LINUX JOURNAL

Writing a Simple USB Driver

Mar 31, 2004 By Greg Kroah-Hartman (/user/800887)

in

Give your Linux box a multicolored light you can see from across the room, and learn how to write a simple driver for the next piece of hardware you want to hook up.

(/issue/120)

From Issue #120
April 2004 (/issue/120)

A Kernel Driver

Armed with our new-found information, we set off to whip up a quick kernel driver. It should be a USB driver, but what kind of interface to user space should we use? A block device does not make sense, as this device does not need to store filesystem data, but a character device would work. If we use a character device driver, however, a major and minor number needs to be reserved for it. And how many minor numbers would we need for this driver? What if someone wanted to plug 100 different USB lamp devices in to this system? To anticipate this, we would need to reserve at least 100 minor numbers, which would be a total waste if all anyone ever used was one device at a time. If we make a character driver, we also would need to invent some way to tell the driver to turn on and off the different colors individually. Traditionally, that could be done using different ioctl commands on the character driver, but we know much better than ever to create a new ioctl command in the kernel.

As all USB devices show up in their own directory in the sysfs tree, so why not use sysfs and create three files in the USB device directory, blue, red and green? This would allow any user-space program, be it a C program or a shell script, to change the colors on our LED device. This also would keep us from having to write a character driver and beg for a chunk of minor numbers for our device.

To start out our USB driver, we need to provide the USB subsystem with five things:

- A pointer to the module owner of this driver: this allows the USB core to control the module reference count of the driver properly.
- The name of the USB driver.
- A list of the USB IDs this driver should provide: this table is used by the USB core to determine which driver should be matched up to which device; the hot-plug user-space scripts use it to load that driver automatically when a device is plugged in to the system.
- A probe() function called by the USB core when a device is found that matches the USB ID table.
- A disconnect() function called when the device is removed from the system.

The driver retrieves this information with the following bit of code:

```
static struct usb_driver led_driver = {
        .owner =        THIS_MODULE,
        .name =         "usbled",
        .probe =        led_probe,
        .disconnect =   led_disconnect,
```

```
        .id_table =     id_table,
};
```

The id_table variable is defined as:

```
static struct usb_device_id id_table [] = {
        { USB_DEVICE(VENDOR_ID, PRODUCT_ID) },
        { },
};
MODULE_DEVICE_TABLE (usb, id_table);
```

The led_probe() and led_disconnect() functions are described later.

When the driver module is loaded, this led_driver structure must be registered with the USB core. This is accomplished with a single call to the usb_register() function:

```
retval = usb_register(&led_driver);
if (retval)
        err("usb_register failed. "
            "Error number %d", retval);
```

Likewise, when the driver is unloaded from the system, it must unregister itself from the USB core:

```
usb_deregister(&led_driver);
```

The led_probe() function is called when the USB core has found our USB lamp device. All it needs to do is initialize the device and create the three sysfs files, in the proper location. This is done with the following code:

```
/* Initialize our local device structure */
dev = kmalloc(sizeof(struct usb_led), GFP_KERNEL);
memset (dev, 0x00, sizeof (*dev));

dev->udev = usb_get_dev(udev);
usb_set_intfdata (interface, dev);

/* Create our three sysfs files in the USB
* device directory */
device_create_file(&interface->dev, &dev_attr_blue);
device_create_file(&interface->dev, &dev_attr_red);
device_create_file(&interface->dev, &dev_attr_green);

dev_info(&interface->dev,
    "USB LED device now attached\n");
return 0;
```

The led_disconnect() function is equally as simple, as we need only to free our allocated memory and remove the sysfs files:

```
dev = usb_get_intfdata (interface);
usb_set_intfdata (interface, NULL);

device_remove_file(&interface->dev, &dev_attr_blue);
device_remove_file(&interface->dev, &dev_attr_red);
device_remove_file(&interface->dev, &dev_attr_green);

usb_put_dev(dev->udev);
kfree(dev);

dev_info(&interface->dev,
```

```
        "USB LED now disconnected\n");
```

When the sysfs files are read from, we want to show the current value of that LED; when it is written to, we want to set that specific LED. To do this, the following macro creates two functions for each color LED and declares a sysfs device attribute file:

```
#define show_set(value)                                 \
static ssize_t                                          \
show_##value(struct device *dev, char *buf)             \
{                                                       \
   struct usb_interface *intf =                         \
      to_usb_interface(dev);                            \
   struct usb_led *led = usb_get_intfdata(intf);        \
                                                        \
   return sprintf(buf, "%d\n", led->value);             \
}                                                       \
                                                        \
static ssize_t                                          \
set_##value(struct device *dev, const char *buf,        \
            size_t count)                               \
{                                                       \
   struct usb_interface *intf =                         \
      to_usb_interface(dev);                            \
   struct usb_led *led = usb_get_intfdata(intf);        \
   int temp = simple_strtoul(buf, NULL, 10);            \
                                                        \
   led->value = temp;                                   \
   change_color(led);                                   \
   return count;                                        \
}                                                       \

static DEVICE_ATTR(value, S_IWUGO | S_IRUGO,
                   show_##value, set_##value);
show_set(blue);
show_set(red);
show_set(green);
```

This creates six functions, show_blue(), set_blue(), show_red(), set_red(), show_green() and set_green(); and three attribute structures, dev_attr_blue, dev_attr_red and dev_attr_green. Due to the simple nature of the sysfs file callbacks and the fact that we need to do the same thing for every different value (blue, red and green), a macro was used to reduce typing. This is a common occurrence for sysfs file functions; an example of this in the kernel source tree is the I2C chip drivers in drivers/i2c/chips.

So, to enable the red LED, a user writes a 1 to the red file in sysfs, which calls the set_red() function in the driver, which calls the change_color() function. The change_color() function looks like:

```
#define BLUE    0x04
#define RED     0x02
#define GREEN   0x01
   buffer = kmalloc(8, GFP_KERNEL);

   color = 0x07;
   if (led->blue)
      color &= ~(BLUE);
   if (led->red)
      color &= ~(RED);
   if (led->green)
      color &= ~(GREEN);
   retval =
      usb_control_msg(led->udev,
```

```
                        usb_sndctrlpipe(led->udev, 0),
                        0x12,
                        0xc8,
                        (0x02 * 0x100) + 0x0a,
                        (0x00 * 0x100) + color,
                        buffer,
                        8,
                        2 * HZ);
    kfree(buffer);
```

Comments

Comment viewing options

Threaded list - expanded ▼ | Date - newest first ▼ | 50 comments per page ▼ | Save settings

Select your preferred way to display the comments and click "Save settings" to activate your changes.

How To Get Port Values and the Resulting LED Patterns ??? (/article/7353#comment-357354)

Submitted by Black Spider (http://WwW.B2S.WeBeGe.CoM) (not verified) on Fri, 10/22/2010 - 13:56.

Hi
Thanks For this guide but I want to know how to get Port Values and the Resulting LED Patterns without documents of the device ???

Souce (/article/7353#comment-356462)

Submitted by techieguy22 (/users/techieguy22) on Fri, 10/01/2010 - 03:13.

(/users/techieguy22)

I was actually trying to write a kernel driver for usb and was hooked up reading this article only to find out it's been around for 6 years! Even so I've learned alot from this article. This is a great tutorial to make usb drivers (http://www.driversupdate.org/usb-drivers.htm), however I'm quite stucked on the process of compiling the source.

Delphi Hid get path (/article/7353#comment-351828)

Submitted by Anonymous on Wed, 05/12/2010 - 22:27.

The first time I use SetupDiGetDeviceInterfaceDetailA
It returns GetLastError = ERROR_INSUFFICIENT_BUFFER or 78
and a byesreturned is a good expected number
DevData (is the record below)

TSPDevInfoData = packed record
Size: DWORD;
ClassGuid: TGUID;
DevInst: DWORD;
Reserved: DWord;
end;
with TSPDevInfoData.ClassGuid = the Registry USB controller GUID

When I call SetupDiGetDeviceInterfaceDetailA a second time it hates me real bad. I'm not sure how to convert BytesReturned into my DevData.size or TSPDeviceInterfaceDetailDataA.size correctly.

I have checked my Structures through and through