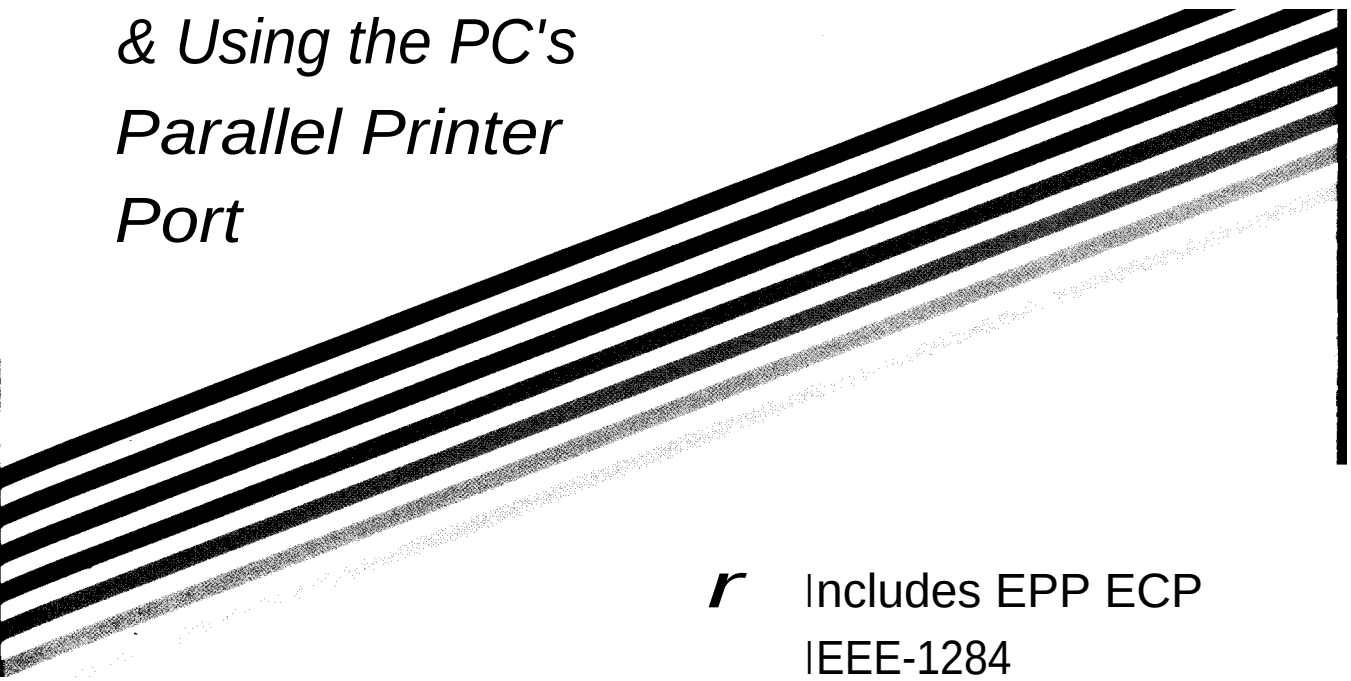
Parallel Port Complete

Programming, Interfacing, & Using the PC's Parallel Printer Port



- Includes EPP ECP IEEE-1284
- Source code in Visual Basic
- User tips

INCLUDES DISK

Jan Axelson

Table of Contents

15 E

Introduction

From its origin as a simple printer interface, the personal computer's parallel port has evolved into a place to plug in just about anything you might want to hook to a computer. The parallel port is popular because it's versatile-you can use it for output, input, or bidirectional links-and because it's available-every PC has one.

Printers are still the most common devices connected to the port, but other popular options include external tape and disk drives and scanners. Laptop computers may use a parallel-port-based network interface or joystick. For special applications, there are dozens of parallel-port devices for use in data collection, testing, and control systems. And the parallel port is the interface of choice for many one-of-a-kind and small-scale projects that require communications between a computer and an external device.

In spite of its popularity, the parallel port has always been a bit of a challenge to work with. Over the years, several variations on the original port's design have emerged, yet there has been no single source of documentation that describes the port in its many variations.

I wrote this book to serve as a practical, hands-on guide to all aspects of the parallel port. It covers both hardware and software, including how to design external

circuits that connect to the port, as well as how to write programs to control and monitor the port, including both the original and improved port designs.

Who should read this book?

The book is designed to serve readers with a variety of backgrounds and interests:

Programmers will find code examples that show how to use the port in all of its modes. If you program in Visual Basic, you can use the routines directly in your programs.

For hardware designers, there are details about the port circuits and how to interface them to the world outside the PC. I cover the port's original design and the many variations and improvements that have evolved. Examples show how to design circuits for reliable data transfers.

System troubleshooters can use the programming techniques and examples for finding and testing ports on a system.

Experimenters will find dozens of circuit and code examples, along with explanations and tips for modifying the examples for a particular application.

Teachers and students have found the parallel port to be a handy tool for experiments with electronics and computer control. Many of the examples in this book are suitable as school projects.

And last but not least, users, or anyone who uses a computer with printers or other devices that connect to the parallel port, will find useful information, including advice on configuring ports, how to add a port, and information on cables, port extenders, and switch boxes.

What's Inside

This book focuses on several areas related to the parallel port:

Using the New Modes

Some of the most frequently asked parallel-port questions relate to using, programming, and interfacing the **port** in the new, advanced modes, including the enhanced parallel **port** (EPP), the extended capabilities **port** (ECP), and the PS/2-type, or simple bidirectional, **port.** This book covers each of these. Examples show how to enable a mode, how to use the mode to transfer data, and how to use software negotiation to enable a PC and peripheral to select the best mode available.

About the Program Code

Every programmer has a favorite language. The choices include various implementations of Basic, CIC++, and Pascal/Delphi, and assembly language.

For the program examples in this book, I wanted to use a popular language so as many readers as possible could use the examples directly, and this prompted my decision to use Microsoft's Visual Basic for Windows. A big reason for Visual Basic's popularity is that the programming environment makes it extremely easy to add controls and displays that enable users to control a program and view the results.

However, this book isn't a tutorial on Visual Basic. It assumes you have a basic understanding of the language and how to create and debug a Visual-Basic program.

I developed the examples originally using Visual Basic Version 3, then ported them to Version 4. As much as possible, the programs are designed to be compatible with both versions, including both 16- and 32-bit Version-4 programs. The companion disk includes two versions of each program, one for Version 3 and one for 16- and 32-bit Version 4 programs.

One reason I decided to maintain compatibility with Version 3 is that the standard edition of Version 4 creates 32-bit programs only. Because Windows 3.1 can't run these programs, many users haven't upgraded to Version 4. Also, many parallel-port programs run on older systems that are put to use as dedicated controllers or data loggers. Running the latest version of Windows isn't practical or necessary on these computers.

Of course, in the software world, nothing stays the same for long. Hopefully, the program code will remain 'compatible in most respects with later versions of Visual Basic.

Compatibility with Version 3 does involve some tradeoffs. For example, Version 3 doesn't support the Byte variable type, so my examples use Integer variables even where Byte variables would be appropriate (as in reading and writing to a byte-wide port). In a few areas, such as some Windows API calls, I've provided two versions, one for use with 16-bit programs, Version 3 or 4, and the other for use with Version 4 programs, 16- or 32-bit.

In the program listings printed in this book, I use Visual Basic 4's line-continuation character (_) to extend program lines that don't fit on one line on the page. In other words, this:

```
PortType =
Left$(ReturnBuffer, NumberOfCharacters)
```

is the same as this:

```
PortType = Left$(ReturnBuffer, NumberOfCharacters)
```

To remain compatible with Version 3, the code on the disk doesn't use this feature.

Most of the program examples are based on a general-purpose Visual-Basic form and routines introduced early in the book. The listings for the examples in each chapter include only the application-specific code added to the listings presented earlier. The routines within a listing are arranged alphabetically, in the same order that Visual Basic displays and prints them.

Of course, the concepts behind the programs can be programmed with any language and for any operating system. In spite of Windows' popularity, MS-DOS programs still have uses; especially for the type of control and monitoring programs that often use the parallel port. Throughout, I've tried to document the code completely enough so that you can translate it easily into whatever programming language and operating system you prefer.

Several of the examples include a parallel-port interface to a microcontroller circuit. The companion disk has the listings for the microcontroller programs.

About the Example Circuits

This book includes schematic diagrams of circuits that you can use or adapt in parallel-port projects. In designing the examples, I looked for circuits that are as easy as possible to put together and program. All use inexpensive, off-the-shelf components that are available from many sources.

The circuit diagrams are complete, with these exceptions:

- Power-supply and ground pins are omitted when they are in standard locations on the package (bottom left for ground, top right for power, assuming pin 1 is top left).
- Power-supply decoupling capacitors are omitted. (This book explains when and how to add these to your circuits.)
- Some chips may have additional, unused gates or other elements that aren't shown.

The manufacturers' data sheets have additional information on the components.

Conventions

These are the typographic conventions used in this book:

Item	Convention	Example
Signal name	italics	*Busy, DO*
Active-low signal	leading n	*nAck nStrobe*
Signal complement	overbar	CO, S7 (equivalent to -CO, -S7 or /CO, /S7)
Program code	monospace font	`DoEvents, End Sub`
File name	italics	*win.ini, inpout16.d11*
Hexadecimal number	trailing h	3BCh (same as &h3BC in Visual Basic)

Corrections and Updates

In researching and putting together this book, I've done my best to ensure that the information is complete and correct. I built and tested every circuit and tested all of the program code, most of it multiple times. But I know from experience that on the way from test to publication, errors and omissions do occur.

Any corrections or updates to this book will be available at Lakeview Research's World Wide Web site on the Internet at *http://Www.Ivr.com.* This is also the place to come for links to other parallel-port information on the Web, including data sheets for parallel-port controllers and software tools for parallel-port programming.

Thanks!

Finally, I want to say thanks to everyone who helped make this book possible. I credit the readers of my articles in *The Microcomputer Journal* for first turning me on to this topic with their questions, comments, and article requests. The series I wrote for the magazine in 1994 was the beginning of this book.

Others deserving thanks are product vendors, who answered many questions, and the Usenet participants who asked some thought-provoking questions that often sent me off exploring areas I wouldn't have thought of otherwise.

Special thanks to SoftCircuits (PO Box 16262, Irvine, CA 92713, Compuserve 72134,263, WWW: http://www.softcircuits.com) for the use of Vbasm.

Essentials

A first step in exploring the parallel port is learning how to get the most from a port with your everyday applications and peripherals. Things to know include how to find, configure, and install a port, how and when to use the new bidirectional, EPP, and ECP modes, and how to handle a system with multiple parallel-port peripherals. This chapter presents essential information and tips relating to these topics.

Defining the Port

What is the "parallel port"? In the computer world, a port is a set of signal lines that the microprocessor, or CPU, uses to exchange data with other components. Typical uses for ports are communicating with printers, modems, keyboards, and displays, or just about any component or device except system memory. Most computer ports are digital, where each signal, or bit, is 0 or 1. A parallel port transfers multiple bits at once, while a serial port transfers a bit at a time (though it may transfer in both directions at once).

This book is about a specific type of parallel port: the one found on just about every PC, or IBM-compatible personal computer. Along with the RS-232 serial port, the parallel port is a workhorse of PC communications. On newer PCs, you

may find other ports such as SCSI, USB, and IrDA, but the parallel port remains popular because it's capable, flexible, and every PC has one.

The term *PC-compatible,* or PC for short, refers to the IBM PC and any of the many, many personal computers derived from it. From another angle, a PC is any computer that can run Microsoft's MS-DOS operating system and whose expansion bus is compatible with the ISA bus in the original IBM PC. The category includes the PC, XT, AT, PS/2, and most computers with 80x86, Pentium, and compatible CPUs. It does not include the Macintosh, Am iga, or IBM mainframes, though these and other computer types may have ports that are similar to the parallel port on the PC.

The original PC's parallel port had eight outputs, five inputs, and four bidirectional lines. These are enough for communicating with many types of peripherals. On many newer PCs, the eight outputs can also serve as inputs, for faster communications with scanners, drives, and other devices that send data to the PC.

The parallel port was designed as a printer port, and many of the original names for the port's signals *(PaperEnd, AutoLineFeed)* reflect that use. But these days, you can find all kinds of things besides printers connected to the port. The term *peripheral,* or *peripheral device* is a catch-all category that includes printers, scanners, modems, and other devices that connect to a PC.

Port Types

As the design of the PC evolved, several manufacturers introduced improved versions of the parallel port. The new port types are compatible with the original design, but add new abilities, mainly for increased speed.

Speed is important because as computers and peripherals have gotten faster, the jobs they do have become more complicated, and the amount of information they need to exchange has increased. The original parallel port was plenty fast enough for sending bytes representing ASCII text characters to a dot-matrix or daisy-wheel printer. But modern printers need to receive much more information to print a page with multiple fonts and detailed graphics, often in color. The faster the computer can transmit the information, the faster the printer can begin processing and printing the result.

A fast interface also makes it feasible to use portable, external versions of peripherals that you would otherwise have to install inside the computer. A parallel-port tape or disk drive is easy to move from system to system, and for occasional use, such as making back-ups, you can use one unit for several systems. Because a backup may involve copying hundreds of Megabytes, the interface has to be fast to be worthwhile.

This book covers the new port types in detail, but for now, here is a summary of the available types:

Original (SPP)

The parallel port in the original IBM PC, and any port that emulates the original port's design, is sometimes called the SPP, for standard parallel port, even though the original **port** had no written standard beyond the schematic diagrams and documentation for the IBM PC. Other names used are AT-type or *ISA-compatible.*

The **port** in the original PC was based on an existing Centronics printer interface. However, the PC introduced a few differences, which other systems have continued.

SPPs can transfer eight bits at once to a peripheral, using a protocol similar to that used by the original Centronics interface. The SPP doesn't have a byte-wide input **port,** but for PC-to-peripheral transfers, SPPs can use a Nibble mode that transfers each byte 4 bits at a time. Nibble mode is slow, but has become popular as a way to use the parallel **port** for input.

PS/2-type (Simple Bidirectional)

An early improvement to the parallel port was the bidirectional data port introduced on IBM's model PS/2. The bidirectional port enables a peripheral to transfer eight bits at once to a PC. The term PS/2-type has come to refer to any parallel port that has a bidirectional data port but doesn't support the EPP or ECP modes described below. Byte mode is an 8-bit data-transfer protocol that PS/2-type ports can use to transfer data from the peripheral to the PC.

EPP

The EPP (enhanced parallel port) was originally developed by chip maker Intel, PC manufacturer Zenith, and Xircom, a maker of parallel-port networking products. As on the PS/2-type port, the data lines are bidirectional. An EPP can read or write a byte of data in one cycle of the ISA expansion bus, or about 1 microsecond, including handshaking, compared to four cycles for an SPP or PS/2-type port. An EPP can switch directions quickly, so it's very efficient when used with disk and tape drives and other devices that transfer data in both directions. An EPP can also emulate an SPP, and some EPPs can emulate a PS/2-type port.

ECP

The ECP (extended capabilities port) was first proposed by Hewlett Packard and Microsoft. Like the EPP, the ECP is bidirectional and can transfer data at ISA-bus speeds. ECPs have buffers and support for DMA (direct memory access) transfers

and data compression. ECP transfers are useful for printers, scanners, and other peripherals that transfer large blocks of data. An ECP can also emulate an SPP or PS/2-type port, and many ECPs can emulate an EPP as well.

Multi-mode Ports

Many newer ports are multi-mode ports that can emulate some or all of the above types. They often include configuration options that can make all of the port types available, or allow certain modes while locking out the others.

System Resources

The parallel port uses a variety of the computer's resources. Every port uses a range of addresses, though the number and location of addresses varies. Many ports have an assigned IRQ (interrupt request) level, and ECPs may have an assigned DMA channel. The resources assigned to a port can't conflict with those used by other system components, including other parallel ports

Addressing

The standard parallel port uses three contiguous addresses, usually in one of these ranges:

```
3BCh, 3BDh, 3BEh
378h, 379h, 37Ah
278h, 279h, 27Ah
```

The first address in the range is the port's base address, also called the Data register or just the port address. The second address is the port's Status register, and the third is the Control register. (See Appendix C for a review of hexadecimal numbers.)

EPPs and ECPs reserve additional addresses for each port. An EPP adds five registers at *base address + 3* through *base address + 7,* and an ECP adds three registers at *base address + 400h* through *base address + 402h.* For a base address of 378h, the EPP registers are at 37Bh through 37Fh, and the ECP registers are at 778h through 77Fh.

On early PCs, the parallel port had a base address of 3BCh. On newer systems, the parallel port is most often at 378h. But all three addresses are reserved for parallel ports, and if the port's hardware allows it, you can configure a port at any of the addresses. However, you normally can't have an EPP at base address 3BCh, because the added EPP registers at this address may be used by the video display.

IBM's Type 3 PS/2 port also had three additional registers, at *base address +3* through *base address + 5,* and allowed a base address of 1278h or 1378h.

Most often, DOS and Windows refer to the first port in numerical order as *LPTI,* the second, *LPT2,* and the third, *LPT3. So* on bootup, LPT1 is most often at 378h, but it may be at any of the three addresses. LPT2, if it exists, may be at 378h or 278h, and LPT3 can only be at 278h. Various configuration techniques can change these assignments, however, so not all systems will follow this convention. LPT stands for line printer, reflecting the port's original intended use.

If your port's hardware allows it, you can add a port at any unused port address in the system. Not all software will recognize these non-standard ports as LPT ports, but you can access them with software that writes directly to the port registers.

Interrupts

Most parallel ports are capable of detecting interrupt signals from a peripheral. The peripheral may use an interrupt to announce that it's ready to receive a byte, or that it has a byte to send. To use interrupts, a parallel port must have an assigned interrupt-request level (IRQ).

Conventionally, LPT1 uses IRQ7 and LPT2 uses IRQ5. But IRQ5 is used by many sound cards, and because free IRQ levels can be scarce on a system, even IRQ7 may be reserved by another device. Some ports allow choosing other IRQ levels besides these two.

Many printer drivers and many other applications and drivers that access the parallel port don't require parallel-port interrupts. If you select no IRQ level for a port, the port will still work in most cases, though sometimes not as efficiently, and you can use the IRQ level for something else.

DMA Channels

ECPs can use direct memory access (DMA) for data transfers at the parallel port. During the DMA transfers, the CPU is free to do other things, so DMA transfers can result in faster performance overall. In order to use DMA, the port must have an assigned DMA channel, in the range 0 to 3.

Finding Existing Ports

DOS and Windows include utilities for finding existing ports and examining other system resources. In Windows 95, click on *Control Panel, System, Devices, Ports,* and click on a port to see its assigned address and (optional) IRQ level and DMA

For this reason, every port *should* come with a simple way to configure the port. If the port is on the motherboard, look in the CMOS setup screens that you can access on bootup. Other ports may use jumpers to enable the modes, or have configuration software on disk.

The provided setup routines don't always offer all of the available options or explain the meaning of each option clearly. For example, one CMOS setup I've seen allows only the choice of *AT* or *PS/2-type* port. The PS/2 option actually configures the port as an ECP, with the ECP's PS/2 mode selected, but there is no documentation explaining this. The only way to find out what mode is actually selected is to read the chip's configuration registers. And although the port also supports EPP, the CMOS. setup includes no way to enable it, so again, accessing the configuration registers is the only option.

If your port is EPP- or ECP-capable but the setup utility doesn't offer these as choices, a last resort is to identify the controller chip, obtain and study its data sheet, and write your own program to configure the port.

The exact terminology and the number of available options can vary, but these are typical configuration options for a multi-mode port:

SPP. Emulates the original port. Also called *AT-type* or *ISA-compatible.*

PS/2, or simple bidirectional. Like an SPP, except that the data port is bidirectional.

EPP. Can do EPP transfers. Also emulates an SPP. Some EPPs can emulate a PS/2-type port.

ECP. Can do ECP transfers. The ECP's internal modes enable the port to emulate an SPP or PS/2-type port. An additional internal mode, *Fast Centronics,* or *Parallel-Port FIFO,* uses the ECP's buffer for faster data transfers with many old-style (SPP) peripherals.

ECP + EPP. An ECP that supports the ECP's internal mode 100, which emulates an EPP. The most flexible port type, because it can emulate all of the others.

Drivers

After setting up the port's hardware, you may need to configure your operating system and applications to use the new port.

For DOS and Windows 3.1 systems, on bootup the operating system looks for ports at the three conventional addresses and assigns each an LPT number.

In Windows 3.1, to assign a printer to an LPT port, click on *Control Panel,* then *Printers.* If the printer model isn't displayed, click *Add* and follow the prompts.

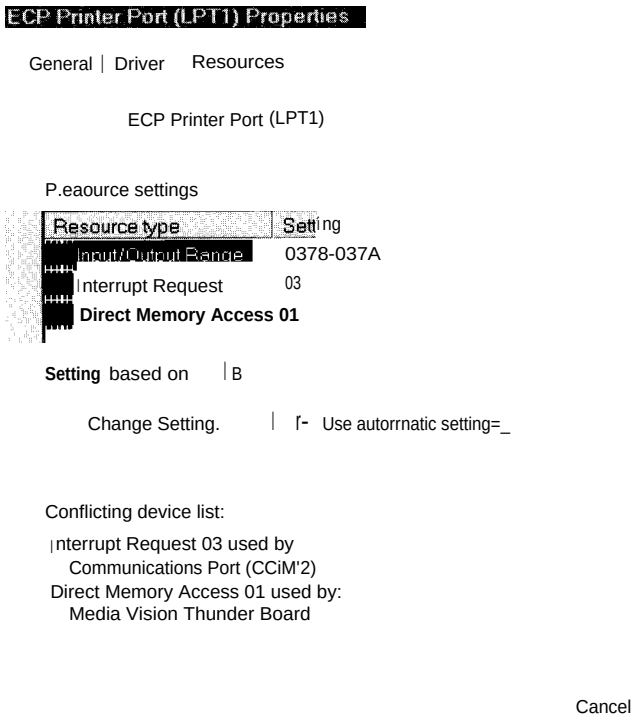


Figure 1-1: In Windows 95, you can select a port configuration in the Device Manager's Resources Window. A message warns if Windows detects any system conflicts with the selected configuration.

Select the desired printer model, then click *Connect* to view the available ports. Select a port and click *OK,* or *Cancel* to make no changes.

In Windows 95, the Control Panel lists available ports under *System Properties, Device Manager, Ports.* There's also a brief description of the port. *Printer Port* means that Windows treats the port as an ordinary SPP, while *ECP Printer Port* means that Windows will use the abilities of an ECP if possible. To change the driver, select the port, then *Properties, Driver,* and Show *All Drivers.* Select the driver and click *OK.* If an ECP doesn't have an IRQ and DMA channel, the Windows 95 printer driver will use the ECP's Fast Centronics mode, which transfers data faster than an SPP, but not as fast as ECP.

The Device Manager also shows the port's configuration. Select the port, then click *Resources.* Figure 1-1 shows an example. Windows attempts to detect these settings automatically. If the configuration shown doesn't match your hardware setup, de-select the *Use Automatic Settings* check box and select a different configuration. If none matches, you can change a setting by double-clicking on the

resource type and entering a new value. Windows displays a message if it detects any conflicts with the selected settings. To assign a printer to a port, click on Con*trol Panel*, *Printers,* and select the printer to assign.

Parallel-port devices that don't use the Windows printer drivers should come with their own configuration utilities. DOS programs generally have their own printer drivers and methods for selecting a port as well.

Adding a Port

Most PCs come with one parallel port. If there's a spare expansion slot, it's easy to add one or two more. Expansion cards with parallel ports are widely available.

Cards with support for bidirectional, EPP, and ECP modes are the best choice unless you're sure that you won't need the new modes, or you want to spend as little as possible. Cards with just an SPP are available for as little as $15. A card salvaged from an old computer may cost you nothing at all.

You can get more use from a slot by buying a card with more than a parallel port. Because the port circuits are quite simple, many multi-function cards include a parallel port. Some have serial and game ports, while others combine a disk controller or other circuits with the parallel port. On older systems, the parallel port is on an expansion card with the video adapter. These should include a way to disable the video adapter, so you can use the parallel port in any system.

When buying a multi-mode port, it's especially important to be sure the port comes with utilities or documentation that shows you how to configure the port in all of its modes. Some multi-mode ports default to an SPP configuration, where all of the advanced modes are locked out. Before you can use the advanced modes, you have to enable them. Because the configuration methods vary from port to port, you need documentation.

Also, because the configuration procedures and other port details vary from chip to chip, manufacturers of ECP and EPP devices may guarantee compatibility with specific chips, computers, or expansion cards. If you're in the market for a new parallel port or peripheral, it's worth trying to find out if the peripheral supports using EPP or ECP mode with your port.

Port Hardware

The parallel port's hardware includes the back-panel connector and the circuits and cabling between the connector and the system's expansion bus. The PC's microprocessor uses the expansion bus's data, address, and control lines to trans-

Figure 1-2: The photo on the left shows the back panel of an expansion card, with a parallel port's 25-pin female D-sub connector on the left side of the panel. (The other connector is for a video monitor.) The photo on the right shows the 36-pin female Centronics connector used on most printers.

fer information between the parallel port and the CPU, memory, and other system components.

Connectors

The PC's back panel has the connector for plugging in a cable to a printer or other device with a parallel-port interface. Most parallel ports use the 25-contact D-sub connector shown in Figure 1-2. The shell (the enclosure that surrounds the contacts) is roughly in the shape of an upper-case D. Other names for this connector are the subminiature D, DB25, D-shell, or just D connector. The IEEE 1284 standard for the parallel port calls it the IEEE 1284-A connector.

Newer parallel ports may use the new, compact, 36-contact IEEE 1284-C connector described in Chapter 6.

The connector on the computer is female, where the individual contacts are sockets, or receptacles. The cable has a mating male connector, whose contacts are pins, or plugs.

The parallel-port connector is usually the only female 25-pin D-sub on the back panel, so there should be little confusion with other connectors. Some serial ports use a 25-contact D-sub, but with few exceptions, a 25-pin serial D-sub on a PC is male, with the female connector on the cable-the reverse of the parallel-port convention. (Other serial ports use 9-pin D-subs instead.)

SCSI is another interface whose connector might occasionally be confused with the parallel port's. The SCSI interface used by disk drives, scanners, and other devices usually has a 50-contact connector, but some SCSI devices use a 25-contact D-sub that is identical to the parallel-port's connector.

If you're unsure about which is the parallel-port connector, check your system documentation. When all else fails, opening up the enclosure and tracing the cable from the connector to an expansion board may offer clues.

The Circuits Inside

Inside the computer, the parallel-port circuits may be on the motherboard or on a card that plugs into the expansion bus.

The motherboard is the main circuit board that holds the computer's microprocessor chip as well as other circuits and slots for expansion cards. Because just about all computers have a parallel port, the port circuits are often right on the motherboard, freeing the expansion slot for other uses. Notebook and laptop computers don't have expansion slots, so the port circuits in these computers must reside on the system's main circuit board.

The port circuits connect to address, data, and control lines on the expansion bus, and these in turn interface to the microprocessor and other system components.

Cables

Most printer cables have a 25-pin male D-sub connector on one end and a male 36-contact connector on the other. Many refer to the 36-contact connector as the Centronics connector, because it's the same type formerly used on Centronics printers. Other names are parallel-interface connector or just printer connector. IEEE 1284 calls it the 1284-B connector.

Peripherals other than printers may use different connectors and require different cables. Some use a 25-pin D-sub like the one on the PC. A device that uses only a few of the port's signals may use a telephone connector, either a 4-wire RJI I or an 8-wire RJ45. Newer peripherals may have the 36-contact 1284-C connector.

In any case, because the parallel-port's outputs aren't designed for transmitting over long distances, it's best to keep the cable short: 6 to 10 feet, or 33 feet for an IEEE-1284-compliant cable. Chapter 6 has more on cable choices.

Multiple Uses for One Port

If you have more than one parallel-port peripheral, the easiest solution is to add a port for each. But there may be times when multiple ports aren't an option. In this case, the alternatives are to swap cables as needed, use a switch box, or daisy-chain multiple devices to one port.

If you use only one device at a time and switch only occasionally, it's easy enough to move the cable when you want to use a different device.

For frequent swapping, a more convenient solution is a switch box. A typical manual switch box has three female D-sub connectors. A switch enables you route

the contacts of one connector to either of the others. To use the switch box to access two peripherals on one port, you'll need a cable with two male D-subs to connect the PC to the switch box, plus an appropriate cable from the switch box to each peripheral.

You can also use a switch box to enable two PCs to share one printer or other peripheral. This requires two cables with two male D-subs on each, and one peripheral cable. Switch boxes with many other connector types are also available.

Manual switches are inexpensive, though some printer manufacturers warn that using them may damage the devices they connect to. A safer choice is a switch that uses active electronic circuits to route the signals. Some auto-sensing switches enable you to connect multiple computers to one printer, with first-come, first-served access. When a printer is idle, any computer can access it. When the printer is in use, the switch prevents the other computers from accessing it. However, these switches may not work properly if the peripherals use bidirectional communications, or if the peripheral uses the control or status signals in an unconventional way.

The parallel ports on some newer peripherals support a daisy-chain protocol that allows up to eight devices to connect to a single port. The PC assigns a unique address to each peripheral, which then ignores communications intended for the other devices in the chain. The software drivers for these devices must use the protocol when they access the port. The last device in the chain can be daisy-chain-unaware; it doesn't have to support the protocol. Chapter 11 has more on daisy chains.

Security Keys

Security keys, or dongles, are a form of copy protection that often uses the parallel port. Some software-usually expensive, specialized applications-includes a security key that you must plug into the parallel port in order to run the software. If you don't have the key installed, the software won't run.

The key is a small device with a male D-sub connector on one end and a female D-sub on the other. You plug the key into the parallel-port connector, then plug your regular cable into the security key. When the software runs, it attempts to find and communicate with the key, which contains a code that the software recognizes. The key usually doesn't use any conventional handshaking signals, so it should be able to live in harmony with other devices connected to the port.

The keys do require power, however. If you have a key that draws more than a small amount of current, and if your parallel port has weak outputs, you may have problems in using other devices on the same port as the key.

Alternatives to the Parallel Port

The parallel port is just one of many ways to interface inputs and outputs to a computer. In spite of its many virtues, the parallel port isn't the best solution for every project. These are some of the alternatives:

Serial Interfaces

One large group of parallel-port alternatives is serial interfaces, where data bits travel on a single wire or pair of wires (or in the case of wireless links, a single transmission path.) Both ends of the link require hardware or software to translate between serial and parallel data. There are many types of serial interfaces available for PCs, ranging from the ubiquitous RS-232 port to the newer RS-485, USB, IEEE-1394, and IrDA interfaces.

RS-232

Just about every PC has at least one RS-232 serial port. This interface is especially useful when the PC and the circuits that you want to connect are physically far apart.

As a rule, parallel-port cables should be no longer than 10 to 15 feet, though the IEEE-1284 standard describes an improved interface and cable that can be 10 meters (33 feet). In contrast, RS-232 links can be 80 feet or more, with the exact limit depending on the cable specifications and the speed of data transfers.

RS-232 links are slow, however. Along with each byte, the transmitting device normally adds a start and stop bit. Even at 115,200 bits per second, which is a typical maximum rate for a serial port, the data-transfer rate with one start and stop bit per byte is just 11,520 bytes per second.

RS-485

Another useful serial interface is RS-485, which can use cables as long as 4000 feet and allows up to 32 devices to connect to a single pair of wires. You can add an expansion card that contains an RS-485 port, or add external circuits that convert an existing RS-232 interface to RS-485. Other interfaces similar to RS-232 and RS-485 are RS-422 and RS-423.

Universal Serial Bus

A new option for I/O interfacing is the Universal Serial Bus (USB), a project of a group that includes Intel and Microsoft. A single USB port can have up to 127 devices communicating at either 1.5 Megabits/second or 12 Megabits/second over a 4-wire cable. The USB standard also describes both the hardware interface and software protocols. Newer PCs may have a USB port built-in, but because it's so new, most existing computers can't use it without added hardware and software drivers.

IEEE 1394

The IEEE-1394 high-performance serial bus, also known as Firewire, is another new interface. It allows up to 63 devices to connect to a PC, with transmission rates of up to 400 Megabits per second. The 6-wire cables can be as long as 15 feet, with daisy chains extending to over 200 feet. The interface is especially popular for connecting digital audio and video devices. IEEE-1394 expansion cards are available for PCs.

IrDA

The IrDA (Infrared Data Association) interface allows wireless serial communications over distances of 3 to 6 feet. The link transmits infrared energy at up to 115,200 bits/second. It's intended for convenient (no cables or connectors) transmitting of files between a desktop and laptop computer, or any short-range communications where a cabled interface is inconvenient. Some computers and peripherals now have IrDA interfaces built-in.

Other Parallel Interfaces

SCSI and IEEE-488 are two other parallel interfaces used by some PCs.

SCSI

SCSI (small computer system interface) is a parallel interface that allows up to seven devices to connect to a PC along a single cable, with each device having a unique address. Many computers use SCSI for interfacing to internal or external hard drives, tape back-ups, and CD-ROMs. SCSI interfaces are fast, and the cable can be as long as 19 feet (6 meters). But the parallel-port interface is simpler, cheaper, and much more common.

IEEE 488

The IEEE-488 interface began as Hewlett Packard's GPIB (general-purpose interface bus). It's a parallel interface that enables up to 15 devices to communicate at

speeds of up to 1 Megabyte per second. This interface has long been popular for interfacing to lab instruments. Expansion cards with IEEE-488 interfaces are available.

Custom I/O Cards

Many other types of input and output circuits are available on custom expansion cards. An advantage of these is that you're not limited by an existing interface design. The card may contain just about any combination of analog and digital inputs and outputs. In addition, the card may hold timing or clock circuits, function generators, relay drivers, filters, or just about any type of component related to the external circuits. With the standard parallel port, you can add these components externally, but a custom I/O card allows you to place them inside the computer.

To use an expansion card, you of course need an empty expansion slot, which isn't available in portable computers and some desktop systems. And the custom hardware requires custom software.

PC Cards

Finally, instead of using the expansion bus, some UO cards plug into a PC Card slot, which accepts slim circuit cards about the size of a playing card. An earlier name for these was PCMCIA cards, which stands for *Personal Computer Memory Card International Association,* whose members developed the standard. Many portable computers and some desktop models have PC-Card slots. Popular uses include modems and data acquisition circuits. There are even PC Cards that function as parallel ports. You don't need an internal expansion slot, and you don't have to open up the computer to plug the card in. But again, the standard parallel-port interface is cheaper and more widely available.

2

Accessing Ports

Windows, DOS, and Visual Basic provide several ways to read and write to parallel ports. The most direct way is reading and writing to the port registers. Most programming languages include this ability, or at least allow you to add it. Visual Basic includes other options, including the *Printer* object, the *PrintForm* method, and *Open LPTx.* Windows also has API calls for accessing LPT ports, and 16-bit programs can use BIOS and DOS software interrupts for LPT access.

This chapter introduces the parallel port's signals and ways of accessing them in the programs you write.

The Signals

Table 2-1 shows the functions of each of the 25 contacts at the parallel port's connector, along with additional information about the signals and their corresponding register bits. Table 2-2 shows the information arranged by register rather than by pin number, and including register bits that don't appear at the connector. Most of the signal names and functions are based on a convention established by the Centronics Data Computer Corporation, an early manufacturer of dot-matrix printers. Although Centronics no longer makes printers, its interface lives on.

Table 2-1: Parallel Port Signals, arranged by pin number.

Pin: D-sub	Signal	Function	Source	Register		Inverted at connector?	Pin: Centronics
				Name	Bit #		
1	nStrobe	Strobe DO-D7	PC1	Control	0	Y	1
2	DO	Data Bit 0	PC,	Data	0	N	2
3	D1	Data Bit 1	pC2	Data	1	N	3
4	D2	Data Bit 2	pC2	Data	2	N	4
5	D3	Data Bit 3	PC,	Data	3	N	5
6	D4	Data Bit 4	pC2	Data	4	N	6
7	D5	Data Bit 5	pC	Data	5	N	7
8	D6	Data Bit 6	pC2	Data	6	N	8
9	D7	Data Bit 7	pC2	Data	7	N	9
10	nAck	Acknowledge (may trigger interrupt)	Printer	Status	6	N	10
11	Busy	Printer busy	Printer	Status	7	Y	11
12	PaperEnd	Paper end, empty (out of paper)	Printer	Status	5	N	12
13	Select	Printer selected (on line)	Printer	Status	4	N	13
14	nAutoLF	Generate automatic line feeds after carriage returns	pC1	Control	1	Y	14
15	nError (nFault)	Error	Printer	Status	3	N	32
16	nInit	Initialize printer (Reset)	PC1	Control	2	N	31
17	nSelectIn	Select printer (Place on line)	PC]	Control	3	Y	36
18	Gnd	Ground return for nStrobe, DO					19,20
19	God	Ground return for D1, D2					21,22
20	God	Ground return for D3, D4					23,24
21	God	Ground return for D5, D6					25,26
22	Gnd	Ground return for D7, nAck					27,28
23	Gnd	Ground return for nSelectIn					33
24	Gnd	Ground return for Busy					29
25	God	Ground return for nInit					30
	Chassis	Chassis ground					17
	NC	No connection					15,18,34
	NC	Signal ground					16
	NC	+5V	Printer				35

1 Setting this bit high allows it to be used as an input (SPP only). 2 Some Data ports are bidirectional.

The signal names in the tables are those used by the parallel port in the original IBM PC. The names describe the signals' functions in PC-to-peripheral transfers. In other modes, the functions and names of many of the signals change.

Table 2-2: Parallel port bits, arranged by register.

Data Register (Base Address)					
Bit	**Pin: D-sub**	**Signal Name**	**Source**	**Inverted at connector?**	Pin: **Centron-ics**
0	2	Data bit 0	PC	no	2
1	3	Data bit 1	PC	no	3
2	4	Data bit 2	PC	no	4
3	5	Data bit 3	PC	no	5
4	6	Data bit 4	PC	no	6
5	7	Data bit 5	PC	no	7
6	8	Data bit 6	PC	no	8
7	9	Data bit 7	PC	no	9
Some Data ports are bidirectional. (See Control register, bit 5 below.)					
Status Register (Base Address +1)					
Bit	**Pin: D-sub**	**Signal Name**	**Source**	**Inverted at connector?**	Pin: **Centron-ics**
3	15	nError (nFault)	Peripheral	no	32
4	13	Select	Peripheral	no	13
5	12	PaperEnd	Peripheral	no	12
6	10	nAck	Peripheral	no	10
7	11	Busy	Peripheral	yes	11
Additional bits not available at the connector: 0: may indicate timeout (1=timeout). 1, 2: unused.					
Control Register (Base Address +2)					
Bit	**Pin: D-sub**	**Signal Name**	**Source**	**Inverted at connector?**	Pin: **Centron-ics**
0	1	nStrobe	PCI	yes	1
1	14	nAutoLF	PC1	yes	14
2	16	nInit	PC	no	31
3	17	nSelectIn	PC'	yes	36
'When high, PC can read external input (SPP only). Additional bits not available at the connector: 4: Interrupt enable. 1=IRQs pass from *nAck* to system's interrupt controller. O=IRQs do not pass to interrupt controller. 5: Direction control for bidirectional Data ports. 0=outputs enabled. 1 =outputs disabled; Data port can read external logic voltages. 6,7: unused					

Centronics Roots

The original Centronics interface had 36 lines, and most printers still use the same 36-contact connector that Centronics printers had. The PC, however, has a 25-pin connector, probably chosen because it was small enough to allow room for another connector on the back of an expansion card.

The 25-pin connector obviously can't include all of the original 36 contacts. Some non-essential control signals are sacrificed, along with some ground pins. The PC also assigns new functions to a couple of the contacts. Table 2-3 summarizes the differences between the signals on the original Centronics and PC interfaces.

Naming Conventions

The standard parallel port uses three 8-bit port registers in the PC. The PC accesses the parallel-port signals by reading and writing to these registers, commonly called the Data, Status, and Control registers.

Each of the signals has a name that suggests its function in a printer interface. In interfaces to other types of peripherals, you don't have to use the signals for their original purposes. For example, if you're not interfacing to a printer, you don't need a paper-end signal, and you can use the input for something else.

Because this book concentrates on uses other than the standard printer interface, I often use more generic names to refer to the parallel-port signals. The eight Data bits are *DO-D7,* the five Status bits are S3-S7, and the four Control bits are CO-C3. The letter identifies the port register, and the number identifies the signal's bit position in the register.

To complicate things, the port's hardware inverts four of the signals between the connector and the corresponding register bits. For S7, CO, Cl, and C3, the logic state at the connector is the complement, or inverse, of the logic state of the corresponding register bit. *When you write to any of these bits, you have to remember to write the inverse of the bit you want at the connector. When you read these bits, you have to remember that you're reading the inverse of what's at the connector.*

In this book, when I refer to the signals by their register bits, an overbar indicates a connector signal that is the inverse of its register bit. For example, register bit CO becomes CO at the connector. The descriptive names *(nStrobe, Busy)* always refer to the signals at the connector, with a leading *n* indicating that a signal is active-low. For example, *nStrobe* and CO are the same signal. *nStrobe* tells you that the signal is a low-going pulse whose function is to strobe data into a peripheral, but the name tells you nothing about which register bit controls the signal. CO tells you that you that the signal is controlled by bit 0 in the Control register, and

Table 2-3: Differences between original Centronics interface and PC interface

Pin (Centronics)	Original Function	New (PC) Function
14	signal ground	nAutoLF
15	oscillator out	no connection
16	signal ground	no connection
17	chassis ground	no connection
18	+5V	no connection
33	light detect	Ground return for nSelectIn
34	line count	no connection
35	Ground return for line count	no connection
36	Reserved	nSelectIn

The PC's D-sub connector has just 25 contacts, compared to the Centronics connector's 36. Six of the original Centronics signals have no connection at the PC, and the PC has five fewer ground-return pins.

The PC interface also redefines three signals. Pin 14 *(Signal Ground) is nAutoLF* on the PC, pin 36 *(Reserved)* is nSelectIn, and pin 33 *(Light Detect)* is the ground return for *nSelectIn.*

that the register bit is the inverse of the signal at the connector, but the name says nothing about the signal's purpose. Whether to use *nStrobe* or CO depends on which type of information is more relevant to the topic at hand.

The Data Register

The Data port, or Data register, *(DO-D7)* holds the byte written to the Data outputs. In bidirectional Data ports, when the port is configured as input, the Data register holds the byte read at the connector's Data pins. Although the Centronics interface and the IEEE-1284 standard refer to the Data lines as *DI* through *D8,* in this book, I use *DO-D7* throughout, to correspond to the register bits.

The Status Register

The Status port, or Status register, holds the logic states of five inputs, S3 through *S7.* Bits SO-S2 don't appear at the connector. The Status register is read-only, except for SO, which is a timeout flag on ports that support EPP transfers, and can be cleared by software. On many ports, the Status inputs have pull-up resistors. In their conventional uses, the Status bits have the following functions:

SO: Timeout. In EPP mode, this bit may go high to indicate a timeout of an EPP data transfer. Otherwise unused. This bit doesn't appear on the connector.

Sl: Unused.

S2: Unused, except for a few ports where this bit indicates parallel port interrupt status (PIRQ). 0 = parallel-port interrupt has occurred; 1 = no interrupt has occurred. On these ports, reading the Status register sets PIRQ = l.

S3: nError or *nFault.* Low when the printer detects an error or fault. (Don't confuse this one with *PError (S5).* below.)

S4: Select. High when the printer is on-line (when the printer's Data inputs are enabled).

S5: PaperEnd, PaperEmpty, or *PError.* High when the printer is out of paper.

S6: nAck or *nAcknowledge.* Pulses low when the printer receives a byte.- When interrupts are enabled, a transition (usually the rising edge) on this pin triggers an interrupt.

S7 Busy. Low when the printer isn't able to accept new data. Inverted at the connector.

The Control Register

The Control port, or Control register, holds the states of four bits, CO through C3. Conventionally, the bits are used as outputs. On most SPPs, however, the Control bits are open-collector or open-drain type, which means that they may also function as inputs. To read an external logic signal at a Control bit, you write 1 to the corresponding output, then read the register bit. However, in most ports that support EPP and ECP modes, to improve switching speed, the Control outputs are push-pull type and can't be used as inputs. On some multi-mode ports, the Control bits have push-pull outputs in the advanced modes, and for compatibility they switch to open-collector/open-drain outputs when emulating an SPP. (Chapter 5 has more on output types.) Bits C4 through *C7* don't appear at the connector. In conventional use, the Control bits have the following functions:

CO: nStrobe. The rising edge of this low-going pulse signals the printer to read *DO-D7.* Inverted at the connector. After bootup, normally high at the connector.

CI: AutoLF or *Automatic line feed.* A low tells the printer to automatically generate a line feed (ASCII code *OAh)* after each Carriage Return (ASCII *ODh).* Inverted at the connector. After bootup, normally high at the connector.

C2: nInit or *nInitialize.* Pulses low to reset the printer and clear its buffer. Minimum pulse width: 50 microseconds. After bootup, normally high at the connector.

C3: *nSelectIn.* High to tell the printer to enable its Data inputs. Inverted at the connector. After bootup, normally low at the connector.

C4: *Enable interrupt requests.* High to allow interrupt requests to pass from *nAck (S6)* to the computer's interrupt-control circuits. If C4 is high and the port's IRQ

level is enabled at the interrupt controller, transitions at *nAck* will cause a hardware interrupt request. Does not appear at the connector.

C5: Direction control. In bidirectional ports, sets the direction of the Data port. Set to 0 for output (Data outputs enabled), 1 for input (Data outputs disabled). Usually you must first configure the port for bidirectional use (PS/2 mode) in order for this bit to have an effect. Does not appear at the connector. Unused in SPPs.

C6: Unused.

C7: Unused, except for a few ports where this bit performs the direction-setting function normally done by *C5.*

Bidirectional Ports

On the original parallel port, the Data port was designed as an output-only port. The Status port does have five inputs, and on some ports the Control port's four bits may be used as inputs, but reading eight bits of data requires reading two bytes, either the Status and Control ports, or reading one port twice, then forming a byte of data from the values read. For many projects it would be more convenient to use the Data port as an 8-bit input, and sometimes you can do just this.

In the original PC's parallel port, a 74LS374 octal flip-flop drives the Data outputs *(DO-D7).* The Data-port pins also connect to an input buffer, which stores the last value written to the port. Reading the port's Data register returns this value.

If there were a way to disable the Data-port's outputs, you could connect external signals to the Data pins and read these signals at the Data port's input buffer. The 74LS374 even has an output-enable *(OE)* pin. When *OE* is low, the outputs are enabled, and when it's high, the outputs are *tri-stated,* or in a high-impedance state that effectively disables them. On the original PC's port, *OE is* wired directly to ground, so the outputs are permanently enabled.

Beginning with its PS/2 model in 1987, IBM included a bidirectional parallel port whose Data lines can function as inputs as well as outputs. Other computer makers followed with their own bidirectional ports. EPPs and ECPs have other, high-speed modes for reading the Data port with handshaking, but these ports can also emulate the PS/2's simple bidirectional ability.

Configuring for Bidirectional Operation

Most bidirectional ports have two or more modes of operation. To remain compatible with the original port, most have an SPP mode, where the Data port is output-only. This is often the default mode, because it's the safest-it's impossible to disable the Data outputs accidentally. To use a bidirectional Data port for input,

you must first configure the port as bidirectional. The configuration may be in a software utility, or in the system's CMOS setup screen that you can access on bootup, or it may be a jumper on the port's circuit board.

After the port is configured as bidirectional, you can use the Data lines as inputs or outputs by setting and clearing bit 5 in the port's Control register, as described earlier. A 0 selects output, or write (the default), and a 1 selects input, or read. (Just remember that 1 looks like 1 for input, and 0 looks like O for output.) Chapter 4 includes program code to test for the presence of a bidirectional port.

A few ports use bit 7 instead of bit 5 as a direction control. To ensure compatibility with all ports, software can toggle both bits 5 and 7 to set the direction.

In an SPP or a port that hasn't been configured as bidirectional, bit C5 may read as 1 or 0. It's also possible, though rare, to have a bidirectional port whose direction bit is write-only, so you can set and clear the bit, but you can't read the bit to determine its current state. This is especially important to be aware of if you use the technique of reading the Control port, altering selected bits, then writing the value back to the Control port. If bit 5 always reads 1, you'll end up always writing 1 back to the bit, even when you don't want to disable the Data-port outputs! To avoid this problem, keep track of the desired state of bit 5 and always be sure to set or clear it as appropriate when you write to the Control port.

If you have an older output-only parallel port with a 74LS374 driving the Data port, it's possible to modify the circuits so that you can use the Data port for input. Chapter 5 shows how.

On some output-only ports, you may be able to bring the Data outputs high and drive the input buffer with external signals, with no modifications at all. But in doing so, you run the risk of damaging the port circuits. The outputs on non-bidirectional ports aren't designed to be used in this way, and connecting logic outputs to Data lines with enabled outputs can cause damaging currents in both devices. Even if the circuits don't fail right away, the added stress may cause them to fail over time. If the circuit does work, the voltages will be marginal and susceptible to noise, and performance will be slow. So, although some have used this method without problems, I don't recommend it.

Addressing

There are many ways to access a parallel port in software, but all ultimately read or write to the port's registers. The registers are in a special area dedicated to accessing input and output (1/O) devices, including printers as well as the keyboard, disk drives, display, and other components. To distinguish between I/O

ports and system memory, the microprocessor uses different instructions and control signals for each. You can read and write to the ports using assembly language or higher-level languages like Basic, Pascal, and C.

On the original PC, port addresses could range from 0 to 3FFh (decimal 1024). Many newer parallel ports decode an eleventh address line to extend the range to 7FFh (decimal 2048). The number of available ports may seem like a lot, but existing devices use or reserve many of these, so only a few areas are free for other uses. Each address stores 8 bits.

Finding Ports

The PC has some parallel-port support built into its BIOS (Basic Input/Output Services), a set of program routines that perform many common tasks. The BIOS routines are normally stored in a ROM or Flash-memory chip in the computer.

When a PC boots, a BIOS routine automatically tests for parallel ports at each of three addresses: 3BCh, 378h, and 278h, in that order. To determine whether or not a port exists, the BIOS writes to the port, then reads back what it wrote. If the read is successful, the port exists. (This write/read operation doesn't require anything connected to the port; it just reads the port's internal buffer.)

The BIOS routine stores the port addresses in the BIOS data area, a section of memory reserved for storing system information. The port addresses are in a table from 40:08h to 40:0Dh in memory, beginning with LPT1. Each address uses two bytes. An unused address should read 0000.

In rare cases, the next two addresses in the BIOS data area (40:0Eh and 40:0Fh) hold an address for LPT4. But few computers have four parallel ports and not all software supports a fourth port. Some systems use 40:0Eh to store the starting address of an extended BIOS area, so in these systems, the location isn't available for a fourth port. Windows 95 doesn't depend on the BIOS table for storing port addresses, and does allow a fourth LPT port.

Many programs that access the parallel port use this table to get a port's address. This way, users only have to select LPTI, LPT2, or LPT3, and the program can find the address. By changing the values in the BIOS table, you can swap printer addresses or even enter a nonstandard address. This enables you to vary from the port assignments that were stored on boot-up. For example, some older DOS software supported only LPT1. If you want to use a printer assigned to LPT2, you can do so by swapping the two printers' addresses in the table. However, Windows and most DOS programs now allow selecting of any available port, so the need to swap addresses in the BIOS table has become rare. Windows 95's Control Panel allows you to assign any address to an LPT port.

Chapter 2

Direct Port I/O

Reading and writing directly to the port registers gives you the most complete control over the parallel-port signals. Unlike other methods, direct 1/O doesn't automatically add handshaking or control signals; it just reads or writes a byte to the specified port. (In EPP and ECP modes, however, a simple port read or write will cause an automatic handshake.)

To write directly to a port, you specify a port register and the data to write, and instruct the CPU to write the data to the requested port. To read a port, you specify a port register and where to store the data read, and instruct the CPU to read the data into the requested location.

You can use direct port reads and writes under DOS, Windows 3.1, and Windows 95. Under Windows NT, the ports are protected from direct access by applications. You can access ports under NT by using a kernel-mode device driver, such as WinRT's, described in Chapter 10.

Programming in Basic

Basic has long been popular as a programming language, partly because many have found it easy to learn and use. Although the Basic language has evolved hugely over the years, a major focus of Basic has always been to make it as simple as possible to get programs up and running quickly. The latest version of Visual Basic is much more complicated and powerful than the BasicA interpreter that shipped with the original PC, yet many of the keywords and syntax rules are still familiar to anyone who's programmed in any dialect of Basic.

Basic under DOS

For creating DOS programs, two popular Basics are Microsoft's QuickBasic and the QBasic interpreter included with MS-DOS. PowerBasic is another DOS Basic that evolved from Borland's TurboBasic. In all of these, you use Inp and out to access 1/O ports.

This statement writes AAh to a Data port at 378h:

```
OUT(&h378,&hAA)
```

This statement displays the value of a Status port at 379h, using hexadecimal notation:

```
PRINT HEX$(INP(&h379))
```

Visual Basic for Windows

Microsoft's Visual Basic has been the most popular choice for Basic programmers developing Windows programs. Unlike other Basics, however, Visual Basic for Windows doesn't include `Inp` and `out` for port access. However, you can add `Inp` and `out` to the language in a dynamic linked library (DLL).

A DLL contains code that any Windows program can access, including the programs you write in Visual Basic. This book includes two DLLs for port access: *inpout16.dll,* for use with 16-bit programs, including all Visual Basic 3 programs and 16-bit Visual Basic 4 programs, and *inpout32.dll,* for use with 32-bit Visual Basic 4 programs.

The *InpoutI6* files include these:

Inpout16.dll. This is the DLL itself, containing the routines that your programs will access.

Inpout16.bas. This file (Listing 2-1) contains the declarations you must add to any program that uses the new subroutine and function added by the *inpout* DLL. Each `Declare` statement names a subroutine or function, the argument(s) passed to it, and the name of the DLL that contains the subroutine or function.

The use of `Alias` in the Declares enables Visual Basic to use alternate names for the routines. This feature is handy any time that you don't want to, or can't, use the routines' actual names. In this case, the `inp` and `out` routines were compiled with PowerBasic's DLL compiler. Because `Inp` and `Out` are reserved words in PowerBasic, and a routine can't have the same name as a reserved word, I named the routines `Inp16` and `Out16`. Using `Alias` enables you to call them in Visual Basic with the conventional `Inp` and `Out`.

On the user's system, the file *Inpout16.dll.* should be copied to one of these locations: the default Windows directory (usually *\Windows),* the default System directory (usually *\Windows\System),* or the application's working directory. These are the locations that Windows automatically searches when it loads a DLL. If for some reason the DLL is in a different directory, you'll need to add its path to the filename in the Declare statements.

With `Inp` and `out` declared in your program, you can use them much like `Inp` and `Out` in QuickBasic. This statement writes AAh to a Data port at 378h:

```
Out(&h378,&hAA)
```

This statement displays the value of a Status port at 379h, using hexadecimal notation:

```
Debug.Print HEX$(Inp(&h379))
```

InpoutI6 is a 16-bit DLL, which means that you can call it from any 16-bit Visual-Basic program.

```
Declare Function Inp% Lib "InpOut.Dll" Alias "Inp16"   -
(ByVal PortAddress%)
Declare Sub Out Lib "InpOut.Dll" Alias "Out16" -
(ByVal PortAddress%, ByVal ByteToWrite%)
```

Listing 2-1: Declarations for Inp and Out in 16-bit programs.

Calling a 16-bit DLL from a 32-bit program will result in the error message *Bad DLL Calling Convention.* A 32-bit program needs a 32-bit DLL, and this book provides *inpout32* for this purpose. As with *inpout16, you* copy the DLL to a directory where Windows can find it, and declare Inp and out in a.bas module.

Listing 2-2 shows a single declaration file that you can use in both 16-bit and 32-bit Visual Basic 4 programs. It uses Version 4's conditional compiling ability to decide which routines to declare. In a 32-bit program, Win3 2 is True, and the program declares the Inp32 and Out32 contained in *inpout32.* In a 16-bit program, Visual Basic ignores the Win32 section and declares the Inp16 and out 16 contained in *inpout16.*

Visual Basic 3 doesn't support the conditional-compile directives, so version 3 programs have to use the 16-bit-only Declares in Listing 2-1.

The Declares for *inpout32* also use Aliases, but for a different reason. *inpout32 is* compiled with Borland's Delphi. Inp and Out aren't reserved words in Delphi, so the compiler doesn't object to these names. However, in Win32, DLLs' declared procedure names are case-sensitive. If the procedures had the names Inp and out you would have to be very careful to call them exactly that, not INP, out, or any other variation. The Alias enables Visual Basic to define Inp and Out without regard to case, so if you type INP or inp, Visual Basic will know that you're referring to the Inp 3 2 function.

Why did Microsoft leave Inp, Out (and other direct memory-access functions) out of Visual Basic? Direct writes to ports and memory have always held the possibility of crashing the system if a critical memory or port address is overwritten by mistake. Under Windows, where multiple applications may be running at the same time, the dangers are greater. A program that writes directly to a parallel port has no way of knowing whether another application is already using the port.

Under Windows 95, a more sophisticated way to handle port I/O is to use a virtual device driver *(VxD).* The VxD can ensure that only applications with permission to access a port are able to do so, and it can inform other applications when a port isn't available to them.

```
Attribute VB Name = "inpout"
'Declare Inp and Out for port I/0
 Two versions, for 16-bit and 32-bit programs.

#If Win32 Then
'DLL procedure names are case-sensitive in VB4.
 Use Alias so Inp and Out don't have to have matching case in VB.
Public Declare Function Inp Lib "inpout32.dll"
Alias "Inp32" (ByVal PortAddress As Integer) As Integer
Public Declare Sub Out Lib "inpout32.dll"
Alias "Out32" (ByVal PortAddress As Integer, ByVal Value
As Integer)

#Else
Public Declare Function Inp Lib "inpoutl6.Dll"
Alias "Inp16" (ByVal PortAddress As Integer) As Integer
Public Declare Sub Out Lib "inpoutl6.Dll" -
Alias "Out16" (ByVal PortAddress As Integer, ByVal Value As
Integer)
#End If
```

Listing 2-2: Declarations for Inp and Out in version 4 programs, 16-bit or 32-bit.

But sometimes a port is intended just for use with a single application. For example, an application may communicate with instrumentation, control circuits, or other custom hardware. If other applications have no reason to access the port, direct I/O with `Inp` and out should cause no problems, and is much simpler than writing a VxD. (Chapter 3 has more on VxDs.)

Other Programming Languages

Other programming languages, including C, Pascal/Delphi, and of course assembly language, include the ability to access UO ports. Briefly, here's how to do it:

C

In C, you can access a parallel port with the `inp` and outp functions, which are much like Basic's `inp` and out.

This writes AAh to a Data port at 378h:

```
unsigned DataAddress=0x378;
int DataPort;

return 0;
```

This **displays the** value of a Status port at 379h:

```
unsigned StatusAddress=0x379;
int StatusPort;
StatusPort=inp(StatusAddress);
printf ("Status port = %Xh\n",StatusPort);
return 0;
```

Pascal

Pascal programmers can use the port function to access parallel ports.

To write *AAh* to a Data port at 378h:

```
port[378h]:=AAh
```

To read a Status port at 379h:

```
value:=port[379h]
```

Delphi 2.0

The 32-bit version of Borland's Delphi Object Pascal compiler has no port function, but you can access ports by using the in-line assembler.

To write *AAh* to a Data *port* at 378h:

```
asm
 push dx
 mov dx,$378
 mov al, $AA
 out dx,al
 pop dx
end;
```

To read a Status *port* at 379h into the variable ByteValue:

```
var
  ByteValue:byte;
asm
  push dx
  mov dx, $379
  in al,dx
  mov ByteValue,al
  pop dx
end;
```

Assembly Language

In assembly language, you use the microprocessor's `In` and out instructions for *port* access.

To write *AAh* to a Data *port* at 378h:

```
mov dx,378h    ;store port address in dx
```

```
mov al,AAh      ;store data to write in al
out dx,al       ;write data in al to port address in dx
```

To read a Status port at 379h into register al:

```
mov dx,379h     ;store port address in dx
in al,dx        ;read data at port address into al
```

Other Ways to Access Ports

Visual Basic, Windows, and DOS include other ways to access ports that have been assigned an LPT number. These options are intended for use with printers and other devices with similar interfaces. They write bytes to the parallel port's Data port, and automatically check the Status inputs and send a strobe pulse with each byte. Because this book focuses on uses other than printer drivers, most of the examples use direct port reads and writes rather than LPT functions. But the other options do have uses. This section describes these alternate ways to access ports.

LPT Access in Visual Basic

Although Visual Basic has no built-in ability for simple port I/O it does include ways to access LPT ports, including the `Printer` object, the `PrintForm` method, and the `open LPTx` statement. Their main advantage is that they're built into Visual Basic, so you don't have to declare a DLL to use them. The main limitation is that these techniques perform only a few common functions. For example, there's no way to write a specific value to the Control port, or to read the Data port.

Each of the options for accessing LPT ports automates some of the steps used in accessing a device. This can be a benefit or a hindrance, depending on the application. When using these methods to write to a port, instead of having to include code to toggle the strobe line and check the Status port, these details are taken care of automatically. And instead of having to know a port's address, you can select an LPT port by number.

But if your application doesn't need the control signals or error-checking, using these techniques adds things you don't need, and will cause problems if you're using any of the Status and Control signals in unique ways. For example, if you're using the *nStrobe* output for another purpose, you won't want your program toggling the bit every time it writes to the Data port.

These methods won't write to the Data port if the Status port's *Busy* input is high. Of course, if the *Busy* line indicates that the peripheral is busy, this is exactly what you want, but it won't work if you're using the bit for something else.

The Printer Object

Visual Basic's `Printer` object sends output to the default printer. (In Version 4 you can change the printer with a `Set` statement.) Sending the output requires two steps. First, use the `Print` method to place the data to write on the Printer object, then use the `NewPage` or `EndDoc` method to send the data to the printer.

The Printer Object isn't very useful for writing to devices other than printers or other peripherals that dxpect to receive ASCII text, because `NewPage` and `EndDoc` send a form-feed character (OCh) after the data. The device has to be able to recognize the form feed as an end-of-data character rather than as a data byte.

A possible non-printer use for the Printer object would be to send ASCII text to an input port on a microcontroller. Plain ASCII text uses only the characters 21h to 7Eh, so it's easy to identify the form feeds and other control codes. For sending numeric data, ASCII hex format provides a way to send values from 0 to 255 using only the characters 0-9 and A-F. Appendix C has more on this format.

For writing simple data to the parallel port, select Windows' printer driver for the *Generic Line Printer* driver.

To send data to the Printer object, Status bit S3 must be high, and SS and S7 must be low. If not, the program will wait.

Here's an example of using the Printer object.

```
'place the byte AAh on the printer object
Printer.Print Chr$(&hAA)
'place the byte 1Fh on the printer object
Printer.Print Chr$(&h1F)
'or use this format to send text
Printer.Print "hello"
'send the bytes to the printer
Printer.NewPage
```

PrintForm

The `PrintForm` method sends an image of a form to the default printer. Because the form is sent as an image, or pattern of dots, rather than as a byte to represent each character, it's useful mainly for sending data to printers and other devices that can print or display the images.

Here's an example of the PrintForm method:

```
'First, print "hello" on Forml.
Forml.Print "hello"
```

```
'Then send the form's image to the printer.
    Form1.PrintForm
```

Open "LPT1"

The documentation for Visual Basic's `open` statement refers only to using it to open a file, but you can also use it to allow access to a parallel (or serial) **port.**

Here's an example:

```
ByteToWrite=&h55
Open "LPT1" for Output as #1
Print #1, Chr$(ByteToWrite);
```

`"LPT1"` selects the **port** to .write to, and #1 is the unique file number, or in this case the device number, assigned to the port. The semicolon after the value to print suppresses the line-feed or space character that Visual Basic would otherwise add after each write. At the Status port, *nError* (S3) must be high, and *PaperEnd* (S5) and Busy (S7) must be low. If Busy is high, the program will wait, while incorrect levels at *nError* or *PaperEnd will* cause an error message.

Windows API Calls

The Windows API offers yet another way to access parallel ports. The API, or *Application Programming Interface,* contains functions that give programs a simple and consistent way to perform many common tasks in Windows. The API's purpose is much like that of the BIOS and DOS functions under DOS, except that Windows and its API are much more complicated (and capable). To perform a task, a program calls an appropriate API function. Although Windows has no API calls for generic port I/O, it does have extensive support for printer access. If Visual Basic doesn't offer the printer control you need, you can probably find a solution in the API.

Windows uses printer-driver DLLs to handle the details of communicating with different models of printers. Under Windows 3.1, there are dozens of printer drivers, with each driver supporting just one model or a set of similar models. Under Windows 95, most printers use the universal driver *unidrv.dll,* which in turn accesses a data file that holds printer-specific information. The Windows API includes functions for sending documents and commands to a printer, controlling and querying the print spooler, adding and deleting available printers, and getting information about a printer's abilities.

The API's `OpenComm` and `WriteComm` functions offer another way to .write to parallel ports.

This book concentrates on port uses other than the printer interface, so it doesn't include detail on the API's printer functions. Appendix A lists sources with more on the Windows API.

DOS and BIOS Interrupts

In 16-bit programs, MS-DOS and BIOS software interrupts provide another way to write to parallel ports. For DOS programs, QuickBasic has `Call Interrupt` and `Call Interruptx`. The QBasic interpreter included with DOS doesn't have these, however.

In 16-bit Visual-Basic programs, you can use the Vbasm DLL on this book's companion disk. Vbasm includes three interrupt functions: `VbInterrupt`, `VbInterruptX`, and `VbRealModeIntX`. Each is useful in certain situations. (`VbInterrupt` doesn't pass microprocessor registers `ds` and `es`, while `VbInterruptX` and `VbRealModeIntX` do. `VbRealModeIntX` switches the CPU to real mode before calling the interrupt, while the others execute under Windows protected mode. `VbRealModeIntX` is slower, but sometimes necessary.) Vbasm includes many other subroutines and functions, such as `VbInp` and `VbOut` for port access (similar to inpout16), and `Vbpeek` and `Vbpoke` for reading and writing to memory locations.

The Vbasm.txt file includes the declarations for Vbasm's subroutines and functions. You declare and call the DLL's routines in the same way as the `Inp` and `out` examples above. Vbasm is for use with 16-bit programs only. There is no equivalent for 32-bit programs.

BIOS Functions

The PC's BIOS includes three parallel-port functions. You call each with software interrupt 17h.

The BIOS functions are intended for printer operations, but you can use them with other devices with compatible interfaces. Before calling interrupt 17h of the BIOS, you place information (such as the function number, port number, and data to write) in specified registers in the microprocessor.

When you call the interrupt, the BIOS routine performs the action requested and writes the printer status information to the microprocessor's `ah` register, where your program can read it or perform other operations on it.

Just to keep things confusing, when the BIOS routine returns the Status register, it inverts bits 3 and 6. Bit 7 is already inverted in hardware, so the result is that bits 3, 6, and 7 in ah are the complements of the logic states at the connector. (In con-

trast, if you read the Status register directly, only bit 7 will be inverted from the logic states at the connector.)

These are the details of each of the BIOS functions at INT 17h:

Function 00
Sends a byte to the printer.
Called with:
ah=0 (function number)
al=the byte to print
dx=0 for LPT1, dx=1 for LPT2, dx=2 for LPT3
Returns:
ah=printer status

When a program calls function 0, the routine first verifies that Busy (S7) is low. If it's high, the routine waits for it to go low. When Busy is low, the routine writes the value in al to the LPT port specified in dx. *nStrobe* (CO) pulses low after each write. The function returns with the value of the Status port in ah.

Listing 2-3 is an example of how to use interrupt 17, function 0 to write a byte to a parallel port in Visual Basic:

Function 01
Initializes the printer.
Called with:
ah=1 (function number)
dx=0 for LPTI, 1 for LPT2, or 2 for LPT3
Returns:
ah=printer status

Calling function 01 brings *nInit (C2)* of the specified port low for at least 50 microseconds. It also stores the value read from the Status port in ah.

Function 02
Gets printer status.
Called with:
ah=2 (function number)
dx=0 for LPT1, 1 for LPT2, or 2 for LPT3
Returns:
ah=printer status

Function 02 is a subset of Function 0. It reads the Status port and stores the value read in ah, but doesn't write to the port.

MS-DOS Functions

In addition to the BIOS interrupt functions, MS-DOS has functions for parallel-port access. Both use interrupt 21h. Like the BIOS functions, these pulse *nStrobe (CO)* low on each write. These functions won't write to the port unless

```
Dim InRegs As VbRegs
Dim OutRegs As VbRegs
Dim LPT%
Dim TestData%
Dim Status%

'Change to 1 for LPT2, or 2 for LPT3
LPT = 0
TestData = &h55

'Place the data to write in al, place the function# (0)  in ah.
InRegs.ax = TestData ,
'Place (LPT# - 1) in dl.
InRegs.dx = LPT

'Write TestData to the port.
Call VbInterruptX(&H17, InRegs, OutRegs)

'Status is returned in high byte of OutRegs.ax
Status = (OutRegs.ax And &HFF00) / &H100 - &HFF00
'Reinvert bits 3, 6, & 7 so they match the logic states  at the
'connector.
Status = Hex$(Status Xor &HC8)
```

Listing 2-3: Using Bios Interrupt 17h, Function 0 to write to a parallel port.

Busy (S7) and *Paper End* (SS) are low and *nError* (S3) is high. If *Busy* is high, the routine will wait for it to go low. Unlike the BIOS functions, the MS-DOS functions don't return the Status-port information in a register.

Both of the following functions write to the PRN device, which is normally LPTI. MS-DOS's `MODE` command can redirect PRN to another LPT port or a serial port.

Function 05
Writes a byte to the printer.
Called with:
ah=5 (function number)
dl=the byte to write

Listing 2-4 is an example of using Interrupt 21h, Function 5 with Vbasm in Visual Basic.

Function 40h
Writes a block of data to a file or device:
Called with:
ah=40h (function number)
bx=file handle (4 for printer port)

```
Dim InRegs As Vbregs
Dim OutRegs As Vbregs
Dim I%
Dim LPT %
'Change to 1 for LPT2, or 2 for LPT3:
LPT = 0
TestData = &h55
InRegs.dx = TestData   'place the byte to write in dl
InRegs.ax = &H500      'place LPT#-1 in ah
I = VbRealModeIntX(&H21, InRegs, OutRegs)
```

Listing 2-4: Using DOS Interrupt 21 h, Function 5, to write to the parallel port.

cx= number of bytes to be written
dx=offset of first byte of buffer to write
ds=segment of first byte in buffer to write
Returns:
ax=number of bytes read, or error code if carry flag (cf)=1:
5 (access denied), 6 (invalid handle).

Listing 2-5 is an example of using Interrupt 21h, Function 40h in Visual Basic.

Two additional DOS functions provide other options for accessing ports. Function 3Fh accesses files and devices (including the printer port) using a handle assigned by DOS. The standard handle for the `LPT` or `PRN` device is 4. Function 44h reads and writes to disk drives and other devices, including devices that connect to the parallel port.

```
Dim ArrayByte
Dim BytesWritten%
,array containing data to write:
Dim A(0 To 127)
Dim DataWritten as String
LPT = 0              'Change to 1 for LPT2, or 2 for LPT3
NL = Chr(13) + Chr(10) `new line
`create an array that stores 128 bytes
For ArrayByte = 0 To 127
      A(ArrayByte) = ArrayByte
Next ArrayByte
`get the segment and offset of the array
ArraySegment = VbVarSeg(A(0))
ArrayOffset = VbVarPtr(A(0))
InRegs.dx = 4 `file handle for PRN device
InRegs.dx = 128 `number of bytes to write
InRegs.dx = ArrayOffset `array's starting address in segment
InRegs.ax = &H4000 `function # (40h) stored in ah
'write 128 bytes to the parallel port
BytesWritten = VbRealModeIntX(&H21, InRegs, OutRegs)
```

Listing 2-5: Using DOS Interrupt 21 h, Function 40h, to write a block of data to the parallel port.

3

Programming Issues

In many ways, writing a program that accesses a parallel port is much like writing any application. Two programming topics that are especially relevant to parallel-port programming are where to place the code that communicates with the port and how to transfer data as quickly as possible. This chapter discusses options and issues related to these.

Options for Device Drivers

For communicating with printers and other peripherals, many programs isolate the code that controls the port in a separate file or set of routines called a *device driver.* The driver may be as simple as a set of subroutines within an application, or as complex as a Windows virtual device driver that controls accesses to a port by all applications.

The device driver translates between the specific commands that control a device's hardware and more general commands used by an application program or operating system. Using a driver isolates the application from the hardware details. For example, a device driver may translate commands like *Print a character* or *Read a block of data* to code that causes these actions to occur in a specific device. Instead of reading and writing directly to the device, the application or operating system communicates with the driver, which in turn accesses the device.

To access a different device, the application or operating system uses a different driver.

Under MS-DOS, some drivers, such as the mouse driver, install on bootup and any program may access the driver. Other drivers are specific to an application. For example, DOS applications typically ship with dozens of printer drivers. When you select a different printer, the application uses a different driver. Under Windows, the operating system handles the printer drivers, and individual applications use Windows API calls to communicate with the drivers. Individual applications can also install their own device drivers under Windows.

There are several ways to implement a device driver in software. You can include the driver code directly in an application. You can write a separate program and assemble or compile it as a DOS device driver or as a terminate-and-stay-resident program (TSR). You can use any of these methods under MS-DOS and-with some cautions-under Windows. Windows also has the additional options of placing the device-driver code in a dynamic link library (DLL) or a virtual device driver (VxD). Each of these has its pluses and minuses.

Simple Application Routines

For simple port input and output with a device that a single application accesses, you can include the driver code right in the application. This method is fine when the application and driver code are short and simple. If the code is in an isolated subroutine or set of subroutines, it's easy to reuse it in other applications if the need arises. Most of the examples in this book use this technique for the code that handles port accesses.

DOS Drivers

A driver installed as an MS-DOS device driver is accessible to all programs, so it's useful if multiple programs will access the same device. The code has a special format and header that identifies it as a device driver. MS-DOS drivers may have an extension of *.sys, .exe,* or *.com.* A .sys driver is listed in MS-DOS's config.sys file, with the form device=driver. sys, with device being the device name, and driver. sys being the filename of the driver. The driver then installs automatically on bootup. An *exe* or *com* file is an executable file that users can run anytime. To install this type of driver on bootup, include it in the system's *autoexec.bat* file. A common use for DOS drivers is the mouse driver *(mouse.sys, mouse.com).*

DOS Drivers under Windows

DOS device drivers are usable under Windows, with some limitations and drawbacks. Although this book concentrates on Windows programming and won't go into detail about how to write a DOS device driver, some background about using DOS device drivers under Windows is helpful in understanding the alternatives.

The 80286 and higher microprocessors used in PCs can run in either of two modes, *real* or *protected*. In real mode, only one application runs at a time and the application has complete control over memory and other system resources. MS-DOS runs in real mode. Although early versions of Windows could run in real mode, Windows 3.1 and higher require protected mode, which enables multiple applications to run at the same time. To ensure that applications don't interfere with each other, Windows has more sophisticated ways of managing memory and other system resources.

In real mode, reading or writing to a specific memory address will access a particular location in physical memory. In protected mode, Windows uses a descriptor table to translate between an address and the physical memory it points to.

When the microprocessor is in protected mode, Windows can run in either *standard* or *enhanced* mode. Most systems use enhanced mode because the operating system can access more memory-up to 4 Gigabytes-and swap between memory and disk to create a virtual memory space that is much larger than the installed physical memory. Systems with 80286 CPUs must use standard mode, however.

In enhanced mode, Windows divides memory into pages, and the operating system may move the information on a page to a different location in physical memory or to disk. If a program bypasses the operating system and accesses memory directly, there's no guarantee that a value written to a particular address will be at that same physical address later.

MS-DOS device drivers must run in real mode. When a Windows program calls a DOS driver, Windows has to translate between the real and protected-mode addresses. Each time it executes the driver code, Windows switches from protected mode to real mode, then switches back when the driver returns control of the system. All of this takes time, and while the MS-DOS driver has control of the system, other programs can't access the operating system. In a single-tasking operating system like MS-DOS, this isn't a problem. But under Windows, where multiple applications may need to perform actions without delay, an MS-DOS device driver may not be the best choice.

TSRs

Another option is a driver written as a TSR (terminate and stay resident) program. A TSR can reside in memory while other DOS programs run, and users can load

TSRs as needed. You can create TSRs with many DOS programming languages, including C, Turbo Pascal, and PowerBasic, but not QuickBasic.

Like DOS device drivers, TSRs run in real mode, with the same drawbacks. An added complication under Windows is that in a TSR, the program, rather than the operating system, must translate between real- and protected-mode addresses.

Windows Drivers

Windows has other options for device drivers, including DLLs and VxDs. A Visual-Basic program can call a DLL directly or use a Vbx or Ocx to access a DLL or VxD.

DLLs

A DLL (dynamic linked library) is a set of procedures that Windows applications can call. When an application runs, it links to the DLLs declared in its program code, and the corresponding DLLs load into memory. Multiple applications can access the same DLL. The application calls DLL procedures much like any other subroutine or function.

Many programming languages enable you to write and compile DLLs. Creating a DLL can be as simple as writing the code and choosing to compile it as a DLL rather than as an executable *(.exe)* file. Basic programmers can use products like PowerBasic's DLL Compiler to write DLLs in Basic. Visual-Basic programs can call any DLL, whether it was originally written in Basic or another language.

As Chapter 2 showed, a DLL is also a simple way to add the Inp and out that Visual Basic lacks.

VxDs

A VxD (virtual device driver) is the most sophisticated way of implementing a device driver under Windows 3.1 or Windows 95. A VxD can trap any access to a port, whether it's from a Windows or DOS program, and whether it uses a direct port read or write or a BIOS or API call. When a program tries to access a port, the VxD can determine whether or not the program has permission to do so. If it does, the port access is allowed, and if not, the VxD can pass a message to the virtual machine that requested it. A VxD also can respond quickly to hardware interrupts, including interrupts caused by transitions at the parallel port's *nAck* input.

Creating a VxD isn't a simple process. It requires a wealth of knowledge about Windows, the system hardware, and how they interact. Most VxD developers use Microsoft's Device Developers Kit, which includes an assembler and other tools-for use in developing VxDs. Some C compilers also support VxD development.

Because how to write VxDs is a book-length topic in itself, this book won't go into detail on it. Appendix A lists resources on VxD writing. But because Visual-Basic programs can make use of VxDs, some background on how they work is useful.

VxDs require Windows to be in enhanced mode, where a supervisor process called the Virtual Machine Manager (VMM) controls access to system resources. Instead of allowing Windows and DOS programs complete access to the system hardware, the VMM creates one or more Virtual Machines, with each application belonging to a Virtual Machine. The VMM creates a single System Virtual Machine for the Windows operating system and its applications, and a separate virtual machine for each DOS program.

To an application, the Virtual Machine that owns the application appears to be a complete computer system. In reality, many hardware accesses first go through the VMM. The VMM also ensures that each Virtual Machine gets its share of CPU time. This arrangement allows DOS programs, which know nothing about multitasking or Windows, to co-exist with Windows programs.

A process called port trapping can control conflicts between DOS applications, or between a DOS and Windows application. For example, if a Windows program is using the printer port, the VMM will be aware of this, and can prevent a DOS program from accessing the same port.

The VMM is able to control port accesses from any program because it has a higher level of privilege than the applications it's controlling. The 80386 and higher CPUs allow four levels of privilege, though most systems use just two. Ring 3 is the lowest (least powerful), and Ring 0 is the highest. The Virtual Machines run under Ring 3, and the VMM runs under Ring 0.

VxDs run under Ring 0, and this is why they're powerful. A VxD can have complete control over port accesses from any Virtual Machine, and can respond quickly to parallel-port events.

Printer accesses in Windows 95 use two VxDs. Vcomm.vxd is the Windows 95 communications driver, which controls accesses to a variety of devices, including the Windows print spooler. Womm in turn accesses a printer driver called lpt.vxd, which handles functions that are specific to parallel ports. And lpt. vxd in turn accesses data files that contain printer-specific information.

A Virtual Printer Device (VPD) handles contentions when a Windows program requests to use a printer port that is already in use by another Windows program. Windows may display a dialog box that asks the user to decide which application gets to use the port.

Under Windows NT, a kernel-mode driver can control port accesses much like VxDs do under Windows 95.

Hardware Interrupts

Interrupt service routines, like VxDs, run under Ring 0, in protected mode. When a hardware interrupt occurs, the VMM switches to Ring 0 and passes the interrupt request to a special VxD, called the VPICD, that acts as an interrupt controller.

A VxD that wants to service a hardware interrupt must first register the interrupt-service routine (ISR) with the VPICD. When the interrupt occurs, the VPICD calls the VxD.

If no VxD has registered the interrupt, the ISR belongs to one of the Virtual Machines. The VPICD must determine which Virtual Machine owns the interrupt, and then schedule that Virtual Machine so it can service the interrupt. If the interrupt was enabled when Windows started, the interrupt is global and any of the Virtual Machines can execute the ISR. If the interrupt was enabled after Windows started, the interrupt is local, and the VPICD considers the owner of the interrupt to be the Virtual Machine that enabled it.

Custom Controls

Visual-Basic programs can access a special type of software component called the Custom Control. A common use for Custom Controls is to add abilities and features that Visual Basic lacks, such as port I/O or hardware interrupt detecting. Other Custom Controls don't do anything that you couldn't do in Visual Basic alone, but they offer a quick and easy way to add needed functions to an application, often with better performance. Visual Basic includes some Custom Controls, and many more are available from other vendors. Visual Basic supports two types of Custom Controls: the Vbx and the Ocx. Either of these may handle parallel-port accesses.

Vbx

A Vbx is a Custom Control that Visual-Basic 3 and 16-bit Visual-Basic 4 programs can use. A Vbx is a form of DLL that includes properties, events, and methods, much like Visual-Basic's Toolbox controls. The Grid control is an example of a custom control included with Visual Basic. To use a Grid control, you add the file *Grid. vbx* to your project. A Grid item then appears in the Toolbox, and you can add a grid to your project and configure it much as you do with the standard controls.

Ocx

Visual Basic 4 introduced a new form of Custom Control: the Ocx. Like a Vbx, an Ocx has properties and can respond to events. In addition, Ocx's use Object Linking and Embedding (OLE) technology, which enables applications to display and alter data from other applications. An Ocx may be 16-bit or 32-bit. Ocx's aren't limited to Visual Basic; other programming languages can use them as well. Visual Basic 3 programs can't use Ocx's, however. Chapter 10 shows an example of an Ocx that handles port accesses and interrupts in 32-bit programs.

Speed

How fast can you transfer data at the parallel port? The answer depends on many factors, both hardware- and software-related.

Hardware Limits

The circuits in the PC and peripheral are one limiting factor for port accesses.

Bus speed

The clock rate on the PC's expansion bus limits the speed of parallel-port accesses. This is true even if the port's circuits are located on the motherboard, because the CPU still uses the expansion bus's clock and control signals to access the parallel port.

Figure 3-1 shows the timing of the signals on the ISA expansion bus for reading and writing to a parallel port. The signal that controls the timing is *BCLK.* One *BCLK* cycle equals one T-cycle, and a normal read or write to a port takes six T-cycles. During T1, the CPU places the port address on SA0-SA19. These lines connect to the port's address-decoding circuits. (The port hardware usually decodes only the lower 10 or 11 address lines.) On the falling edge of *IOR* (read I/O port) or IOW (write to 1/O port), the port latches the address.

For a write operation, the CPU places the data on SDO-SD7, and on the rising edge of IOW, the data is written to the port register. A normal write allows four wait states (T2-T5) before IOW goes high.

A read operation is similar, except that after four wait states, the data from a port register is available on SDO-SD7, and the CPU reads the data on the rising edge of *IOR.*

In most modern PC's, *BCLK* runs at about 8 Mhz, so a read or write to a port takes at least 750 nanoseconds, for a maximum transfer rate of 1.33 Megabytes/second.

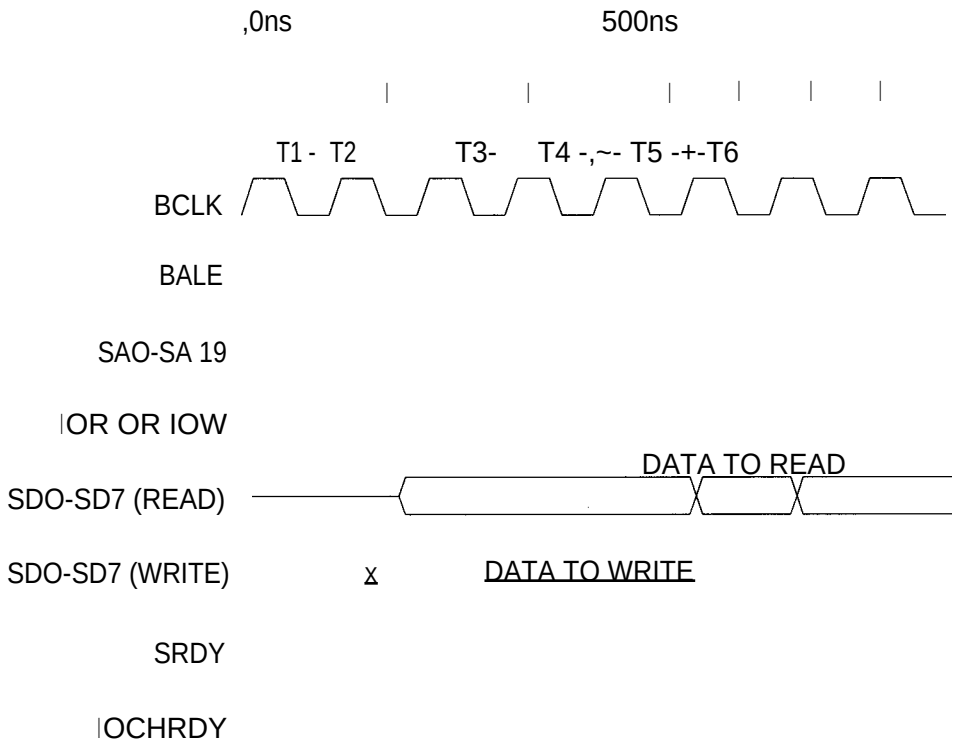


Figure 3-1: Timing diagram for port I/O cycles.

According to the IEEE's ISA-bus standard, *BCLK* may actually vary from 4 to 8.33 Mhz, so you can't assume it will be a particular value. The clock speed of the bus and microprocessor in the original IBM PC was 4.77 Mhz. The 8.33 Mhz rate is the result of dividing a 50-Mhz clock by 6.

For faster access, there is a shortened, or zero-wait-state memory-access cycle achieved by eliminating three of the wait states on the bus. This occurs if the port circuits bring NOWS (no wait states) on the ISA bus low during T2. The data to be read or written must be available by the end of T2. This doubles the speed of port accesses, to 2.67 Megabytes per second on an 8-Mhz bus. Using the shortened cycles requires both hardware and software support. Some of the newer parallel-port controllers support the shortened cycles.

CPU Speed

Because all applications do more than just read and write to ports, the CPU (microprocessor) speed also affects the speed at which a program can transfer data at the parallel port. The speed of a microprocessor's internal operations depends on the clock rate of the timing crystal that controls the chip's operations; a faster clock means faster processing.

The internal architecture of the microprocessor chip also affects how fast it can execute instructions. For example, the Pentium supports pipelining of instructions, which enables new instructions to begin feeding into the chip before previous instructions have finished. Older 80x86 chips don't have this ability.

EPP and ECP Support

A port that supports EPP or ECP modes of data transfer has the best chance for fast parallel-port transfers. An SPP requires four port writes to read the Status port, write a byte to the port, and bring *nStrobe* low, then high. With this handshaking, the fastest that you can write to the port is the time it takes for four port writes, or around 300,000 data bytes per second. If you use the DOS or BIOS software interrupts to write to a port, the speed will be much less because these routines stretch the strobe pulse.

In EPP and ECP modes, the port's hardware takes care of the handshaking automatically, within a single read or write operation. When the PC and peripheral both support one of these modes, you can transfer data at the speed of port writes on the ISA bus, typically 1.3 Mbytes/sec, or 2.7 Mbytes/sec with the shortened cycles. ECPs also support DMA transfers and data compression, discussed below.

For faster switching, a port's Control outputs often switch from open-collector to push-pull type when the port is in ECP or EPP mode.

Cables and Terminations

Cable design and the line-terminating circuits for the cable signals may also affect the maximum speed of data transfers. Chapter 6 has more on this topic.

Software Limits

Software issues that affect access speed include the choice of programming language as well as the program code itself.

Language Choices

Three basic categories of programming languages are assemblers, compilers, and interpreters.

Assemblers

With an assembler, you write programs in an assembly language whose instructions correspond directly to each of the instructions in the microprocessor's instruction set. The assembler translates the program code into machine-level, binary instructions that the microprocessor executes.

Because assembly language gives intimate control over the microprocessor, assembly-language programs can be very fast. But assembly language is a very low-level language that requires detailed knowledge of the microprocessor's architecture. Even the simplest operation requires specifying particular registers in the chip. For example, for the simple task of reading a port, you first store the port address in the dx register, then read the port register into the al register. Then you can perform calculations on the value or move the data to another memory location.

Higher-Level Languages

Higher-level languages make things easier by providing functions, operators, and other language tools that help you perform these and other complex operations more easily.

For example, in Basic, this statement reads a port into a variable:

```
DataRead = INP(PortAddress)
```

You can then use the `DataRead` variable in any way you wish, without concerning yourself with the specific registers or memory locations where the data is stored.

Higher-level languages also include tools that make it easy to display information, read keyboard input, send text and graphics to a printer, store information in files, perform complex calculations, and do other common tasks. Most higher-level languages also have programming environments with tools for easier testing, debugging, and compiling of programs.

Higher-level languages are also somewhat portable. If you learn to program in Basic on a PC, you don't have to learn an entirely new language in order to write Basic programs for a Macintosh, or even a microcontroller like the 8052-Basic.

Two types of higher-level languages are compilers and interpreters.

Compilers

With a compiled language, you create one or more source files that hold your program code. From the source files, the compiler program creates an executable file that runs on its own. Like assembled programs, a compiled program consists of machine code that the microprocessor executes. Examples of compiled languages include the CIC++ compilers from Microsoft, Borland, and others, and Borland's Delphi.

Interpreters

With an interpreted language, you also create source files, but there is no stand-alone executable file. Instead, each time you want to run a program, you run

an interpreter program that translates the source file line by line into machine code.

An advantage to interpreters is that while you're developing a program, you can run the program immediately without having to compile the code first. But because the interpreter has to translate the code each time the program runs, interpreted programs tend to be much slower than compiled ones.

Although future versions may include a compiler, as of Version 4, Visual Basic is an interpreted language. Visual-Basic does create executable *(.exe)* files, but the *.exe* file must have access to a Vbrun DLL, which performs the function of an interpreter on it. QBasic is also an interpreted language. QuickBasic's programming environment includes an interpreter, and you can also compile QuickBasic programs *into.exe* files.

Choices

Different vendors' implementations of the same language will also vary in execution speed. Some compilers allow in-line assembly code, so you can have the best of both worlds by writing the most time-critical code in assembler. An optimizing compiler examines the source files and uses various techniques to make the compiled program as fast as possible. Some compilers claim to produce programs that are as fast as assembled programs, so there's no need to use assembly language at all.

In an interpreted language like Visual Basic, how you write programs has an especially big effect on execution speed. Visual Basic's documentation includes tips for optimizing your code for faster performance, such as using integer variables for calculations and assigning frequently-used object properties to variables. You can also speed execution by eliminating subroutine and function calls in favor of fewer, longer routines. But there's a tradeoff with this technique, because it also tends to make the code less readable, less portable, and harder to maintain.

Programmers endlessly debate the merits of different languages and products, and the products themselves change frequently. Visual Basic's strength is its ease of use, rather than the performance, or speed, of its programs. When speed is essential, a Visual-Basic program can call a DLL that contains the critical code in compiled form. Power Basic's DLL Compiler offers an easy way to place code in a compiled DLL, while still programming in a dialect of Basic.

Windows versus DOS

For the fastest data transfers, and especially for the fastest response to hardware interrupts, DOS beats Windows. A DOS system runs just one program at a time, while a Windows application has to share system time with whatever other appli-

cations a user decides to run. When a hardware interrupt occurs, a DOS program can jump quickly to an interrupt-service routine. Under Windows, the operating system has to decide which driver or virtual machine should service the interrupt and pass control to it, all the while handling the demands of whatever other applications are running. All of that takes time, so under Windows, the interrupt latency, or the time before an interrupt is serviced, is much longer than under DOS, and isn't as predictable.

Code Efficiency

In addition to the programming language you use, how you write your programs can affect execution speed. A complete discussion on how to write efficient program code is well beyond the scope of this book, but a simple example illustrates the issues involved:

You can generate a sine wave or other waveform by connecting a parallel port's outputs to the inputs of a digital-to-analog converter (DAC, and writing a repeating series of bytes to the port. One way to generate the series of bytes would be to use a Sine function to calculate the value for each point in the waveform before writing it. Another, usually faster way is to calculate the values just once, store them, and write the stored values in sequence to the port.

Data Compression

For the fastest data transfers, compressing the data in software can reduce the number of bytes to write. Even though the number of port writes per second doesn't change, the effective transmission rate (the amount of uncompressed data sent per second) is greater. To use this method, you of course have to have software on the receiving end that knows how to decompress what it receives. Parallel ports in ECP mode can automatically decompress incoming data that uses ECP mode's protocol for data compression.

Application-related Limits

The simplest I/O operations just write data from a register to the port, or read the port into a register. But all programs have to do more than just this, and the extra time required for processing and moving data will also limit the rate at which you can access a port in an application.

For example, a program might read an analog-to-digital converter's output in two nibbles, combine the nibbles into a byte, store the byte along with time and date information, display the information, and use the information to decide if the system needs to take an action such as sounding an alarm or adjusting a temperature control. All of this takes time!

Ports that support ECP mode can use direct memory access (DMA), where data can transfer between memory and a port without intervention by the CPU. The DMA transfers use the system's expansion bus, but the CPU is free to perform other tasks during the DMA transfers, and this can speed up the overall performance of some applications.