

FROM I/O PORTS TO PROCESS MANAGEMENT

3rd Edition
Covers Version 2.6

Understanding the **LINUX** **KERNEL**



O'REILLY®

DANIEL P. BOVET & MARCO CESATI

Chapter 13. I/O Architecture and Device Drivers.....	1
Section 13.1. I/O Architecture.....	1
Section 13.2. The Device Driver Model.....	8
Section 13.3. Device Files.....	18
Section 13.4. Device Drivers.....	22
Section 13.5. Character Device Drivers.....	34



I/O Architecture and Device Drivers

The Virtual File System in the last chapter depends on lower-level functions to carry out each read, write, or other operation in a manner suited to each device. The previous chapter included a brief discussion of how operations are handled by different filesystems. In this chapter, we look at how the kernel invokes the operations on actual devices.

In the section “I/O Architecture,” we give a brief survey of the 80×86 I/O architecture. In the section “The Device Driver Model,” we introduce the Linux device driver model. Next, in the section “Device Files,” we show how the VFS associates a special file called “device file” with each different hardware device, so that application programs can use all kinds of devices in the same way. We then introduce in the section “Device Drivers” some common characteristics of device drivers. Finally, in the section “Character Device Drivers,” we illustrate the overall organization of character device drivers in Linux. We’ll defer the discussion of block device drivers to the next chapters.

Readers interested in developing device drivers on their own may want to refer to Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman’s *Linux Device Drivers*, Third Edition (O’Reilly).

I/O Architecture

To make a computer work properly, data paths must be provided that let information flow between CPU(s), RAM, and the score of I/O devices that can be connected to a personal computer. These data paths, which are denoted as the *buses*, act as the primary communication channels inside the computer.

Any computer has a *system bus* that connects most of the internal hardware devices. A typical system bus is the PCI (*Peripheral Component Interconnect*) bus. Several other types of buses, such as ISA, EISA, MCA, SCSI, and USB, are currently in use. Typically, the same computer includes several buses of different types, linked

together by hardware devices called *bridges*. Two high-speed buses are dedicated to the data transfers to and from the memory chips: the *frontside bus* connects the CPUs to the RAM controller, while the *backside bus* connects the CPUs directly to the external hardware cache. The *host bridge* links together the system bus and the frontside bus.

Any I/O device is hosted by one, and only one, bus. The bus type affects the internal design of the I/O device, as well as how the device has to be handled by the kernel. In this section, we discuss the functional characteristics common to all PC architectures, without giving details about a specific bus type.

The data path that connects a CPU to an I/O device is generically called an *I/O bus*. The 80×86 microprocessors use 16 of their address pins to address I/O devices and 8, 16, or 32 of their data pins to transfer data. The I/O bus, in turn, is connected to each I/O device by means of a hierarchy of hardware components including up to three elements: I/O ports, interfaces, and device controllers. Figure 13-1 shows the components of the I/O architecture.

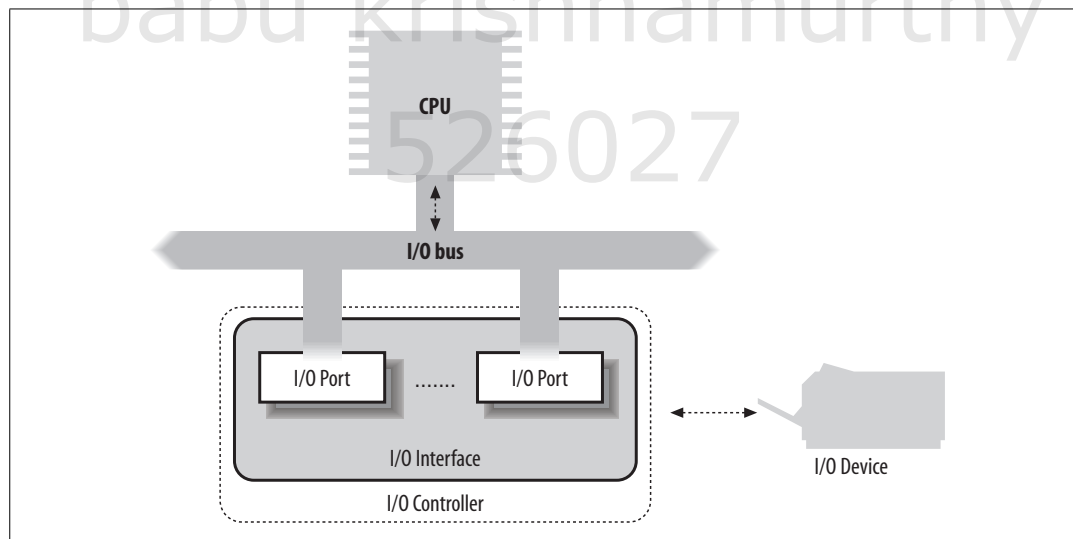


Figure 13-1. PC's I/O architecture

I/O Ports

Each device connected to the I/O bus has its own set of I/O addresses, which are usually called *I/O ports*. In the IBM PC architecture, the *I/O address space* provides up to 65,536 8-bit I/O ports. Two consecutive 8-bit ports may be regarded as a single 16-bit port, which must start on an even address. Similarly, two consecutive 16-bit ports may be regarded as a single 32-bit port, which must start on an address that is a multiple of 4. Four special assembly language instructions called *in*, *ins*, *out*, and *outs* allow the CPU to read from and write into an I/O port. While executing one of these instructions, the CPU selects the required I/O port and transfers the data between a CPU register and the port.

I/O ports may also be mapped into addresses of the physical address space. The processor is then able to communicate with an I/O device by issuing assembly language instructions that operate directly on memory (for instance, `mov`, `and`, `or`, and so on). Modern hardware devices are more suited to mapped I/O, because it is faster and can be combined with DMA.

An important objective for system designers is to offer a unified approach to I/O programming without sacrificing performance. Toward that end, the I/O ports of each device are structured into a set of specialized registers, as shown in Figure 13-2. The CPU writes the commands to be sent to the device into the *device control register* and reads a value that represents the internal state of the device from the *device status register*. The CPU also fetches data from the device by reading bytes from the *device input register* and pushes data to the device by writing bytes into the *device output register*.

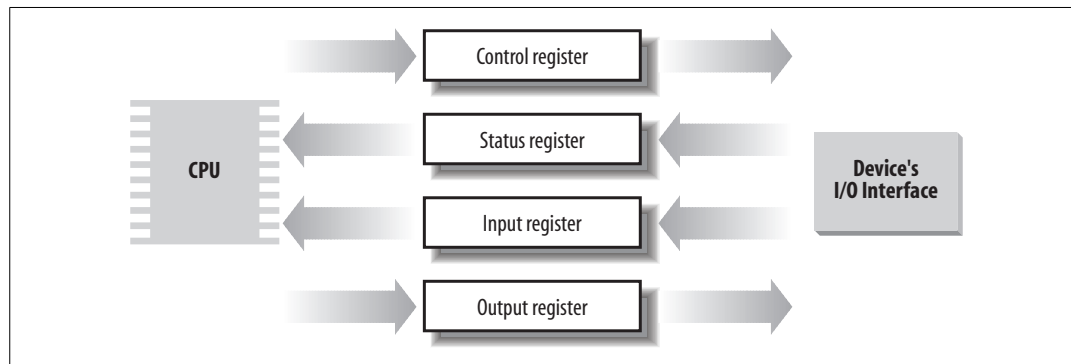


Figure 13-2. Specialized I/O ports

To lower costs, the same I/O port is often used for different purposes. For instance, some bits describe the device state, while others specify the command to be issued to the device. Similarly, the same I/O port may be used as an input register or an output register.

Accessing I/O ports

The `in`, `out`, `ins`, and `outs` assembly language instructions access I/O ports. The following auxiliary functions are included in the kernel to simplify such accesses:

`inb()`, `inw()`, `inl()`

Read 1, 2, or 4 consecutive bytes, respectively, from an I/O port. The suffix “b,” “w,” or “l” refers, respectively, to a byte (8 bits), a word (16 bits), and a long (32 bits).

`inb_p()`, `inw_p()`, `inl_p()`

Read 1, 2, or 4 consecutive bytes, respectively, from an I/O port, and then execute a “dummy” instruction to introduce a pause.

`outb()`, `outw()`, `outl()`

Write 1, 2, or 4 consecutive bytes, respectively, to an I/O port.

`outb_p()`, `outw_p()`, `outl_p()`

Write 1, 2, and 4 consecutive bytes, respectively, to an I/O port, and then execute a “dummy” instruction to introduce a pause.

`insb()`, `insw()`, `insl()`

Read sequences of consecutive bytes in groups of 1, 2, or 4, respectively, from an I/O port. The length of the sequence is specified as a parameter of the functions.

`outsb()`, `outsw()`, `outsl()`

Write sequences of consecutive bytes, in groups of 1, 2, or 4, respectively, to an I/O port.

While accessing I/O ports is simple, detecting which I/O ports have been assigned to I/O devices may not be easy, in particular, for systems based on an ISA bus. Often a device driver must blindly write into some I/O port to probe the hardware device; if, however, this I/O port is already used by some other hardware device, a system crash could occur. To prevent such situations, the kernel keeps track of I/O ports assigned to each hardware device by means of “resources.”

A *resource* represents a portion of some entity that can be exclusively assigned to a device driver. In our case, a resource represents a range of I/O port addresses. The information relative to each resource is stored in a resource data structure, whose fields are shown in Table 13-1. All resources of the same kind are inserted in a tree-like data structure; for instance, all resources representing I/O port address ranges are included in a tree rooted at the node `ioport_resource`.

Table 13-1. The fields of the resource data structure

Type	Field	Description
const char *	name	Description of owner of the resource
unsigned long	start	Start of the resource range
unsigned long	end	End of the resource range
unsigned long	flags	Various flags
struct resource *	parent	Pointer to parent in the resource tree
struct resource *	sibling	Pointer to a sibling in the resource tree
struct resource *	child	Pointer to first child in the resource tree

The children of a node are collected in a list whose first element is pointed to by the `child` field. The `sibling` field points to the next node in the list.

Why use a tree? Well, consider, for instance, the I/O port addresses used by an IDE hard disk interface—let’s say from `0xf000` to `0xf00f`. A resource with the `start` field set to `0xf000` and the `end` field set to `0xf00f` is then included in the tree, and the conventional name of the controller is stored in the `name` field. However, the IDE device

driver needs to remember another bit of information, namely that the subrange from 0xf000 to 0xf007 is used for the master disk of the IDE chain, while the subrange from 0xf008 to 0xf00f is used for the slave disk. To do this, the device driver inserts two children below the resource corresponding to the whole range from 0xf000 to 0xf00f, one child for each subrange of I/O ports. As a general rule, each node of the tree must correspond to a subrange of the range associated with the parent. The root of the I/O port resource tree (`ioport_resource`) spans the whole I/O address space (from port number 0 to 65535).

Each device driver may use the following three functions, passing to them the root node of the resource tree and the address of a resource data structure of interest:

`request_resource()`

Assigns a given range to an I/O device.

`allocate_resource()`

Finds an available range having a given size and alignment in the resource tree; if it exists, assigns the range to an I/O device (mainly used by drivers of PCI devices, which can be configured to use arbitrary port numbers and on-board memory addresses).

`release_resource()`

Releases a given range previously assigned to an I/O device.

The kernel also defines some shortcuts to the above functions that apply to I/O ports: `request_region()` assigns a given interval of I/O ports and `release_region()` releases a previously assigned interval of I/O ports. The tree of all I/O addresses currently assigned to I/O devices can be obtained from the `/proc/iports` file.

I/O Interfaces

An *I/O interface* is a hardware circuit inserted between a group of I/O ports and the corresponding device controller. It acts as an interpreter that translates the values in the I/O ports into commands and data for the device. In the opposite direction, it detects changes in the device state and correspondingly updates the I/O port that plays the role of status register. This circuit can also be connected through an IRQ line to a Programmable Interrupt Controller, so that it issues interrupt requests on behalf of the device.

There are two types of interfaces:

Custom I/O interfaces

Devoted to one specific hardware device. In some cases, the device controller is located in the same *card** that contains the I/O interface. The devices attached to

* Each card must be inserted in one of the available free bus slots of the PC. If the card can be connected to an external device through an external cable, the card supports a suitable connector in the rear panel of the PC.

a custom I/O interface can be either *internal devices* (devices located inside the PC's cabinet) or *external devices* (devices located outside the PC's cabinet).

General-purpose I/O interfaces

Used to connect several different hardware devices. Devices attached to a general-purpose I/O interface are usually external devices.

Custom I/O interfaces

Just to give an idea of how much variety is encompassed by custom I/O interfaces—thus by the devices currently installed in a PC—we'll list some of the most commonly found:

Keyboard interface

Connected to a keyboard controller that includes a dedicated microprocessor. This microprocessor decodes the combination of pressed keys, generates an interrupt, and puts the corresponding scan code in an input register.

Graphic interface

Packed together with the corresponding controller in a graphic card that has its own *frame buffer*, as well as a specialized processor and some code stored in a Read-Only Memory chip (ROM). The frame buffer is an on-board memory containing a description of the current screen contents.

Disk interface

Connected by a cable to the disk controller, which is usually integrated with the disk. For instance, the IDE interface is connected by a 40-wire flat conductor cable to an intelligent disk controller that can be found on the disk itself.

Bus mouse interface

Connected by a cable to the corresponding controller, which is included in the mouse.

Network interface

Packed together with the corresponding controller in a network card used to receive or transmit network packets. Although there are several widely adopted network standards, Ethernet (IEEE 802.3) is the most common.

General-purpose I/O interfaces

Modern PCs include several general-purpose I/O interfaces, which connect a wide range of external devices. The most common interfaces are:

Parallel port

Traditionally used to connect printers, it can also be used to connect removable disks, scanners, backup units, and other computers. The data is transferred 1 byte (8 bits) at a time.

Serial port

Like the parallel port, but the data is transferred 1 bit at a time. It includes a Universal Asynchronous Receiver and Transmitter (UART) chip to string out the

bytes to be sent into a sequence of bits and to reassemble the received bits into bytes. Because it is intrinsically slower than the parallel port, this interface is mainly used to connect external devices that do not operate at a high speed, such as modems, mice, and printers.

PCMCIA interface

Included mostly on portable computers. The external device, which has the shape of a credit card, can be inserted into and removed from a slot without rebooting the system. The most common PCMCIA devices are hard disks, modems, network cards, and RAM expansions.

SCSI (Small Computer System Interface) interface

A circuit that connects the main PC bus to a secondary bus called the *SCSI bus*. The SCSI-2 bus allows up to eight PCs and external devices—hard disks, scanners, CD-ROM writers, and so on—to be connected. Wide SCSI-2 and the SCSI-3 interfaces allow you to connect 16 devices or more if additional interfaces are present. The *SCSI standard* is the communication protocol used to connect devices via the SCSI bus.

Universal serial bus (USB)

A general-purpose I/O interface that operates at a high speed and may be used for the external devices traditionally connected to the parallel port, the serial port, and the SCSI interface.

Device Controllers

A complex device may require a *device controller* to drive it. Essentially, the controller plays two important roles:

- It interprets the high-level commands received from the I/O interface and forces the device to execute specific actions by sending proper sequences of electrical signals to it.
- It converts and properly interprets the electrical signals received from the device and modifies (through the I/O interface) the value of the status register.

A typical device controller is the *disk controller*, which receives high-level commands such as a “write this block of data” from the microprocessor (through the I/O interface) and converts them into low-level disk operations such as “position the disk head on the right track” and “write the data inside the track.” Modern disk controllers are very sophisticated, because they can keep the disk data in on-board fast disk caches and can reorder the CPU high-level requests optimized for the actual disk geometry.

Simpler devices do not have a device controller; examples include the Programmable Interrupt Controller (see the section “Interrupts and Exceptions” in Chapter 4) and the Programmable Interval Timer (see the section “Programmable Interval Timer (PIT)” in Chapter 6).

Several hardware devices include their own memory, which is often called *I/O shared memory*. For instance, all recent graphic cards include tens of megabytes of RAM in the frame buffer, which is used to store the screen image to be displayed on the monitor. We will discuss I/O shared memory in the section “Accessing the I/O Shared Memory” later in this chapter.

The Device Driver Model

Earlier versions of the Linux kernel offered few basic functionalities to the device driver developers: allocating dynamic memory, reserving a range of I/O addresses or an IRQ line, activating an interrupt service routine in response to a device’s interrupt. Older hardware devices, in fact, were cumbersome and difficult to program, and two different hardware devices had little in common even if they were hosted on the same bus. Thus, there was no point in trying to offer a unifying model to the device driver developers.

Things are different now. Bus types such as PCI put strong demands on the internal design of the hardware devices; as a consequence, recent hardware devices, even of different classes, support similar functionalities. Drivers for such devices should typically take care of:

- Power management (handling of different voltage levels on the device’s power line)
- Plug and play (transparent allocation of resources when configuring the device)
- Hot-plugging (support for insertion and removal of the device while the system is running)

Power management is performed globally by the kernel on every hardware device in the system. For instance, when a battery-powered computer enters the “standby” state, the kernel must force every hardware device (hard disks, graphics card, sound card, network card, bus controllers, and so on) in a low-power state. Thus, each driver of a device that can be put in the “standby” state must include a callback function that puts the hardware device in the low-power state. Moreover, the hardware devices must be put in the “standby” state in a precise order, otherwise some devices could be left in the wrong power state. For instance, the kernel must put in “standby” first the hard disks and then their disk controller, because in the opposite case it would be impossible to send commands to the hard disks.

To implement these kinds of operations, Linux 2.6 provides some data structures and helper functions that offer a unifying view of all buses, devices, and device drivers in the system; this framework is called the *device driver model*.

The sysfs Filesystem

The *sysfs* filesystem is a special filesystem similar to */proc* that is usually mounted on the */sys* directory. The */proc* filesystem was the first special filesystem designed to allow User Mode applications to access kernel internal data structures. The */sysfs* filesystem has essentially the same objective, but it provides additional information on kernel data structures; furthermore, */sysfs* is organized in a more structured way than */proc*. Likely, both */proc* and */sysfs* will continue to coexist in the near future.

A goal of the *sysfs* filesystem is to expose the hierarchical relationships among the components of the device driver model. The related top-level directories of this filesystem are:

block

The block devices, independently from the bus to which they are connected.

devices

All hardware devices recognized by the kernel, organized according to the bus in which they are connected.

bus

The buses in the system, which host the devices.

drivers

The device drivers registered in the kernel.

class

The types of devices in the system (audio cards, network cards, graphics cards, and so on); the same class may include devices hosted by different buses and driven by different drivers.

power

Files to handle the power states of some hardware devices.

firmware

Files to handle the firmware of some hardware devices.

Relationships between components of the device driver models are expressed in the *sysfs* filesystem as symbolic links between directories and files. For example, the */sys/block/sda/device* file can be a symbolic link to a subdirectory nested in */sys/devices/pci0000:00* representing the SCSI controller connected to the PCI bus. Moreover, the */sys/block/sda/device/block* file is a symbolic link to */sys/block/sda*, stating that this PCI device is the controller of the SCSI disk.

The main role of regular files in the *sysfs* filesystem is to represent attributes of drivers and devices. For instance, the *dev* file in the */sys/block/hda* directory contains the major and minor numbers of the master disk in the first IDE chain.

Kobjects

The core data structure of the device driver model is a generic data structure named *kobject*, which is inherently tied to the *sysfs* filesystem: each *kobject* corresponds to a directory in that filesystem.

Kobjects are embedded inside larger objects—the so-called “containers”—that describe the components of the device driver model.* The descriptors of buses, devices, and drivers are typical examples of containers; for instance, the descriptor of the first partition in the first IDE disk corresponds to the */sys/block/hda/hda1* directory.

Embedding a *kobject* inside a container allows the kernel to:

- Keep a reference counter for the container
- Maintain hierarchical lists or sets of containers (for instance, a *sysfs* directory associated with a block device includes a different subdirectory for each disk partition)
- Provide a User Mode view for the attributes of the container

Kobjects, ksets, and subsystems

A *kobject* is represented by a *kobject* data structure, whose fields are listed in Table 13-2.

Table 13-2. The fields of the *kobject* data structure

Type	Field	Description
char *	k_name	Pointer to a string holding the name of the container
char []	name	String holding the name of the container, if it fits in 20 bytes
struct k_ref	kref	The reference counter for the container
struct list_head	entry	Pointers for the list in which the <i>kobject</i> is inserted
struct kobject *	parent	Pointer to the parent <i>kobject</i> , if any
struct kset *	kset	Pointer to the containing <i>kset</i>
struct kobj_type *	ktype	Pointer to the <i>kobject</i> type descriptor
struct dentry *	dentry	Pointer to the <i>dentry</i> of the <i>sysfs</i> file associated with the <i>kobject</i>

The *ktype* field points to a *kobj_type* object representing the “type” of the *kobject*—essentially, the type of the container that includes the *kobject*. The *kobj_type* data structure includes three fields: a release method (executed when the *kobject* is being freed), a *sysfs_ops* pointer to a table of *sysfs* operations, and a list of default attributes for the *sysfs* filesystem.

* Kobjects are mainly used to implement the device driver model; however, there is an ongoing effort to change some other kernel components—such as the module subsystem—so as to use them.

The `kref` field is a structure of type `k_ref` consisting of a single `refcount` field. As the name implies, this field is the reference counter for the `kobject`, but it may act also as the reference counter for the container of the `kobject`. The `kobject_get()` and `kobject_put()` functions increase and decrease, respectively, the reference counter; if the counter reaches the value zero, the resources used by the `kobject` are released and the release method of the `kobj_type` object of the `kobject` is executed. This method, which is usually defined only if the container of the `kobject` was allocated dynamically, frees the container itself.

The `kobjects` can be organized in a hierarchical tree by means of *ksets*. A *kset* is a collection of `kobjects` of the same type—that is, included in the same type of container. The fields of the *kset* data structure are listed in Table 13-3.

Table 13-3. The fields of the *kset* data structure

Type	Field	Description
<code>struct subsystem *</code>	<code>subsys</code>	Pointer to the subsystem descriptor
<code>struct kobj_type *</code>	<code>ktype</code>	Pointer to the <code>kobject</code> type descriptor of the <i>kset</i>
<code>struct list_head</code>	<code>list</code>	Head of the list of <code>kobjects</code> included in the <i>kset</i>
<code>struct kobject</code>	<code>kobj</code>	Embedded <code>kobject</code> (see text)
<code>struct kset_hotplug_ops *</code>	<code>hotplug_ops</code>	Pointer to a table of callback functions for <code>kobject</code> filtering and hot-plugging

The `list` field is the head of the doubly linked circular list of `kobjects` included in the *kset*; the `ktype` field points to the same `kobj_type` descriptor shared by all `kobjects` in the *kset*.

The `kobj` field is a `kobject` embedded in the *kset* data structure; the `parent` field of the `kobjects` contained in the *kset* points to this embedded `kobject`. Thus, a *kset* is a collection of `kobjects`, but it relies on a `kobject` of higher level for reference counting and linking in the hierarchical tree. This design choice is code-efficient and allows the greatest flexibility. For instance, the `kset_get()` and `kset_put()` functions, which increase and decrease respectively the reference counter of the *kset*, simply invoke `kobject_get()` and `kobject_put()` on the embedded `kobject`; because the reference counter of a *kset* is merely the reference counter of the `kobj` `kobject` embedded in the *kset*. Moreover, thanks to the embedded `kobject`, the *kset* data structure can be embedded in a “container” object, exactly as for the `kobject` data structure. Finally, a *kset* can be made a member of another *kset*: it suffices to insert the embedded `kobject` in the higher-level *kset*.

Collections of *ksets* called *subsystems* also exist. A *subsystem* may include *ksets* of different types, and it is represented by a *subsystem* data structure having just two fields:

`kset`

An embedded *kset* that stores the *ksets* included in the *subsystem*

rwsem

A read-write semaphore that protects all ksets and kobjects recursively included in the subsystem

Even the subsystem data structure can be embedded in a larger “container” object; the reference counter of the container is thus the reference counter of the embedded subsystem—that is, the reference counter of the kobject embedded in the kset embedded in the subsystem. The `subsys_get()` and `subsys_put()` functions respectively increase and decrease this reference counter.

Figure 13-3 illustrates an example of the device driver model hierarchy. The *bus* subsystem includes a *pci* subsystem, which, in turn, includes a *drivers* kset. This kset contains a *serial* kobject—corresponding to the device driver for the serial port—having a single *new-id* attribute.

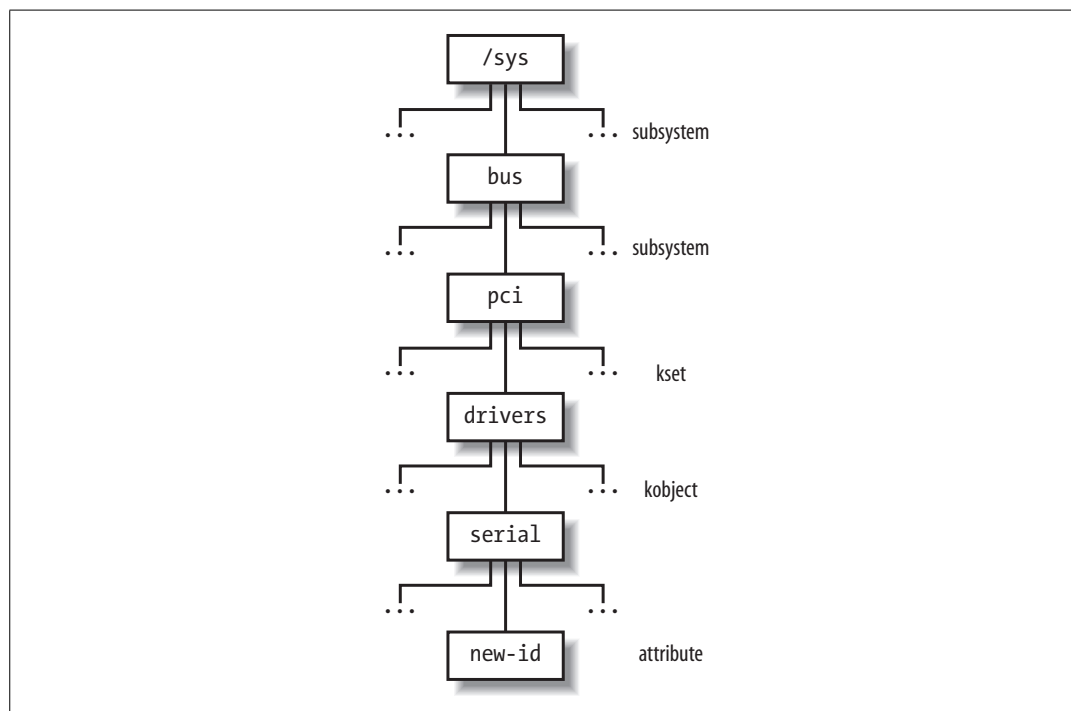


Figure 13-3. An example of device driver model hierarchy

Registering kobjects, ksets, and subsystems

As a general rule, if you want a kobject, kset, or subsystem to appear in the `sysfs` subtree, you must first *register* it. The directory associated with a kobject always appears in the directory of the parent kobject. For instance, the directories of kobjects included in the same kset appear in the directory of the kset itself. Therefore, the structure of the `sysfs` subtree represents the hierarchical relationships between the various registered kobjects and, consequently, between the various container objects.

Usually, the top-level directories of the *sysfs* filesystem are associated with the registered subsystems.

The `kobject_register()` function initializes a `kobject` and adds the corresponding directory to the *sysfs* filesystem. Before invoking it, the caller should set the `kset` field in the `kobject` so that it points to the parent `kset`, if any. The `kobject_unregister()` function removes a `kobject`'s directory from the *sysfs* filesystem. To make life easier for kernel developers, Linux also offers the `kset_register()` and `kset_unregister()` functions, and the `subsystem_register()` and `subsystem_unregister()` functions, but they are essentially wrapper functions around `kobject_register()` and `kobject_unregister()`.

As stated before, many `kobject` directories include regular files called *attributes*. The `sysfs_create_file()` function receives as its parameters the addresses of a `kobject` and an attribute descriptor, and creates the special file in the proper directory. Other relationships between the objects represented in the *sysfs* filesystem are established by means of symbolic links: the `sysfs_create_link()` function creates a symbolic link for a given `kobject` in a directory associated with another `kobject`.

Components of the Device Driver Model

The device driver model is built upon a handful of basic data structures, which represent buses, devices, device drivers, etc. Let us examine them.

Devices

Each device in the device driver model is represented by a device object, whose fields are shown in Table 13-4.

Table 13-4. The fields of the device object

Type	Field	Description
struct list_head	node	Pointers for the list of sibling devices
struct list_head	bus_list	Pointers for the list of devices on the same bus type
struct list_head	driver_list	Pointers for the driver's list of devices
struct list_head	children	Head of the list of children devices
struct device *	parent	Pointer to the parent device
struct kobject	kobj	Embedded <code>kobject</code>
char []	bus_id	Device position on the hosting bus
struct bus_type *	bus	Pointer to the hosting bus
struct device_driver *	driver	Pointer to the controlling device driver
void *	driver_data	Pointer to private data for the driver
void *	platform_data	Pointer to private data for legacy device drivers

Table 13-4. The fields of the device object (continued)

Type	Field	Description
struct dev_pm_info	power	Power management information
unsigned long	detach_state	Power state to be entered when unloading the device driver
unsigned long long *	dma_mask	Pointer to the DMA mask of the device (see the later section “Direct Memory Access (DMA)”)
unsigned long long	coherent_dma_mask	Mask for coherent DMA of the device
struct list_head	dma_pools	Head of a list of aggregate DMA buffers
struct dma_coherent_mem *	dma_mem	Pointer to a descriptor of the coherent DMA memory used by the device (see the later section “Direct Memory Access (DMA)”)
void (*)(struct device *)	release	Callback function for releasing the device descriptor

The device objects are globally collected in the `devices_subsys` subsystem, which is associated with the `/sys/devices` directory (see the earlier section “Kobjects”). The devices are organized hierarchically: a device is the “parent” of some “children” devices if the children devices cannot work properly without the parent device. For instance, in a PCI-based computer, a bridge between the PCI bus and the USB bus is the parent device of every device hosted on the USB bus. The parent field of the device object points to the descriptor of the parent device, the children field is the head of the list of children devices, and the node field stores the pointers to the adjacent elements in the children list. The parenthood relationships between the kobjects embedded in the device objects reflect also the device hierarchy; thus, the structure of the directories below `/sys/devices` matches the physical organization of the hardware devices.

Each driver keeps a list of device objects including all managed devices; the `driver_list` field of the device object stores the pointers to the adjacent elements, while the `driver` field points to the descriptor of the device driver. For each bus type, moreover, there is a list including all devices that are hosted on the buses of the given type; the `bus_list` field of the device object stores the pointers to the adjacent elements, while the `bus` field points to the bus type descriptor.

A reference counter keeps track of the usage of the device object; it is included in the `kobj` kobject embedded in the descriptor. The counter is increased by invoking `get_device()`, and it is decreased by invoking `put_device()`.

The `device_register()` function inserts a new device object in the device driver model, and automatically creates a new directory for it under `/sys/devices`. Conversely, the `device_unregister()` function removes a device from the device driver model.

Usually, the device object is statically embedded in a larger descriptor. For instance, PCI devices are described by `pci_dev` data structures; the `dev` field of this structure is

a device object, while the other fields are specific to the PCI bus. The `device_register()` and `device_unregister()` functions are executed when the device is being registered or de-registered in the PCI kernel layer.

Drivers

Each driver in the device driver model is described by a `device_driver` object, whose fields are listed in Table 13-5.

Table 13-5. The fields of the `device_driver` object

Type	Field	Description
char *	name	Name of the device driver
struct bus_type *	bus	Pointer to descriptor of the bus that hosts the supported devices
struct semaphore	unload_sem	Semaphore to forbid device driver unloading; it is released when the reference counter reaches zero
struct kobject	kobj	Embedded kobject
struct list_head	devices	Head of the list including all devices supported by the driver
struct module *	owner	Identifies the module that implements the device driver, if any (see Appendix B)
int (*)(struct device *)	probe	Method for probing a device (checking that it can be handled by the device driver)
int (*)(struct device *)	remove	Method invoked on a device when it is removed
void (*)(struct device *)	shutdown	Method invoked on a device when it is powered off (shut down)
int (*)(struct device *, unsigned long, unsigned long)	suspend	Method invoked on a device when it is put in low-power state
int (*)(struct device *, unsigned long)	resume	Method invoked on a device when it is put back in the normal state (full power)

The `device_driver` object includes four methods for handling hot-plugging, plug and play, and power management. The probe method is invoked whenever a bus device driver discovers a device that could possibly be handled by the driver; the corresponding function should probe the hardware to perform further checks on the device. The remove method is invoked on a hot-pluggable device whenever it is removed; it is also invoked on every device handled by the driver when the driver itself is unloaded. The shutdown, suspend, and resume methods are invoked on a device when the kernel must change its power state.

The reference counter included in the `kobj` kobject embedded in the descriptor keeps track of the usage of the `device_driver` object. The counter is increased by invoking `get_driver()`, and it is decreased by invoking `put_driver()`.

The `driver_register()` function inserts a new `device_driver` object in the device driver model, and automatically creates a new directory for it in the `sysfs` filesystem. Conversely, the `driver_unregister()` function removes a driver from the device driver model.

Usually, the `device_driver` object is statically embedded in a larger descriptor. For instance, PCI device drivers are described by `pci_driver` data structures; the `driver` field of this structure is a `device_driver` object, while the other fields are specific to the PCI bus.

Buses

Each bus type supported by the kernel is described by a `bus_type` object, whose fields are listed in Table 13-6.

Table 13-6. The fields of the `bus_type` object

Type	Field	Description
<code>char *</code>	<code>name</code>	Name of the bus type
<code>struct subsystem</code>	<code>subsys</code>	Kobject subsystem associated with this bus type
<code>struct kset</code>	<code>drivers</code>	The set of kobjects of the drivers
<code>struct kset</code>	<code>devices</code>	The set of kobjects of the devices
<code>struct bus_attribute *</code>	<code>bus_attrs</code>	Pointer to the object including the bus attributes and the methods for exporting them to the <code>sysfs</code> filesystem
<code>struct device_attribute *</code>	<code>dev_attrs</code>	Pointer to the object including the device attributes and the methods for exporting them to the <code>sysfs</code> filesystem
<code>struct driver_attribute *</code>	<code>drv_attrs</code>	Pointer to the object including the device driver attributes and the methods for exporting them to the <code>sysfs</code> filesystem
<code>int (*)(struct device *, struct device_driver *)</code>	<code>match</code>	Method for checking whether a given driver supports a given device
<code>int (*)(struct device *, char **, int, char *, int)</code>	<code>hotplug</code>	Method invoked when a device is being registered
<code>int (*)(struct device *, unsigned long)</code>	<code>suspend</code>	Method for saving the hardware context state and changing the power level of a device
<code>int (*)(struct device *)</code>	<code>resume</code>	Method for changing the power level and restoring the hardware context of a device

Each `bus_type` object includes an embedded subsystem; the subsystem stored in the `bus_subsys` variable collects all subsystems embedded in the `bus_type` objects. The `bus_subsys` subsystem is associated with the `/sys/bus` directory; thus, for example, there exists a `/sys/bus/pci` directory associated with the PCI bus type. The per-bus subsystem typically includes only two ksets named *drivers* and *devices* (corresponding to the `drivers` and `devices` fields of the `bus_type` object, respectively).

The `drivers kset` contains the `device_driver` descriptors of all device drivers pertaining to the bus type, while the `devices kset` contains the device descriptors of all devices of the given bus type. Because the directories of the devices' `kobjects` already appear in the `sysfs` filesystem under `/sys/devices`, the `devices` directory of the per-bus subsystem stores symbolic links pointing to directories under `/sys/devices`. The `bus_for_each_drv()` and `bus_for_each_dev()` functions iterate over the elements of the lists of drivers and devices, respectively.

The `match` method is executed when the kernel must check whether a given device can be handled by a given driver. Even if each device's identifier has a format specific to the bus that hosts the device, the function that implements the method is usually simple, because it searches the device's identifier in the driver's table of supported identifiers. The `hotplug` method is executed when a device is being registered in the device driver model; the implementing function should add bus-specific information to be passed as environment variables to a User Mode program that is notified about the new available device (see the later section "Device Driver Registration"). Finally, the `suspend` and `resume` methods are executed when a device on a bus of the given type must change its power state.

Classes

Each class is described by a class object. All class objects belong to the `class_subsys` subsystem associated with the `/sys/class` directory. Each class object, moreover, includes an embedded subsystem; thus, for example, there exists a `/sys/class/input` directory associated with the `input` class of the device driver model.

Each class object includes a list of `class_device` descriptors, each of which represents a single *logical device* belonging to the class. The `class_device` structure includes a `dev` field that points to a device descriptor, thus a logical device always refers to a given device in the device driver model. However, there can be several `class_device` descriptors that refer to the same device. In fact, a hardware device might include several different sub-devices, each of which requires a different User Mode interface. For example, the sound card is a hardware device that usually includes a DSP, a mixer, a game port interface, and so on; each sub-device requires its own User Mode interface, thus it is associated with its own directory in the `sysfs` filesystem.

Device drivers in the same class are expected to offer the same functionalities to the User Mode applications; for instance, all device drivers of sound cards should offer a way to write sound samples to the DSP.

The classes of the device driver model are essentially aimed to provide a standard method for exporting to User Mode applications the interfaces of the logical devices. Each `class_device` descriptor embeds a `kobject` having an attribute (special file) named `dev`. Such attribute stores the major and minor numbers of the device file that is needed to access to the corresponding logical device (see the next section).

Device Files

As mentioned in Chapter 1, Unix-like operating systems are based on the notion of a file, which is just an information container structured as a sequence of bytes. According to this approach, I/O devices are treated as special files called *device files*; thus, the same system calls used to interact with regular files on disk can be used to directly interact with I/O devices. For example, the same `write()` system call may be used to write data into a regular file or to send it to a printer by writing to the `/dev/lp0` device file.

According to the characteristics of the underlying device drivers, device files can be of two types: *block* or *character*. The difference between the two classes of hardware devices is not so clear-cut. At least we can assume the following:

- The data of a block device can be addressed randomly, and the time needed to transfer a data block is small and roughly the same, at least from the point of view of the human user. Typical examples of block devices are hard disks, floppy disks, CD-ROM drives, and DVD players.
- The data of a character device either cannot be addressed randomly (consider, for instance, a sound card), or they can be addressed randomly, but the time required to access a random datum largely depends on its position inside the device (consider, for instance, a magnetic tape driver).

Network cards are a notable exception to this schema, because they are hardware devices that are not directly associated with device files.

Device files have been in use since the early versions of the Unix operating system. A device file is usually a real file stored in a filesystem. Its inode, however, doesn't need to include pointers to blocks of data on the disk (the file's data) because there are none. Instead, the inode must include an identifier of the hardware device corresponding to the character or block device file.

Traditionally, this identifier consists of the type of device file (character or block) and a pair of numbers. The first number, called the *major number*, identifies the device type. Traditionally, all device files that have the same major number and the same type share the same set of file operations, because they are handled by the same device driver. The second number, called the *minor number*, identifies a specific device among a group of devices that share the same major number. For instance, a group of disks managed by the same disk controller have the same major number and different minor numbers.

The `mknod()` system call is used to create device files. It receives the name of the device file, its type, and the major and minor numbers as its parameters. Device files are usually included in the `/dev` directory. Table 13-7 illustrates the attributes of some device files. Notice that character and block devices have independent numbering, so block device (3,0) is different from character device (3,0).

Table 13-7. Examples of device files

Name	Type	Major	Minor	Description
<code>/dev/fd0</code>	block	2	0	Floppy disk
<code>/dev/hda</code>	block	3	0	First IDE disk
<code>/dev/hda2</code>	block	3	2	Second primary partition of first IDE disk
<code>/dev/hdb</code>	block	3	64	Second IDE disk
<code>/dev/hdb3</code>	block	3	67	Third primary partition of second IDE disk
<code>/dev/tty0</code>	char	3	0	Terminal
<code>/dev/console</code>	char	5	1	Console
<code>/dev/lp1</code>	char	6	1	Parallel printer
<code>/dev/ttyS0</code>	char	4	64	First serial port
<code>/dev/rtc</code>	char	10	135	Real-time clock
<code>/dev/null</code>	char	1	3	Null device (black hole)

Usually, a device file is associated with a hardware device (such as a hard disk—for instance, `/dev/hda`) or with some physical or logical portion of a hardware device (such as a disk partition—for instance, `/dev/hda2`). In some cases, however, a device file is not associated with any real hardware device, but represents a fictitious logical device. For instance, `/dev/null` is a device file corresponding to a “black hole;” all data written into it is simply discarded, and the file always appears empty.

As far as the kernel is concerned, the name of the device file is irrelevant. If you create a device file named `/tmp/disk` of type “block” with the major number 3 and minor number 0, it would be equivalent to the `/dev/hda` device file shown in the table. On the other hand, device filenames may be significant for some application programs. For example, a communication program might assume that the first serial port is associated with the `/dev/ttyS0` device file. But most application programs can be configured to interact with arbitrarily named device files.

User Mode Handling of Device Files

In traditional Unix systems (and in earlier versions of Linux), the major and minor numbers of the device files are 8 bits long. Thus, there could be at most 65,536 block device files and 65,536 character device files. You might expect they will suffice, but unfortunately they don’t.

The real problem is that device files are traditionally allocated once and forever in the `/dev` directory; therefore, each logical device in the system should have an associated device file with a well-defined device number. The official registry of allocated device numbers and `/dev` directory nodes is stored in the `Documentation/devices.txt` file; the macros corresponding to the major numbers of the devices may also be found in the `include/linux/major.h` file.

Unfortunately, the number of different hardware devices is so large nowadays that almost all device numbers have already been allocated. The official registry of device numbers works well for the average Linux system; however, it may not be well suited for large-scale systems. Furthermore, high-end systems may use hundreds or thousands of disks of the same type, and an 8-bit minor number is not sufficient. For instance, the registry reserves device numbers for 16 SCSI disks having 15 partitions each; if a high-end system has more than 16 SCSI disks, the standard assignment of major and minor numbers has to be changed—a non trivial task that requires modifying the kernel source code and makes the system hard to maintain.

In order to solve this kind of problem, the size of the device numbers has been increased in Linux 2.6: the major number is now encoded in 12 bits, while the minor number is encoded in 20 bits. Both numbers are usually kept in a single 32-bit variable of type `dev_t`; the `MAJOR` and `MINOR` macros extract the major and minor numbers, respectively, from a `dev_t` value, while the `MKDEV` macro encodes the two device numbers in a `dev_t` value. For backward compatibility, the kernel handles properly old device files encoded with 16-bit device numbers.

The additional available device numbers are not being statically allocated in the official registry, because they should be used only when dealing with unusual demands for device numbers. Actually, today's preferred way to deal with device files is highly dynamic, both in the device number assignment and in the device file creation.

Dynamic device number assignment

Each device driver specifies in the registration phase the range of device numbers that it is going to handle (see the later section “Device Driver Registration”). The driver can, however, require the allocation of an interval of device numbers without specifying the exact values: in this case, the kernel allocates a suitable range of numbers and assigns them to the driver.

Therefore, device drivers of new hardware devices no longer require an assignment in the official registry of device numbers; they can simply use whatever numbers are currently available in the system.

In this case, however, the device file cannot be created once and forever; it must be created right after the device driver initialization with the proper major and minor numbers. Thus, there must be a standard way to export the device numbers used by each driver to the User Mode applications. As we have seen in the earlier section “Components of the Device Driver Model,” the device driver model provides an elegant solution: the major and minor numbers are stored in the *dev* attributes contained in the subdirectories of */sys/class*.

Dynamic device file creation

The Linux kernel can create the device files dynamically: there is no need to fill the */dev* directory with the device files of every conceivable hardware device, because the device

files can be created “on demand.” Thanks to the device driver model, the kernel 2.6 offers a very simple way to do so. A set of User Mode programs, collectively known as the *udev* toolset, must be installed in the system. At the system startup the */dev* directory is emptied, then a *udev* program scans the subdirectories of */sys/class* looking for the *dev* files. For each such file, which represents a combination of major and minor number for a logical device supported by the kernel, the program creates a corresponding device file in */dev*. It also assigns device filenames and creates symbolic links according to a configuration file, in such a way to resemble the traditional naming scheme for Unix device files. Eventually, */dev* is filled with the device files of all devices supported by the kernel on this system, and nothing else.

Often a device file is created after the system has been initialized. This happens either when a module containing a device driver for a still unsupported device is loaded, or when a hot-pluggable device—such as a USB peripheral—is plugged in the system. The *udev* toolset can automatically create the corresponding device file, because the device driver model supports *device hotplugging*. Whenever a new device is discovered, the kernel spawns a new process that executes the User Mode */sbin/hotplug* shell script,* passing to it any useful information on the discovered device as environment variables. The User Mode scripts usually reads a configuration file and takes care of any operation required to complete the initialization of the new device. If *udev* is installed, the script also creates the proper device file in the */dev* directory.

VFS Handling of Device Files

Device files live in the system directory tree but are intrinsically different from regular files and directories. When a process accesses a regular file, it is accessing some data blocks in a disk partition through a filesystem; when a process accesses a device file, it is just driving a hardware device. For instance, a process might access a device file to read the room temperature from a digital thermometer connected to the computer. It is the VFS’s responsibility to hide the differences between device files and regular files from application programs.

To do this, the VFS changes the default file operations of a device file when it is opened; as a result, each system call on the device file is translated to an invocation of a device-related function instead of the corresponding function of the hosting filesystem. The device-related function acts on the hardware device to perform the operation requested by the process.†

Let’s suppose that a process executes an `open()` system call on a device file (either of type block or character). The operations performed by the system call have already

* The pathname of the User Mode program invoked on hot-plugging events can be changed by writing into the */proc/sys/kernel/hotplug* file.

† Notice that, thanks to the name-resolving mechanism explained in the section “Pathname Lookup” in Chapter 12, symbolic links to device files work just like device files.

been described in the section “The `open()` System Call” in Chapter 12. Essentially, the corresponding service routine resolves the pathname to the device file and sets up the corresponding inode object, dentry object, and file object.

The inode object is initialized by reading the corresponding inode on disk through a suitable function of the filesystem (usually `ext2_read_inode()` or `ext3_read_inode()`; see Chapter 18). When this function determines that the disk inode is relative to a device file, it invokes `init_special_inode()`, which initializes the `i_rdev` field of the inode object to the major and minor numbers of the device file, and sets the `i_fop` field of the inode object to the address of either the `def_blk_fops` or the `def_chr_fops` file operation table, according to the type of device file. The service routine of the `open()` system call also invokes the `dentry_open()` function, which allocates a new file object and sets its `f_op` field to the address stored in `i_fop`—that is, to the address of `def_blk_fops` or `def_chr_fops` once again. Thanks to these two tables, every system call issued on a device file will activate a device driver’s function rather than a function of the underlying filesystem.

Device Drivers

A *device driver* is the set of kernel routines that makes a hardware device respond to the programming interface defined by the canonical set of VFS functions (*open*, *read*, *lseek*, *ioctl*, and so forth) that control a device. The actual implementation of all these functions is delegated to the device driver. Because each device has a different I/O controller, and thus different commands and different state information, most I/O devices have their own drivers.

There are many types of device drivers. They mainly differ in the level of support that they offer to the User Mode applications, as well as in their buffering strategies for the data collected from the hardware devices. Because these choices greatly influence the internal structure of a device driver, we discuss them in the sections “Direct Memory Access (DMA)” and “Buffering Strategies for Character Devices.”

A device driver does not consist only of the functions that implement the device file operations. Before using a device driver, several activities must have taken place. We’ll examine them in the following sections.

Device Driver Registration

We know that each system call issued on a device file is translated by the kernel into an invocation of a suitable function of a corresponding device driver. To achieve this, a device driver must *register* itself. In other words, registering a device driver means allocating a new `device_driver` descriptor, inserting it in the data structures of the device driver model (see the earlier section “Components of the Device Driver Model”), and linking it to the corresponding device file(s). Accesses to device files

whose corresponding drivers have not been previously registered return the error code `-ENODEV`.

If a device driver is statically compiled in the kernel, its registration is performed during the kernel initialization phase. Conversely, if a device driver is compiled as a kernel module (see Appendix B), its registration is performed when the module is loaded. In the latter case, the device driver can also unregister itself when the module is unloaded.

Let us consider, for instance, a generic PCI device. To properly handle it, its device driver must allocate a descriptor of type `pci_driver`, which is used by the PCI kernel layer to handle the device. After having initialized some fields of this descriptor, the device driver invokes the `pci_register_driver()` function. Actually, the `pci_driver` descriptor includes an embedded `device_driver` descriptor (see the earlier section “Components of the Device Driver Model”); the `pci_register_function()` simply initializes the fields of the embedded driver descriptor and invokes `driver_register()` to insert the driver in the data structures of the device driver model.

When a device driver is being registered, the kernel looks for unsupported hardware devices that could be possibly handled by the driver. To do this, it relies on the `match` method of the relevant `bus_type` bus type descriptor, and on the `probe` method of the `device_driver` object. If a hardware device that can be handled by the driver is discovered, the kernel allocates a device object and invokes `device_register()` to insert the device in the device driver model.

Device Driver Initialization

Registering a device driver and initializing it are two different things. A device driver is registered as soon as possible, so User Mode applications can use it through the corresponding device files. In contrast, a device driver is initialized at the last possible moment. In fact, initializing a driver means allocating precious resources of the system, which are therefore not available to other drivers.

We already have seen an example in the section “I/O Interrupt Handling” in Chapter 4: the assignment of IRQs to devices is usually made dynamically, right before using them, because several devices may share the same IRQ line. Other resources that can be allocated at the last possible moment are page frames for DMA transfer buffers and the DMA channel itself (for old non-PCI devices such as the floppy disk driver).

To make sure the resources are obtained when needed but are not requested in a redundant manner when they have already been granted, device drivers usually adopt the following schema:

- A usage counter keeps track of the number of processes that are currently accessing the device file. The counter is increased in the open method of the device file and decreased in the release method.*
- The open method checks the value of the usage counter before the increment. If the counter is zero, the device driver must allocate the resources and enable interrupts and DMA on the hardware device.
- The release method checks the value of the usage counter after the decrement. If the counter is zero, no more processes are using the hardware device. If so, the method disables interrupts and DMA on the I/O controller, and then releases the allocated resources.

Monitoring I/O Operations

The duration of an I/O operation is often unpredictable. It can depend on mechanical considerations (the current position of a disk head with respect to the block to be transferred), on truly random events (when a data packet arrives on the network card), or on human factors (when a user presses a key on the keyboard or when she notices that a paper jam occurred in the printer). In any case, the device driver that started an I/O operation must rely on a monitoring technique that signals either the termination of the I/O operation or a time-out.

In the case of a terminated operation, the device driver reads the status register of the I/O interface to determine whether the I/O operation was carried out successfully. In the case of a time-out, the driver knows that something went wrong, because the maximum time interval allowed to complete the operation elapsed and nothing happened.

The two techniques available to monitor the end of an I/O operation are called the *polling mode* and the *interrupt mode*.

Polling mode

According to this technique, the CPU checks (polls) the device's status register repeatedly until its value signals that the I/O operation has been completed. We have already encountered a technique based on polling in the section "Spin Locks" in Chapter 5: when a processor tries to acquire a busy spin lock, it repeatedly polls the variable until its value becomes 0. However, polling applied to I/O operations is usually more elaborate, because the driver must also remember to check for possible time-outs. A simple example of polling looks like the following:

```
for (;;) {
    if (read_status(device) & DEVICE_END_OPERATION) break;
```

* More precisely, the usage counter keeps track of the number of file objects referring to the device file, because clone processes could share the same file object.

```

        if (--count == 0) break;
    }

```

The count variable, which was initialized before entering the loop, is decreased at each iteration, and thus can be used to implement a rough time-out mechanism. Alternatively, a more precise time-out mechanism could be implemented by reading the value of the tick counter `jiffies` at each iteration (see the section “Updating the Time and Date” in Chapter 6) and comparing it with the old value read before starting the wait loop.

If the time required to complete the I/O operation is relatively high, say in the order of milliseconds, this schema becomes inefficient because the CPU wastes precious machine cycles while waiting for the I/O operation to complete. In such cases, it is preferable to voluntarily relinquish the CPU after each polling operation by inserting an invocation of the `schedule()` function inside the loop.

Interrupt mode

Interrupt mode can be used only if the I/O controller is capable of signaling, via an IRQ line, the end of an I/O operation.

We’ll show how interrupt mode works on a simple case. Let’s suppose we want to implement a driver for a simple input character device. When the user issues a `read()` system call on the corresponding device file, an input command is sent to the device’s control register. After an unpredictably long time interval, the device puts a single byte of data in its input register. The device driver then returns this byte as the result of the `read()` system call.

This is a typical case in which it is preferable to implement the driver using the interrupt mode. Essentially, the driver includes two functions:

1. The `foo_read()` function that implements the read method of the file object.
2. The `foo_interrupt()` function that handles the interrupt.

The `foo_read()` function is triggered whenever the user reads the device file:

```

ssize_t foo_read(struct file *filp, char *buf, size_t count,
                 loff_t *ppos)
{
    foo_dev_t * foo_dev = filp->private_data;
    if (down_interruptible(&foo_dev->sem)
        return -ERESTARTSYS;
    foo_dev->intr = 0;
    outb(DEV_FOO_READ, DEV_FOO_CONTROL_PORT);
    wait_event_interruptible(foo_dev->wait, (foo_dev->intr==1));
    if (put_user(foo_dev->data, buf))
        return -EFAULT;
    up(&foo_dev->sem);
    return 1;
}

```

The device driver relies on a custom descriptor of type `foo_dev_t`; it includes a semaphore `sem` that protects the hardware device from concurrent accesses, a wait queue `wait`, a flag `intr` that is set when the device issues an interrupt, and a single-byte buffer `data` that is written by the interrupt handler and read by the read method. In general, all I/O drivers that use interrupts rely on data structures accessed by both the interrupt handler and the read and write methods. The address of the `foo_dev_t` descriptor is usually stored in the `private_data` field of the device file's file object or in a global variable.

The main operations of the `foo_read()` function are the following:

1. Acquires the `foo_dev->sem` semaphore, thus ensuring that no other process is accessing the device.
2. Clears the `intr` flag.
3. Issues the read command to the I/O device.
4. Executes `wait_event_interruptible` to suspend the process until the `intr` flag becomes 1. This macro is described in the section “Wait queues” in Chapter 3.

After some time, our device issues an interrupt to signal that the I/O operation is completed and that the data is ready in the proper `DEV_FOO_DATA_PORT` data port. The interrupt handler sets the `intr` flag and wakes the process. When the scheduler decides to reexecute the process, the second part of `foo_read()` is executed and does the following:

1. Copies the character ready in the `foo_dev->data` variable into the user address space.
2. Terminates after releasing the `foo_dev->sem` semaphore.

For simplicity, we didn't include any time-out control. In general, time-out control is implemented through static or dynamic timers (see Chapter 6); the timer must be set to the right time before starting the I/O operation and removed when the operation terminates.

Let's now look at the code of the `foo_interrupt()` function:

```
irqreturn_t foo_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    foo->data = inb(DEV_FOO_DATA_PORT);
    foo->intr = 1;
    wake_up_interruptible(&foo->wait);
    return 1;
}
```

The interrupt handler reads the character from the input register of the device and stores it in the `data` field of the `foo_dev_t` descriptor of the device driver pointed to by the `foo` global variable. It then sets the `intr` flag and invokes `wake_up_interruptible()` to wake the process blocked in the `foo->wait` wait queue.

Notice that none of the three parameters are used by our interrupt handler. This is a rather common case.

Accessing the I/O Shared Memory

Depending on the device and on the bus type, I/O shared memory in the PC's architecture may be mapped within different physical address ranges. Typically:

For most devices connected to the ISA bus

The I/O shared memory is usually mapped into the 16-bit physical addresses ranging from 0xa0000 to 0xfffff; this gives rise to the “hole” between 640 KB and 1 MB mentioned in the section “Physical Memory Layout” in Chapter 2.

For devices connected to the PCI bus

The I/O shared memory is mapped into 32-bit physical addresses near the 4 GB boundary. This kind of device is much simpler to handle.

A few years ago, Intel introduced the *Accelerated Graphics Port* (AGP) standard, which is an enhancement of PCI for high-performance graphic cards. Beside having its own I/O shared memory, this kind of card is capable of directly addressing portions of the motherboard's RAM by means of a special hardware circuit named *Graphics Address Remapping Table* (GART). The GART circuitry enables AGP cards to sustain much higher data transfer rates than older PCI cards. From the kernel's point of view, however, it doesn't really matter where the physical memory is located, and GART-mapped memory is handled like the other kinds of I/O shared memory.

How does a device driver access an I/O shared memory location? Let's start with the PC's architecture, which is relatively simple to handle, and then extend the discussion to other architectures.

Remember that kernel programs act on linear addresses, so the I/O shared memory locations must be expressed as addresses greater than `PAGE_OFFSET`. In the following discussion, we assume that `PAGE_OFFSET` is equal to `0xc0000000`—that is, that the kernel linear addresses are in the fourth gigabyte.

Device drivers must translate I/O physical addresses of I/O shared memory locations into linear addresses in kernel space. In the PC architecture, this can be achieved simply by ORing the 32-bit physical address with the `0xc0000000` constant. For instance, suppose the kernel needs to store the value in the I/O location at physical address `0x00b0fe4` in `t1` and the value in the I/O location at physical address `0xfc000000` in `t2`. One might think that the following statements could do the job:

```
t1 = *((unsigned char *) (0xc00b0fe4));  
t2 = *((unsigned char *) (0xfc000000));
```

During the initialization phase, the kernel maps the available RAM's physical addresses into the initial portion of the fourth gigabyte of the linear address space.

Therefore, the Paging Unit maps the 0xc00b0fe4 linear address appearing in the first statement back to the original I/O physical address 0x000b0fe4, which falls inside the “ISA hole” between 640 KB and 1 MB (see the section “Paging in Linux” in Chapter 2). This works fine.

There is a problem, however, for the second statement, because the I/O physical address is greater than the last physical address of the system RAM. Therefore, the 0xfc000000 linear address does not correspond to the 0xfc000000 physical address. In such cases, the kernel Page Tables must be modified to include a linear address that maps the I/O physical address. This can be done by invoking the `ioremap()` or `ioremap_nocache()` functions. The first function, which is similar to `vmalloc()`, invokes `get_vm_area()` to create a new `vm_struct` descriptor (see the section “Descriptors of Noncontiguous Memory Areas” in Chapter 8) for a linear address interval that has the size of the required I/O shared memory area. The functions then update the corresponding Page Table entries of the canonical kernel Page Tables appropriately. The `ioremap_nocache()` function differs from `ioremap()` in that it also disables the hardware cache when referencing the remapped linear addresses properly.

The correct form for the second statement might therefore look like:

```
io_mem = ioremap(0xfb000000, 0x200000);
t2 = *((unsigned char *)(io_mem + 0x100000));
```

The first statement creates a new 2 MB linear address interval, which maps physical addresses starting from 0xfb000000; the second one reads the memory location that has the 0xfc000000 address. To remove the mapping later, the device driver must use the `iounmap()` function.

On some architectures other than the PC, I/O shared memory cannot be accessed by simply dereferencing the linear address pointing to the physical memory location. Therefore, Linux defines the following architecture-dependent functions, which should be used when accessing I/O shared memory:

`readb()`, `readw()`, `readl()`

Reads 1, 2, or 4 bytes, respectively, from an I/O shared memory location

`writeb()`, `writew()`, `writel()`

Writes 1, 2, or 4 bytes, respectively, into an I/O shared memory location

`memcpy_fromio()`, `memcpy_toio()`

Copies a block of data from an I/O shared memory location to dynamic memory and vice versa

`memset_io()`

Fills an I/O shared memory area with a fixed value

The recommended way to access the 0xfc000000 I/O location is thus:

```
io_mem = ioremap(0xfb000000, 0x200000);
t2 = readb(io_mem + 0x100000);
```

Thanks to these functions, all dependencies on platform-specific ways of accessing the I/O shared memory can be hidden.

Direct Memory Access (DMA)

In the original PC architecture, the CPU is the only *bus master* of the system, that is, the only hardware device that drives the address/data bus in order to fetch and store values in the RAM's locations. With more modern bus architectures such as PCI, each peripheral can act as bus master, if provided with the proper circuitry. Thus, nowadays all PCs include auxiliary *DMA circuits*, which can transfer data between the RAM and an I/O device. Once activated by the CPU, the DMA is able to continue the data transfer on its own; when the data transfer is completed, the DMA issues an interrupt request. The conflicts that occur when CPUs and DMA circuits need to access the same memory location at the same time are resolved by a hardware circuit called a *memory arbiter* (see the section “Atomic Operations” in Chapter 5).

The DMA is mostly used by disk drivers and other devices that transfer a large number of bytes at once. Because setup time for the DMA is relatively high, it is more efficient to directly use the CPU for the data transfer when the number of bytes is small.

The first DMA circuits for the old ISA buses were complex, hard to program, and limited to the lower 16 MB of physical memory. More recent DMA circuits for the PCI and SCSI buses rely on dedicated hardware circuits in the buses and make life easier for device driver developers.

Synchronous and asynchronous DMA

A device driver can use the DMA in two different ways called *synchronous DMA* and *asynchronous DMA*. In the first case, the data transfers are triggered by processes; in the second case the data transfers are triggered by hardware devices.

An example of synchronous DMA is a sound card that is playing a sound track. A User Mode application writes the sound data (called *samples*) on a device file associated with the *digital signal processor (DSP)* of the sound card. The device driver of the sound card accumulates these samples in a kernel buffer. At the same time, the device driver instructs the sound card to copy the samples from the kernel buffer to the DSP with a well-defined timing. When the sound card finishes the data transfer, it raises an interrupt, and the device driver checks whether the kernel buffer still contains samples yet to be played; if so, the driver activates another DMA data transfer.

An example of asynchronous DMA is a network card that is receiving a frame (data packet) from a LAN. The peripheral stores the frame in its I/O shared memory, then raises an interrupt. The device driver of the network card acknowledges the interrupt, then instructs the peripheral to copy the frame from the I/O shared memory into a kernel buffer. When the data transfer completes, the network card raises another interrupt, and the device driver notifies the upper kernel layer about the new frame.

Helper functions for DMA transfers

When designing a driver for a device that makes use of DMA, the developer should write code that is both architecture-independent and, as far as DMA is concerned, bus-independent. This goal is now feasible thanks to the rich set of DMA helper functions provided by the kernel. These helper functions hide the differences in the DMA mechanisms of the various hardware architectures.

There are two subsets of DMA helper functions: an older subset provides architecture-independent functions for PCI devices; a more recent subset ensures both bus and architecture independence. We'll now examine some of these functions while pointing out some hardware peculiarities of DMAs.

Bus addresses

Every DMA transfer involves (at least) one memory buffer, which contains the data to be read or written by the hardware device. In general, before activating the transfer, the device driver must ensure that the DMA circuit can directly access the RAM locations.

Until now we have distinguished three kinds of memory addresses: logical and linear addresses, which are used internally by the CPU, and physical addresses, which are the memory addresses used by the CPU to physically drive the data bus. However, there is a fourth kind of memory address: the so-called *bus address*. It corresponds to the memory addresses used by all hardware devices except the CPU to drive the data bus.

Why should the kernel be concerned at all about bus addresses? Well, in a DMA operation, the data transfer takes place without CPU intervention; the data bus is driven directly by the I/O device and the DMA circuit. Therefore, when the kernel sets up a DMA operation, it must write the bus address of the memory buffer involved in the proper I/O ports of the DMA or I/O device.

In the 80×86 architecture, bus addresses coincide with physical addresses. However, other architectures such as Sun's SPARC and Hewlett-Packard's Alpha include a hardware circuit called the *I/O Memory Management Unit (IO-MMU)*, analog to the paging unit of the microprocessor, which maps physical addresses into bus addresses. All I/O drivers that make use of DMAs must set up properly the IO-MMU before starting the data transfer.

Different buses have different bus address sizes. For instance, bus addresses for ISA are 24-bits long, thus in the 80×86 architecture DMA transfers can be done only on the lower 16 MB of physical memory—that's why the memory for the buffer used by such DMA has to be allocated in the `ZONE_DMA` memory zone with the `GFP_DMA` flag. The original PCI standard defines bus addresses of 32 bits; however, some PCI hardware devices have been originally designed for the ISA bus, thus they still cannot access RAM locations above physical address `0x00ffffff`. The recent PCI-X standard uses 64-bit bus addresses and allows DMA circuits to address directly the high memory.

In Linux, the `dma_addr_t` type represents a generic bus address. In the 80×86 architecture `dma_addr_t` corresponds to a 32-bit integer, unless the kernel supports PAE (see the section “The Physical Address Extension (PAE) Paging Mechanism” in Chapter 2), in which case `dma_addr_t` corresponds to a 64-bit integer.

The `pci_set_dma_mask()` and `dma_set_mask()` helper functions check whether the bus accepts a given size for the bus addresses (mask) and, if so, notify the bus layer that the given peripheral will use that size for its bus addresses.

Cache coherency

The system architecture does not necessarily offer a coherency protocol between the hardware cache and the DMA circuits at the hardware level, so the DMA helper functions must take into consideration the hardware cache when implementing DMA mapping operations. To see why, suppose that the device driver fills the memory buffer with some data, then immediately instructs the hardware device to read that data with a DMA transfer. If the DMA accesses the physical RAM locations but the corresponding hardware cache lines have not yet been written to RAM, then the hardware device fetches the old values of the memory buffer.

Device driver developers may handle DMA buffers in two different ways by making use of two different classes of helper functions. Using Linux terminology, the developer chooses between two different *DMA mapping types*:

Coherent DMA mapping

When using this mapping, the kernel ensures that there will be no cache coherency problems between the memory and the hardware device; this means that every write operation performed by the CPU on a RAM location is immediately visible to the hardware device, and vice versa. This type of mapping is also called “synchronous” or “consistent.”

Streaming DMA mapping

When using this mapping, the device driver must take care of cache coherency problems by using the proper synchronization helper functions. This type of mapping is also called “asynchronous” or “non-coherent.”

In the 80×86 architecture there are never cache coherency problems when using the DMA, because the hardware devices themselves take care of “snooping” the accesses to the hardware caches. Therefore, a driver for a hardware device designed specifically for the 80×86 architecture may choose either one of the two DMA mapping types: they are essentially equivalent. On the other hand, in many architectures—such as MIPS, SPARC, and some models of PowerPC—hardware devices do not always snoop in the hardware caches, so cache coherency problems arise. In general, choosing the proper DMA mapping type for an architecture-independent driver is not trivial.

As a general rule, if the buffer is accessed in unpredictable ways by the CPU and the DMA processor, coherent DMA mapping is mandatory (for instance, buffers for SCSI adapters' command data structures). In other cases, streaming DMA mapping is preferable, because in some architectures handling the coherent DMA mapping is cumbersome and may lead to lower system performance.

Helper functions for coherent DMA mappings

Usually, the device driver allocates the memory buffer and establishes the coherent DMA mapping in the initialization phase; it releases the mapping and the buffer when it is unloaded. To allocate a memory buffer and to establish a coherent DMA mapping, the kernel provides the architecture-dependent `pci_alloc_consistent()` and `dma_alloc_coherent()` functions. They both return the linear address and the bus address of the new buffer. In the 80×86 architecture, they return the linear address and the physical address of the new buffer. To release the mapping and the buffer, the kernel provides the `pci_free_consistent()` and the `dma_free_coherent()` functions.

Helper functions for streaming DMA mappings

Memory buffers for streaming DMA mappings are usually mapped just before the transfer and unmapped thereafter. It is also possible to keep the same mapping among several DMA transfers, but in this case the device driver developer must be aware of the hardware cache lying between the memory and the peripheral.

To set up a streaming DMA transfer, the driver must first dynamically allocate the memory buffer by means of the zoned page frame allocator (see the section “The Zoned Page Frame Allocator” in Chapter 8) or the generic memory allocator (see the section “General Purpose Objects” in Chapter 8). Then, the drivers must establish the streaming DMA mapping by invoking either the `pci_map_single()` or the `dma_map_single()` function, which receives as its parameter the linear address of the buffer and returns its bus address. To release the mapping, the driver invokes the corresponding `pci_unmap_single()` or `dma_unmap_single()` functions.

To avoid cache coherency problems, right before starting a DMA transfer from the RAM to the device, the driver should invoke `pci_dma_sync_single_for_device()` or `dma_sync_single_for_device()`, which flush, if necessary, the cache lines corresponding to the DMA buffer. Similarly, a device driver should not access a memory buffer right after the end of a DMA transfer from the device to the RAM: instead, before reading the buffer, the driver should invoke `pci_dma_sync_single_for_cpu()` or `dma_sync_single_for_cpu()`, which invalidate, if necessary, the corresponding hardware cache lines. In the 80×86 architecture, these functions do almost nothing, because the coherency between hardware caches and DMAs is maintained by the hardware.

Even buffers in high memory (see the section “Kernel Mappings of High-Memory Page Frames” in Chapter 8) can be used for DMA transfers; the developer uses `pci_map_page()`—or `dma_map_page()`—passing to it the descriptor address of the page

including the buffer and the offset of the buffer inside the page. Correspondingly, to release the mapping of the high memory buffer, the developer uses `pci_unmap_page()` or `dma_unmap_page()`.

Levels of Kernel Support

The Linux kernel does not fully support all possible existing I/O devices. Generally speaking, in fact, there are three possible kinds of support for a hardware device:

No support at all

The application program interacts directly with the device's I/O ports by issuing suitable in and out assembly language instructions.

Minimal support

The kernel does not recognize the hardware device, but does recognize its I/O interface. User programs are able to treat the interface as a sequential device capable of reading and/or writing sequences of characters.

Extended support

The kernel recognizes the hardware device and handles the I/O interface itself. In fact, there might not even be a device file for the device.

The most common example of the first approach, which does not rely on any kernel device driver, is how the X Window System traditionally handles the graphic display. This is quite efficient, although it constrains the X server from using the hardware interrupts issued by the I/O device. This approach also requires some additional effort to allow the X server to access the required I/O ports. As mentioned in the section “Task State Segment” in Chapter 3, the `iopl()` and `ioperm()` system calls grant a process the privilege to access I/O ports. They can be invoked only by programs having root privileges. But such programs can be made available to users by setting the `setuid` flag of the executable file (see the section “Process Credentials and Capabilities” in Chapter 20).

Recent Linux versions support several widely used graphic cards. The `/dev/fb` device file provides an abstraction for the frame buffer of the graphic card and allows application software to access it without needing to know anything about the I/O ports of the graphics interface. Furthermore, the kernel supports the Direct Rendering Infrastructure (DRI) that allows application software to exploit the hardware of accelerated 3D graphics cards. In any case, the traditional do-it-yourself X Window System server is still widely adopted.

The minimal support approach is used to handle external hardware devices connected to a general-purpose I/O interface. The kernel takes care of the I/O interface by offering a device file (and thus a device driver); the application program handles the external hardware device by reading and writing the device file.

Minimal support is preferable to extended support because it keeps the kernel size small. However, among the general-purpose I/O interfaces commonly found on a

PC, only the serial port and the parallel port can be handled with this approach. Thus, a serial mouse is directly controlled by an application program, such as the X server, and a serial modem always requires a communication program, such as Minicom, Seyon, or a Point-to-Point Protocol (PPP) daemon.

Minimal support has a limited range of applications, because it cannot be used when the external device must interact heavily with internal kernel data structures. For example, consider a removable hard disk that is connected to a general-purpose I/O interface. An application program cannot interact with all kernel data structures and functions needed to recognize the disk and to mount its filesystem, so extended support is mandatory in this case.

In general, every hardware device directly connected to the I/O bus, such as the internal hard disk, is handled according to the extended support approach: the kernel must provide a device driver for each such device. External devices attached to the Universal Serial Bus (USB), the PCMCIA port found in many laptops, or the SCSI interface—in short, every general-purpose I/O interface except the serial and the parallel ports—also require extended support.

It is worth noting that the standard file-related system calls such as `open()`, `read()`, and `write()` do not always give the application full control of the underlying hardware device. In fact, the lowest-common-denominator approach of the VFS does not include room for special commands that some devices need or let an application check whether the device is in a specific internal state.

The `ioctl()` system call was introduced to satisfy such needs. Besides the file descriptor of the device file and a second 32-bit parameter specifying the request, the system call can accept an arbitrary number of additional parameters. For example, specific `ioctl()` requests exist to get the CD-ROM sound volume or to eject the CD-ROM media. Application programs may provide the user interface of a CD player using these kinds of `ioctl()` requests.

Character Device Drivers

Handling a character device is relatively easy, because usually sophisticated buffering strategies are not needed and disk caches are not involved. Of course, character devices differ in their requirements: some of them must implement a sophisticated communication protocol to drive the hardware device, while others just have to read a few values from a couple of I/O ports of the hardware devices. For instance, the device driver of a multiport serial card device (a hardware device offering many serial ports) is much more complicated than the device driver of a bus mouse.

Block device drivers, on the other hand, are inherently more complex than character device drivers. In fact, applications are entitled to ask repeatedly to read or write the same block of data. Furthermore, accesses to these devices are usually very slow. These peculiarities have a profound impact on the structure of the disk drivers. As

we'll see in the next chapters, however, the kernel provides sophisticated components—such as the page cache and the block I/O subsystem—to handle them. In the rest of this chapter we focus our attention on the character device drivers.

A character device driver is described by a `cdev` structure, whose fields are listed in Table 13-8.

Table 13-8. The fields of the `cdev` structure

Type	Field	Description
<code>struct kobject</code>	<code>kobj</code>	Embedded <code>kobject</code>
<code>struct module *</code>	<code>owner</code>	Pointer to the module implementing the driver, if any
<code>struct file_operations *</code>	<code>ops</code>	Pointer to the file operations table of the device driver
<code>struct list_head</code>	<code>list</code>	Head of the list of inodes relative to device files for this character device
<code>dev_t</code>	<code>dev</code>	Initial major and minor numbers assigned to the device driver
<code>unsigned int</code>	<code>count</code>	Size of the range of device numbers assigned to the device driver

The `list` field is the head of a doubly linked circular list collecting inodes of character device files that refer to the same character device driver. There could be many device files having the same device number, and all of them refer to the same character device. Moreover, a device driver can be associated with a range of device numbers, not just a single one; all device files whose numbers fall in the range are handled by the same character device driver. The size of the range is stored in the `count` field.

The `cdev_alloc()` function allocates dynamically a `cdev` descriptor and initializes the embedded `kobject` so that the descriptor is automatically freed when the reference counter becomes zero.

The `cdev_add()` function registers a `cdev` descriptor in the device driver model. The function initializes the `dev` and `count` fields of the `cdev` descriptor, then invokes the `kobj_map()` function. This function, in turn, sets up the device driver model's data structures that glue the interval of device numbers to the device driver descriptor.

The device driver model defines a *kobject mapping domain* for the character devices, which is represented by a descriptor of type `kobj_map` and is referenced by the `cdev_map` global variable. The `kobj_map` descriptor includes a hash table of 255 entries indexed by the major number of the intervals. The hash table stores objects of type `probe`, one for each registered range of major and minor numbers, whose fields are listed in Table 13-9.

Table 13-9. The fields of the `probe` object

Type	Field	Description
<code>struct probe *</code>	<code>next</code>	Next element in hash collision list
<code>dev_t</code>	<code>dev</code>	Initial device number (major and minor) of the interval

Table 13-9. The fields of the probe object (continued)

Type	Field	Description
unsigned long	range	Size of the interval
struct module *	owner	Pointer to the module that implements the device driver, if any
struct kobject *(*)(dev_t, int *, void *)	get	Method for probing the owner of the interval
int (*)(dev_t, void *)	lock	Method for increasing the reference counter of the owner of the interval
void *	data	Private data for the owner of the interval

When the `kobj_map()` function is invoked, the specified interval of device numbers is added to the hash table. The data field of the corresponding probe object points to the `cdev` descriptor of the device driver. The value of this field is passed to the `get` and `lock` methods when they are executed. In this case, the `get` method is implemented by a short function that returns the address of the `kobject` embedded in the `cdev` descriptor; the `lock` method, instead, essentially increases the reference counter in the embedded `kobject`.

The `kobj_lookup()` function receives as input parameters a `kobject` mapping domain and a device number; it searches the hash table and returns the address of the `kobject` of the owner of the interval including the number, if it was found. When applied to the mapping domain of the character devices, the function returns the address of the `kobject` embedded in the `cdev` descriptor of the device driver that owns the interval of device numbers.

Assigning Device Numbers

To keep track of which character device numbers are currently assigned, the kernel uses a hash table `chrdevs`, which contains intervals of device numbers. Two intervals may share the same major number, but they cannot overlap, thus their minor numbers should be all different. The table includes 255 entries, and the hash function masks out the four higher-order bits of the major number—therefore, major numbers less than 255 are hashed in different entries. Each entry points to the first element of a collision list ordered by increasing major and minor numbers.

Each list element is a `char_device_struct` structure, whose fields are shown in Table 13-10.

Table 13-10. The fields of the `char_device_struct` descriptor

Type	Field	Description
unsigned char_device_struct *	next	The pointer to next element in hash collision list
unsigned int	major	The major number of the interval

Table 13-10. The fields of the `char_device_struct` descriptor (continued)

Type	Field	Description
unsigned int	baseminor	The initial minor number of the interval
int	minorct	The interval size
const char *	name	The name of the device driver that handles the interval
struct file_operations *	fops	Not used
struct cdev *	cdev	Pointer to the character device driver descriptor

There are essentially two methods for assigning a range of device numbers to a character device driver. The first method, which should be used for all new device drivers, relies on the `register_chrdev_region()` and `alloc_chrdev_region()` functions, and assigns an arbitrary range of device numbers. For instance, to get an interval of numbers starting from the `dev_t` value `dev` and of size `size`:

```
register_chrdev_region(dev, size, "foo");
```

These functions do not execute `cdev_add()`, so the device driver must execute `cdev_add()` after the requested interval has been successfully assigned.

The second method makes use of the `register_chrdev()` function and assigns a fixed interval of device numbers including a single major number and minor numbers from 0 to 255. In this case, the device driver must not invoke `cdev_add()`.

The `register_chrdev_region()` and `alloc_chrdev_region()` functions

The `register_chrdev_region()` function receives three parameters: the initial device number (major and minor numbers), the size of the requested range of device numbers (as the number of minor numbers), and the name of the device driver that is requesting the device numbers. The function checks whether the requested range spans several major numbers and, if so, determines the major numbers and the corresponding intervals that cover the whole range; then, the function invokes `__register_chrdev_region()` (described below) on each of these intervals.

The `alloc_chrdev_region()` function is similar, but it is used to allocate dynamically a major number; thus, it receives as its parameters the initial minor number of the interval, the size of the interval, and the name of the device driver. This function also ends up invoking `__register_chrdev_region()`.

The `__register_chrdev_region()` function executes the following steps:

1. Allocates a new `char_device_struct` structure, and fills it with zeros.
2. If the major number of the interval is zero, then the device driver has requested the dynamic allocation of the major number. Starting from the last hash table entry and proceeding backward, the function looks for an empty collision list

(NULL pointer), which corresponds to a yet unused major number. If no empty entry is found, the function returns an error code.*

3. Initializes the fields of the `char_device_struct` structure with the initial device number of the interval, the interval size, and the name of the device driver.
4. Executes the hash function to compute the hash table index corresponding to the major number.
5. Walks the collision list, looking for the correct position of the new `char_device_struct` structure. Meanwhile, if an interval overlapping with the requested one is found, it returns an error code.
6. Inserts the new `char_device_struct` descriptor in the collision list.
7. Returns the address of the new `char_device_struct` descriptor.

The `register_chrdev()` function

The `register_chrdev()` function is used by drivers that require an old-style interval of device numbers: a single major number and minor numbers ranging from 0 to 255. The function receives as its parameters the requested major number `major` (zero for dynamic allocation), the name of the device driver `name`, and a pointer `fops` to a table of file operations specific to the character device files in the interval. It executes the following operations:

1. Invokes the `__register_chrdev_region()` function to allocate the requested interval. If the function returns an error code (the interval cannot be assigned), it terminates.
2. Allocates a new `cdev` structure for the device driver.
3. Initializes the `cdev` structure:
 - a. Sets the type of the embedded `kobject` to the `ktype_cdev_dynamic` type descriptor (see the earlier section “Kobjects”).
 - b. Sets the `owner` field with the contents of `fops->owner`.
 - c. Sets the `ops` field with the address `fops` of the table of file operations.
 - d. Copies the characters of the device driver `name` into the `name` field of the embedded `kobject`.
4. Invokes the `cdev_add()` function (explained previously).

* Notice that the kernel can dynamically allocate only major numbers less than 255, and that in some cases allocation can fail even if there is a unused major number less than 255. We might expect that these constraints will be removed in the future.

5. Sets the `cdev` field of the `char_device_struct` descriptor `__register_chrdev_region()` returned in step 1 with the address of the `cdev` descriptor of the device driver.
6. Returns the major number of the assigned interval.

Accessing a Character Device Driver

We mentioned in the earlier section “VFS Handling of Device Files” that the `dentry_open()` function triggered by the `open()` system call service routine customizes the `f_op` field in the file object of the character device file so that it points to the `def_chr_fops` table. This table is almost empty; it only defines the `chrdev_open()` function as the `open` method of the device file. This method is immediately invoked by `dentry_open()`.

The `chrdev_open()` function receives as its parameters the addresses `inode` and `filp` of the `inode` and file objects relative to the device file being opened. It executes essentially the following operations:

1. Checks the `inode->i_cdev` pointer to the device driver’s `cdev` descriptor. If this field is not `NULL`, then the `inode` has already been accessed: increases the reference counter of the `cdev` descriptor and jumps to step 6.
2. Invokes the `kobj_lookup()` function to search the interval including the number. If such interval does not exist, it returns an error code; otherwise, it computes the address of the `cdev` descriptor associated with the interval.
3. Sets the `inode->i_cdev` field of the `inode` object to the address of the `cdev` descriptor.
4. Sets the `inode->i_cindex` field to the relative index of the device number inside the interval of the device driver (index zero for the first minor number in the interval, one for the second, and so on).
5. Adds the `inode` object into the list pointed to by the `list` field of the `cdev` descriptor.
6. Initializes the `filp->f_ops` file operations pointer with the contents of the `ops` field of the `cdev` descriptor.
7. If the `filp->f_ops->open` method is defined, the function executes it. If the device driver handles more than one device number, typically this function sets the file operations of the file object once again, so as to install the file operations suitable for the accessed device file.
8. Terminates by returning zero (success).

Buffering Strategies for Character Devices

Traditionally, Unix-like operating systems divide hardware devices into block and character devices. However, this classification does not tell the whole story. Some devices are capable of transferring sizeable amounts of data in a single I/O operation, while others transfer only a few characters.

For instance, a PS/2 mouse driver gets a few bytes in each read operation corresponding to the status of the mouse button and to the position of the mouse pointer on the screen. This kind of device is the easiest to handle. Input data is first read one character at a time from the device input register and stored in a proper kernel data structure; the data is then copied at leisure into the process address space. Similarly, output data is first copied from the process address space to a proper kernel data structure and then written one at a time into the I/O device output register. Clearly, I/O drivers for such devices do not use the DMA, because the CPU time spent to set up a DMA I/O operation is comparable to the time spent to move the data to or from the I/O ports.

On the other hand, the kernel must also be ready to deal with devices that yield a large number of bytes in each I/O operation, either sequential devices such as sound cards or network cards, or random access devices such as disks of all kinds (floppy, CD-ROM, SCSI disk, etc.).

Suppose, for instance, that you have set up the sound card of your computer so that you are able to record sounds coming from a microphone. The sound card samples the electrical signal coming from the microphone at a fixed rate, say 44.14 kHz, and produces a stream of 16-bit numbers divided into blocks of input data. The sound card driver must be able to cope with this avalanche of data in all possible situations, even when the CPU is temporarily busy running some other process.

This can be done by combining two different techniques:

- Use of DMA to transfer blocks of data.
- Use of a circular buffer of two or more elements, each element having the size of a block of data. When an interrupt occurs signaling that a new block of data has been read, the interrupt handler advances a pointer to the elements of the circular buffer so that further data will be stored in an empty element. Conversely, whenever the driver succeeds in copying a block of data into user address space, it releases an element of the circular buffer so that it is available for saving new data from the hardware device.

The role of the circular buffer is to smooth out the peaks of CPU load; even if the User Mode application receiving the data is slowed down because of other higher-priority tasks, the DMA is able to continue filling elements of the circular buffer because the interrupt handler executes on behalf of the currently running process.

A similar situation occurs when receiving packets from a network card, except that in this case, the flow of incoming data is asynchronous. Packets are received independently from each other and the time interval that occurs between two consecutive packet arrivals is unpredictable.

All considered, buffer handling for sequential devices is easy because the same buffer is *never reused*: an audio application cannot ask the microphone to retransmit the same block of data.

We'll see in Chapter 15 that buffering for random access devices (all kinds of disks) is much more complicated.