13.4. 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.

13.4.1. 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.

13.4.2. 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.[*]

  [*] 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.

- 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.

13.4.3. 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 monitoringtechnique 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*.

13.4.3.1. 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;
    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.

13.4.3.2. 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.

13.4.4. 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 `0x000b0fe4` 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.

13.4.5. 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.

13.4.5.1. 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.

13.4.5.2. 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.

13.4.5.3. 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 x 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 x 86 architecture DMA transfers can be done only on the lower 16 MB of physical memorythat'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 x 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.

13.4.5.4. 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 x 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 x 86 architecture may choose either one of the two DMA mapping types: they are essentially equivalent. On the other hand, in many architecturessuch as MIPS, SPARC, and some models of PowerPChardware 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.

13.4.5.5. 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 x 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.

13.4.5.6. 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 x 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( )`.

13.4.6. 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 interfacein short, every general-purpose I/O interface except the serial and the parallel portsalso 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.