PRENTICE HALL

PRENTICE HALL OPEN SOURCE SOFTWARE DEVELOPMENT SERIES

Embedded Linux Primer

Second Edition

A Practical Real-World Approach

Christopher Hallinan

Table of Contents

Chapter 18

Universal Serial Bus

In This Chapter

By anyone's measure, Universal Serial Bus (USB) has been wildly successful. USB was originally designed to overcome the shortcomings of the various I/O interfaces found on the PC architecture. Today it is difficult to find an electronic device at your local electronics superstore that does not have a USB port. Digital cameras, printers, cell phones, IP telephones, and, of course, keyboards and mice are typical examples of devices that have USB interfaces. However, the list is much longer than the average reader would guess. Even some of my guitar pedals have USB interfaces!

Gone are the days when you needed special-purpose input/output hardware for common devices. The promise of USB has actually been realized, unlike a host of other technologies that have come and gone. Indeed, as this second edition was being prepared, the first experimental Linux drivers for USB 3.0 had just been released. And it is notable that Linux was the first OS to have such support!

18.1 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.

18.1.1 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 18-1 illustrates the basic topology.

Chapter 18. Universal Serial Bus. Embedded Linux Primer: A Practical, Real-World Approach, Second Edition, ISBN: 9780137061129

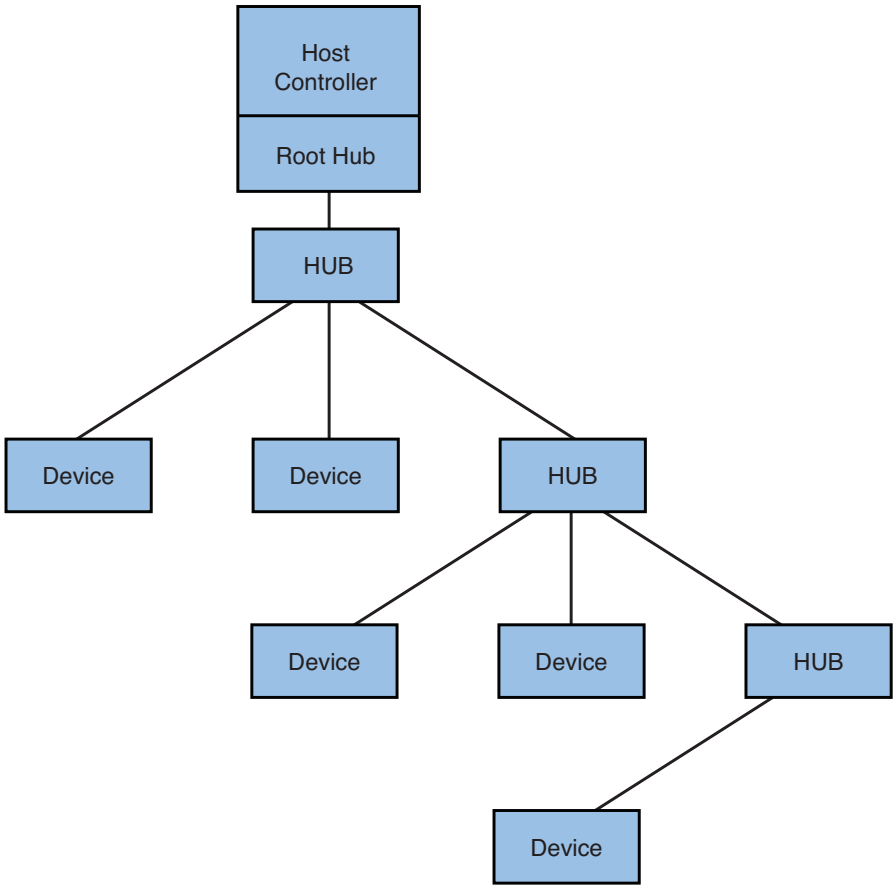

FIGURE 18-1 Simple USB topology

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 18-1 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.

18.1.2 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 18-2 shows the logical makeup of a hypothetical USB device.

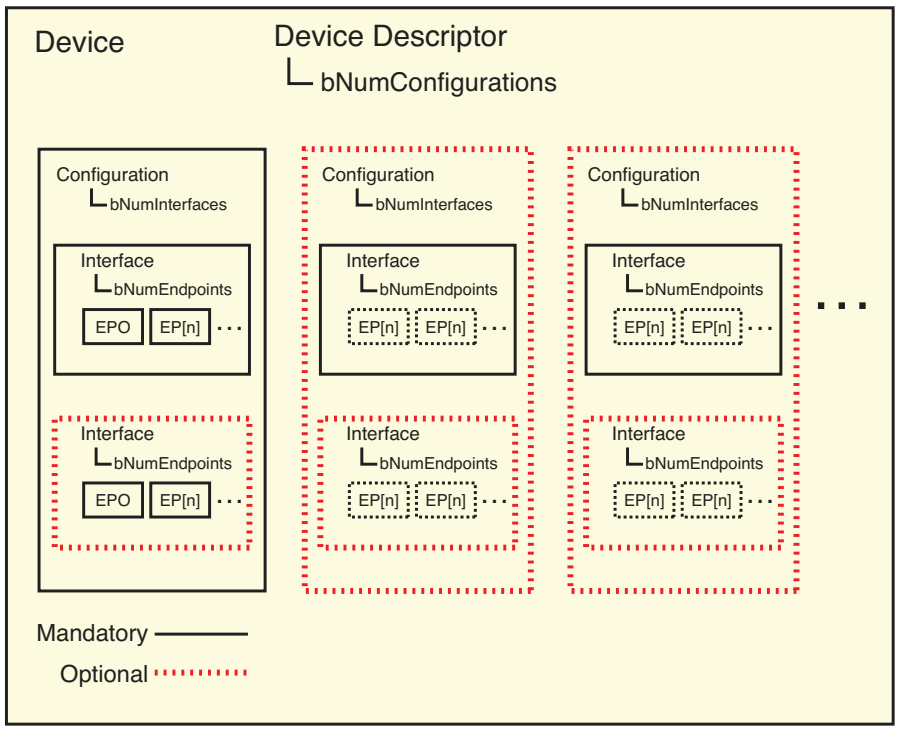

FIGURE 18-2 USB device functional block diagram

Each USB device has a number of descriptors[1] 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

[1] These descriptors are described in Chapter 9, "File Systems," of the USB 2.0 specification, referenced at the end of this chapter.

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.

More details can be found in the complete Universal Serial Bus specification referenced at the end of this chapter.

18.1.3 USB Revisions

USB 1.0 was introduced well over a decade ago. The current USB 2.0 specification document shows a revision history as early as November 1994, and that was already revision 0.7. The original specification called for a data transfer rate of 12 megabits per second (Mbps), with a low-speed rate defined as 1.5 Mbps. The 2.0 revision of the USB spec was finalized in April 2000.[2] USB 2.0 defined a high-speed data transfer rate of 480 Mbps. There is now a 3.0 revision of the USB specification, specifying

[2] According to the revision history in the current USB 2.0 specification.

data transfer rates into gigabits per second and adding yet another speed definition: SuperSpeed.

It can be difficult to remember the difference between full speed and high speed. Here is a summary:

USB 1.0 Low speed	1.5 Mbps
USB 1.0 Full speed	12 Mbps
USB 2.0 High speed	480 Mbps
USB 3.0 Super speed	5 Gbps

18.1.4 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 18-3 shows a standard USB A plug.

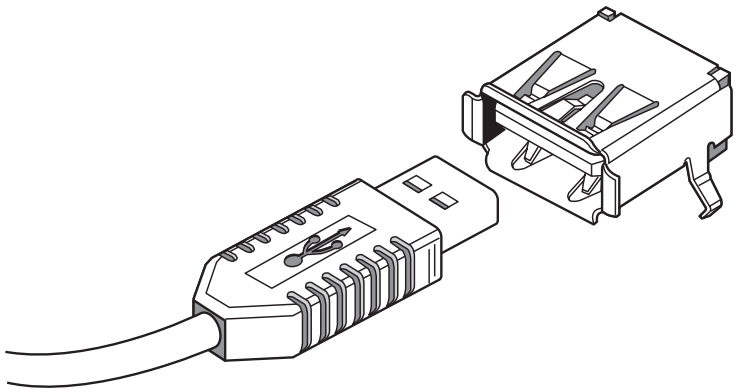

FIGURE 18-3 USB A plug

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 18-4 shows a USB B plug.

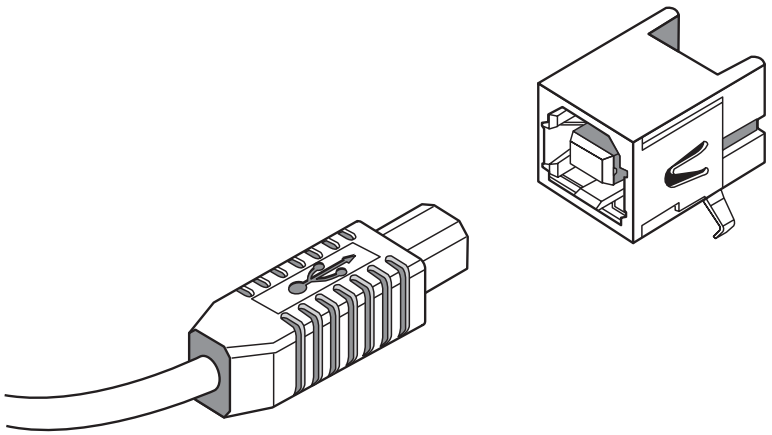

FIGURE 18-4 USB B plug

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 18-1 summarizes these characteristics.

TABLE 18-1 USB Connector Summary

Connector	Plug	Receptacle
A series	Points toward the host (or hub)	Functions as host (or hub) output
B series	Points toward the peripheral	Functions as peripheral (or hub) input

18.1.5 USB Cable Assemblies

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

TABLE 18-2 USB Cables and Typical Applications

Cable Type	Typical Application
Standard-A plug to Standard-B plug	Standard host (PC) to a peripheral device such as a printer
Standard-A plug to Mini-B plug	Standard host (PC) to a small form factor peripheral such as a cell phone or camera
Captive cable with Standard-A plug	USB mouse, keyboard tether
Micro-A plug to Micro-B plug	OTG peripheral to peripheral, such as a camera to a printer
Micro-A plug to Standard-A receptacle	Adapter to Micro-A, connecting a keyboard to a PDA, for example
Micro-B plug to Standard-A plug	Host to OTG device
Captive cable with Micro-A plug	Small form factor peripheral tether

It is possible to purchase other types of cable assemblies that are not listed here. These are typically available to resolve special cases, such as obsolete (or misdesigned) hardware. For example, the BeagleBoard requires a special adapter with a Mini-A plug on one end and a USB-A receptacle on the other. The Mini-A plug has pins 4 and 5 shorted, which appears to be required by the transceiver on the BeagleBoard for it to be configured for host mode operation.

18.1.6 USB Modes

One of the more confusing aspects of USB to the uninitiated is the various modes of operation. We hear terms such as USB On-The-Go (OTG), gadget, and peripheral. Hopefully we have taken the confusion out of the different speed ratings and connector types. This section briefly covers the various modes of USB controllers and devices.

The USB controller and receptacle on a standard desktop PC is called a USB host. Because USB is a master-slave bus, by definition one node on a USB bus must provide the master functionality. This is the host. You will hear it referred to as host mode or simply the USB host. The host controller, in association with the root hub, is the low-level piece of hardware that operates the USB master/slave bus protocol. The USB host is always the bus master.

The other end of a simple USB network is the device end. Sometimes this is called a USB gadget.[3] Gadget functionality within the Linux kernel simply refers to the ability to operate as a device in slave mode. Once you enter the embedded world, you no longer assume that the processing device acts as a host. For example, you might have a Linux-powered smart phone with a USB connector and a USB controller designed to operate a device, or the slave end of a USB link.

Many embedded systems need to operate in both master (host) and slave (device) modes from the same controller. A personal digital assistant (PDA) is a good example of this requirement. Your PDA might have a requirement to connect to a USB host, such as your desktop PC in order to synchronize its data to a master database located on your PC, or to get a software update. On the other hand, you might want to connect a USB keyboard to your PDA to facilitate typing. These USB devices must be able to operate in both host and device modes. This is called USB On-The-Go (OTG). As an added benefit, the USB OTG specification allows switching roles on-the-fly, without having to pull out a plug and reconnect it in the opposite mode.

18.2 Configuring USB

Like most functionality not directly related to the core Linux kernel operations, USB functionality is optional and must be enabled in your kernel configuration. As with most other auxiliary functionality, USB can be compiled into the kernel image or can be configured as loadable modules for dynamically loading into a booted kernel. For purposes of this text, we will use loadable modules, because this helps bring visibility to which components are required for specific functionality.

One of the barriers to properly configuring USB is simply the volume of options in a typical kernel config for USB. On a very recent kernel, configured as allmodconfig,[4] almost 300 different device driver modules (`*.ko` files) were related to USB. Of course, many of those drivers are for particular USB devices, but looking through the configuration options for USB, it is clear that a little know-how will go a long way!

In the following examples, we'll look at the Freescale Semiconductor i.MX31 Applications Processor on the i.MX31 PDK development platform. It makes for an interesting example, because it contains three host controllers that can be configured in a variety of operational modes.

[3] The Linux USB developers coined the term "gadget" to avoid confusion with the overused term "device," as in device drivers.

[4] The Linux kernel make target, mostly for build testing, that builds a configuration with all modules where possible.

Figure 18-5 shows a portion of the USB configuration options from a recent Linux kernel configured for the ARM architecture and the Freescale Semiconductor i.MX31 PDK reference board.

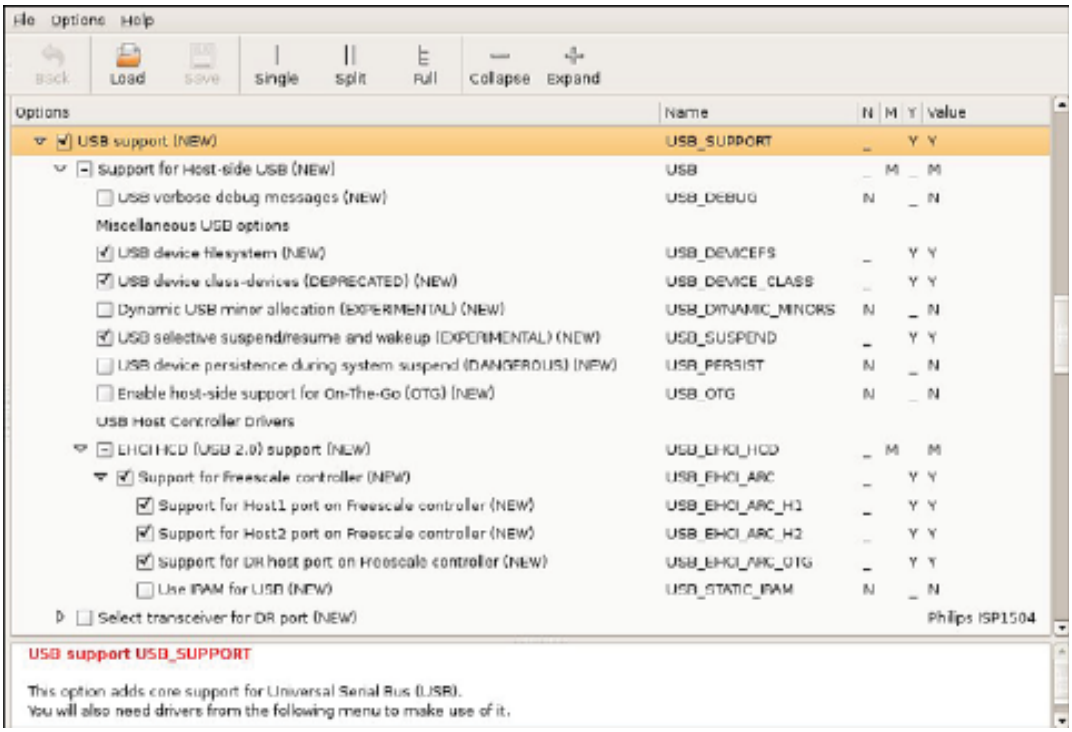

FIGURE 18-5 Part of the USB configuration for i.MX31

There are many more options than can be shown here. These are just the initial ones. You must first select `USB_SUPPORT` to see any further USB configuration options. Notice that USB is selected as M (module), meaning that the kernel build system will compile the USB drivers as loadable modules.

Let's look at the minimal configuration to get USB operational on the i.MX31. For all USB configurations, the module called `usbcore.ko` is required. It is selected automatically whenever `CONFIG_USB` is selected.[5] You can see this easily by looking at the makefile for the core part of the USB driver support, shown in Listing 18-1. This makefile is found at `.../drivers/usb/core/Makefile`.[6]

[5] You may recall from earlier chapters that actual configuration variables always start with `CONFIG_`, but this is omitted from the GUIs that we use to configure the kernel. For example, `USB_SUPPORT` in Figure 18-3 actually translates to `CONFIG_USB_SUPPORT` in the kernel's configuration file, `.config`.

[6] Remember that this notation describes the location of the file from your top-level kernel directory.

LISTING 18-1 Makefile for the USB Core

```
#
# Makefile for USB Core files and filesystem
#

usbcore-objs    := usb.o hub.o hcd.o urb.o message.o driver.o \
                   config.o file.o buffer.o sysfs.o endpoint.o \
                   devio.o notify.o generic.o quirks.o

ifeq ($(CONFIG_PCI),y)
        usbcore-objs    += hcd-pci.o
endif

ifeq ($(CONFIG_USB_DEVICEFS),y)
        usbcore-objs    += inode.o devices.o
endif

obj-$(CONFIG_USB)       += usbcore.o

ifeq ($(CONFIG_USB_DEBUG),y)
EXTRA_CFLAGS += -DDEBUG
endif
```

Looking at the various makefiles that drive the kernel's build system is always a great way to figure out what components are required, or the reverse—what configuration options are required for a specific function. For example, in Listing 18-1, you can see that we need to enable `CONFIG_USB` to get `usbcore` included in the build. As we learned in Chapter 4, "The Linux Kernel: A Different Perspective," if `CONFIG_USB` is set to `m`, this device driver is compiled as a loadable kernel module and provides USB host support.

18.2.1 USB Initialization

Booting the i.MX31 into a minimal configuration, we can load the `usbcore` module as follows:

```
# modprobe usbcore
```

This loads the USB core module, which handles common functions for the rest of the USB collection of drivers. These functions include housekeeping such as register

and deregister functions for various elements and provide an interface to drive USB hardware. This takes much of the complexity out of writing drivers for USB hardware. You can see the public symbols by using your cross-nm utility. You learned about `nm` in Chapter 13, "Development Tools." Here is an example of using your cross-nm to display USB registration functions:

```
$ arm_v6_vfp_le-nm usbcore.ko  | grep T.*_register
0000adc0 T usb_register_dev
000094b0 T usb_register_device_driver
00009560 T usb_register_driver
0000fcf8 T usb_register_notify
```

`usbcore` also contains functions for buffer handling; support for usbfs, the hub class driver; and many other functions dealing with communication with the underlying USB controller.

`usbcore` is not very useful by itself. It does not contain any low-level host controller drivers. These are separate modules, and they can differ depending on the hardware you are using. For example, the BeagleBoard, which contains the TI OMAP3530 Applications Processor, has a built-in dual-role[7] controller. This controller is the Inventra™ USB Hi-Speed Dual-Role Controller. The driver for this host controller is called `musb_hdrc`.

On many platforms, the USB host controller is designed to conform to the Enhanced Host Controller Interface (EHCI),[8] which describes the register-level interface for an industry-standard USB host controller. In this case, the driver, when compiled as a loadable module, is called `ehci-hcd.ko`. Let's see what happens when we load that driver on the Freescale i.MX31, as shown in Listing 18-2.

LISTING 18-2 Installing the USB Host Controller Driver on i.MX31

```
root@imx31:~# modprobe ehci-hcd
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb
fsl-ehci fsl-ehci.0: Freescale On-Chip EHCI Host Controller
fsl-ehci fsl-ehci.0: new USB bus registered, assigned bus number 1
fsl-ehci fsl-ehci.0: irq 35, io mem 0x43f88200
```

[7] Dual-role controllers can act as USB hosts or USB peripherals and can switch between the two functions while in operation. As you will quickly learn, this is referred to as USB On-The-Go (OTG).

[8] A reference to this EHCI specification appears at the end of this chapter.

LISTING 18-2 Continued

```
fsl-ehci fsl-ehci.0: USB 2.0 started, EHCI 1.00, driver 10 Dec 2004
usb usb1: configuration #1 chosen from 1 choice
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 1 port detected
fsl-ehci fsl-ehci.1: Freescale On-Chip EHCI Host Controller
fsl-ehci fsl-ehci.1: new USB bus registered, assigned bus number 2
fsl-ehci fsl-ehci.1: irq 36, io mem 0x43f88400
fsl-ehci fsl-ehci.1: USB 2.0 started, EHCI 1.00, driver 10 Dec 2004
usb usb2: configuration #1 chosen from 1 choice
hub 2-0:1.0: USB hub found
hub 2-0:1.0: 1 port detected
fsl-ehci fsl-ehci.2: Freescale On-Chip EHCI Host Controller
fsl-ehci fsl-ehci.2: new USB bus registered, assigned bus number 3
fsl-ehci fsl-ehci.2: irq 37, io mem 0x43f88000
fsl-ehci fsl-ehci.2: USB 2.0 started, EHCI 1.00, driver 10 Dec 2004
usb usb3: configuration #1 chosen from 1 choice
hub 3-0:1.0: USB hub found
hub 3-0:1.0: 1 port detected
```

There is much useful information here. Note that we had the console log level set so that these messages appear on the console. First we see `usbcore` registration functions in action. We see registration messages for the usbfs interface driver, the hub interface driver, and the EHCI (usb host mode) driver itself. You will learn about usbfs shortly.

Next we see the three host controllers being initialized. Recall that the Freescale i.MX31 Application Processor contains three separate USB controllers. We see these controllers being enumerated as `fsl-ehci.0`, `fsl-ehci.1`, and `fsl-ehci.2`. You can see the interrupt (IRQ) assignment and the base address of the register file that is associated with each USB controller. Notice that in each case the root hub is enumerated. The root hub is a fundamental component of the USB architecture and always is associated with the host controller. It is required in order to connect peripheral devices (hubs or other USB devices) to the host controller.

It is worth noting that we did not need to separately install `usbcore`, as we did at the start of this section. `modprobe` understands how to determine required dependencies for a given module. Simply issue the following command:

```
# modprobe ehci-hcd
```

This loads its dependency, `usbcore`, before loading the `ehci-hcd` driver. This mechanism is explained in Chapter 8, "Device Driver Basics." In fact, on many platforms that use the udev system, these steps are automated by udev. This is covered in Chapter 19.

18.3 sysfs and USB Device Naming

As described in Chapter 9, "File Systems," the sysfs file system is basically a view of kernel objects, or *kobjects*. Each USB device is represented in sysfs. Look at `/sys/bus/usb/devices` on any USB-enabled system. Listing 18-3 shows how it looks on my Freescale i.MX31 PDK board. Note that I have truncated the `ls -l` format to fit the page width by removing all the columns except the filename.

LISTING 18-3 Output of `/sys/bus/usb/devices` on i.MX31

```
root@imx31:~# ls -l /sys/bus/usb/devices/
total 0
1-0:1.0 -> ../../../devices/platform/fsl-ehci.0/usb1/1-0:1.0
2-0:1.0 -> ../../../devices/platform/fsl-ehci.1/usb2/2-0:1.0
2-1 -> ../../../devices/platform/fsl-ehci.1/usb2/2-1
2-1.1 -> ../../../devices/platform/fsl-ehci.1/usb2/2-1/2-1.1
2-1.1:1.0 -> ../../../devices/platform/fsl-ehci.1/usb2/2-1/2-1.1/2-1.1:1.0
2-1:1.0 -> ../../../devices/platform/fsl-ehci.1/usb2/2-1/2-1:1.0
3-0:1.0 -> ../../../devices/platform/fsl-ehci.2/usb3/3-0:1.0
usb1 -> ../../../devices/platform/fsl-ehci.0/usb1
usb2 -> ../../../devices/platform/fsl-ehci.1/usb2
usb3 -> ../../../devices/platform/fsl-ehci.2/usb3
```

All the entries in `/sys/bus/usb/devices` are links to other portions of the sysfs hierarchy. Notice the numeric names, such as `1-0:1.0`. In this naming scheme, the first numeral is the root hub, or the top level of the USB hierarchy for this particular bus. The second number is the port number that a given device is connected to. The third number (the one after the colon) is the configuration number of the USB device, followed by the device's interface number. The configuration and interface elements are logical components of all USB devices, as described in Section 18.1.2.

To summarize:

```
1-0:1.0
| | | |----- interface number
| | |------- configuration number
| |--------- hub port
|----------- root_hub
```

If additional hubs are added to the topology, they are added to the device name by adding a dot (.) followed by the hub port number, chained to the upstream port. For example, if we add a hub to the topology just shown, we might end up with `1-0.2:1.0`. This assumes we have a downstream hub with at least two ports, with the device plugged into port number 2.

If you examine the content of each directory linked, you can determine what component is being referenced. For example, the first entry in Listing 18-3 represents a logical USB interface. You can tell this by looking at the files (also called sysfs attributes) pointed to by the symbolic link `1-0:1.0`. It contains entries from the interface descriptor, such as `bInterfaceNumber` and `bNumEndpoints`, both elements of the USB 2.0 interface descriptor. The first two entries in Listing 18-3 represent the single interface associated with two of the internal USB controllers/root hubs in the Freescale i.MX31.

The third entry in Listing 18-3 (`2-1`) represents an external hub connected to the second USB bus. In particular, it is a representation of the `struct usb_device` for this hub.

The three entries in Listing 18-3 starting with `usb` represent the buses themselves. You can see that this system has three buses, because the Freescale i.MX31 processor has three USB controllers. The bus number is the final digit in the `usb` name. The USB buses are numbered starting with 1.

In summary, here are some examples:

- `3-0:1.0` represents an interface (`struct usb_interface`) connected to bus 3. This is the root hub interface for bus 3.
- `2-1` represents a device (`struct usb_device`). In the configuration from this example, this is an external hub connected to the root hub on bus 2.
- `2-1.3` represents a downstream device (`struct usb_device`) from device `2-1`, connected to its port 3.

- `usb2` represents a USB bus, number 2 (also a `struct usb_device`).
- `2-1.3.4:1.0` represents an interface (my iPod) running in configuration 1, connected to port 4 of its parent hub, which is connected to port 3 of its parent and then to the root hub on bus 2!

18.4 Useful USB Tools

A number of useful tools and utilities can help you better understand your system, configure drivers, and get detailed information about your USB subsystem. This section introduces them.

18.4.1 USB File System

The USB File System (USBFS) is another type of virtual file system. It is not available until you mount it. Some Linux systems automatically mount USBFS, but many do not. For you to use USBFS, it must also be enabled in your kernel. Select CONFIG_USB_DEVICEFS under USB Support in your kernel configuration. Note that this system is deprecated but is still included in recent kernels and is useful for understanding your USB system configuration. Because several utilities depend on it, it will likely be around for some time.

After your kernel is configured for USB_DEVICEFS (most Linux distributions have this enabled by default), you must mount this virtual file system to use it:

```
# mount -t usbfs usbfs /proc/bus/usb
```

After it is enabled, you should get a listing similar to Listing 18-4.

LISTING 18-4 Directory Listing: `/proc/bus/usb`

```
root@imx31:~# ls -l /proc/bus/usb
total 0
dr-xr-xr-x 2 root root 0 Jun  7 14:00 001
-r--r--r-- 1 root root 0 Jun  7 14:36 devices
```

After the USBFS is mounted, you can get a human-readable listing of the devices found in the USB topology, as shown in Listing 18-5.

LISTING 18-5 Output of /proc/bus/usb/devices

```
root@imx31:~# cat /proc/bus/usb/devices
T:  Bus=03 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=480 MxCh= 1
B:  Alloc=  0/800 us ( 0%), #Int=  0, #Iso=  0
D:  Ver= 2.00 Cls=09(hub  ) Sub=00 Prot=01 MxPS=64 #Cfgs=  1
P:  Vendor=0000 ProdID=0000 Rev= 2.06
S:  Manufacturer=Linux 2.6.24-335-g47af517 ehci_hcd
S:  Product=Freescale On-Chip EHCI Host Controller
S:  SerialNumber=fsl-ehci.2
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr=  0mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   4 Ivl=256ms

T:  Bus=02 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=480 MxCh= 1
B:  Alloc=  0/800 us ( 0%), #Int=  1, #Iso=  0
D:  Ver= 2.00 Cls=09(hub  ) Sub=00 Prot=01 MxPS=64 #Cfgs=  1
P:  Vendor=0000 ProdID=0000 Rev= 2.06
S:  Manufacturer=Linux 2.6.24-335-g47af517 ehci_hcd
S:  Product=Freescale On-Chip EHCI Host Controller
S:  SerialNumber=fsl-ehci.1
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr=  0mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   4 Ivl=256ms

T:  Bus=02 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  2 Spd=480 MxCh= 4
D:  Ver= 2.00 Cls=09(hub  ) Sub=00 Prot=01 MxPS=64 #Cfgs=  1
P:  Vendor=05e3 ProdID=0608 Rev= 7.02
S:  Product=USB2.0 Hub
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr=100mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   1 Ivl=256ms

T:  Bus=01 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=480 MxCh= 1
B:  Alloc=  0/800 us ( 0%), #Int=  0, #Iso=  0
D:  Ver= 2.00 Cls=09(hub  ) Sub=00 Prot=01 MxPS=64 #Cfgs=  1
P:  Vendor=0000 ProdID=0000 Rev= 2.06
S:  Manufacturer=Linux 2.6.24-335-g47af517 ehci_hcd
S:  Product=Freescale On-Chip EHCI Host Controller
S:  SerialNumber=fsl-ehci.0
C:* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr=  0mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   4 Ivl=256ms
```

The format of this printout is documented in `.../Documentation/usb/proc_usb_info.txt` in the kernel source tree. In summary, each T line documents a new USB device. T stands for topology. The T line contains additional information that can be used to build a topological diagram of the current USB bus. Each T line contains, in order, the bus number (Bus), level in the hierarchy (Lev), parent level (Prnt), port on the parent device (Port), the count of devices at this level (Cnt), device number (Dev#), speed (Spd), and the maximum number of children (MxCh).

Additional lines describe bandwidth requirements (`B:`), device descriptor (`D:`), product ID (`P:`), string descriptors associated with the device (`S:`), configuration descriptor (`C:`), interface descriptor (`I:`), and endpoint descriptor (`E:`).

18.4.2 Using `usbview`

The most valuable way to use this information is by using a program that builds the bus topology with this information contained in it. Greg Kroah-Hartman wrote a program called `usbview` that is still available on many Linux distributions. It uses the GTK library to graphically display the USB topology containing the information retrieved from the usbfs file system. I have taken that program and removed the GTK stuff so that it can be run in text mode on an embedded system without graphics. It is available on this book's companion website. Search for `usbview-text`.

Listing 18-6 shows how the output looks from `usbview-text` when run on the Freescale i.MX31. The output is truncated to show only a single device—in this case, my iPod. This device contains multiple configurations and illustrates many of the concepts we've discussed up to now.

LISTING 18-6 Output of `usbview-text`

```
root@imx31:~# /tmp/usbview-text
Bus 2 Device 3  ******** New Device *********
Device Name: iPod
Manufacturer: Apple Inc.
Serial Number: 00xxxxxxxxxx
Speed: 480Mb/s (high)
USB Version:  2.00
Device Class: 00(>ifc )
Device Subclass: 00
Device Protocol: 00
Maximum Default Endpoint Size: 64
Number of Configurations: 2
```

LISTING 18-6 Continued

```
Vendor Id: 05ac
Product Id: 1261
Revision Number:  0.01

Config Number: 1
    Number of Interfaces: 1
    Attributes: c0
    MaxPower Needed: 500mA

    Interface Number: 0
        Name: usb-storage
        Alternate Number: 0
        Class: 08(stor.)
        Sub Class: 06
        Protocol: 50
        Number of Endpoints: 2

            Endpoint Address: 83
            Direction: in
            Attribute: 2
            Type: Bulk
            Max Packet Size: 512
            Interval: 0ms

            Endpoint Address: 02
            Direction: out
            Attribute: 2
            Type: Bulk
            Max Packet Size: 512
            Interval: 0ms

Config Number: 2
    Number of Interfaces: 3
    Attributes: c0
    MaxPower Needed: 500mA

    Interface Number: 0
        Name:
        Alternate Number: 0
        Class: 01(audio)
        Sub Class: 01
        Protocol: 00
```

LISTING 18-6 Continued

```
        Number of Endpoints: 0

    Interface Number: 1
        Name:
        Alternate Number: 0
        Class: 01(audio)
        Sub Class: 02
        Protocol: 00
        Number of Endpoints: 0

    Interface Number: 1
        Name:
        Alternate Number: 1
        Class: 01(audio)
        Sub Class: 02
        Protocol: 00
        Number of Endpoints: 1

            Endpoint Address: 81
            Direction: in
            Attribute: 1
            Type: Isoc
            Max Packet Size: 192
            Interval: 1ms

    Interface Number: 2
        Name:
        Alternate Number: 0
        Class: 03(HID  )
        Sub Class: 00
        Protocol: 00
        Number of Endpoints: 1

            Endpoint Address: 83
            Direction: in
            Attribute: 3
            Type: Int.
            Max Packet Size: 64
            Interval: 125us
```

This output lists the information from the various descriptors (device, configuration, interface, and endpoint descriptors) for the iPod in question. The first thing to notice is that it contains two configurations—representing a mass storage device and an audio recording and playback device.

Configuration 1 contains a single interface with two endpoints, both designed for bulk data transfer, one in each direction. These endpoints are for reading and storing data on the internal Flash as represented by the USB storage device.

Configuration 2 contains three interfaces. Interface 0 contains no endpoints. Interface 1 contains one endpoint, which is an isochronous interface. This transfer type is for streaming real-time data such as audio or video, which occupies a predetermined amount of bandwidth. Interface 2 contains a single endpoint of type Interrupt. This endpoint transfer type is designed for timely but reliable delivery of data, such as from a mouse or keyboard.

18.4.3 USB Utils (`lsusb`)

A package called usbutils provides a utility called `lsusb`, which provides functionality similar to `lspci`. `lsusb` makes use of `libusb`, which must be on your system before you can use it. Check to make sure that your embedded distribution contains both packages (most do).

`lsusb` allows you to enumerate all the USB buses in your system and display information about each device on those buses. Listing 18-7 displays the physical bus topology by passing the `-t` flag to `lsusb`.

LISTING 18-7 USB Bus Physical Topology

```
root@imx31:~# lsusb -t
/:  Bus 03.Port 1: Dev 1, Class=root_hub, Driver=fsl-ehci/1p, 480M
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=fsl-ehci/1p, 480M
    |__ Port 1: Dev 2, If 0, Class=hub, Driver=hub/4p, 480M
        |__ Port 1: Dev 3, If 0, Class=stor., Driver=usb-storage, 480M
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=fsl-ehci/1p, 480M
```

Here you can see all three buses of the i.MX31 with a hub plugged into the second bus and my iPod plugged into port 1 of that hub. You can also display the descriptors for all devices or target just a single device:

```
root@imx31:~# lsusb -s 2:3
Bus 002 Device 003: ID 05ac:1261 Apple, Inc. iPod Classic
```

This format of the `lsusb` command displays device number 3 from bus number 2. Adding the `-v` flag would dump all the descriptors for the device, which would produce far too much data for a listing here. I leave that as an exercise for you. If you have an iPod, that makes for interesting output, because it has two configurations and multiple interfaces and endpoints.

18.5 Common USB Subsystems

This section introduces the more common USB subsystems that you are likely to encounter. Most are easy to set up and use. Together with udev (covered in Chapter 19) they can be automatically configured and ready to use immediately after you plug in your favorite USB device.

The following sections detail a standard USB class as defined by the USB Class Specification Documents. The intent of a standardized USB class is that one common class driver can be used to support a wide variety of different vendors' devices. When the device manufacturer conforms to the class specifications, you can operate the device without a vendor-specific device driver, using only the class driver for the class.

18.5.1 USB Mass Storage Class

Probably the single most common USB "class" or subsystem used in embedded systems is USB mass storage. It enables external USB Flash drives and other peripherals with internal storage or a high-speed external disk drive such as the Western Digital My Book™. USB storage must be configured in your kernel. One of the more confusing aspects of USB storage is that it requires SCSI subsystem support. In fact, on more recent kernels, when you select kernel support for USB mass storage, CONFIG_SCSI and CONFIG_BLK_DEV_HD are selected automatically.

As usual, these modules can be statically compiled into the kernel or configured as dynamically loadable modules. In these examples, we will use modules because they better illustrate what components and, therefore, configuration are required to get things working properly. Of course, you should know by now that you need `usbcore` and `ehci-hcd` as the base set of USB drivers. (Note that this assumes that we are continuing to use the Freescale i.MX31 Applications Processor in these examples. Other boards may need a different host controller driver, such as BeagleBoard, which uses `musb_hdrc`.)

To recognize the USB storage class of device, we need to include the `usb_storage` driver. However, this driver depends on `scsi_mod`, so we need that also. If your system

is properly configured, `modprobe` should be able to detect the dependency and load `scsi_mod` for you when you load `usb_storage`. These dependencies are located in your `modules.dep` file, which is found in `/lib/modules/`uname -r`/modules.dep`. Simply search for `usb-storage`, and you will see its dependencies listed on the same line.

Let's see what this looks like. Listing 18-8 shows the results of inserting the `usb-storage` module.

LISTING 18-8 Modules for `usb-storage`

```
root@imx31:~# modprobe usb-storage
SCSI subsystem initialized
Initializing USB Mass Storage driver...
usbcore: registered new interface driver usb-storage
USB Mass Storage support registered.
root@imx31:~# lsmod
Module                  Size  Used by
usb_storage            35872  0
scsi_mod               97552  1 usb_storage
ehci_hcd               30836  0
usbcore               129752  3 usb_storage,ehci_hcd
```

Now if we plug a USB storage device into a hub port, it will be detected as a `usb_storage` device by the `usb-storage` driver:

```
usb 2-1.2: new high speed USB device using fsl-ehci and address 6
usb 2-1.2: configuration #1 chosen from 2 choices
scsi1 : SCSI emulation for USB Mass Storage devices
scsi 1:0:0:0: Direct-Access     Apple    iPod    1.62 PQ: 0 ANSI: 0
```

However, we need an additional driver in order to access the partitions on the USB storage device. This is where the SCSI emulation layer comes in. The `sd-mod` driver is responsible for handling SCSI disk devices. After we load this module, the disk device within the USB storage device is enumerated, together with any and all partitions on the device. Listing 18-9 displays the results.

LISTING 18-9 Adding the `sd-mod` Driver

```
root@imx31:~# modprobe sd_mod
sd 1:0:0:0: [sda] 19488471 4096-byte hardware sectors (79825 MB)
sd 1:0:0:0: [sda] Write Protect is off
sd 1:0:0:0: [sda] Assuming drive cache: write through
```

LISTING 18-9 Continued

```
sd 1:0:0:0: [sda] 19488471 4096-byte hardware sectors (79825 MB)
sd 1:0:0:0: [sda] Write Protect is off
sd 1:0:0:0: [sda] Assuming drive cache: write through
 sda: sda1
sd 1:0:0:0: [sda] Attached SCSI removable disk
root@imx31:~# lsmod
Module                  Size  Used by
sd_mod                 20720  0
usb_storage            35872  0
scsi_mod               97552  2 sd_mod,usb_storage
ehci_hcd               30836  0
usbcore               129752  3 usb_storage,ehci_hcd
```

Now we have everything we need to mount the disk partition and access its contents. We can see from Listing 18-9 that the single partition on this USB storage device was enumerated as sda1. Assuming that we have a device node `/dev/sda1` and kernel support for the file system type, we can now mount this device:

```
# mount /dev/sda1 /your/favorite/mount/point
```

When udev is installed and properly configured on your embedded device, all the device node creation is done automatically. We cover udev in detail in the next chapter. For completeness, we'll show you how to create this device node on your system if udev is not present. The sysfs file system contains an entry for sda1:

```
root@imx31:~# find /sys -name sda1
/sys/block/sda/sda1
```

This sysfs entry has an attribute called `dev`. This attribute lists the major and minor number that the kernel assigned to the `/dev/sda1` device. On my Freescale i.MX31, it is assigned major=8, minor=1. Using this information, we create a device node, and then we can mount and access the partition. Listing 18-10 contains the final results.

LISTING 18-10 Creating a Device Node and Mounting the SD Device

```
root@imx31~:# cat /sys/block/sda/sda1/dev
8:1
root@imx31:~# mknod /dev/sda1 b 8 1
root@imx31:~# mkdir /media/disk
root@imx31:~# mount /dev/sda1 /media/disk
```

LISTING 18-10 Continued

```
root@imx31:~# dir /media/disk
total 80
drwxr-xr-x 2 root root 16384 Oct 13  2008 Calendars
drwxr-xr-x 2 root root 16384 Oct 13  2008 Contacts
drwxr-xr-x 2 root root 16384 Oct 13  2008 Notes
drwxr-xr-x 2 root root 16384 Oct 13  2008 Recordings
drwxr-xr-x 8 root root 16384 Oct 13  2008 iPod_Control
```

`mknod` was covered in detail in Chapter 8.

18.5.2 USB HID Class

USB HID (Human Input Devices) is probably the most common USB device found on desktop Linux boxes, and sometimes on embedded systems. HID devices are relatively simple to use. The configuration options for USB HID support are found under Device Drivers --> HID Devices on the kernel configuration menu. You need to enable `CONFIG_HID_SUPPORT`, `CONFIG_HID`, and `CONFIG_USB_HID`. This is the generic HID driver that implements the USB-defined HID class driver support. Simply insert the `usbhid` module, and `modprobe` automatically includes its dependency, the HID core driver (`hid`):

```
root@imx31:~# modprobe usbhid
usbcore: registered new interface driver usbhid
usbhid: v2.6:USB HID core driver
root@imx31:~# lsmod
Module                  Size  Used by
usbhid                 18980  0
hid                    31428  1 usbhid
ehci_hcd               30836  0
usbcore               129752  3 usbhid,ehci_hcd
```

With this infrastructure in place, most common HID devices, such as mice, keyboards, and joysticks that conform to the HID Class Driver specification, should be recognized and enabled. When I plug in my Kensington USB wireless travel mouse, I see this:

```
usb 2-1.4: new low speed USB device using fsl-ehci and address 4
usb 2-1.4: configuration #1 chosen from 1 choice
input: Kensington USB Mouse as /devices/platform/fsl-ehci.1/usb2/2-1/
2-1.4/2-1.4:1.0/input/input2
input: USB HID v1.10 Mouse [Kensington Kensington USB Mouse] on
usb-fsl-ehci.1-1.4
```

As we did for USB storage, we can find the device numbers that the kernel assigned when it enumerated the mouse in the sysfs file system. Because we already know that the device numbers are contained in the `dev` attribute, we search for something reasonable:

```
root@imx31:~# find /sys -name dev | grep input
/sys/devices/platform/fsl-ehci.1/usb2/2-1/2-1.4/2-1.4:1.0/input/input2/event2/dev
/sys/devices/virtual/input/input0/event0/dev
/sys/devices/virtual/input/input1/event1/dev
```

Because we plugged the mouse device into port 4 of an external hub, we select the device with physical address 2-1.4:1-0:

```
root@imx31:~# cat /sys/devices/platform/fsl-ehci.1/usb2/2-1/2-1.4/2-1.4:1.0/input
➥/input2/event2/dev
13:66
```

Finally, we create the device node using these device numbers:

```
root@imx31:~# mknod /dev/mouse c 13 66
```

You are now ready to use the device. As with USB storage, if you have a working `udev` configuration (as described in the next chapter), you don't need to create the device node manually. This is the job of `udev`.

18.5.3 USB CDC Class Drivers

USB Communications Device Class (CDC) drivers were designed to provide a common driver framework for entire classes of common communications devices. Several standard CDC classes have been defined, including ATM, Ethernet, ISDN PSTN (common telephony), wireless mobile devices, cable modems, and similar devices.

One of the best uses of CDC class drivers is the Ethernet CDC functionality. It can be very handy to have Ethernet connectivity on an embedded device during development that might not have an Ethernet interface. There are many examples of such devices, such as cellular phones and PDAs.

There are two different ways to accomplish this functionality. One is to set up a point-to-point link between a host and a peripheral device directly using a standard

USB cable. The other model uses an Ethernet "dongle," basically an Ethernet interface with a USB plug. We will look at both methods.[9]

Setting up a direct USB link between a host and a peripheral is relatively straightforward. Of course, you must have the proper hardware. For example, you cannot connect a laptop PC to a desktop PC using only a USB cable. This is because one end of every USB link must be an "upstream" device (host or hub), and the other end must be a "downstream" device (peripheral device or "gadget"). Remember, USB is a master-slave protocol.

We will use BeagleBoard as an example of USB-USB direct networking using the CDC class driver. You must enable this functionality in your kernel. Of course, you must have your host controller driver, which, for BeagleBoard, is `musb_hdrc.ko`. This is enabled by selecting `USB_MUSB_HDRC` in your beagle kernel config. For BeagleBoard, you must also select `TWL4030_USB`[10] to enable the USB transceiver. Enable peripheral mode by selecting `USB_GADGET_MUSB_HDRC` under USB Gadget support. Finally, select Ethernet Gadget (`USB_ETH`) support under USB Gadget Drivers. This is the driver with CDC Ethernet support for the peripheral (slave) side of the link.

On your desktop or laptop host, you need to load `usbnet.ko`. On most modern desktop distributions, udev does this automatically after you plug in the USB cable coming from the BeagleBoard. This assumes that you have already enabled your Beagle's g_ether driver and configured the interface. Let's see what that looks like. Listing 18-11 shows the relevant steps.

LISTING 18-11 Beagle g_ether Configuration

```
# modprobe twl4030_usb
twl4030_usb twl4030_usb: Initialized TWL4030 USB module
# modprobe g_ether
musb_hdrc: version 6.0, musb-dma, peripheral, debug=0
musb_hdrc: USB Peripheral mode controller at d80ab000 using DMA, IRQ 92
g_ether gadget: using random self ethernet address
g_ether gadget: using random host ethernet address
usb0: MAC ae:9e:55:32:0a:c9
usb0: HOST MAC c2:de:61:36:21:9c
g_ether gadget: Ethernet Gadget, version: Memorial Day 2008
g_ether gadget: g_ether ready
```

[9] Although it may use common elements of CDC, using an Ethernet dongle in the following example is not strictly through a CDC class driver.

[10] It is more convenient to compile `TWL4030_USB` directly into the kernel (`=y`) because there is little reason to ever remove it. It is required for all USB modes and takes up very little space.

LISTING 18-11 Continued

```
# lsmod
Module                  Size  Used by
g_ether                23664  0
musb_hdrc              35524  1 g_ether
twl4030_usb             5744  0
# ifconfig usb0 192.168.4.2
```

After we load the two device drivers as shown in Listing 18-11, you notice that a `usb0` interface has been created and enumerated. `lsmod` shows which modules are loaded after these steps. Notice that `modprobe` automatically loaded the `musb_hdrc` module.

The final step is to configure the interface with a valid IP address. Now the Ethernet interface over `usb0` is ready for showtime. When the USB cable from the BeagleBoard is connected to my Ubuntu 8.04 laptop, here are the resulting log entries:

```
usb 7-3: new high speed USB device using ehci_hcd and address 13
usb 7-3: configuration #1 chosen from 1 choice
usb0: register 'cdc_ether' at usb-0000:00:1d.7-3, CDC Ethernet Device,
32:89:fb:38:00:04
usbcore: registered new interface driver cdc_ether
ADDRCONF(NETDEV_CHANGE): usb0: link becomes ready
```

Notice that the host side of the link has automatically installed `cdc_ether.ko` and its dependent module, `usbnet.ko`. On my Ubuntu 8.04 laptop:

```
$ lsmod | head -n 3
Module                  Size  Used by
cdc_ether               7168  0
usbnet                 20232  1 cdc_ether
```

Now we configure the host-side `usb0` interface with a valid IP address (easiest if it's on the same subnet), and we are done. We now have a working Ethernet interface from the BeagleBoard to a laptop USB host interface. Note that if your laptop/desktop distribution was not running udev (for some unimaginable reason) or was not properly configured, you would have to manually load usbnet and cdc_ether on the host end of your link. If a properly configured peripheral device such as the BeagleBoard is connected to a USB host port on your desktop/laptop host, loading these modules will create and enumerate a `usb0` interface.

18.5.4 USB Network Support

Another method of enabling Ethernet on an embedded device with a USB port is to use an Ethernet "dongle" plugged into a USB host port. These dongles are readily available at many electronics stores and online. The one we will use in this next example was purchased from Radio Shack. It is a nondescript unit manufactured in China, containing an ASIX chipset. When plugged into a USB host port, it becomes a fully operational Ethernet port.

You need to enable support for this functionality in your kernel. For this particular dongle, you must enable USB_USBNET and USB_NET_AX8817X. These options are found under Device Drivers --> Network Devices --> USB Network Support in your kernel configuration utility. After loading the necessary USB low-level drivers, load the ASIX driver and plug in the Ethernet dongle. The ASIX driver is loaded as follows:

```
# modprobe asix
usbcore: registered new interface driver asix
# usb 1-1.3: new high speed USB device using musb_hdrc and address 3
usb 1-1.3: configuration #1 chosen from 1 choice
eth0 (asix): not using net_device_ops yet
eth0: register 'asix' at usb-musb_hdrc-1.3, ASIX AX88772 USB 2.0 Ethernet,
00:50:b6:03:c8:f8
```

First you see the messages from the low-level USB support, enumerating the new USB device (1-1.3). Then the ASIX driver takes over and creates a new Ethernet interface, `eth0`. Unless you override the choices by passing parameters to the ASIX module while loading, it creates a random Ethernet MAC address for you.

The next step is simply to configure the interface with a valid IP address:

```
# ifconfig eth0 192.168.4.159
eth0: link up, 100Mbps, full-duplex, lpa 0xCDE1
eth0: link up, 100Mbps, full-duplex, lpa 0xCDE1
# ping 192.168.4.1
PING 192.168.4.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.4.1: icmp_seq=1 ttl=64 time=1002 ms
64 bytes from 192.168.4.1: icmp_seq=2 ttl=64 time=0.305 ms
64 bytes from 192.168.4.1: icmp_seq=3 ttl=64 time=0.336 ms
```

That's all there is to it!

18.6 USB Debug

Numerous debug options are available when USB support is enabled. Enabling verbose debug messages is a good way to see what is going on in your system. Enable USB_DEBUG in your kernel configuration to see debug messages from `usbcore` and the hub driver (part of `usbcore`.) Enabling USB_ANNOUNCE_NEW_DEVICES produces a report for each device inserted, listing vendor, product, manufacturer, and serial number strings from the device's descriptors. Both of these options enable output to syslog and are found under USB Support --> Support for Host-side USB.

Listing 18-12 displays the syslog entries after you plug in the Ethernet dongle from the preceding example with the debug configuration options just mentioned compiled into the kernel.

LISTING 18-12 Debug Output from Ethernet Dongle Insertion

```
user.info  kernel: usb 1-1.3: new high speed USB device using musb_hdrc
and address 3
user.debug kernel: usb 1-1.3: default language 0x0409
user.info  kernel: usb 1-1.3: New USB device found, idVendor=0b95, idProduct=7720
Jan  1 00:02:38 (none) user.info kernel: usb 1-1.3: New USB device strings:
Mfr=1, Product=2, SerialNumber=3
user.info  kernel: usb 1-1.3: Product: AX88772
user.info  kernel: usb 1-1.3: Manufacturer: ASIX Elec. Corp.
user.info  kernel: usb 1-1.3: SerialNumber: 000001
user.debug kernel: usb 1-1.3: uevent
user.debug kernel: usb 1-1.3: usb_probe_device
user.info  kernel: usb 1-1.3: configuration #1 chosen from 1 choice
user.debug kernel: usb 1-1.3: adding 1-1.3:1.0 (config #1, interface 0)
user.debug kernel: usb 1-1.3:1.0: uevent
user.debug kernel: drivers/usb/core/inode.c: creating file '003'
```

Most of the log entries are self-explanatory. You may be wondering about the last log entry, announcing the creation of file `'003'`. You may especially scratch your head when you can't find this file on your target system. It is part of the usb device file system, usbfs, and it is not visible until you mount usbfs as described earlier:

```
# mount -t usbfs usbfs /proc/bus/usb
```

After usbfs is mounted, you see the file created under `/proc/bus/usb/001`, which represents the USB interface just instantiated. The 001 is the bus number, and 003 represents the device. The file is not human-readable; it contains data from the descriptors of the USB interface.

Some platforms and drivers may have platform-specific debug options. For example, BeagleBoard kernels using the `musb_hdrc` driver can be compiled with debug functionality. Enable `USB_MUSB_DEBUG` in your kernel configuration to get this functionality. To make use of it, you must pass a debug level into the `musb_hdrc` driver when it is loaded. This particular option can produce verbose debug information, including information on each USB message sent between devices. You may recall that you pass a module parameter into a module by specifying it on the `modprobe` command line, or in distribution-specific configuration files. This example sets the debug level to 3 in the `musb_hdrc` driver:

```
# modprobe musb_hdrc debug=3
```

18.6.1 usbmon

If you are hard-core, you can try `usbmon`. This is a USB packet sniffer much like `tcpdump`. The word "packet" is not really used in USB, but the concept is the same. `usbmon` allows you to capture raw traces of the transfers between devices on a USB bus. If you were developing a USB device from scratch, for example, it might come in handy.

To use `usbmon`, you must first enable DEBUG_FS, which is found in the Kernel Hacking menu in your kernel configuration. You must also enable USB_MON, found under Device Drivers --> USB Support --> Support for host side USB.

After it is enabled, you must mount the debugfs file system as follows:

```
# mount -t debugfs debugfs /sys/kernel/debug
```

Then load the `usbmon.ko` driver:

```
# modprobe usbmon
```

After this is done, tracing is enabled. You see the debug sockets created by the `usbmon` driver in `/sys/kernel/debug/usbmon`. The kernel dumps the packets into these debug sockets. Like `tcpdump`, they are dumped in text format, which makes them (somewhat) human-readable. Simply start `cat` on one of the sockets. Listing 18-13 shows a few lines of trace when a USB Flash drive is inserted into a BeagleBoard with `usbmon` enabled.

LISTING 18-13 `usbmon` Trace

```
# cat /sys/kernel/debug/usbmon/0u
c717ee40 1830790883 C Ii:1:002:1 0:2048 1 D
c717ee40 1830790944 S Ii:1:002:1 -115:2048 1 <
c717e140 1830791005 S Ci:1:002:0 s a3 00 0000 0003 0004 4 <
c717e140 1830791249 C Ci:1:002:0 0 4 = 01010100
c717e140 1830791279 S Co:1:002:0 s 23 01 0010 0003 0000 0
c717e140 1830791524 C Co:1:002:0 0 0
c717e140 1830791554 S Co:1:002:0 s 23 03 0016 0003 0000 0
```

The `cat` process was started before the USB Flash drive was inserted. Fifty-three lines of output were generated, indicating that 53 USB packets[11] were transmitted. It is beyond the scope of this book to go into the format and details of this output trace. You can refer to the write-up in the kernel source tree at `.../Documentation/usb/usbmon.txt` for those details. However, more detailed knowledge of internal USB architecture is required to fully understand it. References are given at the end of this chapter if you want to take the next step in your knowledge of USB.

18.6.2 Useful USB Miscellanea

Often, when you plug a USB device into a host, nothing much happens. Many things could be wrong, but the most likely case is that no device driver is available for the device. If, upon insertion of a USB device, you see messages like these in the syslog, this means that the device was recognized, its descriptors were read, and a configuration was chosen, but Linux could not find an appropriate device driver for the device:

```
usb 1-1.1: new high speed USB device using ehci_hcd and address 5
usb 1-1.1: configuration #1 chosen from 1 choice
```

The remedy is to find and enable the correct device driver for your USB widget. Of course, as an embedded developer, you may just have to write one yourself.

Although it may seem obvious, systems that depend on USB devices such as a keyboard or mouse should configure these device drivers as statically linked into the kernel (`=y`). This prevents the possibility that the console could be lost on removal of a dependent module.

[11] The term "packet" is not in the USB vocabulary. However, the details are beyond the scope of this book. See the references at the end of the chapter if you want to dive in deeper.

18.7 Summary

This chapter presented the foundation for understanding the rather complex USB subsystem. It described the topology and concepts and examined how USB is configured and used. The most commonly used USB subsystems were discussed.

- You can better understand USB by examining its physical and logical topology.
- The different types of USB cables and connectors was covered.
- Several examples of USB configuration were presented.
- sysfs was presented in Chapter 9, and its use with USB was covered here.
- Several useful tools for understanding and troubleshooting USB were covered in detail.
- Class drivers including mass storage, HID, and CDC were introduced.
- The chapter concluded with some helpful debug tools and tips.

18.7.1 Suggestions for Additional Reading

Universal Serial Bus System Architecture, 2nd Edition
Don Anderson
Mindshare, Inc., 2001

Enhanced Host Controller Interface Specification for Universal Serial Bus
Version 1.0
Intel Corporation
www.intel.com/technology/usb/download/ehci-r10.pdf

Essential Linux Device Drivers
Chapter 11, "Universal Serial Bus"
Sreekrishnan Venkateswaran
Prentice Hall, 2008

Linux Device Drivers, 3rd Edition
Chapter 13, "USB"
Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman
O'Reilly, 2005

Universal Serial Bus Specification
Revision 2.0, April 27, 2000
www.usb.org/developers/docs/usb_20_052709.zip

USB Approved Class Specification Documents
www.usb.org/developers/devclass_docs