**Universal Serial Bus**

**(Embedded System Final Project)**

**2013 FALL**

**Chang Guem Song**

**Lee Chul Hoon**

**머 리 말**

위대한 수령 **김일성**동지께서는 다음과 같이 교시하시였다.

**《인민경제현대화의 목표는 생산의 종합적기계화, 자동화입니다. 인민경제를 현대화하려면 인민경제 모든 부문에서 뒤떨어진 기술을 발전된 기술로 개조하고 생산의 종합적기계화와 자동화를 실현하여야 합니다.》**

(《**김일성**저작집》 제35권, 326페지)

위대한 령도자 **김정일**동지께서는 다음과 같이 지적하시였다.

**《20세기는 기계제산업의 시대였다면 21세기는 정보산업의 시대로 될것입니다. 물질적부를 창조하는데서 기계제산업의 시대에는 주로 육체로동에 의거하였다면 정보산업의 시대에는 더욱더 지능로동에 의거하게 될것입니다.》**

(《**김정일**선집》 제15권, 110페지)

위대한 장군님께서 명시하신것처럼 현시대는 정보산업의 시대이다. 다시말하여 인간의 자주적이며 창조적인 지능활동이 사회경제발전의 강 력한 추동력으로 되며 사람들이 주로 정보와 정보기술을 활용하여 물질 문화적재부를 창조하고 향유하는 시대이다.

위대한 장군님의 정력적인 령도와 세심한 가르치심속에 끊임없는 발전과 비약의 한길을 걸어온 우리 나라의 정보과학기술은 온 세계앞에 우리 당의 과학기술중시정책의 생활력을 남김없이 과시하고있으며 우리의 과학자, 연구사들은 인민경제활성화와 인민생활향상에 필수적인 능률높은 프로그람들을 수많이 개발하고있다. 특히 오늘날 콤퓨터가 인간생활에 미치는 영향은 자못 큰것으로 하여 콤퓨터분야의 전문가들 앞에는 콤퓨터를 보다 편리하고 효률적으로 리용할수 있도록 해야 할 과제가 나서고있다.

콤퓨터를 자유자재로 리용하는데서 자료입력을 원만히 하는 문제는 대단히 중요하다.최근에 USB에 의한 자료통신이 많이 리용되고 있다.여기서는 USB의 개념과 구성방식, 우리 식 조작체계의 기초인 Linux 체계에서의 USB구동프로그람에 대하여 해설한다.

**Contents**

**머리말** ∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙ 2

Chapter 0. USB Introduction ∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙ 4

Chapter 1. USB Architecture∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙ 8

Chapter 2. Linux-USB Subsystem ∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙ 11

Chapter 3. Driver Data Structures ∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙ 13

Chapter 4. Enumeration ∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙ 16

Chapter 5. Device Example: Telemetry Card ∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙ 17

Chapter 6. Class Drivers ∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙ 29

Chapter 7. Gadget Drivers ∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙ 38

Chapter 8. Debugging ∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙ 39

Chapter 9. Looking at the Sources ∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙ 40

**지침으로 삼은 문헌** ∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙ 43

**참고문헌** ∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙ 44

**맺는말** ∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙∙ 45

1. **USB Introduction**
   1. **USB Overview**

USB can seem complex at first. It has a plethora of devices and a fairly large variety

of embedded host controllers. It has several modes of operation, and a given controller on a processor (or external to it) may have multiple modes of operation. If you’ve looked at the full list of USB configuration options in a recent Linux kernel, you quickly realize that it can be confusing to configure. We can eliminate some of that confusion by understanding some basic USB concepts.

* + 1. **USB Physical Topology**

USB is a master/slave bus topology. Each USB bus can have only one master, which

is called a *host controller*. Figure 0-1 illustrates the basic topology.

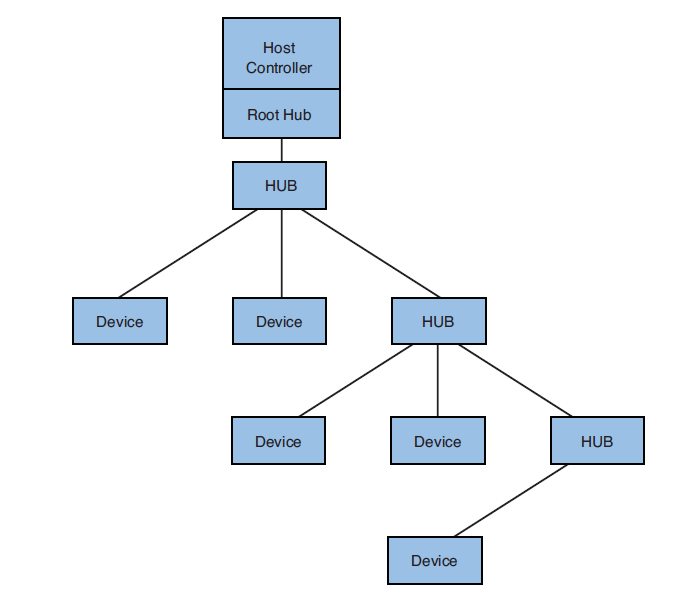
****

FIGURE 0-1 Simple USB topology

The host controller is always associated with a *root hub*. The root hub provides an attachment point to the host controller and provides the hub functions at the top of the USB hierarchy. The most common arrangement is that a host controller and root hub combination are brought out directly to a connector (through a transceiver chip) on the edge of the board. It is this connector that end users see.

The devices shown in Figure 0-1 are *endpoints*—physical USB appliances that plug into a USB hub. A device may support several *functions*, such as an audio interface that provides input and output functionality. The important concept here is that every USB device plugs into one and only one hub upstream of its location in the topology.

Devices on the USB bus are operated in a polled manner, controlled by the host controller. Only one device at a time can communicate on the bus, as directed by the host controller. Mechanisms exist in the specification to allocate a specified portion of bandwidth to a given function within a device.

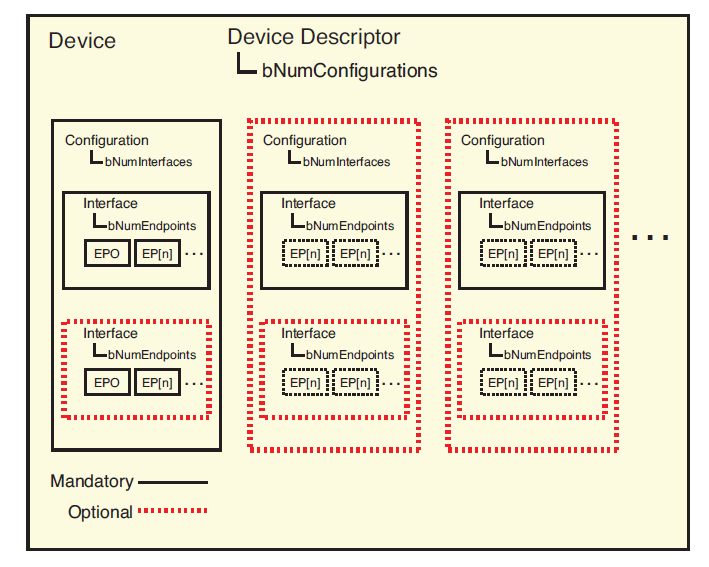
One of USB’s most successful features is that it is dynamic and truly hot-swappable.

Devices can be plugged into the USB bus at any time. Software running on the computer

(Linux, of course) that contains the host controller is responsible for configuring the USB devices when they appear in the topology.

**0.1.2 USB Logical Topology**

To better understand the software components and data flow in a USB system, it is useful to understand USB’s logical topology. Figure 0-2 shows the logical makeup of a hypothetical USB device.

**FIGURE 18-2 USB device functional block diagram**

Each USB device has a number of descriptors1 that allow software to discover capabilities and configure functionality. Every device must have a single device descriptor, which contains information such as manufacturer (idVendor), product (idProduct), serial number (iSerialNumber), and the number of configurations (bNumConfigurations). The identifiers in parentheses are the actual field names referenced in the USB 2.0 specification.

Every configuration identified in the device descriptor has a configuration descriptor.

The configuration descriptor contains the number of interfaces (bNumInterfaces) available for each configuration and also indicates the maximum power required when operated in this configuration (bMaxPower). Most often, a USB device contains only a single configuration. However, some devices may have high and low power modes, or even different functions available in a single device. These types of devices may contain multiple configurations. Plug your iPod into a USB host, and you will see an example of multiple configurations, interfaces, and endpoints!

Each interface described by a configuration descriptor has an interface descriptor.

The interface descriptor contains a field specifying how many endpoints a device has

(bNumEndpoints). Endpoint 0 is always assumed to exist and is not included in the interface count. The interface descriptor also includes information describing the interface class, subclass, and protocol as defined by the USB specifications. The USB endpoint is the actual logical element that software communicates with during operation of the USB device. Each endpoint described by an interface descriptor (excluding endpoint 0) contains an endpoint descriptor. The endpoint descriptor defines the endpoint’s communication parameters, including the endpoint address and various endpoint attributes describing the characteristics of the data transfer from each endpoint. Later in this chapter, we introduce the utility lsusb, which allows you to read these descriptors.

**0.1.3 USB Connectors**

Unless you are a USB expert, the variety of USB connector and cable configurations can be confusing. The most familiar connector type, defined by the original specification, is the USB A connector. This is the familiar rectangular connector most commonly found on laptop and desktop PCs. The plug end of the USB A connector, by definition, always points upstream, toward the host controller/root hub. Figure 0-3 shows a standard USB A plug.

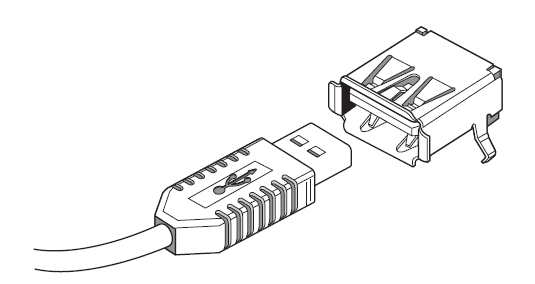
****

FIGURE 0-3 USB A plug

A peripheral (slave) device such as a printer or scanner often has the USB B receptacle and accepts a USB B plug, also defined by the original USB specification. A common cable suitable for connection between a host (such as a PC) and a peripheral (such as a printer) has an A plug on one end and a B plug on the other. It is more narrow than the A plug and has more of a D shape than rectangular. The USB B connector by definition always points downstream, or away from the host controller/root hub. Figure 0-4 shows a USB B plug.

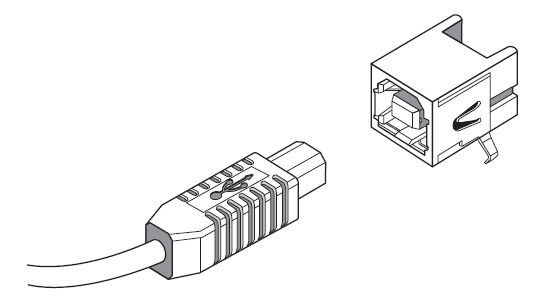
****

FIGURE 18-4 USB B plug

There are also a couple of miniature plug configurations. Smaller form factor devices such as cell phones and PDAs drove the requirement for even smaller plugs and receptacles. The USB Mini-A connector has been made obsolete by the specifications, although it is still in use. The Mini-B connector is widely used on small peripheral devices.

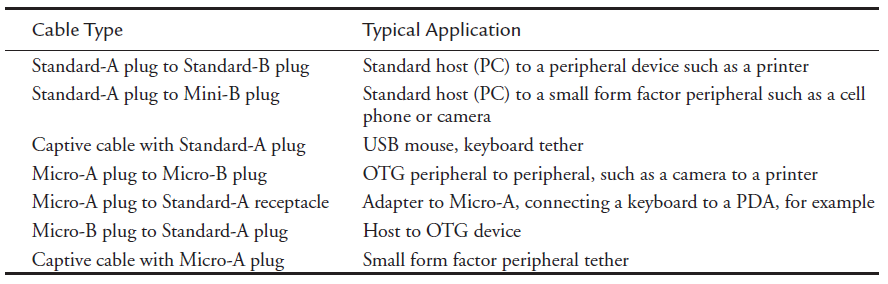
The Micro-USB specification defines three additional connectors—a Micro-B plug and receptacle, a Micro-AB receptacle, and a Micro-A plug. The Micro-AB receptacle is for use only on USB On-The-Go (OTG) appliances, discussed later.

To summarize the standard A and B connectors, the A receptacle is always on the host side (the A plug always points upstream), and the B connector is always on the peripheral side (the B plug always points downstream.) Table 0-1 summarizes these characteristics.

**0.1.5 USB Cable Assemblies**

The latest USB specifications define the cable assemblies listed in Table 18-2 as the only compliant cables.

TABLE 0-2 USB Cables and Typical Applications

****

It is possible to purchase other types of cable assemblies that are not listed here.

1. **USB Architecture**

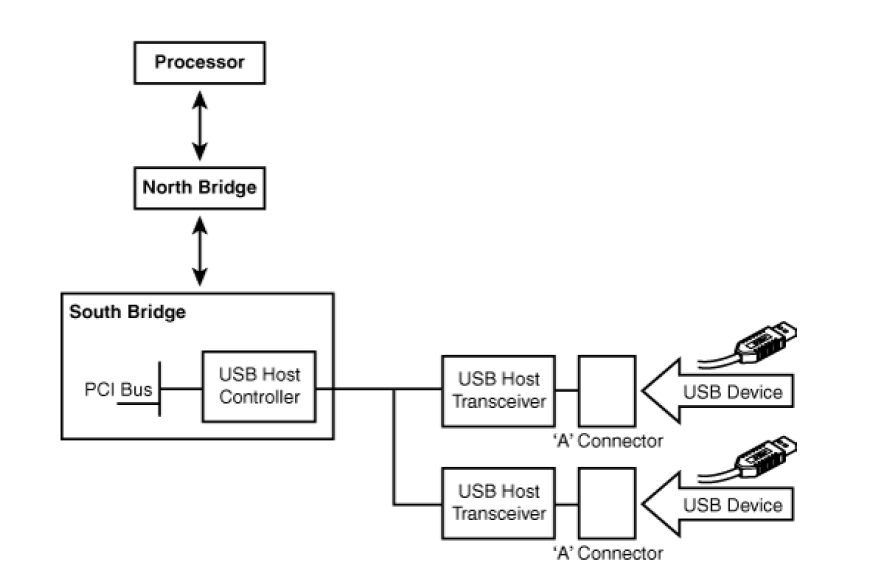
USB is a master-slave protocol where a host controller communicates with client devices. Figure 1 shows USB in the PC environment. The USB host controller is part of the South Bridge chipset and communicates with the processor over the PCI bus.

Figure 2 illustrates USB on an embedded device. The SoC(System On Chip) 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.

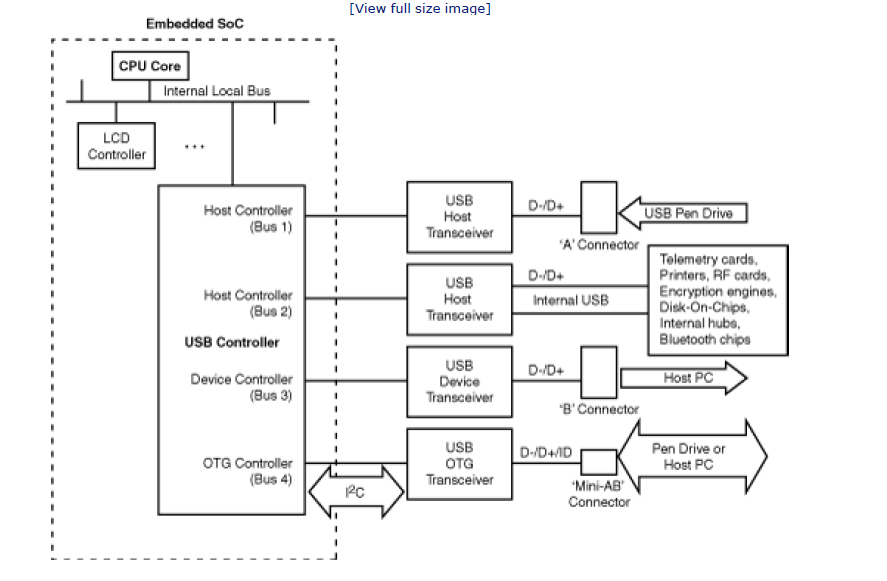


**Figure 1. USB in the PC environment.**

● 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 I2C. 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.

**1.1 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.



**Figure 2. USB on an embedded system.**

**1.2 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.

* 1. **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(같은 시간간격의) 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.

* 1. **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 I2C on some counts and PCI on others as summarized in Table 1. USB's device addressing is similar to I2C, while it supports hotplugging like PCI. USB device addresses, like standard I2C, do not consume a portion of the CPU's address space. Rather, they reside in a private space ranging from 1 to 127.

1. **Linux-USB Subsystem**

Look at Figure 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, 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 .

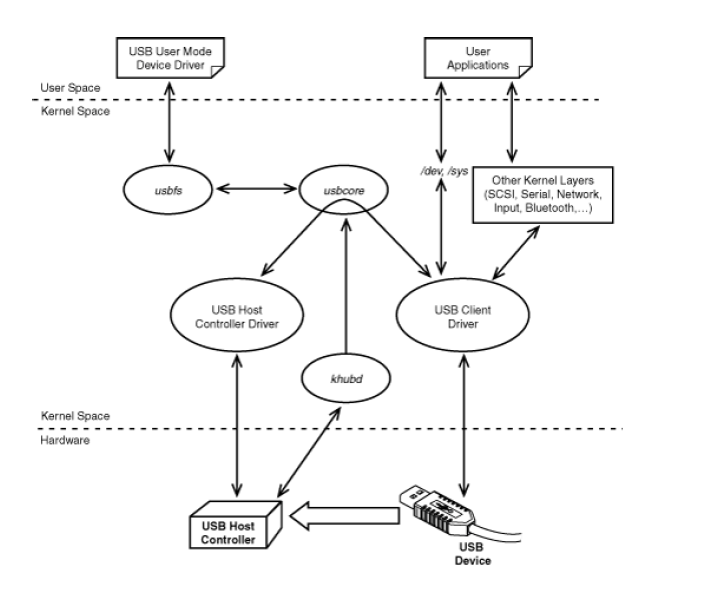
The khubd thread is asleep by default. The hub driver wakes khubd whenever it detects a

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.

**Figure 3. The Linux-USB subsystem.**



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 3. To drive USB-Bluetooth keyboards, the stakeholders are fourfold: the USB subsystem, the Bluetooth layer, the input subsystem, and the tty layer.

1. **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.

**3.1 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.

**3.2 USB Request Blocks**

*USB Request Block* (URB) is the centerpiece of the USB data transfer mechanism. 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! \*/

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.

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.

* 1. **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*.

* 1. **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 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 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.

1. **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(해독하다) 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.

**Listing 1. Print All USB Endpoint Addresses on a Device**



**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.

1. **Device Example: Telemetry(원격측정공학) Card**

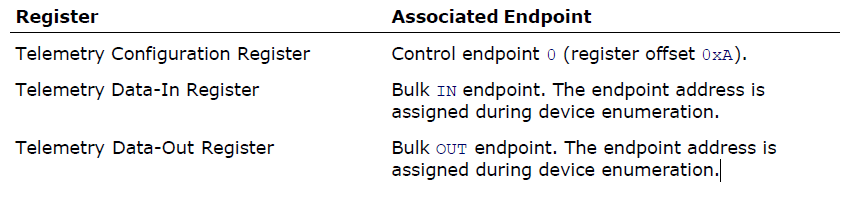
Now that you know the basics of Linux-USB, it's time to look at an example device. Consider a system equipped with a telemetry card connected to the processor via internal USB, as shown in bus 2 of Figure 2. The card acquires data from a remote device and ferries it to the processor over USB. An example 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 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 2. Register Space in the Telemetry Card**

****

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.

* 1. **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 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 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 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 3 \*/

.disconnect = tele\_disconnect, /\* See Listing 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. 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 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 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(수집하다) 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 3, 4, and 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. 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 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.\*/

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;

/\* ... \*/

}

* 1. **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 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 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().
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 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.

* 1. **Data Transfer**

Listing 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 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 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);

}

1. **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.

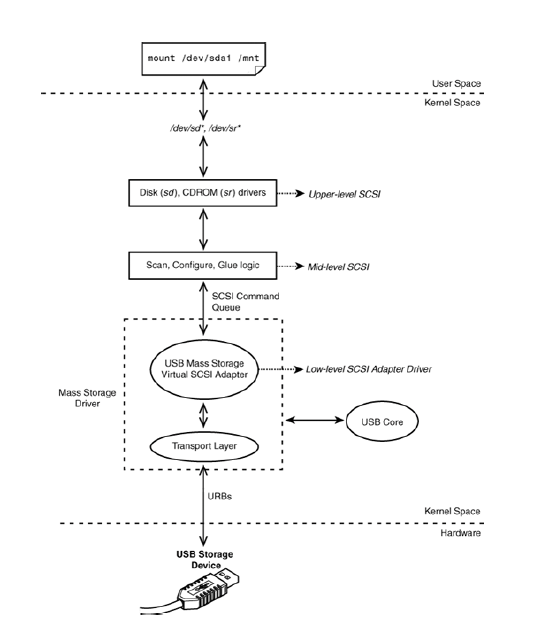
* 1. **Mass Storage**

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 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 4. USB mass storage and SCSI.**

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 6 shows the necessary code changes in the form of a source patch to the 2.6.23.1 kernel tree.

**Listing 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 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 upperlevel 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 usbstorage kernel thread handles using appropriate URBs.

Now that you know the basics, let's dissect Listing 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 6 adds a magic string, "disk," to us\_data. The *usbfs* modification in Listing 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 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=usbstorage

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=usbstorage

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=usbstorage

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. 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.

* 1. **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. Figure 5 illustrates how the USB-Serial layer fits into the kernel's serial framework.

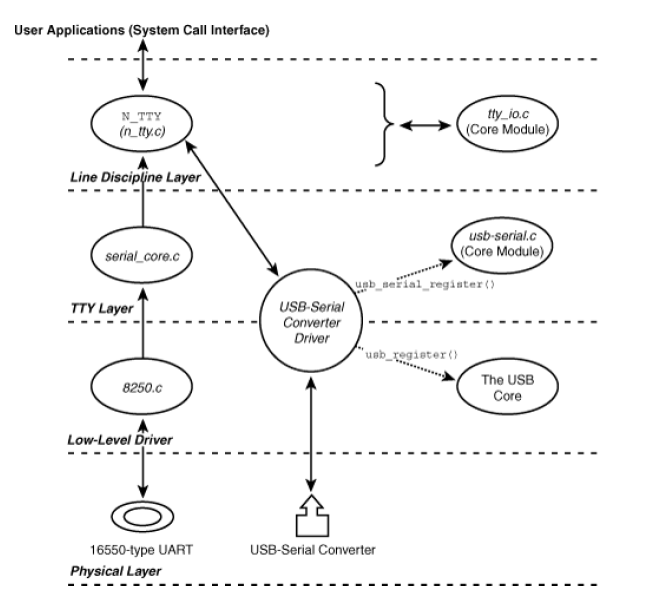
A USB-serial driver is similar to other USB client drivers except that it avails the services of a USB-Serial core in addition to the USB core. The USB-Serial core provides the following:

● 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.

**Figure 5. The USB-Serial layer.**



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 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 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);

/\* ... \*/

}

* 1. **Other Devices**
     1. **Human Interface Devices**

Devices such as keyboards and mice are called *human interface devices* (HIDs).

* + 1. **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.

1. **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 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.

1. **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.

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*.

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.

1. **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.*

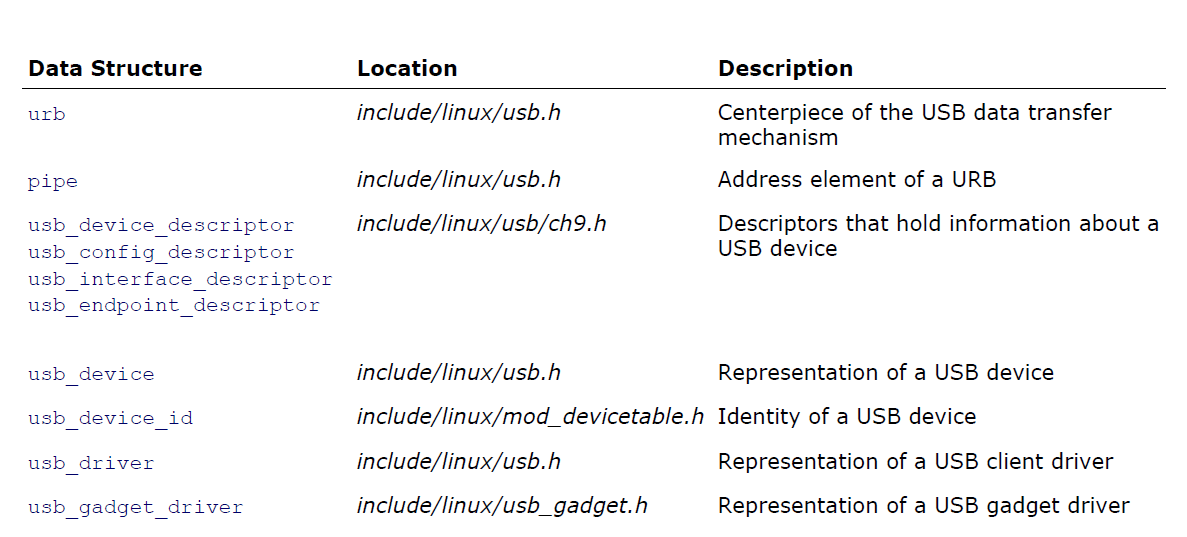
The usbmon tracer is in *drivers/usb/mon/.* Look inside *Documentation/usb/* for Linux-USB documentation. USB class drivers stay in various subdirectories under *drivers/usb/.* The mass storage driver *drivers/usb/storage/*, in tandem with the SCSI subsystem *drivers/scsi/*, implements the USB mass storage 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.

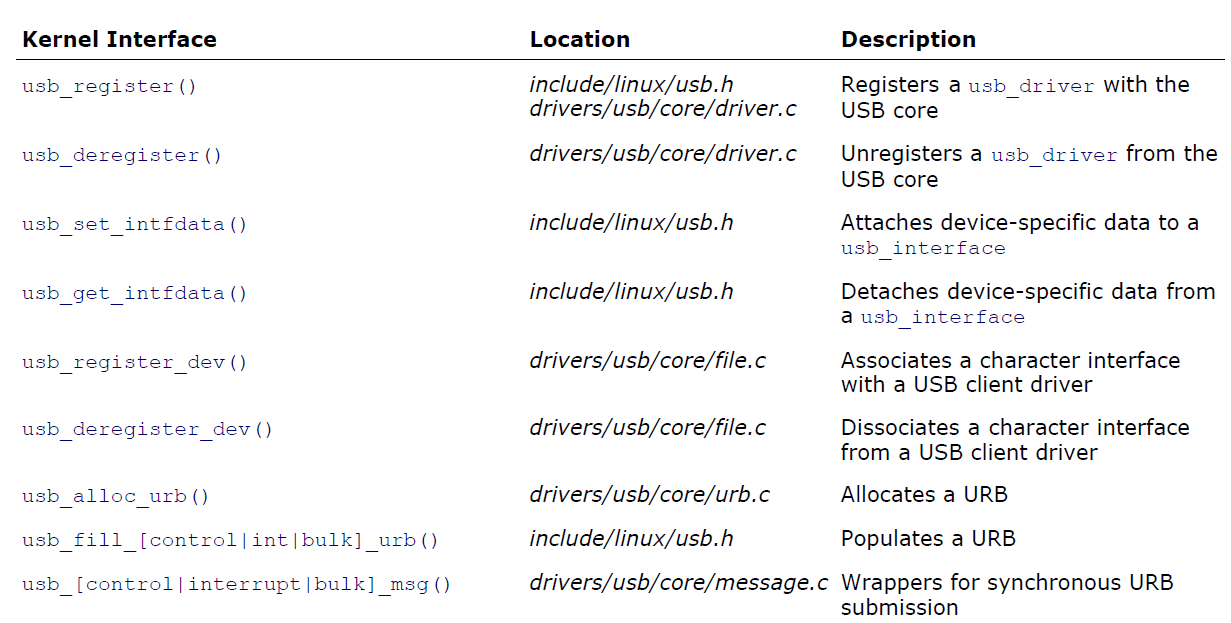
The USB gadget subsystem is in *drivers/usb/gadget/.* This directory contains USB device controller drivers, and gadget drivers for mass storage (*file\_storage.c*), serial converters (*serial.c*), and Ethernet networking (*ether.c*).

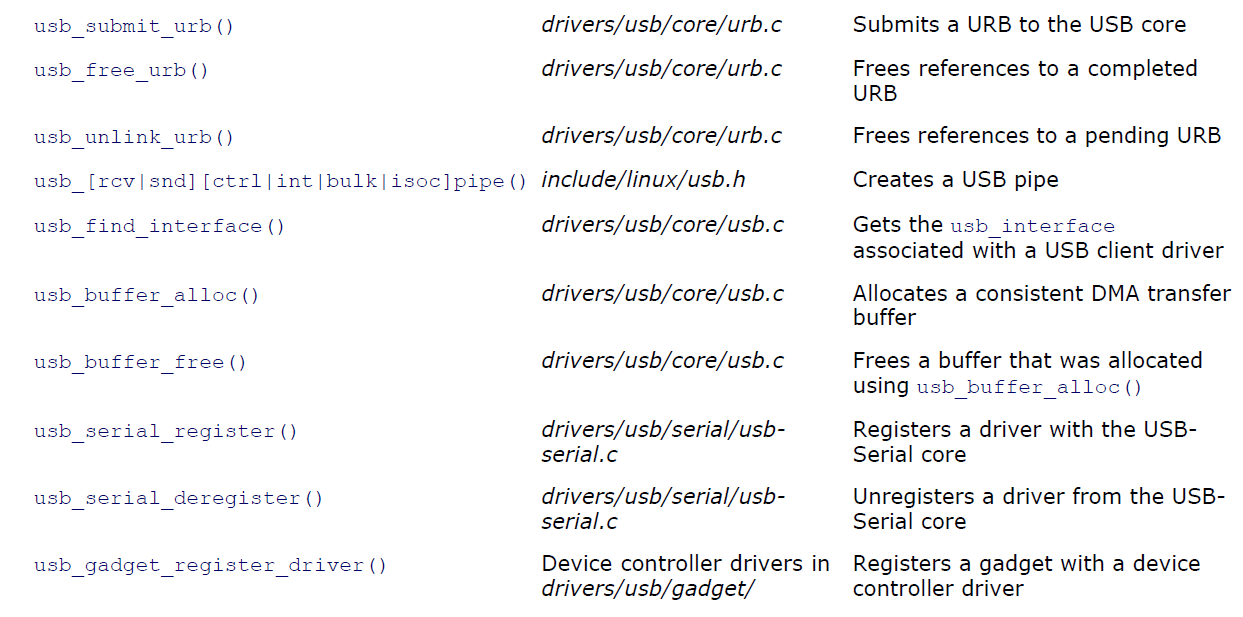
Table 3 contains the main data structures used in this chapter and their location in the source tree. Table 4 lists the main kernel programming interfaces that you used in this chapter along with the location of their definitions.

**Table 3. Summary of Data Structures**



**Table 4. Summary of Kernel Programming Interfaces**





**지침으로 삼은 문헌**

위대한 수령 **김일성**동지의 로작

1. **《과학연구사업에서 새로운 전환을 일으킬데 대하여》**

**(《김일성저작집》 제37권, 392~408페지)**

위대한 령도자 **김정일**동지의 로작

1. **《새 세기, 21세기는 정보산업의 시대이다.》**

**(《김정일선집》 제15권, 110~117페지)**

1. **《과학기술을 더욱 발전시킬데 대하여》**

**(《김정일선집》 제8권, 240~247페지)**

1. **《교육사업을 더욱 발전시킬데 대하여》**

**(《주체혁명위업의 완성을 위하여》 제5권, 177~189페지)**

**참고문헌**

1. Intel,2002《Enhanced Host Controller Interface Specification for Universal Serial Bus》
2. Sreekrishnan Venkateswaran,2008,《Essential Linux Device Drivers》
3. Christopher Hallinan,2010,《Hallinan\_Embedded\_Linux\_Primer\_2nd》
4. MindShare,2001,《USB System Architecture (USB 2.0)》

**맺는말**

지금까지 우리는 매몰형체계에서의 USB부분에 대한 체계자원분석을 진행하였다. 지금USB가 세계적으로 널리 쓰이고 있는 조건에서 매몰형체계에서의USB부분에 대한 체계자원분석은 리론분야에서뿐아니라 실천적으로도 매우 중요한 의의를 가진다. 사용자콤퓨터의 입출력대면부의 약점을 극복하기 위하여 설계된 USB를 지금은 거의 모든 전자장치들에서 쉽게 찾아볼수 있다. 수자식사진기, 인쇄기, 손전화기, IP전화기는 물론 건반과 마우스는USB대면부를 가진 전형적인 장치의 실례들이다. 하지만 방금 렬거한 장치들은 일부분에 지나지 않는다. 지금은 간단한 전기기타까지도 USB포구를 가지고 있다.

일반적인 외부장치를 위해 특별히 제작된 입출력장치를 필요로 하던 시기는 이미 과거로 되여 버렸으며 새로운USB의 시기가 도래하여 현재는USB3.0이 출현하여 그 전송속도가 5Gbps에 달하였다. 그러므로 매몰형체계에서의 USB기능에 대한 분석은 중요한 자리를 차지하며 매몰형체계에 접속할수 있는 장치를 증가하여 체계의 능력을 높이는데서 아주 중요하다.

이번에 우리는 체계자원분석을 통하여 매몰형체계에서의USB부분에 대한 리해뿐아니라 일반적인 환경속에서의USB의 특성과 그 동작과정에 대한 확고한 리해를 가지게 되였다. 특히 USB에서 사용되는 여러가지 대면부규약들과 그에 기초한USB대면부지식, 련결방식과 동작원리들은 우리가 오래전부터 관심해오던것들로써 이번에 그에 대한 완벽한 리해를 가지게 됨으로써USB분야에 대한 지식에서의 커다란 진전을 가져오게 되였다.

우리는 이번 실습과제수행을 통하여 얻은 지식을 리용하여 매몰형체계에 련결되여 동작할수 있는 여러가지 장치들에 대한 추가를 비릇하여 실천에서 제기되는 문제들을 해결할수 있다는 새로운 자신심을 가지게 되였다.

우리는 발은 자기 땅에 붙이고 눈은 세계를 보라고 하신 경애하는 **김정일**대원수님의 말씀을 가슴깊이 새기고 새 세기 산업혁명을 앞장에서 떠메고 나갈 선군혁명의 믿음직한 골간으로 자신들을 튼튼히 준비해 나가기 위해 전공부문 학습을 꾸준히 해나갈 굳은 결의를 다지면서 이번 실습과제가 우리들의 전공학습에 큰 도움을 주었다고 생각한다.