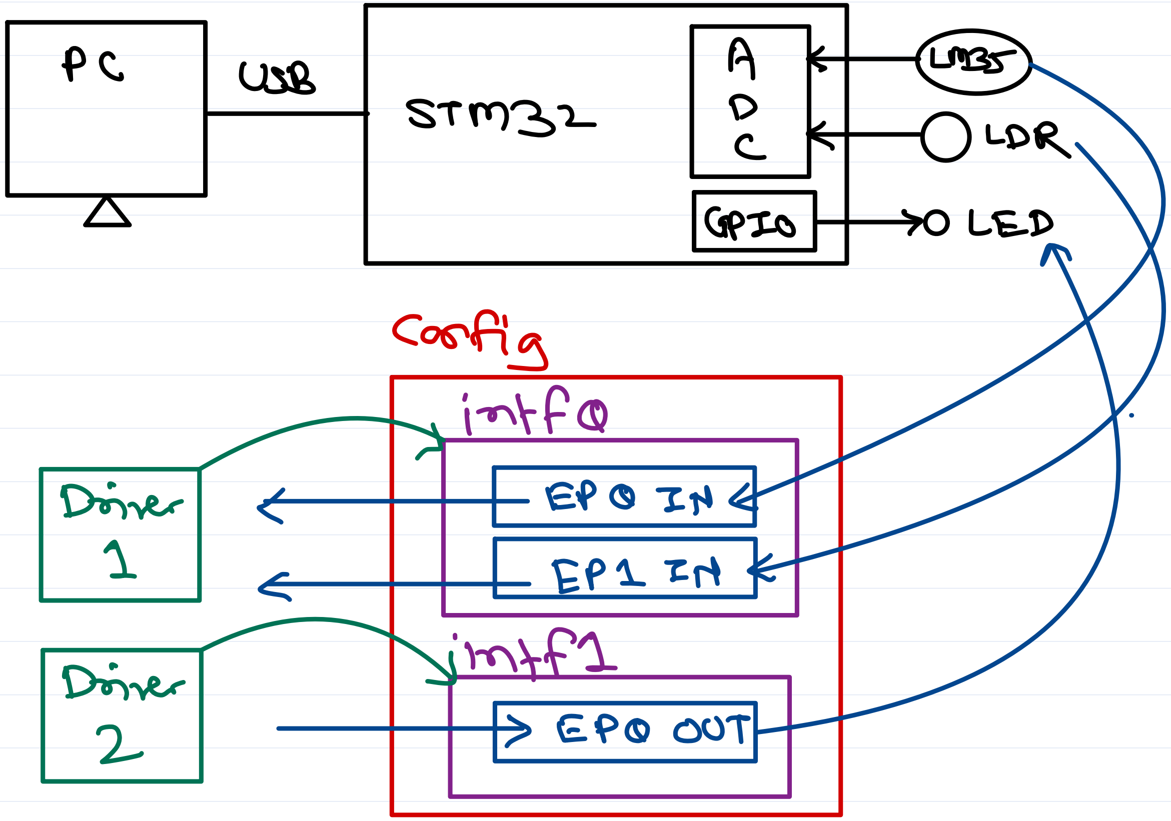
- USB device have one (or more) configurations.
  - Usually USB device have single config.
  - Typically config represent a class of device.
  - If device is multi-function (multi-class), then it will have multiple config.
  - e.g. USB device supporting firmware update, will do it via a separate config than its other functionalities.
- A configuration contains one or more interfaces.
  - Each interface provide different functionality.
  - e.g. Device providing mass storage and also providing audio via USB will have two interfaces.
  - There should be one driver per interface.
- An interface contains one or more endpoints.
  - Endpoints are also called as data pipes.
  - Endpoint is basic unit through which communication is done with device.
  - Endpoint is uni-directional. It can be IN or OUT.

CDC → com port
HID → keyboard, mouse
MSC → all storage
Audio → speaker
Custom →



USB device config

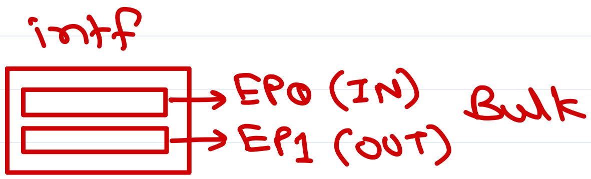




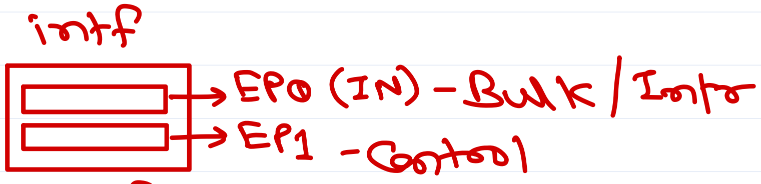
USB device class

USB Device Classes

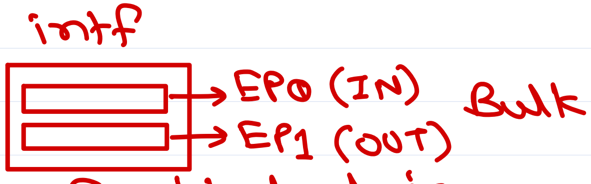
① CDC → like Serial port



② HID → for kbd, mouse, Joystick, etc.



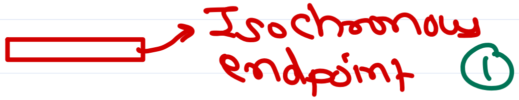
③ MSD → for all storage devices



for block device.

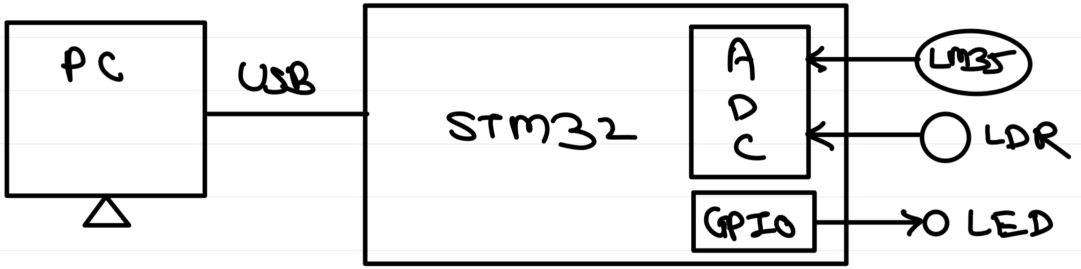
Bulk transfer = 512 bytes.

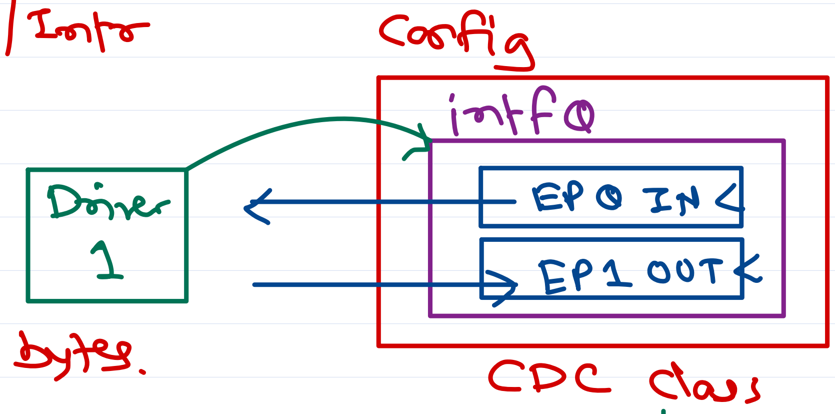
④ Audio → for audio devices like spkr, mic, ..



⑤ Custom

⑥ ...





① Driver OUT → "LDR" → Device gets LDR Reading

② Driver In ← LDR reading

① Driver OUT → "LM32" → Device gets LM35 Reading

② Driver In ← LM35 reading

① Driver OUT → "LED" → Device get state & change LED.

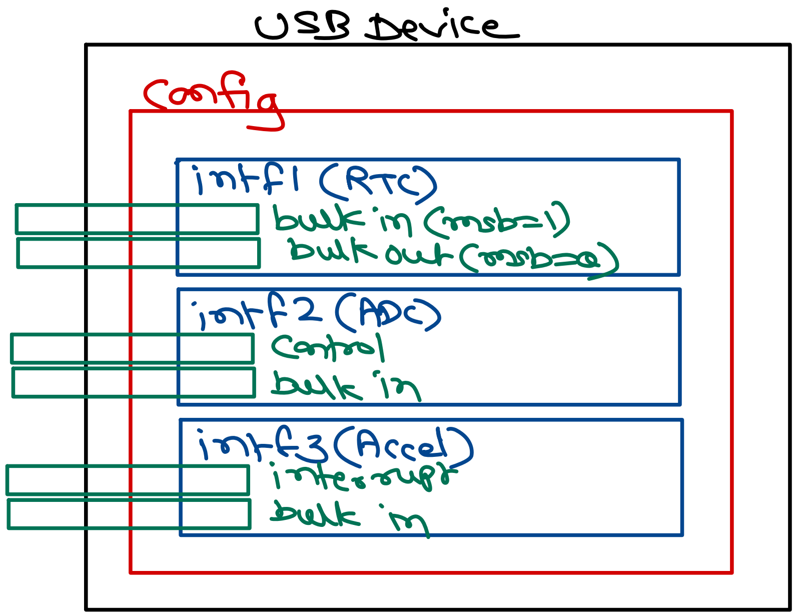
② Driver OUT → 1/0

USB Device structure

- USB Endpoints
  - Based of functionalities there are four types of endpoints:
    - ✓ Control
      - Control EP ~~must be there~~ in each interface.
      - Used for config or getting status. ✓
      - Small in size. ✓
      - USB core will guarantee of the bandwidth. ✓ (alloc buffer).
    - ✓ Interrupt
      - If device is generating interrupt which should be handled by host, then interrupt is passed via this EP to host. ✓
      - Small in size. ✓
      - USB core will guarantee of the bandwidth. ✓ (alloc buffer).
    - ✓ Bulk
      - Data transfer endpoint.
      - Can be IN or OUT. w.r.t. host.
      - Programmer need to allocate buffer for bulk endpoints. ✓ (usually 512 bytes).
    - ✓ Isochronous
      - Data transfer endpoint.
      - Ensures continuity of data transfer, but some data packets might be lost.
      - Mainly used for audio/video streaming.
      - Programmer need to allocate buffer for ~~bulk~~ isochronous endpoints.
  - Control & Interrupt EP are for device & device controller, while bulk & Isochronous EP are mainly for device driver.

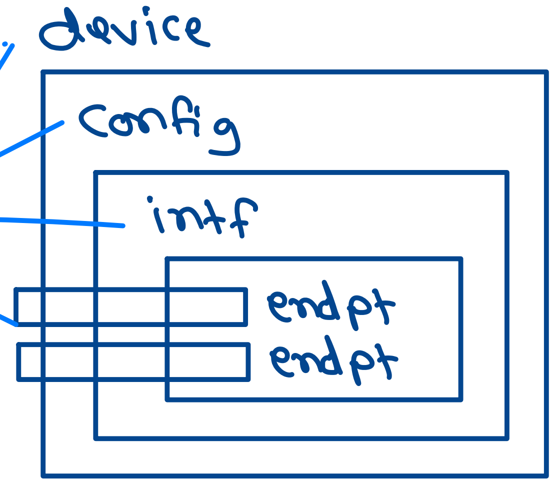
USB Bus Layout

- Tree like (hierarchical) structure.
- Bus → Hub → Ports → Devices.
- USB commands:
  - lsusb -t
  - lsusb -v
  - tree /sys/bus/usb/devices
    - a-b:c-d -- identifying the device (connection)
      - a - USB ~~root hub controller~~ bus
      - b - Port of hub
      - c - Config number
      - d - Interface number
        - For each interface there will be separate driver.
  - sudo tree /sys/bus/pci/devices/0000:00:1d.0 (on PC)
  - cat /proc/bus/usb/devices (Linux kernel 2.6)



USB device structures

- struct usb_host_endpoint
  - struct usb_endpoint_descriptor
    - bEndpointAddress (address & IN/OUT) → MSB=1 → MSB=0
    - bmAttributes (type) ← control, bulk, interrupt, isochronous
    - wMaxPacketSize (amount of data that can be handled by this device)
    - bInterval (time in ms between interrupt requests)
- struct usb_interface
  - struct usb_host_interface *altsetting (set of endpoint configs)
  - unsigned num_altsetting (number of alternate settings)
  - struct usb_host_interface *cur_altsetting (current active endpoint configs).
  - minor (minor number assigned to interface by USB core – valid for usb_register_dev())
- struct usb_host_config
- struct usb_device
  - descriptor, ep_in[], ep_out[], actconfig, …id, …
- interface_to_usbdev(): get usb_device* from usb_interface*

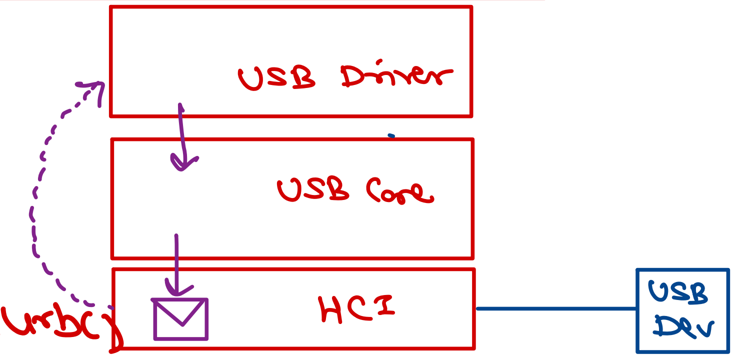


usb_device_id
- → vendor id
- → product id
- → device class
- → device subclass
- → subsystem

USB Request Block

→ like usb packets through which all usb ops are carried out. } sent by host to device.

- struct urb – for asynchronous transfer the data from/to USB endpoint.
  - struct usb_device *dev (device to which this URB is to be sent).
  - unsigned int pipe (EP information using usb_sndbulkpipe(), usb_rcvbulkpipe(), …);
  - void *transfer_buffer (send/receive data from device – to be allocated using kmalloc()).
  - int transfer_buffer_length (length of allocated buffer).
  - usb_complete_t complete (completion handler to free/reuse URB). – callback
- Same URB can be reused for multiple data transfer or new URB created for each transfer.
- Endpoint can handle queue of URB.
- URB life cycle
  - Created by a USB device driver. → usb_alloc_urb() → usb_fill_bulk_urb()
  - Assigned to a specific endpoint of a specific USB device.
  - Submitted to the USB core, by the USB device driver. → usb_submit_urb()
  - Submitted to the specific USB host controller driver for the specified device by the USB core.
  - Processed by the USB host controller driver that makes a USB transfer to the device.
  - When the URB is completed, the USB host controller driver notifies the USB device driver. → by calling callback.



usb_free_urb() ← free the urb ←

URB functions

- struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
- void usb_free_urb(struct urb *urb); → usually in completion callback.
- void usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev, unsigned int pipe, void *transfer_buffer, int buffer_length, usb_complete_t complete, void *context);
- void usb_fill_control_urb(struct urb *urb, struct usb_device *dev, unsigned int pipe, void *transfer_buffer, int buffer_length, usb_complete_t complete, void *context);
- int usb_submit_urb(struct urb *urb, int mem_flags);
- int usb_kill_urb(struct urb *urb); – to cancel urb
- int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe, void *data, int len, int *actual_length, int timeout);
  - arg1: device to which bulk msg to send.
  - arg2: pipe -- endpoint number
  - arg3 & 4: data buffer & its length
  - arg5: out param -- number of bytes transferred
  - arg6: waiting time for the transfer

↖ internally use urb (helper fn).

USB driver

vendor id, device id, class, subcls, ...

array

- Declare table of usb_device_id and initialize it using USB_DEVICE() to USB devices to be handled.
- Export this table to kernel using MODULE_DEVICE_TABLE(usb, table);
- Declare and initialize usb_driver structure with probe and remove functions (globally).
- In module initialization, register usb driver using usb_register().
- In module exit, unregister usb driver using ~~usb_degister()~~. usb_deregister().
- In device probe operation initialize usb_class_driver with device name and device file_operations. Then register usb device interface using usb_register_dev().
- In device remove operation, register usb device interface using usb_deregister_dev().
- Implement USB device operation. Typically read/write operation can be done using URB or using usb_bulk_msg().

open()
close()
read()
write()

usb_driver operations

✓① probe() – called by core when device arrived.

✓② disconnect() – called by core when device detached.

③ ioctl() – called when user space appln calls ioctl() - used for usb hub.

④ suspend() – called by core when device is suspended due to idle state.

⑤ resume() – called by core when device is resumed.

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>