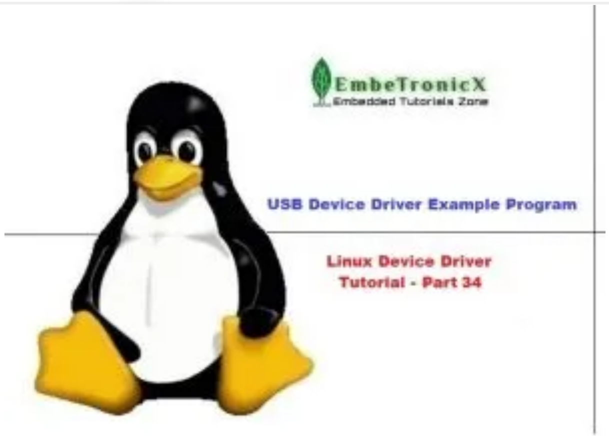
EmbeTronicX
Embedded Tutorials Zone

Sidebar

Device Drivers



Linux Device Driver Tutorial Part 34 – USB Device Driver Example – 2

This is the Series on Linux Device Driver. The aim of this series is to provide easy and practical examples that anyone can understand. This is the Linux Device Driver Tutorial Part 34 – USB Device Driver Example program in the Linux Device Driver.

Apple Music Turns 5 as It Continues Rivalry With Spotify

Post Contents [hide]

USB Device Driver Example

Prerequisites

Before starting this USB device driver example, I would recommend you to understand the USB device concepts using the below link.

- USB Device Driver Basics

Introduction

In our last tutorial, we have gone through the big theory part that explains the functionality of the USB device and its subsystem in Linux. So now, we can go straight away to the example.

USB Driver API

This is just like a character device driver. But the structure and others will be different. Let's see.

usb_driver structure

USB driver needs to do is register itself with the Linux USB subsystem (USB core). So while registering we need to give some information about which devices the driver supports and which functions to call when a device supported by the driver is inserted or removed from the system. All of this information is passed to the USB subsystem via **usb_driver** 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);
  int (* 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 soft_unbind:1;
};
```

Where,

<**name**>: The driver name should be unique among USB drivers, and should normally be the same as the module name.

<**probe**>: The function needs to be called when a USB device is connected.

<**disconnect**>: The function needs to be called when a USB device is disconnected.

<**ioctl**>: Used for drivers that want to talk to userspace through the “**usbfs**” filesystem.

<**suspend**>: Called when the device is going to be suspended by the system.

<**resume**>: Called when the device is being resumed by the system.

<**reset_resume**>: Called when the suspended device has been reset instead of being resumed.

<**pre_reset**>: Called by **usb_reset_device** when the device is about to be reset.

<**post_reset**>: Called by **usb_reset_device** after the device has been reset.

<**id_table**>: USB drivers use an ID table to support hotplugging. Export this with MODULE_DEVICE_TABLE(usb,...). This must be set or your driver’s probe function will never get called.

<**dynids**>: used internally to hold the list of dynamically added device ids for this driver.

<**drvwrap**>: Driver-model core structure wrapper.

<**no_dynamic_id**>: if set to 1, the USB core will not allow dynamic ids to be added to this driver by preventing the sysfs file from being created.

<**supports_autosuspend**>: if set to 0, the USB core will not allow auto suspend for interfaces bound to this driver.

<**soft_unbind**>: if set to 1, the USB core will not kill URBs and disable endpoints before calling the driver's disconnect method.

USB interface drivers must provide a name, **probe** and **disconnect** methods, and an **id_table**. Other driver fields are optional.

id_table

The **id_table** is used in hotplugging. It holds a set of descriptors, and specialized data may be associated with each entry. That table is used by both user and kernel mode hotplugging support.

The following code tells the hotplug scripts that this module supports a single device with a specific vendor and product ID:

```
const struct usb_device_id etx_usb_table[] = {
    { USB_DEVICE( USB_VENDOR_ID, USB_PRODUCT_ID ) },    //Put your USB device's Vendo
    { } /* Terminating entry */
};

MODULE_DEVICE_TABLE(usb, etx_usb_table);
```

Notes: Make sure that you have replaced the vendor id & device id with your USB device in the above code example.

probe

When a device is plugged into the USB bus that matches the device ID pattern that your driver registered with the USB core, the **probe** function is called.

The driver now needs to verify that this device is actually accepted or not. If it is accepted, it returns 0. If not, or if any error occurs during initialization, an error code (such as **-ENOMEM** or **-ENODEV**) is returned from the probe function.

Example snippet of the probe

```c
/*
** This function will be called when USB device is inserted.
*/
static int etx_usb_probe(struct usb_interface *interface,
                         const struct usb_device_id *id)
{
    dev_info(&interface->dev, "USB Driver Probed: Vendor ID : 0x%02x,\t"
             "Product ID : 0x%02x\n", id->idVendor, id->idProduct);

    return 0;  //return 0 indicates we are managing this device
}
```

disconnect

When a device is plugged out or removed, this function will be getting called.

Example snippet of the disconnect

```c
/*
** This function will be called when USB device is removed.
*/
static void etx_usb_disconnect(struct usb_interface *interface)
{
    dev_info(&interface->dev, "USB Driver Disconnected\n");
}
```

Example snippet of usb_driver structure

Once you have written the **probe**, **disconnect** functions, and **id_table**, then you have to assign their address to the **usb_driver** structure like below.

```c
static struct usb_driver etx_usb_driver = {
    .name       = "EmbeTronicX USB Driver",
    .probe      = etx_usb_probe,
    .disconnect = etx_usb_disconnect,
    .id_table   = etx_usb_table,
};
```

As of now, we have finished the basic kinds of stuff. Now we need to register the USB device with a USB core.

Register the USB device driver to the USB Subsystem (USB core)

This API is used to register the USB driver to the USB subsystem.

```
usb_register (struct usb_driver * your_usb_driver);
```

Where,

<`your_usb_driver`>: The structure which will tell the address of `probe`, `disconnect`, and `id_table`.

Example

```
usb_register(&etx_usb_driver);
```

Deregister the USB device driver from the USB subsystem

This API is used to deregister the USB driver from the USB subsystem.

```
usb_deregister (struct usb_driver * your_usb_driver);
```

Where,

<`your_usb_driver`>: The structure which will tell the address of `probe`,

`disconnect`, and **`id_table`**.

Example

```
usb_deregister(&etx_usb_driver);
```

Initialize and exit function

We have completed all the things. But where should we call the **`usb_register`** and **`usb_deregister`** function? It is just simple. Like a character device driver, we have to do this in **`__init`** and **`__exit`** functions. Refer to the below example.

```
static int __init etx_usb_init(void)
{
    return usb_register(&etx_usb_driver);
}

static void __exit etx_usb_exit(void)
{
    usb_deregister(&etx_usb_driver);
}

module_init(etx_usb_init);
module_exit(etx_usb_exit);
```

module_usb_driver

Is that all? Yes if you use the kernel older than 3.3. But if you are using the latest Linux kernel which is greater than 3.3, then there is one another option that you can use. You can eliminate this **`usb_register`**, **`usb_deregister`** ,**`__init`** , **`__exit`**, **`module_init`** and **`module_exit`** functions in one line. ie, You can eliminate all the code which has been provided above can be replaced by below one line.

```
module_usb_driver(__usb_driver);
```

Where,

<**`__usb_driver`**>: usb_driver structure

This is the helper macro for registering a USB driver. This macro for USB drivers which do not do anything special in module init/exit. This eliminates a lot of boilerplate. Each module may only use this macro once, and calling it replaces **module_init** and **module_exit**.

Example Programming

We have written some very basic USB device drivers which will just print the Interface descriptor and Endpoint descriptor while inserting the device. And we have used both the old method (**module_init** and **module_exit**) and the new method (**module_usb_driver**). You can choose anyone by changing the macro **IS_NEW_METHOD_USED**. If you set 0, then it will use the old method. If you set 1 or non-zero value, it will use the new method.

And one more point, we have used the **dev_info** API here to print the data. **dev_info** is similar to **pr_info**, but also print some information about device(**struct device**), passed to them as the first argument. This information may help to filter the system log for messages, belonging to the concrete device.

Just go through the code below. You can easily understand this if you have followed our previous character device driver tutorials.

Note: In this tutorial, I have not used any separate microcontrollers. I have used my ubuntu as a host which is installed by using VirtualBox and My mobile phone as a USB device. This is just for learning purposes. And I have used the Linux kernel 5.3.0-42-generic.

Driver Source Code

```
/***************************************************************************//**
*  \file       usb_driver.c
*
*  \details    Simple USB driver explanation
*
*  \author     EmbeTronicX
*
*  \Tested with kernel 5.3.0-42-generic
*
*******************************************************************************/
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/usb.h>

/*
** This macro is used to tell the driver to use old method or new method.
```

```
**
**  If it is 0, then driver will use old method. ie: __init and __exit
**  If it is non zero, then driver will use new method. ie: module_usb_driver
*/
#define IS_NEW_METHOD_USED  ( 1 )


#define USB_VENDOR_ID       ( 0x22d9 )      //USB device's vendor ID
#define USB_PRODUCT_ID      ( 0x2764 )      //USB device's product ID


#define PRINT_USB_INTERFACE_DESCRIPTOR( i )                         \
{                                                                   \
    pr_info("USB_INTERFACE_DESCRIPTOR:\n");                         \
    pr_info("-----------------------------\n");                     \
    pr_info("bLength: 0x%x\n", i.bLength);                          \
    pr_info("bDescriptorType: 0x%x\n", i.bDescriptorType);          \
    pr_info("bInterfaceNumber: 0x%x\n", i.bInterfaceNumber);        \
    pr_info("bAlternateSetting: 0x%x\n", i.bAlternateSetting);      \
    pr_info("bNumEndpoints: 0x%x\n", i.bNumEndpoints);              \
    pr_info("bInterfaceClass: 0x%x\n", i.bInterfaceClass);          \
    pr_info("bInterfaceSubClass: 0x%x\n", i.bInterfaceSubClass);    \
    pr_info("bInterfaceProtocol: 0x%x\n", i.bInterfaceProtocol);    \
    pr_info("iInterface: 0x%x\n", i.iInterface);                    \
    pr_info("\n");                                                  \
}

#define PRINT_USB_ENDPOINT_DESCRIPTOR( e )                          \
{                                                                   \
    pr_info("USB_ENDPOINT_DESCRIPTOR:\n");                          \
    pr_info("------------------------\n");                          \
    pr_info("bLength: 0x%x\n", e.bLength);                          \
    pr_info("bDescriptorType: 0x%x\n", e.bDescriptorType);          \
    pr_info("bEndPointAddress: 0x%x\n", e.bEndpointAddress);        \
    pr_info("bmAttributes: 0x%x\n", e.bmAttributes);                \
    pr_info("wMaxPacketSize: 0x%x\n", e.wMaxPacketSize);            \
    pr_info("bInterval: 0x%x\n", e.bInterval);                      \
    pr_info("\n");                                                  \
}

/*
** This function will be called when USB device is inserted.
*/
static int etx_usb_probe(struct usb_interface *interface,
                        const struct usb_device_id *id)
{
    unsigned int i;
    unsigned int endpoints_count;
    struct usb_host_interface *iface_desc = interface->cur_altsetting;

    dev_info(&interface->dev, "USB Driver Probed: Vendor ID : 0x%02x,\t"
             "Product ID : 0x%02x\n", id->idVendor, id->idProduct);

    endpoints_count = iface_desc->desc.bNumEndpoints;

    PRINT_USB_INTERFACE_DESCRIPTOR(iface_desc->desc);

     for ( i = 0; i < endpoints_count; i++ ) {
          PRINT_USB_ENDPOINT_DESCRIPTOR(iface_desc->endpoint[i].desc);
     }
    return 0;  //return 0 indicates we are managing this device
}

```

```
/*
** This function will be called when USB device is removed.
*/
static void etx_usb_disconnect(struct usb_interface *interface)
{
    dev_info(&interface->dev, "USB Driver Disconnected\n");
}

//usb_device_id provides a list of different types of USB devices that the driver s
const struct usb_device_id etx_usb_table[] = {
    { USB_DEVICE( USB_VENDOR_ID, USB_PRODUCT_ID ) },    //Put your USB device's Ven
    { } /* Terminating entry */
};

//This enable the linux hotplug system to load the driver automatically when the de
MODULE_DEVICE_TABLE(usb, etx_usb_table);

//The structure needs to do is register with the linux subsystem
static struct usb_driver etx_usb_driver = {
    .name       = "EmbeTronicX USB Driver",
    .probe      = etx_usb_probe,
    .disconnect = etx_usb_disconnect,
    .id_table   = etx_usb_table,
};

#if ( IS_NEW_METHOD_USED == 0 )
//This will replaces module_init and module_exit.
module_usb_driver(etx_usb_driver);

#else
static int __init etx_usb_init(void)
{
    //register the USB device
    return usb_register(&etx_usb_driver);
}

static void __exit etx_usb_exit(void)
{
    //deregister the USB device
    usb_deregister(&etx_usb_driver);
}

module_init(etx_usb_init);
module_exit(etx_usb_exit);
#endif

MODULE_LICENSE("GPL");
MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com>");
MODULE_DESCRIPTION("A simple device driver - USB Driver");
MODULE_VERSION("1.30");
```

Makefile

```
obj-m += usb_driver.o

KDIR = /lib/modules/$(shell uname -r)/build

all:
        make -C $(KDIR)  M=$(shell pwd) modules

clean:
	make -C $(KDIR)  M=$(shell pwd) clean
```

Execution

Notes: Make sure that you have replaced the vendor id & device id with your USB device in the above code example.

- Build the driver by using Makefile (**`sudo make`**)
- Load the driver using **`sudo insmod usb_driver.ko`**
- Check the **`dmesg`**

```
etx@embetronicx-VirtualBox:~/Desktop/LDD$ dmesg
[10083.327683] usbcore: registered new interface driver EmbeTronicX USB Driver
```

- Our driver is linked with the USB subsystem.
- Connect the appropriate USB device with the correct vendor id and product id. We should see our **`etx_usb_probe`** function getting called.
- Check the **`dmesg`**

```
etx@embetronicx-VirtualBox:~/Desktop/LDD$ dmesg
[10729.917928] usb 1-2: new full-speed USB device number 8 using ohci-pci
[10730.506085] usb 1-2: config 1 interface 0 altsetting 0 endpoint 0x81 has invalid maxp
[10730.506100] usb 1-2: config 1 interface 0 altsetting 0 endpoint 0x1 has invalid maxpa
[10730.533276] usb 1-2: New USB device found, idVendor=22d9, idProduct=2764, bcdDevice=
[10730.533278] usb 1-2: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[10730.533279] usb 1-2: Product: SDM712-MTP _SN:4470FA87
[10730.533280] usb 1-2: Manufacturer: realme
[10730.533281] usb 1-2: SerialNumber: 4470fa87
[10730.553868] EmbeTronicX USB Driver 1-2:1.0: USB Driver Probed: Vendor ID : 0x22d9,
[10730.553869] USB_INTERFACE_DESCRIPTOR:
[10730.553869] -----------------------------
[10730.553870] bLength: 0x9
[10730.553870] bDescriptorType: 0x4
[10730.553871] bInterfaceNumber: 0x0
[10730.553871] bAlternateSetting: 0x0
[10730.553872] bNumEndpoints: 0x3
[10730.553872] bInterfaceClass: 0xff
[10730.553872] bInterfaceSubClass: 0xff
[10730.553873] bInterfaceProtocol: 0x0
[10730.553873] iInterface: 0x5

[10730.553874] USB_ENDPOINT_DESCRIPTOR:
[10730.553874] ------------------------
[10730.553875] bLength: 0x7
[10730.553875] bDescriptorType: 0x5
[10730.553876] bEndPointAddress: 0x81
[10730.553876] bmAttributes: 0x2
[10730.553877] wMaxPacketSize: 0x40
[10730.553877] bInterval: 0x0

[10730.553878] USB_ENDPOINT_DESCRIPTOR:
[10730.553878] ------------------------
[10730.553878] bLength: 0x7
```

```
[10730.553879] bDescriptorType: 0x5
[10730.553879] bEndPointAddress: 0x1
[10730.553879] bmAttributes: 0x2
[10730.553880] wMaxPacketSize: 0x40
[10730.553880] bInterval: 0x0

[10730.553881] USB_ENDPOINT_DESCRIPTOR:
[10730.553881] ------------------------
[10730.553882] bLength: 0x7
[10730.553882] bDescriptorType: 0x5
[10730.553882] bEndPointAddress: 0x82
[10730.553883] bmAttributes: 0x3
[10730.553883] wMaxPacketSize: 0x1c
[10730.553884] bInterval: 0x6
```

- Cool. Now disconnect the USB device and check `dmesg`

```
etx@embetronicx-VirtualBox:~/Desktop/LDD$ dmesg
[12177.714853] EmbeTronicX USB Driver 1-2:1.0: USB Driver Disconnected
[12178.036702] usb 1-2: USB disconnect, device number 12
```

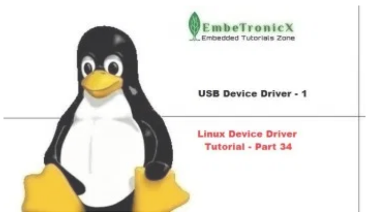
- Unload the driver using `sudo rmmod usb_driver` and check the `dmesg`

```
etx@embetronicx-VirtualBox:~/Desktop/LDD$ dmesg
[12351.066544] usbcore: deregistering interface driver EmbeTronicX USB Driver
```

- Now the driver is removed from the USB subsystem.

That’s all now. In our next tutorial, we will discuss other functionalities of the USB driver.

Related



Linux Device Driver Tutorial Part 33 – USB Device Driver Basics 1
In "Device Drivers"

Linux Device Driver Part 1 : Introduction
In "Device Drivers"

Linux Device Driver Tutorial Part 32 – Misc Device Driver
In "Device Drivers"

Subscribe

Connect with | Login

Be the First to Comment!

B I U S {} [+]

This site uses Akismet to reduce spam. Learn how your comment data is processed.

0 COMMENTS

report this ad