

5. Devices and Device Management

ST machines are simple computers that are usually used to control a set of devices within the context of some larger system. For example, an ST computer might control the guidance system of an unmanned rocket – the rocket is the larger system, and the ST computer’s job might be just to control the speed and to guide the rocket. Such a ST computer might have one device to determine the current heading of the rocket and another to detect the speed of the rocket. Speed might be controlled by metering the amount of fuel used by the rocket, varying from no fuel (the engine is off and the rocket is coasting) to full fuel (the engine is producing its maximum amount of power). Obviously, the fuel could also be metered at any fractional rate between 0% and 100%. Guidance might be accomplished by directing the engine thrust to push the rocket left, right, up, or down (or combinations of these directions such as 40° left and 25° up). The ST computer could then be directed to fly the rocket in a chosen direction at a selected speed. The application program in the ST computer would then read the current heading to compute the new direction, and to control the amount and direction of engine thrust to orient the rocket in the specified direction. Once the rocket is pointed in the right direction, the application program would reset the thrust direction and then choose a fuel rate to attain and sustain the rocket motion. As the rocket proceeds on its trip, the ST computer would constantly monitor the heading and speed, adjusting the thrust as necessary.

The rocket guidance application program primarily reads data from sensor devices, decides if any action is required (and if so, chooses the action), and then transmits commands to one or more actuator devices. Its activity is focused on device I/O.

A **device** is a unit of hardware that can be integrated into a computer by physically connecting it to the computer’s bus, and logically connecting it to the software via the OS device manager. The device can perform operations independently of the operation of the computer’s central processing unit. The **device manager** provides software that enables the application program(s) to use devices with a minimum of requisite knowledge and effort.

In order for the software to control many different kinds of devices, every device is designed to conform to a simple, but standard, architecture, similar to the general form shown in Figure 5-1. This organization is relatively general, and it also describes the form of devices used in larger MT and MP SCCs. The purpose of the **controller** is to adapt the device to the particular computer: besides adapting the device to physically attach to the computer, the controller translates the processor’s software commands into electronic signals that control the behavior of the device, and translates information that the device produces – data or information about the operation of the device – into a format that the software can use. For example, if the device were a thermometer, the controller might be required to convert the thermometer device reading into a floating point number that the software is expecting. Thus the controller is a hardware component that provides a physical and logical interface between the device and the software that executes on the CPU.

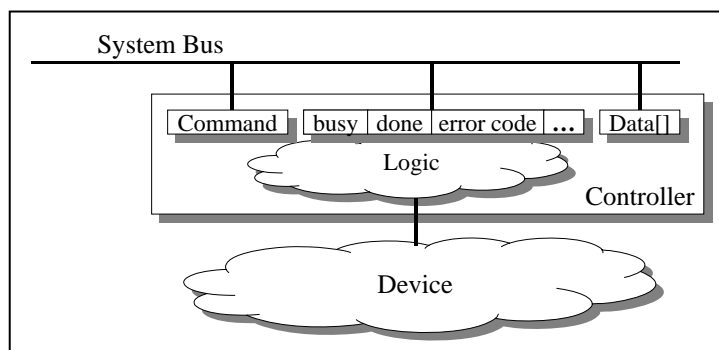


Figure 5-1: Device Organization

In this book we describe a hypothetical controller interface that represents the spectrum of interfaces that are used in real device controllers. By studying this model you will learn the concepts of device I/O that you can apply to any given system. Each controller contains a set of registers that can be read and written by the physical device, and by the software executing on the CPU:

- **Command register:** The software on the CPU loads device commands into the command register to direct the device to read, write, seek, and so on.
- **Status register (busy, done, error code, ...):** The device controller reports the device's logical status via this register (see below for the details of the fields in the register).
- **Data[...] registers:** One or more registers that hold data that is staged to be written to the device, or that has been read from the device in its most recent read operation. The CPU writes output data to these registers for output commands, and reads input data from these registers on input commands.

From the application software perspective, programs expect that the OS device manager will contain code to handle the details for controlling the device so that the application program can simply call OS functions to read and write information from/to the device. The OS device manager will implement all of the details required to operate the device. In particular, the device manager will handle the following three tasks:

- **Device command detail:** The device controller executes a set of commands to control the device. A system programmer reads the documentation for the device, then creates one or more function programs to write desired command sequence into the command register to accomplish device read, write, and other high level commands. For example, there may be commands to position a read/write pointer (for example to control a storage device), to place the device in "sleep" mode so that it does not use full power, to invert black and white on a display screen, and so on.
- **Synchronization:** Since the device is a physically distinct hardware subassembly from the CPU, it executes its commands independently of ongoing CPU activity. For example, the software may issue a command to read a block of data from a disk device, causing the device to start executing the read operation at the same time the CPU is executing its own instructions. In contemporary systems, the simultaneous operation between the CPU and the device is **asynchronous**, meaning that the device command will not be completed until *an undetermined* number of CPU cycles have transpired. The application software needs to be able to dynamically determine when the device's asynchronous activity has completed, so in the case of devices without interrupts, the device manager periodically reads the busy and done fields in the status register under the high level direction from the application software.
- **Error handling:** The controller can handle minor errors that the device encounters (such as retrying a read operation if a parity error occurs), but it may ultimately need to report a recurring or serious error to the OS software. This is accomplished by setting the error code field in the status register to identify the occurrence and type of the error, then setting the busy field in the status register to 0 and the done field to 1. The device manager attempts to resolve the errors; if the errors are too severe, then the device manager will report I/O failure to the application software.

5.1 Types of Devices

As you know, every computer is made up of a CPU, executable memory, and a collection of devices. As we know from our previous experience with computers, they can have a myriad of different kinds of devices, ranging from a mouse to a numerical control machine. Let's begin to inspect device manager design a little more closely by observing that some devices can only accept output data that is written from the computer – data sink devices in Figure 5-2. Other devices are input-only devices, that is, they are data sources rather than data sinks. Examples of data sink (output-only) devices include video displays, printers, audio speakers, and actuators (such as a machine controller); you can probably think of dozens more output-only devices (such as the rocket engine controllers described at the beginning of this chapter). Input-only (data source) devices include the keyboard, mouse, an audio microphone, a sensor device, a CD-ROM reader, and a digital camera.

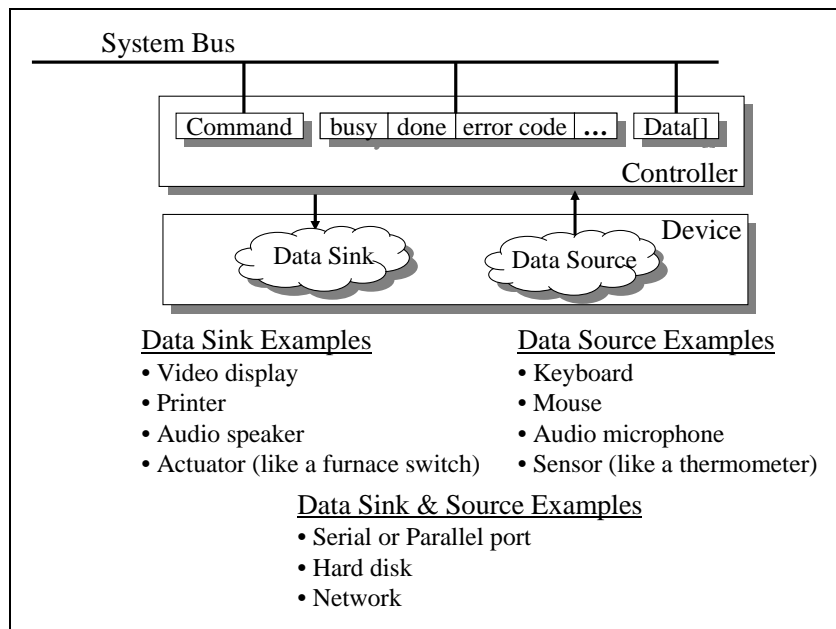


Figure 5-2: Data Flow Between the CPU and a Device

Some devices are both input and output devices, for example a hard disk device, a PCMCIA memory card on a laptop machine (but not a PCMCIA digital camera card), a Compact Flash (CF) memory card, or a network. Whether the device is an input-only, output-only, or input and output device, it will export an interface to the software as shown in Figure 5-1 – there will be a physical interface that is used to connect the device to the computer's internal system bus, and the software that executes on the CPU will use a set of registers to control the device, sense its status, and to transmit/receive data.

The controller will be designed so that it is able to interact with the OS **device driver** software that controls the device, so that it can provide the device-specific interface to manage the device – whether it is a keyboard, video display, or rocket engine control. Obviously, the controller for a simple device like a thermometer need not be very complex, compared to, say, a controller for a hard disk. The thermometer device controller is only required to read a scalar value from the device, but the disk controller must locate blocks of data on the disk, power the disk up and down as needed, and so on.

Serial port devices are the most common devices on desktops, notebooks, and many SCCs. They are so widely used that they are often integrated into the basic parts of the computer, for example as part of the circuit board on which the microprocessor chip is mounted (or even as part of the microprocessor chip). A serial port is a somewhat specialized type of device in that it is used to read or write a single byte of information at a time, transferring the byte to/from a CPU register from/to the external world via an external connector on the computer (see Figure 5-3). Because they typically read/write information from/to a physical connector, they are informally called “ports” instead of devices.

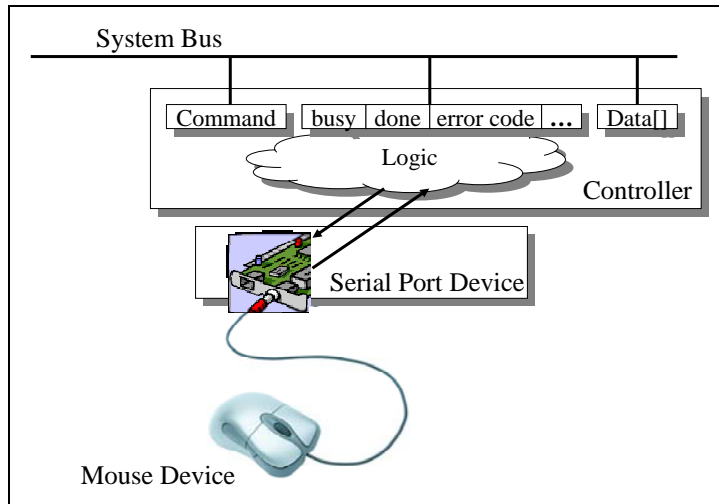


Figure 5-3: Adding a Device to a Port

In this specialized case, the I/O device is actually composed of two parts: the serial port *and* the real device that is connected to the port connector with a cable. For example, in Figure 5-3 the cable connects the serial port device to an external device such as a serial port mouse (a mouse can also be connected to a contemporary computer using a “PS/2” port or a USB port). Of course this is an essential part of the value of a serial port – it provides enough functionality to control many different physical devices such as a mouse. Here is how it works: the port device is designed to read and write bytes of data in a particular format from/to a particular kind of physical connector (a DB-9 connector, if that makes any sense to you). Part of the operation of the serial port device is that it will send or receive a particular pattern of electronic signals via the pins on the connector; any secondary device (like a mouse) can be designed to receive/send the corresponding electronic signals. Serial ports are so widely used that there is a standard (called the RS-232 protocol) that defines the pattern of electronic signals to convey information between the external device and the serial port connector. Any external device that uses the RS-232 protocol can be connected to a serial port, and the device manager serial port driver software will be able to read and write bytes from/to the external device via the serial port.

Serial ports actually only use 4 wires in the 9-pin connector: one of the wires has power, one is a ground wire, one transmits control signals from the serial port device to the external device, and the fourth wire transfers the data between the external device and the serial port device. That is, 8-bit bytes are transferred back and forth, as a serial stream of bits, between the serial port device and the external device one bit at a time. There is another common port device called a parallel ports that uses a DB-25 (25-pin) connector. A parallel port uses the same basic idea as a serial port, although it can transmit/receive all 8 bits in a byte simultaneously, hence the name “parallel” port.

Serial and parallel ports are examples of **communications devices**. Most communication devices can handle input and output, although any particular external device may be input-only (like a mouse or keyboard) or output only (like a printer). Communication devices are used to transmit information between the SCC and an external mechanism, which could be a remote computer, using a physical transmission medium such as public broadcast (wireless) network, coaxial cable, or telephone wires. There are several challenges in building effective communication devices: an important one is that the two devices that communicate using a communication device must agree on how electronic signals will be transmitted, the frequency of signaling, the meaning of signals in every possible context, and so on. This potentially elaborate agreement is called a **communication protocol**. In the case of serial ports, the RS-232 standard defines the communication protocol between the external device and the serial port device. As you learned in Sections [2.1](#) and [3.2](#), the communication protocol can be very elaborate with many choices.

A **storage device** is an input and an output device that is capable of saving the information written to it even when the computer is turned off. When the machine is powered back up again, the SCC can read the information from the storage device so that the information can be used later. In a conventional computer, a floppy disk, digital audio tape (DAT), USB flash drive, and hard disk are all examples of storage devices. In an SCC, a PCMCIA memory or CF memory can also be used as storage devices. Storage device technology spans an enormous range of products. CF memory, USB flash drives, and PCMCIA memory may be as small as 256 MB, while RAID (**R**edundant **A**rray of **I**nexpensive **D**isks) systems exceed 10 TB (terabyte or trillions of bytes) capacity.

SCCs may also incorporate one or more **special devices**, meaning devices such as a stylus input device (as used in a tablet computer or PDA), devices that determine the SCC's current position and orientation (such as a GPS device), sensors to detect external readings, actuators to control external machines, and so on.

5.1.1 Communication Devices

Serial and parallel port technology was developed in the 1970s and came into wide use in the 1980s. Modern communication devices are generalizations of port devices, and still tend to share the basic idea behind a port, namely that the device is intended to read and write information through an external port. In modern technologies, the external device might be a hub that multiplexes multiple ports into and out of the single device port, or even a connection to a data network.

By 1990, it had become apparent that computers needed to have alternatives to serial and parallel ports that could be used for higher speed data interchange than would be possible using the older port-based technology. In one usage domain (consumer electronics), devices such as video cameras, digital cameras, external CD-ROM drives, and then personal digital assistants began to appear as devices that could be plugged into a computer to download or synchronize information with the computer. A serial port could be used to accomplish this task, but it was relatively slow. The USB and Firewire alternatives began to appear as options in personal computers in the early 1990s.

The **Universal Serial Bus (USB)** was developed to provide an external port that would act like a bi-directional, 1.5 to 12 Mbps internal connection to the computer's bus.¹ (The USB 2.0 specification calls for data transmission rates as high as 480 Mbps.) The USB computer port can be connected to directly to a device (like a serial port), or to a hub that can connect multiple devices to the computer via the USB port. The device driver and USB hardware support the normal I/O operations, although they tend to transfer blocks of bytes rather than a byte at a time. In addition they provide the ability to handle dynamic connection and disconnection of devices via the USB socket, and to ensure that the appropriate detailed device driver is installed to manage the device over the USB port. USB also provides power to the device via the USB port; this means that the system has additional responsibility for routing power over the USB port to the device or hub.

¹ The USB interface is controlled by the USB Implementers Forum, Inc., see <http://www.usb.org/home>.

The IEEE 1394 **Firewire** specification is a higher speed alternative to USB. Firewire was originally developed by Apple Computer in the 1980s, and then it was adopted as an IEEE standard in 1995 (the same time as USB).² The IEEE 1394a standard defines a serial bus that operates at 100, 200, or 400 Mbps, with plans to transmit data at 3.2 Gbps for devices in close proximity. Like the USB, Firewire has one or more external connectors to allow users to connect a digital camera or other devices directly onto the serial bus. The high speeds possible with Firewire make it better suited than USB for streaming media applications – Firewire can support an external DVD device.

The next level of sophistication in communication devices are network devices. Recall from Chapters 2 and 3 that a computer network is made up of a collection of computers, referred to as **nodes** in the network, and a means of interconnecting the computers, called the **subcommunications subnetwork**, so that they can exchange information with one another. Each computer has a network interface controller (NIC) device that is capable of interacting with the communications subnetwork (see Figure 5-4). There is a spectrum of designs for the network: in the context of a device discussion, this is most obvious by observing that in some networks the subcommunication network is a complex switching machine such as a central office switch for a telephone company. In these cases, the network-dependent logic in the NIC can be relatively simple, since it need only perform simple operations in order to transmit/receive information to/from another computer on the network. On the other end of the spectrum (such as in the Ethernet), the subcommunication network can be as simple as a shared cable, in which case each the network-dependent logic in each NIC must incorporate all of the functionality required to format information and route it along the shared cable so that the desired recipient receives it. Hence, in some cases the NIC and its device driver could be almost as simple as a serial port, but in other network designs they implement significant portions of the data link and network layers and can be relatively complex.

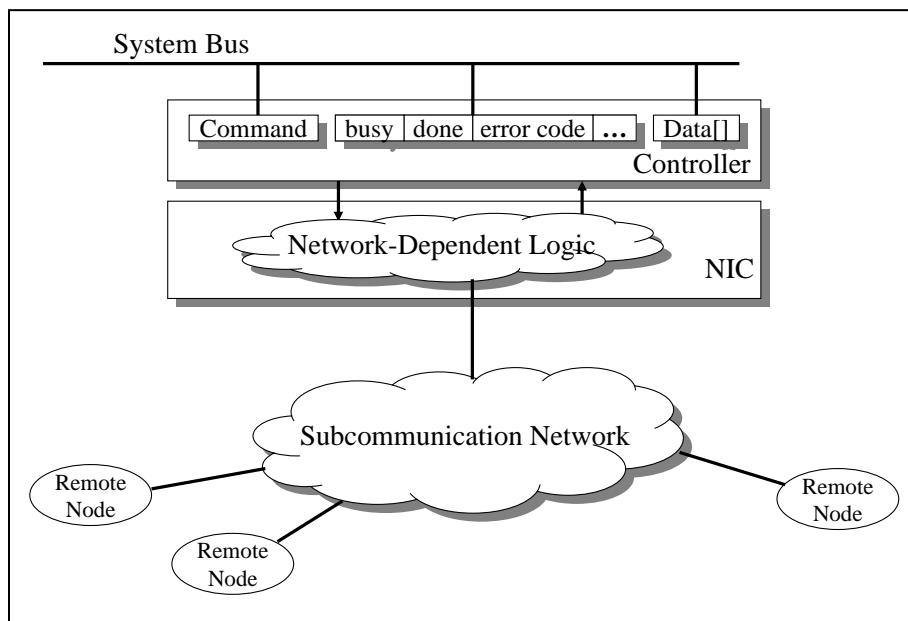


Figure 5-4: The NIC Device and the Communication Subnetwork

² “IEEE Standard for a High Performance Serial Bus – Description,” IEEE Std 1394-1995, see http://standards.ieee.org/reading/ieee/std_public/description/busarch/1394-1995_desc.html.

5.1.2 Storage Devices

Storage devices refer to any kind of device that can have information written to it, and the device will preserve that information even if the computer is powered down and then powered back up again. Disk drivers – hard and floppy – are the most widely known storage devices. CD-ROMs, DVDs, digital audio tape (DAT), flash memory, and so on are also storage devices. Essentially all storage devices are **block-oriented** devices, meaning that the software can only read or write information from/to the device as a collection of bytes. For example, classic Microsoft Windows disk drives use 512-byte blocks. Other disks are designed to read and write much larger block, for example 8,192 bytes or more.

Randomly accessed storage devices (such as hard disk drives and flash memories) are organized as a collection of linearly addressed blocks of information. Each **block** is an array of n bytes, and the blocks are identified by their unique index (or **block address**) – see Figure 5-5. Software can execute a device write operation that will copy an n -byte block of data from the computer’s executable memory into the device block that has address i (Blk_i in the figure) on the storage device. The software can retrieve a copy of the n bytes stored in the device block with address j by reading Blk_j . Devices in this class are said to be “randomly accessed” since a program can read Blk_i , then Blk_j , where i and j are not necessarily consecutive block indices (block addresses). By contrast, older storage devices (such as magnetic tape) can only be accessed by reading/writing the blocks sequentially – Blk_j can only be read after reading Blk_{j-1} .

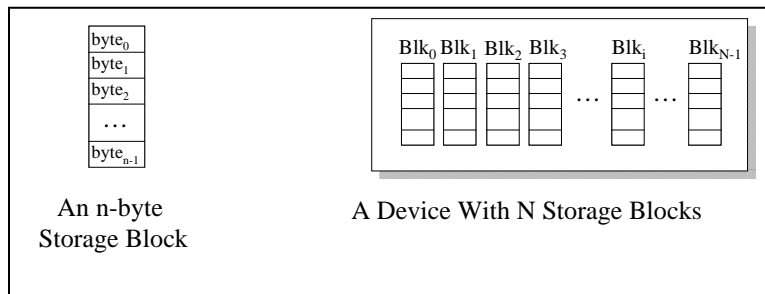


Figure 5-5: Block-oriented Storage Device

ST SCCs often incorporate some form of randomly accessed storage device, but it usually has relatively small physical size, low power consumption, and moderate storage capacity (compared to hard disks on desktop computers). For example the device may be packaged in a CF card, but either be a tiny hard disk or a block-oriented flash memory. In an ST SCC, these storage devices can be downloaded from an associated server, or they can temporarily save information until the SCC uploads it to an associated server.

5.1.3 Sensors and Actuators

Sensors and actuators are the predominant devices in a small embedded system that uses an ST OS. Each sensor is frequently associated with an actuator, for example a sensor might measure velocity, and the associated actuator might control a throttle or brake. Sensors and actuators are generally specialized devices enable an embedded system to interact with its host system. From the software perspective, a sensor is a device from which the software can read fixed-length record – often just a scalar number. Suppose an embedded system incorporated a digital compass sensor device. Whenever the software reads the compass sensor, the device returns an integer between 0 and 359, representing the heading of the compass, in degrees, at the moment it is read. An alternative compass might return a pair of integers to represent the current heading in degrees and minutes, or as a floating point number with the minutes expressed as the fractional part of the result (for example a heading of 173.5 might mean 173° 30 minutes). Yet another sensor device might measure 50 different pressure points on the wing of an aircraft. In this case, the sensor read operation returns 50 different values on a single read operation.

From the software perspective, an actuator device is one to which the program can write one or more values. The value might be a single integer scalar, or it might be an array of numbers. For example, suppose that an actuator opens or closes a valve on a zone in a sprinkler system. The interface to the actuator might be designed so that if a 0 is written to the actuator, the valve is closed, and if a number between 1 and 10 is written to the actuator, then valve is opened an amount proportional to the number. For example, writing a 10 to the valve might open it to allow maximum flow, but writing a 3 to the actuator would open the valve to permit 30% of the maximum flow.

Given this insight into sensors and actuators, one can see that these devices are similar to ports: the software simply reads or writes a value from/to the device. The application software on the ST SCC interprets input values because it is written with the knowledge of the operation of the sensor. Similarly, the application program chooses a value to write to the actuator based on the algorithm encoded in the application. From the device manager perspective, sensors and actuators are somewhat like ports to which data can be written or read.

5.2 Software Perspective of Devices

The device manager provides a common abstraction of the operation of the hardware devices (controllers) that may be attached to the computer. The nature of the abstraction is chosen by the OS designer – recall the discussion about abstractions in [Chapter 1](#) where we considered different ways that a person might think about writing information onto a bitmapped display. The OS designers decide the nature of the abstraction that they will implement, and with which applications programs can use to read or write information from/to various devices. In the next section we will discuss the device manager design; in this subsection we will describe device abstractions that are used in contemporary operating systems such as Linux.

Consider the nature of the OS software that interacts directly with the device controller to read a single byte from a character-oriented device (such as a serial port). In this type of device the controller Data register holds only a single byte. Figure 5-6 is a pseudo code representation of the device driver's behavior. In the first line of code, the software loads the `READ_CMD` command into device `dev`'s Command register to start the device on an input operation. The `while`-loop continuously reads the controller Status register until the Status register indicates that the device has completed the read command. Notice that the `while`-loop has no statements in its range, since all we want to do is to continuously poll the Status register to determine when the device has completed the command. When the device has completed, the contents of the device's Data register is copied to an application program variable, in this case, the char variable named `ch`.

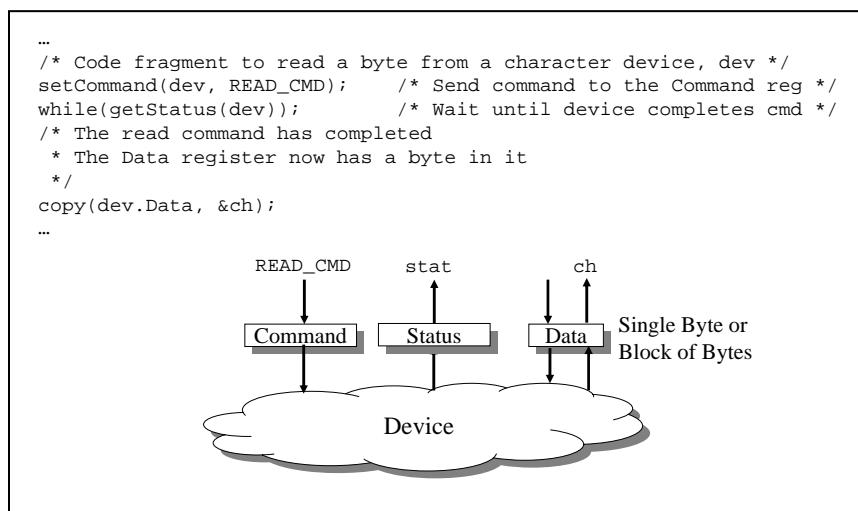


Figure 5-6: The Software Interface to a Device

Although this pseudo code omits all details of an actual OS function, it conveys the basic idea of how the device driver software can interact with the device controller hardware to cause it to read a byte. The device driver would provide a similar function to write a byte to the device: this would be done by copying the data into the controller's Data register, copying a write command into the controller's Command register, and then polling the Status register until the device reported that it had completed the operation using the status register.

Application software uses system calls to invoke the device driver functions (see [Section 4.3.1](#)). The system calls are generic for all devices (recall that they operate on abstract devices rather than physical devices). The system call on a particular device will result in the device manager passing the call to the proper entry point in the proper device driver. This enables the system call interface to have a single system call interface that works with every type of device. If necessary, the device manager can perform some preliminary processing to the system call before invoking the device driver to actually perform the I/O operation.

5.3 The Device Manager

Recall from our earlier discussion that part of the device manager is responsible for all aspects of device management, including writing commands to a device, reading and reacting to its status, and for transferring data to/from the device's data registers (see Figure 5-7). Obviously, each device (controller) has its own set of command to which it responds (for example, it is not usually possible to read data from a printer, or to write data to a mouse). As a result, it is necessary the OS must incorporate a specialized device driver to send the appropriate sequence of commands to each type of device, to properly interpret its status, and to handle the data transfer correctly. The shaded portion of Figure 5-7 represent the device drivers in a device manager organization: there is a unique device driver program for every particular type of device that is configured into the SCC – one for each type of sensor, actuator, communication device, and storage device.

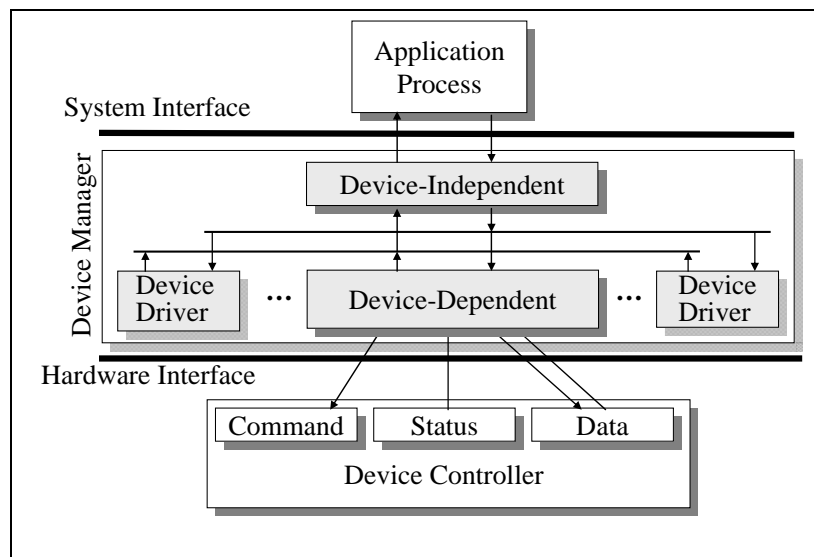


Figure 5-7: Device Management Organization

In the I/O portion of the system call interface described in Section 4.3.1, the device manager exports a single API for all abstract devices to application programs. This enables the application programmer to learn one API, and then to use it for all devices on the system. Of course, the general purpose API will have both read and write functions on it, but if the application program attempts to write to a mouse, the OS may simply ignore the system call, or it may result in an error – the choice of action in this case is part of the definition of the API semantics. So the device driver encapsulates the knowledge necessary to manipulate each type of device, while providing a generic API to the software that calls it. As suggested by the figure, device management software in modern operating systems is typically organized as a **device-dependent** part that encodes all of the specific knowledge required to manage a particular device type, and a device-independent part that generally provides an execution environment for device drivers, and which exports the standard device API for all devices. The **device-independent** part defines an infrastructure in which device drivers are invoked by application programs.

The computer can be configured with m different device types. Each device could have n_i different functions to control the device number i . That is, a programmer would have to be able to call any one of the n_i functions whenever it wants to control device number i . This means that the device manager would have to export have $n_0 + n_1 + \dots + n_{m-1}$ functions, each with a different function name. This would result in a huge system call interface, and worse, it would have to grow each time a new device type was added to a computer.

The hypothetical ST OS system call interface supports only 7 standard function names (`Open()`, `Close()`, `Read()`, `Write()`, `Poll()`, `Seek()` and `Ioctl()`). The device driver for each device type then implements all 7 operations on the corresponding device. Since devices may have very different behavior, how can the device manager API possibly export a fixed set of only 7 functions? This works by making some compromises: some devices can only be read, some only written, some have special commands to power the device up/down, and so on. Most devices commands can easily be cast as one of the 7 function names (`Read()` for all input style operations, `Write()` for output operations, and so on). Even though there are 7 possible functions to call for any device, any particular device driver *may not respond* to some of the system calls – a keyboard device will probably not respond to a `Write(...)` call – but the function name still exists on the system call interface. Finally the system call device interface contains a single “wild card” function, `Ioctl()`, that is used to perform special operations such as powering down a disk drive. The `Ioctl()` function takes an argument that is itself a command, hence `Ioctl()` can be used to issue any specialized command to a device.

Each of the drivers for the m different types of devices exports the 7 standard functions, with the implementation of the function depending on the details of the particular device. A disk driver `Read()` function implements a very different functionality from a keyboard device `Read()` function, but both have exactly the same name and arguments.

The device manager uses the device identification and the function name to call the appropriate device driver function: each of the standard function uses a device identification as an argument. Suppose `DEV_ID` is such an identification: then

```
d = Open(DEV_ID, ...);
...
Read(fid, buffer, bufferLength);
...
```

results in a call to the generic system function named `Read()` in the device-independent part of the device manager. The code schema for the `Read()` function for the d^{th} device is shown in Figure 5-8. When a user program wants to call the `Read()` function for the d^{th} device number, it issues a system call of the form `Read(d, ...)`, which invokes the `dev_read()` internal system function. If there is any processing that can be done that is common to all devices, it is performed in this function either before or after calling the device-specific code, `devj_read(...)`. Though it is not illustrated in the figure, each case in the `switch` statement could also have more code, for example, to package parameters correctly for the device-specific function call.

```

dev_read(devID, ...) {
// Processing common to all devices
...
switch(devID) {
case dev1:  dev1_read(...);
            break;
case dev2:  dev2_read(...);
            break;
...
case devM:  devM_read(...);
            break;
};
// Processing common to all devices
...
}

```

Figure 5-8: Device-Independent Function Call

In this design, whenever a new device is added to the system, the `dev_read()` function must be changed and recompiled. Of course it is even possible to avoid changing the device manager code by storing the device type (called the device **major number** in POSIX) and the specific ID within the device type (called the **minor number** in POSIX) in a table. This table might be an array of `struct MinorDevice_t` (shown in Figure 5-9) instances. Each device type would have a corresponding instance of `struct MajorDevice_t` to define the function entry points for the particular driver type. With this organization, a new device type can be added to the machine by defining a new driver (described by the `struct MajorDevice_t` instance) if necessary, and then creating a new instance of `struct MinorDevice_t` for the new device.

```

struct MajorDevice_t {
    int majorNum;
    int (*Open)(char *, int);
    int (*Close)(int);
    int (*Read)(int, void *, int);
    int (*Write)(int, void *, int);
    int (*Poll)(int);
    int (*Seek)(int*, int);
    int (*Ioctl)(int, int, char *);
    ...
};

struct MinorDevice_t {
    int minorNum;
    struct MajorDevice_t *driverFuncs;
    ...
};

```

Figure 5-9: A Table to Describe Device Drivers

5.4 Device Drivers

A **device driver** is a collection of device-dependent functions that abstracts the operation of the device to a fixed interface.

Each device driver implements the subset of the system call interface counterparts that it desires. If a function is not implemented in the device driver, then the system call forwarding table will contain a null pointer, meaning that the device manager will return an error code in response to the system call for that particular function on that particular device.

The device driver functions can be simple or complex, depending on the nature of the device controller and the physical device. For example a serial port device driver that works with a serial port device implemented by with a UART, will tend to have very little code to perform a read or write operation. On the other hand, the hard disk device driver read and write functions are likely to be complex. It is probably obvious to you, but it is worth emphasizing that people who write device driver code soon become quite expert at understanding exactly how the device hardware is controlled. They usually spend many hours poring over the device hardware documentation while writing the driver, and then many more hours debugging it. However, this investment of time means that application programmers can use the device without ever even seeing hardware documentation!

Certain aspects of a device cannot be hidden from the application programmer. In an ST system, the lack of device interrupts is one such aspect. ST applications use the **direct I/O** model, meaning that the CPU is responsible for transferring the data between the machine's primary memory and the device controller data registers. While managing the I/O, some aspect of the software must periodically poll the device `busy-done` flags to detect the operation's completion. If the OS only supports blocking reads and writes, then the device driver must incorporate a polling loop to test the status of the device (to determine when it has completed the I/O operation. If nonblocking reads and writes are used (as in our hypothetical system call interface, then a read or write system call semantics are for the call to *start* the device, but to return to the application program so that it can execute other code while the device is in operation. Thus the driver does not poll the device – the application program must do it (using the `Poll()` system call in our hypothetical system call interface).

Let's first consider the organization of the device drivers that use the blocking I/O paradigm, then we will consider the nonblocking case. Consider the organization of the `BlockingRead()` device driver function (see Figure 5-10) – the `BlockingWrite()` code would be similar. In other words, the CPU starts the device and then polls the status register to determine when the operation has completed. (see Figure 5-11):

1. The application process requests a read operation.
2. The device driver queries the status register to determine if the device is idle. If the device is busy, the driver waits for it to become idle.
3. The driver stores an input command into the controller's command register, thereby starting the device.
4. The driver repeatedly reads the status register while waiting for the device to complete its operation.
5. The driver copies the contents of the controller's data register(s) into the user process's space.

The steps to perform an output operation are as follows:

1. The application process requests a write operation.
2. The device driver queries the status register to determine if the device is idle. If the device is busy, the driver waits for it to become idle.
3. The driver copies data from user space memory to the controller's data register(s).
4. The driver stores an output command into the command register, thereby starting the device.
5. The driver repeatedly reads the status register while waiting for the device to complete its operation.

```

int BlockingRead(dev, buf, bufLen) {
    ...
    /* Step 2: Wait until device is available */
    while(getStatus(dev) != IDLE);
    /* Code fragment to read a byte from a character device, dev
    */
    /* Step 3: Send command to the Command reg */
    setCommand(dev, READ_CMD);
    /* Step 4: Wait until device completes cmd */
    while(getStatus(dev) != BUSY);
    if(getStatus(dev) == ERROR) {
        /* Error recovery here ... */
    }
    ...
}
/* Step 5: The read command has completed
 * The Data register now has a byte in it
 */
for(i=0; i<bufLen; i++)
    copy(dev.Data[i], &(buf[i]));
/* Let the device know we have emptied the controller regs */
setCommand(dev, READ_FINISH);
...
}

```

Figure 5-10: Blocking Read Driver Skeleton

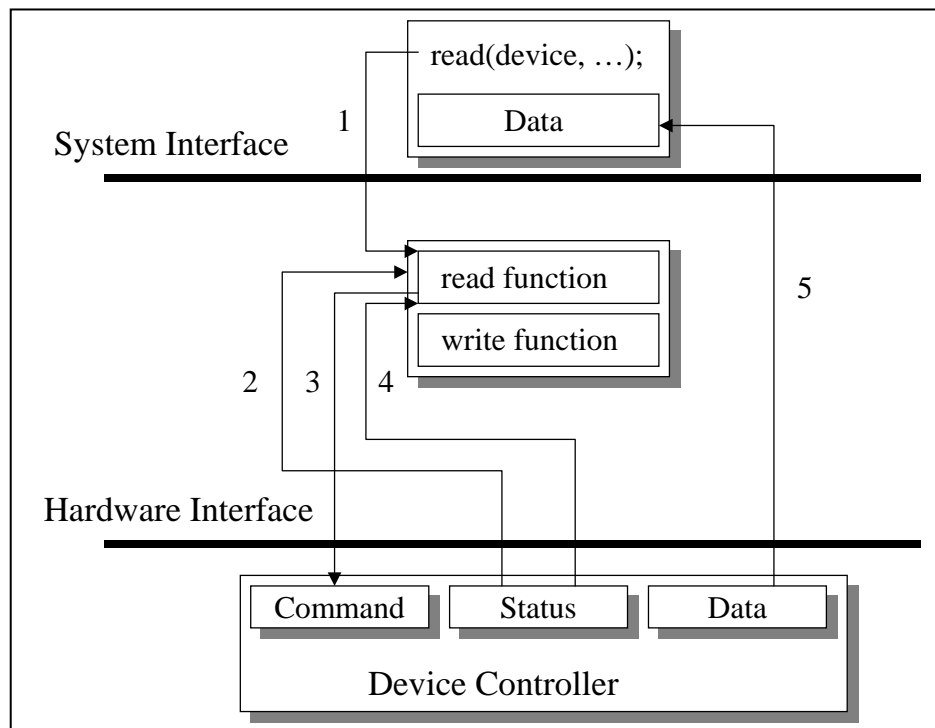


Figure 5-11: Blocking Read Operation

Next, let's consider the difference in the driver for a `NonblockingRead()` – see fig X. The driver initiates the I/O operation, but then returns control to the application program. When the application program needs to determine the status of the device, it will use the `Poll()` system call. For example, could immediately go into a while-loop to poll the device after it calls `NonblockingRead()`.

```

int NonblockingRead(dev, buf, buflen) {
    ...
    /* Wait until device is available */
    while(getStatus(dev) != IDLE);
    /* Code fragment to read a byte from a character device, dev
    */
    /* Send command to the Command reg */
    setCommand(dev, READ_CMD);
    /* Return control to the app - it will be responsible for
    * determining when the device has completed
    */
    ...
}

```

Figure 5-12: Nonblocking Read Driver Skeleton

5.5 I/O-Processor Overlap within an Application

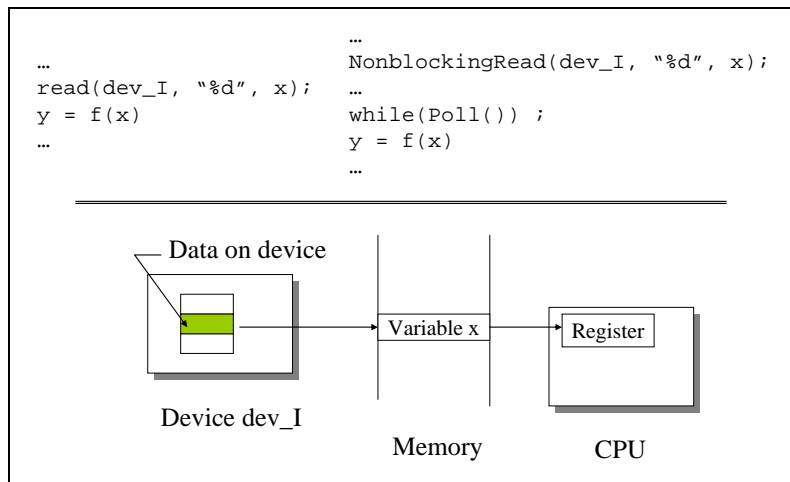


Figure 5-13: Overlapping the Operation of a Device and the CPU

C/Unix programmers have a preconceived model of the semantics of I/O operations, since the default form of the read is for it to block until the operation completes. That is, these programmers generally expect sequential execution semantics for the I/O operation, meaning that read and write operations behave as if they were sequential operations. For example suppose an AUC executes code such as the code on the left side of Figure 5-13 shows the situation while the `read()` function is being executed. The device driver has started the `dev_I` device but the operation has not completed. If an AUC were to execute the assignment statement, $y = f(x)$, at that moment, then the $f(x)$ function would be executed using an old value of x , not the new one being read from the device. To prevent this situation, the device manager suspends the AUC until the `read()` call (a `BlockingRead()` call in the schema above) has completed. From the AUC's perspective, the abstract machine environment waits for the device to complete the I/O operation before it executes the assignment statement.

More complex semantics might allow the programmer to initiate the `read()` operation—that is, to *start* the device and then to continue processing without waiting for it to complete (see the code on the right side of Figure 5-13). In order to support serial execution semantics, the device manager would have to export functions to `NonBlockingRead()` and the `Poll()` function to check the device status. These would be alternatives to the sequential execution `read()` function.

Figure 5-14 shows another way for thinking about this kind of CPU and device execution (illustrated using a **Gantt chart**). Suppose the application program uses the `NonBlockingRead()` and the `Poll()` functions:

- At time t_1 , the processor starts the controller with the `NonBlockingRead()` operation but then continues using the CPU. As long as the application does not need the result of the read operation, it can execute the program on the CPU at the same time the I/O operation is taking place.
- At time t_2 the code needs the result of the pending read operation, it calls `Poll()`. This will cause the execution to wait in the `while`-loop until the read has completed.
- At time t_3 the device completes the I/O operation and the executing program detects this fact with the result from the `Poll()` function. The program can now use the result of the read.
- At time t_4 , the AUC again starts the device and executes other instructions until it needs the results of the I/O at time t_5 , and so on.

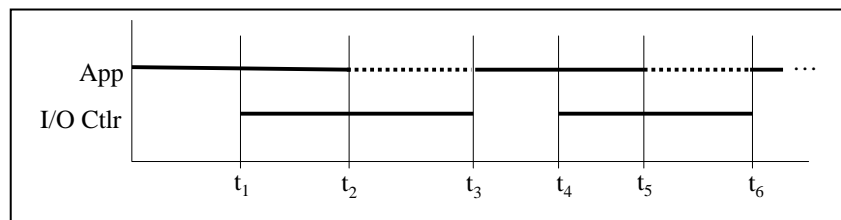


Figure 5-14: Overlapping CPU-Controller Operations in a Process

This overlapped CPU-device operation results in an overall reduction in the amount of time it takes to execute the program and its I/O serially. From time t_1 to t_2 and from t_4 to t_5 , the CPU and device are operating in parallel, thus reducing the overall runtime from the sum of the two time intervals to the maximum of their time intervals. From t_2 to t_3 and from t_5 to t_6 , the process is in a busy-wait state, so from the viewpoint of achieving effective computation, the use of the CPU is wasted, since there is no overlapped operation. Further, since this is a single-threaded system, there is no other thread to execute while the program waits for the device to complete the read operation.

5.6 Summary

Devices can take on many different characteristics, since they represent the connection between the computer and its physical environment. Communication devices (or a computer and a remote device) are used to allow different computers to share information. Storage devices provide a means for software to save data and programs while the machine is not in use. There are a growing number of specialized I/O devices (including sensors and actuators) that are used to read thermometers, or other devices, and to write control information to other external machinery (such as a heater or cooler, or a guidance system).

Devices vary widely in their characteristics, so device drivers are implemented to export a common API. The trend toward open systems has encouraged OS designers to make it easy for a systems administrator to add a device and driver to the system without having to change the OS source code. POSIX and Windows employ this technique by providing tools to the administrator to assist in configuring new devices as they are added.

Device drivers are the heart of the device manager. Each type of device has a unique device driver that implements a common set of driver functions (such as `open()`, `close()`, `read()`, and `write()` in POSIX).