

Linux Device Driver

Sunbeam Infotech



Slab allocator

cat /proc/slabinfo

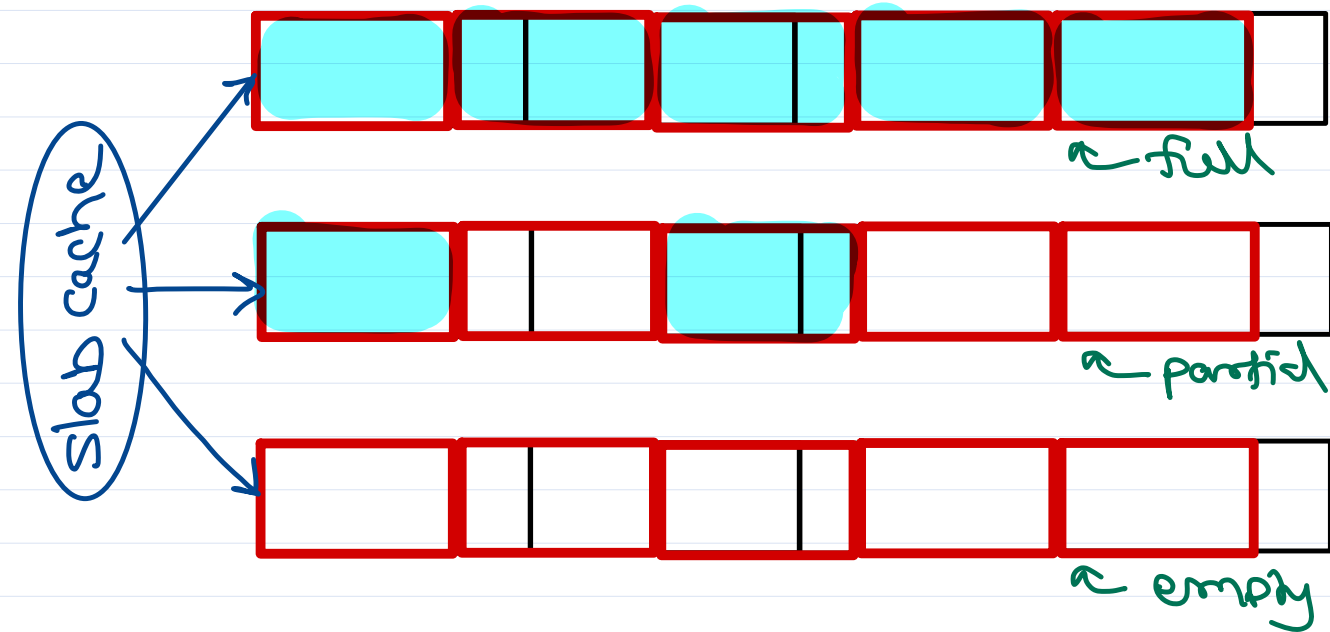
```
ptr = kmalloc(size, mask);  
:  
kfree(ptr);
```

↳ GFP_XYZ

- ✓ allocates smaller contiguous block from slab cache.
- ✓ lesser internal fragmentation.
- ✓ faster allocation for commonly required (well-known) objects.
e.g. task_struct, mm_struct, inode, ...

Example:

- ✓ object size = 3 kB
- ✓ pages per slab = 4
- ✓ objects per slab = 5
- ✓ slabs per cache = 3



- ✓ A slab is set of contiguous pages.
- ✓ Slab allocator allocates slab - using buddy allocator.



vmalloc()

In user space, contiguous virtual addresses are allocated by `mmap()`. It internally calls `vmalloc()`.

```
ptr = vmalloc(size);  
:  
vfree(ptr);
```

- ✓ Allocates contiguous range of virtual address.
 - ✓ allocates a new VAD (`vm_area_struct`).
 - ✓ allocates corresponding page table entries.



Page replacement

If RAM is full, existing page(s) needs to be swapped out - to make space available for new pages.

Local Replacement:

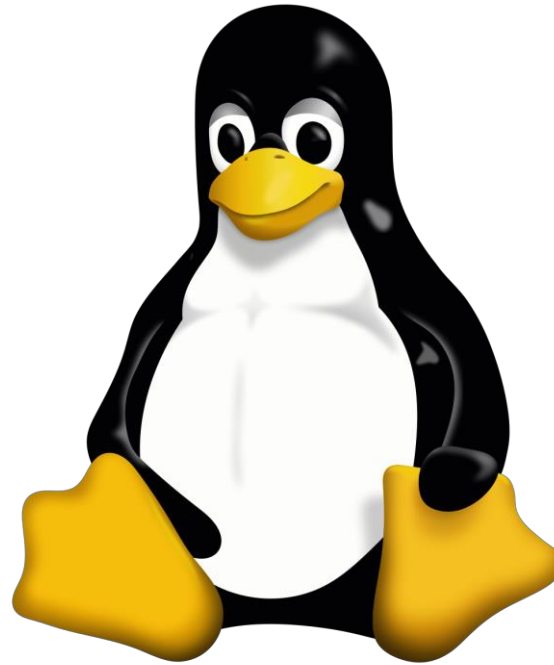
- To make space avail for process swap out a few pages of that process itself.

Global Replacement:

- To make space avail for process swap out a few pages of any process.

Kernel runs a daemon process which ensures that avail memory is not going below a threshold level. If needed, it will swap out pages from RAM (as per page replacement algo). This process is called as "page stealing process". In Linux Kernel, this is done by "kswapd" daemon.





Linux PCI Device Driver

Sunbeam Infotech

Not in Syllabus



PCI

- PCI is a specification/standard for connecting devices to PC.
- PCI is developed to overcome problems of ISA (Industry Standard Architecture).
- Design Goals:
 - Support for high transfer bandwidths to cater to multimedia applications with large data streams
 - Simple and easily automated configuration of attached peripherals.
 - Platform independence; that is, not tied to a specific processor type or system platform.
- The modern PCI bus also support hot-plugging.
- PCI bus is used in x86, x64, SPARC, PowerPC & Alpha.

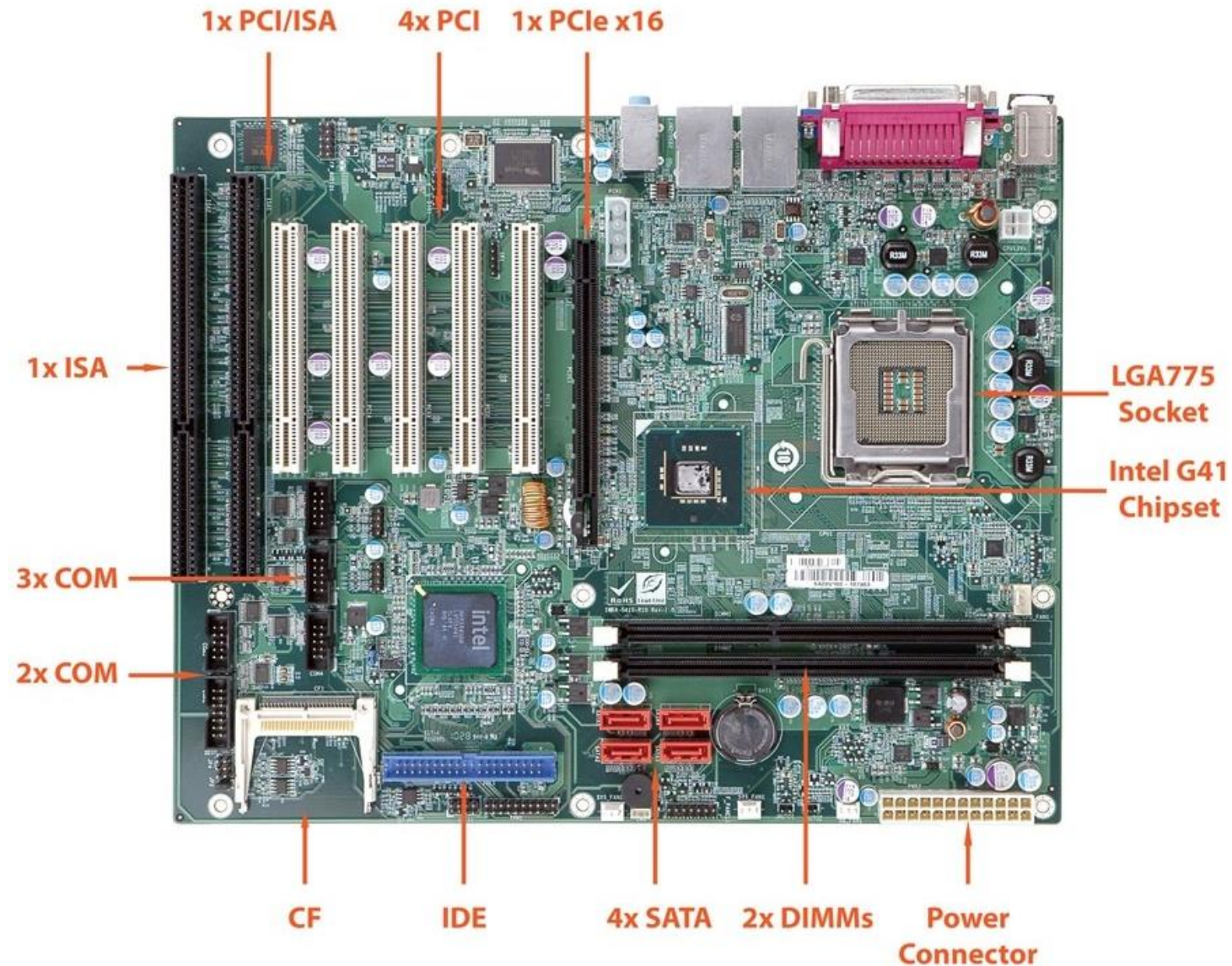


ISA

- IBM developed PC.
- IBM developed micro-channel to connect hw card (expansion cards) to PC. This was patented & proprietary.
- Hw vendors develop a new way of connecting hw cards to PC -- ISA. Later on this adopted by IBM.
- ISA arch was too simple, but not hassle free.
 - Expansion cards were configured using jumpers.
 - Device addressing is mapped to physical address space.
 - Not very good performance with 32-bit CPU.
- Most common use of ISA was AGP -- Accelerated Graphics Port.



Motherboard – PCI slots

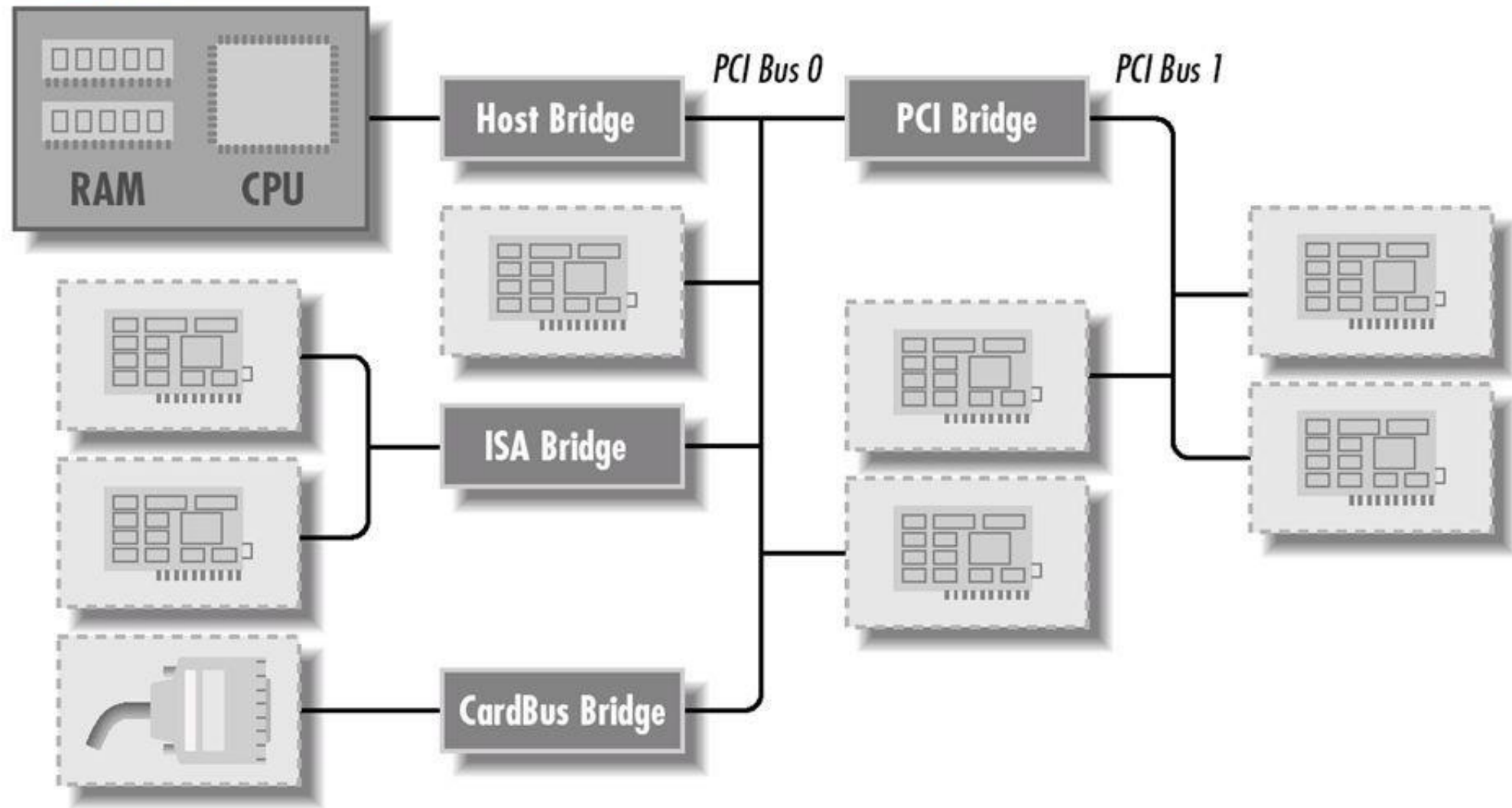


PCI – Advantages

- Better platform/architecture independence.
 - PCI standard is not arch dependent.
 - PCI nowadays used with x86, x64, PowerPC, Alpha, SPARC.
 - PCI works well with Desktop, Laptops & Workstations.
- Better data transfer rate.
 - Standard data transfer rate.
 - PCI 0.1 -- 25 MHz, 33 MHz, 66 MHz.
 - PCI 2.0 -- 133 MHz.
 - PCI 3.0 – 266 MHz.
- Better acceptance for new devices.
 - Jumper-less config.
 - No need of probing the devices i.e. all PCI devices are auto detected & initialized by BIOS/PROM/NVRAM during booting.



PCI – Bus layout



PCI bus architecture

- Multiple PCI buses can be connected to system.
 - PCI buses are connected via PCI bridges.
 - Max 256 buses can be connected to a system.
 - For big systems this may not be sufficient.
 - Linux added concept of PCI domain. A domain can have up to 256 buses. System can have multiple domains.
 - Domain is identified by 16-bit number - domain id.
 - Bus is identified by 8-bit number - bus id.
 - PCI buses are hierarchical -- tree like structure.
- Multiple PCI devices can be connected on bus.
 - Max 32 devices can be connected on a bus.
 - Device is identified by 5-bit number - device id.
- Multiple functionalities can be provided into a single PCI device.
 - Max 8 functionalities can be provided on a device.
 - It depends on device manufacturing.
 - e.g. A network PCI device with 2 network cards -- 2 fns.
 - e.g. A PCI card with 2 serial & 1 parallel port -- 3 fns.
 - Device function is identified by 3-bit number - function id.
 - Usually different device functions need different drivers.



PCI bus architecture

- Each device on bus identified by 3 numbers.
 - bus number: the bus to which the device is assigned.
 - numbering starts from 0.
 - PCI specs allows max 255 buses in a system.
 - Multiple buses are connected in a bridge.
 - device number: a unique identifying number within a bus
 - PCI specs allows max 32 devices on a bus.
 - function number: identify expansion device(s) on single plug-in card.
 - e.g. Dual UART & a Parallel port on single PCI card.
 - IDE controller, USB controller, Network are all connected to PCI bus.
 - PCI specs allows max 8 functions on a device.
 - domain number: identifies domain -- Linux only.
 - One domain can have max 255 buses.
 - Domain number of 16 bit.
- PCI device number
 - Each device is uniquely identified by a 16-bit number, where 8 bits are reserved for the bus number, 5 for the device number, and 3 for the function number.
 - 0000:00:14.0
 - 0000 -- domain
 - 00 -- bus
 - 14 -- device
 - 0 -- function



PCI bus architecture

- PCI bus related commands
 - `lspci -t`
 - `lspci -t -v`
 - `lspci -vv`
 - `tree /sys/bus/pci/devices/`
 - `cat /proc/bus/pci/devices`
 - `tree /proc/bus/pci`



PCI address space

- A typical PCI device will have IO ports, memory regions & config space.
- If device have multiple functions, IO ports & memory locations will be same, but config for each function is different.
- The IO ports are always 32 bit.
 - Provides a maximum of 4 GB for the port addresses used to communicate with the device.
- The memory addresses are 32 bit or 64 bit.
 - 64-bit is supported only on 64-bit CPUs.
- If device to be used on multiple arch, IO ports are not recommended. Because certain archs doesn't support IO addresses.
- The IO ports & memory addresses can be accessed using IO port kernel fns e.g. `inb()`, `outb()`, `inw()`, `outw()`, `inl()`, `outl()`, ...

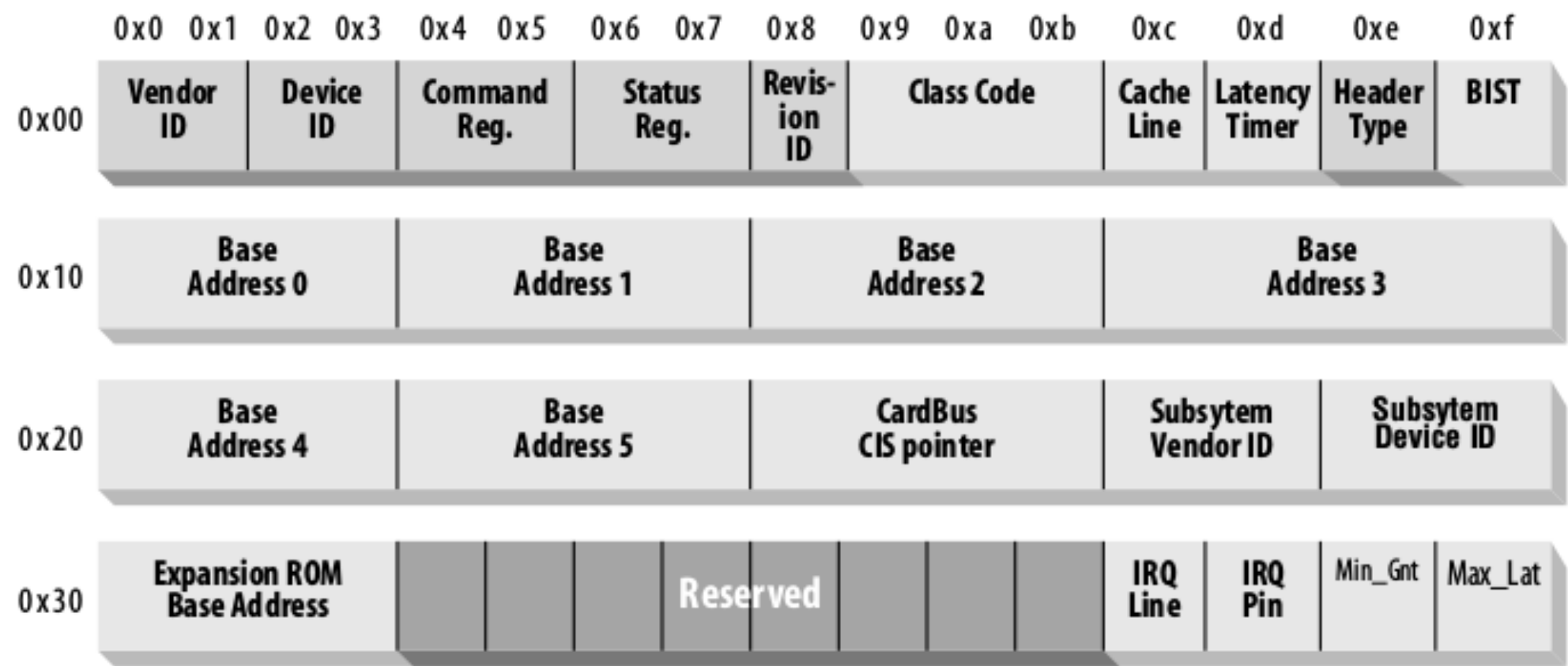




PCI address space

- The config space for each device function is 256 bytes.
- It is not directly accessible from CPU. For them special kernel fns are provided.
 - `pci_bus_read_config_byte()` \leftarrow `pci_read_config_byte()`
 - `pci_bus_read_config_word()` \leftarrow `pci_read_config_word()`
 - `pci_bus_read_config_dword()` \leftarrow `pci_read_config_dword()`
 - `pci_bus_write_config_byte()` \leftarrow `pci_write_config_byte()`
 - `pci_bus_write_config_word()` \leftarrow `pci_write_config_word()`
 - `pci_bus_write_config_dword()` \leftarrow `pci_write_config_dword()`
- The reading & writing these registers is called as "config transactions".
- PCI devices (config & addresses) are initialized during boot by BIOS by config transactions.
- Device drivers need not to initialize or change these addresses. If modified, may cause address clashing, which leads to kernel crash.



PCI – Configuration



-  - Required Register
-  - Optional Register

PCI configuration

- The configuration space contains detailed information on the type and characteristics of the individual devices. The PCI configuration space consists of 256 bytes for each device function.
 - Accessible for read from sysfs: `xxd /sys/bus/pci/devices/<device-addr>/config`
 - All config registers are little endian. Programmer should read in proper byte order.
- During booting BIOS or kernel assigns IO & memory space to each IO device by accessing its config registers. Device driver need to only deal with config not with IO & memory space.
- The first 64 bytes are standard -- irrespective of arch & device. The remaining bytes depends on standard & device.
- Configuration registers
 - vendor id (2) -- represent manufacturer of device.
 - manufacturer need to register with PCI association.
 - device id (2) -- represent device.
 - class (3) -- group(1) + class(2)
 - Subsystem VendorId (2)
 - Subsystem DeviceId (2)



PCI Configuration

- Configuration registers
 - IRQ Line (1) -- interrupt line assigned to the PCI device. The interrupt lines (like memory & IO addresses) are allocated to each PCI device during booting. Not hardcoded into the device.
 - IRQ Pin (1) -- when 0, no interrupts are given by this device. when non-zero, some interrupt line is needed for this device
 - BAR: Base Address Registers
 - BAR0 to BAR5 -- 6 32-bit bars
 - OR BAR0 to BAR2 -- 3 64-bit bars
 - 32-bit BAR -- memory or IO bar.
 - Memory bar
 - bit0 -- memory bar(1)
 - bit1:2 -- memory address type
 - 0 -- 32-bit addr
 - 1 -- 32-bit addr (< 1MB)
 - 2 -- 64-bit addr
 - bit3 -- prefetchable(0/1)
 - bit4-31
 - memory address (16 byte aligned)
 - IO bar
 - bit0 -- IO bar(0)



PCI Configuration

- Configuration registers
 - revision id (1) -- represent version of device.
 - command (2) -- command for pci device (wr)
 - status (2) -- command of pci device (rd)
 - cache line (1) -- cache to be used or not.
 - timer latency (1)
 - header type (1)
 - BIST (1) -- Built-In Self Test.
 - CardBus Pointer
 - Expansion RAM base address
 - Reserved
 - Min Gnt (1)
 - Max Lat (1)



PCI device addressing/identification

- struct pci_device_id
 - device, vendor, subvendor, subdevice, class, class_mask, driver_data
- To init pci_device_id struct two macros:
 - PCI_DEVICE(vendorId, deviceId) OR PCI_DEVICE_CLASS(class, class_mask)
- Linux PCI Device Driver must contain array of device id, which will be handled by that device.

```
#define MY_DEV_VENDOR    0x1234
#define MY_DEVICE    0x4321
static struct pci_device_id ids[] = {
    { PCI_DEVICE(MY_DEV_VENDOR, MY_DEVICE) },
    { 0 } // last entry in this array must be zero.
};
```
- Device Driver should inform these ids to the kernel using macro.

```
MODULE_DEVICE_TABLE(pci, ids);
```
- This macro internally expose array with a special name i.e. __mod_pci_device_table.
- If this module is present under /lib/modules/<kernel-version>/, then "depmod" command generates a "modules.pcimap" file. This file contains device id & corresponding driver module name. During booting the drivers from this file are loaded into memory.



PCI driver initialization & registration

- Each PCI device must have a PCI driver -- struct pci_driver.
- struct pci_driver members
 - name -- visible /sys/bus/pci/drivers/
 - id_table -- table of PCI ids handled by this driver.
 - (*probe)() -- method is called by kernel (PCI core) when it found a device that can be handled by this device driver.
 - (*remove)() -- method is called by kernel (PCI core) when it device is to be removed.
 - (*suspend)() -- optional
 - (*resume)() -- optional
- This pci_driver object is created & initialized globally.

```
struct pci_driver my_driver = {  
    .name = "mydev",  
    .id_table = ids,  
    .probe = my_probe,  
    .remove = my_remove  
};
```
- This driver object registered with kernel during module_init() using pci_register_driver(&my_driver);
- This driver object unregistered from kernel during module_exit() using pci_unregister_driver(&my_driver);



PCI driver initialization & registration

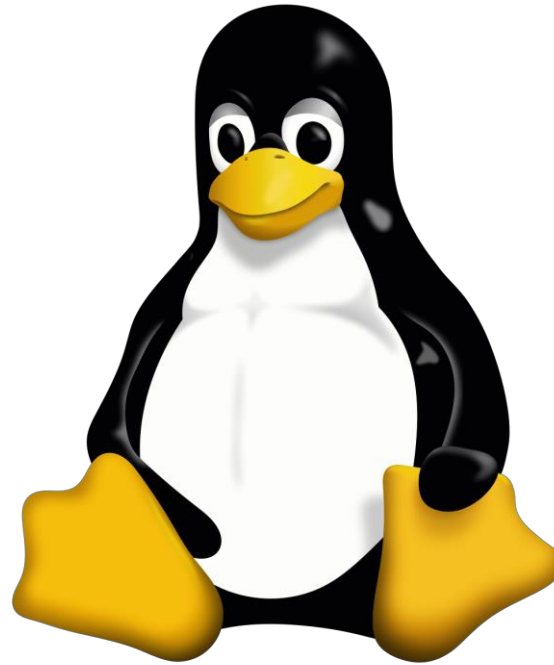
- probe() operation
 - It is called at booting or when driver inserted into the system (insmod).
 - Typically kernel call probe() only for those devices, which are registered by the driver using MODULE_DEVICE_TABLE().
 - This function have two args:
 - struct pci_dev *dev -- info about pci device.
 - struct pci_device_id *id -- pci device id.
 - This function should return:
 - 0 if device can be handled by this driver.
 - -ve if device cannot be handled by this driver.
 - If device can be handled by the driver, driver should initialize the device by taking its memory/io addresses from bars.
 - This initialization also includes creating cdev or block_device struct & adding into the systems with appropriate char device/block device operations.
 - This init also includes creating device files for the device.
 - This method must call pci_device_enable() to enable the device.
- remove() operation
 - This method is called during shutdown or while removing driver (rmmod).
 - This method should perform de-initialization (as init is done in probe()).



PCI device

- IO/Memory address
 - These addresses are available from BAR.
 - Note BAR reg's depends on device. The info in BAR will change from device to device. Refer datasheet of the device, to get info about the BAR.
 - It is read using
 - `addr = pci_resource_start(dev, barNum);`
 - The return address may be IO port or Memory address depending on type BAR OR may be different depending on device (as mentioned datasheet).
 - `pci_resource_end()` returns end address.
 - The IO addresses can be used after acquiring them using `request_region()`.
- Device interrupt
 - If PCI device is supporting interrupts, then interrupts must be handled using `request_irq()` like any other interrupts.
 - The IRQ number can be read using `pci_read_config_byte()` or directly using `dev->irq_no`.
- `struct pci_dev`
 - This struct is initialized by the kernel (PCI core) by reading config space of the device.
 - It is passed as arg to the `probe()` & `remove()` methods.





Linux USB Device Driver

Sunbeam Infotech

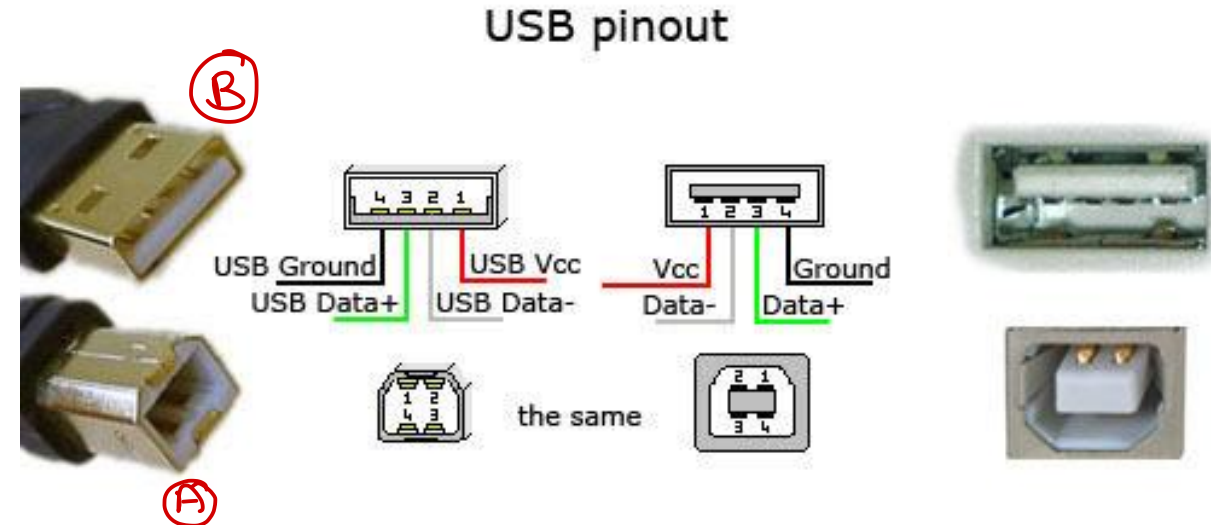


Universal Serial Bus

- 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.

USB 1.0 → FS- USB 1.1 → 12 MHz
HS- USB 2.0 → 48 MHz
USB 3.0 →

} TTL



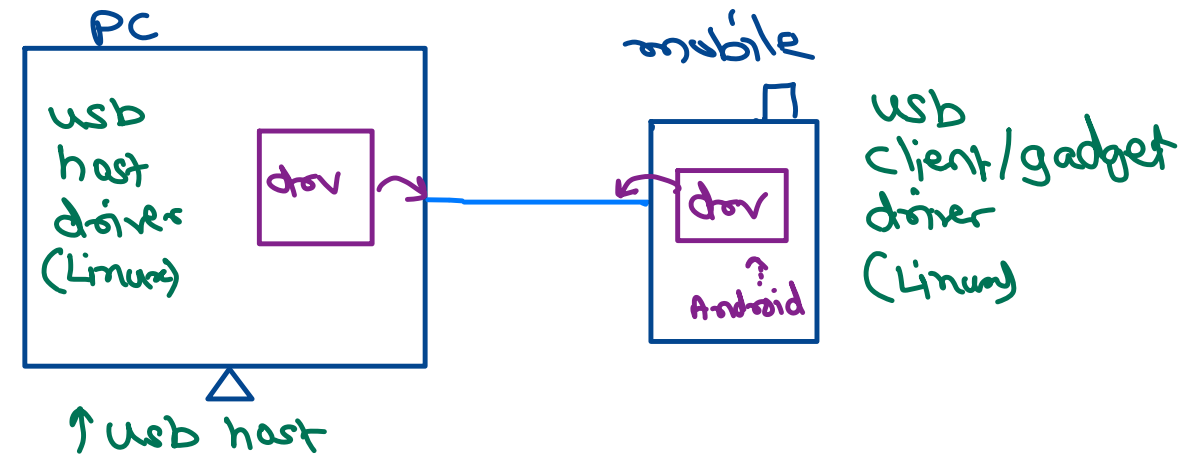
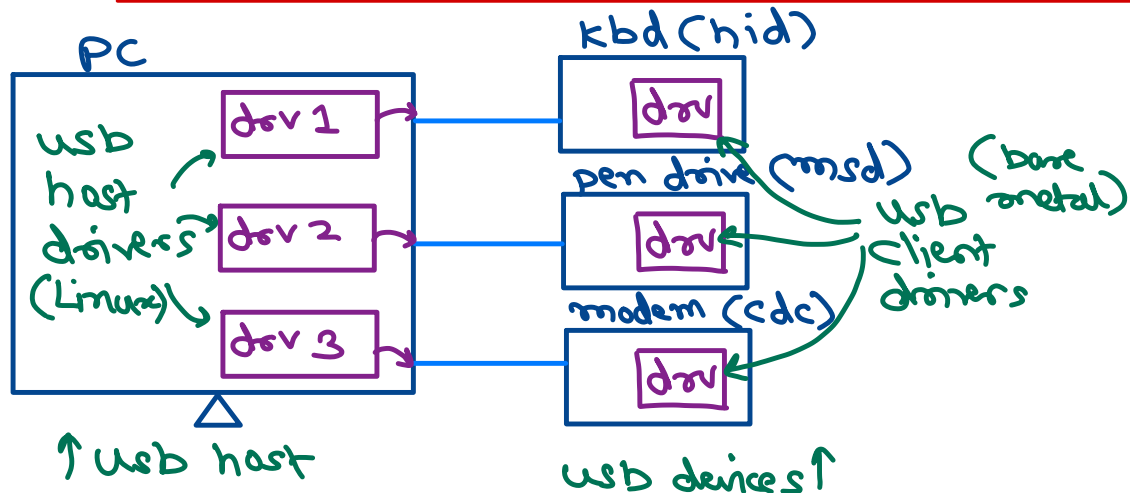
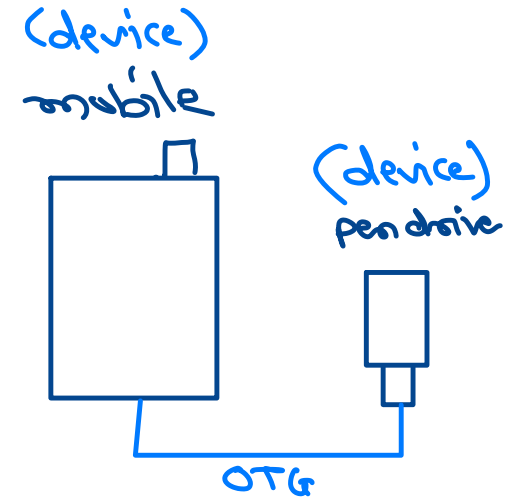
USB is a serial bus. It uses 4 shielded wires: two for power (+5v & GND) and two for differential data signals (labelled as D+ and D- in pinout)

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



USB Drivers

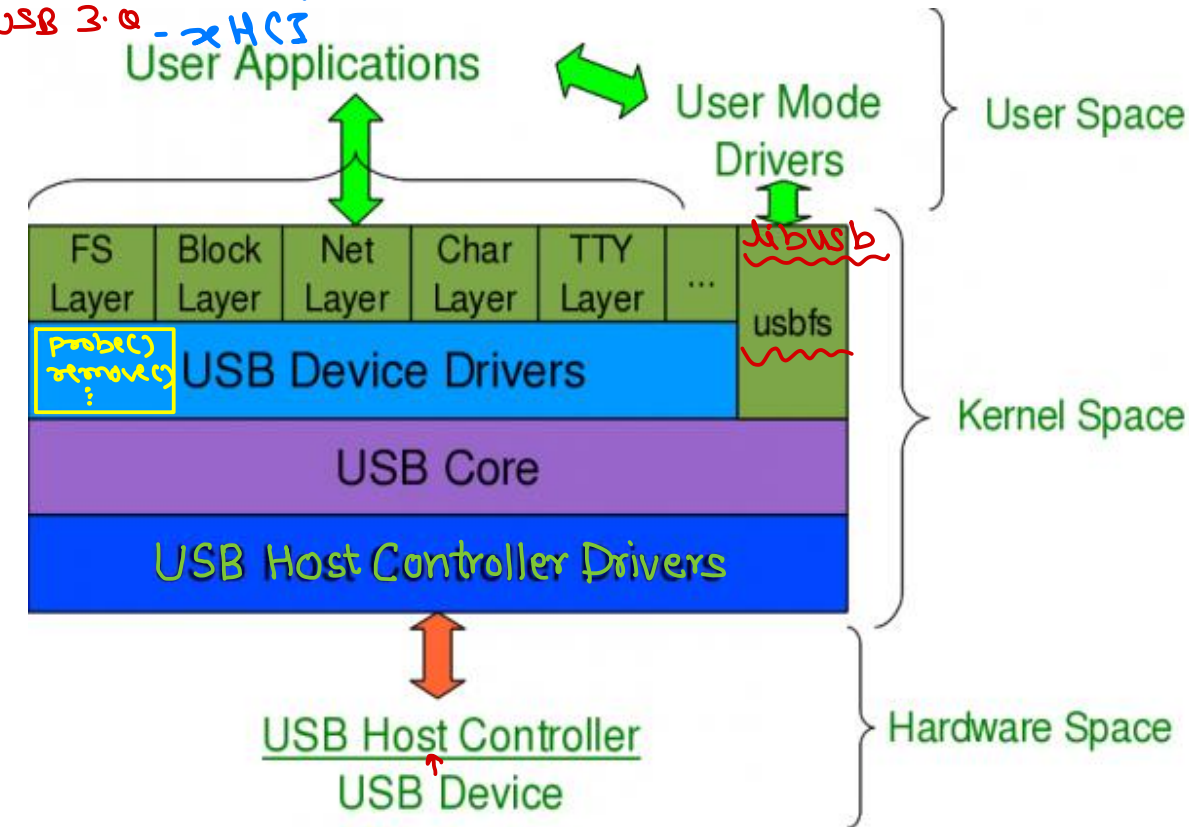
- **USB Host driver:**
 - The driver runs on host machine (in Linux system).
 - Responsible for giving commands to the device and retrieving data from device.
 - Majority of drivers fall in this type.
 - e.g. Pen drive driver, Keyboard driver, Mouse driver, ...
- **USB Client driver:**
 - The driver 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 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".

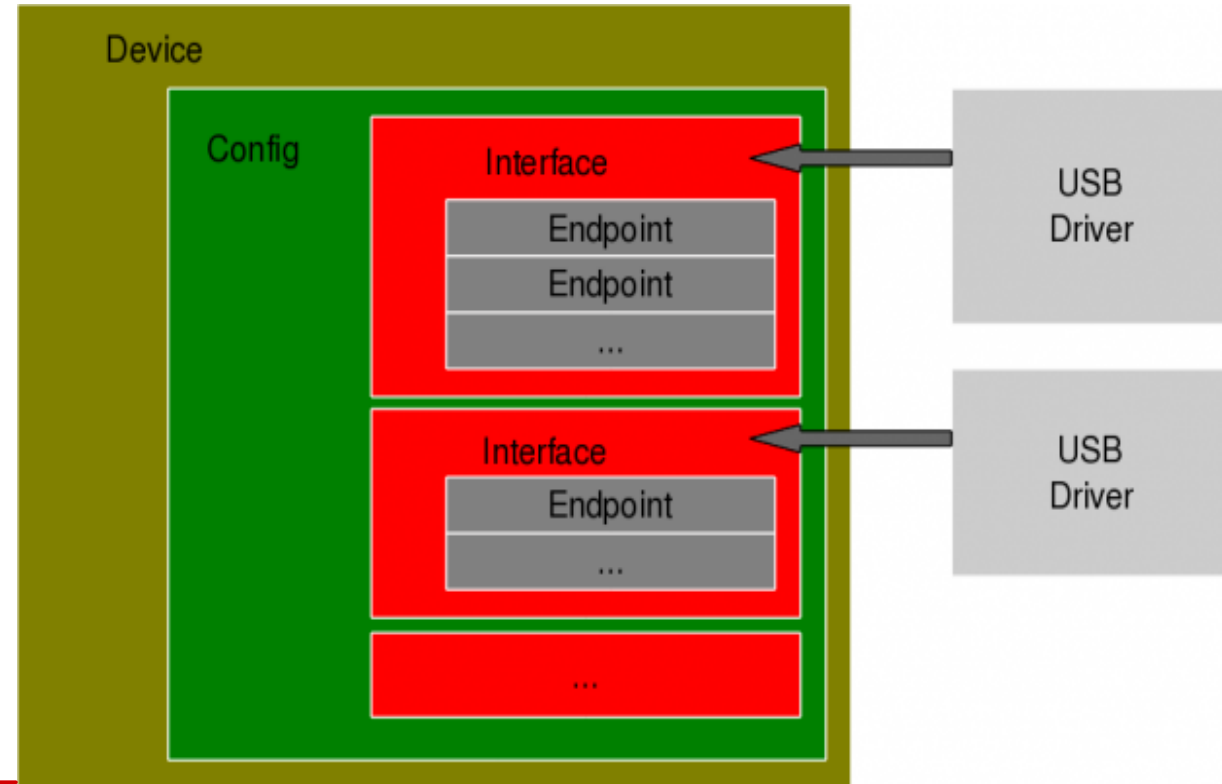
USB 1.0 - UHCI
USB 1.1 - OHCI
USB 2.0 - EHCI
USB 3.0 - xHCI



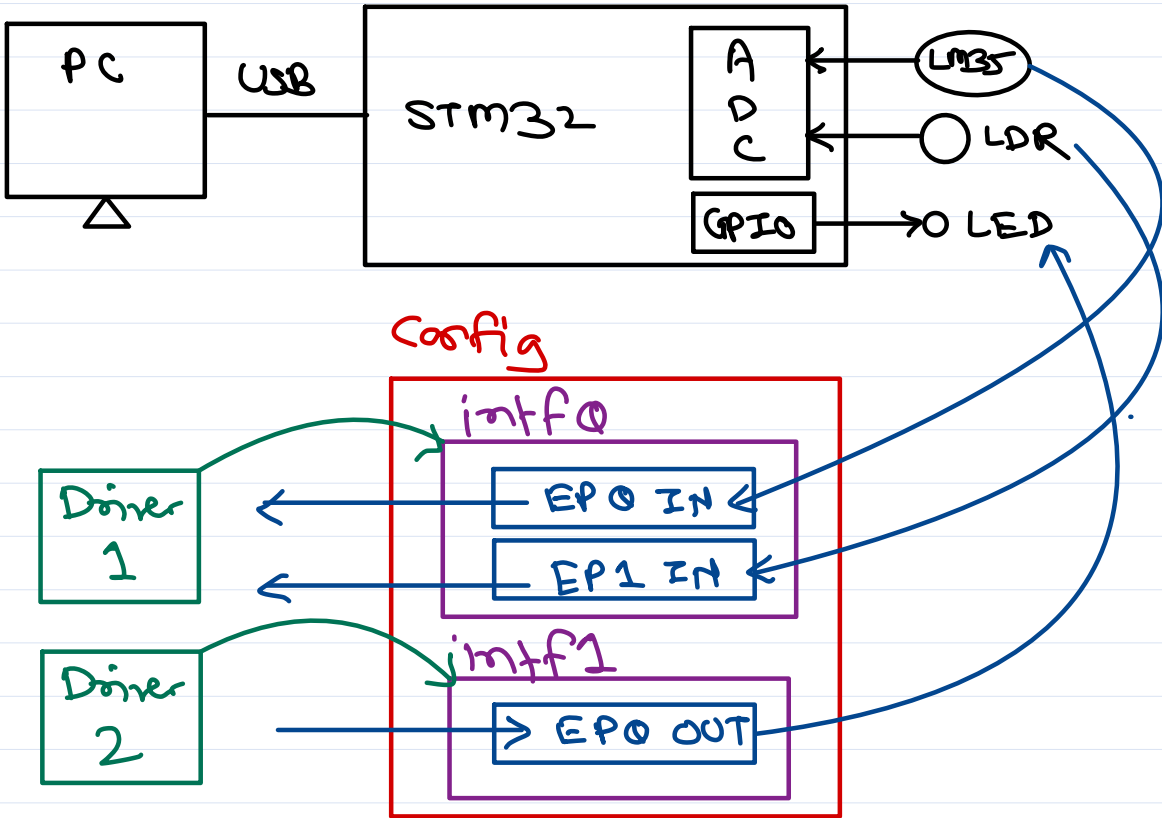
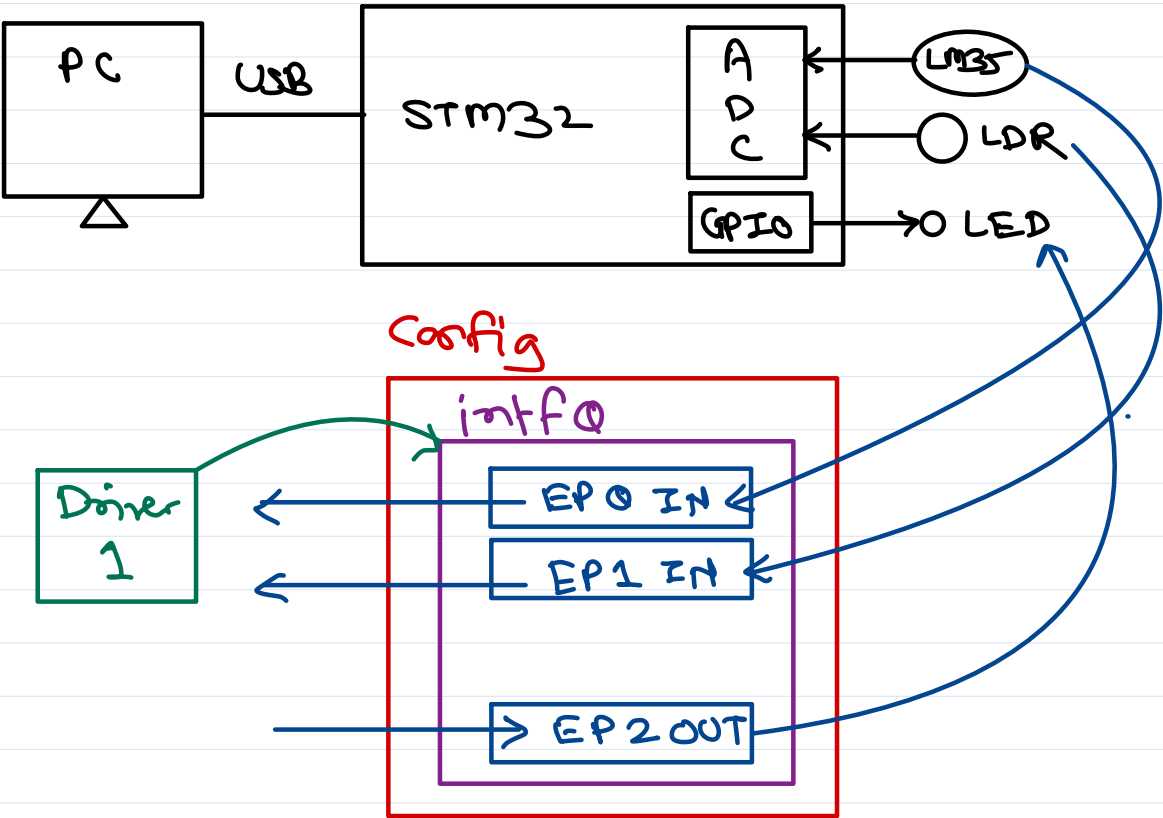
USB Device structure

- 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 → m



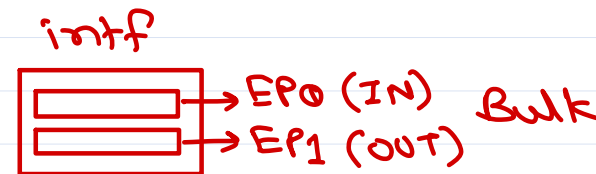
USB device config



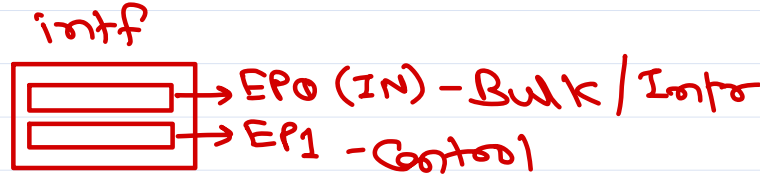
USB device class

USB Device Classes

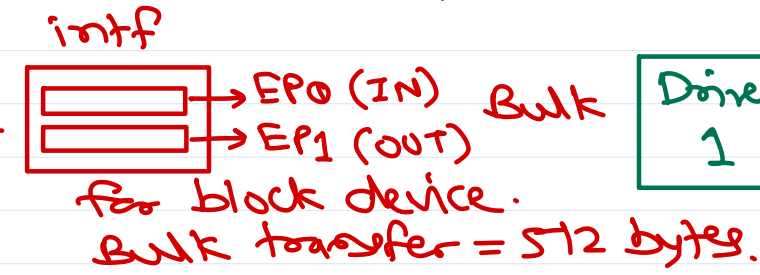
① CDC → like serial port



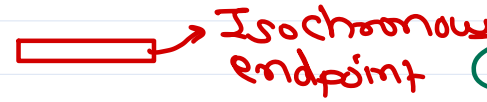
② HID → for kbd, mouse, joystick, etc.



③ MSD → for all storage devices

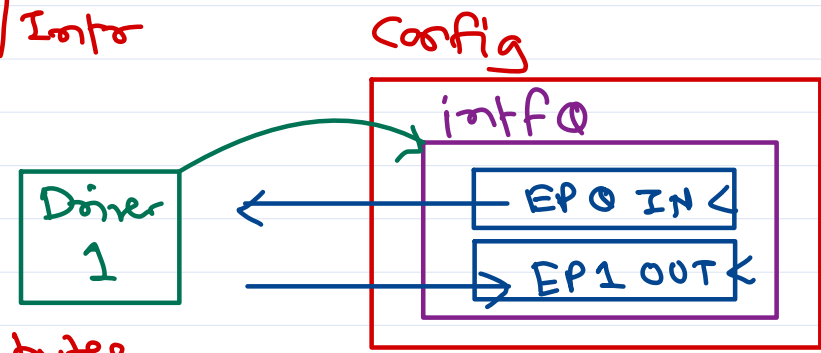
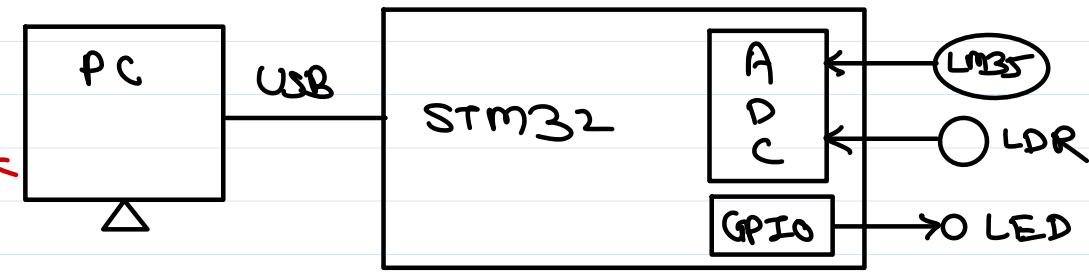


④ Audio → for audio devices like spkr, mic, ...



⑤ Custom

⑥ ...



CDC class

- ① Driver OUT → "LDR" → Device get LDR Reading
- ② Driver IN ← LDR reading
- ① Driver OUT → "LM35" → Device get 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 on 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.

- 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.

- Bulk

- Data transfer endpoint.
 - Can be IN or OUT. w.r.t. host.
 - Programmer need to allocate buffer for bulk endpoints.

- Isochronous

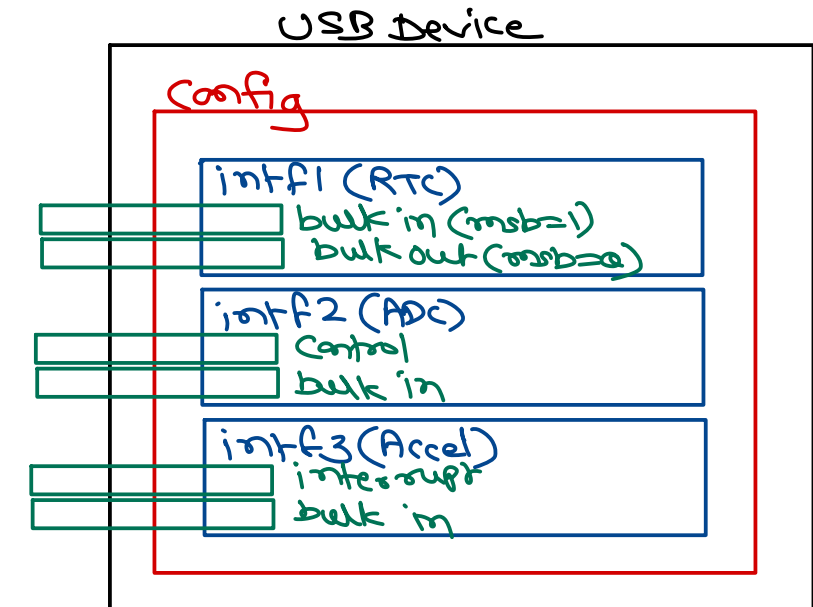
- Data transfer endpoint.
 - Ensures continuity of data transfer, but some data packets might be lost.
 - Mainly used for audio/video streaming. isochronous
 - Programmer need to allocate buffer for ~~bulk~~ endpoints.

- 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
 - 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)
 - 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*`

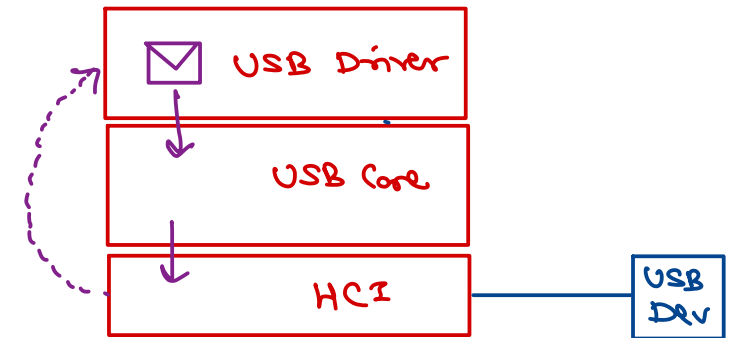
usb_device_id

- 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.
 - Assigned to a specific endpoint of a specific USB device.
 - Submitted to the USB core, by the USB device driver.
 - 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.



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



USB driver

→ vendor id, device id, class, subclass, ...

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_unregister(). 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 app 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.



USB device driver example





Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>

