Device Drivers, Part 11: USB Drivers in Linux

By Anil Kumar Pugalia on October 1, 2011 in Coding, Developers · 75 Comments

This article, which is part of the series on Linux device drivers, gets you started with writing your first USB driver in Linux.

Pugs' pen drive was the device Shweta was playing with, when both of them sat down to explore the world of USB drivers in Linux. The fastest way to get the hang of it, and Pugs' usual way, was to pick up a USB device, and write a driver for it, to experiment with. So they chose a pen drive (a.k.a. USB stick) that was at hand — a JetFlash from Transcend, with vendor ID `0x058f` and product ID `0x6387`.

USB device detection in Linux

Whether a driver for a USB device is there or not on a Linux system, a valid USB device will always be detected at the hardware and kernel spaces of a USB-enabled Linux system, since it is designed (and detected) as per the USB protocol specifications. Hardware-space detection is done by the USB host controller — typically a native bus device, like a PCI device on x86 systems. The corresponding host controller driver would pick and translate the low-level physical layer information into higher-level USB protocol-specific information. The USB protocol formatted information about the USB device is then populated into the generic USB core layer (the usbcore driver) in kernel-space, thus enabling the detection of a USB device in kernel-space, even without having its specific driver.

After this, it is up to various drivers, interfaces, and applications (which are dependent on the various Linux distributions), to have the user-space view of the detected devices. Figure 1 shows a top-to-bottom view of the USB subsystem in Linux.

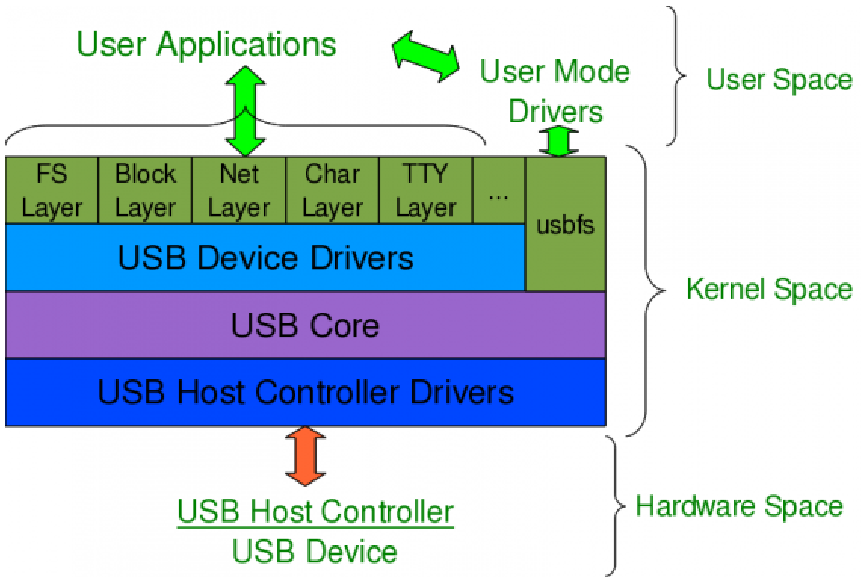
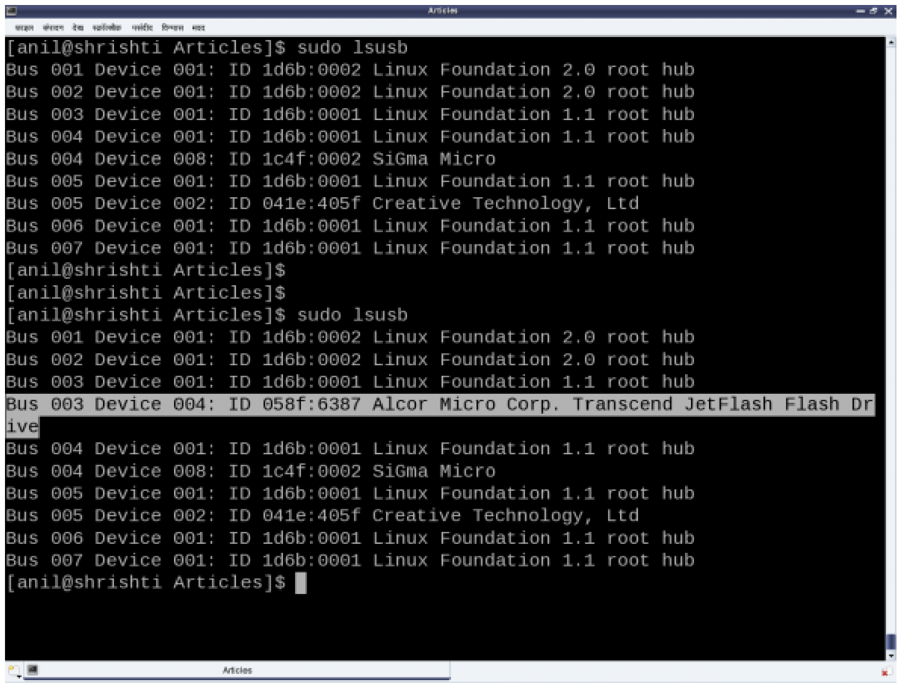


Figure 1: USB subsystem in Linux

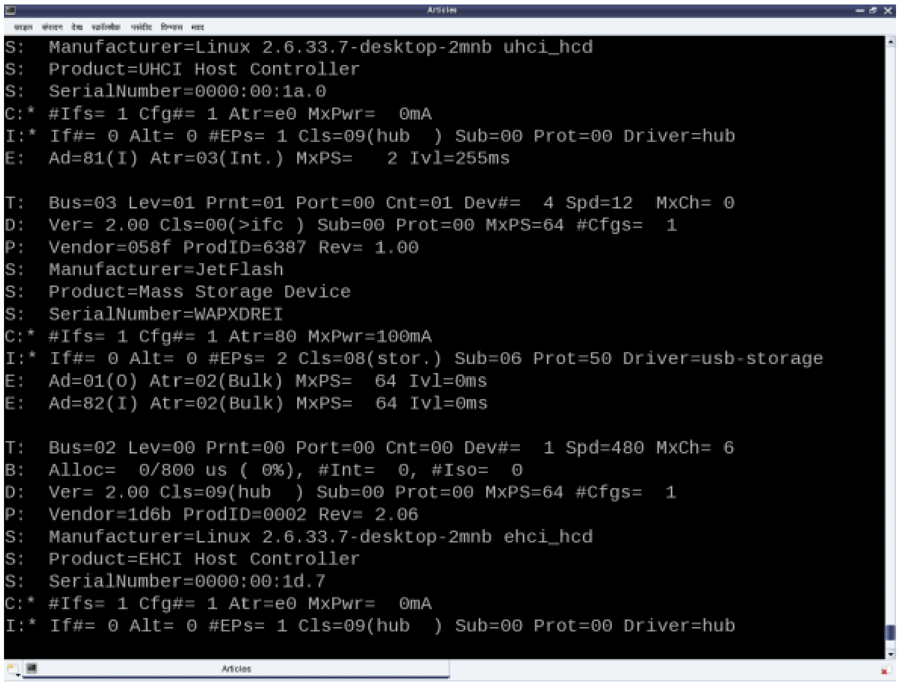
A basic listing of all detected USB devices can be obtained using the `lsusb` command, as root. Figure 2 shows this, with and without the pen drive plugged in. A `-v` option to `lsusb`provides detailed information.



```
[anil@shrishti Articles]$ sudo lsusb
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 008: ID 1c4f:0002 SiGma Micro
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 005 Device 002: ID 041e:405f Creative Technology, Ltd
Bus 006 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 007 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
[anil@shrishti Articles]$
[anil@shrishti Articles]$
[anil@shrishti Articles]$ sudo lsusb
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 004: ID 058f:6387 Alcor Micro Corp. Transcend JetFlash Flash Dr
ive
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 008: ID 1c4f:0002 SiGma Micro
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 005 Device 002: ID 041e:405f Creative Technology, Ltd
Bus 006 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 007 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
[anil@shrishti Articles]$ 
```

Figure 2: Output of lsusb

In many Linux distributions like Mandriva, Fedora,… the `usbfs` driver is loaded as part of the default configuration. This enables the detected USB device details to be viewed in a more techno-friendly way through the `/proc` window, using `cat /proc/bus/usb/devices`. Figure 3 shows a typical snippet of the same, clipped around the pen drive-specific section. The listing basically contains one such section for each valid USB device detected on the system.



```
S:  Manufacturer=Linux 2.6.33.7-desktop-2mnb uhci_hcd
S:  Product=UHCI Host Controller
S:  SerialNumber=0000:00:1a.0
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=255ms

T:  Bus=03 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  4 Spd=12  MxCh= 0
D:  Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
P:  Vendor=058f ProdID=6387 Rev= 1.00
S:  Manufacturer=JetFlash
S:  Product=Mass Storage Device
S:  SerialNumber=WAPXDREI
C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=100mA
I:* If#= 0 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50 Driver=usb-storage
E:  Ad=01(O) Atr=02(Bulk) MxPS=  64 Ivl=0ms
E:  Ad=82(I) Atr=02(Bulk) MxPS=  64 Ivl=0ms

T:  Bus=02 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=480 MxCh= 6
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= 2.06
S:  Manufacturer=Linux 2.6.33.7-desktop-2mnb ehci_hcd
S:  Product=EHCI Host Controller
S:  SerialNumber=0000:00:1d.7
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
```

Figure 3: USB's proc window snippet

Decoding a USB device section

To further decode these sections, a valid USB device needs to be understood first. All valid USB devices contain one or more configurations. A configuration of a USB device is like a profile, where the default one is the commonly used one. As such, Linux supports only one configuration per device — the default one. For every configuration, the device may have one or more interfaces. An interface corresponds to a function provided by the device.

There would be as many interfaces as the number of functions provided by the device. So, say an MFD (multi-function device) USB printer can do printing, scanning and faxing, then it most likely would have at least three interfaces, one for each of the functions. So, unlike other device drivers, a USB device driver is typically associated/written per interface, rather than the device as a whole — meaning that one USB device may have multiple device drivers, and different device interfaces may have the same driver — though, of course, one interface can have a maximum of one driver only.

It is okay and fairly common to have a single USB device driver for all the interfaces of a USB device. The `Driver=...` entry in the `proc` window output (Figure 3) shows the interface to driver mapping — a `(none)` indicating no associated driver.

For every interface, there would be one or more end-points. An end-point is like a pipe for transferring information either into or from the interface of the device, depending on the functionality. Based on the type of information, the endpoints have four types: Control, Interrupt, Bulk and Isochronous.

As per the USB protocol specification, all valid USB devices have an implicit special control end-point zero, the only bi-directional end-point. Figure 4 shows the complete pictorial representation of a valid USB device, based on the above explanation.

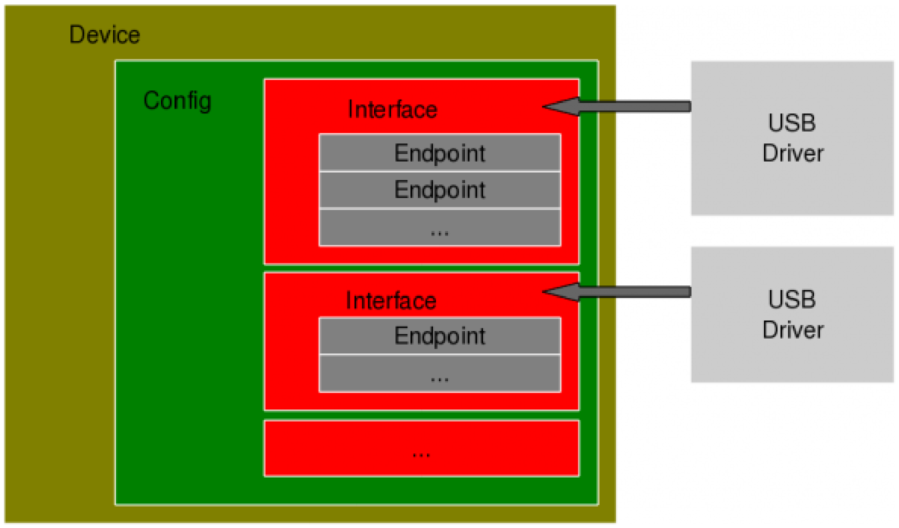


Figure 4: USB device overview

Coming back to the USB device sections (Figure 3), the first letter on each line represents the various parts of the USB device specification just explained. For example, D for device, C for configuration, I for interface, E for endpoint, etc. Details about these and various others are available in the kernel source, in `Documentation/usb/proc_usb_info.txt`.

The USB pen drive driver registration

“Seems like there are so many things to know about the USB protocol, to be able to write the first USB driver itself — device configuration, interfaces, transfer pipes, their four types, and so many other symbols like T, B, S, … under a USB device specification,” sighed Shweta.

“Yes, but don’t you worry — all of that can be covered in detail later. Let’s do first things first — get the pen drive’s interface associated with our USB device driver (`pen_register.ko`),” consoled Pugs.

Like any other Linux device driver, here, too, the constructor and the destructor are required — basically the same driver template that has been used for all the drivers. However, the content would vary, as this is a hardware protocol layer driver, i.e., a horizontal driver, unlike a character driver, which was one of the vertical drivers discussed earlier. The difference would be that instead of registering with and unregistering from VFS, here this would be done with the corresponding protocol layer — the USB core in this case; instead of providing a user-space interface like a device file, it would get connected with the actual device in hardware-space.

The USB core APIs for the same are as follows (prototyped in `<linux/usb.h>`):

```
int usb_register(struct usb_driver *driver);
void usb_deregister(struct usb_driver *);
```

As part of the `usb_driver` structure, the fields to be provided are the driver’s name, ID table for auto-detecting the particular device, and the two callback functions to be invoked by the USB core during a hot plugging and a hot removal of the device, respectively.

Putting it all together, `pen_register.c` would look like what follows:

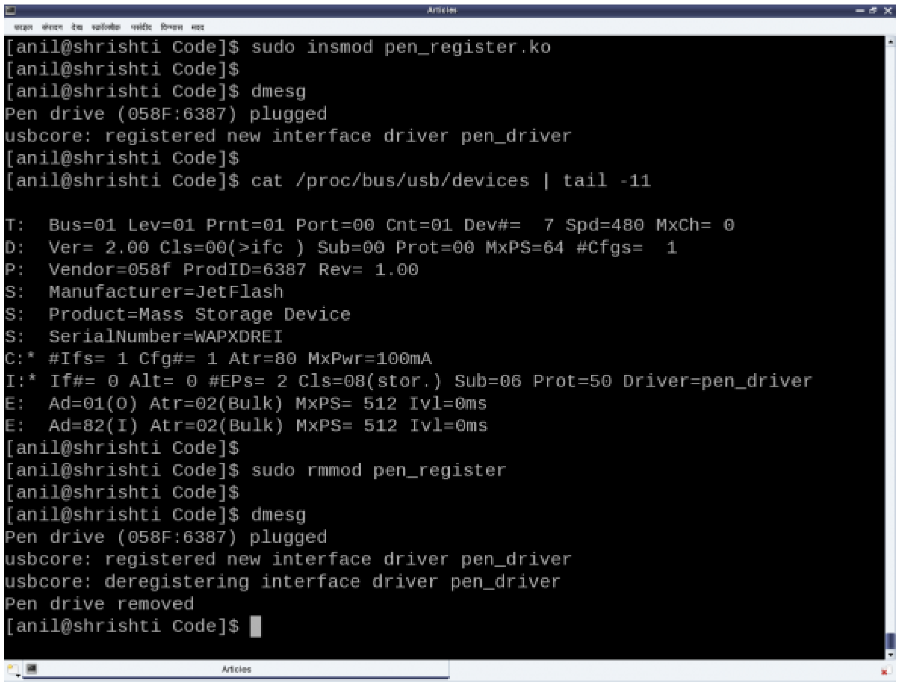
```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/usb.h>

static int pen_probe(struct usb_interface *interface, const struct usb_device_id
{
    printk(KERN_INFO "Pen drive (%04X:%04X) plugged\n", id->idVendor, id->idProd
    return 0;
}

static void pen_disconnect(struct usb_interface *interface)
{
    printk(KERN_INFO "Pen drive removed\n");
}

static struct usb_device_id pen_table[] =
{
    { USB_DEVICE(0x058F, 0x6387) },
    {} /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, pen_table);

static struct usb_driver pen_driver =
{
    .name = "pen_driver",
    .id_table = pen_table,
    .probe = pen_probe,
    .disconnect = pen_disconnect,
};

static int __init pen_init(void)
{
    return usb_register(&pen_driver);
}

static void __exit pen_exit(void)
{
    usb_deregister(&pen_driver);
}

module_init(pen_init);
module_exit(pen_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Anil Kumar Pugalia <email_at_sarika-pugs_dot_com>");
MODULE_DESCRIPTION("USB Pen Registration Driver");
```

Then, the usual steps for any Linux device driver may be repeated:

- Build the driver (`.ko` file) by running `make`.
- Load the driver using `insmod`.
- List the loaded modules using `lsmod`.
- Unload the driver using `rmmod`.

But surprisingly, the results wouldn't be as expected. Check `dmesg` and the `proc` window to see the various logs and details. This is not because a USB driver is different from a character driver — but there's a catch. Figure 3 shows that the pen drive has one interface (numbered 0), which is already associated with the usual usb-storage driver.

Now, in order to get our driver associated with that interface, we need to unload the usb-storage driver (i.e., `rmmod usb-storage`) and replug the pen drive. Once that's done, the results would be as expected. Figure 5 shows a glimpse of the possible logs and a `proc`window snippet. Repeat hot-plugging in and hot-plugging out the pen drive to observe the probe and disconnect calls in action.



```
[anil@shrishti Code]$ sudo insmod pen_register.ko
[anil@shrishti Code]$
[anil@shrishti Code]$ dmesg
Pen drive (058F:6387) plugged
usbcore: registered new interface driver pen_driver
[anil@shrishti Code]$
[anil@shrishti Code]$ cat /proc/bus/usb/devices | tail -11

T:  Bus=01 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  7 Spd=480 MxCh= 0
D:  Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
P:  Vendor=058f ProdID=6387 Rev= 1.00
S:  Manufacturer=JetFlash
S:  Product=Mass Storage Device
S:  SerialNumber=WAPXDREI
C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=100mA
I:* If#= 0 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50 Driver=pen_driver
E:  Ad=01(O) Atr=02(Bulk) MxPS= 512 Ivl=0ms
E:  Ad=82(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms
[anil@shrishti Code]$
[anil@shrishti Code]$ sudo rmmod pen_register
[anil@shrishti Code]$
[anil@shrishti Code]$ dmesg
Pen drive (058F:6387) plugged
usbcore: registered new interface driver pen_driver
usbcore: deregistering interface driver pen_driver
Pen drive removed
[anil@shrishti Code]$
```

Figure 5: Pen driver in action

Summing up