CSE 530 - Embedded Operating System Internals - Spring 2020

Final Project Report - USB Driver in Linux

Shaym Joshi - 1218594676

sjoshi46@asu.edu

USB Overview:

Universal Serial Bus (USB) is a short distance digital communication connection between a host and a number of peripherals. It was created with the intention to support many devices over a single bus type. The USB bus is a single-master type implementation which means that the host is responsible to communicate with every device that is connected on the bus and continuously ask the device if it has any data to send. This feature allows any USB device to connect to the host without rebooting the system. USB protocol defines a set of standard rules for any type of devices to follow for communication. The device can be anything from mice, keyboard, storage device, camera, scanner, printer, external hard drives.

The Linux kernel USB system is depicted in the figure above. The Linux kernel has two types of drivers: driver on the host system and drivers on a device. The driver on the host system controls how to communicate with the device whereas the driver on the device is responsible for how the device looks to the host system.

To gain more insight into the linux USB system, we need to understand what USB driver and device is in the linux.

USB Device and Driver Architecture

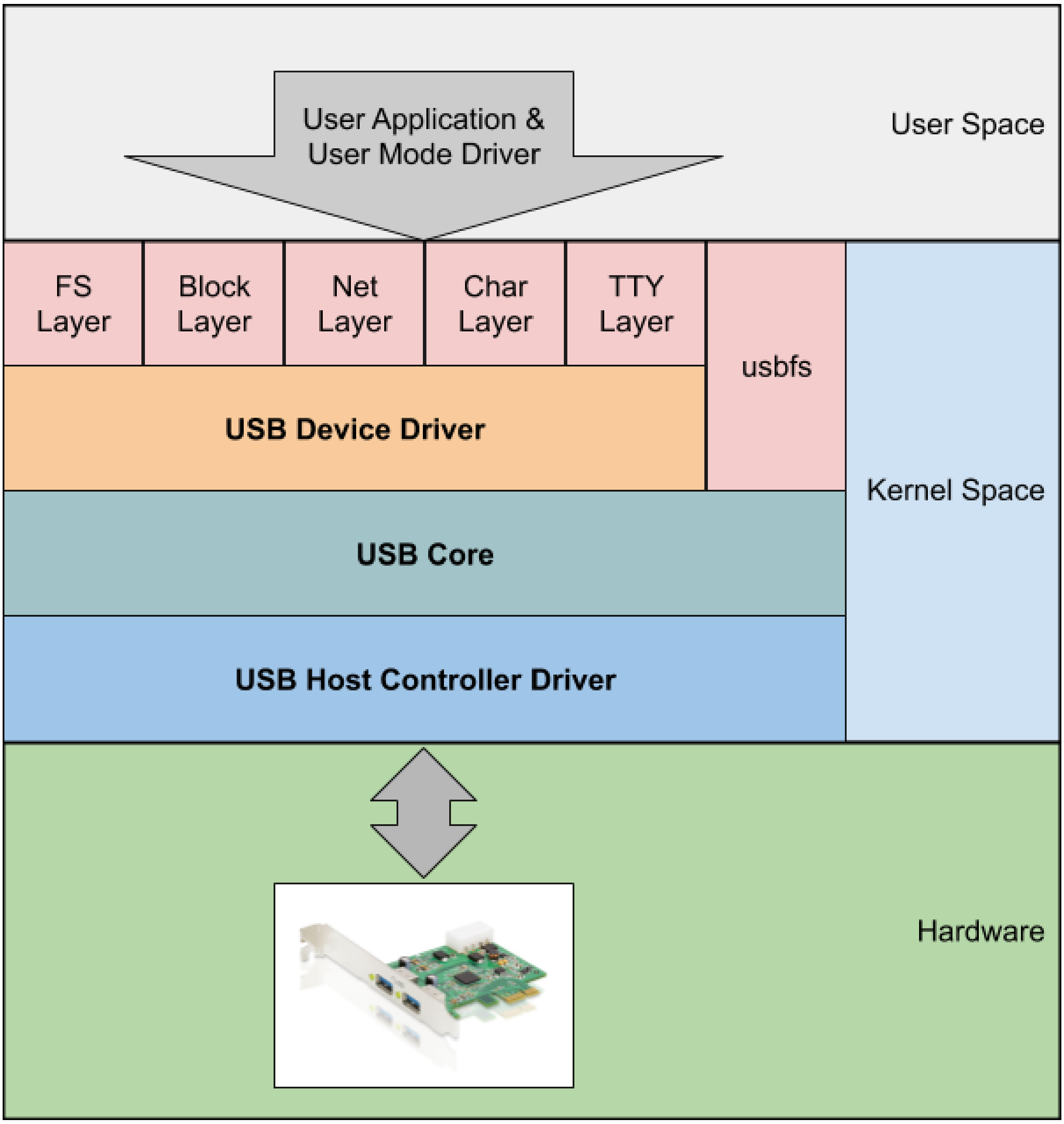


Figure 1

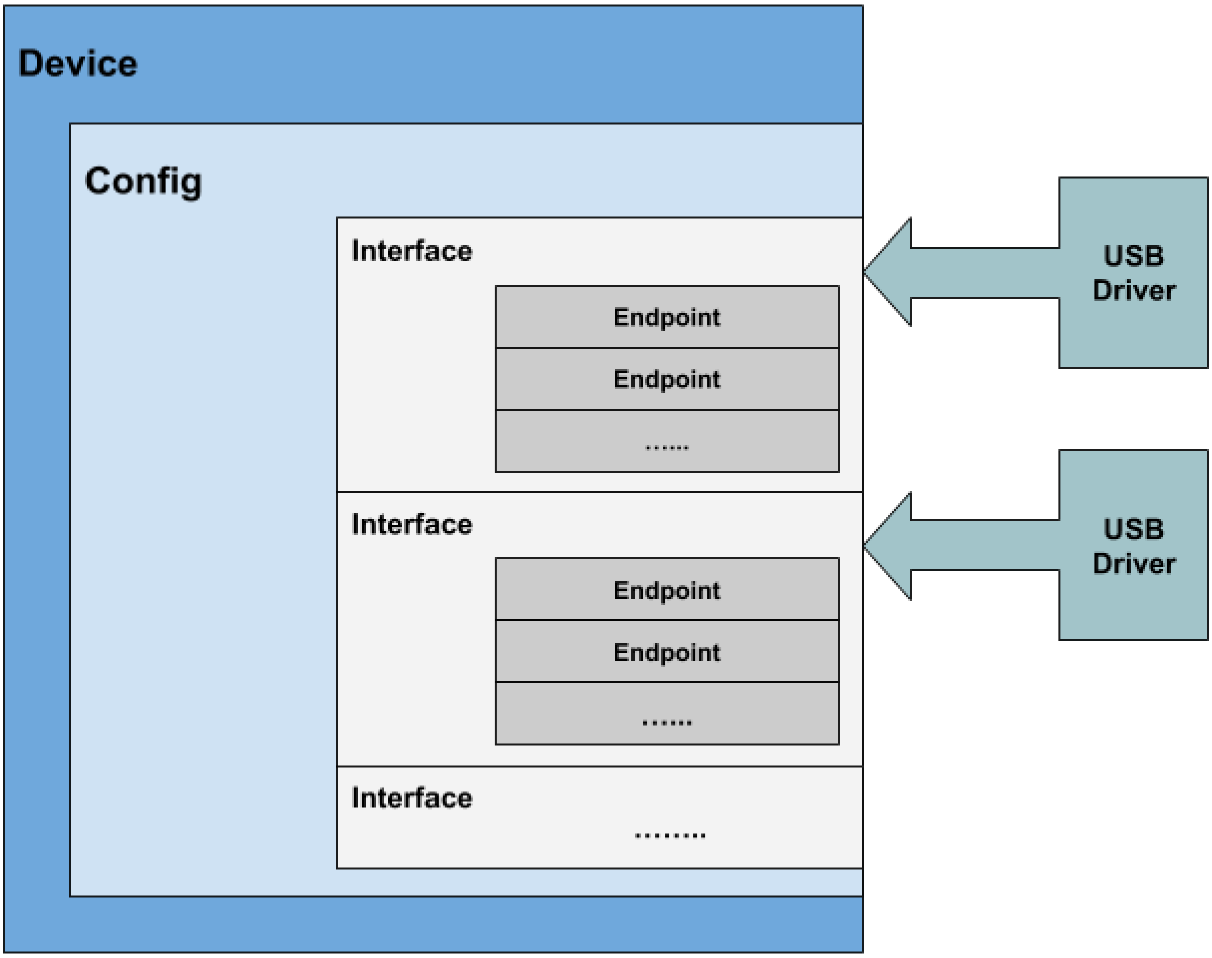


Figure 2

The figure 1 and figure 2 shows how the USB driver and device overview looks like.
Let's dwell deeper into each of these blocks. Every USB device contains one or more configurations [Linux supports only one configuration per device]. Depending on the configuration, the device may support one or more interfaces. For every interface, there will be one or more endpoints. The interface loosely translates to the functionality of the device that it provides, for example a printer can print, scan and fax through the same USB cable. Therefore, unlike other device drivers, USB device drivers are associated with the respective interfaces that it supports. Endpoint is a form of communication for that interface.

Endpoints

Endpoints can be thought of a medium through which information can enter the device from the driver or exit the device to the driver. At one time, the information can only from one point to the other meaning they are unidirectional. Depending on flow of information, direction of the endpoint can be IN (from device to host) represented by I in linux terminal and OUT (from host to device) represented by O in linux terminal.
Depending on the interface and type of information, there are four categories of endpoints for a device and they are:

1. Control
2. Interrupt
3. Bulk

4. Isochronous

Control endpoint

The control endpoints are meant for gaining information about the device during insertion, configuring the device through commands and reading status about the device. Every USB device has an endpoint called endpoint 0. It is used for asynchronous transfers. Driver for that device decides when to send data.

Interrupt endpoint

They are used for carrying small amounts of data and are the main mode of communication for a USB supported mouse and keyboard. They are used for periodic data transfers and a fixed bandwidth is reserved by the USB core.

Bulk endpoint

They are used for carrying large amounts of data with no time bound and mode of communication for storage devices, printers etc. It is also used for asynchronous transfers like control endpoints.

Isochronous endpoint

They are also used for carrying large amounts of data but with no implicit guarantee of reaching the destination. They are used where devices can bear loss of data. They are also used for periodic data transfers.

Control and bulk endpoints use bandwidth as it is available and the other two require reserved bandwidth for data transfers.

Go to terminal and type the following command: `$ usb-devices / $lsusb -v`

Figure 3 shows basic specifications about the devices that are presently connected on your computer. The first letter on each line depicts specific information for that device. For example D is for devices, C is for configuration, E is for endpoints, I is for Interfaces, P is for product details, T indicates the position of the device on the USB tree [USB system is represented in linux as a tree built from several point to point links and this tree is managed by hub driver] etc.

```
T:  Bus=01 Lev=01 Prnt=01 Port=04 Cnt=03 Dev#=  4 Spd=480 MxCh= 0
D:  Ver= 2.00 Cls=ef(misc ) Sub=02 Prot=01 MxPS=64 #Cfgs=  1
P:  Vendor=04f2 ProdID=b56c Rev=88.03
S:  Manufacturer=Generic
S:  Product=HP TrueVision HD
S:  SerialNumber=200901010001
C:  #Ifs= 2 Cfg#= 1 Atr=80 MxPwr=500mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=0e(video) Sub=01 Prot=00 Driver=uvcvideo
I:  If#= 1 Alt= 0 #EPs= 0 Cls=0e(video) Sub=02 Prot=00 Driver=uvcvideo
```

Figure 3

Every linux USB driver is represented in linux through following structure:

```
struct usb_driver {
       const char *name;

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

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

       const struct usb_device_id *id_table;

       struct usb_dynids dynids;
       struct usbdrv_wrap drvwrap;
       ..
       ..
};
```

- name: It is the pointer to the name of the driver.
- id_table: It is the pointer to the structure of type struct usb_device_id that contains a list of devices that is registered to this driver.
- probe: It is the pointer to the probe function of the USB driver and it is called when the USB core finds a suitable device for it.
- disconnect: It is the pointer to the disconnect function of the driver. It is invoked when the device registered for this driver is removed.name: It is the pointer to the name of the driver.
- id_table: It is the pointer to the structure of type struct usb_device_id that contains a list of devices that is registered to this driver.
- probe: It is the pointer to the probe function of the USB driver and it is called when the USB core finds a suitable device for it.
- disconnect: It is the pointer to the disconnect function of the driver. It is invoked when the device registered for this driver is removed.

```
struct usb_device_id {
       __u16          match_flags;
       __u16          idVendor;
       __u16          idProduct;
       __u16          bcdDevice_lo;
       __u16          bcdDevice_hi;
       __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))));
};
```

This list is used by USB core to decide which driver to invoke for a device. idVendor and idProduct are numbers assigned by USB forum and it is unique to every member. Remaining members from bcdDevice_lo till bInterfaceNumber are all assigned by USB forum [b int the beginning of the name specifies that it's values are expressed in BCD]. All these values are used by the USB drivers to differentiate between the different devices when the probe function gets invoked.

The probe function has first argument of type struct usb_interface *intf. Every interface in linux is defined by that structure.

```
struct usb_interface {
	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 */
	..
	..
	struct device dev;		/* interface specific device info
*/
	struct device *usb_dev;
	struct work_struct reset_ws;	/* for resets in atomic
context */
};
```

The usb_interface in the kernel has a member of type struct usb_host_interface and has pointer objects altsetting and cur_altsetting. They both are an array of alternate settings that may be used for a particular interface. The cur_altsetting denotes the currently available settings for this interface. num_altsetting denotes the number of alternate settings available for the configuration.

struct usb_host_interface has two members of interests and they are struct usb_interface_descriptor desc and struct usb_host_endpoint *endpoint.

```
struct usb_host_interface {
      struct usb_interface_descriptor  desc;

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

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

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

Structure usb_interface_descriptor contains all the information about the particular interface that is available in the given configuration. The information you obtained after `$ usb-device` command, in particular from the line beginning with letter 'I' is obtained from below structure.

```
struct usb_host_endpoint {
      struct usb_endpoint_descriptor        desc;
       ..
};
```

```
struct usb_endpoint_descriptor {
      __u8  bLength;
      __u8  bDescriptorType;
      __u8  bEndpointAddress;
      __u8  bmAttributes;
      __le16 wMaxPacketSize;
      __u8  bInterval;
};
```

- bEndpointAdress is the USB endpoint address of this specific endpoint.
- wMaxPacketSize is the maximum amount of data this endpoint is capable of transmitting.
- bInterval is the amount of time (ms) to wait between simultaneous transmissions of data.
- bmAttributes defines the type of the endpoint. USB_ENDPOINT_XFERTYPE_MASK should be masked with this value and compared with USB_ENDPOINT_XFER_ISOC, USB_ENDPOINT_XFER_BULK or USB_ENDPOINT_XFER_INT to determine if it’s type is isochronous, bulk or interrupt respectively.

USB Urbs

There are two ways of communicating with USB device and one of them is through urb i.e. USB request blocks and the other is through usb_bulk_msg function which we will discuss later. A urb is used to send or receive data to or from a specific USB endpoint on a specific USB device in an asynchronous manner. Depending on need, a USB driver can allocate multiple urb to communicate with the device for a single endpoint and every endpoint can handle a queue of urbs, so the driver can send multiple urb as per the status of the queue. Below figure 4 depicts the typical life cycle of a USB urb.

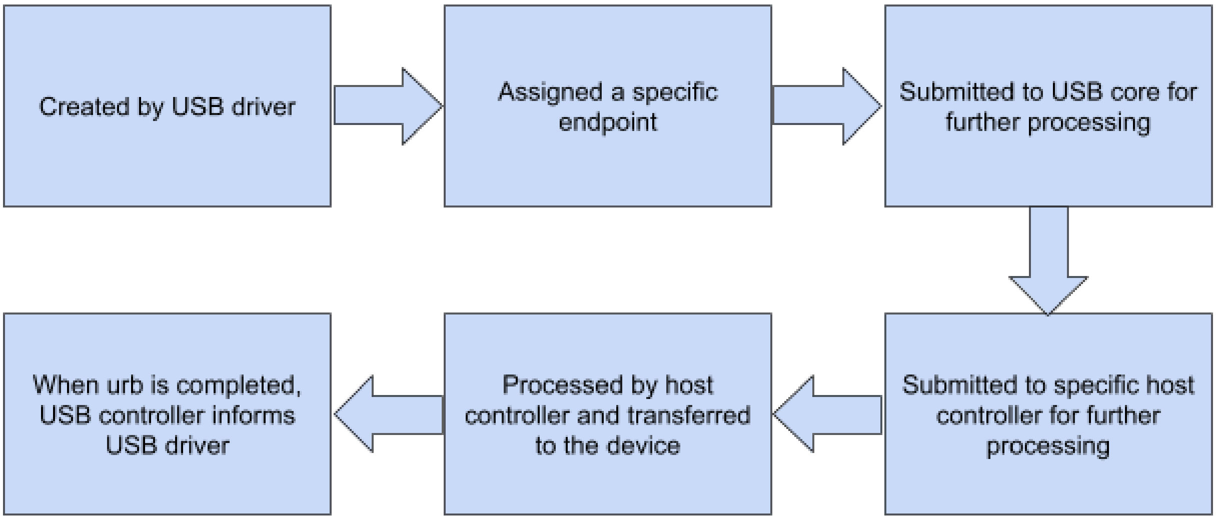


Figure 4

USB urb is represented by the following structure:

```
struct urb {
  struct usb_device * dev;
  unsigned int pipe;
  unsigned int transfer_flags;
  void * transfer_buffer;
  u32 transfer_buffer_length;
  u32 actual_length;
  int start_frame;
  int number_of_packets;
  int interval;
  int error_count;
  void * context;
  usb_complete_t complete;
  struct usb_iso_packet_descriptor iso_frame_desc[0];
  ..
};
```

- dev is the pointer object to the USB device.
- pipe contains the endpoint information for a specific endpoint.

There are various functions to set the type of pipe endpoint and they are listed as follows:

```
unsigned int usb_sndctrlpipe(struct usb_device *dev, unsigned int endpoint)
unsigned int usb_rcvctrlpipe(struct usb_device *dev, unsigned int endpoint)
unsigned int usb_sndbulkpipe(struct usb_device *dev, unsigned int endpoint)
unsigned int usb_rcvbulkpipe(struct usb_device *dev, unsigned int endpoint)
 unsigned int usb_sndintpipe(struct usb_device *dev, unsigned int endpoint)
 unsigned int usb_rcvintpipe(struct usb_device *dev, unsigned int endpoint)
unsigned int usb_sndisocpipe(struct usb_device *dev, unsigned int endpoint)
unsigned int usb_rcvisocpipe(struct usb_device *dev, unsigned int endpoint)
```

[snd denotes send which means the direction is OUT. rcv denotes receive which means the direction is IN.ctrl denotes control endpoint, bulk denotes bulk endpoint, int denotes interrupt endpoint, iso denotes isochronous endpoint]

- transfer_flags is used by the driver to describe characteristics of the urb message.
- transfer_buffer is the pointer to the buffer through which data is to be transferred in case of OUT endpoint or in case of IN endpoint it is the receiving buffer. It must be initialized using kmalloc before using it.
- transfer_buffer_length is the length of the buffer pointed by the transfer buffer pointer.
- complete is the pointer to the completion handler function called by the USB core urb has reached the device or when error has occurred.
- actual_length is set when the USB core is finished with urb transfer and it is the actual amount of data that is transferred or received depending on the direction of the endpoint.
- status value is accessed in the completion handler function for checking the status of process

Creating urbs

urb are dynamically created and they contain an internal reference count which helps automatically freed when the user releases it. It is created using the function usb_alloc_urb.

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

The first argument is used only when you want to create an isochronous urb otherwise it should be set to 0. The second argument is the same argument as the kmalloc flag argument.

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

	urb = kmalloc(struct_size(urb, iso_frame_desc, iso_packets),
		      mem_flags);
	if (!urb)
		return NULL;
	usb_init_urb(urb);
	return urb;
}
```

It eventually expands to kmalloc function and when memory is allocated, it calls the usb_init_urb function which is as follows:

```
void usb_init_urb(struct urb *urb)
{
	if (urb) {
		memset(urb, 0, sizeof(*urb));
		kref_init(&urb->kref);
		INIT_LIST_HEAD(&urb->urb_list);
		INIT_LIST_HEAD(&urb->anchor_list);
	}
}
```

It sets the memory allocated to urb to zero. kref is a structure to keep reference count for urb creation. `kref_init` function initializes the kref object.

After the urb has been created, it must be initialized properly **by the driver** before it can be used by the USB core. There are several APIs in linux that will help in doing so and depending on the type of urb to be created, one of the following APIs can be used.

1. 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, void *context, int interval);
2. 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);
3. 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, void *context);

As can be guessed from the name of the function, usb_fill_int_urb, usb_fill_bulk_urb, usb_fill_control_urb creates an interrupt, bulk and control urb respectively. Unfortunately there is no API for creating an isochronous urb. We need to manually initialize urb's members for that.

There are minor differences in each of them. We'll look at usb_fill_bulk_urb function:

```
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;
}
```

We’ll take a look into usb_device later, the rest of the function body is self explanatory, it merely assigns the members of urb with respective arguments passed in the function. For assigning value of pipe, usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr) can be used as described in the previous pages [or we can use usb_rcvbulkpipe function].

There is an extra argument in usb_fill_int_urb function “interval” that is for scheduling an interrupt urb. Also, there is an extra argument in usb_fill_control_hub function “setup_packet” that points to the setup data pack that is to be sent.

Submitting urbs

int usb_submit_urb(struct urb *urb, int mem_flags) function is used to submit an urb. The first argument is a pointer to the urb structure and the second argument is the same as used in the kmalloc flag argument.

Upon completion of usb_submit_urb, the completion handler function is called and the USB core handles the control of the urb to the driver. There are only three possibilities of completion handler function getting invoked and they are as follows:

1. The urb was successfully sent/received to/from the device. [return value of usb_submit_urb is 0]
2. Error occurred in midst of transferring the urb [which can be checked in the status value]
3. The device might’ve been unplugged in the midst of the function or when a cancellation call is made to the urb that has been submitted [usb_unlink_urb function is used for this purpose].

Cancelling urbs

There are two functions that are used to cancel an urb and they are:

```
int usb_kill_urb(struct urb *urb);
```

```
int usb_unlink_urb(struct urb *urb);
```

usb_kill_urb is to be used in the disconnect function of the driver when the device is disconnected and usb_unlink_urb is used when a urb that is submitted to the USB core needs to be stopped.

To be able to send a urb we need to create a custom driver for a device.

Sending USB message without urb

```
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,
void *data, int len, int *actual_length, int timeout);
```

The above function initializes a bulk urb message and sends it to the device [specified in the first argument], waits for completion before returning a value.

```
int usb_control_msg(struct usb_device *dev, unsigned int pipe,
__u8 request, __u8 requesttype, __u16 value, __u16 index, void
*data, __u16 size, int timeout);
```

The above function is the same as usb_bulk_msg except that it allows the driver to both send and receive control messages.

Writing USB driver

As we have seen earlier, USB driver in linux is represented by the following:

```
struct usb_driver {
      const char *name;

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

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

      const struct usb_device_id *id_table;

      struct usb_dynids dynids;
      struct usbdrv_wrap drvwrap;
      ..
      ..
};
```

As explained earlier we need to fill in the usb_device_id structure with appropriate details of our USB device to be registered with our driver.

We just need to fill in the idVendor and idProduct details and we can use USB_DEVICE marco as following:

```
static struct usb_device_id psuedo_table[] =
{
    { USB_DEVICE(0x04f2, 0xb56c) },
    {} /* Terminating entry */
};
```

USB_DEVICE macro is used to create a struct usb_device_id that matches a specific device.

```
#define USB_DEVICE(vend,prod)
      .idVendor = (vend),
      .idProduct = (prod)
```

[You can get your device's details using $ usb-device command]

```
MODULE_DEVICE_TABLE (usb, psuedo_table);
```

MODULE_DEVICE_TABLE macro is necessary to allow user-space tools to figure out what devices this driver can control.

To create our driver, only five fields need to be initialized of struct usb_driver as follows:

```
static struct usb_driver pseudo_driver =
{
    .name = "pseudo_driver",
    .probe = pseudo_probe,
    .disconnect = pseudo_disconnect,
    .id_table = pseudo_table,
};
```

Combining all of them we can provide a user space interface through character device to communicate with our device. Program for the same is shown below:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/usb.h>

static struct usb_device *device;
static struct usb_class_driver class;
static struct urb *pseudo_urb;

static struct buff
{
   int data;
} *bulk_buffer;

static int pseudo_open(struct inode *i, struct file *f)
{
   return 0;
}
static int pseudo_close(struct inode *i, struct file *f)
{
   return 0;
}
static ssize_t pseudo_read(struct file *f, char __user *buf,
size_t cnt, loff_t *off)
{
   return 0;
}

static void pseudo_write_bulk_callback(struct urb *urb, struct
pt_regs *regs)
```

```
{
   usb_free_urb(urb);
   return 0;
}

static ssize_t pseudo_write(struct file *f, const char __user
*buf, size_t cnt, loff_t *off)
{
   int retval;

   if (copy_from_user(bulk_buffer,buf,cnt)
   {
       return -EFAULT;
   }

   usb_fill_bulk_urb(urb, device->udev,
usb_sndbulkpipe(device->udev, device->bulk_out_endpointAddr),
bulk_buffer, cnt, pseudo_write_bulk_callback, device);

   usb_submit_urb(urb,GFP_KERNEL);

   return 0;
}

static struct file_operations fops =
{
   .owner = THIS_MODULE,
   .open = psuedo_open,
   .release = psuedo_close,
   .read = psuedo_read,
   .write = psuedo_write,
};

static int psuedo_probe(struct usb_interface *interface, const
struct usb_device_id *id)
{
   int retval;

   psuedo_urb = usb_alloc_urb(0,GFP_KERNEL);

   device = interface_to_usbdev(interface);

   class.name = "psuedo_device";

   class.fops = &fops;
```

```
    if ((retval = usb_register_dev(interface, &class)) < 0)
    {
        printk("Bad minor number request \n");
    }

    return retval;
}

static void psuedo_disconnect(struct usb_interface *interface)
{
    usb_deregister_dev(interface, &class);
}


static struct usb_device_id psuedo_table[] =
{
    { USB_DEVICE(0x04f2, 0xb56c) },
    {}
};

MODULE_DEVICE_TABLE (usb, psuedo_table);

static struct usb_driver psuedo_driver =
{
    .name = "psuedo_driver",
    .probe = psuedo_probe,
    .disconnect = psuedo_disconnect,
    .id_table = psuedo_table,
};

static int __init psuedo_init(void)
{
    int result;

    bulk_buffer = kmalloc(sizeof(struct buff),GFP_KERNEL);

    if ((result = usb_register(&psuedo_driver)))
    {
        printk(KERN_ERR "usb_register failed. Error number %d",
result);
    }
    return result;
}

static void __exit psuedo_exit(void)
{
```

```
    usb_deregister(&psuedo_driver);
}

module_init(psuedo_init);
module_exit(psuedo_exit);
```

psuedo_open, psuedo_close, psuedo_read, psuedo_write are the file operations associated with our device and through which we can communicate from user space to our device. All the information about the interface can be saved in the usb device using the function usb_set_intfdata(struct usb_interface *interface, void *dev). Usually this information is retrieved through usb_get_intfdata(struct usb_interface *interface) in the open function.

File operations are binded with the struct usb_device's object and struct usb_class_driver's object through the following function:

```
int usb_register_dev (struct usb_interface *intf, struct
usb_class_driver *class_driver);
```

```
struct usb_class_driver {
      char *name;
      char *(*devnode)(struct device *dev, umode_t *mode);
      const struct file_operations *fops;
      int minor_base;
};
```

The structure of usb_class_driver is given above.

- name is the device file system name associated with this device.
- fops is the pointer to the struct file_operations which is invoked when the device is accessed.
- minor_base is the start of the minor range value available.

```
int     usb_register_dev(struct     usb_interface     *intf,struct
usb_class_driver *class_driver)
{
      int retval;
      int minor_base = class_driver->minor_base;
      int minor;
      char name[20];

#ifdef CONFIG_USB_DYNAMIC_MINORS
      minor_base = 0;
```

```
#endif

	if (class_driver->fops == NULL)
		return -EINVAL;
	if (intf->minor >= 0)
		return -EADDRINUSE;

	mutex_lock(&init_usb_class_mutex);
	retval = init_usb_class();
	mutex_unlock(&init_usb_class_mutex);

	if (retval)
		return retval;

dev_dbg(&intf->dev,  "looking  for  a  minor,  starting  at  %d\n",
minor_base);

	down_write(&minor_rwsem);
	for (minor = minor_base; minor < MAX_USB_MINORS; ++minor) {
		if (usb_minors[minor])
			continue;

		usb_minors[minor] = class_driver->fops;
		intf->minor = minor;
		break;
	}
	if (intf->minor < 0) {
		up_write(&minor_rwsem);
		return -EXFULL;
	}

	/* create a usb class device for this usb interface */
	snprintf(name,   sizeof(name),   class_driver->name,   minor   -
minor_base);
	intf->usb_dev = device_create(usb_class->class, &intf->dev,
	MKDEV(USB_MAJOR,       minor),       class_driver,       "%s",
kbasename(name));
	if (IS_ERR(intf->usb_dev)) {
		usb_minors[minor] = NULL;
		intf->minor = -1;
		retval = PTR_ERR(intf->usb_dev);
	}
	up_write(&minor_rwsem);
	return retval;
}
```

usb_register_dev program is described above. When usb_register_dev is invoked, the devfs node will be created if devfs is enabled, and a usb class device is created in sysfs at /sys/class/usb/. Exact opposite is done in usb_unregister_dev [which should be invoked when the device has been removed from the system] , node will be deleted from devfs, class will be deleted and minor number will be returned to USB core.

[It will assign a minor number (given CONFIG_USB_DYNAMIC_MINORS is enabled) dynamically from the next available minor number. [USB has a default major number reserved and that is 180] If CONFIG_USB_DYNAMIC_MINORS is disabled then it will assign the next available minor number beginning from class_driver->minor_base.]

Real time example - FTDI USB-Serial Interface

As we are using a USB to Serial converter cable to get the debug data on the host system from the target environment contains FT232RL a USB to Serial converter IC. It sends the USB descriptor information using the control endpoint to the host when it is connected to the host.

To get the same details as seen in the image provided below, run the following command:

```
$ lsusb -d 0403:6001 -v
```

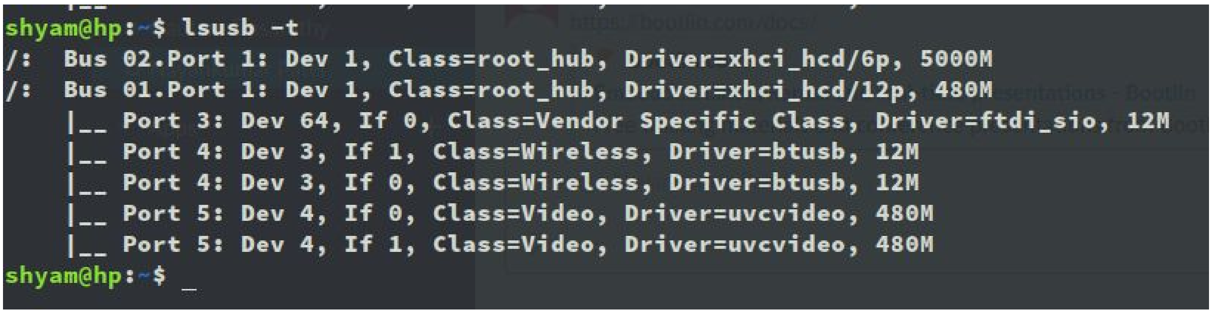
where 0403 and 6001 is the vendor and product id which can be obtained using the `$lsusb` command when you connect the cable to your computer.

```
shyam@hp:~$ lsusb -d 0403:6001 -v

Bus 001 Device 063: ID 0403:6001 Future Technology Devices International, Ltd FT232 USB-Serial (UART) IC
Couldn't open device, some information will be missing
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass            0 (Defined at Interface level)
  bDeviceSubClass         0
  bDeviceProtocol         0
  bMaxPacketSize0         8
  idVendor           0x0403 Future Technology Devices International, Ltd
  idProduct          0x6001 FT232 USB-Serial (UART) IC
  bcdDevice            6.00
  iManufacturer           1
  iProduct                2
  iSerial                 3
  bNumConfigurations      1
  Configuration Descriptor:
    bLength                 9
    bDescriptorType         2
    wTotalLength           32
    bNumInterfaces          1
    bConfigurationValue     1
    iConfiguration          0
    bmAttributes         0xa0
      (Bus Powered)
      Remote Wakeup
    MaxPower               90mA
    Interface Descriptor:
      bLength                 9
      bDescriptorType         4
      bInterfaceNumber        0
      bAlternateSetting       0
      bNumEndpoints           2
      bInterfaceClass       255 Vendor Specific Class
      bInterfaceSubClass    255 Vendor Specific Subclass
      bInterfaceProtocol    255 Vendor Specific Protocol
      iInterface              2
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x81  EP 1 IN
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0040  1x 64 bytes
        bInterval               0
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x02  EP 2 OUT
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0040  1x 64 bytes
        bInterval               0
```

Figure 5

Now, let's see how we got this information to the user level. We also look into how we can communicate with the connected device using a user space library like libusb/libftd2xx.



```
shyam@hp:~$ lsusb -t
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/6p, 5000M
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/12p, 480M
    |__ Port 3: Dev 64, If 0, Class=Vendor Specific Class, Driver=ftdi_sio, 12M
    |__ Port 4: Dev 3, If 1, Class=Wireless, Driver=btusb, 12M
    |__ Port 4: Dev 3, If 0, Class=Wireless, Driver=btusb, 12M
    |__ Port 5: Dev 4, If 0, Class=Video, Driver=uvcvideo, 480M
    |__ Port 5: Dev 4, If 1, Class=Video, Driver=uvcvideo, 480M
shyam@hp:~$ _
```

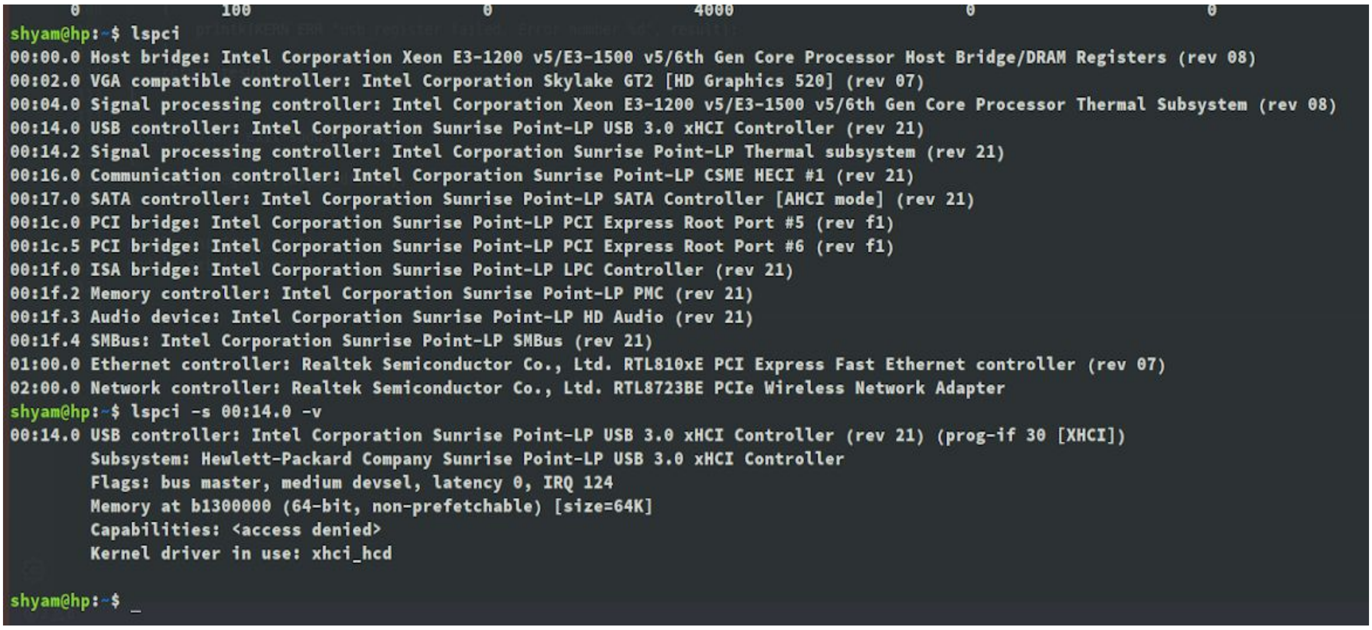
Figure 6

When you run the following command,

```
$ lsusb -t
```

the information you get will look like figure 6. The driver for our USB-serial device is xhci_hcd/12p which is our host control driver. The host control driver [refer to figure 1] is responsible for informing the USB core that a USB device has been detected [hardware space detection] irrespective of the presence of it's driver in linux. It is responsible for giving device specific information [regardless of whether it is a USB device or not] such as device id, product id, endpoint information, to the USB core.

You can check the xhcd driver registered on pci bus using the commands shown in the below image.



```
shyam@hp:~$ lspci
00:00.0 Host bridge: Intel Corporation Xeon E3-1200 v5/E3-1500 v5/6th Gen Core Processor Host Bridge/DRAM Registers (rev 08)
00:02.0 VGA compatible controller: Intel Corporation Skylake GT2 [HD Graphics 520] (rev 07)
00:04.0 Signal processing controller: Intel Corporation Xeon E3-1200 v5/E3-1500 v5/6th Gen Core Processor Thermal Subsystem (rev 08)
00:14.0 USB controller: Intel Corporation Sunrise Point-LP USB 3.0 xHCI Controller (rev 21)
00:14.2 Signal processing controller: Intel Corporation Sunrise Point-LP Thermal subsystem (rev 21)
00:16.0 Communication controller: Intel Corporation Sunrise Point-LP CSME HECI #1 (rev 21)
00:17.0 SATA controller: Intel Corporation Sunrise Point-LP SATA Controller [AHCI mode] (rev 21)
00:1c.0 PCI bridge: Intel Corporation Sunrise Point-LP PCI Express Root Port #5 (rev f1)
00:1c.5 PCI bridge: Intel Corporation Sunrise Point-LP PCI Express Root Port #6 (rev f1)
00:1f.0 ISA bridge: Intel Corporation Sunrise Point-LP LPC Controller (rev 21)
00:1f.2 Memory controller: Intel Corporation Sunrise Point-LP PMC (rev 21)
00:1f.3 Audio device: Intel Corporation Sunrise Point-LP HD Audio (rev 21)
00:1f.4 SMBus: Intel Corporation Sunrise Point-LP SMBus (rev 21)
01:00.0 Ethernet controller: Realtek Semiconductor Co., Ltd. RTL810xE PCI Express Fast Ethernet controller (rev 07)
02:00.0 Network controller: Realtek Semiconductor Co., Ltd. RTL8723BE PCIe Wireless Network Adapter
shyam@hp:~$ lspci -s 00:14.0 -v
00:14.0 USB controller: Intel Corporation Sunrise Point-LP USB 3.0 xHCI Controller (rev 21) (prog-if 30 [XHCI])
        Subsystem: Hewlett-Packard Company Sunrise Point-LP USB 3.0 xHCI Controller
        Flags: bus master, medium devsel, latency 0, IRQ 124
        Memory at b1300000 (64-bit, non-prefetchable) [size=64K]
        Capabilities: <access denied>
        Kernel driver in use: xhci_hcd

shyam@hp:~$ _
```

Figure 7

When the information about the device is sent to USB core by host controller device, the USB core at the end does the following things:

1. bus_register(): It registers a driver core subsystem
2. usb_host_init(): It is used to initialize a usb_bus structure
3. usb_major_init(): This call registers a character driver as well as devfs_mk_dir("usb") and registers class
4. usbfs_init(): Creates a node in usb fs for our device.

Now we have the usbfs interface which is an abstraction for user space of kernel level. At user space we can use user mode device drivers usually packed as applications or libraries like libusb in linux.

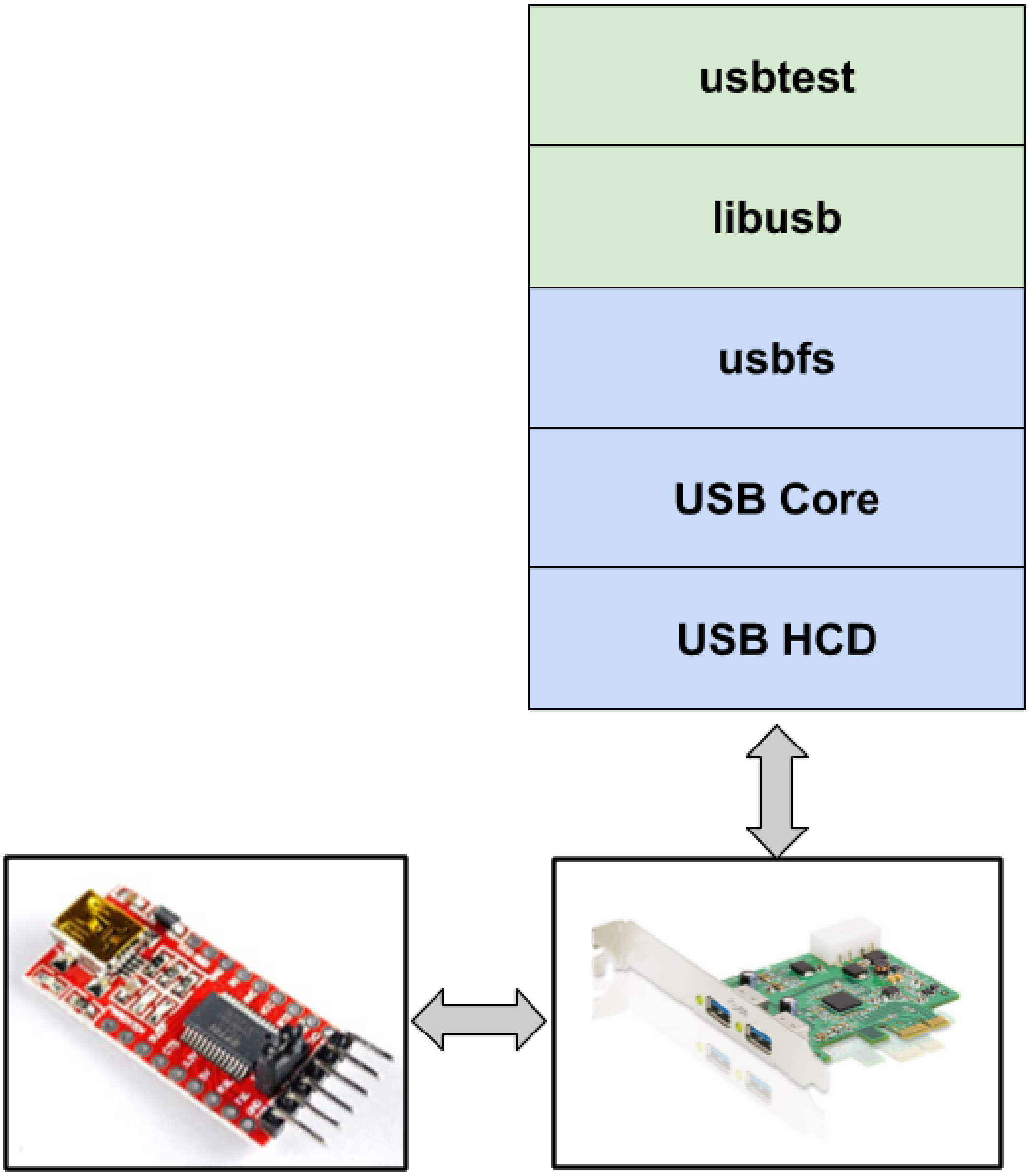


Figure 8

Test application using libusb to get the device information of our FTDI cable.

Run the usbtest.c file using the instructions given in the README.txt provided with the zip file.

You will get output similar to the figure given below.

References

1. https://elixir.bootlin.com/linux/latest/source
2. Linux Device Driver Third Edition by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman.
3. https://stackoverflow.com/
4. https://www.kernel.org/doc/html/v4.12/driver-api/usb/index.html