# Chapter 11. Universal Serial Bus

## In This Chapter



U*niversal serial bus* (USB) is the de facto external bus in today's computers. USB, with its support for hotplugging, generic class drivers, and versatile data-transfer modes, is the usual route in the consumer electronics space to bring a diverse spectrum of technologies to computer systems. Its sweeping popularity and the accompanying economics of volume have played a part in fueling the adoption and acceptance of computer peripheral technologies around the world.

## USB Architecture

USB is a master-slave protocol where a host controller communicates with client devices. Figure 11.1 shows USB in the PC environment. The USB host controller is part of the South Bridge chipset and communicates with the processor over the PCI bus.

**Figure 11.1. USB in the PC environment.**

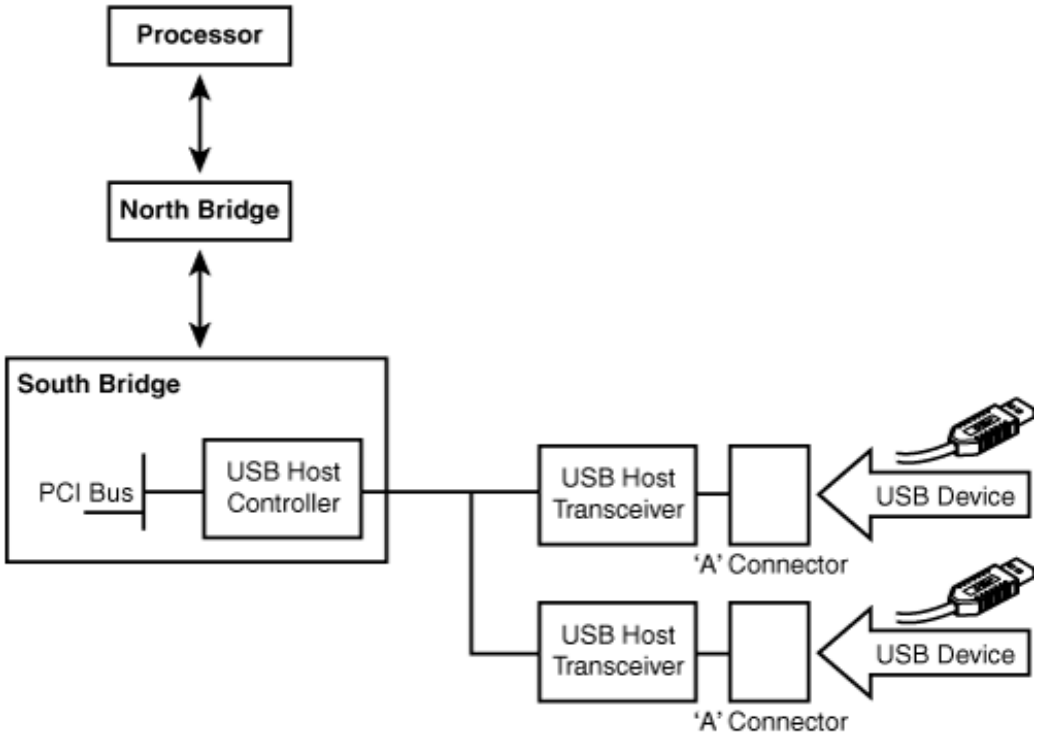



Figure 11.2 illustrates USB on an embedded device. The SoC in the figure has built-in USB controller silicon that supports four buses and three modes of operation:

- Bus 1 runs in host mode and is wired to an A-type receptacle via a USB transceiver (see the sidebar "USB Receptacles and Transceivers"). You can connect a USB pen drive or a keyboard to this port.

- Bus 2 also functions in host mode but the associated transceiver is connected to an internal USB device rather than to a receptacle. Examples of internal USB devices are biometric scanners, cryptographic engines, printers, *Disk-On-Chips* (DOCs), touch controllers, and telemetry cards.

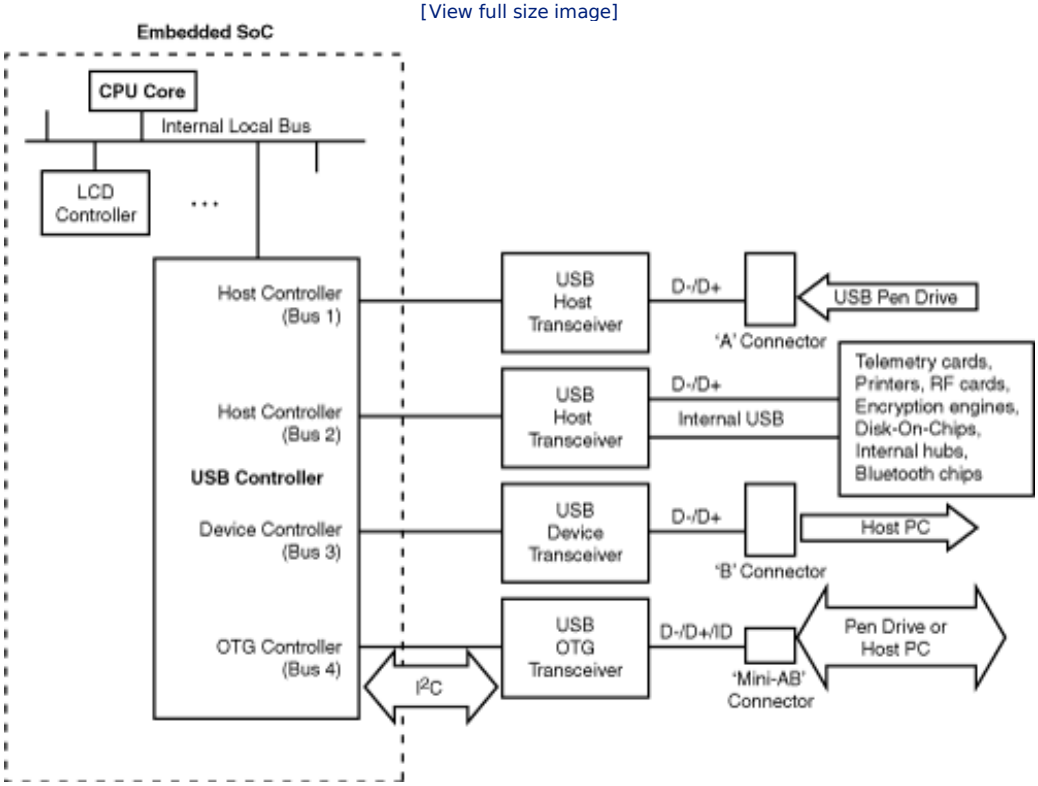
- Bus 3 runs in device mode and is wired to a B-type receptacle through a transceiver. The B-type receptacle connects to a host computer via a B-to-A cable. In this mode, the embedded device functions as, for example, a USB pen drive, and exports a storage partition to the outside world. Embedded devices such as MP3 players and cell phones are more likely than PC systems to be at the device side of USB, so many embedded SoCs support a USB device controller in addition to a host controller.

- Bus 4 is driven by an *On-The-Go* (OTG) controller. You can use this port, for example, to either connect a pen drive to your system or to turn your system into a pen drive and connect it to a host. Unlike buses 1

to 3, bus 4 uses an intelligent transceiver that exchanges control information with the processor over I $^2$C. The transceiver is wired to a Mini-AB OTG receptacle. If two embedded devices support OTG, they can directly communicate without the intervention of a host computer.

**Figure 11.2. USB on an embedded system.**

[View full size image]




Most of this chapter is written from the perspective of a system residing at the host-side of USB. We briefly look at the device function in the section "Gadget Drivers." Mainstream *host controller drivers* (HCDs) are already available, so in this chapter we further confine ourselves to drivers for USB devices (also called *client drivers*).

USB Receptacles and Transceivers

USB hosts use four-pin A-type rectangular receptacles, whereas USB devices connect via four-pin B-type square receptacles. In both cases, the four pins are differential data signals D+ and D-, a voltage line VBUS, and ground. VBUS is used to supply power from USB hosts to USB devices. VBUS is thus pulled high on an A connector but receives power on a B connector. USB OTG controllers connect to five-pin Mini-AB rectangular receptacles having a smaller form factor. Four of the Mini-AB pins are identical to what we discussed previously; the fifth is an ID pin used to detect whether the connected peripheral is a host or a device.

The same transceiver chip (such as TUSB1105 from Texas Instruments) can be used on USB hosts and devices. You may thus choose to use the same transceiver part on buses 1 through 3 in Figure 11.2. OTG requires a special-purpose transceiver chip (such as ISP1301 from Philips Semiconductors), however.

## Bus Speeds

USB supports three operational speeds. The original USB 1.0 specification supports 1.5MBps, referred to as low-speed USB. USB 1.1, the next version of the specification, handles 12MBps, called full-speed USB. The current level of the specification is USB 2.0, which supports 480MBps, or high-speed USB. USB 2.0 is backward-compatible with the earlier versions of the specification. Peripherals such as USB keyboards and mice are examples of low-speed devices, and USB storage drives are examples of full-speed and high-speed devices. Today's PC systems are USB 2.0-compliant and allow all three target speeds, but some embedded controllers adhere to USB 1.1 and support only full-speed and low-speed modes of operation.

## Host Controllers

USB host controllers conform to one of a few standards:

- **Universal Host Controller Interface (UHCI):** The UHCI specification was initiated by Intel, so your PC is likely to have this controller if it's Intel-based.

- **Open Host Controller Interface (OHCI):** The OHCI specification originated from companies such as Compaq and Microsoft. An OHCI-compatible controller has more intelligence built in to hardware than UHCI, so an OHCI HCD is relatively simpler than a UHCI HCD.

- **Enhanced Host Controller Interface (EHCI):** This is the host controller that supports high-speed USB 2.0 devices. EHCI controllers usually have either a UHCI or OHCI companion controller to handle slower devices.

- **USB OTG controllers:** They are getting increasingly popular in embedded microcontrollers. With OTG support, each communicating end can act as a *dual-role device* (DRD). By initiating a dialog using the *Host Negotiation Protocol* (HNP), a DRD can switch itself to host mode or device mode based on the desired functionality.

In addition to these mainstream USB host controllers, Linux supports a few more controllers. An example is the HCD for the ISP116x chip.

Host controllers have a built-in hardware component called the *root hub.* The root hub is a virtual hub that sources USB ports. The ports, in turn, can connect to external or internal physical hubs and source more ports,

yielding a tree topology.

## Transfer Types

Data exchange with a USB device can be one of four types:

- Control transfers, used to carry configuration and control information
- Bulk transfers that ferry large quantities of time-insensitive data
- Interrupt transfers that exchange small quantities of time-sensitive data
- Isochronous transfers for real-time data at predictable bit rates

A USB storage drive, for example, uses control transfers to issue disk access commands and bulk transfers to exchange data. A keyboard uses interrupt transfers to carry key strokes within predictable delays. A device that needs to stream audio data in real time uses isochronous transfers. The responsibilities of the four transfer types for USB Bluetooth devices are discussed in the section "Device Example: USB Adapter" in Chapter 16, "Linux Without Wires."

## Addressing

Each addressable unit in a USB device is called an *endpoint.* The address assigned to an endpoint is called an *endpoint address.* Each endpoint address has an associated data transfer type. If an endpoint is responsible for bulk data transfer, for example, it's called a *bulk endpoint.* Endpoint address 0 is used exclusively for device configuration. A control pipe is attached to this endpoint for device enumeration (see the section "Enumeration").

An endpoint can be associated with upstream or downstream data transfer. Data arriving upstream from a device is called an ***IN*** transfer, whereas data flowing downstream to a device is an ***OUT*** transfer. ***IN*** and ***OUT*** transfers own separate address spaces. So, you can have a bulk ***IN*** endpoint and a bulk ***OUT*** endpoint answering to the same address.

USB resembles $I^2C$ on some counts and PCI on others as summarized in Table 11.1. USB's device addressing is similar to $I^2C$, while it supports hotplugging like PCI. USB device addresses, like standard $I^2C$, do not consume a portion of the CPU's address space. Rather, they reside in a private space ranging from 1 to 127.

**Table 11.1. USB's Similarities with $I^2C$ and PCI**

USB's similarities with $I^2C$:

- USB and $I^2C$ are master-slave protocols.
- Device addresses reside in a private 7-bit space.
- Device-resident memory is not mapped to the CPU's memory or I/O space, so it does not consume CPU resources.

USB's similarities with PCI:

- Devices can be hotplugged.
- Device driver architecture resembles PCI drivers. Both classes of drivers own ***probe()/disconnect()***[1] methods and ID tables identifying the devices they support.
- Supports high speeds. Slower than PCI, however. See Table 10.1 in Chapter 10, "Peripheral Component Interconnect," for the speeds supported by different members of the PCI family.
- USB host controllers, like PCI controllers, usually have built-in DMA engines that can master the bus.
- Supports multifunction devices. USB supports interface descriptors per function. Each PCI device function has its own device ID and configuration space.

[1] ***disconnect()*** is called ***remove()*** in PCI parlance.

## Linux-USB Subsystem

Look at Figure 11.3 to understand the architecture of the Linux-USB subsystem. The constituent pieces of the subsystem are as follows:

- The USB core. Like the core layers of driver subsystems that you saw in previous chapters, the USB core is a code base consisting of routines and structures available to HCDs and client drivers. The core also provides a level of indirection that renders client drivers independent of host controllers.

- HCDs that drive different host controllers.

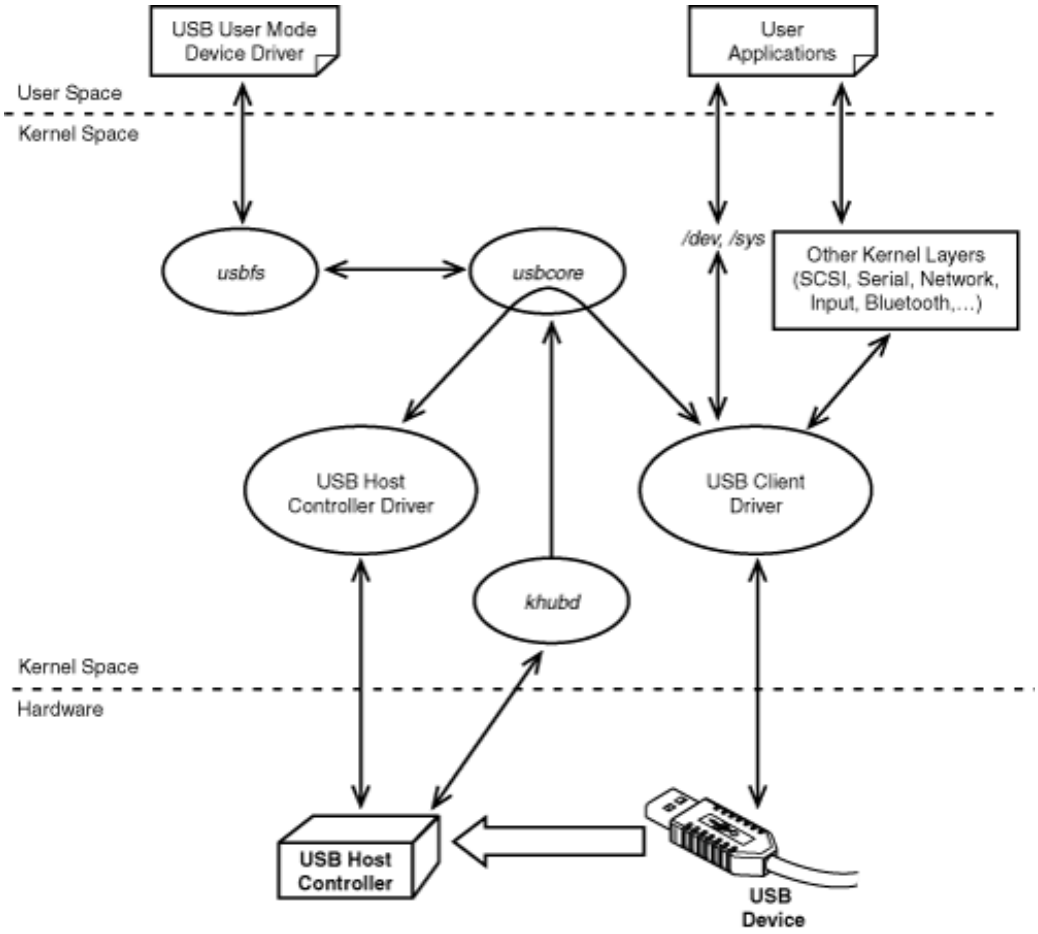
- A hub driver for the root hub (and physical hubs) and a helper kernel thread *khubd* that monitors all ports connected to the hub. Detecting port status changes and configuring hotplugged devices is time-consuming and is best accomplished using a helper thread for reasons you learned in Chapter 3, "Kernel Facilities." The khubd thread is asleep by default. The hub driver wakes khubd whenever it detects a status change on a USB port connected to it.

- Device drivers for USB client devices.

- The USB filesystem *usbfs* that lets you drive USB devices from user space. We discuss user mode USB drivers in Chapter 19, "Drivers in User Space."

**Figure 11.3. The Linux-USB subsystem.**

[View full size image]




For end-to-end operation, the USB subsystem calls on various other kernel layers for assistance. To support USB mass storage devices, for example, the USB subsystem works in tandem with SCSI drivers, as shown in Figure 11.3. To drive USB-Bluetooth keyboards, the stakeholders are fourfold: the USB subsystem, the Bluetooth layer, the input subsystem, and the tty layer.

## Driver Data Structures

When you write a USB client driver, you have to work with several data structures. Let's look at the important ones.

### The *usb_device* Structure

Each device driver subsystem relies on a special-purpose data structure to internally represent a device. The *usb_device* structure is to the USB subsystem, what *pci_dev* is to the PCI layer, and what *net_device* is to the network driver layer. *usb_device* is defined in *include/linux/usb.h* as follows:

```
struct usb_device {
  /* ... */
  enum usb_device_state state; /* Configured, Not Attached, etc */
  enum usb_device_speed speed; /* High/full/low (or error) */
  /* ... */
  struct usb_device *parent;   /* Our hub, unless we're the root */
  /* ... */
  struct usb_device_descriptor descriptor; /* Descriptor */
  struct usb_host_config *config;          /* All of the configs */
  struct usb_host_config *actconfig;       /* The active config */
  /* ... */
  int maxchild;                                /* No: of ports if hub */
  struct usb_device *children[USB_MAXCHILDREN]; /* Child devices */
  /* ... */
};
```

We use this structure when we develop an example driver for a USB telemetry card later.

### USB Request Blocks

*USB Request Block* (URB) is the centerpiece of the USB data transfer mechanism. A URB is to the USB stack, what an *sk_buff* (discussed in Chapter 15, "Network Interface Cards") is to the networking stack.

Let's take a peek inside a URB. The following definition is from *include/linux/usb.h*, omitting fields not of particular interest to device drivers:

Code View:

```
struct urb
{
  struct kref kref;              /* Reference count of the URB */
  /* ... */
  struct usb_device *dev;        /* (in) pointer to associated
                                    device */
  unsigned int pipe;             /* (in) pipe information */
  int status;                    /* (return) non-ISO status */
  unsigned int transfer_flags;   /* (in) URB_SHORT_NOT_OK | ...*/
  void *transfer_buffer;         /* (in) associated data buffer */
  dma_addr_t transfer_dma;       /* (in) dma addr for
                                    transfer_buffer */
  int transfer_buffer_length;    /* (in) data buffer length */
```

```
  /* ... */
  unsigned char *setup_packet;   /* (in) setup packet */
  /* ... */
  int interval;                  /* (modify) transfer interval
                                    (INT/ISO) */
  /* ... */
  void *context;                 /* (in) context for completion */
  usb_complete_t complete;       /* (in) completion routine */
  /* ... */
};
```

There are three steps to using a URB: create, populate, and submit. To create a URB, use ***usb_alloc_urb()***. This function allocates and zeros-out URB memory, initializes a kobject associated with the URB, and initializes a spinlock to protect the URB.

To populate a URB, use the following helper routines offered by the USB core:

```
void usb_fill_[control|int|bulk]_urb(
       struct urb *urb,               /* URB pointer */
       struct usb_device *usb_dev,    /* USB device structure */
       unsigned int pipe,             /* Pipe encoding */
       unsigned char *setup_packet,   /* For Control URBs only! */
       void *transfer_buffer,         /* Buffer for I/O */
       int buffer_length,             /* I/O buffer length */
       usb_complete_t completion_fn,  /* Callback routine */
       void *context,                 /* For use by completion_fn */
       int interval);                 /* For Interrupt URBs only! */
```

The semantics of the previous routines will get clearer when we develop the example driver later on. These helper routines are available to control, interrupt, and bulk URBs but not to isochronous ones.

To submit a URB for data transfer, use ***usb_submit_urb()***. URB submission is asynchronous. The ***usb_fill_[control|int|bulk]_urb()*** functions listed previously take the address of a callback function as argument. The callback routine executes after the URB submission completes and accomplishes things such as checking submission status and freeing the data-transfer buffer.

The USB core also offers wrapper interfaces that provide a façade of synchronous URB submission:

```
int usb_[control|interrupt|bulk]_msg(struct usb_device *usb_dev,
                                     unsigned int pipe, ...);
```

***usb_bulk_msg ()***, for example, builds a bulk URB, submits it, and blocks until the operation completes. You don't have to supply a callback function because a generic completion routine serves that purpose. You don't need to explicitly create and populate the URB either, because ***usb_bulk_msg()*** does that for you at no additional cost. We will use this interface in our example driver.

***usb_free_urb()*** is used to free a reference to a completed URB, whereas ***usb_unlink_urb()*** cancels a pending URB operation.

As mentioned in the section "Sysfs, Kobjects, and Device Classes" in Chapter 4, "Laying the Groundwork," a URB contains a kref object to track references to it. ***usb_submit_urb()*** increments the reference count using ***kref_get()***. ***usb_free_urb()*** decrements the reference count using ***kref_put()*** and performs the free

operation only if there are no remaining references.

A URB is associated with an abstraction called a *pipe*, which we discuss next.

## Pipes

A pipe is an integer encoding of a combination of the following:

- The endpoint address
- The direction of data transfer (***IN*** or ***OUT***)
- The type of data transfer (control, interrupt, bulk, or isochronous)

A pipe is the address element of each USB data transfer and is an important field in the URB structure. To help populate this field, the USB core provides the following helper macros:

```
usb_[rcv|snd][ctrl|int|bulk|isoc]pipe(struct usb_device *usb_dev,
                                      __u8 endpointAddress);
```

where ***usb_dev*** is a pointer to the associated ***usb_device*** structure, and ***endpointAddress*** is the assigned endpoint address between 1 and 127. To create a bulk pipe in the ***OUT*** direction, for example, call ***usb_sndbulkpipe()***. For a control pipe in the ***IN*** direction, invoke ***usb_rcvctrlpipe()***.

While referring to a URB, it's often qualified by the transfer type of the associated pipe. If a URB is attached to a bulk pipe, for example, it's called a *bulk URB*.

## Descriptor Structures

The USB specification defines a series of *descriptors* to hold information about a device. The Linux-USB core defines data structures corresponding to each descriptor. Descriptors are of four types:

- *Device descriptors* contain general information such as the product ID and vendor ID of the device. ***usb_device_descriptor*** is the structure corresponding to device descriptors.
- *Configuration descriptors* are used to describe different configuration modes such as bus-powered and self-powered operation. ***usb_config_descriptor*** is the data structure associated with configuration descriptors.
- *Interface descriptors* allow USB devices to support multiple functions. ***usb_interface_descriptor*** defines interface descriptors.
- *Endpoint descriptors* carry information associated with the final endpoints of a device. ***usb_endpoint_descriptor*** is the structure in question.

These descriptor formats are defined in Chapter 9 of the USB specification, whereas the matching structures are

defined in *include/linux/usb/ch9.h.* Listing 11.1 shows the hierarchical topology of the descriptors and prints all endpoint addresses associated with a USB device. To this end, it traverses the tree consisting of the four types of descriptors described previously. The following is the output generated by Listing 11.1 for a USB CD drive:

```
Endpoint Address = 1
Endpoint Address = 82
Endpoint Address = 83
```

The first address belongs to a bulk ***IN*** endpoint, the second address is owned by a bulk ***OUT*** endpoint, and the third addresses an interrupt ***IN*** endpoint.

There are more data structures associated with USB client drivers, such as ***usb_device_id***, ***usb_driver***, and ***usb_class_driver***. We will meet them when we do hands-on development in the section "Device Example: Telemetry Card."

**Listing 11.1. Print All USB Endpoint Addresses on a Device**

[View full size image]

```
/* ... */
/* USB device */
struct usb_device *udevice;
/* ... */
struct usb_device_descriptor u_d_desc = udevice->descriptor;

/* Device's active configuration */
struct usb_host_config *uconfig;
struct usb_config_desriptor u_c_desc;

/* Interfaces in the active configuration */
struct usb_interface *uinterface;

/* Alternate Setting for this interface */
struct usb_host_interface *ualtsetting;
struct usb_interface_descriptor u_i_desc;

/* Endpoints for this altsetting */
struct usb_host_endpoint *uendpoint;
struct usb_endpoint_descriptor u_e_desc;

uconfig = udevice->actconfig;

u_c_desc = uconfig->desc;

for (i = 0; i < u_c_desc.bNumInterfaces;
     i++) {
  uinterface = udevice->actconfig->interface[i];
  for (j = 0; j < uinterface->num_altsetting; j++) {
ualtsetting = &uinterface->altsetting[j];
u_i_desc = ualtsetting->desc;
  for (k = 0; k < u_i_desc.bNumEndpoints; k++) {
    uendpoint = &ualtsetting->endpoint[k];
    u_e_desc = uendpoint->desc;
    printk ("Endpoint Address = %d\n",
            u_e_desc.bEndpointAddress\n");
  }
}
/* ... */
```

## Enumeration

The life of a hotplugged USB device starts with a process called *enumeration* by which the host learns about the device's capabilities and configures it. The hub driver is the component in the Linux-USB subsystem responsible for enumeration. Let's look at the sequence of steps that achieve device enumeration when you plug in a USB pen drive into a host computer:

1. The root hub reports a change in the port's current due to the device attachment. The hub driver detects this status change, called a ***`USB_PORT_STAT_C_CONNECTION`*** in Linux-USB terminology, and awakens khubd.

2. Khubd deciphers the identity of the USB port subjected to the status change. In this case, it's the port where you plugged in the pen drive.

3. Next, khubd chooses a device address between 1 and 127 and assigns it to the pen drive's bulk endpoint using a control URB attached to endpoint ***`0`***.

4. Khubd uses the above control URB attached to endpoint ***`0`*** to obtain the device descriptor from the pen drive. It then requests the device's configuration descriptors and selects a suitable one. In the case of the pen drive, only a single configuration descriptor is on offer.

5. Khubd requests the USB core to bind a matching client driver to the inserted device.

When enumeration is complete and the device is bound to a driver, khubd invokes the associated client driver's ***`probe()`*** method. In this case, khubd calls ***`storage_probe()`*** defined in *drivers/usb/storage/usb.c.* From this point on, the mass storage driver is responsible for normal device operation.

## Device Example: Telemetry Card

Now that you know the basics of Linux-USB, it's time to look at an example device. Consider a system equipped with a telemetry card connected to the processor via internal USB, as shown in bus 2 of Figure 11.2. The card acquires data from a remote device and ferries it to the processor over USB. An example telemetry card is a medical-grade board that monitors or programs an implanted device.

Let's assume that our example telemetry card has the following endpoints having the semantics described in Table 11.2:

- A control endpoint attached to an on-card configuration register
- A bulk ***IN*** endpoint that passes remote telemetry information collected by the card to the processor
- A bulk ***OUT*** endpoint that transfers data in the reverse direction

**Table 11.2. Register Space in the Telemetry Card**

| Register | Associated Endpoint |
|---|---|
| Telemetry Configuration Register | Control endpoint ***0*** (register offset ***0xA***). |
| Telemetry Data-In Register | Bulk ***IN*** endpoint. The endpoint address is assigned during device enumeration. |
| Telemetry Data-Out Register | Bulk ***OUT*** endpoint. The endpoint address is assigned during device enumeration. |

Let's build a minimal driver for this card partly based on the USB skeleton driver, *drivers/usb/usb-skeleton.c*.

Because PCMCIA, PCI, and USB devices have similar characteristics such as hotplug support, some driver methods and data structures belonging to these subsystems resemble each other. This is especially true for the portions responsible for initializing and probing. As we progress through the telemetry driver and notice parallels with what we learned for PCI drivers in Chapter 10, we will pause and take note.

### Initializing and Probing

Like PCI and PCMCIA drivers, USB drivers have ***probe()/disconnect()***[2] methods to support hotplugging, and a table that contains the identity of devices they support. A USB device is identified by the ***usb_device_id*** structure defined in *include/linux/mod_devicetable.h*. You may recall from the previous chapter that the ***pci_device_id*** structure, also defined in the same header file, identifies PCI devices.

[2] ***disconnect()*** is called ***remove()*** in PCI and PCMCIA parlance.

```
struct usb_device_id {
  /* ... */
  __u16         idVendor;        /* Vendor ID */
```

```
  __u16         idProduct;       /* Device ID */
  /* ... */
  __u8          bDeviceClass;    /* Device class */
  __u8          bDeviceSubClass; /* Device subclass */
  __u8          bDeviceProtocol; /* Device protocol */
 /* ... */
};
```

***idVendor*** and ***idProduct***, respectively, hold the manufacturer ID and product ID, whereas ***bDeviceClass***, ***bDeviceSubClass***, and ***bDeviceProtocol*** categorize the device based on its functionality. This classification, determined by the USB specification, allows implementation of generic client drivers as discussed in the section "Class Drivers" later.

Listing 11.2 implements the telemetry driver's initialization routine, ***usb_tele_init()***, which calls on ***usb_register()*** to register its ***usb_driver*** structure with the USB core. As shown in the listing, ***usb_driver*** ties the driver's ***probe()*** method, ***disconnect()*** method, and ***usb_device_id*** table together. ***usb_driver*** is similar to ***pci_driver***, except that the ***disconnect()*** method in the former is named ***remove()*** in the latter.

### Listing 11.2. Initializing the Driver

Code View:

```
#define USB_TELE_VENDOR_ID    0xABCD  /* Manufacturer's Vendor ID */
#define USB_TELE_PRODUCT_ID   0xCDEF  /* Device's Product ID */

/* USB ID Table specifying the devices that this driver supports */
static struct usb_device_id tele_ids[] = {
  { USB_DEVICE(USB_TELE_VENDOR_ID, USB_TELE_PRODUCT_ID) },
  { } /* Terminate */
};

MODULE_DEVICE_TABLE(usb, tele_ids);

/* The usb_driver structure for this driver */
static struct usb_driver tele_driver
{
  .name       =  "tele",           /* Unique name */
  .probe      =  tele_probe,       /* See Listing 11.3 */
  .disconnect =  tele_disconnect,  /* See Listing 11.3 */
  .id_table   =  tele_ids,         /* See above */
};

/* Module Initialization */
static int __init
usb_tele_init(void)
{
  /* Register with the USB core */
  result = usb_register(&tele_driver);

  /* ... */
  return 0;
}

/* Module Exit */
static void __exit
usb_tele_exit(void)
{
  /* Unregister from the USB core */
```

```
  usb_deregister(&tele_driver);
  return;
}

module_init(usb_tele_init);
module_exit(usb_tele_exit);
```

The **USB_DEVICE()** macro creates a **usb_device_id** from the vendor and product IDs supplied to it. This is analogous to the **PCI_DEVICE()** macro discussed in the previous chapter. The **MODULE_DEVICE_TABLE()** macro marks **tele_ids** in the module image so that the module can be loaded on demand if the card is hotplugged. This is again similar to what we discussed for PCMCIA and PCI devices in the previous two chapters.

When the USB core detects a device with properties matching the ones declared in the **usb_device_id** table belonging to a client driver, it invokes the **probe()** method registered by that driver. When the device is unplugged or if the module is unloaded, the USB core invokes the driver's **disconnect()** method.

Listing 11.3 implements the **probe()** and **disconnect()** methods of the telemetry driver. It starts by defining a device-specific structure, **tele_device_t**, which contains the following fields:

- A pointer to the associated **usb_device**.
- A pointer to the **usb_interface**. Revisit Listing 11.1 to see this structure in use.
- A control URB (**ctrl_urb**) to communicate with the telemetry configuration register, and a **ctrl_req** to formulate programming requests to this register. These fields are described in the next section "Accessing Registers."
- The card has a bulk **IN** endpoint through which you can glean the collected telemetry information. Associated with this endpoint are three fields: **bulk_in_addr**, which holds the endpoint address; **bulk_in_buf**, which stores received data; and **bulk_in_len**, which contains the size of the receive data buffer.
- The card has a bulk **OUT** endpoint to facilitate downstream data transfer. **tele_device_t** has a field called **bulk_out_addr** to store the address of this endpoint. There are fewer data structures in the **OUT** direction because in this simple case we use a synchronous URB submission interface that hides several implementation details.

Khubd invokes the card's **probe()** method, **tele_probe()**, soon after enumeration. **tele_probe()** performs three tasks:

1. Allocates memory for the device-specific structure, ***tele_device_t***.

2. Initializes the following fields in ***tele_device_t*** related to the device's bulk endpoints: ***bulk_in_buf***, ***bulk_in_len***, ***bulk_in_addr***, and ***bulk_out_addr***. For this, it uses the data collected by the hub driver during enumeration. This data is available in descriptor structures discussed in the section "Descriptor Structures."

3. Exports the character device */dev/tele* to user space. Applications operate over */dev/tele* to exchange data with the telemetry card. ***tele_probe()*** invokes ***usb_register_dev()*** and supplies it the ***file_operations*** that form the underlying pillars of the */dev/tele* interface via the ***usb_class_driver*** structure.

The address of the device-specific structure allocated in Step 1 has to be saved so that other methods can access it. To achieve this, the telemetry driver uses a threefold strategy depending on the function arguments available to various driver routines. To save this structure pointer between the ***probe()*** and ***open()*** invocation threads, the driver uses the device's ***driver_data*** field via the pair of functions, ***usb_set_intfdata()*** and ***usb_get_intfdata()***. To save the address of the structure pointer between the ***open()*** thread and other entry points, ***open()*** stores it in the */dev/tele*'s ***file->private_data*** field. This is because the kernel supplies these char entry points with */dev/tele*'s ***inode*** pointer as argument rather than the ***usb_interface*** pointer. To glean the address of the device-specific structure from URB callback functions, the associated submission threads use the URB's ***context*** field as the storage area. Look at Listings 11.3, 11.4, and 11.5 to see these mechanisms in action.

All USB character devices answer to major number ***180***. If you enable ***CONFIG_USB_DYNAMIC_MINORS*** during kernel configuration, the USB core dynamically selects a minor number from the available pool. This behavior is similar to registering misc drivers after specifying ***MISC_DYNAMIC_MINOR*** in the ***miscdevice*** structure (as discussed in the section "Misc Drivers" in Chapter 5, "Character Drivers"). If you choose not to enable ***CONFIG_USB_DYNAMIC_MINORS***, the USB subsystem selects an available minor number starting at the minor base set in the ***usb_class_driver*** structure.

### Listing 11.3. Probing and Disconnecting

Code View:

```
/* Device-specific structure */
typedef struct {
  struct usb_device      *usbdev;         /* Device representation */
  struct usb_interface   *interface;      /* Interface representation*/
  struct urb             *ctrl_urb;       /* Control URB for
                                             register access */
  struct usb_ctrlrequest  ctrl_req;       /* Control request
                                             as per the spec */
  unsigned char          *bulk_in_buf;    /* Receive data buffer */
  size_t                  bulk_in_len;    /* Receive buffer size */
  __u8                    bulk_in_addr;   /* IN endpoint address */
  __u8                    bulk_out_addr;  /* OUT endpoint address */
  /* ... */                               /* Locks, waitqueues,
                                             statistics.. */
} tele_device_t;

#define TELE_MINOR_BASE 0xAB    /* Assigned by the Linux-USB
                                   subsystem maintainer */

/* Conventional char driver entry points.
   See Chapter 5, "Character Drivers." */
static struct file_operations tele_fops =
{
  .owner   =  THIS_MODULE,  /* Owner */
```

```
  .read    =  tele_read,    /* Read method */
  .write   =  tele_write,   /* Write method */
  .ioctl   =  tele_ioctl,   /* Ioctl method */
  .open    =  tele_open,    /* Open method */
  .release =  tele_release, /* Close method */
};


static struct usb_class_driver tele_class = {
  .name       = "tele",
  .fops       = &tele_fops,      /* Connect with /dev/tele */
  .minor_base = TELE_MINOR_BASE, /* Minor number start */
};
/* The probe() method is invoked by khubd after device
   enumeration. The first argument, interface, contains information
   gleaned during the enumeration process. id is the entry in the
   driver's usb_device_id table that matches the values read from
   the telemetry card. tele_probe() is based on skel_probe()
   defined in drivers/usb/usb-skeleton.c */
static int
tele_probe(struct usb_interface *interface,
           const struct usb_device_id *id)
{
  struct usb_host_interface *iface_desc;
  struct usb_endpoint_descriptor *endpoint;
  tele_device_t *tele_device;
  int retval = -ENOMEM;

  /* Allocate the device-specific structure */
  tele_device = kzalloc(sizeof(tele_device_t), GFP_KERNEL);

  /* Fill the usb_device and usb_interface */
  tele_device->usbdev =
               usb_get_dev(interface_to_usbdev(interface));
  tele_device->interface = interface;

  /* Set up endpoint information from the data gleaned
     during device enumeration */
  iface_desc = interface->cur_altsetting;
  for (int i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
    endpoint = &iface_desc->endpoint[i].desc;

    if (!tele_device->bulk_in_addr &&
        usb_endpoint_is_bulk_in(endpoint)) {
      /* Bulk IN endpoint */
      tele_device->bulk_in_len =
                     le16_to_cpu(endpoint->wMaxPacketSize);
      tele_device->bulk_in_addr = endpoint->bEndpointAddress;
      tele_device->bulk_in_buf =
                kmalloc(tele_device->bulk_in_len, GFP_KERNEL);
    }

    if (!tele_device->bulk_out_addr &&
       usb_endpoint_is_bulk_out(endpoint)) {
      /* Bulk OUT endpoint */
      tele_device->bulk_out_addr = endpoint->bEndpointAddress;
    }
   }
```

```
   if (!(tele_device->bulk_in_addr && tele_device->bulk_out_addr)) {
     return retval;
   }

   /* Attach the device-specific structure to this interface.
      We will retrieve it from tele_open() */
   usb_set_intfdata(interface, tele_device);

   /* Register the device */
   retval = usb_register_dev(interface, &tele_class);
   if (retval) {
     usb_set_intfdata(interface, NULL);
     return retval;
   }

   printk("Telemetry device now attached to /dev/tele\n");
   return 0;
}

/* Disconnect method. Called when the device is unplugged or when the module is
   unloaded */
static void
tele_disconnect(struct usb_interface *interface)
{
  tele_device_t *tele_device;
  /* ... */

  /* Reverse of usb_set_intfdata() invoked from tele_probe() */
  tele_device = usb_get_intfdata(interface);

  /* Zero out interface data */
  usb_set_intfdata(interface, NULL);

  /* Release /dev/tele */
  usb_deregister_dev(interface, &tele_class);

  /* NULL the interface. In the real world, protect this
     operation using locks */
  tele_device->interface = NULL;
  /* ... */
}
```

## Accessing Registers

The `open()` method initializes the on-card telemetry configuration register when an application opens */dev/tele*. To set the contents of this register, `tele_open()` submits a control URB attached to the default endpoint `0`. When you submit a control URB, you have to supply an associated control request. The structure that sends a control request to a USB device has to conform to Chapter 9 of the USB specification and is defined as follows in *include/linux/usb/ch9.h*:

```
struct usb_ctrlrequest {
  __u8 bRequestType;
  __u8 bRequest;
```

```
  __le16 wValue;
  __le16 wIndex;
  __le16 wLength;
} __attribute__ ((packed));
```

Let's take a look at the components that make up a ***usb_ctrlrequest***. The ***bRequest*** field identifies the control request. ***bRequestType*** qualifies the request by encoding the data transfer direction, the request category, and whether the recipient is a device, interface, endpoint, or something else. ***bRequest*** can either belong to a set of standard values or be vendor-defined. In our example, the ***bRequest*** for writing to the telemetry configuration register is a vendor-defined one. ***wValue*** holds the data to be written to the register, ***wIndex*** is the desired offset into the register space, and ***wLength*** is the number of bytes to be transferred.

Listing 11.4 implements ***tele_open()***. Its main task is to program the telemetry configuration register with appropriate values. Before browsing the listing, revisit the ***tele_device_t*** structure defined in Listing 11.3 focusing on two fields: ***ctrl_urb*** and ***ctrl_req***. The former is a control URB for communicating with the configuration register, whereas the latter is the associated ***usb_ctrlrequest***.

To program the telemetry configuration register, ***tele_open()*** does the following:

1. Allocates a control URB to prepare for the register write.

2. Creates a ***usb_ctrlrequest*** and populates it with the request identifier, request type, register offset, and the value to be programmed.

3. Creates a control pipe attached to endpoint ***0*** of the telemetry card to carry the control URB.

4. Because ***tele_open()*** submits the URB asynchronously, it needs to wait for the associated callback function to finish before returning to its caller. To this end, ***tele_open()*** calls on the kernel's completion API for assistance using ***init_completion()***. Step 7 calls the matching ***wait_for_completion()*** that waits until the callback function invokes ***complete()***. We discussed the completion API in the section "Completion Interface" in Chapter 3.

5. Initializes fields in the control URB using ***usb_fill_control_urb()***. This includes the ***usb_ctrlrequest*** populated in Step 2.

6. Submits the URB to the USB core using ***usb_submit_urb()***.

7. Waits until the callback function signals that the register programming is complete.

8. Returns the status.

**Listing 11.4. Initialize the Telemetry Configuration Register**

Code View:

```
/* Offset of the Telemetry configuration register
   within the on-card register space */
#define TELEMETRY_CONFIG_REG_OFFSET      0x0A

/* Value to program in the configuration register */
#define TELEMETRY_CONFIG_REG_VALUE       0xBC

/* The vendor-defined bRequest for programming the
   configuration register */
```

```
#define TELEMETRY_REQUEST_WRITE          0x0D

/* The vendor-defined bRequestType */
#define TELEMETRY_REQUEST_WRITE_REGISTER 0x0E

/* Open method */
static int
tele_open(struct inode *inode, struct file *file)
{
  struct completion tele_config_done;
  tele_device_t *tele_device;
  void *tele_ctrl_context;
  char *tmp;
  __le16 tele_config_index = TELEMETRY_CONFIG_REG_OFFSET;
  unsigned int tele_ctrl_pipe;
  struct usb_interface *interface;

  /* Obtain the pointer to the device-specific structure.
     We saved it using usb_set_intfdata() in tele_probe() */
  interface = usb_find_interface(&tele_driver, iminor(inode));
  tele_device = usb_get_intfdata(interface);

  /* Allocate a URB for the control transfer */
  tele_device->ctrl_urb = usb_alloc_urb(0, GFP_KERNEL);
  if (!tele_device->ctrl_urb) {
    return -EIO;
  }

  /* Populate the Control Request */
  tele_device->ctrl_req.bRequestType = TELEMETRY_REQUEST_WRITE;
  tele_device->ctrl_req.bRequest =
                       TELEMETRY_REQUEST_WRITE_REGISTER;
  tele_device->ctrl_req.wValue =
                       cpu_to_le16(TELEMETRY_CONFIG_REG_VALUE);
  tele_device->ctrl_req.wIndex =
                       cpu_to_le16p(&tele_config_index);
  tele_device->ctrl_req.wLength = cpu_to_le16(1);
  tele_device->ctrl_urb->transfer_buffer_length = 1;
  tmp = kmalloc(1, GFP_KERNEL);
  *tmp = TELEMETRY_CONFIG_REG_VALUE;

  /* Create a control pipe attached to endpoint 0 */
  tele_ctrl_pipe = usb_sndctrlpipe(tele_device->usbdev, 0);

  /* Initialize the completion mechanism */
  init_completion(&tele_config_done);

  /* Set URB context. The context is part of the URB that is passed
     to the callback function as an argument. In this case, the
     context is the completion structure, tele_config_done */
  tele_ctrl_context = (void *)&tele_config_done;

  /* Initialize the fields in the control URB */
  usb_fill_control_urb(tele_device->ctrl_urb, tele_device->usbdev,
                       tele_ctrl_pipe,
                       (char *) &tele_device->ctrl_req,
                       tmp, 1, tele_ctrl_callback,
                       tele_ctrl_context);
  /* Submit the URB */
```

```
  usb_submit_urb(tele_device->ctrl_urb, GFP_ATOMIC);

  /* Wait until the callback returns indicating that the telemetry
     configuration register has been successfully initialized */
  wait_for_completion(&tele_config_done);

  /* Release our reference to the URB */
  usb_free_urb(urb);

  kfree(tmp);

  /* Save the device-specific object to the file's private_data
     so that you can directly retrieve it from other entry points
     such as tele_read() and tele_write() */
  file->private_data = tele_device;

  /* Return the URB transfer status */
  return(tele_device->ctrl_urb->status);
}


/* Callback function */
static void
tele_ctrl_callback(struct urb *urb)
{
  complete((struct completion *)urb->context);
}
```

You can render **`tele_open()`** simpler using **`usb_control_msg()`**, a blocking version of **`usb_submit_urb()`** that internally hides synchronization and callback details for control URBs. We preferred the asynchronous approach for learning purposes.

## Data Transfer

Listing 11.5 implements the **`read()`** and **`write()`** entry points of the telemetry driver. These methods perform the real work when an application reads or writes to */dev/tele*. **`tele_read()`** performs synchronous URB submission because the calling process wants to block until telemetry data is available. **`tele_write()`**, however, uses asynchronous submission and returns to the calling thread without waiting for a confirmation that the data accepted by the driver has been successfully transferred to the device.

Because asynchronous transfers go hand in hand with a callback routine, Listing 11.5 implements **`tele_write_callback()`**. This routine examines **`urb->status`** to decipher the submission status. It also frees the transfer buffer allocated by **`tele_write()`**.

### Listing 11.5. Data Exchange with the Telemetry Card

Code View:

```
/* Read entry point */
static ssize_t
tele_read(struct file *file, char *buffer,
          size_t count, loff_t *ppos)
{
```

```
  int retval, bytes_read;
  tele_device_t *tele_device;

  /* Get the address of tele_device */
  tele_device = (tele_device_t *)file->private_data;

  /* ... */

  /* Synchronous read */
  retval = usb_bulk_msg(tele_device->usbdev, /* usb_device */
       usb_rcvbulkpipe(tele_device->usbdev,
                tele_device->bulk_in_addr),  /* Pipe */
       tele_device->bulk_in_buf,             /* Read buffer */
       min(tele_device->bulk_in_len, count), /* Bytes to read */
       &bytes_read,                          /* Bytes read */
       5000);                                /* Timeout in 5 sec */

  /* Copy telemetry data to user space */
  if (!retval) {
    if (copy_to_user(buffer, tele_device->bulk_in_buf,
                     bytes_read)) {
      return -EFAULT;
    } else {
      return bytes_read;
    }
  }
  return retval;
}

/* Write entry point */
static ssize_t
tele_write(struct file *file, const char *buffer,
           size_t write_count, loff_t *ppos)
{
  char *tele_buf = NULL;
  struct urb *urb = NULL;
  tele_device_t *tele_device;

  /* Get the address of tele_device */
  tele_device = (tele_device_t *)file->private_data;

  /* ... */

  /* Allocate a bulk URB */
  urb = usb_alloc_urb(0, GFP_KERNEL);
  if (!urb) {
    return -ENOMEM;
  }

  /* Allocate a DMA-consistent transfer buffer and copy in
     data from user space. On return, tele_buf contains
     the buffer's CPU address, while urb->transfer_dma
     contains the DMA address */
  tele_buf = usb_buffer_alloc(tele_dev->usbdev, write_count,
                              GFP_KERNEL, &urb->transfer_dma);
  if (copy_from_user(tele_buf, buffer, write_count)) {
    usb_buffer_free(tele_device->usbdev, write_count,
                    tele_buf, urb->transfer_dma);
    usb_free_urb(urb);
```

```
    return -EFAULT
  }

  /* Populate bulk URB fields */
  usb_fill_bulk_urb(urb, tele_device->usbdev,
                    usb_sndbulkpipe(tele_device->usbdev,
                    tele_device->bulk_out_addr),
                    tele_buf, write_count, tele_write_callback,
                    tele_device);
  /* urb->transfer_dma is valid, so preferably utilize
     that for data transfer */
  urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;

  /* Submit URB asynchronously */
  usb_submit_urb(urb, GFP_KERNEL);
  /* Release URB reference */
  usb_free_urb(urb);

  return(write_count);
}

/* Write callback */
static void
tele_write_callback(struct urb *urb)
{
  tele_device_t *tele_device;

  /* Get the address of tele_device */
  tele_device = (tele_device_t *)urb->context;

  /* urb->status contains the submission status. It's 0 if
     successful. Resubmit the URB in case of errors other than
     -ENOENT, -ECONNRESET, and -ESHUTDOWN */
  /* ... */

  /* Free the transfer buffer. usb_buffer_free() is the
     release-counterpart of usb_buffer_alloc() called
     from tele_write() */
  usb_buffer_free(urb->dev, urb->transfer_buffer_length,
                  urb->transfer_buffer, urb->transfer_dma);

}
```

## Class Drivers

The USB specification introduces the concept of device classes and describes the functionality of each class driver. Examples of standard device classes include mass storage, networking, hubs, serial converters, audio, video, imaging, modems, printers, and *human interface devices* (HIDs). Class drivers are generic and let you plug and play a wide array of cards without the need for developing and installing drivers for every single device. The Linux-USB subsystem includes support for major class drivers.

Each USB device has a class and a subclass code. The mass storage class (***0x08***), for example, supports subclasses such as compact disc (***0x02***), tape (***0x03***), and solid-state storage (***0x06***). As you saw previously, device drivers populate the ***usb_device_id*** structure with the classes and subclasses they support. You can glean a device's class and subclass information by looking at the "I:" lines in the */proc/bus/usb/devices* output.

Let's take a look at some important class drivers.

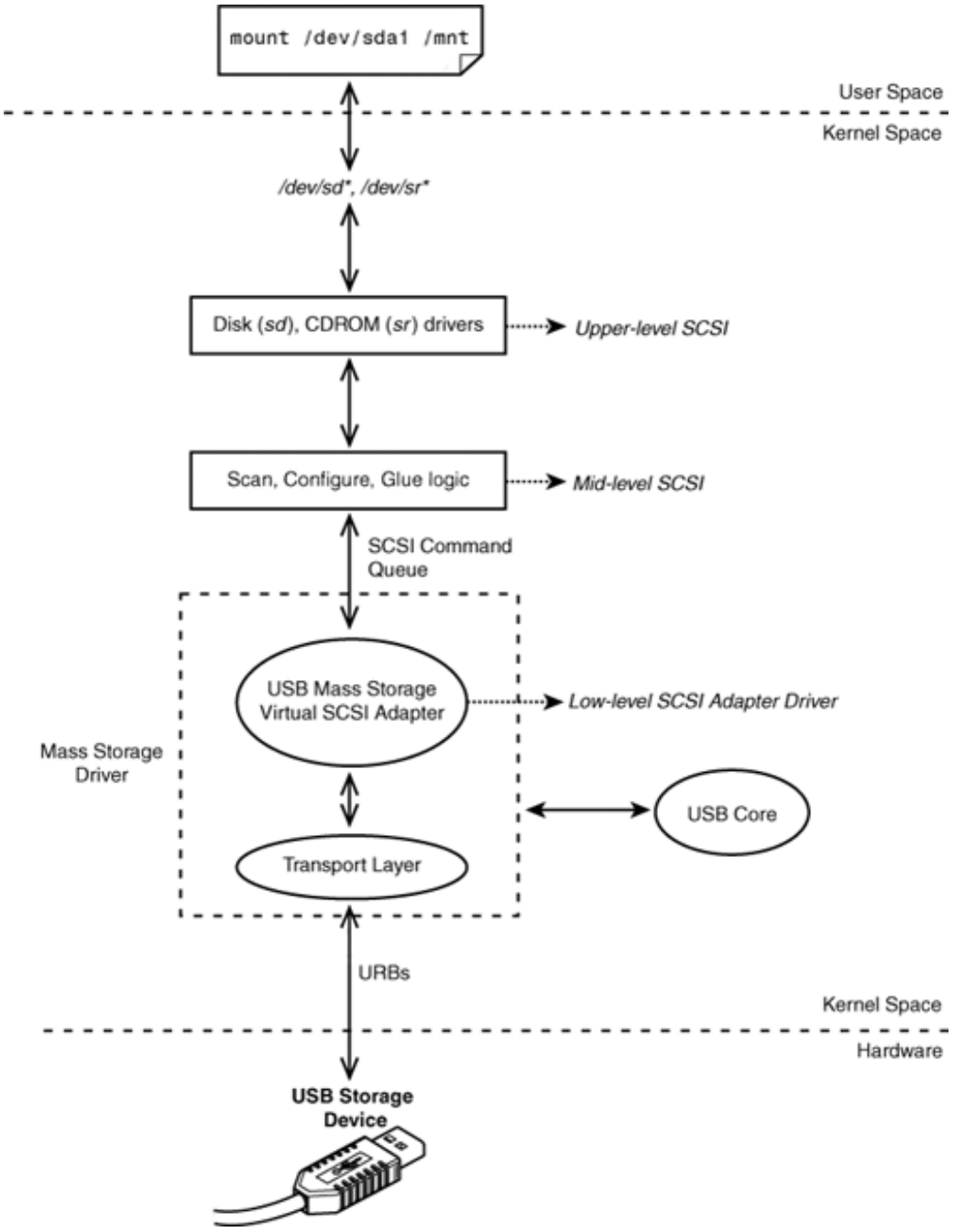
### Mass Storage

In USB parlance, mass storage refers to USB hard disks, pen drives, CD-ROMs, floppy drives, and similar storage devices. USB mass storage devices adhere to the *Small Computer System Interface* (SCSI) protocol to communicate with host systems. Block access to USB storage devices is hence routed through the kernel's SCSI subsystem. Figure 11.4 provides you an overview of the interaction between USB storage and SCSI subsystems. As shown in the figure, the SCSI subsystem is architected into three layers:

1. Top-level drivers for devices such as disks (*sd.c*) and CD-ROMs (*sr.c*)
2. A middle-level layer that scans the bus, configures devices, and glues top-level drivers to low-level drivers
3. Low-level SCSI adapter drivers

**Figure 11.4. USB mass storage and SCSI.**

[View full size image]




The mass storage driver registers itself as a virtual SCSI adapter. The virtual adapter communicates upstream via SCSI commands and downstream using URBs. A USB disk appears to higher layers as a SCSI device attached to this virtual adapter.

To better understand the interactions between the USB and SCSI layers, let's implement a modification to the USB mass storage driver. The *usbfs* node */proc/bus/usb/devices*, contains the properties and connection details of all USB devices attached to the system. The "T:" line in the */proc/bus/usb/devices* output, for example, contains the bus number, the device's depth from the root hub, operational speed, and so on. The "P:" line contains the vendor ID, product ID, and revision number of the device. All the information available in */proc/bus/usb/devices* is managed by the USB subsystem, but there is one piece missing that is under the jurisdiction of the SCSI subsystem. The */dev* node name associated with the USB storage device (*sd[a-z][1-9]*

for disks and *sr[0-9]* for CD-ROMs) is not available in */proc/bus/usb/devices.* To overcome this limitation, let's add an "N:" line that displays the */dev* node name associated with the device. Listing 11.6 shows the necessary code changes in the form of a source patch to the 2.6.23.1 kernel tree.

### Listing 11.6. Adding a Disk's */dev* Name to usbfs

Code View:

```
include/scsi/scsi_host.h:
struct Scsi_Host {
  /* ... */
  void *shost_data;
+ char snam[8];      /* /dev node name for this disk */
  /* ... */
};


drivers/usb/storage/usb.h:
struct us_data {
  /* ... */
+ char magic[4];
};


include/linux/usb.h:
struct usb_interface {
  /* ... */
+ void *private_data;
};


drivers/usb/storage/usb.c:
static int storage_probe(struct usb_interface *intf,
                         const struct usb_device_id *id)
{
  /* ... */
  memset(us, 0, sizeof(struct us_data));
+ intf->private_data = (void *) us;
+ strncpy(us->magic, "disk", 4);
  mutex_init(&(us->dev_mutex));
  /* ... */
}


drivers/scsi/sd.c:
static int sd_probe(struct device *dev)
{
  /* ... */
  add_disk(gd);
+ memset(sdp->host->snam,0, sizeof(sdp->host->snam));
+ strncpy(sdp->host->snam, gd->disk_name, 3);
  sdev_printk(KERN_NOTICE, sdp, "Attached scsi %sdisk %s\n",
              sdp->removable ? "removable " : "", gd->disk_name);
  /* ... */
}

drivers/scsi/sr.c:
static int sr_probe(struct device *dev)
{
```

```
  /* ... */
  add_disk(disk);
+ memset(sdev->host->snam,0, sizeof(sdev->host->snam));
+ strncpy(sdev->host->snam, cd->cdi.name, 3);
  sdev_printk(KERN_DEBUG, sdev, "Attached scsi CD-ROM %s\n",
              cd->cdi.name);
  /* ... */
}

drivers/usb/core/devices.c:
  /* ... */
  #include <asm/uaccess.h>
+ #include <scsi/scsi_host.h>
+ #include "../storage/usb.h"

static ssize_t usb_device_dump(char __user **buffer, size_t *nbytes,
                         loff_t *skip_bytes, loff_t *file_offset,
                         struct usb_device *usbdev,
                         struct usb_bus *bus, int level,
                         int index, int count)
{
  /* ... */
  ssize_t total_written = 0;
+ struct us_data *us_d;
+ struct Scsi_Host *s_h;
  /* ... */
  data_end = pages_start + sprintf(pages_start, format_topo,
                                   bus->busnum, level,
                                   parent_devnum,
                                   index, count, usbdev->devnum,
                                   speed, usbdev->maxchild);
+ /* Assume this device supports only one interface */
+ us_d =  (struct us_data *)
+           (usbdev->actconfig->interface[0]->private_data);
+
+ if ((us_d) && (!strncmp(us_d->magic, "disk", 4))) {
+   s_h = (struct Scsi_Host *) container_of((void *)us_d,
+                                           struct Scsi_Host,
+                                           hostdata);
+   data_end += sprintf(data_end, "N: ");
+   data_end += sprintf(data_end, "Device=%.100s",s_h->snam);
+   if (!strncmp(s_h->snam, "sr", 2)) {
+     data_end += sprintf(data_end, " (CDROM)\n");
+   } else if (!strncmp(s_h->snam, "sd", 2)) {
+     data_end += sprintf(data_end, " (Disk)\n");
+   }
+ }
   /* ... */
}
```

To understand Listing 11.6, let's first trace the code flow, continuing from where we left off in the section "Enumeration." In that section, we inserted a USB pen drive and followed the execution train until the invocation of ***storage_probe()***, the ***probe()*** method of the mass storage driver. Moving further:

1. ***storage_probe()*** registers a virtual SCSI adapter by calling ***scsi_add_host()***, supplying a private data structure called ***us_data*** as argument. ***scsi_add_host()*** returns a ***Scsi_Host*** structure for this virtual adapter, with space at the end for ***us_data***.

2. It starts a kernel thread called *usb-storage* to handle all SCSI commands queued to the virtual adapter.

3. It schedules a kernel thread called *usb-stor-scan* that requests the SCSI middle-level layer to scan the bus for attached devices.

4. The device scan initiated in Step 3 discovers the presence of the inserted pen drive and binds the upper-level SCSI disk driver (*sd.c*) to the device. This results in the invocation of the SCSI disk driver's probe method, ***sd_probe()***.

5. The *sd* driver allocates a */dev/sd** node to the disk. From this point on, applications use this interface to access the USB disk. The SCSI subsystem queues disk commands to the virtual adapter, which the usb-storage kernel thread handles using appropriate URBs.

Now that you know the basics, let's dissect Listing 11.6, looking at the data structure additions first. The listing adds a ***snam*** field to the ***Scsi_Host*** structure to hold the associated SCSI */dev* name that we are interested in. It also adds a private field to the ***usb_interface*** structure to associate each USB interface with its ***us_data***. Because ***us_data*** is relevant only for storage devices, we need to ensure the validity of the private field of a USB interface before accessing it as ***us_data***. For this, Listing 11.6 adds a magic string, "disk," to ***us_data***.

The *usbfs* modification in Listing 11.6 (to *drivers/usb/core/devices.c*) pulls out the ***us_data*** associated with each interface via the private data field of its ***usb_interface***. It then latches on to the associated ***Scsi_Host*** using the ***container_of()*** function, because as you saw in Step 1 previously, ***usb_data*** is glued to the end of the corresponding ***Scsi_Host***. As you further saw in Step 5, ***Scsi_Host*** contains the */dev* node names that the sd and sr drivers populate. Usbfs stitches together an "N:" line using this information.

The following is the */proc/bus/usb/devices* output after integrating the changes in Listing 11.6 and attaching a PNY USB pen drive, an Addonics CD-ROM drive, and a Seagate hard disk to a laptop via a USB hub. The "N:" lines announce the identity of the */dev* node corresponding to each device:

Code View:
```
bash> cat /proc/bus/usb/devices
...
T:  Bus=04 Lev=02 Prnt=02 Port=00 Cnt=01 Dev#= 3 Spd=480 MxCh= 0
N:  Device=sda(Disk)
D:  Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1
P:  Vendor=154b ProdID=0002 Rev= 1.00
S:  Manufacturer=PNY
S:  Product=USB 2.0 FD
S:  SerialNumber=6E5C07005B4F
C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr= 0mA
I:* If#= 0 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50 Driver=usb-
    storage
E:  Ad=81(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms
E:  Ad=02(O) Atr=02(Bulk) MxPS= 512 Ivl=0ms

T:  Bus=04 Lev=02 Prnt=02 Port=01 Cnt=02 Dev#= 5 Spd=480 MxCh= 0
N:  Device=sr0(CDROM)
D:  Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1
P:  Vendor=0bf6 ProdID=a002 Rev= 3.00
S:  Manufacturer=Addonics
```

```
S:  Product=USB to IDE Cable
S:  SerialNumber=1301011002A9AFA9
C:* #Ifs= 1 Cfg#= 2 Atr=c0 MxPwr= 98mA
I:* If#= 0 Alt= 0 #EPs= 3 Cls=08(stor.) Sub=06 Prot=50 Driver=usb-
    storage
E:  Ad=01(O) Atr=02(Bulk) MxPS= 512 Ivl=125us
E:  Ad=82(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms
E:  Ad=83(I) Atr=03(Int.) MxPS=   2 Ivl=32ms

T:  Bus=04 Lev=02 Prnt=02 Port=02 Cnt=03 Dev#= 4 Spd=480 MxCh= 0
N:  Device=sdb(Disk)
D:  Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1
P:  Vendor=0bc2 ProdID=0501 Rev= 0.01
S:  Manufacturer=Seagate
S:  Product=USB Mass Storage
S:  SerialNumber=000000062459
C:* #Ifs= 1 Cfg#= 1 Atr=c0 MxPwr=  0mA
I:* If#= 0 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50 Driver=usb-
    storage
E:  Ad=02(O) Atr=02(Bulk) MxPS= 512 Ivl=0ms
E:  Ad=88(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms
...
```

As you can see, the SCSI subsystem has allotted *sda* to the pen drive, *sr0* to the CD-ROM, and *sdb* to the hard disk. User-space applications operate on these nodes to communicate with the respective devices. As you saw in Chapter 4, with the arrival of udev, however, you have the option of creating higher-level abstractions to identify each device without relying on the identity of the */dev* names allocated by the SCSI subsystem.

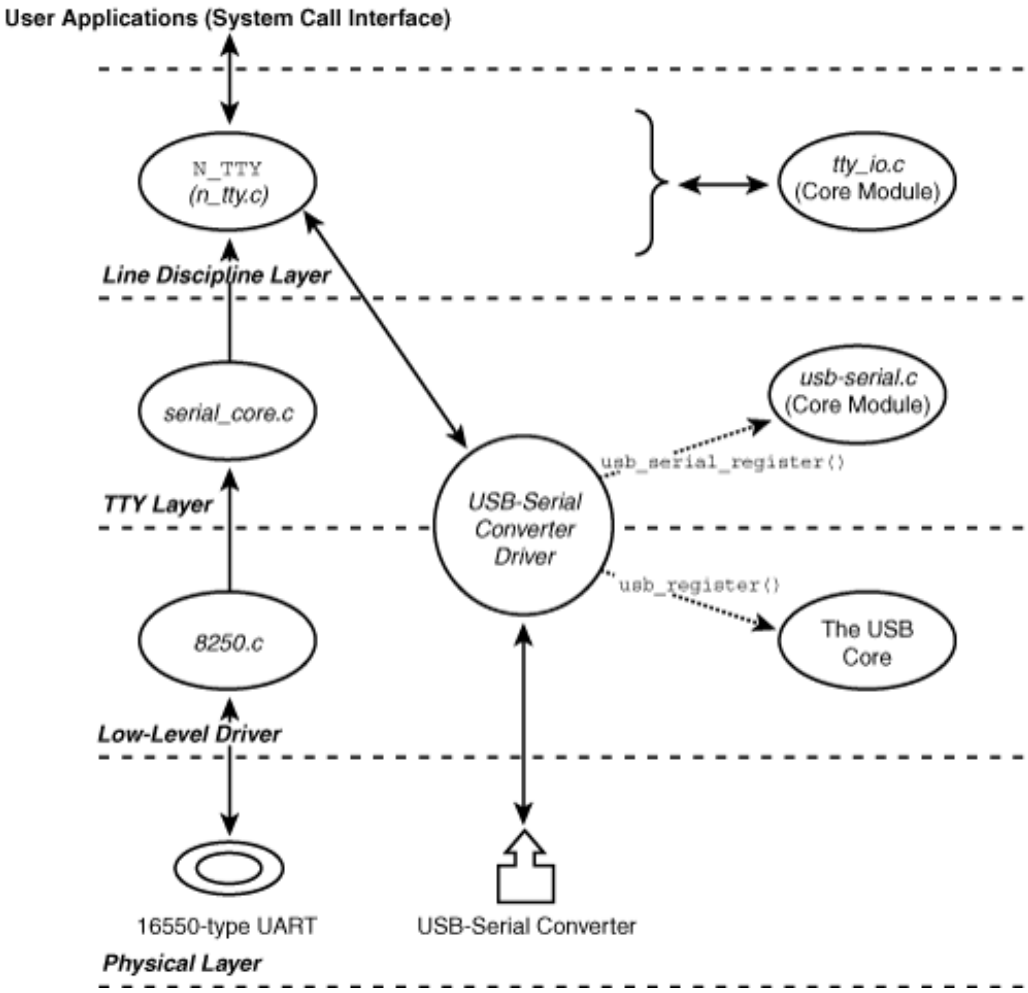
## USB-Serial

USB-to-serial converters bring serial port capabilities to your computer via USB. You can use a USB-to-serial converter, for example, to get a serial debug console from an embedded device on a development laptop that has no serial ports.

In Chapter 6, "Serial Drivers," you learned the benefits of the kernel's layered serial architecture. Figure 11.5 illustrates how the USB-Serial layer fits into the kernel's serial framework.

**Figure 11.5. The USB-Serial layer.**

[View full size image]




A USB-serial driver is similar to other USB client drivers except that it avails the services of a USB-Serial core in addition to the USB core. The USB-Serial core provides the following:

- A tty driver that insulates low-level USB-to-serial converter drivers from higher serial layers such as line disciplines.
- Generic ***probe()*** and ***disconnect()*** routines that individual USB-serial drivers can leverage.
- Device nodes to access USB-serial ports from user space. Applications operate on USB-serial ports via */dev/ttyUSBX*, where *X* is the serial port number. Terminal emulators such as *minicom* and protocols such as PPP run unchanged over these interfaces.

A low-level USB-to-serial converter driver essentially does the following:

1. Registers a ***usb_serial_driver*** structure with the USB-Serial core using ***usb_serial_register()***. The entry points supplied as part of ***usb_serial_driver*** form the crux of the driver.

2. Populates a ***usb_driver*** structure and registers it with the USB core using ***usb_register()***. This is similar to what the example telemetry driver does, except that a serial converter driver can count on the generic ***probe()*** and ***disconnect()*** routines provided by the USB-Serial core.

Listing 11.7 contains snippets from the FTDI driver (*drivers/usb/serial/ftdi_sio.c*) that accomplish these two registrations for USB-to-serial converters based on FTDI chipsets.

**Listing 11.7. A Snippet from the FTDI Driver**

Code View:

```
/* The usb_driver structure */
static struct usb_driver ftdi_driver = {
  .name          =  "ftdi_sio",           /* Name */
  .probe         =  usb_serial_probe,     /* Provided by the
                                             USB-Serial core */
  .disconnect    =  usb_serial_disconnect,/* Provided by the
                                             USB-Serial core */
  .id_table      =  id_table_combined,    /* List of supported
                                             devices built
                                             around the FTDI chip */
  .no_dynamic_id =  1,                    /* Supported ids cannot be
                                             added dynamically */
};

/* The usb_serial_driver structure */
static struct usb_serial_driver ftdi_sio_device = {
  /* ... */
  .num_ports           = 1,
  .probe               = ftdi_sio_probe,
  .port_probe          = ftdi_sio_port_probe,
  .port_remove         = ftdi_sio_port_remove,
  .open                = ftdi_open,
  .close               = ftdi_close,
  .throttle            = ftdi_throttle,
  .unthrottle          = ftdi_unthrottle,
  .write               = ftdi_write,
  .write_room          = ftdi_write_room,
  .chars_in_buffer     = ftdi_chars_in_buffer,
  .read_bulk_callback  = ftdi_read_bulk_callback,
  .write_bulk_callback = ftdi_write_bulk_callback,
  /* ... */
};

/* Driver Initialization */
static int __init ftdi_init(void)
{
   /* ... */
   /* Register with the USB-Serial core */
   retval = usb_serial_register(&ftdi_sio_device);
   /* ... */
   /* Register with the USB core */
   retval = usb_register(&ftdi_driver);
   /* ... */
}
```

## Human Interface Devices

Devices such as keyboards and mice are called *human interface devices* (HIDs). Take a look at the section "USB and Bluetooth Keyboards" in Chapter 7, "Input Drivers," for a discussion on the USB HID class.

## Bluetooth

A USB-Bluetooth dongle is a quick way to Bluetooth-enable your computer so that it can communicate with Bluetooth-equipped devices such as cell phones, mice, or handhelds. Chapter 16 discusses the USB Bluetooth class.

## Gadget Drivers

In a typical usage scenario, an embedded device connects to a PC host over USB. Embedded computers usually belong to the device side of USB, unlike PC systems that function as USB hosts. Because Linux runs on both embedded and PC systems, it needs support to run on either end of USB. The USB Gadget project brings USB device mode capability to embedded Linux systems. Bus 3 of the embedded Linux device in Figure 11.2 can, for example, use a *gadget driver* to let the device function as a mass storage drive when connected to a host computer.

Before proceeding, let's briefly look at some related terminology. The USB controller at the device side is variously called a *device controller*, *peripheral controller*, *client controller*, or *function controller*. The terms *gadget* and *gadget driver* are commonly used rather than the heavily overloaded words *device* and *device driver*.

USB gadget support is now part of the mainline kernel and contains the following:

- Drivers for USB device controllers integrated into SoC families such as Intel PXA, Texas Instruments OMAP, and Atmel AT91. These drivers additionally provide a *gadget API* that gadget drivers can use.

- Gadget drivers for device classes such as storage, networking, and serial converters. These drivers answer to their class when they receive enumeration requests from host-side software. A storage gadget driver, for example, identifies itself as a class `0x08` (mass storage class) device and exports a storage partition to the host. You can specify the associated block device node or filename via a module-insertion parameter. Because the exported region has to appear to the host as a mass storage device, the gadget driver implements the SCSI interactions required by the USB mass storage protocol. Gadget drivers are also available for Ethernet and serial devices.

- A skeletal gadget driver, *drivers/usb/gadget/zero.c*, that you may use to test device controller drivers.

Gadget drivers use the services of the gadget API provided by device controller drivers. They populate a `usb_gadget_driver` structure and register it with the kernel using `usb_gadget_register_driver()`. Hardware specifics are hidden inside the gadget API implementation offered by individual device controller drivers, so the gadget drivers themselves are hardware independent.

*Documentation/DocBook/gadget.tmpl* provides an overview of the gadget API. Have a look at http://linux-usb.org/gadget/ for more on the gadget project.

## Debugging

A USB bus analyzer magnifies the goings-on in the bus and is useful for debugging low-level problems. If you can't get hold of an analyzer, you might be able to make do with the kernel's soft USB tracer, *usbmon.* This tool captures traffic between USB host controllers and devices. To collect a trace, read from the *debugfs*[3] file */sys/kernel/debug/usbmon/Xt*, where *X* is the bus number to which your device is connected.

[3] An in-memory filesystem to export kernel debug data to user space.

For example, consider a USB disk connected to a PC. From the associated "T:" line in */proc/bus/usb/devices*, you can see that the drive is attached to bus 1:

```
T: Bus=01 Lev=01 Prnt=01 Port=03 Cnt=01 Dev#= 2 Spd=480 MxCh= 0
```

Ensure that you have enabled *debugfs* (`CONFIG_DEBUG_FS`) and usbmon (`CONFIG_USB_MON`) support in your kernel. This is a snapshot of usbmon output while copying a file from the disk:

Code View:

```
bash> mount -t debugfs none_debugs /sys/kernel/debug/
bash> cat /sys/kernel/debug/usbmon/1u
...
ee6a5c40 3718782540 S Bi:1:002:1 -115 20480 <
ee6a5cc0 3718782567 S Bi:1:002:1 -115 65536 <
ee6a5d40 3718782595 S Bi:1:002:1 -115 36864 <
ee6a5c40 3718788189 C Bi:1:002:1 0 20480 = 0f846801 118498f\ 15c60500 01680106
5e846801 608498fe 6f280087 68000000
ee6a5cc0 3718800994 C Bi:1:002:1 0 65536 = 118498fe 15c60500\ 01680106 5e846801
608498fe 6f280087 68000000 00884800
ee6a5d40 3718801001 C Bi:1:002:1 0 36864 = 13608498 fe4f4a01\ 00514a01 006f2800
87680000 00008848 00000100 b7f00100
...
```

Each output line starts with the URB address, followed by an event timestamp. An *S* in the next column indicates URB submission, and a *C* announces a callback. The following field has the format `URBType:Bus#:DeviceAddress:Endpoint#`. In the preceding output, a `URBType` of `Bi` stands for a bulk URB in the `IN` direction. After this, usbmon dumps the URB status, data length, a data tag (`=` or `<` in the preceding output), and the data words (if the tag is `=`). The last three lines in the preceding output are callbacks associated with bulk URBs submitted in earlier lines. You can match the callbacks with the related submissions using the URB addresses. *Documentation/usb/usbmon.txt* details usbmon syntax and contains example code to parse the output into human readable form.

If you turn on *Device Drivers* ⟶ *USB Support* ⟶ *USB Verbose Debug Messages* during kernel configuration, the kernel will emit the contents of all `dev_dbg()` statements present in the USB subsystem.

You can glean device and bus specific information from the USB filesystem (*usbfs*) node, */proc/bus/usb/devices*. And as we discuss in Chapter 19, "Drivers in User Space," usbfs also lets you implement USB device drivers in user space. Even when the final destination of your USB driver is inside the kernel, starting with a user-space driver can ease debugging and testing.

The linux-usb-devel mailing list is the forum to discuss questions related to USB device drivers. Visit https://lists.sourceforge.net/lists/listinfo/linux-usb-devel for subscription and archive retrieval information. Read www.linux-usb.org/usbtest for ideas on USB testing.

The home page of the Linux-USB project is www.linux-usb.org. You may download the USB 2.0 specification, OTG supplement, and other related standards from www.usb.org/developers/docs.

## Looking at the Sources

The USB core layer lives in *drivers/usb/core/.* This directory also contains URB manipulation routines and the usbfs implementation. The hub driver and khubd are part of *drivers/usb/core/hub.c.* The *drivers/usb/host/* directory contains host controller device drivers. USB-related header definitions reside in *include/linux/usb*.h.* The usbmon tracer is in *drivers/usb/mon/.* Look inside *Documentation/usb/* for Linux-USB documentation.

USB class drivers stay in various subdirectories under *drivers/usb/.* The mass storage driver *drivers/usb/storage/*, in tandem with the SCSI subsystem *drivers/scsi/*, implements the USB mass storage protocol. The *drivers/input/*[4] directory tree includes drivers for USB input devices such as keyboards and mice; *drivers/usb/serial/* has drivers for USB-to-serial converters; *drivers/usb/media/* supports USB multimedia devices; *drivers/net/usb/*[5] has drivers for USB Ethernet dongles; and *drivers/usb/misc/* contains drivers for miscellaneous USB devices such as LEDs, LCDs, and fingerprint sensors. Look at *drivers/usb/usb-skeleton.c* for a starting point driver template if you can't zero in on a closer match.

[4] Before the 2.6.22 kernel release, USB input device drivers used to reside in *drivers/usb/input/*.

[5] Before the 2.6.22 kernel release, USB network device drivers used to reside in *drivers/usb/net/*.

The USB gadget subsystem is in *drivers/usb/gadget/.* This directory contains USB device controller drivers, and gadget drivers for mass storage (*file_storage.c*), serial converters (*serial.c*), and Ethernet networking (*ether.c*).

Table 11.3 contains the main data structures used in this chapter and their location in the source tree. Table 11.4 lists the main kernel programming interfaces that you used in this chapter along with the location of their definitions.

**Table 11.3. Summary of Data Structures**

| Data Structure | Location | Description |
|---|---|---|
| ***urb*** | *include/linux/usb.h* | Centerpiece of the USB data transfer mechanism |
| ***pipe*** | *include/linux/usb.h* | Address element of a URB |
| ***usb_device_descriptor*** ***usb_config_descriptor*** ***usb_interface_descriptor*** ***usb_endpoint_descriptor*** | *include/linux/usb/ch9.h* | Descriptors that hold information about a USB device |
| ***usb_device*** | *include/linux/usb.h* | Representation of a USB device |
| ***usb_device_id*** | *include/linux/mod_devicetable.h* | Identity of a USB device |
| ***usb_driver*** | *include/linux/usb.h* | Representation of a USB client driver |
| ***usb_gadget_driver*** | *include/linux/usb_gadget.h* | Representation of a USB gadget driver |

**Table 11.4. Summary of Kernel Programming Interfaces**

| Kernel Interface | Location | Description |
|---|---|---|
| ***usb_register()*** | *include/linux/usb.h drivers/usb/core/driver.c* | Registers a ***usb_driver*** with the USB core |
| ***usb_deregister()*** | *drivers/usb/core/driver.c* | Unregisters a ***usb_driver*** from the USB core |
| ***usb_set_intfdata()*** | *include/linux/usb.h* | Attaches device-specific data to a ***usb_interface*** |
| ***usb_get_intfdata()*** | *include/linux/usb.h* | Detaches device-specific data from a ***usb_interface*** |
| ***usb_register_dev()*** | *drivers/usb/core/file.c* | Associates a character interface with a USB client driver |
| ***usb_deregister_dev()*** | *drivers/usb/core/file.c* | Dissociates a character interface from a USB client driver |
| ***usb_alloc_urb()*** | *drivers/usb/core/urb.c* | Allocates a URB |
| ***usb_fill_[control\|int\|bulk]_urb()*** | *include/linux/usb.h* | Populates a URB |
| ***usb_[control\|interrupt\|bulk]_msg()*** | *drivers/usb/core/message.c* | Wrappers for synchronous URB submission |
| ***usb_submit_urb()*** | *drivers/usb/core/urb.c* | Submits a URB to the USB core |
| ***usb_free_urb()*** | *drivers/usb/core/urb.c* | Frees references to a completed URB |
| ***usb_unlink_urb()*** | *drivers/usb/core/urb.c* | Frees references to a pending URB |
| ***usb_[rcv\|snd][ctrl\|int\|bulk\|isoc]pipe()*** | *include/linux/usb.h* | Creates a USB pipe |
| ***usb_find_interface()*** | *drivers/usb/core/usb.c* | Gets the ***usb_interface*** associated with a USB client driver |
| ***usb_buffer_alloc()*** | *drivers/usb/core/usb.c* | Allocates a consistent DMA transfer buffer |
| ***usb_buffer_free()*** | *drivers/usb/core/usb.c* | Frees a buffer that was allocated using ***usb_buffer_alloc()*** |
| ***usb_serial_register()*** | *drivers/usb/serial/usb-serial.c* | Registers a driver with the USB-Serial core |
| ***usb_serial_deregister()*** | *drivers/usb/serial/usb-serial.c* | Unregisters a driver from the USB-Serial core |
| ***usb_gadget_register_driver()*** | Device controller drivers in *drivers/usb/gadget/* | Registers a gadget with a device controller driver |