7.1 Chapter Overview

A typical program does three basic activities: input, computation, and output. In this section we will discuss the other two activities beyond computation: input and output or I/O. This chapter concentrates on low-level CPU I/O rather than high level file or character I/O. This chapter discusses how the CPU transfers bytes of data to and from the outside world. This chapter discusses the mechanisms and performance issues behind the I/O.

7.2 Connecting a CPU to the Outside World

Most I/O devices interface to the CPU in a fashion quite similar to memory. Indeed, many devices appear to the CPU as though they were memory devices. To output data to the outside world the CPU simply stores data into a "memory" location and the data magically appears on some connectors external to the computer. Similarly, to input data from some external device, the CPU simply transfers data from a "memory" location into the CPU; this "memory" location holds the value found on the pins of some external connector.

An output port is a device that looks like a memory cell to the computer but contains connections to the outside world. An I/O port typically uses a latch rather than a flip-flop to implement the memory cell. When the CPU writes to the address associated with the latch, the latch device captures the data and makes it available on a set of wires external to the CPU and memory system (see Figure 7.1). Note that output ports can be write-only, or read/write. The port in Figure 7.1, for example, is a write-only port. Since the outputs on the latch do not loop back to the CPU s data bus, the CPU cannot read the data the latch contains. Both the address decode and write control lines must be active for the latch to operate; when reading from the latch s address the decode line is active, but the write control line is not.

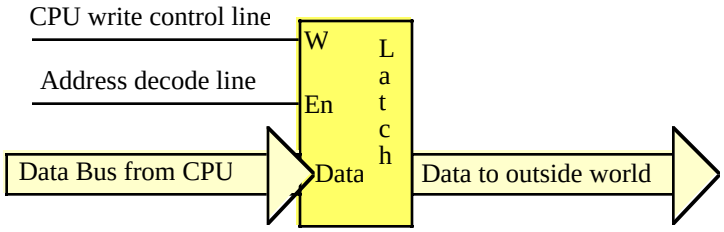


Figure 7.1 A Typical Output Port

Figure 7.2 shows how to create a read/write input/output port. The data written to the output port loops back to a transparent latch. Whenever the CPU reads the decoded address the read and decode lines are active and this activates the lower latch. This places the data previously written to the output port on the CPU s data bus, allowing the CPU to read that data. A read-only (input) port is simply the lower half of Figure 7.2; the system ignores any data written to an input port.

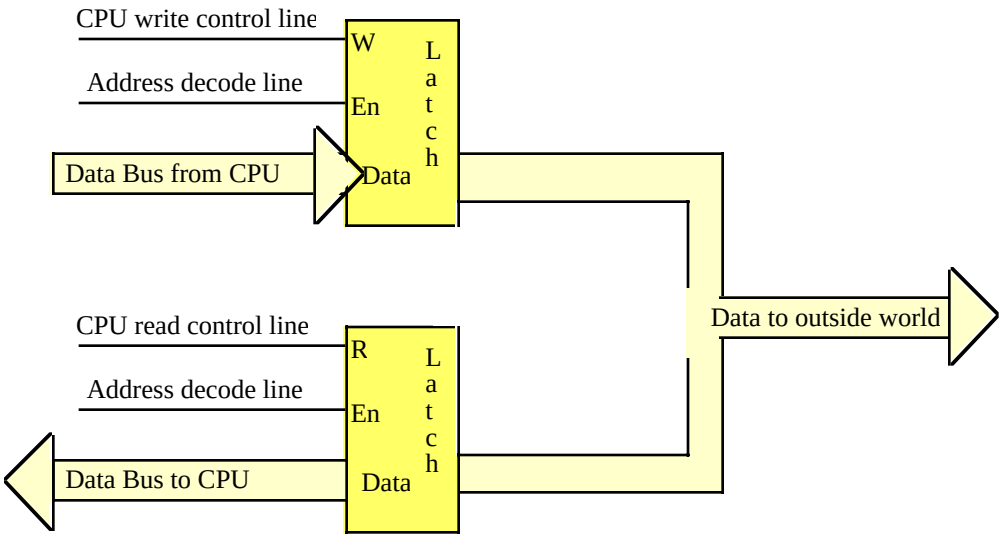


Figure 7.2 An Output Port that Supports Read/Write Access

Note that the port in Figure 7.2 is not an input port. Although the CPU can read this data, this port organization simply lets the CPU read the data it previously wrote to the port. The data appearing on an external connector is an output port (only). One could create a (read-only) input port by using the lower half of the circuit in Figure 7.2. The input to the latch would appear on the CPU s data bus whenever the CPU reads the latch data.

A perfect example of an output port is a parallel printer port. The CPU typically writes an ASCII character to a byte-wide output port that connects to the DB-25F connector on the back of the computer s case. A cable transmits this data to the printer where an input port (to the printer) receives the data. A processor inside the printer typically converts this ASCII character to a sequence of dots it prints on the paper.

Generally, a given peripheral device will use more than a single I/O port. A typical PC parallel printer interface, for example, uses three ports: a read/write port, an input port, and an output port. The read/write port is the data port (it is read/write to allow the CPU to read the last ASCII character it wrote to the printer port). The input port returns control signals from the printer; these signals indicate whether the printer is ready to accept another character, is off-line, is out of paper, etc. The output port transmits control information to the printer such as whether data is available to print.

The first thing to learn about the input/output subsystem is that I/O in a typical computer system is radically different than I/O in a typical high level programming language. In a real computer system you will rarely find machine instructions that behave like *writeln*, *cout*, *printf*, or even the HLA *stdin* and *stdout* statements. In fact, most input/output instructions behave exactly like the 80x86 s MOV instruction. To send data to an output device, the CPU simply moves that data to a special memory location. To read data from an input device, the CPU simply moves data from the address of that device into the CPU. Other than there are usually more wait states associated with a typical peripheral device than actual memory, the input or output operation looks very similar to a memory read or write operation.

7.3 Read-Only, Write-Only, Read/Write, and Dual I/O Ports

We can classify input/output ports into four categories based on the CPU s ability to read and write data at a given port address. These four categories are read-only ports, write-only ports, read/write ports, and dual I/O ports.

A read-only port is (obviously) an input port. If the CPU can only read the data from the port, then that port is providing data appearing on lines external to the CPU. The system typically ignores any attempt to write data to a read-only port[1]. A good example of a read-only port is the status port on a PC s parallel printer interface. Reading data from this port lets you test the current condition of the printer. The system ignores any data written to this port.

A write-only port is always an output port. Writing data to such a port presents the data for use by an external device. Attempting to read data from a write-only port generally returns garbage (i.e., whatever values that just happen to be on the data bus at that time). You generally cannot depend on the meaning of any value read from a write-only port.

A read/write port is an output port as far as the outside world is concerned. However, the CPU can read as well as write data to such a port. Whenever the CPU reads data from a read/write port, it reads the data that was last written to the port. Reading the port does not affect the data the external peripheral device sees, reading the port is a simple convenience for the programmer so that s/he doesn t have to save the value last written to the port should they want to retrieve the value.

A dual I/O port is also a read/write port, but reading the port reads data from some external device while writing data to the port transmits data to a different external device. Figure 7.3 shows how you could interface such a device to the system. Note that the input and output ports are actually a read-only and a write-only port that share the same address. Reading the address accesses one port while writing to the address accesses the other port. Essentially, this port arrangement uses the R/W control line(s) as an extra address bit when selecting these ports.

1. Note, however, that some devices may fail if you attempt to write to their corresponding input ports, so it s never a good idea to write data to a read-only port.

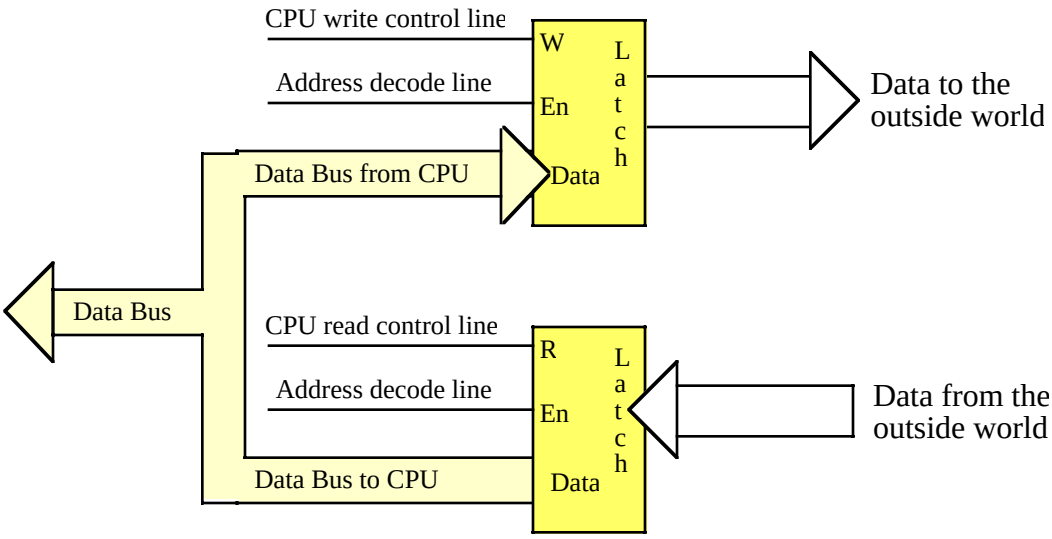


Figure 7.3 An Input and an Output Device That Share the Same Address (a Dual I/O Port)

These examples may leave you with the impression that the CPU always reads and writes data to peripheral devices using data on the data bus (that is, whatever data the CPU places on the data bus when it writes to an output port is the data actually written to that output port). While this is generally true for input ports (that is, the CPU transfers input data across the data bus when reading data from the input port), this isn t necessarily true for output ports. In fact, a very common output mechanism is simply accessing a port. Figure 7.4 provides a very simple example. In this circuit, an address decoder decodes two separate addresses. Any access (read or write) to the first address sets the output line high; any read or write of the second address clears the output line. Note that this circuit ignores the data on the CPU s data lines. It is not important whether the CPU reads or writes data to these addresses, nor is the data written of any consequence. The only thing that matters is that the CPU access one of these two addresses.

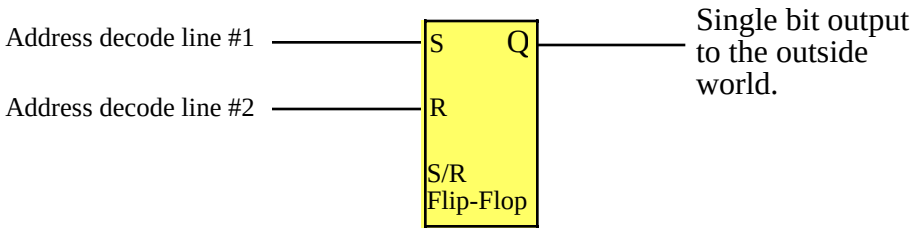


Figure 7.4 Outputting Data to a Port by Simply Accessing That Port

Another possible way to connect an output port to the CPU is to use a D flip-flop and connect the read/write status lines to the D input on the flip-flop. Figure 7.5 shows how you could design such a device. In this diagram any read of the selected port sets the output bit to zero while a write to this output port sets the output bit to one.

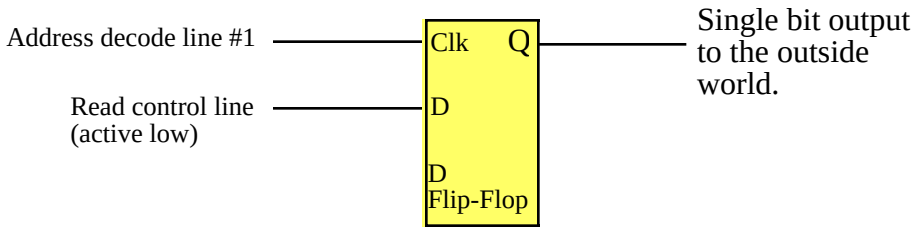


Figure 7.5 Outputting Data Using the Read/Write Control as the Data to Output

There are a wide variety of ways you can connect external devices to the CPU. This section only provides a few examples as a sampling of what is possible. In the real world, there are an amazing number of different ways that engineers connect external devices to the CPU. Unless otherwise noted, the rest of this chapter will assume that the CPU reads and writes data to an external device using the data bus. This is not to imply that this is the only type of I/O that one could use in a given example.

7.4 I/O (Input/Output) Mechanisms

There are three basic forms of input and output that a typical computer system will use: I/O-mapped I/O, memory-mapped I/O, and direct memory access (DMA). I/O-mapped input/output uses special instructions to transfer data between the computer system and the outside world; memory-mapped I/O uses special memory locations in the normal address space of the CPU to communicate with real-world devices; DMA is a special form of memory-mapped I/O where the peripheral device reads and writes data in memory without going through the CPU. Each I/O mechanism has its own set of advantages and disadvantages, we will discuss these in this section.

7.4.1 Memory Mapped Input/Output

A memory mapped peripheral device is connected to the CPU s address and data lines exactly like memory, so whenever the CPU reads or writes the address associated with the peripheral device, the CPU transfers data to or from the device. This mechanism has several benefits and only a few disadvantages.

The principle advantage of a memory-mapped I/O subsystem is that the CPU can use any instruction that accesses memory to transfer data between the CPU and a memory-mapped I/O device. The MOV instruction is the one most commonly used to send and receive data from a memory-mapped I/O device, but any instruction that reads or writes data in memory is also legal. For example, if you have an I/O port that is read/write, you can use the ADD instruction to read the port, add data to the value read, and then write data back to the port.

Of course, this feature is only usable if the port is a read/write port (or the port is readable and you ve specified the port address as the source operand of your ADD instruction). If the port is read-only or write-only, an instruction that reads memory, modifies the value, and then writes the modified value back to memory will be of little use. You should use such read/modify/write instructions only with read/write ports (or dual I/O ports if such an operation makes sense).

Nevertheless, the fact that you can use any instruction that accesses memory to manipulate port data is often a big advantage since you can operate on the data with a single instruction rather than first moving the data into the CPU, manipulating the data, and then writing the data back to the I/O port.

The big disadvantage of memory-mapped I/O devices is that they consume addresses in the memory map. Generally, the minimum amount of space you can allocate to a peripheral (or block of related peripherals) is a four kilobyte page. Therefore, a few independent peripherals can wind up consuming a fair amount of the physical address space. Fortunately, a typical PC has only a couple dozen such devices, so this isn t much of a problem. However, some devices, like video cards, consume a large chunk of the address space (e.g., some video cards have 32 megabytes of on-board memory that they map into the memory address space).

7.4.2 I/O Mapped Input/Output

I/O-mapped input/output uses special instructions to access I/O ports. Many CPUs do not provide this type of I/O, though the 80x86 does. The Intel 80x86 family uses the IN and OUT instructions to provide I/O-mapped input/output capabilities. The 80x86 IN and OUT instructions behave somewhat like the MOV instruction except they transmit their data to and from a special I/O address space that is distinct from the memory address space. The IN and OUT instructions use the following syntax:

```
in( port, al );  // ... or AX or EAX, port is a constant in the range
out( al, port ); // 0..255.


in( dx, al );    // Or AX or EAX.
out( al, dx );
```

The 80x86 family uses a separate address bus for I/O transfers[2]. This bus is only 16-bits wide, so the 80x86 can access a maximum of 65,536 different bytes in the I/O space. The first two instructions encode the port address as an eight-bit constant, so they re actually limited to accessing only the first 256 I/O addresses in this address space. This makes the instruction shorter (two bytes instead of three). Unfortunately, most of the interesting peripheral devices are at addresses above 255, so the first pair of instructions above are only useful for accessing certain on-board peripherals in a PC system.

To access I/O ports at addresses beyond 255 you must use the latter two forms of the IN and OUT instructions above. These forms require that you load the 16-bit I/O address into the DX register and use DX as a pointer to the specified I/O address. For example, to write a byte to the I/O address $378[3] you would use an instruction sequence like the following:

```
mov( $378, dx );
out( al, dx );
```

The advantage of an I/O address space is that peripheral devices mapped to this area do not consume space in the memory address space. This allows you to fully expand the memory address space with RAM or other memory. On the other hand, you cannot use arbitrary memory instructions to access peripherals in the I/O address space, you can only use the IN and OUT instructions.

Another disadvantage to the 80x86 s I/O address space is that it is quite small. Although most peripheral devices only use a couple of I/O address (and most use fewer than 16 I/O addresses), a few devices, like video display cards, can occupy millions of different I/O locations (e.g., three bytes for each pixel on the screen). As

2. Physically, the I/O address bus is the same as the memory address bus, but additional control lines determine whether the address on the bus is accessing memory or and I/O device.
3. This is typically the address of the data port on the parallel printer port.

noted earlier, some video display cards have 32 megabytes of dual-ported RAM on board. Clearly we cannot easily map this many locations into the 64K I/O address space.

7.4.3 Direct Memory Access

Memory-mapped I/O subsystems and I/O-mapped subsystems both require the CPU to move data between the peripheral device and main memory. For this reason, we often call these two forms of input/output programmed I/O. For example, to input a sequence of ten bytes from an input port and store these bytes into memory the CPU must read each value and store it into memory. For very high-speed I/O devices the CPU may be too slow when processing this data a byte (or word or double word) at a time. Such devices generally have an interface to the CPU s bus so they can directly read and write memory. This is known as direct memory access since the peripheral device accesses memory directly, without using the CPU as an intermediary. This often allows the I/O operation to proceed in parallel with other CPU operations, thereby increasing the overall speed of the system. Note, however, that the CPU and DMA device cannot both use the address and data busses at the same time. Therefore, concurrent processing only occurs if the CPU has a cache and is executing code and accessing data found in the cache (so the bus is free). Nevertheless, even if the CPU must halt and wait for the DMA operation to complete, the I/O is still much faster since many of the bus operations during I/O or memory-mapped input/output consist of instruction fetches or I/O port accesses which are not present during DMA operations.

A typical DMA controller consists of a pair of counters and other circuitry that interfaces with memory and the peripheral device. One of the counters serves as an address register. This counter supplies an address on the address bus for each transfer. The second counter specifies the number of transfers to complete. Each time the peripheral device wants to transfer data to or from memory, it sends a signal to the DMA controller. The DMA controller places the value of the address counter on the address bus. At the same time, the peripheral device places data on the data bus (if this is an input operation) or reads data from the data bus (if this is an output operation). After a successful data transfer, the DMA controller increments its address register and decrements the transfer counter. This process repeats until the transfer counter decrements to zero.

7.5 I/O Speed Hierarchy

Different devices have different data transfer rates. Some devices, like keyboards, are extremely slow (comparing their speed to CPU speeds). Other devices like disk drives can actually transfer data faster than the CPU can read it. The mechanisms for data transfer differ greatly based on the transfer speed of the device. Therefore, it makes sense to create some terminology to describe the different transfer rates of peripheral devices.

Low-speed devices are those that produce or consume data at a rate much slower than the CPU is capable of processing. For the purposes of discussion, we ll claim that low-speed devices operate at speeds that are two to three orders of magnitude (or more) slower than the CPU. *Medium-speed devices* are those that transfer data at approximately the same rate (within an order of magnitude slower, but never faster) than the CPU. *High-speed devices* are those that transfer data faster than the CPU is capable of moving data between the device and the CPU. Clearly, high-speed devices must use DMA since the CPU is incapable of transferring the data between the CPU and memory.

With typical bus architectures, modern day PCs are capable of one transfer per microsecond or better. Therefore, high-speed devices are those that transfer data more rapidly than once per microsecond. Medium-speed transfers are those that involve a data transfer every one to 100 microseconds. Low-speed devices usually trans-

fer data less often than once every 100 microseconds. The difference between these speeds will decide the mechanism we use for the I/O operation (e.g., high-speed transfers require the use of DMA or other techniques).

Note that one transfer per microsecond is not the same thing as a one megabyte per second data transfer rate. A peripheral device can actually transfer more than one byte per data transfer operation. For example, when using the "in(dx, eax);" instruction, the peripheral device can transfer four bytes in one transfer. Therefore, if the device is reaching one transfer per microsecond, then the device can transfer four megabytes per second. Likewise, a DMA device on a Pentium processor can transfer 64 bits at a time, so if the device completes one transfer per microsecond it will achieve an eight megabyte per second data transfer rate.

7.6 System Busses and Data Transfer Rates

Earlier in this text (see The System Bus on page 138) you saw that the CPU communicates to memory and I/O devices using the system bus. In that chapter you saw that a typical Von Neumann Architecture machine has three different busses: the address bus, the data bus, and the control bus. If you ve ever opened up a computer and looked inside or read the specifications for a system, you ve probably heard terms like *PCI, ISA, EISA*, or even *NuBus* mentioned when discussing the computer s bus. If you re familiar with these terms, you may wonder what their relationship is with the CPU s bus. In this section we ll discuss this relationship and describe how these different busses affect the performance of a system.

Computer system busses like PCI (Peripheral Connection Interface) and ISA (Industry Standard Architecture) are definitions for physical connectors inside a computer system. These definitions describe a set of signals, physical dimensions (i.e., connector layouts and distances from one another), and a data transfer protocol for connecting different electronic devices. These busses are related to the CPU s bus only insofar as many of the signals on one of the peripheral busses also appear on the CPU s bus. For example, all of the aforementioned busses provide lines for address, data, and control functions.

Peripheral interconnection busses do not necessarily mirror the CPU s bus. All of these busses contain several additional lines that are not present on the CPU s bus. These additional lines let peripheral devices communicate with one other directly (without having to go through the CPU or memory). For example, most busses provide a common set of interrupt control signals that let various I/O devices communicate directly with the system s interrupt controller (which is also a peripheral device). Nor does the peripheral bus always include all the signals found on the CPU s bus. For example, the ISA bus only supports 24 address lines whereas the Pentium IV supports 36 address lines. Therefore, peripherals on the ISA bus only have access to 16 megabytes of the Pentium IV s 64 gigabyte address range.

A typical modern-day PC supports the PCI bus (although some older systems also provide ISA connectors). The organization of the PCI and ISA busses in a typical computer system appears in Figure 7.6.

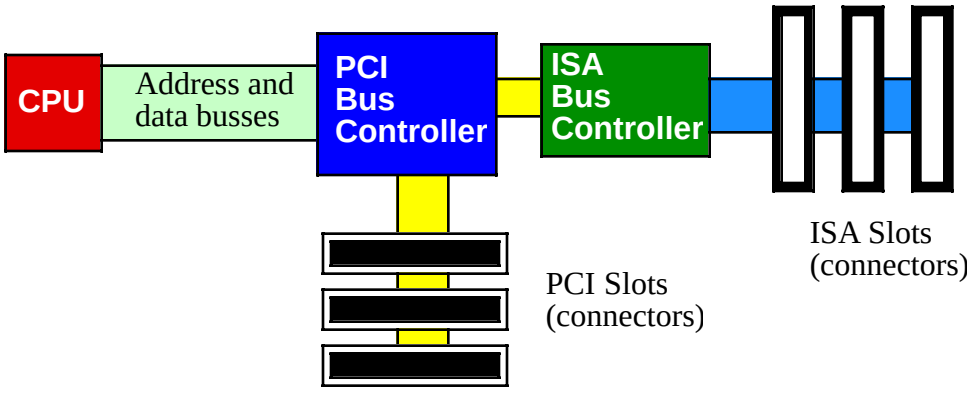


Figure 7.6 Connection of the PCI and ISA Busses in a Typical PC

Notice how the CPU s address and data busses connect to a PCI Bus Controller device (which is, itself, a peripheral of sorts). The actual PCI bus is connected to this chip. Note that the CPU does not connect directly to the PCI bus. Instead, the PCI Bus Controller acts as an intermediary, rerouting all data transfer requests between the CPU and the PCI bus.

Another interesting thing to note is that the ISA Bus Controller is not directly connected to the CPU. Instead, it is connected to the PCI Bus Controller. There is no logical reason why the ISA Controller couldn t be connected directly to the CPU s bus, however, in most modern PCs the ISA and PCI controllers appear on the same chip and the manufacturer of this chip has chosen to interface the ISA bus through the PCI controller for cost or performance reasons.

The CPU s bus (often called the local bus) usually runs at some submultiple of the CPU s frequency. Typical local bus frequencies include 66 MHz, 100 MHz, 133 MHz, 400 MHz, and, possibly, beyond[4]. Usually, only memory and a few selected peripherals (e.g., the PCI Bus Controller) sit on the CPU s bus and operate at this high frequency. Since the CPU s bus is typically 64 bits wide (for Pentium and later processors) and it is theoretically possible to achieve one data transfer per cycle, the CPU s bus has a maximum possible data transfer rate (or maximum bandwidth) of eight times the clock frequency (e.g., 800 megabytes/second for a 100 Mhz bus). In practice, CPU s rarely achieve the maximum data transfer rate, but they do achieve some percentage of this, so the faster the bus, the more data can move in and out of the CPU (and caches) in a given amount of time.

The PCI bus comes in several configurations. The base configuration has a 32-bit wide data bus operating at 33 MHz. Like the CPU s local bus, the PCI is theoretically capable of transferring data on each clock cycle. This provides a theoretical maximum of 132 MBytes/second data transfer rate (33 MHz times four bytes). In practice, the PCI bus doesn t come anywhere near this level of performance except in short bursts. Whenever the CPU wishes to access a peripheral on the PCI bus, it must negotiate with other peripheral devices for the right to use the bus. This negotiation can take several clock cycles before the PCI controller grants the CPU the bus. If a CPU writes a sequence of values to a peripheral a double word per bus request, then the negotiation takes the majority of the time and the data transfer rate drops dramatically. The only way to achieve anywhere near the maximum theoretical bandwidth on the bus is to use a DMA controller and move blocks of data. In this block mode the DMA controller can negotiate just once for the bus and transfer a fair sized block of data without giving up the bus between each transfer. This "burst mode" allows the device to move lots of data quickly.

There are a couple of enhancements to the PCI bus that improve performance. Some PCI busses support a 64-bit wide data path. This, obviously, doubles the maximum theoretical data transfer rate. Another enhance-

4. 400 MHz was the maximum CPU bus frequency as this was being written.

ment is to run the bus at 66 MHz, which also doubles the throughput. In theory, you could have a 64-bit wide 66 MHz bus that quadruples the data transfer rate (over the performance of the baseline configuration). Few systems or peripherals currently support anything other than the base configuration, but these optional enhancements to the PCI bus allow it to grow with the CPU as CPUs increase their performance.

The ISA bus is a carry over from the original PC/AT computer system. This bus is 16 bits wide and operates at 8 MHz. It requires four clock cycles for each bus cycle. For this and other reasons, the ISA bus is capable of about only one data transmission per microsecond. With a 16-bit wide bus, data transfer is limited to about two megabytes per second. This is much slower than the CPU s local bus and the PCI bus . Generally, you would only attach low-speed devices like an RS-232 communications device, a modem, or a parallel printer to the ISA bus. Most other devices (disks, scanners, network cards, etc.) are too fast for the ISA bus. The ISA bus is really only capable of supporting low-speed and medium speed devices.

Note that accessing the ISA bus on most systems involves first negotiating for the PCI bus. The PCI bus is so much faster than the ISA bus that this has very little impact on the performance of peripherals on the ISA bus. Therefore, there is very little difference to be gained by connecting the ISA controller directly to the CPU s local bus.

7.7 The AGP Bus

Video display cards are a very special peripheral that need the maximum possible amount of bus bandwidth to ensure quick screen updates and fast graphic operations. Unfortunately, if the CPU has to constantly negotiate with other peripherals for the use of the PCI bus, graphics performance can suffer. To overcome this problem, video card designers created the AGP (Advanced Graphics Port) interface between the CPU and the video display card.

The AGP is a secondary bus interface that a video card uses in addition to the PCI bus. The AGP connection lets the CPU quickly move data to and from the video display RAM. The PCI bus provides a connection to the other I/O ports on the video display card (see Figure 7.7). Since there is only one AGP port per system, only one card can use the AGP and the system never has to negotiate for access to the AGP bus.

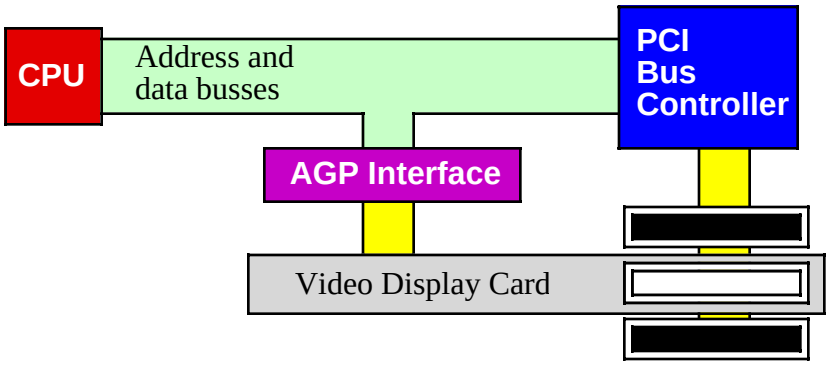


Figure 7.7 AGP Bus Interface

Buffering

If a particular I/O device produces or consumes data faster than the system is capable of transferring data to that device, the system designer has two choices: provide a faster connection between the CPU and the device or slow down the rate of transfer between the two.

Creating a faster connection is possible if the peripheral device is already connected to a slow bus like ISA. Another possibility is going to a wider bus (e.g., to the 64-bit PCI bus) to increase bandwidth, or to use a bus with a higher frequency (e.g., a 66 MHz bus rather than a 33 MHz bus). Systems designers can sometimes create a faster interface to the bus; the AGP connection is a good example. However, once you re using the fastest bus available on the system, improving system performance by selecting a faster connection to the computer can be very expensive.

The other alternative is to slow down the transfer rate between the peripheral and the computer system. This isn t always as bad as it seems. Most high-speed devices don t transfer data at a constant rate to the system. Instead, devices typically transfer a block of data rapidly and then sit idle for some period of time. Although the burst rate is high (and faster than the CPU or system can handle), the average data transfer rate is usually lower than what the CPU/system can handle. If you could average out the peaks and transfer some of the data when the peripheral is inactive, you could easily move data between the peripheral and the computer system without resorting to an expensive, high-bandwidth, solution.

The trick is to use memory to buffer the data on the peripheral side. The peripheral can rapidly fill this buffer with data (or extract data from the buffer). Once the buffer is empty (or full) and the peripheral device is inactive, the system can refill (or empty) the buffer at a sustainable rate. As long as the average data rate of the peripheral device is below the maximum bandwidth the system will support, and the buffer is large enough to hold bursts of data to/from the peripheral, this scheme lets the peripheral communicate with the system at a lower data transfer rate than the device requires during burst operation.

7.8 Handshaking

Many I/O devices cannot accept data at an arbitrary rate. For example, a Pentium based PC is capable of sending several hundred million characters a second to a printer, but that printer is (probably) unable to print that many characters each second. Likewise, an input device like a keyboard is unable to provide several million keystrokes per second (since it operates at human speeds, not computer speeds). The CPU needs some mechanism to coordinate data transfer between the computer system and its peripheral devices.

One common way to coordinate data transfer is to provide some status bits in a secondary input port. For example, a one in a single bit in an I/O port can tell the CPU that a printer is ready to accept more data, a zero would indicate that the printer is busy and the CPU should not send new data to the printer. Likewise, a one bit in a different port could tell the CPU that a keystroke from the keyboard is available at the keyboard data port, a zero in that same bit could indicate that no keystroke is available. The CPU can test these bits prior to reading a key from the keyboard or writing a character to the printer.

Using status bits to indicate that a device is ready to accept or transmit data is known as handshaking. It gets this name because the protocol is similar to two people agreeing on some method of transfer by a hand shake.

Figure 7.8 shows the layout of the parallel printer port s status register. For the LPT1: printer interface, this port appears at I/O address $379. As you can see from this diagram, bit seven determines if the printer is capable of receiving data from the system; this bit will contain a one when the printer is capable of receiving data.

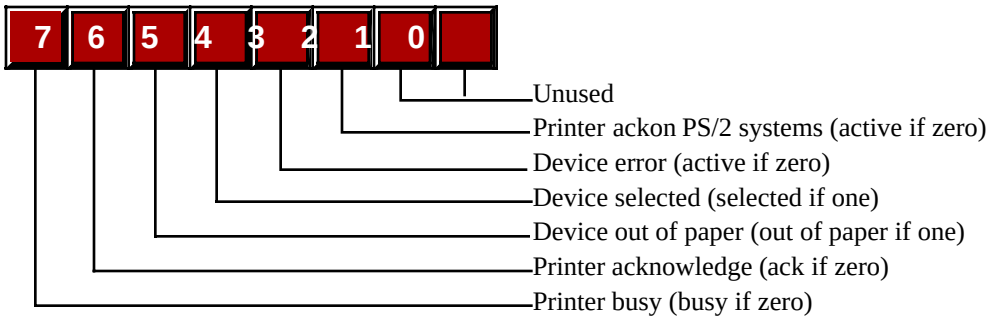


Figure 7.8 The Parallel Port Status Port

The following short program segment will continuously loop while the H.O. bit of the printer status register contains zero and will exit once the printer is ready to accept data:

```
mov( $379, dx );
repeat

    in( dx, al );
    and( $80, al );   // Clears Z flag if bit seven is set.

until( @nz );

// Okay to write another byte to the printer data port here.
```

The code above begins by setting DX to $379 since this is the I/O address of the printer status port. Within the loop the code reads a byte from the status port (the IN instruction) and then tests the H.O. bit of the port using the AND instruction. Note that logically ANDing the AL register with $80 will produce zero if the H.O. bit of AL was zero (that is, if the byte read from the input port was zero). Similarly, logically anding AL with $80 will produce $80 (a non-zero result) if the H.O. bit of the printer status port was set. The 80x86 zero flag reflects the result of the AND instruction; therefore, the zero flag will be set if AND produces a zero result, it will be reset otherwise. The REPEAT..UNTIL loop repeats this test until the AND instruction produces a non-zero result (meaning the H.O. bit of the status port is set).

One problem with using the AND instruction to test bits as the code above is that you might want to test other bits in AL once the code leaves the loop. Unfortunately, the "and($80, al);" instruction destroys the values of the other bits in AL while testing the H.O. bit. To overcome this problem, the 80x86 supports another form of the AND instruction —TEST. The TEST instruction works just like AND except it only updates the flags; it does not store the result of the logical AND operation back into the destination register (AL in this case). One other advantage to TEST is that it only reads its operands, so there are less problems with data hazards when using this instruction (versus AND). Also, you can safely use the TEST instruction directly on read-only memory-mapped I/O ports since it does not write data back to the port. As an example, let s recode the previous loop using the TEST instruction:

```
mov( $379, dx );
```

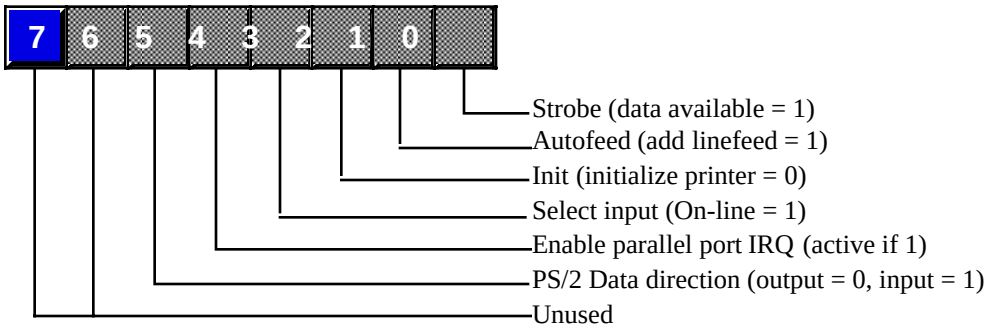
```
repeat

    in( dx, al );
    test( $80, al );   // Clears Z flag if bit seven is set.

until( @nz );

// Okay to write another byte to the printer data port here.
```

Once the H.O. bit of the printer status port is set, it s okay to transmit another byte to the printer. The computer can make a byte available by storing the byte data into I/O address $378 (for LPT1:). However, simply storing data to this port does not inform the printer that it can take the byte. The system must complete the other half of the handshake operation and send the printer a signal to indicate that a byte is available.



Parallel Port Control Register

Figure 7.9 The Parallel Port Command Register

Bit zero (the strobe line) must be set to one and then back to zero when the CPU makes data available for the printer (the term "strobe" suggests that the system pulses this line in the command port). In order to pulse this bit without affecting the other control lines, the CPU must first read this port, OR a one into the L.O. bit, write the data to the port, then mask out the L.O. bit using an AND instruction, and write the final result back to the control port again. Therefore, it takes three accesses (a read and two writes) to send the strobe to the printer. The following code handles this transmission:

```
mov( $378, dx );       // Data port address
mov( Data2Xmit, al );  // Send the data to the printer.
out( al, dx );

mov( $37a, dx );       // Point DX at the control port.
in( dx, al );          // Get the current port setting.
or( 1, al );           // Set the L.O. bit.
out( al, dx );         // Set the strobe line high.
and( $fe, al );        // Clear the L.O. bit.
```

```
out( al, dx );          // Set the strobe line low.
```

The code above would normally follow the REPEAT..UNTIL loop in the previous example. To transmit a second byte to the printer you would jump back to the REPEAT..UNTIL loop and wait for the printer to consume the current byte.

Note that it takes a minimum of five I/O port accesses to transmit a byte to the printer use the code above (minimum one IN instruction in the REPEAT..UNTIL loop plus four instructions to send the byte and strobe). If the parallel port is connected to the ISA bus, this means it takes a minimum of five microseconds to transmit a single byte; that works out to less than 200,000 bytes per second. If you are sending ASCII characters to the printer, this is far faster than the printer can print the characters. However, if you are sending a bitmap or a Postscript file to the printer, the printer port bandwidth limitation will become the bottleneck since it takes considerable data to print a page of graphics. For this reason, most graphic printers use a different technique than the above to transmit data to the printer (some parallel ports support DMA in order to get the data transfer rate up to a reasonable level).

7.9 Time-outs on an I/O Port

One problem with the REPEAT..UNTIL loop in the previous section is that it could spin indefinitely waiting for the printer to become ready to accept additional input. If someone turns the printer off or the printer cable becomes disconnected, the program could freeze up, forever waiting for the printer to become available. Usually, it s a good idea to indicate to the user that something has gone wrong rather than simply freezing up the system. A typical way to handle this problem is using a time-out period to determine that something is wrong with the peripheral device.

With most peripheral devices you can expect some sort of response within a reasonable amount of time. For example, most printers will be ready to accept additional character data within a few seconds of the last transmission (worst case). Therefore, if 30 seconds or more have passed since the printer was last willing to accept a character, this is probably an indication that something is wrong. If the program could detect this, then it could ask the user to check the printer and tell the program to resume printing once the problem is resolved.

Choosing a good time-out period is not an easy task. You must carefully balance the irritation of having the program constantly ask you what s wrong when there is nothing wrong with the printer (or other device) with the program locking up for long periods of time when there is something wrong. Both situations are equally annoying to the end user.

Any easy way to create a time-out period is to count the number of times the program loops while waiting for a handshake signal from a peripheral. Consider the following modification to the REPEAT..UNTIL loop of the previous section:

```
mov( $379, dx );
mov( 30_000_000, ecx );
repeat

    dec( ecx );         // Count down to see if the time-out has expired.
    breakif( @z );      // Leave this loop if ecx counted down to zero.

    in( dx, al );
```

```
        test( $80, al );   // Clears Z flag if bit seven is set.

    until( @nz );

       if( ecx = 0 ) then

        // We had a time-out error.

    else

        // Okay to write another byte to the printer data port here.

    endif;
```

The code above will exit once the printer is ready to accept data or when approximately 30 seconds have expired. You may question the 30 second figure. After all, a software based loop (counting down ECX to zero) should run a different speeds on different processors. However, don t miss the fact that there is an IN instruction inside this loop. The IN instruction reads a port on the ISA bus and that means this instruction will take approximately one microsecond to execute (about the fastest operation on the ISA bus). Hence, every one million times through the loop will take about a second (–50%, but close enough for our purposes). This is true regardless of the CPU frequency.

The 80x86 provides a couple of instructions that are quite useful for implementing time-outs in a polling loop: LOOPZ and LOOPNZ. We ll consider the LOOPZ instruction here since it s perfect for the loop above. The LOOPZ instruction decrements the ECX register by one and falls through to the next instruction if ECX contains zero. If ECX does not contain zero, then this instruction checks the zero flag setting prior to decrementing ECX; if the zero flag was set, then LOOPZ transfers control to a label specified as LOOPZ s operand. Consider the implementation of the previous REPEAT..UNTIL loop using LOOPZ:

```
    mov( $379, dx );
    mov( 30_000_000, ecx );
    PollingLoop:

        in( dx, al );
        test( $80, al );   // Clears Z flag if bit seven is set.

    loopz PollingLoop;    // Repeat while zero and ECX<>0.

       if( ecx = 0 ) then

        // We had a time-out error.
```

```
else

    // Okay to write another byte to the printer data port here.

endif;
```

Notice how this code doesn t need to explicitly decrement ECX and check to see if it became zero.

Warning: the LOOPZ instruction can only transfer control to a label with –127 bytes of the LOOPZ instruction. Due to a design problem, HLA cannot detect this problem. If the branch range exceeds 127 bytes HLA will not report an error. Instead, the underlying assembler (e.g., MASM or Gas) will report the error when it assembles HLA s output. Since it s somewhat difficult to track down these problems in the MASM or Gas listing, the best solution is to never use the LOOPZ instruction to jump more than a few instructions in your code. It s perfect for short polling loops like the one above, it s not suitable for branching large distances.

7.10 Interrupts and Polled I/O

Polling is constantly testing a port to see if data is available. That is, the CPU polls (asks) the port if it has data available or if it is capable of accepting data. The REPEAT..UNTIL loop in the previous section is a good example of polling. The CPU continually polls the port to see if the printer is ready to accept data. Polled I/O is inherently inefficient. Consider what happens in the previous section if the printer takes ten seconds to accept another byte of data — the CPU spins in a loop doing nothing (other than testing the printer status port) for those ten seconds.

In early personal computer systems, this is exactly how a program would behave; when it wanted to read a key from the keyboard it would poll the keyboard status port until a key was available. Such computers could not do other operations while waiting for the keyboard.

The solution to this problem is to provide an *interrupt mechanism*. An interrupt is an external hardware event (such as the printer becoming ready to accept another byte) that causes the CPU to interrupt the current instruction sequence and call a special interrupt service routine. (ISR). An interrupt service routine typically saves all the registers and flags (so that it doesn t disturb the computation it interrupts), does whatever operation is necessary to handle the source of the interrupt, it restores the registers and flags, and then it resumes execution of the code it interrupted. In many computer systems (e.g., the PC), many I/O devices generate an interrupt whenever they have data available or are able to accept data from the CPU. The ISR quickly processes the request in the background, allowing some other computation to proceed normally in the foreground.

An interrupt is essentially a procedure call that the hardware makes (rather than explicit call to some procedure, like a call to the *stdout.put* routine). The most important thing to remember about an interrupt is that it can pause the execution of some program at any point between two instructions when an interrupt occurs. Therefore, you typically have no guarantee that one instruction always executes immediately after another in the program because an interrupt could occur between the two instructions. If an interrupt occurs in the middle of the execution of some instruction, then the CPU finishes that instruction before transferring control to the appropriate interrupt service routine. However, the interrupt generally interrupts execution before the start of the next instruction[5]. Suppose, for example, that an interrupt occurs between the execution of the following two instructions:

```
add( i, eax );
```

```
                <---- Interrupt occurs here.
mov( eax, j );
```

When the interrupt occurs, control transfers to the appropriate ISR that handles the hardware event. When that ISR completes and executes the IRET (interrupt return) instruction, control returns back to the point of interruption and execution of the original code continues with the instruction immediately after the point of interrupt (e.g., the MOV instruction above). Imagine an interrupt service routine that executes the following code:

```
mov( 0, eax );
iret;
```

If this ISR executes in response to the interrupt above, then the main program will not produce a correct result. Specifically, the main program should compute "j := eax +i;" Instead, it computes "j := 0;" (in this particular case) because the interrupt service routine sets EAX to zero, wiping out the sum of *i* and the previous value of EAX. This highlights a very important fact about ISRs: **ISRs must preserve all registers and flags whose values they modify**. If an ISR does not preserve some register or flag value, this will definitely affect the correctness of the programs running when an interrupt occurs. Usually, the ISR mechanism itself preserves the flags (e.g., the interrupt pushes the flags onto the stack and the IRET instruction restores those flags). However, the ISR itself is responsible for preserving any registers that it modifies.

Although the preceding discussion makes it clear that ISRs must preserve registers and the flags, your ISRs must exercise similar care when manipulating any other resources the ISR shares with other processes. This includes variables, I/O ports, etc. Note that preserving the values of such objects isn t always the correct solution. Many ISRs communicate their results to the foreground program using shared variables. However, as you will see, the ISR and the foreground program must coordinate access to shared resources or they may produce incorrect results. Writing code that correctly works with shared resources is a difficult challenge; the possibility of subtle bugs creeping into the program is very great. We ll consider some of these issues a little later in this chapter; the messy details will have to wait for a later volume of this text.

CPUs that support interrupts must provide some mechanism that allows the programmer to specify the address of the ISR to execute when an interrupt occurs. Typically, an interrupt vector is a special memory location that contains the address of the ISR to execute when an interrupt occurs. PCs typically support up to 16 different interrupts.

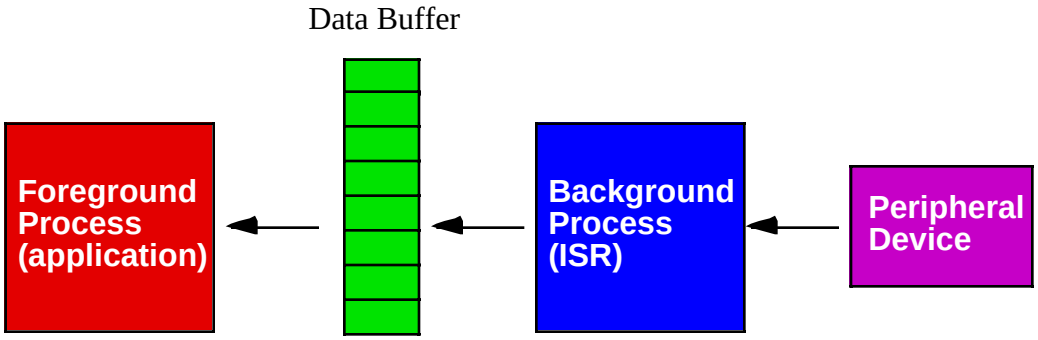
After an ISR completes its operation, it generally returns control to the foreground task with a special return from interrupt instruction. On the Y86 hypothetical processor, for example, the IRET (interrupt return) instruction handles this task. This same instruction does a similar task on the 80x86. An ISR should always end with this instruction so the ISR can return control to the program it interrupted.

7.11 Using a Circular Queue to Buffer Input Data from an ISR

A typical interrupt-driven input system uses the ISR to read data from an input port and buffer it up whenever data becomes available. The foreground program can read that data from the buffer at its leisure without losing any data from the port. A typical foreground/ISR arrangement appears in Figure 7.10. In this diagram the ISR

5. The situation is somewhat fuzzy if you have pipelines and superscalar operation. Exactly what instruction does an interrupt precede if there are multiple instructions executing simultaneously? The answer is somewhat irrelevant, however, since the interrupt does take place between the execution of some pair of instructions; in reality, the interrupt may occur immediately after the last instruction to enter the pipeline when the interrupt occurs. Nevertheless, the system does interrupt the execution of the foreground process after the execution of some instruction.

reads a value from the peripheral device and then stores the data into a common buffer that the ISR shares with the foreground application. Sometime later, the foreground process removes the data from the buffer. If (during a burst of input) the device and ISR produce data faster than the foreground application reads data from the buffer, the ISR will store up multiple unread data values in the buffer. As long as the average consumption rate of the foreground process matches the average production rate of the ISR, and the buffer is large enough to hold bursts of data, there will be no lost data.



The background process produces data (by reading it from the device) and places it in the buffer. The foreground process consumes data by removing it from the buffer.

Figure 7.10 Interrupt Service Routine as a Data Produce/Application as a Data Consumer

If the foreground process in Figure 7.10 consumes data faster than the ISR produces it, sooner or later the buffer will become empty. When this happens the foreground process will have to wait for the background process to produce more data. Typically the foreground process would poll the data buffer (or, in a more advanced system, block execution) until additional data arrives. Then the foreground process can easily extract the new data from the buffer and continue execution.

There is nothing special about the data buffer. It is just a block of contiguous bytes in memory and a few additional pieces of information to maintain the list of data in the buffer. While there are lots of ways to maintain data in a buffer such as this one, probably the most popular technique is to use a *circular buffer*. A typical circular buffer implementation contains three objects: an array that holds the actual data, a pointer to the next available data object in the buffer, and a length value that specifies how many objects are currently in the buffer.

Later in this text you will see how to declare and use arrays. However, in the chapter on Memory Access you saw how to allocate a block of data in the STATIC section (see The Static Sections on page 167) or how to use *malloc* to allocate a block of bytes (see Dynamic Memory Allocation and the Heap Segment on page 187). For our purposes, declaring a block of bytes in the STATIC section is just fine; the following code shows one way to set aside 16 bytes for a buffer:

```
static
    buffer:    byte := 0;                          // Reserves one byte.
               byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0  // 15 additional bytes.
```

Of course, this technique would not be useful if you wanted to set aside storage for a really large buffer, but it works fine for small buffers (like our example above). See the chapter on arrays (appearing later in this text) if you need to allocate storage for a larger buffer.

In addition to the buffer data itself, a circular buffer also needs at least two other values: an index into the buffer that specifies where the next available data object appears and a count of valid items in the buffer. Given that the 80x86's addressing

modes all use 32-bit registers, we'll find it most convenient to use a 32-bit unsigned integer for this purpose even though the index and count values never exceed 16. The declaration for these values might be the following:

```
static
    index: uns32 := 0;  // Start with first element of the array.
    count: uns32 := 0;  // Initially, there is no data in the array.
```

The data producer (the ISR in our example) inserts data into the buffer by following these steps:

- Check the count. If the count is equal to the buffer size, then the buffer is full and some corrective action is necessary.
- Store the new data object at location ((index+count) mod buffer_size).
- Increment the count variable.

Suppose that the producer wishes to add a character to the initially empty buffer. The *count* is zero so we don t have to deal with a buffer overflow. The *index* value is also zero, so ((*index*+*count*) MOD 16) is zero and we store our first data byte at index zero in the array. Finally, we increment *count* by one so that the producer will put the next byte at offset one in the array of bytes.

If the consumer never removes any bytes and the producer keeps producing bytes, sooner or later the buffer will fill up and *count* will hit 16. Any attempt to insert additional data into the buffer is an error condition. The producer needs to decide what to do at that point. Some simple routines may simply ignore any additional data (that is, any additional incoming data from the device will be lost). Some routines may signal an exception and leave it up to the main application to deal with the error. Some other routines may attempt to expand the buffer size to allow additional data in the buffer. The corrective action is application-specific. In our examples we ll assume the program either ignores the extra data or immediately stops the program if a buffer overflow occurs.

You ll notice that the producer stores the data at location ((*index*+*count*) MOD *buffer_size*) in the array. This calculation, as you ll soon see, is how the circular buffer obtains its name. HLA does provide a MOD instruction that will compute the remainder after the division of two values, however, most buffer routines don t compute remainder using the MOD instruction. Instead, most buffer routines rely on a cute little trick to compute this value much more efficiently than with the MOD instruction. The trick is this: if a buffer s size is a power of two (16 in our case), you can compute (*x* MOD *buffer_size*) by logically ANDing *x* with *buffer_size* - 1. In our case, this means that the following instruction sequence computes ((*index*+*count*) MOD 16) in the EBX register:

```
mov( index, ebx );
add( count, ebx );
and( 15, ebx );
```

Remember, this trick only works if the buffer size is an integral power of two. If you look at most programs that use a circular buffer for their data, you ll discover that they commonly use a buffer size that is an integral power of two. The value is not arbitrary; they do this so they can use the AND trick to efficiently compute the remainder.

To remove data from the buffer, the consumer half of the program follows these steps:

- The consumer first checks to the count to see if there is any data in the buffer. If not, the consumer waits until data is available.
- If (or when) data is available, the consumer fetches the value at the location *index* specifies within the buffer.
- The consumer then decrements the *count* and computes *index* := (*index* + 1) MOD *buffer_size*.

To remove a byte from the circular buffer in our current example, you d use code like the following:

```
// wait for data to appear in the buffer.

repeat
until( count <> 0 );

// Remove the character from the buffer.

mov( index, ebx );
mov( buffer[ ebx ], al );  // Fetch the byte from the buffer.
dec( count );              // Note that we've removed a character.
inc( ebx );                // Index := Index + 1;
and( 15, ebx );            // Index := (index + 1) mod 16;
mov( ebx, index );         // Save away the new index value.
```

As the consumer removes data from the circular queue, it advances the index into the array. If you re wondering what happens at the end of the array, well that s the purpose of the MOD calculation. If *index* starts at zero and increments with each character, you d expect the sequence 0, 1, 2, ... At some point or another the *index* will exceed the bounds of the buffer (i.e., when *index* increments to 16). However, the MOD operation resets this value back to zero (since 16 MOD 16 is zero). Therefore, the consumer, after that point, will begin removing data from the beginning of the buffer.

Take a close look at the REPEAT..UNTIL loop in the previous code. At first blush you may be tempted to think that this is an infinite loop if *count* initially contains zero. After all, there is no code in the body of the loop that modifies *count s* value. So if *count* contains zero upon initial entry, how does it ever change? Well, that s the job of the ISR. When an interrupt comes along the ISR suspends the execution of this loop at some arbitrary point. Then the ISR reads a byte from the device, puts the byte into the buffer, and updates the *count* variable (from zero to one). Then the ISR returns and the consumer code above resumes where it left off. On the next loop iteration, however, *count s* value is no longer zero, so the loop falls through to the following code. This is a classic example of how an ISR communicates with a foreground process — by writing a value to some shared variable.

There is a subtle problem with the producer/consumer code in this section. It will fail if the producer is attempting to insert data into the buffer at exactly the same time the consumer is removing data. Consider the following sequence of instructions:

```
// wait for data to appear in the buffer.

repeat
until( count <> 0 );

// Remove the character from the buffer.

mov( index, ebx );
mov( buffer[ ebx ], al );  // Fetch the byte from the buffer.
dec( count );              // Note that we've removed a character.

   *** Assume the interrupt occurs here, so we begin executing
   *** the data insertion sequence:

   mov( index, ebx );
   add( count, ebx );
   and( 15, ebx );
   mov( al, buffer[ebx] );
   inc( count );
```

```
   *** now the ISR returns to the consumer code (assume we've preserved EBX):


   inc( ebx );                // Index := Index + 1;
   and( 15, ebx );            // Index := (index + 1) mod 16;
   mov( ebx, index );         // Save away the new index value.
```

The problem with this code, which is very subtle, is that the consumer has decremented the count variable and an interrupt occurs before the consumer can update the index variable as well. Therefore, upon arrival into the ISR, the *count* value and the *index* value are inconsistent. That is, *index+count* now points at the last value placed in the buffer rather than the next available location. Therefore, the ISR will overwrite the last byte in the buffer rather than properly placing this byte after the (current) last byte. Worse, once the ISR returns to the consumer code, the consumer will update the *index* value and effectively add a byte of garbage to the end of the circular buffer. The end result is that we wipe out the next to last value in the buffer and add a garbage byte to the end of the buffer.

Note that this problem doesn t occur all the time, or even frequently for that matter. In fact, it only occurs in the very special case where the interrupt occurs between the "dec(count);" and "mov(ebx, index);" instructions in this code. If this code executes a very tiny percentage of the time, the likelihood of encountering this error is quite small. This may seem good, but this is actually worse than having the problem occur all the time; the fact that the problem rarely occurs just means that it s going to be really hard to find and correct this problem when you finally do detect that something has gone wrong. ISRs and concurrent programs are among the most difficult programs in the world to test and debug. The best solution is to carefully consider the interaction between foreground and background tasks when writing ISRs and other concurrent programs. In a later volume, this text will consider the issues in concurrent programming, for now, be very careful about using shared objects in an ISR.

There are two ways to correct the problem that occurs in this example. One way is to use a pair of (somewhat) independent variables to manipulate the queue. The original PC s type ahead keyboard buffer, for example, used two index variables rather than an index and a count to maintain the queue. The ISR would use one index to insert data and the foreground process would use the second index to remove data from the buffer. The only sharing of the two pointers was a comparison for equality, which worked okay even in an interrupt environment. Here s how the code worked:

```
// Declarations

static
   buffer: byte := 0; byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
   Ins: uns32 := 0;  // Insert bytes starting here.
   Rmv: uns32 := 0;  // Remove bytes starting here.

// Insert a byte into the queue (the ISR ):

   mov( Ins, ebx );
   inc( ebx );
   and( 15, ebx );
   if( ebx <> Rmv ) then

      mov( al, buffer[ ebx ] );
      mov( ebx, Ins );

   else

      // Buffer overflow error.
      // Note that we don't update INS in this case.
```

```
   endif;


// Remove a byte from the queue (the consumer process).

   mov( Rmv, ebx );
   repeat

      // Wait for data to arrive

   until( ebx <> Ins );
   mov( buffer[ ebx ], al );
   inc( ebx );
   and( 15, ebx );
   mov( ebx, Rmv );
```

If you study this code (very) carefully, you ll discover that the two code sequences don t interfere with one another. The difference between this code and the previous code is that the foreground and background processes don t write to a (control) variable that the other routine uses. The ISR only writes to *Ins* while the foreground process only writes to *Rmv*. In general, this is not a sufficient guarantee that the two code sequences won t interfere with one another, but it does work in this instance.

One drawback to this code is that it doesn t fully utilize the buffer. Specifically, this code sequence can only hold 15 characters in the buffer; one byte must go unused because this code determines that the buffer is full when the value of *Ins* is one less than *Rmv* (MOD 16). When the two indices are equal, the buffer is empty. Since we need to test for both these conditions, we can t use one of the bytes in the buffer.

A second solution, that many people prefer, is to protect that section of code in the foreground process that could fail if an interrupt comes along. There are lots of ways to protect this *critical section*[6] of code. Alas, most of the mechanisms are beyond the scope of this chapter and will have to wait for a later volume in this text. However, one simple way to protect a critical section is to simply disable interrupts during the execution of that code. The 80x86 family provides two instructions, CLI and STI that let you enable and disable interrupts. The CLI instruction (clear interrupt enable flag) disables interrupts by clearing the "I" bit in the flags register (this is the interrupt enable flag). Similarly, the STI instruction enables interrupts by setting this flag. These two instructions use the following syntax:

```
cli();   // Disables interrupts from this point forward...
 .
 .
 .
sti();   // Enables interrupts from this point forward...
```

You can surround a critical section in your program with these two instructions to protect that section from interrupts. The original consumer code could be safely written as follows:

```
// wait for data to appear in the buffer.

repeat
until( count <> 0 );

// Remove the character from the buffer.
```

6. A critical section is a region of code during which certain resources have to be protected from other processes. For example, the consumer code that fetches data from the buffer needs to be protected from the ISR.

```
   cli();                       // Protect the following critical section.
   mov( index, ebx );
   mov( buffer[ ebx ], al );    // Fetch the byte from the buffer.
   dec( count );                // Note that we've removed a character.
   inc( ebx );                  // Index := Index + 1;
   and( 15, ebx );              // Index := (index + 1) mod 16;
   mov( ebx, index );           // Save away the new index value.
   sti();                       // Critical section is done, restore interrupts.
```

Perhaps a better sequence to use is to push the EFLAGs register (that contains the I flag) and turn off the interrupts. Then, rather than blindly turning interrupts back on, you can restore the original I flag setting using a POPFD instruction:

```
   // Remove the character from the buffer.

   pushfd();                    // Preserve current I flag value.
   cli();                       // Protect the following critical section.
   mov( index, ebx );
   mov( buffer[ ebx ], al );    // Fetch the byte from the buffer.
   dec( count );                // Note that we've removed a character.
   inc( ebx );                  // Index := Index + 1;
   and( 15, ebx );              // Index := (index + 1) mod 16;
   mov( ebx, index );           // Save away the new index value.
   popfd();                     // Restore original I flag value
```

This mechanism is arguably safer since it doesn t turn the interrupts on even if they were already off before executing this sequence.

In our simple example (with a single producer and a single consumer) there is no need to protect the code in the ISR. However, if it were possible for two different ISRs to insert data into the buffer, and one ISR could interrupt another, then you would have to protect the code inside the ISR as well.

You must be very careful about turning the interrupts on and off. If you turn the interrupts off and forget to turn them back on, the next time you enter a loop like one of the REPEAT..UNTIL loops in this section the program will lock up because the loop control variable (*count*) will never change if an ISR cannot execute and update its value. This situation is called *deadlock* and you must take special care to avoid it.

Note that applications under Windows or Linux cannot change the state of the interrupt disable flag. This technique is useful mainly in embedded system or under simpler operating systems like DOS. Fortunately, advanced 32-bit operating systems like Linux and Windows provide other mechanisms for protecting critical sections.

7.12 Using a Circular Queue to Buffer Output Data for an ISR

You can also use a circular buffer to hold data waiting for transmission. For example, a program can buffer up data in bursts while an output device is busy and then the output device can empty the buffer at a steady state. The queuing and dequeuing routines are very similar to those found in the previous section with one major difference: output devices don t automatically initiate the transfer like input devices. This problem is a source of many bugs in output buffering routines and one that we ll pay careful attention to in this section.

As noted above, one advantage of an input device is that it automatically interrupts the system whenever data is available. This activates the corresponding ISR that reads the data from the device and places the data in the buffer. No special processing is necessary to prepare the interrupt system for the next input value.

There is a subtle difference between the interrupts an input device generates and the interrupts an output device generates. An input device generates an interrupt when data is available, output devices generate an interrupt when they are ready to accept more data. For example, a keyboard device generates an interrupt when the user presses a key and the system has to read the character from the keyboard. A printer device, on the other hand, generates an interrupt once it is done with the last character transmitted to it and it s ready to accept another character. Whenever the user presses a keyboard for the very first time, the system will generate an interrupt in response to that event. However, the printer does not generate an interrupt when the system first powers up to tell the system that it s ready to accept a character. Even if it did, the system would ignore the interrupt since it (probably) doesn t have any data to transmit to the printer at that point. Later, when the system puts data in the printer s buffer for transmission, there is no interrupt that activates the ISR to remove a character from the buffer and send it to the printer. The printer device only sends interrupts when it is done processing a character; if it isn t processing any characters, it won t generate any interrupts.

This creates a bit of a problem. If the foreground process places characters in the queue and the background process (the ISR, which is the consumer in this case) only removes those characters when an interrupt occurs, the system will never activate the ISR since the device isn t currently processing anything. To correct this problem, the producer code (the foreground process) must maintain a flag that indicates whether the output device is currently processing a character; if so, then the producer can simply queue up the character in the buffer. If the device is not currently processing any data, then the producer should send the data directly to the device rather than queue up the data in the buffer[7]. Old time programmers refer to this as "priming the pump" since we have to put data in the transmission pipeline in order to get the process working properly.

Once the producer "primes the pump" the process continues automatically as long as there is data in the buffer. After the output device processes the current byte it generates an interrupt. The ISR removes a byte from the buffer and transmits this data to the device. When that byte completes transmission the device generates another interrupt and the process repeats. This process repeats automatically as long as there is data in the buffer to transmit to the output device.

When the ISR transmits the last character from the buffer, the output device still generates an interrupt at the end of the transmission. The ISR, upon noting that the buffer is empty, returns without sending any new data to the output device. Since there is no pending data transmission to the output device, there will be no new interrupts to activate the ISR when new data appears in the buffer. Once again the foreground process (producer) will have to prime the pump to get the process going when it attempts to put data in the buffer.

Perhaps the easiest way to handle this process is to use a boolean variable to indicate whether the output device is currently transmitting data (and will generate an interrupt to process the next byte). If the flag is set, the foreground process can simply enqueue the data; if the flag is clear, the foreground process must transmit the data directly to the device (or call the code that does this). In this latter case, the foreground process must also set the flag to denote a transmission in progress.

Here is some code that can implement this functionality:

```
static
   OutBuf: byte := 0; byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
   Index: uns32 := 0;
   Count: uns32 := 0;
   Xmitting: boolean := false;  // Flag to denote transmission in progress.
    .
    .
    .
   // Code to enqueue a byte (foreground process executes this)
```

7. Another possibility is to go ahead and queue up the data and then manually activate the code that dequeues the data and sends it to the output device.

```
if( Count = 16 ) then

   // Error, buffer is full.  Do whatever processing is necessary to
   // deal with this problem.
      .
      .
      .

elseif( Xmitting ) then

   // If we're currently transmitting data, just add the byte to the queue.

   pushfd();          // Critical region! Turn off the interrupts.
   cli();
   mov( Index, ebx ); // Store the new byte at address (Index+Count) mod 16
   add( Count, ebx );
   and( 15, ebx );
   mov( al, OutBuf[ ebx ] );
   inc( Count );
   popfd();           // Restore the interrupt flag's value.

else

   // The buffer is empty and there is no character in transmission.
   // Do whatever is necessary to transmit the character to the output
   // device.
      .
      .
      .

   // Be sure to set the Xmitting flag since a character is now being
   // transmitted to the output device.

   mov( true, Xmitting );

endif;
   .
   .
   .
// Here's the code that would appear inside the ISR to remove a character
// from the buffer and send it to the output device.  The system calls this
// ISR whenever the device finishes processing some character.
// (Presumably, the ISR preserves all registers this code sequence modifies)

if( Count > 0 ) then

   // Okay, there are characters in the buffer.  Remove the first one
   // and transmit it to the device:

   mov( Index, ebx );
   mov( OutBuf[ ebx ], al );   // Get the next character to output
   inc( ebx );                 // Point Index at the next byte in the
   and( 15, ebx );             // circular buffer.
   mov( ebx, Index );
   dec( Count );               // Decrement count since we removed a char.
```

```
    << At this point, do whatever needs to be done in order to
       transmit a character to the output device >>
       .
       .
       .

else

    // At this point, the ISR was called but the buffer is empty.
    // Simply clear the Xmitting flag and return.  (this will force the
    // next buffer insertion operation to transmit the data directly to the
    // device.)

    mov( true, Xmitting );

endif;
```

7.13 I/O and the Cache

It goes without saying that the CPU cannot cache values for memory-mapped I/O ports. If a port is an input port, caching the data from that port would always return the first value read; subsequent reads would read the value in the cache rather than the possible (volatile) data at the input port. Similarly, with a write-back cache mechanism, writes to an output port may never reach that port (i.e., the CPU may save up several writes in the cache and send the last such write to the actual I/O port). Therefore, there must be some mechanism to tell the CPU not to cache up accesses to certain memory locations.

The solution is in the virtual memory subsystem of the 80x86. The 80x86 s page table entries contain information that the CPU can use to determine whether it is okay to map data from a page in memory to cache. If this flag is set one way, then the cache operates normally; if the flag is set the other way, then the CPU does not cache up accesses to that page.

Unfortunately, the granularity (that is, the minimum size) of this access is the 4K page. So if you need to map 16 device registers into memory somewhere and cannot cache them, you must actually consume 4K of the address space to hold these 16 locations. Fortunately, there is a lot of room in the 4 GByte virtual address space and there aren t that many peripheral devices that need to be mapped into the memory address space. So assigning these device addresses sparsely in the memory map will not present too many problems.

7.14 Protected Mode Operation

Windows and Linux employ the 80x86's protected mode of operation. In this mode of operation, direct access to devices is restricted to the operating system and certain privileged programs. Standard applications, even those written in assembly language, are not so privileged. If you write a simple program that attempts to send data to an I/O port via an IN or an OUT instruction, the system will generate an illegal access exception and halt your program. Unless you're willing to write a device driver for your operating system, you'll probably not be able to access the I/O devices directly.

Like Windows, Linux does not allow an arbitrary application program to access I/O ports as it pleases. Only programs with "super-user" (root) priviledges may do so. For limited I/O access, it is possible to use the Linux IOPERM system call to make certain I/O ports accessible from user applications (note that only a process with super-user priviledges may call IOPERM, but that program may then invoke a standard user application and the application it runs will have access to the specified ports). For more details, Linux users should read the "man" page on "ioperm".

This chapter has provided an introduction to I/O in a very general, architectural sense. It hasn't spent too much time discussing the particular peripheral devices present in a typical PC. This is an intended omission; there is no need to confuse readers with information they can't use. Furthermore, as manufacturers introduce new PCs they are removing many of the common peripherals like parallel and serial ports that are relatively easy to program in assembly language. They are replacing these devices with complex peripherals like USB and Firewire. Unfortunately, programming these newer peripheral devices is well beyond the scope of this text (Microsoft's USB code, for example, is well over 100 pages of C++ code).

Those who are interested in additional information about programming standard PC peripherals may want to consult one of the many excellent hardware references available for the PC or take a look at the DOS/16-bit version of this text.

IN and OUT aren't the only instructions that you cannot execute in an application running under protected mode. The system considers many instructions to be "privileged" and will abort your program if you attempt to use these instructions. The CLI and STI instructions are good examples. If you attempt to execute either of these instructions, the system will stop your program.

Some instructions will execute in an application, but behave differently than they do when the operating system executes them. The PUSHFD and POPFD instructions are good examples. These instruction push and pop the interrupt enable flag (among others). Therefore, you could use PUSHFD to push the flags on the stack, pop this double word off the stack and clear the bit associated with the interrupt flag, push the value back onto the stack and then use POPFD to restore the flags (and, in the process, clear the interrupt flag). This would seem like a sneaky way around clearing the interrupt flag. The CPU must allow applications to push and pop the flags for other reasons. However, for various security reasons the CPU cannot allow applications to manipulate the interrupt disable flag. Therefore, the POPFD instruction behaves a little differently in an application that it does when the operating system executes it. In an application, the CPU ignores the interrupt flag bit it pops off the stack. In operating system ("kernel") mode, popping the flags register does restore the interrupt flag.

7.15 Device Drivers

If Linux and Windows don t allow direct access to peripheral devices, how does a program communicate with these devices? Clearly this can be done since applications interact with real-world devices all the time. If you reread the previous section carefully, you ll note that it doesn t claim that programs can t access the devices, it only states that user application programs are denied such access. Specially written modules, known as device drivers, are able to access I/O ports by special permission from the operating system. Writing device drivers is well beyond the scope of this chapter (though it will make an excellent subject for a later volume in this text). Nevertheless, an understanding of how device drivers work may help you understand the possibilities and limitations of I//O under a "protected mode" operating system.

A device driver is a special type of program that connects to the operating system. The device driver must follow some special protocols and it must make some special calls to the operating system that are not available to standard applications. Further, in order to install a device driver in your system you must have administrator privileges (device drivers create all kinds of security and resource allocation problems; you can t have every hacker in the world taking advantage of rogue device drivers running on your system). Therefore, "whipping out a device driver" is not a trivial process and application programs cannot load and unload arbitrary drivers at will.

Fortunately, there are only a limited number of devices you d typically find on a PC, therefore you only need a limited number of device drivers. You would typically install a device driver in the operating system the same time you install the device (or when you install the operating system if the device is built into the PC). About the only time you d really need to write your own device driver is when you build your own device or in some special instance when you need to take advantage of some devices capabilities that the standard device drivers don t allow for.

One big advantage to the device driver mechanism is that the operating system (or device vendors) must provide a reasonable set of device drivers or the system will never become popular (one of the reasons Microsoft and IBM s OS/2 operating system was never successful was the dearth of device drivers). This means that applications can easily manipulate lots of devices without the application programmer having to know much about the device itself; the real work has been taken care of by the operating system.

The device driver model does have a few drawbacks, however. The device driver model is great for low-speed devices, where the OS and device driver can respond to the device much more quickly than the device requires. The device driver model is also great for medium and high-speed devices where the system transmits large blocks of data in one direction at a time; in such a situation the application can pass a large block of data to the operating system and the OS can transmit this data to the device (or conversely, read a large block of data from the device and place it in an application-supplied buffer). One problem with the device driver model is that it does not support medium and high-speed data transfers that require a high degree of interaction between the device and the application.

The problem is that calling the operating system is an expensive process. Whenever an application makes a call to the OS to transmit data to the device it could actually take hundreds of microseconds, if not milliseconds, before the device driver actually sees the data. If the interaction between the device and the application requires a constant flurry of bytes moving back and forth, there will be a big delay if each transfer has to go through the operating system. For such applications you will need to write a special device driver to handle the transactions directly in the driver rather than continually returning to the application.

7.16 Putting It All Together

Although the CPU is where all the computation takes place in a computer system, that computation would be for naught if there was no way to get information into and out of the computer system. This is the responsibility of the I/O subsystem. I/O at the machine level is considerably different than the interface high level languages and I/O subroutine libraries (like *stdout.put*) provide. At the machine level, I/O transfers consist of moving bytes (or other data units) between the CPU and device registers or memory.

The 80x86 family supports two types of *programmed I/O*: memory-mapped input/output and I/O-mapped I/O. PCs also provide a third form of I/O that is mostly independent of the CPU: direct memory access or DMA. Memory-mapped input/output uses standard instructions that access memory to move data between the system and the peripheral devices. I/O-mapped input/output uses special instructions, IN and OUT, to move data between the CPU and peripheral devices. I/O-mapped devices have the advantage that they do not consume memory addresses normally intended for system memory. However, the only access to devices using this scheme is through the IN and OUT instructions; you cannot use arbitrary instructions that manipulate memory to control such peripherals. Devices that use DMA have special hardware that let them transmit data to and from system memory without going through the CPU. Devices that use DMA tend to be very high performance, but this I/O mechanism is really only useful for devices that transmit large blocks of data at high speeds.

I/O devices have many different operating speeds. Some devices are far slower than the CPU while other devices can actually produce or consume data faster than the CPU. For devices that are slower than the CPU, some sort of handshaking mechanism is necessary in order to coordinate the data transfer between the CPU and the device. High-speed devices require a DMA controller or buffering since the CPU cannot handle the data rates of these devices. In all cases, some mechanism is necessary to tell the CPU that the I/O operation is complete so the CPU can go about other business.

In modern 32-bit operating systems like Windows and Linux, applications programs do not have direct access to the peripheral devices. The operating system coordinates all I/O via the use of device drivers. The good thing about device drivers is that you (usually) don't have to write them – the operating system provides them for you. The bad thing about writing device drivers is if you have to write one, they are very complex. A later volume in this text may discuss how to do this.

Because HLA programs usually run as applications under the OS, you will not be able to use most of the coding techniques this chapter discusses within your HLA applications. Nevertheless, understanding how device I/O works can help you write better applications. Of course, if you ever have to write a device driver for some device, then the basic knowledge this chapter presents is a good foundation for learning how to write such code.