Kernel Korner

How to Write a Linux USB Device Driver

Greg shares his USB driver skeleton and shows how it can be customized for your specific device.

by Greg Kroah-Hartman

The Linux USB subsystem has grown from supporting only two different types of devices in the 2.2.7 kernel (mice and keyboards), to over 20 different types of devices in the 2.4 kernel. Linux currently supports almost all USB class devices (standard types of devices like keyboards, mice, modems, printers and speakers) and an ever-growing number of vendor-specific devices (such as USB to serial converters, digital cameras, Ethernet devices and MP3 players). For a full list of the different USB devices currently supported, see Resources.

The remaining kinds of USB devices that do not have support on Linux are almost all vendor-specific devices. Each vendor decides to implement a custom protocol to talk to their device, so a custom driver usually needs to be created. Some vendors are open with their USB protocols and help with the creation of Linux drivers, while others do not publish them, and developers are forced to reverse-engineer. See Resources for some links to handy reverse-engineering tools.

Because each different protocol causes a new driver to be created, I have written a generic USB driver skeleton, modeled after the pci-skeleton.c file in the kernel source tree upon which many PCI network drivers have been based. This USB skeleton can be found at drivers/usb/usb-skeleton.c in the kernel source tree. In this article I will walk through the basics of the skeleton driver, explaining the different pieces and what needs to be done to customize it to your specific device.

If you are going to write a Linux USB driver, please become familiar with the USB protocol specification. It can be found, along with many other useful documents, at the USB home page (see Resources). An excellent introduction to the Linux USB subsystem can be found at the USB Working Devices List (see Resources). It explains how the Linux USB subsystem is structured and introduces the reader to the concept of USB urbs, which are essential to USB drivers.

The first thing a Linux USB driver needs to do is register itself with the Linux USB subsystem, giving it 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 in the usb_driver structure. The skeleton driver declares a usb_driver as:

```
static struct usb_driver skel_driver = {
     name:        "skeleton",
     probe:       skel_probe,
     disconnect:  skel_disconnect,
     fops:        &skel_fops,
     minor:       USB_SKEL_MINOR_BASE,
     id_table:    skel_table,
};
```

The variable name is a string that describes the driver. It is used in informational messages printed to the system log. The probe and disconnect function pointers are called when a device that matches the information provided in the id_table variable is either seen or removed.

The fops and minor variables are optional. Most USB drivers hook into another kernel subsystem, such as the SCSI, network or TTY subsystem. These types of drivers register themselves with the other kernel subsystem, and any user-space interactions are provided through that interface. But for drivers that do not have a matching kernel subsystem, such as MP3 players or scanners, a method of interacting with user space is needed. The USB subsystem provides a way to register a minor device number and a set of file_operations function pointers that enable this user-space interaction. The skeleton driver needs this kind of interface, so it provides a minor starting number and a pointer to its file_operations functions.

The USB driver is then registered with a call to usb_register, usually in the driver's init function, as shown in Listing 1.

Listing 1. Registering the USB Driver

When the driver is unloaded from the system, it needs to unregister itself with the USB subsystem. This is done with the usb_unregister function:

```
static void __exit usb_skel_exit(void)
{
   /* deregister this driver with the USB subsystem */
   usb_deregister(&skel_driver);
}
module_exit(usb_skel_exit);
```

To enable the linux-hotplug system to load the driver automatically when the device is plugged in, you need to create a MODULE_DEVICE_TABLE. The following code tells the hotplug scripts that this module supports a single device with a specific vendor and product ID:

```
/* table of devices that work with this driver */
static struct usb_device_id skel_table [] = {
    { USB_DEVICE(USB_SKEL_VENDOR_ID,
      USB_SKEL_PRODUCT_ID) },
    { }                      /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, skel_table);
```

There are other macros that can be used in describing a usb_device_id for drivers that support a whole class of USB drivers. See usb.h for more information on this.

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 usb_device structure, interface number and the interface ID are passed to the function:

```
static void * skel_probe(struct usb_device *dev,
unsigned int ifnum, const struct usb_device_id *id)
```

The driver now needs to verify that this device is actually one that it can accept. If not, or if any error occurs during initialization, a NULL value is returned from the probe function. Otherwise a pointer to a private data structure containing the driver's state for this device is returned. That pointer is stored in the usb_device structure, and all callbacks to the driver pass that pointer.

In the skeleton driver, we determine what end points are marked as bulk-in and bulk-out. We create buffers to hold the data that will be sent and received from the device, and a USB urb to write data to the device is initialized. Also, we register the device with the devfs subsystem, allowing users of devfs to access our device. That registration looks like the following:

```
/* initialize the devfs node for this device
   and register it */
sprintf(name, "skel%d", skel->minor);
```

```
skel->devfs = devfs_register
              (usb_devfs_handle, name,
               DEVFS_FL_DEFAULT, USB_MAJOR,
               USB_SKEL_MINOR_BASE + skel->minor,
               S_IFCHR | S_IRUSR | S_IWUSR |
               S_IRGRP | S_IWGRP | S_IROTH,
               &skel_fops, NULL);
```

If the devfs_register function fails, we do not care, as the devfs subsystem will report this to the user.

Conversely, when the device is removed from the USB bus, the disconnect function is called with the device pointer. The driver needs to clean any private data that has been allocated at this time and to shut down any pending urbs that are in the USB system. The driver also unregisters itself from the devfs subsystem with the call:

```
/* remove our devfs node */
devfs_unregister(skel->devfs);
```

Now that the device is plugged into the system and the driver is bound to the device, any of the functions in the file_operations structure that were passed to the USB subsystem will be called from a user program trying to talk to the device. The first function called will be open, as the program tries to open the device for I/O. Within the skeleton driver's open function we increment the driver's usage count if it is a module with a call to MODULE_INC_USE_COUNT. With this macro call, if the driver is compiled as a module, the driver cannot be unloaded until a corresponding MODULE_DEC_USE_COUNT macro is called. We also increment our private usage count and save off a pointer to our internal structure in the file structure. This is done so that future calls to file operations will enable the driver to determine which device the user is addressing. All of this is done with the following code:

```
/* increment our usage count for the module */
MOD_INC_USE_COUNT;
++skel->open_count;
/* save our object in the file's private structure */
file->private_data = skel;
```

After the open function is called, the read and write functions are called to receive and send data to the device. In the skel_write function, we receive a pointer to some data that the user wants to send to the device and the size of the data. The function determines how much data it can send to the device based on the size of the write urb it has created (this size depends on the size of the bulk out end point that the device has). Then it copies the data from user space to kernel space, points the urb to the data and submits the urb to the USB subsystem (see Listing 2).

Listing 2. The skel_write Function

When the write urb is filled up with the proper information using the FILL_BULK_URB function, we point the urb's completion callback to call our own skel_write_bulk_callback function. This function is called when the urb is finished by the USB subsystem. The callback function is called in interrupt context, so caution must be taken not to do very much processing at that time. Our implementation of skel_write_bulk_callback merely reports if the urb was completed successfully or not and then returns.

The read function works a bit differently from the write function in that we do not use an urb to transfer data from the device to the driver. Instead we call the usb_bulk_msg function, which can be used to send or receive data from a device without having to create urbs and handle urb completion callback functions. We call the usb_bulk_msg function, giving it a buffer into which to place any data received from the device and a timeout value. If the timeout period expires without receiving any data from the device, the function will fail and return an error message (see Listing 3).

Listing 3. The usb_bulk_msg Function

The usb_bulk_msg function can be very useful for doing single reads or writes to a device; however, if you need to read or write constantly to a device, it is recommended to set up your own urbs and submit them to the USB subsystem.

When the user program releases the file handle that it has been using to talk to the device, the release function in the driver is called. In this function we decrement the module usage count with a call to MOD_DEC_USE_COUNT (to match our previous call to MOD_INC_USE_COUNT). We also determine if there are any other programs that are currently talking to the device (a device may be opened by more than one program at one time). If this is the last user of the device, then we shut down any possible pending writes that might be currently occurring. This is all done with:

```
/* decrement our usage count for the device */
--skel->open_count;
if (skel->open_count <= 0) {
   /* shutdown any bulk writes that might be
      going on */
   usb_unlink_urb (skel->write_urb);
   skel->open_count = 0;
}
/* decrement our usage count for the module */
MOD_DEC_USE_COUNT;
```

One of the more difficult problems that USB drivers must be able to handle smoothly is the fact that the USB device may be removed from the system at any point in time, even if a program is currently talking to it. It needs to be able to shut down any current reads and writes and notify the user-space programs that the device is no longer there (see Listing 4).

Listing 4. The skel_disconnect Function

If a program currently has an open handle to the device, we only null the usb_device structure in our local structure, as it has now gone away. For every read, write, release and other functions that expect a device to be present, the driver first checks to see if this usb_device structure is still present. If not, it releases that the device has disappeared, and a -ENODEV error is returned to the user-space program. When the release function is eventually called, it determines if there is no usb_device structure and if not, it does the cleanup that the skel_disconnect function normally does if there are no open files on the device (see Listing 5).

Listing 5. Cleanup

This usb-skeleton driver does not have any examples of interrupt or isochronous data being sent to or from the device. Interrupt data is sent almost exactly as bulk data is, with a few minor exceptions. Isochronous data works differently with continuous streams of data being sent to or from the device. The audio and video camera drivers are very good examples of drivers that handle isochronous data and will be useful if you also need to do this.

Writing Linux USB device drivers is not a difficult task as the usb-skeleton driver shows. This driver, combined with the other current USB drivers, should provide enough examples to help a beginning author create a working driver in a minimal amount of time. The linux-usb-devel mailing list archives also contain a lot of helpful information.

Resources

Greg Kroah-Hartman *is one of the Linux kernel USB developers. His free software is being used by more people than any closed-source projects he has ever been paid to develop.*

Magazine Table of Contents