Device Drivers, Part 12: USB Drivers in Linux Continued

By Anil Kumar Pugalia on November 1, 2011 in Coding, Developers · 34 Comments

The 12th part of the series on Linux device drivers takes you further along the path to writing your first USB driver in Linux — a continuation from the previous article.

Pugs continued, "Let's build upon the USB device driver coded in our previous session, using the same handy JetFlash pen drive from Transcend, with the vendor ID 0x058f and product ID 0×6387. For that, let's dig further into the USB protocol, and then convert our learning into code."

USB endpoints and their types

Depending on the type and attributes of information to be transferred, a USB device may have one or more endpoints, each belonging to one of the following four categories:

- Control — to transfer control information. Examples include resetting the device, querying information about the device, etc. All USB devices always have the default control endpoint point as zero.
- Interrupt — for small and fast data transfers, typically of up to 8 bytes. Examples include data transfer for serial ports, human interface devices (HIDs) like keyboards, mouse, etc.
- Bulk — for big but comparatively slower data transfers. A typical example is data transfers for mass-storage devices.
- Isochronous — for big data transfers with a bandwidth guarantee, though data integrity may not be guaranteed. Typical practical usage examples include transfers of time-sensitive data like audio, video, etc.

Additionally, all but control endpoints could be "in" or "out", indicating the direction of data transfer; "in" indicates data flow from the USB device to the host machine, and "out", the other way.

Technically, an endpoint is identified using an 8-bit number, the most significant bit (MSB) of which indicates the direction — 0 means "out", and 1 means "in". Control endpoints are bi-directional, and the MSB is ignored.

Figure 1 shows a typical snippet of USB device specifications for devices connected on a system.

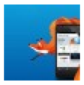
Search for: Search

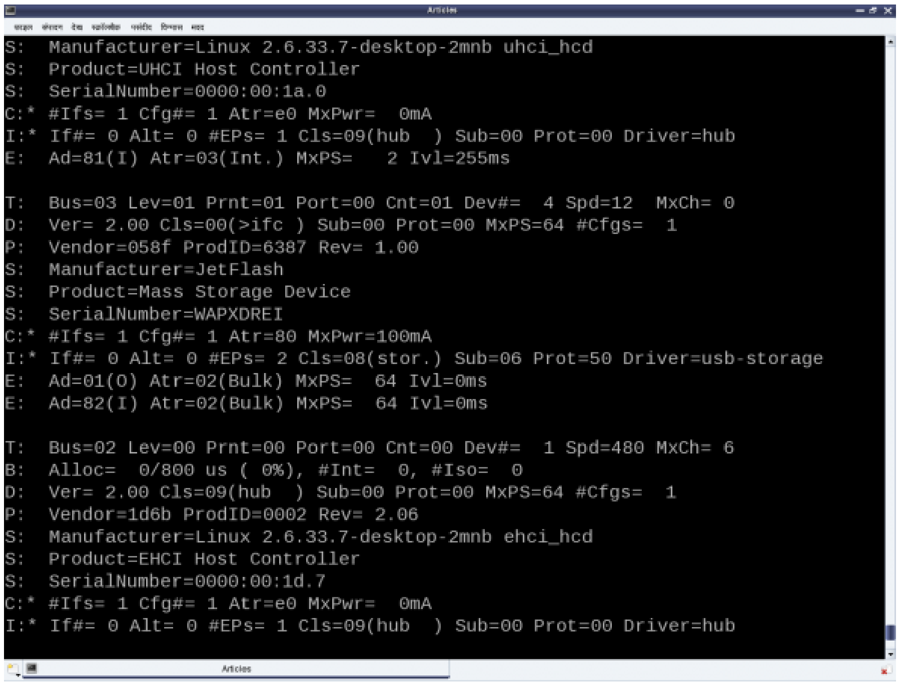
Get Connected

RSS Feed Twitter

Popular Comments Tag cloud

December 10, 2013 · 4 Comments · Diksha P Gupta
"We are where we are because of open source technology"


February 14, 2014 · 3 Comments · Sahil Chelaramani
Learn How to Write Apps for the Firefox OS



```
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 1: USB's proc window snippet (click for larger view)

To be specific, the `E:` lines in the figure show examples of an interrupt endpoint of a UHCI Host Controller, and two bulk endpoints of the pen drive under consideration. Also, the endpoint numbers (in hex) are, respectively, `0x81`, `0x01` and `0x82` — the MSB of the first and third being `1`, indicating 'in' endpoints, represented by `(I)` in the figure; the second is an `(O)` or 'out' endpoint. `MxPS` specifies the maximum packet size, i.e., the data size that can be transferred in a single go. Again, as expected, for the interrupt endpoint, it is `2 (<=8)`, and `64` for the bulk endpoints. `Ivl` specifies the interval in milliseconds to be given between two consecutive data packet transfers for proper transfer, and is more significant for the interrupt endpoints.

Decoding a USB device section

As we have just discussed regarding the `E:` line, it is the right time to decode the relevant fields of others as well. In short, these lines in a USB device section give a complete overview of the device as per the USB specifications, as discussed in our previous article.

Refer back to Figure 1. The first letter of the first line of every device section is a `T`, indicating the position of the device in the USB tree, uniquely identified by the triplet `<usb bus number, usb tree level, usb port>`. `D` represents the device descriptor, containing at least the device version, device class/category, and the number of configurations available for this device.

There would be as many `C` lines as the number of configurations, though typically, it is one. `C`, the configuration descriptor, contains its index, the device attributes in this configuration, the maximum power (actually, current) the device would draw in this configuration, and the number of interfaces under this configuration.

Depending on this, there would be at least that many `I` lines. There could be more in case of an interface having alternates, i.e., the same interface number but with different properties — a typical scenario for Web-cams.

`I` represents the interface descriptor with its index, alternate number, the functionality class/category of this interface, the driver associated with this interface, and the number of endpoints under this interface.

The interface class may or may not be the same as that of the device class. And depending on the number of endpoints, there would be as many `E` lines, details of which have already been discussed earlier.

The `*` after the `C` and `I` represents the currently active configuration and interface, respectively. The `P` line provides the vendor ID, product ID, and the product revision. `S` lines are string

November 1, 2013 · 3 Comments · Shobhit Gupta
Want Freedom? Initiate Linux In Your Life Today

February 14, 2014 · 3 Comments · Tuukka Turto
Programming Can Be Fun with Hy

February 10, 2014 · 2 Comments · Priyanka Sarkar
FOSS Helps Naaptol Take E-Commerce to New Heights!

descriptors showing up some vendor-specific descriptive information about the device.

“Peeping into cat `/proc/bus/usb/devices` is good in order to figure out whether a device has been detected or not, and possibly to get the first-cut overview of the device. But most probably this information would be required to write the driver for the device as well. So, is there a way to access it using `C` code?” Shweta asked.

“Yes, definitely; that’s what I am going to tell you about, next. Do you remember that as soon as a USB device is plugged into the system, the USB host controller driver populates its information into the generic USB core layer? To be precise, it puts that into a set of structures embedded into one another, exactly as per the USB specifications,” Pugs replied.

The following are the exact data structures defined in `<linux/usb.h>`, ordered here in reverse, for flow clarity:

```
struct usb_device
{
    …
    struct usb_device_descriptor descriptor;
    struct usb_host_config *config, *actconfig;
    …
};
struct usb_host_config
{
    struct usb_config_descriptor desc;
    …
    struct usb_interface *interface[USB_MAXINTERFACES];
    …
};
struct usb_interface
{
    struct usb_host_interface *altsetting /* array */, *cur_altsetting;
    …
};
struct usb_host_interface
{
    struct usb_interface_descriptor desc;
    struct usb_host_endpoint *endpoint /* array */;
    …
};
struct usb_host_endpoint
{
    struct usb_endpoint_descriptor desc;
    …
};
```

So, with access to the `struct usb_device` handle for a specific device, all the USB-specific information about the device can be decoded, as shown through the `/proc` window. But how does one get the device handle?

In fact, the device handle is not available directly in a driver; rather, the per-interface handles (pointers to `struct usb_interface`) are available, as USB drivers are written for device interfaces rather than the device as a whole.

Recall that the probe and disconnect callbacks, which are invoked by the USB core for every interface of the registered device, have the corresponding interface handle as their first parameter. Refer to the prototypes below:

```
int (*probe)(struct usb_interface *interface, const struct usb_device_id *id);
void (*disconnect)(struct usb_interface *interface);
```

So, with the interface pointer, all information about the corresponding interface can be accessed — and to get the container device handle, the following macro comes to the rescue:

```
struct usb_device device = interface_to_usbdev(interface);
```

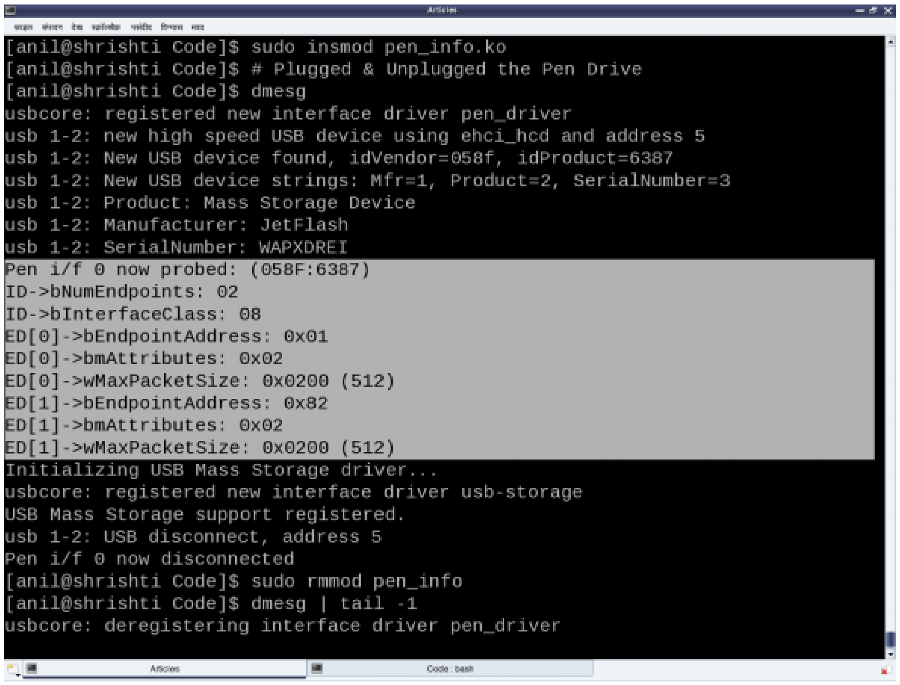
Adding this new learning into last month’s registration-only driver gets the following code listing (`pen_info.c`):

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

static struct usb_device *device;

static int pen_probe(struct usb_interface *interface, const struct usb_device_id
{
    struct usb_host_interface *iface_desc;
    struct usb_endpoint_descriptor *endpoint;
    int i;

    iface_desc = interface->cur_altsetting;
    printk(KERN_INFO "Pen i/f %d now probed: (%04X:%04X)\n",
            iface_desc->desc.bInterfaceNumber, id->idVendor, id->idProduct);
    printk(KERN_INFO "ID->bNumEndpoints: %02X\n",
            iface_desc->desc.bNumEndpoints);
    printk(KERN_INFO "ID->bInterfaceClass: %02X\n",
            iface_desc->desc.bInterfaceClass);

    for (i = 0; i < iface_desc->desc.bNumEndpoints; i++)
    {
        endpoint = &iface_desc->endpoint[i].desc;

        printk(KERN_INFO "ED[%d]->bEndpointAddress: 0x%02X\n",
                i, endpoint->bEndpointAddress);
        printk(KERN_INFO "ED[%d]->bmAttributes: 0x%02X\n",
                i, endpoint->bmAttributes);
        printk(KERN_INFO "ED[%d]->wMaxPacketSize: 0x%04X (%d)\n",
                i, endpoint->wMaxPacketSize, endpoint->wMaxPacketSize);
    }

    device = interface_to_usbdev(interface);
    return 0;
}

static void pen_disconnect(struct usb_interface *interface)
{
    printk(KERN_INFO "Pen i/f %d now disconnected\n",
            interface->cur_altsetting->desc.bInterfaceNumber);
}

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",
    .probe = pen_probe,
    .disconnect = pen_disconnect,
    .id_table = pen_table,
};

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@sarika-pugs.com>");
MODULE_DESCRIPTION("USB Pen Info Driver");
```

Then, the usual steps for any Linux device driver may be repeated, along with the pen drive steps:

- Build the driver (`pen_info.ko` file) by running make.
- Load the driver using `insmod pen_info.ko`.
- Plug in the pen drive (after making sure that the usb-storage driver is not already loaded).
- Unplug the pen drive.
- Check the output of `dmesg` for the logs.
- Unload the driver using `rmmod pen_info`.

Figure 2 shows a snippet of the above steps on Pugs' system. Remember to ensure (in the output of `cat /proc/bus/usb/devices`) that the usual `usb-storage` driver is not the one associated with the pen drive interface, but rather the `pen_info` driver.



```
[anil@shrishti Code]$ sudo insmod pen_info.ko
[anil@shrishti Code]$ # Plugged & Unplugged the Pen Drive
[anil@shrishti Code]$ dmesg
usbcore: registered new interface driver pen_driver
usb 1-2: new high speed USB device using ehci_hcd and address 5
usb 1-2: New USB device found, idVendor=058f, idProduct=6387
usb 1-2: New USB device strings: Mfr=1, Product=2, SerialNumber=3
usb 1-2: Product: Mass Storage Device
usb 1-2: Manufacturer: JetFlash
usb 1-2: SerialNumber: WAPXDREI
Pen i/f 0 now probed: (058F:6387)
ID->bNumEndpoints: 02
ID->bInterfaceClass: 08
ED[0]->bEndpointAddress: 0x01
ED[0]->bmAttributes: 0x02
ED[0]->wMaxPacketSize: 0x0200 (512)
ED[1]->bEndpointAddress: 0x82
ED[1]->bmAttributes: 0x02
ED[1]->wMaxPacketSize: 0x0200 (512)
Initializing USB Mass Storage driver...
usbcore: registered new interface driver usb-storage
USB Mass Storage support registered.
usb 1-2: USB disconnect, address 5
Pen i/f 0 now disconnected
[anil@shrishti Code]$ sudo rmmod pen_info
[anil@shrishti Code]$ dmesg | tail -1
usbcore: deregistering interface driver pen_driver
```

Figure 2: Output of dmesg

Summing up

Before taking another break, Pugs shared two of the many mechanisms for a driver to specify its device to the USB core, using the `struct usb_device_id` table. The first one is by specifying the `<vendor id, product id>` pair using the `USB_DEVICE()` macro (as done above). The second one is by specifying the device class/category using the `USB_DEVICE_INFO()` macro. In fact, many more macros are available in `<linux/usb.h>` for various combinations. Moreover, multiple of these macros could be specified in the `usb_device_id` table (terminated by a null entry), for matching with any one of the criteria, enabling to write a single driver for possibly many devices.

“Earlier, you mentioned writing multiple drivers for a single device, as well. Basically, how do we selectively register or not register a particular interface of a USB device?”, queried Shweta. “Sure. That’s next in line of our discussion, along with the ultimate task in any device driver — the data-transfer mechanisms,” replied Pugs.

Related Posts:

- Device Drivers, Part 11: USB Drivers in Linux
- Device Drivers, Part 13: Data Transfer to and from USB…
- Device Drivers, Part 6: Decoding Character Device File…
- Device Drivers, Part 5: Character Device Files —…
- Device Drivers, Part 17: Module Interactions

Tags: LFY November 2011, linux device drivers, Linux Device Drivers Series, mass storage devices, serial ports, usb device driver, usb devices, USB host controller driver, usb protocol

Article written by:

Anil Kumar Pugalia

The author is a freelance trainer in Linux internals, Linux device drivers, embedded Linux and related topics. Prior to this, he had worked at Intel and Nvidia. He has been exploring Linux since 1994. A gold medallist from the Indian Institute of Science, Linux and knowledge-sharing are two of his many passions.

Connect with him: **Website** - **Twitter** - **Facebook** - **Google+**