

# **USB driver in Linux**

by

Danduri Datta Manikanta Srihari

ASU Id: 1217423272

Email Id: ddanduri@asu.edu

Final Report for CSE 530: Embedded Operating Systems Internals

ARIZONA STATE UNIVERSITY

May 2020

# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

## FLOW OF THE REPORT

The flow of the report is same as the flow in which I started learning about the USB drivers in Linux. **First, I started learning the concept behind the USB drivers in Linux by going through textbook “Linux Device Drivers, 3rd Edition by Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman”. Most of my investigation comes from here and the Linux USB API documentation.** In the end I wrote a kernel module that steals the data of all USB devices that are inserted. In order to go through Linux source code and see the implementation I used <https://elixir.bootlin.com/linux/v3.19.8/> all the source code snippets are taken from here if required I also provided hyperlinks. The Linux version I used for review is 3.19.8 in my entire report unless specified. **All the additional resources which I used for this report will be in resources section of this document.** Here is the flow of the report we will be looking into.

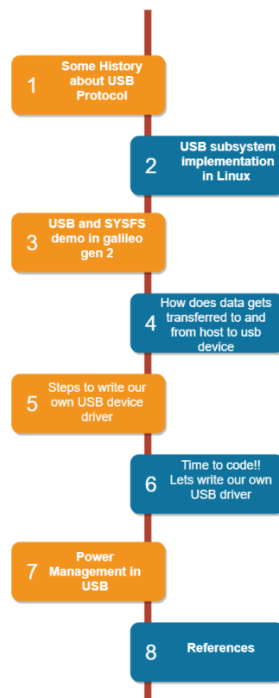


Fig 1: flow of my report

## What is USB and how does it work in Linux?

We all know USB we all use it in our day to day life. Before we start learning about it let's try to know a little bit of its history. USB is generally a connection between hosts and various peripheral devices. It was designed so that they can replace all slow and different bus types with just one type for which everyone can connect to. Four Industry Partners (Compaq, Intel, Microsoft, NEC) they started to specify the Universal Serial bus (USB). In Linux USB is implemented as a tree structure. You can imagine it as host being the master and all other hubs

# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

as interior nodes and the peripheral devices as slaves. This type of master/slave protocol is easy to use. Kernel developers added USB support to Linux in 2.2 kernel from then onwards they were supporting and developing it. Linux can be running on both USB device on peripheral side as well as in the host machine. Both perform different functionalities. USB drivers running inside the peripheral all called as “gadget drivers”. USB drivers running on host machine are called as “USB device drivers”.

## USB Hot plugging in Linux

Hot plugging allows to plugin new devices and use them immediately. Whenever a new device is plugged in to the system the system must perform some operations like

1. Find a driver which can handle this device load the modules required by the driver.
2. Bind the driver to the device and invoke the probe() function of the driver to perform required actions.
3. If there are any specific actions to be performed it also needed to be done.

## USB device drivers (Host side)

Host side USB device drivers communicate with “usbcore” Api’s. In host side itself we can see two types of drivers. One is general purpose drivers and the other are part of USB core itself like USB hub drivers, USB host controller drivers. The USB host controller asks every USB device whether it needs to send any data. The USB devices cannot send data on its own without being asked by the host controller. The USB protocol have a set of standards which any device can follow. For example, generic USB devices like keyboard mouse etc. does not require any special drivers whereas if you have a different device which does not fall under this category you need to write your own driver.

USB drivers live between kernel subsystem and the hardware controllers. USB core provides interface for the drivers to access the USB hardware. It provides abstraction so that drivers need not to worry about USB hardware controllers present on the system.

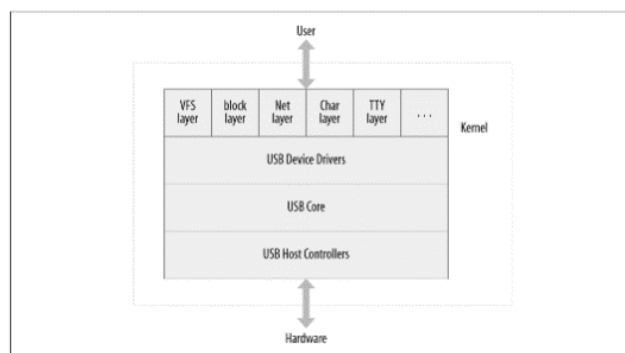


Fig 2. This figure shows overview of USB driver working<sup>[1]</sup>

# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

## USB DEVICE BASIC IDEA

USB core handles majority of complexity The USB driver communicates with the USB core for its requirements. Generally, USB devices have configurations, interfaces and endpoints lets go through them to understand it better.

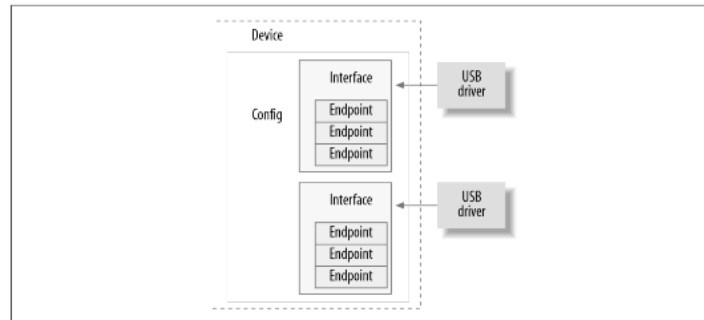


Fig 3. This figure shows overview of USB device.<sup>[1]</sup>

## ENDPOINTS

Think of endpoints as pipe for transferring information it is basically a communication link. Either from host computer to USB device which is called “out” or from USB device to host computer which is referred to as “in”. All seems to be fine we specified data transfer but what type of data what does it signify? Let’s try to answer this. So, in total there are 4 different types of USB endpoints

### 1.CONTROL

It is basically for transferring control information. We use this to configure the device, retrieving status of the device and various information about the device. All USB devices have the default endpoint as 0.

### 2.INTERRUPT

Used for small data transfer but at regular intervals of time. Most HID (Human Interface Devices) use it. For example, say you have a USB keyboard and mouse they send very small amount of data at fixed rate whenever USB host ask if they want to send the data.

### 3.BULK

It is suitable for bulk amount of transfer. Let’s say a printer you have to transfer a lot of data without any data loss similarly if you want to transfer data in to pen drive this endpoint is used.

# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

## 4.ISOCHRONOUS

This endpoint is also for bulk transfer, but it can tolerate loss of data. Here the transfers are very time sensitive. For example, audio and video devices use this endpoint

To summarize all endpoints are unidirectional either in or out except for the control endpoint. Moreover, Control and Bulk endpoints are asynchronous data transfers. But interrupt and isochronous are periodic. Since these occur periodically USB core reserve some bandwidth for them

### Let's Dive into source code (v3.19.8)

Everything is cool but where all these are stored how do we represent various endpoints? Is there any data structure inside kernel that holds it? To get answers to all these questions Let's investigate it.

There is a data structure in kernel that is `usb_host_endpoint`

```
/**
 * struct usb_host_endpoint - host-side endpoint descriptor and queue
 * @desc: descriptor for this endpoint, wMaxPacketSize in native byteorder
 * @ss_ep_comp: SuperSpeed companion descriptor for this endpoint
 * @urb_list: urbs queued to this endpoint; maintained by usbcore
 * @hcdpriv: for use by HCD; typically holds hardware dma queue head (QH)
 *           with one or more transfer descriptors (TDs) per urb
 * @ep_dev: ep_device for sysfs info
 * @extra: descriptors following this endpoint in the configuration
 * @extralen: how many bytes of "extra" are valid
 * @enabled: URBs may be submitted to this endpoint
 * @streams: number of USB-3 streams allocated on the endpoint
 *
 * USB requests are always queued to a given endpoint, identified by a
 * descriptor within an active interface in a given USB configuration.
 */
struct usb_host_endpoint {
    struct usb_endpoint_descriptor    desc;
    struct usb_ss_ep_comp_descriptor ss_ep_comp;
    struct list_head                  urb_list;
    void                              *hcdpriv;
    struct ep_device                  *ep_dev;      /* For sysfs info */

    unsigned char *extra; /* Extra descriptors */
    int extralen;
    int enabled;
    int streams;
};
```

Fig 4: `usb_host_endpoint` data structure<sup>[5]</sup>

If we investigate it there is a member called `usb_endpoint_descriptor` this contains all the USB specific data that device itself specified let's look into it

# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

```
struct usb_endpoint_descriptor
/* USB_DT_ENDPOINT: Endpoint descriptor */
struct usb_endpoint_descriptor {
    __u8 bLength;
    __u8 bDescriptorType;

    __u8 bEndpointAddress;
    __u8 bmAttributes;
    __le16 wMaxPacketSize;
    __u8 bInterval;

    /* NOTE: these two are _only_ in audio endpoints. */
    /* use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof. */
    __u8 bRefresh;
    __u8 bSynchAddress;
} __attribute__((packed));
```

Fig 5: usb\_endpoint\_descriptor data structure <sup>[5]</sup>

Let's go through important member elements of the data structure.

**bEndpointAddress:** This is the USB address of the specific endpoint we can use bit mask USB\_DIR\_OUT and USB\_DIR\_IN to determine whether data is directed towards host or towards device.

**bmAttributes:** By using appropriate bit masks on this we can know whether the endpoint is bulk or isochronous or interrupt endpoint.

**wMaxPacketSize:** This is the maximum size (bytes) this endpoint can handle at once. If anything above this it is divided in to chunks of wMaxPacketSize sizes.

**bInterval:** This is the time between requests for the endpoints. It is represented in milli seconds. This is only for interrupt endpoint

## Interfaces

Now let's explore about interfaces. If you see in the figure 2 lot of endpoints are bundled together to form an interface. One interface can handle only one logical connection let's say we have an USB speaker it has both audio and keyboard input it implies that it has two interfaces, so kernel requires two drivers to control it. An USB interface may have alternate setting as well. You can think of this alternate setting as different values for the endpoints.

Ok but where is this interface represented in kernel lets investigate the data structures provided by the kernel. It is in struct USB interface.

# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

## struct usb\_interface

```
struct usb_interface {
    /* array of alternate settings for this interface,
     * stored in no particular order */
    struct usb_host_interface *altsetting;

    struct usb_host_interface *cur_altsetting; /* the currently
     * active alternate setting */
    unsigned num_altsetting; /* number of alternate settings */

    /* If there is an interface association descriptor then it will list
     * the associated interfaces */
    struct usb_interface_assoc_descriptor *intf_assoc;

    int minor; /* minor number this interface is
     * bound to */

    enum usb_interface_condition condition; /* state of binding */
    unsigned sysfs_files_created:1; /* the sysfs attributes exist */
    unsigned ep_devs_created:1; /* endpoint "devices" exist */
    unsigned unregistering:1; /* unregistration is in progress */
    unsigned needs_remote_wakeup:1; /* driver requires remote wakeup */
    unsigned needs_altsetting0:1; /* switch to altsetting 0 is pending */
    unsigned needs_binding:1; /* needs delayed unbind/rebind */
    unsigned resetting_device:1; /* true: bandwidth alloc after reset */

    struct device dev; /* interface specific device info */
    struct device *usb_dev;
    atomic_t pm_usage_cnt; /* usage counter for autosuspend */
    struct work_struct reset_ws; /* for resets in atomic context */
};
```

Fig 6: usb\_interface data structure [5]

The USB core passes this structure to the USB driver and it's up to the driver what it needs to do with this.

struct usb\_host\_interface \*altsetting: It is a pointer pointing to array of interface structure containing all the alternate settings that may be selected for this interface. Let's look in to usb\_host\_interface to understand in detail

## struct usb\_host\_interface

```
/* host-side wrapper for one interface setting's parsed descriptors */
struct usb_host_interface {
    struct usb_interface_descriptor desc;

    int extralen;
    unsigned char *extra; /* Extra descriptors */

    /* array of desc.bNumEndpoint endpoints associated with this
     * interface setting. these will be in no particular order.
     */
    struct usb_host_endpoint *endpoint;

    char *string; /* iInterface string, if present */
};
```

Fig 7: usb\_host\_interface data structure [5]

If you see into it has pointer pointing to USB\_host\_endpoint which is already explained above.

It also has usb\_interface\_descriptor this descriptor holds all the information about the interface

# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

```
struct usb_interface_descriptor

/* USB_DT_INTERFACE: Interface descriptor */
struct usb_interface_descriptor {
    __u8  bLength;
    __u8  bDescriptorType;

    __u8  bInterfaceNumber;
    __u8  bAlternateSetting;
    __u8  bNumEndpoints;
    __u8  bInterfaceClass;
    __u8  bInterfaceSubClass;
    __u8  bInterfaceProtocol;
    __u8  iInterface;
} __attribute__((packed));
```

Fig 8: usb\_interface\_descriptor data structure <sup>[5]</sup>

unsigned num\_altsetting: It is basically number of alternate settings pointed by the altsetting pointer.

struct usb\_host\_interface \*cur\_altsetting: we should know what the current active interface setting is. it is given by this cur\_altsetting pointer.

int minor: This is given by usb core to interface if USB driver is bound to this interface using major number.

## Configurations

If you observe the above figure 2 we can see that all interfaces are bundled to form as configurations. A USB device can have multiple configurations. The example given in the textbook [Linux Device Drivers, 3rd Edition by Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman] is if any device allows firmware to be downloaded to it must have multiple configurations to achieve this.

Okay but what is the data structure that handles all this?

struct usb\_host\_config contains all these.

Let's look in to bootlin for further investigation.



# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

```
struct usb_host_config
{
    struct usb_config_descriptor desc;

    char *string;          /* iConfiguration string, if present */

    /* List of any Interface Association Descriptors in this
     * configuration. */
    struct usb_interface_assoc_descriptor *intf_assoc[USB_MAXIADS];

    /* the interfaces associated with this configuration,
     * stored in no particular order */
    struct usb_interface *interface[USB_MAXINTERFACES];

    /* Interface information available even when this is not the
     * active configuration */
    struct usb_interface_cache *intf_cache[USB_MAXINTERFACES];

    unsigned char *extra;  /* Extra descriptors */
    int extralen;
};
```

Fig 9: usb\_host\_config data structure [5]

If we see the member elements it has desc is of type usb\_config\_descriptor, and an array of pointers pointing to maximum number of interfaces. usb\_config\_descriptor holds all the data of the configuration.

```
struct usb_config_descriptor
{
    struct usb_config_descriptor {
        __u8 bLength;
        __u8 bDescriptorType;

        __le16 wTotalLength;
        __u8 bNumInterfaces;
        __u8 bConfigurationValue;
        __u8 iConfiguration;
        __u8 bmAttributes;
        __u8 bMaxPower;
    } __attribute__((packed));
};
```

Fig 10: usb\_config\_descriptor data structure [5]

If you look in to figure 2 the entire configuration is enclosed inside a device. But how does kernel represent a USB device?

The answer is straightforward It is done in data structure struct usb\_device

link to usb\_device data structure [5]

If you investigate it has various member elements completely describing the USB device. It is the responsibility of the USB device driver to convert the data from USB interface structure to USB device structure. USB core takes this USB device structure to perform various operations.

There is a method called interface\_to\_usbdev which driver can use to do this.

# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

## interface to usbdev

function provided in /include/linux/usb.h

```
static inline struct usb_device *interface_to_usbdev(struct usb_interface *intf)
{
    return to_usb_device(intf->dev.parent);
}
```

Fig 11: interface\_to\_usbdev data structure <sup>[5]</sup>

If you see it takes struct USB interface pointer and gives us a struct USB device pointer.

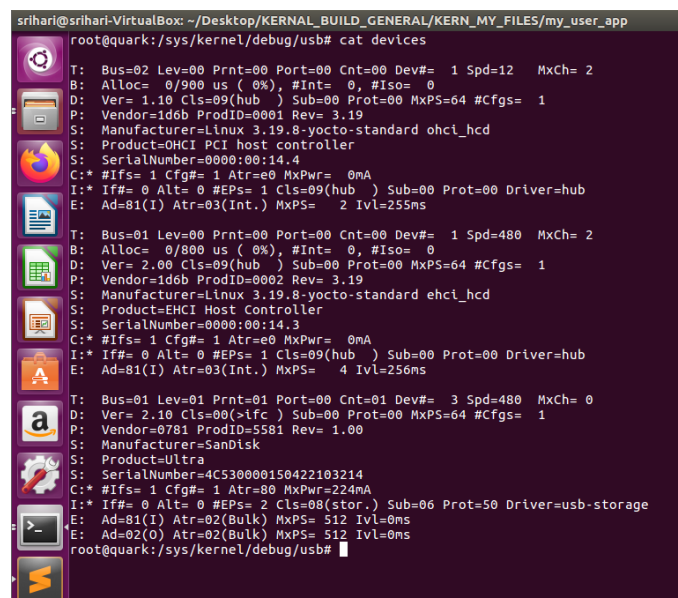
## LET'S SUMMARIZE TO GET OVERVIEW TILL NOW

A view from top to bottom in figure 2.

1. Devices have one or more configurations
2. Configurations have one or more interfaces
3. Interfaces have one or more settings
4. Interfaces have zero or more endpoints.

## USB AND SYSFS

USB devices are exported via debugfs. I connected my SanDisk pen drive to the Galileo board. Looking in to the /sys/kernel/debug/usb/ we can see all details of devices connected as USB devices.



```
srihari@srihari-VirtualBox: ~/Desktop/KERNAL_BUILD_GENERAL/KERN_MY_FILES/my_user_app
root@quark:/sys/kernel/debug/usb# cat devices
T: Bus=02 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#= 1 Spd=12 MxCh= 2
B: Alloc= 0/900 us ( 0%), #Int= 0, #Iso= 0
D: Ver= 1.10 Cls=09(hub ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1
P: Vendor=1d6b ProdID=0001 Rev= 3.19
S: Manufacturer=Linux 3.19.8-yocto-standard ohci_hcd
S: Product=OHCI PCI host controller
S: SerialNumber=0000:00:14.4
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr= 0mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=hub
E: Ad=81(I) Atr=03(Int.) MxPS= 2 IvL=255ns

T: Bus=01 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#= 1 Spd=480 MxCh= 2
B: Alloc= 0/800 us ( 0%), #Int= 0, #Iso= 0
D: Ver= 2.00 Cls=09(hub ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1
P: Vendor=1d6b ProdID=0002 Rev= 3.19
S: Manufacturer=Linux 3.19.8-yocto-standard ehci_hcd
S: Product=EHCI Host Controller
S: SerialNumber=0000:00:14.3
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr= 0mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=hub
E: Ad=81(I) Atr=03(Int.) MxPS= 4 IvL=256ns

T: Bus=01 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#= 3 Spd=480 MxCh= 0
D: Ver= 2.10 Cls=08(stor.) Sub=06 Prot=00 MxPS=64 #Cfgs= 1
P: Vendor=0781 ProdID=5581 Rev= 1.00
S: Manufacturer=SanDisk
S: Product=Ultra
S: SerialNumber=4C530000150422103214
C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=224mA
I:* If#= 0 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50 Driver=usb-storage
E: Ad=81(I) Atr=02(Bulk) MxPS= 512 IvL=0ms
E: Ad=02(O) Atr=02(Bulk) MxPS= 512 IvL=0ms
root@quark:/sys/kernel/debug/usb#
```

Figure 12. List of USB devices connected to Galileo board.

**Name: Danduri datta Manikanta Srihari**  
**ASU Id: 1217423272**

Here each line is tagged with a one letter

- T = Topology (etc.)
- B = Bandwidth (applies only to USB host controllers, which are virtualized as root hubs)
- D = Device descriptor info.
- P = Product ID info. (from Device descriptor, but they won't fit together on one line)
- S = String descriptors.
- C = Configuration descriptor info. (\* = active configuration)
- I = Interface descriptor info.
- E = Endpoint descriptor info.

Figure 13. Meaning of each character tagged to a line <sup>[2]</sup>

## `/sys/kernel/debug/usb/devices` output format

Legend::

d = decimal number (may have leading spaces or 0's) x = hexadecimal number (may have leading spaces or 0's) s = string

## Topology information

```
T: Bus=dd Lev=dd Prnt=dd Port=dd Cnt=dd Dev#=ddd Spd=dddd MxCh=dd  
| | | | | | | | | |  
| | | | | | | | | |__MaxChildren  
| | | | | | | | | |__Device Speed in Mbps  
| | | | | | | | | |__DeviceNumber  
| | | | | | | | | |__Count of devices at this level  
| | | | | | | | | |__Connector/Port on Parent for this device  
| | | | | | | | | |__Parent DeviceNumber  
| | | | | | | | | |__Level in topology for this bus  
| | | | | | | | | |__Bus number  
| | | | | | | | | |__Topology info tag
```

Figure 14. Topology line<sup>[2]</sup>

### Bandwidth information

```

B: Alloc=ddd/ddd us (xx%), #Int=ddd, #Iso=ddd
| | | | |
| | | | |__Number of isochronous requests
| | | | |__Number of interrupt requests
| | | | |__Total Bandwidth allocated to this bus
| | | | |__Bandwidth info tag

```

## Device descriptor information and product id information

```
D: Ver=x.xx Cls=xx(s) Sub=xx Prot=xx MxPS=dd #Cfgs=dd
P: Vendor=xxxx ProdID=xxxx Rev=xx.xx
```

**Name: Danduri datta Manikanta Srihari**  
**ASU Id: 1217423272**

```
D: Ver=xx CIs=xx(sssss) Sub=xx Prot=xx MxPS=dd #Cfgs=dd
| | | | | | |__NumberConfigurations
| | | | | | |__MaxPacketSize of Default Endpoint
| | | | | | |__DeviceProtocol
| | | | | | |__DeviceSubClass
| | | | | | |__DeviceClass
| | | | | | |__Device USB version
| | | | | | |__Device info tag #1
```

```
P: Vendor=xxxx ProdID=xxxx Rev=xx.xx
| | | | | | |__Product revision number
| | | | | | |__Product ID code
| | | | | | |__Vendor ID code
| | | | | | |__Device info tag #2
```

Figure 15. Bandwidth line<sup>[2]</sup>

In our Galileo board the inserted pen drive have product id = 5581 , Vendor id = 0781 we use this at the time we write our own USB device driver. We provide a match table where we mention these details. Whenever this USB device is plugged in to the system it gets matched and we can use our driver to handle the USB device

## String Descriptor information

```
S: Manufacturer=ssss
|  |__Manufacturer of this device as read from the device.
|  |   For USB host controller drivers (virtual root hubs) this may
|  |   be omitted, or (for newer drivers) will identify the kernel
|  |   version and the driver which provides this hub emulation.
|__String info tag

S: Product=ssss
|  |__Product description of this device as read from the device.
|  |   For older USB host controller drivers (virtual root hubs) this
|  |   indicates the driver; for newer ones, it's a product (and vendor)
|  |   description that often comes from the kernel's PCI ID database.
|__String info tag

S: SerialNumber=sssss
|  |__Serial Number of this device as read from the device.
|  |   For USB host controller drivers (virtual root hubs) this is
|  |   some unique ID, normally a bus ID (address or slot name) that
|  |   can't be shared with any other device.
|__String info tag
```

Figure 16. String line<sup>[2]</sup>

### Configuration descriptor information

```
C:* #Ifs=dd Cfg#dd Attr=xx MPwr=dddmA  
| | | | |  
| | | | |__MaxPower in mA  
| | | | |__Attributes  
| | | | |__ConfigurationNumber  
| | | | |__NumberOfInterfaces  
| | | | |__ "*" indicates the active configuration (others are ")  
| | | | |__Config info tag
```

Figure 17. Configuration line<sup>[2]</sup>

### Interface descriptor information

**Name: Danduri datta Manikanta Srihari**  
**ASU Id: 1217423272**

```

1:* If#=dd Alt=dd #EPS=dd Cls=xx(sssss) Sub=xx Prot=xx Driver=ssss
| | |      |      |      |      |      |      |      |      |
| | |      |      |      |      |      |      |      |__Driver name
| | |      |      |      |      |      |      |      |or "(none)"
| | |      |      |      |      |      |      |      |__InterfaceProtocol
| | |      |      |      |      |      |      |      |__InterfaceSubClass
| | |      |      |      |      |      |      |      |__InterfaceClass
| | |      |      |      |      |      |      |      |__NumberOfEndpoints
| | |      |      |      |      |      |      |      |__AlternateSettingNumber
| | |      |      |      |      |      |      |      |__InterfaceNumber
|_|_"_" indicates the active altsetting (others are " ")
|_Interface info tag
```

Figure 18. Interface line <sup>[2]</sup>

## Endpoint descriptor information

```
E: Ad=xx(s) Atr=xx(ssss) MxPS=dddd Ivl=dddss  
| | | |  
| | | |__Interval (max) between transfers  
| | | |__EndpointMaxPacketSize  
| | | |__Attributes(EndpointType)  
| | | |__EndpointAddress(I=In,0=Out)  
| | | |__Endpoint info tag
```

Figure 19. Endpoint line<sup>[2]</sup>

## HOW DOES DATA TRANSFER OCCUR?

## USB Urbs

Urb stands for USB request block. Ok but where is it present?

It is present in

Link to bootlin urb data structure<sup>[5]</sup> `struct urb`

Okay cool but for what is it used for?

It is used to send or receive the data to or from a specific USB endpoint on a USB device. We know the working of skbuff from sockets it is similar to that.

A USB driver can create a single urb for a single endpoint or it may use the same urb for multiple endpoints. Every endpoint can handle multiple urbs so they can be placed in a queue.

Let's see the life cycle of a urb

1. A usb driver creates it
2. It is assigned to a specific endpoint of a specific usb device
3. Usb driver submits it to usb core
4. Usb core sees it and gives it to specific usb host controller driver for the specified device

# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

5. The usb host controller driver does the transfer of data

6. When it is done it notifies to usb device driver

An urb created can be destroyed anytime either by usb driver or by the usb core itself if the driver is removed

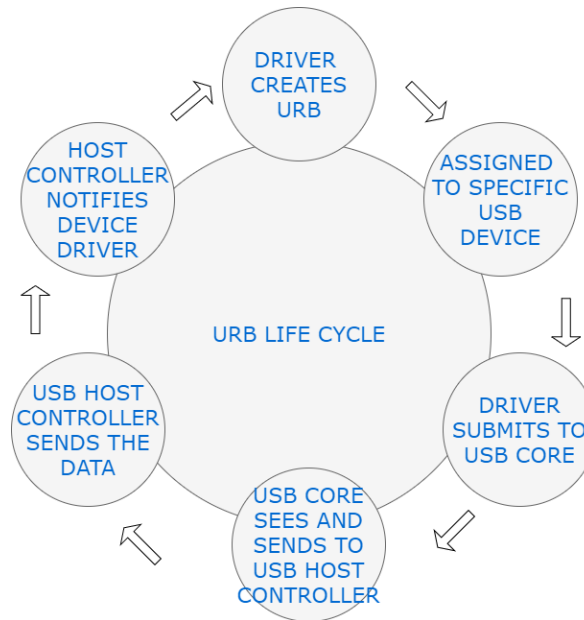


Figure 20. URB life cycle

struct urb

```
struct urb {
    /* private: usb core and host controller only fields in the urb */
    struct kref kref; /* reference count of the URB */
    void *hpriv; /* private data for host controller */
    atomic_t use_count; /* concurrent submissions counter */
    atomic_t reject; /* submissions will fail */
    int unlinked; /* unlink error code */

    /* public: documented fields in the urb that can be used by drivers */
    struct list_head urb_list; /* List head for use by the urb's
                                * current owner */
    struct list_head anchor_list; /* the URB may be anchored */
    struct usb_anchor *anchor; /* (in) pointer to associated device */
    struct usb_device *dev; /* (internal) pointer to endpoint */
    struct usb_host_endpoint *ep; /* (in) pipe information */
    unsigned int pipe; /* (in) stream ID */
    unsigned int stream_id; /* (return) non-ISO status */
    int status; /* (in) URB_SHORT_NOT_OK | ... */
    unsigned int transfer_flags; /* (in) associated data buffer */
    void *transfer_buffer; /* (in) dma addr for transfer_buffer */
    dma_addr_t transfer_dma; /* (in) scatter gather buffer list */
    struct scatterlist *sg; /* (internal) mapped sg entries */
    int num_mapped_sgs; /* (in) number of entries in the sg list */
    int num_sgs; /* (in) data buffer length */
    u32 transfer_buffer_length; /* (return) actual transfer length */
    u32 actual_length; /* (in) setup packet (control only) */
    unsigned char *setup_packet; /* (in) dma addr for setup_packet */
    dma_addr_t setup_dma; /* (modify) start frame (ISO) */
    int start_frame; /* (in) number of ISO packets */
    int number_of_packets; /* (modify) transfer interval
                                * (INT/ISO) */
    int interval; /* (return) number of ISO errors */
    int error_count; /* (in) context for completion */
    void *context; /* (in) completion routine */
    usb_complete_t complete; /* (in) ISO ONLY */
    struct usb_iso_packet_descriptor iso_desc[0];
};
```

Fig 21: urb data structure<sup>[5]</sup>

## REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

Usb device driver cares about specific fields of struct urb structure they are

struct usb\_device \*dev: This is the usb device to which we are going to send the urb. Usb device driver should initialize it before sending it to usb core.

unsigned int pipe: Holds endpoint information like type,direction etc

The driver must fill the structure before it sends the urb to the device to fill the structure there are a lot of functions defined as macros like usb\_sndctrlpipe, usb\_rcvctrlpipe and so on.

unsigned int transfer\_flags: Depending up on what usb driver wants to happen for the urb these flags can be set.

void \*transfer\_buffer: It is the pointer pointing to the buffer that is used to sending of data or receiving of data

int transfer\_buffer\_length: It is length of buffer pointed by the transfer\_buffer.

unsigned char \*setup\_packet: it is a pointer to setup a packet for a control urb.

usb\_complete\_t complete: when urb is completely transferred completion handler function is called by the usb core and it is defined in kernel as

```
typedef void (*usb_complete_t)(struct urb *);
```

Fig 22: usb\_complete\_t data structure<sup>[5]</sup>

void \*context: it is set initially by the usb driver when urb transfer is done and when completion handler is called it can be used there.

int status: it shows the status of the urb 0 indicating successful transfer. Refer to the documentation for various error codes if error occurs.

### Creating and Destroying Urbs

Since we know what is urb and in which data structure it is present let's see how we create and destroy the urbs. We shouldn't statically create urb in a driver or any other structure. Since reference counting scheme used by the usb core gets disturbed. We should create urb by calling usb\_alloc\_urb function.

# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

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

    urb = kmalloc(sizeof(struct urb) +
                  iso_packets * sizeof(struct usb_iso_packet_descriptor),
                  mem_flags);
    if (!urb) {
        printk(KERN_ERR "alloc_urb: kmalloc failed\n");
        return NULL;
    }
    usb_init_urb(urb);
    return urb;
}
```

Fig 23: usb\_alloc\_urb data structure <sup>[5]</sup>

If you see the arguments to the function we have isopackets and mem\_flags if you don't want to create a isochronous packet you can set the value of iso\_packet to 0.

Now we have the urb. before using it we need to initialize it properly we can free the urb by calling

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

Lets see how to initialize different types of urbs

There are 4 different kind of these

1. Interrupt urbs
2. Bulk urbs
3. Control urbs
4. Isochronous urbs

## 1. Interrupt urbs

If you want to send urb to interrupt endpoint of usb device, you must initialize by calling usb\_fill\_int\_urb helper function

Let's look in to the source code of this function



# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

`static inline void usb_fill_int_urb`

```
static inline void usb_fill_int_urb(struct urb *urb,
                                   struct usb_device *dev,
                                   unsigned int pipe,
                                   void *transfer_buffer,
                                   int buffer_length,
                                   usb_complete_t complete_fn,
                                   void *context,
                                   int interval)
{
    urb->dev = dev;
    urb->pipe = pipe;
    urb->transfer_buffer = transfer_buffer;
    urb->transfer_buffer_length = buffer_length;
    urb->complete = complete_fn;
    urb->context = context;

    if (dev->speed == USB_SPEED_HIGH || dev->speed == USB_SPEED_SUPER) {
        /* make sure interval is within allowed range */
        interval = clamp(interval, 1, 16);
        urb->interval = 1 << (interval - 1);
    } else {
        urb->interval = interval;
    }

    urb->start_frame = -1;
}
```

Fig 24: `usb_fill_int_urb` data structure<sup>[5]</sup>

This contains a lot of parameters lets go through them

`struct urb *urb`: the urb to be initialized

`struct usb_device *dev`: the usb device this urb needs to be sent

`unsigned int pipe`: all specific endpoint information of a device is present in this parameter

`void *transfer_buffer`: pointer to a buffer where data needs to come in or go out to the device.

`int buffer_length`: buffer length pointed by `transfer_buffer`

`usb_complete_t complete`: completion handler pointer. This handler is executed when urb transfer is done

`void *context`: data pointer which can be used in completion handler

`int interval`: the interval at which this urb should be scheduled

## Bulk urbs

It is initialized same as interrupt urbs. We use `usb_fill_bulk_urb`

# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

```
static inline 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_fn,
    void *context)
{
    urb->dev = dev;
    urb->pipe = pipe;
    urb->transfer_buffer = transfer_buffer;
    urb->transfer_buffer_length = buffer_length;
    urb->complete = complete_fn;
    urb->context = context;
}
```

Fig 25: urbs data structure <sup>[5]</sup>

If you see the function parameters there is no interval parameter since bulk urbs will not have interval value

## Control urbs

Control urbs are almost initialized same as bulk urbs we can use `usb_fill_control_urb` function to initialize our control urb

```
static inline void usb_fill_control_urb(
    struct urb *urb,
    struct usb_device *dev,
    unsigned int pipe,
    unsigned char *setup_packet,
    void *transfer_buffer,
    int buffer_length,
    usb_complete_t complete_fn,
    void *context)
{
    urb->dev = dev;
    urb->pipe = pipe;
    urb->setup_packet = setup_packet;
    urb->transfer_buffer = transfer_buffer;
    urb->transfer_buffer_length = buffer_length;
    urb->complete = complete_fn;
    urb->context = context;
}
```

Fig 26: urb\_fill\_control\_urb data structure. <sup>[5]</sup>

If you look into parameters, you can see new parameter called unsigned char \*setup\_packet

We must setup the packet data that is to be sent to the end point.

# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

## Isochronous urbs

For Isochronous urbs there is no function that can be used to initialize the urb we need to do it manually

## Submitting Urbs

Since we are done with initialization of our urb we can now submit the urb to the core the core will handle rest of the task ie sending it to usb host controller from there to device.

We can submit the urb by calling `usb_submit_urb`

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

link to `usb_submit_urb` data structure<sup>[5]</sup>

If you see the parameters, we need to pass our urb and `mem_flags` it does the rest of task

## Completing Urbs: The Completion Callback Handler

After successful submission and transfer of urb the completion handler is executed after that usb core transfer backs the control to the usb device driver

## Canceling Urbs

We can also cancel submitted urb by calling `int usb_kill_urb(struct urb *urb);` and `int usb_unlink_urb(struct urb *urb);`

## USB ANCHORS

Anchor is a data structure which keeps track of all the URB's. We can create anchor by simply declaring `struct usb_anchor`. Before we use it it needs to be initialized by calling `init_usb_anchor()`.

In order to attach urb to the anchor we can use the method `usb_anchor_urb()`. To detach the urb from the anchor we can use the method `usb_unanchor_urb()`.

## Writing a USB Driver

Now by acquiring all the knowledge about usb we are ready to start writing our own usb device driver.

The process is very straightforward the driver registers itself to the usb subsystem and we use vendor id and device id to identify and see if hardware has been installed.

Ok but which devices does this driver can access?

# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

There is this structure called `usb_device_id` it maintains all the list of usb devices this driver can support

Let's look in to this structure in detail.

`struct usb_device_id`

```
struct usb_device_id {
    /* which fields to match against? */
    __u16      match_flags;

    /* Used for product specific matches; range is inclusive */
    __u16      idVendor;
    __u16      idProduct;
    __u16      bcdDevice_lo;
    __u16      bcdDevice_hi;

    /* Used for device class matches */
    __u8      bDeviceClass;
    __u8      bDeviceSubClass;
    __u8      bDeviceProtocol;

    /* Used for interface class matches */
    __u8      bInterfaceClass;
    __u8      bInterfaceSubClass;
    __u8      bInterfaceProtocol;

    /* Used for vendor-specific interface matches */
    __u8      bInterfaceNumber;

    /* not matched against */
    kernel_ulong_t driver_info
        __attribute__((aligned(sizeof(kernel_ulong_t))));
};
```

Fig 27: `usb_device_id` data structure.<sup>[5]</sup>

Let's go through some of member elements important to us the vendor id and the productid are compared when a device is plugged in. For example below `skel_table` is registered to this usb driver.

It can support device with vendor id: `USB_SKEL_VENDOR_ID` and product id: `USB_SKEL_PRODUCT_ID`

`/* table of devices that work with this driver */`

```
static struct usb_device_id skel_table [] = { { USB_DEVICE(USB_SKEL_VENDOR_ID,
USB_SKEL_PRODUCT_ID) },
```

```
{ }          /* Terminating entry */
```

```
};
```

# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

//adding of the table

MODULE\_DEVICE\_TABLE (usb, skel\_table);

## Registering a USB Driver

This is the main structure that every usb device driver needs to use if they want register themselves as usb device driver.

Lets look in to source code of it

`struct usb_driver`

```
struct usb_driver {
    const char *name;

    int (*probe) (struct usb_interface *intf,
                  const struct usb_device_id *id);

    void (*disconnect) (struct usb_interface *intf);

    int (*unlocked_ioctl) (struct usb_interface *intf, unsigned int code,
                           void *buf);

    int (*suspend) (struct usb_interface *intf, pm_message_t message);
    int (*resume) (struct usb_interface *intf);
    int (*reset_resume)(struct usb_interface *intf);

    int (*pre_reset)(struct usb_interface *intf);
    int (*post_reset)(struct usb_interface *intf);

    const struct usb_device_id *id_table;

    struct usb_dynids dynids;
    struct usbdrv_wrap drvwrap;
    unsigned int no_dynamic_id:1;
    unsigned int supports_autosuspend:1;
    unsigned int disable_hub_initiated_lpm:1;
    unsigned int soft_unbind:1;
};
```

Fig 28: usb\_driver data structure<sup>[5]</sup>

Let's explore one by one structure members

const char \*name: It is the name of the usb driver it should be unique among other device drivers names. It appears in /sys/bus/usb/drivers/ of sysfs

const struct usb\_device\_id \*id\_table: This specifies what usb devices this driver can handle. You must add all those devices in the table here.

int (\*probe) (struct usb\_interface \*intf, const struct usb\_device\_id \*id): whenever a usb device is recognized if usb core thinks this driver could handle that device it calls this probe function of the usb driver.

void (\*disconnect) (struct usb\_interface \*intf): whenever usb driver is removed or if usb core detects that the usb interface is removed from the filesystem this disconnect function of the

# REPORT ON USB DRIVERS IN LINUX

**Name:** Danduri datta Manikanta Srihari

**ASU Id:** 1217423272

driver is called. It breaks any connection with the interface. Once this method is done you will not be able to send any IO signals to the device. Any unsubmitted urbs you have will be killed.

int (\*ioctl) (struct usb\_interface \*intf, unsigned int code, void \*buf): if any userspace application calls ioctl call on usb filesystem to this driver then this function is executed

int (\*suspend) (struct usb\_interface \*intf, u32 state): if usb core decides to suspend the device then this function is called

int (\*resume) (struct usb\_interface \*intf): if usb core wants to resume the device then this function is called

post\_reset : reset is completed any saved state of the device will be restored and device is used again.

Suspend and Resume are very useful in power management which we will be looking into in this report.

## MY\_OWN\_USB\_DRIVER

In the zip file submitted along with this report you could find the example code. The instructions on how to run the code is given in the readme file. In the report I will explain the working of the code.

I have 2 versions of my code one is folder USB\_data\_collector and one is usb\_device\_info\_stealer. USB\_data\_collector is completely working model and usb\_device\_info\_stealer is not yet completed. I am struck at extracting the usb\_data pointer from the stack register values.

### USB\_data\_collector

Kernel module:

My code steals the stack values during any usb device is inserted. When ever any Usb device is inserted usb-storage drivers probe function is executed. Here I exploited the usb\_stor\_probe1 method which is exported. This method is called inside the main probe function. I am placing a kprobe inside the probe function of this driver. My module once inserted keeps collecting all the data and it will dynamically store the data in a list. For the user space application to access the stored data it can use sysfs to invoke and read data to user space.

User application:

User space application uses sysfs interface to get the data. The password sent by user space application will be verified by the module if it matches then only stack register values are printed.

# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

## usb\_device\_info\_stealer

Kernel Module:

This is an extension to USB\_data\_collector. In the USB\_data\_collector I am just stealing the stack register values in this I am trying to get the struct usb\_data \*us pointer. If I could get that pointer, I could store entire device info in my list. I am trying to do this by appropriately placing probe at usb\_stor\_disconnect when ever device is removed this method is called by the Usb core. Since the argument to usb\_stor\_disconnect is struct usb\_interface \*intf my assumption is that it pushes this value on to stack. From there we could extract the pointer.

My understanding might not be accurate. According to my best of knowledge I thought of this process. Since all the structures are not available in kernel linux headers I am taking the usb.h header file directly all we require is just equivalent data structure so that we could allocate same amount of memory so that we can copy the data using memcpy. I tried this approach I was able to steal the data pointed by struct usb\_host\_interface \*cur\_altsetting; pointer. When I try it for struct usb\_data \*us pointer my module is crashing.

```
int Pre_Handler(struct kprobe *kp, struct pt_regs *regs){
    struct all_usb_data *per_dev_node;
    char address[20];
    struct us_data *us;
    unsigned int adder;

    struct usb_interface **temp;
    printk(KERN_INFO "INSIDE PRE HANDLER\n");

    per_dev_node = (struct all_usb_data *)kmalloc(sizeof(struct all_usb_data), GFP_KERNEL);
    per_dev_node->regs = (struct pt_regs *)kmalloc(sizeof(struct pt_regs), GFP_KERNEL);
    per_dev_node->intf = (struct usb_interface *)kmalloc(sizeof(struct usb_interface), GFP_KERNEL);

    per_dev_node->device_no=device_counter++;

    sprintf(address,"%08lx",regs->si);
    printk(KERN_INFO"address is %08lx\n",regs->sp);
    //per_dev_node->intf=(struct usb_interface *)address;
    per_dev_node->intf=(struct usb_interface *)address;
    //memcpy(per_dev_node->intf,(struct usb_interface *)address,sizeof(struct usb_interface));
    //us = usb_get_intfdata(*temp);
    printk(KERN_INFO "bNumEndpoints is %d",per_dev_node->intf->cur_altsetting->desc.bNumEndpoints );

    // memset((void *)address, 0, sizeof(address));

    regs_cpy(regs,per_dev_node->regs);
    list_add(&per_dev_node->dlist, &all_usb_data_list);
    return 0;
}
```

Fig 29: problems in my second code

Inside the pre handler I am unable to figure out in which registers does that argument pointer will be. As far as my knowledge about x86 registers if the argument is call by value type then the register rbx contains it. In this case the argument is of a pointer which I am unable to figure out.

I will have my code uploaded in

[https://github.com/dandurisrihari/EOSI\\_FINAL\\_REPORT\\_USB\\_DATA\\_HACK](https://github.com/dandurisrihari/EOSI_FINAL_REPORT_USB_DATA_HACK) any future updates or contributions are highly appreciated

For both the codes I have readme file and make files. This code is written for Intel Galileo gen 2.

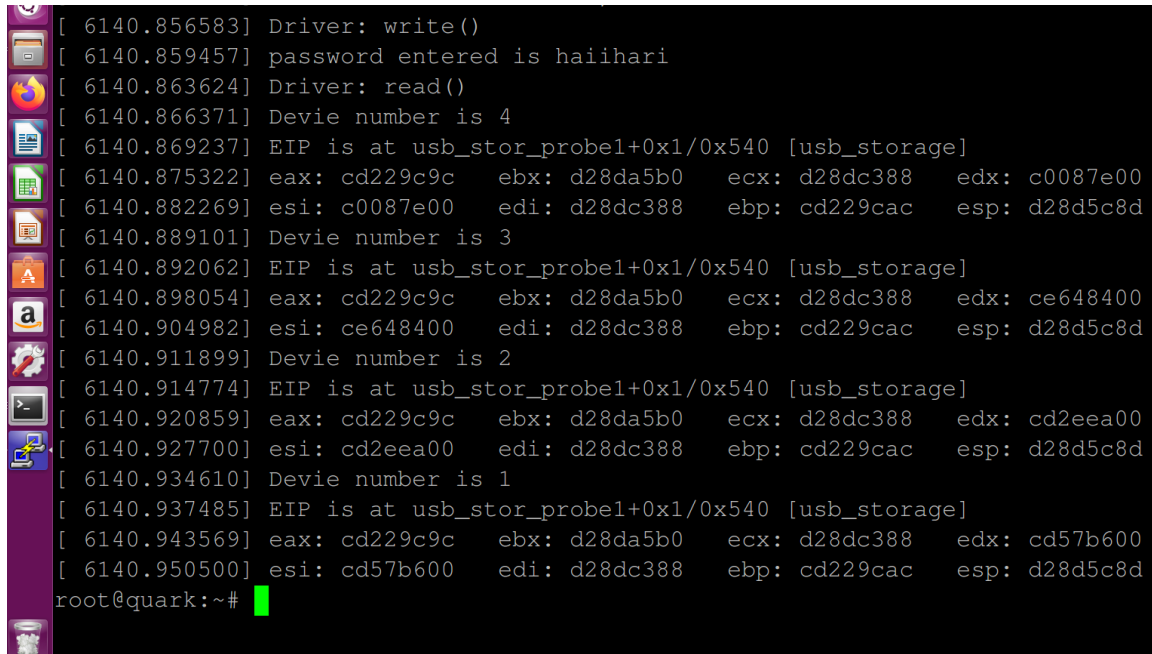
# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

## SAMPLE OUTPUT

The output is for usb\_data\_collector code which runs perfectly. I inserted my module in to kernel later I plugged and unplugged 4 usb pen drives in to it. My module records that and when my user application runs it prints the recorded log to the screen.



```
[ 6140.856583] Driver: write()
[ 6140.859457] password entered is haiihari
[ 6140.863624] Driver: read()
[ 6140.866371] Devie number is 4
[ 6140.869237] EIP is at usb_stor_probe1+0x1/0x540 [usb_storage]
[ 6140.875322] eax: cd229c9c ebx: d28da5b0 ecx: d28dc388 edx: c0087e00
[ 6140.882269] esi: c0087e00 edi: d28dc388 ebp: cd229cac esp: d28d5c8d
[ 6140.889101] Devie number is 3
[ 6140.892062] EIP is at usb_stor_probe1+0x1/0x540 [usb_storage]
[ 6140.898054] eax: cd229c9c ebx: d28da5b0 ecx: d28dc388 edx: ce648400
[ 6140.904982] esi: ce648400 edi: d28dc388 ebp: cd229cac esp: d28d5c8d
[ 6140.911899] Devie number is 2
[ 6140.914774] EIP is at usb_stor_probe1+0x1/0x540 [usb_storage]
[ 6140.920859] eax: cd229c9c ebx: d28da5b0 ecx: d28dc388 edx: cd2eea00
[ 6140.927700] esi: cd2eea00 edi: d28dc388 ebp: cd229cac esp: d28d5c8d
[ 6140.934610] Devie number is 1
[ 6140.937485] EIP is at usb_stor_probe1+0x1/0x540 [usb_storage]
[ 6140.943569] eax: cd229c9c ebx: d28da5b0 ecx: d28dc388 edx: cd57b600
[ 6140.950500] esi: cd57b600 edi: d28dc388 ebp: cd229cac esp: d28d5c8d
root@quark:~#
```

Fig 30: output of my first code

## POWER MANAGEMENT IN USB

Power management is all about saving energy by suspending different parts of a computer when they are not in use. When kernel suspends a device, the device goes into nonfunctional low power state or it might even be completely turned off. When kernel needs to use the device, it can resume the device i.e. (device can be operated at full power state).

### DYNAMIC POWER MANAGEMENT

Kernel can autosuspend the device if it is idle for a minimum amount of time. If any program tries to use the device then kernel will automatically resume the device. Generally, all devices that are autosuspended by the kernel will have remote wakeup enabled.

### USER INTERFACE FOR DYNAMIC POWER MANAGEMENT

User interface to control dynamic power management is present in /sys/bus/usb/devices/power directory. Let's take a peek in to Galileo board power management.



# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

```
srihari@srihari-VirtualBox: ~/Desktop/KERNAL_BUILD_GENERAL/KERN_MY_FILES/my_user_app
root@quark:/sys/bus/usb/devices/usb1/power# ls
active_duration      connected_duration   runtime_active_time  wakeup              wakeup_active_count  wakeup_last_time_ms
autosuspend          control              runtime_status        wakeup_abort_count   wakeup_count          wakeup_max_time_ms
autosuspend_delay_ms level                runtime_suspended_time wakeup_active         wakeup_expire_count   wakeup_total_time_ms
root@quark:/sys/bus/usb/devices/usb1/power# cat wakeup
disabled
root@quark:/sys/bus/usb/devices/usb1/power# cat control
auto
root@quark:/sys/bus/usb/devices/usb1/power# cat autosuspend_delay_ms
0
root@quark:/sys/bus/usb/devices/usb1/power#
```

Fig 31: Galileo usb power management

Figure 11. Dynamic power management UI in Galileo board.

Wakeup: By changing this setting we can enable or disable remote wakeup for the device.

Control: On means that device should be resumed and autosuspend not allowed. Auto implies that kernel can autoresume and autosuspend the device

I connected my SanDisk pen drive to the intel Galileo board. Following are the various parameters displayed.

```
Terminal File Edit View Search Terminal Help
root@quark:/sys/bus/usb/devices/1-1# ls
1-1:1.0      authorized      bDeviceClass      bMaxPower          bmAttributes       devnum             ep_00             manufacturer       product            serial            urbrnum
avoid_reset_quirk bConfigurationValue bDeviceProtocol    bNumConfigurations busnum             idProduct          maxchild           quirks             speed             version
bConfigurationValue bMaxPacketSize0    bDeviceSubclass    bNumInterfaces     configuration       idVendor           port              removable          subsystem         uevent
root@quark:/sys/bus/usb/devices/1-1# cat manufacturer
SanDisk
root@quark:/sys/bus/usb/devices/1-1# cd power
root@quark:/sys/bus/usb/devices/1-1/power# ls
active_duration      autosuspend_delay_ms control            persist            runtime_active_time runtime_status      runtime_suspended_time
root@quark:/sys/bus/usb/devices/1-1/power# [ 347.027042] random: nonblocking pool is initialized
root@quark:/sys/bus/usb/devices/1-1/power# cat autosuspend_delay_ms
2000
root@quark:/sys/bus/usb/devices/1-1/power# cat control
on
root@quark:/sys/bus/usb/devices/1-1/power# cat con
connected_duration control
root@quark:/sys/bus/usb/devices/1-1/power# cat connected_duration
166440
root@quark:/sys/bus/usb/devices/1-1/power# cat autosuspend
2
root@quark:/sys/bus/usb/devices/1-1/power# cat active_duration
195530
root@quark:/sys/bus/usb/devices/1-1/power# cat level
[ 442.505725] WARNING: power/level is deprecated; use power/control instead
on
root@quark:/sys/bus/usb/devices/1-1/power# cat runtime_status
active
root@quark:/sys/bus/usb/devices/1-1/power# cat runtime_suspended_time
0
root@quark:/sys/bus/usb/devices/1-1/power#
```

Fig 32. Dynamic power management UI in Galileo board.

## DRIVER INTERFACE FOR POWER MANAGEMENT

The driver needs to define suspend, resume and reset resume methods.

Suspend: This method is called to warn the driver that device is going to be suspended.

Resume: This method is called to tell the driver that device is back ie(it is resumed) and driver can return to normal operation.

# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

Reset\_resume: This method is called to tell the driver that device is resumed and also reset and driver can return to its normal operation.

Drivers can enable autosuspend for the devices by calling : `usb_enable_autosuspend(struct usb_device *udev);` and can disable autosuspend by calling `usb_disable_autosuspend(struct usb_device *udev);`

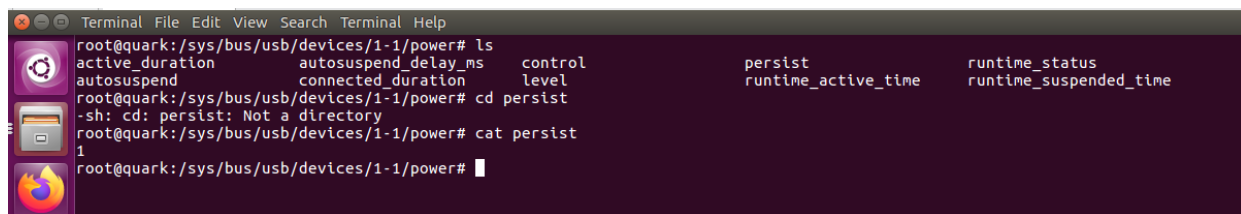
## USB DEVICE PERSISTENCE DURING SYSTEM SUSPEND

USB specification says that even when USB bus is suspended it must continuously supply suspended current so that device can maintain their internal state. If USB host controller losses its power during a system suspend it interrupts this power session of devices. When system wakes up Linux assumes that previous device was disconnected, and new device of same type was connected. This type of behavior might not cause problems in all type of devices, but problems might arise if the device is of mass-storage type. For example, say you have mounted file system on the storage device when system goes in to suspended state your entire file system becomes in accessible even when system comes out of suspended state.

In order to solve this issue a feature called USB-persist is included in the kernel. It basically makes USB data structures to persist through power session disruptions.

Whenever system comes out of suspended state the kernel sees whether USB host controller is same as expected if not it will perform a persistence check for all the USB devices for which the persist value is set. It sends reset signal to those devices if all the device descriptor data match i.e. device data before suspending and device data after resume then it continues to use old device data structures. If no device is detected after resuming it will destroy the previous data structure assuming that device is removed. You can see the persist enabled or not for a device by using the `sysfs`.

Let's see by connecting pen drive to Galileo board whether it enables this option. Since pen drive is a mass-storage device it should enable it.

A terminal window titled 'Terminal' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the following commands and output:

```
root@quark:/sys/bus/usb/devices/1-1/power# ls
active_duration      autosuspend_delay_ms  control
autosuspend          connected_duration    level
root@quark:/sys/bus/usb/devices/1-1/power# cd persist
-sh: cd: persist: Not a directory
root@quark:/sys/bus/usb/devices/1-1/power# cat persist
1
root@quark:/sys/bus/usb/devices/1-1/power#
```

Fig 33. Dynamic power management for connected USB pen drive device in Galileo board.

The value is 1 it indicates that it is enabled. If it is 0 then it indicates that it's not enabled.

# REPORT ON USB DRIVERS IN LINUX

Name: Danduri datta Manikanta Srihari

ASU Id: 1217423272

## Does USB-persist always good?

The straight answer is no. let's take a scenario where you have two identical pen drives and in one of them you have your filesystem mounted. When the system is in suspend state, you replaced your old pen drive with the new one which doesn't have any filesystem mounted. When system comes out of suspended state it will run the scan since both devices are identical it will assume it is the same old device and leaves all mappings unchanged. Thereby kernel gets fooled and it crashes data corruption also occurs. Although there are very few drawbacks USB-persist is very useful feature.

## Mutual Exclusion

When ever suspend or resume methods are called the (udev->dev.sem) will be held. It means that external suspend or resume events are mutually exclusive with probe, disconnect, pre\_reset, post\_reset calls. If a driver wants to block all suspend or resume calls during a critical section it can do it by calling usb\_autopm\_get\_interface() do the critical task and later it can reverse it.

## USB on the go?

We all use OTG in our day to day life. OTG stands for on the go it is the low performance, low cost way for a device to switch to be an host. Low power devices can use this way to communicate with devices.

## References

1. Linux Device Drivers, 3rd Edition by Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman.
2. <https://www.kernel.org/doc/html/v4.13/driver-api/usb/index.html>
3. <https://sysplay.github.io/books/LinuxDrivers/book/Content/Part11.html>
4. <http://linux-hotplug.sourceforge.net/>
5. <https://elixir.bootlin.com/linux/v3.19.8/>
6. <https://www.usb.org/>
7. [https://github.com/dandurisrihari/EOSI\\_FINAL\\_REPORT\\_USB\\_DATA\\_HACK](https://github.com/dandurisrihari/EOSI_FINAL_REPORT_USB_DATA_HACK)
8. Programming Guide for Linux USB Device Drivers (c) 2000 by Detlef Fliegl, deti@fliegl.de
9. <https://github.com/sslab-gatech/janus/tree/master/lkl/Documentation/usb>