BACHELOR OF COMPUTER APPLICATIONS

SEMESTER 4

DCA2203

SYSTEM SOFTWARE

Unit 10

Device Driver – II

Table of Contents

1. INTRODUCTION

You were introduced to l drivers and their functions in the previous unit. The various categories of equipment that are connected to the computer system. Additionally, you read that when designing codes and device drivers, one should be mindful of security and design difficulties. In this section, you will learn about USB and PCI bus drivers as they pertain to the Linux operating system. .

Since all the peripheral devices and components attached to the computer system communicate with each other through the bus, it is very important to get familiar with the technique of how the bus drivers enable these buses communication between devices. PCI devices and their bus types are used for the interconnection of peripheral devices in most recent day's computers. USB-based devices are also used for various works that are connected through computers.

1.1 Learning Objectives

After studying this unit,learnersshould be able to:

- *List various PCI buses and PCI device types.*
- *Describe about the typical PCI system that is used for interconnection of PCI devices.*
- *Explain how the devices are configured when the computer boots.*
- *Discuss how the Linux kernel initializes the PCI system.*
- *Explain the working of USB devices and their configuration.*
- *Discuss about sysfs and urbs.*

2. PCI BUS DRIVERS

This unit provides an overview of the basic components that the PCI bus drivers are related to it. and also how PCI devices are initialized.

Peripheral Component Interconnect (PCI) is a standard to connect the peripheral components of a system in a structured and controlled way. This standard describes how the system components are electrically connected and how they should behave.

Let us first get introduced to the buses and PCI devices that belong to the PCI family.

PCI buses: The following buses belong to the PCI family:

- PCI: It is a 32-bit bus supporting 33 or 66 MHz.
- Mini PCI: It is found as a smaller slot in laptops.
- Card Bus: It is an external card slot in laptops.
- PIX Extended (PCIX): It is a wider slot than PCI, 64 bits, but can accept a standard PCI card.
- PCI Express (PCIe or PCIE): It is a current generation of PCI. It is Serial instead of parallel.
- PCI Express Mini Card: It replaces Mini PCI in recent laptops.
- Express Card: It replaces Card Bus in recent laptops.

All these bus technologies are compatible and handled by the same kernel drivers. The kernel can know which exact slot and bus variant is used.

PCI device types: These device types are the main types of devices found on the PCI bus. They are:

- Network cards (wired or wireless): A network card is a hardware that allows a computer to connect to a computer network.
- SCSI adapters: It is a device used to connect SCSI devices to a computer bus.
- Bus controllers (e.g., USB, PCMCIA, I2C, FireWire, IDE)

- Graphics and video cards: The Video Card, also known as a graphics card or graphics adapter, or video adapter, is an expansion card that allows the computer to send graphical information to a video display device such as a monitor or projector.
- Sound cards: It is a card that facilitates the input and output of audio signals to and from a computer.

PCI Bus Driver Installation Procedure is seen here:

- The driver/software normally comes together with the hardware. However, if you did not find one, then you may download it from the official website.
- You may require a driver to play hardware that is plugged into the PCI.
- Specifications and user manual must be checked when it is required to connect a device using a PCI bus driver.
- Same slot connection should be done as a 32-bit card in the 32-bit slot and a 64-bit card in the 64-bit slot.
- The number of PCI slots depends on how many have been provided by the manufacturer. Later it can have more.

Next is Registering PCI Bus Driver:

The driver's name must be single between all PCI drivers in the kernel. It is generally set to the same name as the module name of the driver. It displays in sysfs under /sys/bus/ PCI/drivers/ when the driver is in the kernel.

The struct PCI driver structure is the primary methods command that the entire PCI driver should execute to properly register with the kernel.

The structure includes several variables and function callbacks that define the PCI driver to the PCI core.

Const char* field names must be known by the PCI driver.

To create a proper structure, only four fields need to be initialized, as shown here:

- static struct pci_driver pci_driver =

{

.name = "pci_skel",

.id_table = ids,

.probe = probe,

.remove = remove,

};

Name, table, probe & remove are initialized as seen.

PCI Addressing for PCI Bus Drivers:

Windows OS automatically installs the driver that allows computers to recognize basic motherboard functions

In the 1086 process family, PCI enables independent memory and I/O ports for both 64-bits and 32-bits.

A particular structure is used rather than a binary address to interact with device drivers. .

By using inb, readb, and other conventional addressing methods, memory and I/O areas are used.

Self-Assessment Questions - 1

1. "_____" is a standard to connect the peripheral components of a system in a structured and controlled way.
2. PIX Extended (PCIX) is a wider slot than PCI, 64-bit, but can accept a standard PCI card. (True/False)

3. THE PERIPHERAL COMPONENT INTERCONNECT (PCI) INTERFACE

In the early age of computers, Industry Standard Architecture (ISA) bus, originally called the "AT bus," was used as an expansion bus that accepted plug-in boards for sound, video display, and other peripheral connectivity. The PCI architecture was meant to replace the ISA standard with three main objectives:

i) provide better performance while transferring data between the computer and its peripherals.
ii) be platform independent
iii) simplicity in adding and removing peripherals to the system.

The PCI bus performs better due to a higher clock rate than ISA. PCI is currently used extensively on IA-32, Alpha, PowerPC, SPARC64, IA- 64 systems, and some other platforms as well.

PCI devices configure automatically at boot time. It means that the device driver access configuration information in the device to complete initialization during boot time.

The types of PCI Interface working:

- The PCI interface can work in synchronous or asynchronous modes.
- The Bus typically runs synchronously at the external clock frequency of the microprocessor or a submultiple of it.
- The PCI Bus speed is independent of the processor's clock when operating asynchronously.

The synchronous working of the PCI interface has been explained:

- 66-MHz Pentium could synchronously connect to a PCI Bus at half its clock frequency (33 MHz).
- In synchronous mode, the standard PCI clock can be between 20 and 33 MHz

Let us see the asynchronous working of the PCI interface:

- For the best possible performance, the asynchronous mode is used while operating at the maximum PCI Bus frequency.
- By utilizing flow-control signals that indicate when a board is prepared to send or receive data, the PCI standard additionally assists cards unable to operate at the full Bus speed (33 or 66 MHz).

The expansion devices have been shown here.

These are connected to the motherboard with the help of the expansion Bus.

- Expansion Devices are
- Additional Memory
- SCSI Controller
- ISA Interface
- Ethernet Interface
- USB Controller

This slide shows the different versions of PCI Bus drivers commonly used.

PCI Bus Specifications:

- PCI version 1.0 was developed by Intel in 1992
- PCI revision 2.0 was released in 1993, having 32-bit, 33MHz bus
- PCI revision 2.1 was released in 1995, having 32-bit, 33MHz / 64-bit, 66MHz
- PCI revision 2.2 was released in 1998, having minor enhancements
- PCI revision 2.3 was released in 2002 and removed 5v-only cards
- PCI revision 3.0 was released in 2004 and removed 5-volt interfaces altogether

Let us see the PCI Bandwidth Based on Clock Rates:

A Bus number, a device number, and a function number identify each PCI peripheral. .

132 MB/s utilizing a 64-bit data path at a 33 MHz clock rate

264 MB/s peak utilizing a 32-bit data path at 66 MHz

264 MB/s peak utilizing a 64-bit data path at 66 MHz

532 MB/s peak utilizing a 32-bit data path at 133 MHz

1064 MB/s peak utilizing a 64-bit data path at 133 MHz

The advantages of PCI Bus Drivers have been explained here:

- A maximum of five components can be connected to the PCI, and all of them can be further replaced by the built-in devices on the motherboard.
- On related machines, you'll find many PCI Buses.
- Exchange speeds will improve from 33 MHz to 133 MHz with a 1 GB/s transmission rate thanks to PCI transport.
- The PCI can support devices with a maximum voltage of 5 volts, and the pins used can switch more than one flag by one stick.

3.1 Peripheral Component Interconnect (PCI) Addressing

Before we proceed to read about PCI addressing, let us get introduced to PCI address spaces and Bridges.

PCI address spaces

The CPU and the PCI devices access the memory shared between them. Memory is used by device drivers to control the PCI devices and to pass information between them. Generally, that the shared memory contains control and status registers for the device. These registers control the devices and read their status.

Peripheral devices have their own memory spaces. The CPU can access these memory spaces. However, DMA (Direct Memory Access) channels control the way devices access the system's memory.

ISA devices have access to two address spaces: ISA I/O (Input/Output) and ISA memory, whereas PCI has access to three address spaces: PCI I/O, PCI Memory, and PCI Configuration space. CPU accesses these address spaces with the PCI I/O and PCI Memory address spaces used by the device drivers and the PCI Configuration space used by the PCI initialization code within the Linux kernel.

Some of the processors, like the Alpha AXP processors access only the system address space and not address spaces. It uses support chipsets to access other address spaces, such as PCI Configuration. A sparse address mapping scheme is used to steal part of the large virtual address space to map it to the PCI address spaces.

Please note that each PCI peripheral can be identified by:

i) *a bus number,*
ii) *a device number and*
iii) *a function number.*

The PCI specification allows a single system to host up to 256 buses, but 256 buses are not sufficient for many large systems. However, Linux is supported through PCI *domains*. Each PCI domain can host up to 256 buses. Each bus hosts up to 32 devices, and each can be a multifunction board (such as an audio device with an accompanying CD-ROM drive) with a maximum of eight functions. Therefore, each function can be identified at the hardware level by a 16-bit address or key. However, device drivers written

for Linux do not need to deal with those binary addresses because they use a specific data structure pci_dev, to act on the devices.

Modern days computers feature at least two PCI buses. Hence, you can plug more than one bus into a single system using bridges and use special-purpose PCI peripherals to join two buses.

The overall layout of a PCI system is seen as a tree where each bus is connected to an upper-layer bus up to bus 0 at the root of the tree. The Card Bus PC-card system is also connected to the PCI system via bridges.

***PCI I/O and PCI memory addresses*:** These two address spaces are used by the devices to communicate with their device drivers that run in the Linux kernel on the CPU. Note that only the PCI configuration code reads and writes PCI configuration addresses. The Linux device drivers only read and write PCI I/O and PCI memory addresses.

***PCI-ISA bridges**:* These bridges support legacy ISA devices by translating PCI I/O, and PCI Memory space accesses into ISA I/O and ISA Memory accesses.

PCI-PCI bridges: PCI-PCI bridges are special PCI devices that connect the PCI buses of the system. Simple systems have a single PCI bus, but there is an electrical limit on the number of PCI devices that a single PCI bus can support. PCI-PCI bridges add more PCI buses and allow the system to support many more PCI devices.

PCI-PCI bridges (PCI I/O and PCI memory windows): PCI-PCI bridges only pass a subset of PCI I/O, and PCI memory reads and writes requests downstream.

Figure 10.1 shows a typical PCI system. The 16-bit hardware addresses associated with PCI peripherals are depicted in the struct *pci_dev* object. However, one can visualize lists of devices using the command *lspci* (part of the *pciutils* package, available with most distributions) and find the layout of information in */proc/pci and /proc/bus/pci*.

This addressing scheme is also shown by the sysfs representation of PCI devices, with the addition of the PCI domain information. Sysfs is a feature of the Linux 2.6 kernel that allows kernel code to export information to user processes via an in-memory file system. When the hardware address is displayed, it can be shown as two values (an 8-bit bus number and an 8-bit device and function number), as three values (bus, device, and function), or as four values (domain, bus, device, and function); all the values are usually displayed in hexadecimal.

For example, */proc/bus/pci/devices* use a single 16-bit field (to ease parsing and sorting), while */proc/bus/bus number* splits the address into three fields. The following shows how those addresses appear, showing only the beginning of the output lines:

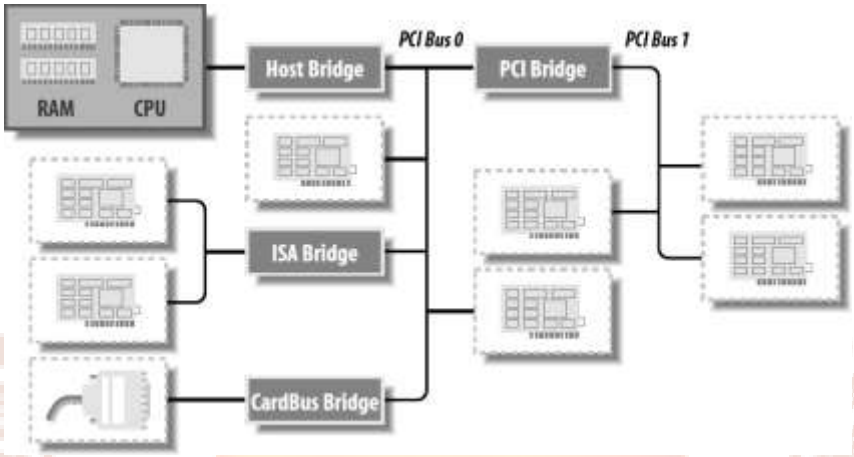

Figure 10.1: Layout of a Typical PCI System

All three lists of devices (bus, device, and function) are sorted in the same order since lspciuses the /proc files as its source of information. Taking the VGA video controller as an example, 0 x 00a0 means 0000:00:14.0 when split into a domain (16 bits), bus (8 bits), device (5 bits), and function (3 bits).

The hardware circuitry of each peripheral board answers queries about three address spaces:

i) memory locations,
ii) I/O ports, and
iii) configuration registers.

All the devices share the first two address spaces on the same PCI bus. PCI peripherals also can work when interrupted (hardware or software). Every PCI slot has four interrupt pins, and each one of them can be used by the device functions. PCI specifies the interrupt lines to be shared; hence, even a processor with a limited number of IRQ lines, such as the x86, can host many PCI interface boards (each with four interrupt pins).

The I/O space in a PCI bus uses a 32-bit address bus (leading to 4 GB of I/O ports), while the memory space can be accessed with either 32-bit or 64-bit addresses. You can find 64-bit addresses on present-day platforms.

The PCI configuration space consists of 256 bytes for each device function (except for PCI Express devices, which have 4 KB of configuration space for each function), and the layout of the configuration registers is standardized. Four bytes of the configuration space hold a unique function ID, so the driver can identify its device by looking for the specific ID for that peripheral.

The main innovation of the PCI interface standard over ISA is the configuration address space. Therefore, in addition to the usual driver code, a PCI driver needs the ability to access the configuration space.

Self-Assessment Questions - 2

3. In Linux, each PCI domain can host up to "_____"buses.
4. The I/O space in a PCI bus uses a "_____ "-bit address bus, while the memory space can be accessed with either "_____"-bit "_____bit addresses.

3.2 Boot Time

Now let's read about PCI's operation. Keep in mind that all devices are configured at system startup time.

When power is applied to a PCI device, the hardware remains inactive. That is, the device responds only to configuration transactions. When power is provided, the device has no memory, and no I/O ports are mapped in the computer's address space. All the device-specific feature, such as interrupt reporting, is also disabled.

Every PCI motherboard is equipped with PCI-aware firmware, called BIOS, NVRAM, or PROM, depending on the platform. This firmware performs configuration transactions with every PCI peripheral to allocate a safe place for each address region it offers. Memory and

I/O areas of the device are already mapped into the processor's address space by the time a device driver accesses it.

In Linux, the user can look at the PCI device list and the devices' configuration registers by reading */proc/bus/pci/*devices and

/proc/bus/pci//*.*

The former is a text file with (hexadecimal) device information, and the latter are binary files that report a snapshot of the configuration registers of each device, one file per device. The individual PCI device directories in the sysfs tree can be found in */sys/bus/pci/*devices.

3.3 Configuration Registers And Initialization

This section, deals with the configuration registers that PCI devices contains. All PCI devices feature at least a 256-byte address space. The first 64 bytes are standardized, while the rest are device dependent. The layout of the device-independent configuration space is shown in Figure 10.2.

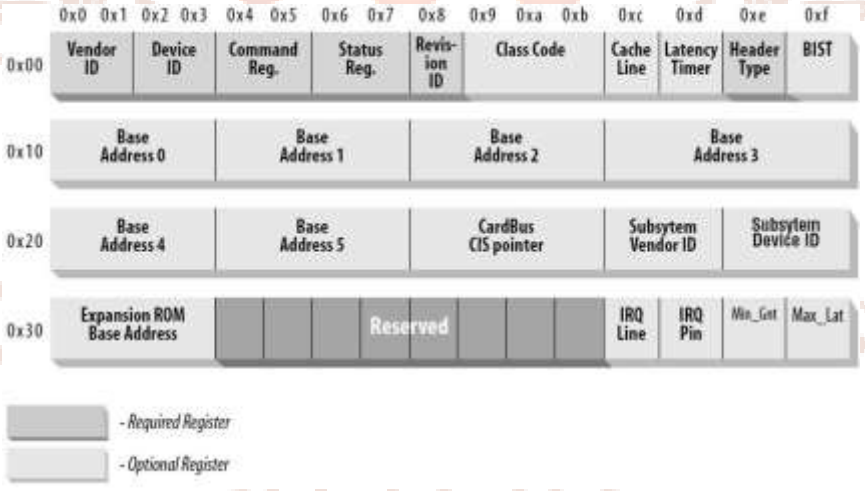

Figure 10.2: The Standardized PCI Configuration Registers

There are two registers. Required registers and Optional registers. Depending upon the value in the Header Type field, some of the PCI configuration registers are optional in the PCI configuration registers whereas some are required. Every PCI device contains some important data in the required registers, whereas the data in the optional registers depend

on the actual capabilities of the peripheral. The optional fields are not used unless the contents of the required fields indicate that they are valid. You can also see in the figure that a few registers are reserved for further use.

Every PCI device in the system has a configuration data structure in the PCI configuration address space. The PCI Configuration header helps the system to identify and control the device.

Typically, systems are designed so that every PCI slot has its PCI Configuration Header in an offset related to its slot on the board. A system-specific hardware mechanism is defined so that the PCI configuration code can attempt to examine all possible PCI Configurations.

Let us now read the description of each field in the PCI configuration registers. Note that the technical documentation released with each device usually describes the supported registers.

Vendor ID: This 16-bit register identifies the originator of the PCI device. E.g., Digital's PCI Vendor Identification is 0x1011, and Intel's is 0x8086.

Device ID: This 16-bit register is a unique number to identify the device. This ID is usually paired with the vendor ID to make a unique 32-bit identifier for a hardware device.

Command Reg: This field is used to control the device to generate and respond to PCI cycles.

Status Reg: This field gives the status of the device with the meaning of the bits of this field set by the standard.

Class Code: This identifies the type of device that this is. There are standard classes for every sort of device, video, SCSI, and so on.

Cache Line: It specifies the system cache line size in word (2-byte) units.

Latency Timer: It specifies the latency timer in units of PCI bus clocks.

Header Type: It identifies the layout of the rest of the header beginning at byte 0x10 of the header and and whether the equipment has numerous functions is made clear. Where a value of 0x00 specifies a general device, a value of 0x01 specifies a PCI-to-PCI bridge, and a

value of 0x02 specifies a Card Bus bridge. If bit 7 of this register is set, the device has multiple functions; otherwise, it is a single function device.

BIST: Represents that status and allows control of a devices BIST (built-in self-test).

Base Address Registers: These registers determine and allocate the type, amount, and location of PCI I/O and PCI memory space that the device can use.

Card Bus CIS Pointer: It Points to the Card Information Structure (CSI) and is used by devices that share silicon between Card Bus and PCI.

Subsystem Vendor ID, Subsystem Device ID: The vendor sometimes sets these fields to further differentiate similar devices.

Expansion ROM base address: It consists of the base address of the expansion ROM.

Max_Lat: Max Latency is a read-only register that specifies how often the device needs access to the PCI bus.

Min_Gnt: Min Grant is a read-only register that specifies the burst period length that the device needs.

Interrupt Pin: The Interrupt Pin field describes which pins the PCI device uses. Generally, it is hardwired for a particular device.

Interrupt Line: The Interrupt Line field of the device's PCI Configuration header is used to pass an interrupt handle between the PCI initialization code, the device's driver, and the interrupt handling subsystem. It allows the interrupt handler to correctly route an interrupt from the PCI device to the correct device driver's interrupt handling code within the Linux operating system.

For the x86 architecture, this register corresponds to the PIC IRQ numbers 0-15 (and not I/O APIC IRQ numbers), and a value of 0xFF defines no connection.

Self-Assessment Questions - 3

5. When power is provided, the device has no memory, and no I/O ports are mapped in the computer's _____ space.
6. Typically, systems are designed so that every PCI slot has its PCI Configuration _____ in an offset that is related to its _____on the board.
7. Which of the following field describes the pins of PCI device use?
   (a) The Interrupt Pin field
   (b) Interrupt Line field
   (c) Class Code field
   (d) Header Type field

3.4 Linux Pci Initialization

The PCI initialization code in Linux consists of three logical parts:

i) **PCI device driver:** This driver searches the PCI system starting at Bus 0 and locates all PCI devices and bridges. It builds a linked list of data structures and describes the ' system's topology.

ii) **PCI BIOS:** It is software that provides the services described in bib-pci-bios-specification. Even though Alpha AXP does not have BIOS services, there is equivalent code in the Linux kernel providing the same functions.

The PCI BIOS functions are a series of standard routines and are common across all platforms. For example, they are the same for both Intel and Alpha AXP based systems. They allow the CPU controlled access to all the PCI address spaces.

iii) PCI fixup

System specific fixup code ties up the system specific loose ends of PCI initialization.

Linux kernel initializes the PCI system and builds data structures reflecting the real PCI topology of the system. A pci_dev data structure describes each PCI device (including the PCI-PCI Bridges) . Each PCI bus is described by a ***pci_bus*** data structure. The result is a tree

structure of PCI buses, each of which has several child PCI devices attached to it. As a PCI bus can only be reached using a PCI-PCI Bridge (except the primary PCI bus, bus 0), each pci_bus contains a pointer to the PCI device (the PCI- PCI Bridge) that it is accessed through. That PCI device is a child of the PCI Bus's parent PCI bus.

A pointer is there to all the PCI devices in the system called pci_devices. All PCI devices in the system have their pci_dev data structures. This helps the Linux kernel discover all the PCI devices in the system.

Registering a PCI driver

The main structure that all PCI drivers must create to be registered with the kernel properly is the struct ***pci_driver*** structure. This structure consists of several function callbacks and variables that describe the PCI driver to the PCI core.

The PCI device driver is not a device driver at all but a function of the operating system called system initialization time. The PCI initialization code scans every PCI bus in the system that needs to connect to all PCI devices (including PCI-PCI bridge devices).

There are various fields in this structure that a PCI driver needs to be aware of. Those fields provide:

- the name of the driver is unique among all PCI drivers in the kernel and is normally set to the same name as the module name of the driver.
- Pointer to the probe function in the PCI driver etc.
- Pointer to the function that the PCI core calls when the struct pci_dev is being removed from the system or when the PCI driver is being unloaded from the kernel
- Pointer to the function that the PCI core calls when the struct pci_dev
  - is being suspended
- Pointer to the function that the PCI core calls when the struct pci_dev
  - is being resumed.

To register the struct ***pci_driver*** with the PCI core, a call to ***pci_register_driver*** is made with a pointer to the struct ***pci_driver***. Note that the pci_register_driver function either returns a

negative error number or 0 if everything was registered successfully. It does not return the number of devices bound to the driver or an error number if no devices were bound to the driver.

When the PCI driver is to be unloaded, the struct ***pci_driver*** must be unregistered from the kernel. This is done with a call to pci_unregister_driver. When this call happens, any PCI devices currently bound to this driver are removed, and the remove function for this PCI driver is called before the pci_unregister_driver function returns.

3.5 Old-Style PCI Probing

In older kernel versions, the function pci_register_driver, was not always used by PCI drivers. Instead, the list of PCI devices in the system was dealt with manually, or a function would be called that searched for a specific PCI device. The ability to deal with the list of PCI devices in the system within a driver has been removed from the 2.6 kernel to prevent drivers from crashing the kernel, which occurs while modifying the PCI device lists, and a device is being removed simultaneously.

In old-style PCI probing, the ability to find a specific PCI device is provided by the following available functions:

- function to scan the list of PCI devices currently in the system.
- function to allow the subsystem vendor and subsystem device IDs to be specified when looking for the device.
- function to search the list of PCI devices in the system on the specified struct ***pci_bus*** for the specified device and function number of the PCI device.

Note that these functions cannot be called from an interrupt context. If they are, a warning is printed out to the system log.

Self-Assessment Questions - 4

8. The PCI BIOS functions are a series of standard " "common _____ across all platforms.
9. To register the structpci_driver with the PCI core, a call to "_____" is made with a pointer to the structpci_driver.

4. USB DRIVERS

The full form of USB is Universal Serial Bus. The USB provides a connection between a host computer and several peripheral devices. Nowadays, USB can support almost every type of device that can be connected to a PC. The USB transfer speed can be categorized as:

- Low Speed: data transfer is up to 1.5 Mbps by the USB 1.0
- Full Speed: data transfer is up to 12 Mbps by the USB 1.1
- Hi-Speed: data transfer is up to 480 Mbps by USB 2.0

Topologically, a USB subsystem is a tree built out of several point-to-point links. The links are four-wire cables (ground, power, and two signal wires) that connect a device and a hub, just like twisted-pair Ethernet. The USB host controller asks every USB device if it has any data to send. It is because of this topology; a USB device can send data only when asked by the host controller. This configuration allows for a very easy plug-and-play type of system, whereby the host computer can automatically configure devices.

At the technological level, the USB is a single-master implementation in which the host computer polls the various peripheral devices. The USB bus has the ability for a device to request a fixed bandwidth for its data transfers to reliably support video and audio I/O. USB acts only as a communication channel between the device and the host. It does not require specific meaning or structure to the data it delivers.

The USB protocol specifications define a set of standards that any device of a specific type can follow. If a device follows that standard, then a special driver for that device is unnecessary. The different types of standards are called classes, consisting of devices like storage devices, keyboards, mice, joysticks, network devices, and modems. Those devices that do not fit into these classes require a special vendor-specific driver to be written for that specific device. Video and USB-to-serial devices are good where there is no defined standard, and a driver is needed for every device from different manufacturers.

Since a USB is built like a tree out of several point-to-point links and it has an inherent hot-plug capability of the design, a USB is handy and provides a low-cost mechanism to connect

(and disconnect) several devices to the computer without the need to shut the system down, open the cover, and swear over screws and wires.

The Linux kernel supports two main types of USB drivers:

i) Drivers on a host system and
ii) Drivers on a device

The USB drivers for a host system control the USB devices plugged into it from the host's point of view (a common USB host is a desktop computer.) The USB drivers in a device control how that single device looks to the host computer as a USB device. As the term "USB device drivers" is very confusing, USB developers have created the term "USB gadget drivers" to describe the drivers that control a USB device that connects to a computer (remember that Linux also runs in those tiny, embedded devices, too.) USB host drivers have a different driver for each USB control hardware and is usually available in the Board Support Package. These are architecture and platform dependent.

In this section, you will read how the USB system runs on a desktop computer, i.e., how drivers work with the devices.

In Figure 10.3 of the USB driver, you can see that the USB drivers live between the kernel subsystems (block, net, char, etc.) and the USB hardware controllers. The USB core provides an interface for USB drivers to use to access and control the USB hardware without worrying about the different types of USB hardware controllers on the system.

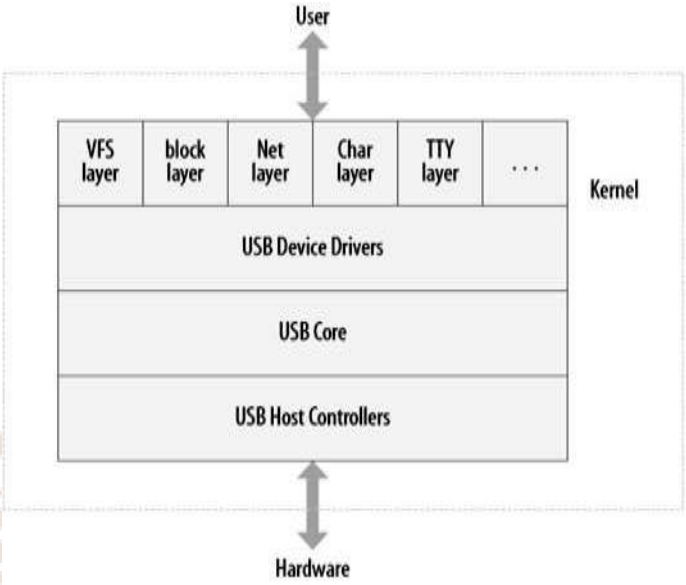

Figure 10.3: USB Driver Overview

Self-Assessment Questions - 5

10. The USB protocol specifications define a set of "_____"that any device of a specific type can follow.
11. The Linux kernel supports two main types of USB drivers: (i) Drivers on a "_____" and (ii) Drivers on a_____".

4.1 USB Device Basics

A USB device is described as complex in the official USB documentation . However, the Linux kernel provides a subsystem called the USB core to handle most of the complexity. In this section, you will read about the basics of a driver and the USB core. Figure 10.4 shows how USB devices consist of configurations, interfaces, and endpoints and how USB drivers bind to interfaces, not the entire USB device.

Endpoints

It is through the endpoint that the most basic form of USB communication happens. Endpoints are sources or sinks of data. Endpoints occur at the end of the communications channel at the USB function, as the bus is host-centric. A USB endpoint carries data in one

direction only. If data is carried from the host computer to the device, it is called an OUT endpoint, whereas if it is from the device to the host computer, it is called an IN endpoint. Endpoints can be thought of as unidirectional pipes.

A USB endpoint can be one of four different types that describe how the data is transmitted:

i) **Control:** Control endpoints allow access to different parts of the USB device. They configure the device, retrieve information about it, send commands to it, or retrieve status reports about it . These endpoints are usually small in size. Every USB device consists of a control endpoint called "endpoint 0" that is used by the USB core to configure the device when it is inserted to computer. The USB protocol guarantees reserved bandwidth to access through to the device.

ii) **Interrupt:** Interrupt endpoints transfer small data at a fixed rate every time the USB host asks the device for data. As an example, these endpoints are used for USB keyboards and mice. It is also used to send data to USB devices to control the device. The USB protocol guarantees enough reserved bandwidth to access the device.

iii) **Bulk:** Bulk endpoints transfer large amounts of data. These endpoints are usually much larger (they can hold more characters at once) than interrupt endpoints.
These endpoints are commonly used for devices that transfer data without data loss. The USB protocol does not guarantee always make it through in a specific amount of time. If the bandwidth of the bus is not enough to send the whole BULK packet, data is split up across multiple transfers to or from the device. For example, printers, storage, and network devices use this endpoint.

iv) **Isochronous:** Isochronous endpoints also transfer large amounts of data, but the data is only sometimes guaranteed to make it through. Those devices that can handle data loss and rely more on keeping a constant stream of data flow use these endpoints. Real-time data collections, such as audio and video devices, use these endpoints.

Control and bulk endpoints are used for asynchronous data transfers whenever the driver decides to use them. Interrupt, and isochronous endpoints are periodic. This means that these endpoints are set up to transfer data at fixed times continuously, which causes their bandwidth to be reserved by the USB core.

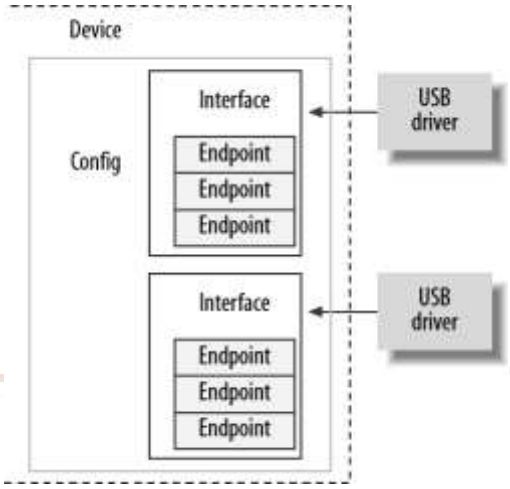

Figure 10.4: USB Device Overview

USB endpoints are described in the kernel with the structure struct ***usb_host_endpoint***. This structure contains the real endpoint information in another structure called struct ***usb_endpoint_descriptor.*** The latter structure contains all the USB-specific data in the exact format specified by the device. The fields of this structure directly correspond to the field names in the USB specification. The fields of this structure that drivers care about are: **bEndpoint Address, bmAttributes, wMax PacketSize, bInterval** which provides specification about the USB devices.

Interfaces

USB endpoints are bundled up into interfaces. USB interfaces handle only one type ofa USB logical connection, such as a mouse, a keyboard, or an audio stream. Some USB devices have multiple interfaces. E.g., a USB speaker might consist of two interfaces: (i) a USB keyboard for the buttons and (ii) a USB audio stream. A USB interface only represents basic functionality, so each USB driver controls an interface. For example, for the speaker, Linux needs two different drivers for one hardware device.

USB interfaces may have alternate settings for different parameters of the interface. The initial state of an interface is in the first set and is numbered zero. Alternate settings can be used to control individual endpoints for various purposes. E.g., USB bandwidth for the device can be changed. Every device with an isochronous endpoint uses alternate settings for the same interface.

USB interfaces are described in the kernel with the struct ***usb_interface*** structure. The USB core passes this structure to USB drivers, and the USB driver is responsible for controlling any functions.

Configurations

USB interfaces are also bundled up into configurations. A USB device can have multiple configurations to change the various states of the device. A single configuration can be enabled only at one point in time. Linux does not handle multiple configurations for USB devices very well, but it is a rare case.

Linux describes USB configurations with the structure struct ***usb_host_config*** and entire USB devices with the structure *struc**tusb_device***. You can find descriptions of them in the file include/linux/usb.h in the kernel source tree.

A USB device driver converts data from a given struct ***usb_interface*** structure into a struct ***usb_device*** structure that the USB core needs for various function calls. To do this, the function ***interface_to_usbdev*** is provided.

To summarize, USB devices are made up of lots of different logical units. The relationships among these units can be simply described as follows:

- Devices usually have one or more configurations.
- Configurations often have one or more interfaces.
- Interfaces usually have one or more settings.
- Interfaces have zero or more endpoints.

Self-Assessment Questions - 6

12. What type of endpoint in a USB can carry data from the host computer to the device?
    (a) in point
    (b) out point
    (c) mid-point
    (d) common point
13. Isochronous endpoints also transfer large amounts of data and are always guaranteed to make it through. (True/False)
14. A USB interface represents basic functionality only, and hence each USB driver controls an

Advantages of USB Drivers

- Since USB connectors are fairly straightforward, many computing devices, including desktop and laptop computers, have them. There are USB hubs available to add more ports (if any).
- Installing a USB device is simple and only needs the OS (Operating System).
- When compared to RS232 and other parallel interfaces, USB ports are much smaller.
- External power is unnecessary.
- USB can transfer data at various distances and speeds—from 1.5 Mbps to 5 Gbps.
- In the event of faults, USB protocol alerts the transmitter to retransmit the data. To ensure error-free transmission, the USB driver is used.

4.2 USB And Sysfs

Sysfs is a feature of the Linux 2.6 kernel that allows kernel code to export information to user processes via an in-memory file system. Sysfs is used to set up and export data about devices and drivers from the kernel device model to user space.

The physical USB device is represented by a struct ***usb_device*** and the individual USB interfaces are represented by a struct ***usb_interface***. These are shown in sysfs as individual devices, as both structures contain a ***struct_device*** structure.

The USB sysfs device naming scheme is given as:

root_hub-hub_port:config.interface

E.g., The struct ***usb_device*** for the simple USB mouse, the sysfs device naming is:

/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1/2-1:1.0

Sysfs does not expose all the different parts of a USB device, as it stops at the interface level. Any alternate configurations that the device may contain and the endpoints associated with the interfaces are not shown. The system's /proc/bus/usb/ directory contains the information about it in the usbfs file system. The identical data displayed in sysfs is also shown in the file /proc/bus/usb/devices. It also shows alternate configuration and endpoint information for all USB devices in the system.

4.3 USB Urbs

The USB code in the Linux kernel communicates with all USB devices using urb (USB request block). A new driver is found by message passing. This message itself is called USB Request Block, or URB for short. This request block is described with the struct urb structure and is found in the include/linux/usb.h file.

A urb is used to send or receive data to or from a specific USB endpoint on a USB device in an asynchronous manner.

A USB device driver may allocate many urbs for a single endpoint or may reuse a single urb for many different endpoints, depending on the need of the driver. Every endpoint in a device can handle a queue of urbs, so multiple urbs can be sent to the same endpoint before the queue is empty. The typical lifecycle of an urb is as follows:

- Created by a USB device driver.
- Assigned to a specific endpoint of a specific USB device.

- Submitted to the USB core by the USB device driver.
- Submitted to the specific USB host controller driver for the specified device by the USB core.
- Processed by the USB host controller driver that transfers a USB to the device.
- When the urb is completed, the USB host controller driver notifies the USB device driver.

Urbs can also be canceled at any time by the driver that submitted the urb, or by the USB core if the device is removed from the system. urbs are dynamically created and contain an internal reference count that enables them to be automatically freed when the last user of the urb releases it.

Self-Assessment Questions - 7

15. Sysfs is a feature of the Linux 2.6 kernel that exports information about devices and drivers from the "_____" device model to user space.
16. The basic idea of the new driver is message passing, the message itself is called "_____"Request Block, or for short.

5. SUMMARY

Let us recapitulate the important concepts discussed in this unit:

- ISA devices have access to two address spaces: ISA I/O (Input/Output) and ISA memory, whereas PCI has access to three address spaces: PCI I/O, PCI Memory, and PCI Configuration space.
- All PCI devices feature at least a 256-byte address space. The first 64 bytes are standardized, while the rest are device dependent.
- As the Linux kernel initializes the PCI system, it builds data structures mirroring the real PCI topology of the system.
- The main structure that all PCI drivers must create to be registered with the kernel properly is the struct ***pci_driver*** structure.
- In older kernel versions, the list of PCI devices in the system was dealt with manually, or a function would be called that searched for a specific PCI device.
- Topologically, a USB subsystem is a tree built out of several point-to-point links.
- The USB protocol specifications define standards that any device of a specific type can follow.
- USB endpoints are bundled up into interfaces and configurations.
- Sysfs is a feature of the Linux 2.6 kernel that exports information about devices and drivers from the kernel device model to user space and is also used for configuration.
- The basic idea of the new driver is message passing; the message itself is called USB Request Block, or URB for short.

6. GLOSSARY

Interface requirement: A requirement that specifies an external item with which a system or system component must interact or sets forth constraints on formats, timing, or other factors caused by such an interaction

PCI: "Peripheral Component Interconnect "is a hardware bus used for adding internal components to a desktop computer

PCI-eXtended (PCI-X): An expansion bus and expansion card standard that enhances the 32-bit PCI Local Bus for higher bandwidth demanded by servers.

PCI Express (PCIe): An expansion bus standard designed to replace the older PCI, PCI-X, and AGP bus standards.

Universal Serial Bus (USB):

A specification to establish communication between devices and a host controller.

7. TERMINAL QUESTIONS

Short Answer Questions

1. List the various types of PCI buses and PCI device types.
2. Draw a figure of the Standardized PCI Configuration Registers and explain it in brief.
3. How is the PCI driver registered? Explain.
4. What is the endpoint in USB? Explain briefly about each of the endpoints.
5. Write short notes on (a) Sysfs (b) urbs

8. ANSWERS

Self-Assessment Questions

1. Peripheral Component Interconnect (PCI)
2. True
3. 256
4. 32,32,64
5. address
6. Header, slot
7. (a) The Interrupt Pin field
8. routines
9. pci_register_driver
10. standards
11. host system, device
12. (b) out point
13. False
14. interface
15. kernel
16. USB, URB

Short Answer Questions

1. The numerous types that make up PCI buses include: PCI, Mini PCI, Card Bus, etc. The main types of PCI devices are Network cards, SCSI adapters, etc. (Refer to section 10.2 for more details.)
2. Every PCI device in the system has a configuration data structure in the PCI configuration address space. (Refer to section 10.3.3 for more details)
3. The main structure that all PCI drivers must create to be registered with the kernel properly is the struct ***pci_driver*** structure. (Refer to section 10.3.4 for more details)
4. Endpoints are sources or sinks of data. (Refer to section 10.4.1 for more details)
5. Sysfs is a feature of the Linux 2.6 kernel. A urb is a message to send or receive data to or from a USB endpoint. (Refer to section 10.4.2 and 10.4.3 for more details)

8. SUGGESTED BOOKS

- Corbet J., Kroah-Hartman G., & Rubini A. (2005). Linux Device Drivers (3rd Edition ed.). California: O'Reilly Media.
- The Linux Tutorial. (N.D.). Retrieved 09 18, 2012, from Linux-Tutorial:http://www.linuxtutorial.info/modules.php?name=MContent&pageid=303