



# The Parallel Port

Three ways are available to access the parallel port: Direct hardware programming, through the ROM-BIOS or with DOS function calls. In this chapter we'll discuss direct programming and using ROM-BIOS functions to access the parallel port. The first section describes the BIOS functions used in printing. The ROM-BIOS functions offer an advantage over equivalent DOS functions because they allow better control over printer status. DOS immediately fails when a printer triggers the critical error interrupt but BIOS offers other options.

In the second section of this chapter ("Direct Programming And The Parallel Port") we'll talk about direct programming of the parallel port. We'll show you how to connect two computers through their parallel ports and transfer data quickly between these computers using a file transfer program.

## Accessing The Parallel Port From BIOS

BIOS interrupt 17H is reserved exclusively for communication with the parallel port. Most users call interrupt 17H the BIOS printer interrupt although other peripheral devices could also be connected to this port.

### The BIOS printer interrupt

A maximum of three different parallel ports can be connected to the PC (see the "Direct Programming And The Parallel Port" section for more information). Interrupt 17H provides three different functions for addressing these ports. These functions perform three specialized tasks and can control all three parallel ports.

Function	Task
00H	Display characters
01H	Initialize printer
02H	Request printer status

### About these functions

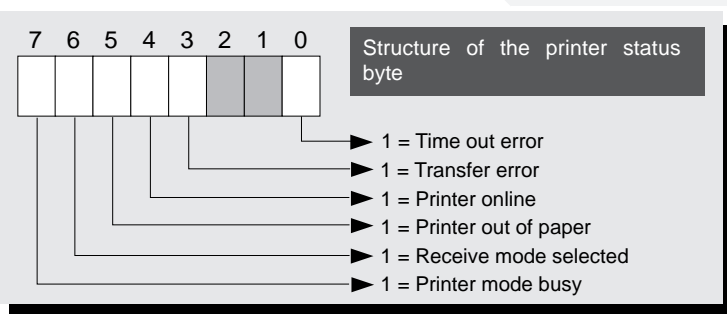
These functions are superior to the equivalent DOS functions in the choice of the addressed port. The DOS functions only control the first parallel port (PRN or LPT1). The three BIOS functions are more flexible in this respect and, when you call them, expect to find the number of the parallel port to be addressed in the DX register. You can specify values of 0, 1, and 2 for the ports: 0 corresponds to LPT1, 1 corresponds to LPT2, and 2 corresponds to LPT3.

### Printer status

These functions have something else in common besides being passed the port number. After being called, each function returns the current printer status in the AH register. The bits of this status byte convey information about whether the printer is currently busy, still has paper, or has encountered an error while receiving characters. This status is very important to the communication with the parallel port.

### Time out error

A time out error, signaled by bit 0 of the status byte, always occurs when the BIOS attempts to send data to the printer over a



certain period of time and the printer is BUSY (bit 7=0) or not accepting the data. This often happens because a parallel port can send up to 100,000 characters per second but no printers on the market can keep up with that pace.

The number of failures that occur before a time out occurs varies with the contents of a BIOS variable. Each parallel port has a byte allocated at a memory range beginning at address 0040:0078H. These bytes specify the number of unsuccessful attempts allowed.

BIOS time out counter for parallel ports	
Address	Meaning
0040:0078	Time out counter: first parallel port
0040:0079	Time out counter: second parallel port
0040:007A	Time out counter: third parallel port

Instead of referring to a given period of time, the values contained in these variables refer to the number of failed attempts that are allowed before BIOS reports a time out error. The program code of the ROM-BIOS continually prompts the parallel port within a program loop. Since this loop consists of only a few assembly language instructions running in cycles of a few microseconds, the time out value from each BIOS variable acts as a factor used in the loop's counter, instead of as the loop counter itself. This factor is multiplied by the constant 262,140 (4 x 65535). The value 20, which the BIOS enters in all three memory locations after the system boots, corresponds to more than five million attempts.

If you use a loop counter instead of a time unit, the period of time that can elapse before a time out occurs depends on the processing speed of the computer. This means that the time span varies with the system's CPU and clock speed. That's why the loop counter must be increased on faster systems, because there isn't a connection between the printer speed and the CPU speed. If you don't make this adjustment, you'll discover, after purchasing a 486, that the system will suddenly send a time out error message during printing.

The BIOS manufacturers usually make this adjustment by using a larger constant instead of a larger default value. For example, if your new computer is twice as fast as a normal AT, instead of 4x65535, you might multiply 8 by 65535. This is done because applications also access the three BIOS variables to change the time out rate for one of the ports. This is possible because these variables are accessible to any program as part of RAM. This gives the programmer the option of setting a higher time out rate for situations in which the printer would otherwise send a time out error. However, increasing the time out rate wouldn't work on a faster system, unless a large enough constant factor is also chosen.

### Other printer status flags

Other bits provide more information about the printer's status. Bit 3 shows a transfer error (a data error in the line), while bit 4 indicates whether the printer is currently online or offline. This bit is the equivalent of the online button found on printers, with an LED indicating its status.

Bit 5 indicates whether the printer has any paper. Bit 6 confirms the printer's receipt of the last character. To determine whether a printer is connected to a particular port, simply prompt for this bit. If it contains a value of 1, then a printer exists.

Bit 7 represents the BUSY signal. It's used by the printer to indicate that it's busy and cannot accept any characters. This bit is also important to the time out error. This signal instructs the ROM-BIOS to repeat the output loop because a character cannot be sent to the printer. This is negative logic: If this bit is set to 0, the printer is busy, and if it is set to 1, the printer is not busy.

Different states of the printer can also result in changes to a series of bits in the status byte. For example, if the printer is ready to print and is online, bits 7 and 4 are set. If you switch the printer offline, not only are bits 7 and 4 cleared, but bit 3 is also set, which signals a transfer error.

### Checking printer status

You may be wondering how you can use this status byte in programs. First, the status byte can prompt for the various states that correspond to the single bits before or after transferring a character. This means that you can determine whether the printer is out of paper, switched offline, or connected to the parallel port.

The status byte can also be used to check for printer access. If bit 1 (time out error), 3 (transfer error), or 5 (out of paper) is set, or bit 4 (printer online) or 7 (printer busy) is cleared, you cannot send characters to the printer. The following pseudocode demonstrates how this is done:

```

pstatus = PrinterStatus;
if ( ( (pstatus and 29h) <> 0 ) or
     ( (pstatus and 80h) = 0 ) or
     ( (pstatus and 10h) = 0 ) ) then
  PrinterOK = FALSE
else
  PrinterOK = TRUE;

```

Now let's return to the three BIOS printer interrupt functions.

Function 00H:	Write character
---------------	-----------------

Function 00H writes a character to the printer. Place the function number (00H) in the AH register and the ASCII code of the desired character in the AL register. After the function call, the AH register receives the status byte.

Function 01H:	Initialize parallel port
---------------	--------------------------

Function 01H initializes the parallel port and printer. Always execute this function before sending data to the printer. Place the function number (01H) in the AH register. After the function call, the AH register receives the status byte.

Function 02H:	Get status byte
---------------	-----------------

Function 02H reads the status byte. With this function, a printing job and the initialization of the parallel port aren't involved. After the function call, the AH register receives the status byte.

### Calling the BIOS functions

Each of these functions can be called from a high level language program in the same way you would call any interrupt. Some C compilers support these functions through their runtime libraries. The table on the right shows the corresponding routines with Borland and Microsoft compilers.

Although QuickBASIC and Turbo Pascal also have printer support, this support uses DOS output functions instead of BIOS functions. If an output error occurs, the DOS functions call critical error interrupt 24H instead of returning an error code to the program. However, this error can be intercepted (e.g., by Turbo Pascal).

QuickBASIC contains the LPRINT statement for transmitting printed output. Turbo Pascal uses Write and WriteLn for the same purpose, provided that the programmer opens a file variable, directs this variable toward the printer, and specifies the printer before calling WriteLn or Write. The PRINTER.TPU unit included with Turbo Pascal performs this task for you (refer to your Turbo Pascal documentation for more information).

C Compiler Support for Printer Functions	
Compiler	Function
Turbo C	
Borland C	
Borland C++	biosprint
Microsoft C	
QuickC	bios_printer

### Redirecting the BIOS printer interrupt

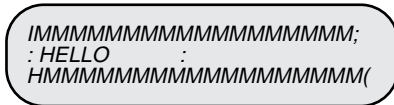
TSR programs redirect the BIOS printer interrupt to their own routines to suit the needs of these TSRs. This allowed the development of print spoolers (programs that intercept characters sent by the original BIOS functions and store these characters in a buffer for later printing).

### Demonstration program

The PRCVT.ASM assembly language program on the companion CD-ROM will help users whose printers uses a different character set than their PCs. For example, if you attempt to print a program listing or file containing PC linedraw characters on some older model Epson printers, the printout may look different than you expected. If the data on the screen looks as follows:



an older Epson printout may look like the following:



The PRCVT.ASM program converts some linedraw characters to ASCII equivalents. This enables you to see how the printout will look on printers that have IBM mode:



PRCVT converts these characters before transferring them to the printer by deflecting the BIOS printer interrupt to its own routine, which is called whenever the BIOS printer interrupt is called. This eliminates the need for a definition of the conversion tables provided by many word processing programs.

#### Automatic character conversion

The new interrupt handler, which is the focus of the PRCVT.ASM TSR program, first checks whether function 00H should be called. This is the only function to be changed. If another function is called, the call is passed to the old printer interrupt. If a character should be output, the program checks a table called CODETAB to determine whether it contains this character. This table, which you can see at the beginning of the program listing of PRCVT.ASM, consists of 2-byte entries, with the first byte (the low byte) containing the new code of a character that will be converted, while the subsequent byte reflects the character's old code. The table is closed by a byte with the value 0.

This new function 00H checks the second byte of a table entry to determine whether it's identical to the character to be printed. If the character isn't found in the table, then it's passed, unchanged, to the old printer interrupt for output. If it's detected in the table, it's replaced by the first byte of the table entry and then passed on for output. The rest of the program is structurally related to other TSR programs documented in this book. PRCVT was created as a COM program and doesn't require any parameters when called from the command line. After being called, it first checks whether it has been installed. If it hasn't been installed, it installs itself; otherwise it removes the installed copy from memory.

**You'll find the following program(s) on the companion CD-ROM**



PRCVT.ASM (Assembler listing)

This program can be used for both BIOS and DOS printed output.

## Direct Programming And The Parallel Port

If the receiver can keep up with the sender, the BIOS functions for parallel port character output work efficiently. Communicating with a printer is the safest method, but linking two computers through their parallel ports is more complicated. This often requires data transfer rates that extend beyond the capabilities of the BIOS functions. A special type of cable with different pin assignments (called a parallel transfer cable) is needed to connect two computers. The BIOS functions cannot be used with this type of cable because they assume that the normal assignments are being used for each line in the cable.

## The I/O ports

Up to three parallel ports can easily be installed in your computer. The I/O address space reserves three ranges for parallel interfaces (see the table on the right). The port addresses in the previous table are listed in the sequence in which BIOS examines them on startup, instead of in numerical order. From this table, BIOS determines which port addresses are LPT1, LPT2, and LPT3.

Port	Interface
3BCH - 3BFH	Parallel interface on MDA card
378H - 37FH	Parallel interface 1
278H - 27FH	Parallel interface 2

The BIOS begins by checking the block at address 3BCH-3BFH. This is part of a large address block that extends from 3B0H to 3BFH, and is reserved for a Monochrome Display Adapter (MDA) or Hercules Graphics Card. During the 1980s most PCs were delivered with this type of video card. In addition to the video logic, these cards included a parallel port.

If the BIOS finds a video card with a parallel port, the BIOS addresses that parallel port as LPT1. The next parallel port found will then be LPT2. If the video card doesn't have a parallel port, then the first parallel port located will be identified as LPT1.

The other two address blocks listed in the table are for additional parallel ports. These ports may exist on two different cards, or on a single I/O card. Regardless of how the hardware for the parallel port is detected, the BIOS checks for the existence of parallel ports according to the previous table. Suppose that only one parallel port is installed, but it's using the address block reserved for the second. This port will still be recognized by the BIOS as LPT1.

### Assigning LPT1, LPT2, and LPT3

The BIOS assigns the names LPT1, LPT2 and LPT3 to the parallel ports by entering their base addresses as variables in the BIOS variable segment. A four-word array starting at offset address 0008H of this segment retains the port addresses of the parallel ports (see the table on the right).

Address	Contents
0040:0008H	Base address of LPT1
0040:000AH	Base address of LPT2
0040:000CH	Base address of LPT3
0040:000EH	Base address of LPT4

The variable segment can accommodate four parallel ports, even though the BIOS will only look for three parallel ports when you start your system. The BIOS functions will also let you work with a fourth parallel port if you enter its base address by hand at offset address 000EH in the BIOS variables segment. The BIOS then addresses it as interface 3.

The LPT terminology originates from DOS rather than BIOS. LPT1 represents interface number 0, LPT2 represents interface number 1, etc. (DOS doesn't recognize LPT4).

If you want to change interface numbers (e.g., have DOS send output intended for LPT2 to LPT1), you must change the port addresses in these three BIOS variables. The pseudocode for this example would look similar to the following:

```
DummyWord = MEM[ 0040H: 0008H ]
MEM[ 0040H: 0008H ] = MEM[ 0040H: 000AH ]
MEM[ 0040H: 000AH ] = DummyWord
```

## Port registers

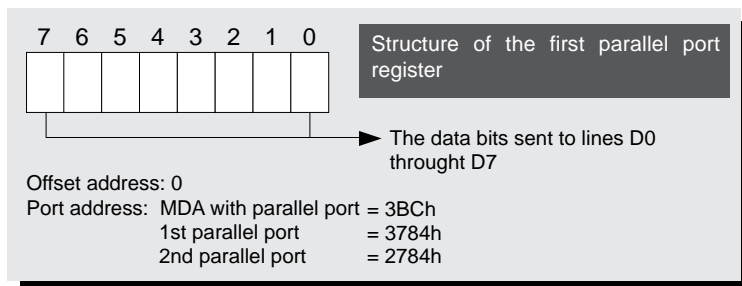
Regardless of their locations in the addressable memory, all parallel ports use the same register interface, which consists of three ports. These ports occupy the first three port addresses of the card (e.g., 378H, 379H and 37AH) for the first parallel port. The following illustrations show the meanings of each bit in the port registers. When you compare the assignments of each bit in the tables to the structure of a parallel cable, you'll see that they mainly coincide. A direct connection exists between the bits of the port registers and the lines in a parallel cable. When a bit in one of the registers is set to 1, then an electrical signal is immediately sent over the corresponding line. If the bit value is set to 0, then the current in the line returns to "low" status. The current in the line will always reflect the status of the corresponding register bit as it is manipulated by the software.

Some of the lines in the cable use negative logic. These lines have names preceded by minus signs. The condition associated with this type of line will be executed if the corresponding bit has a value of 0. For example, the ERROR line indicates a problem with printer output only if the corresponding bit is 0. As long as this bit remains set to 1, no error will be indicated.

### Data lines

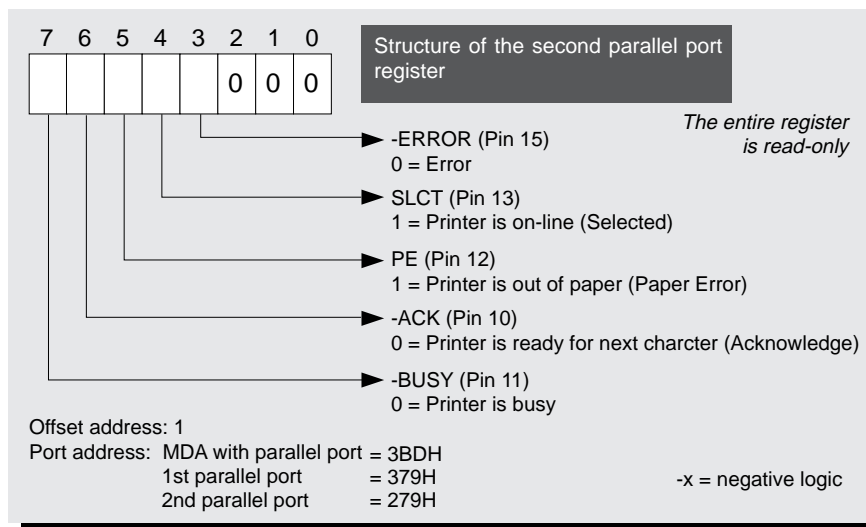
The eight bits of the first parallel port register use positive logic. This register stores the eight data bits that will be carried along data lines D0 to D7 and transferred to the receiver. Remember that this register was intended to be only an output register on a parallel port. It wasn't designed to receive data.

Remember that printers don't send data back to their hosts, and this type of port was never intended to be used for connecting two computers. This will cause some problems when you're developing a communications program because you must deal with communication between two computers as sender and receiver (more on this later).



### Printer status

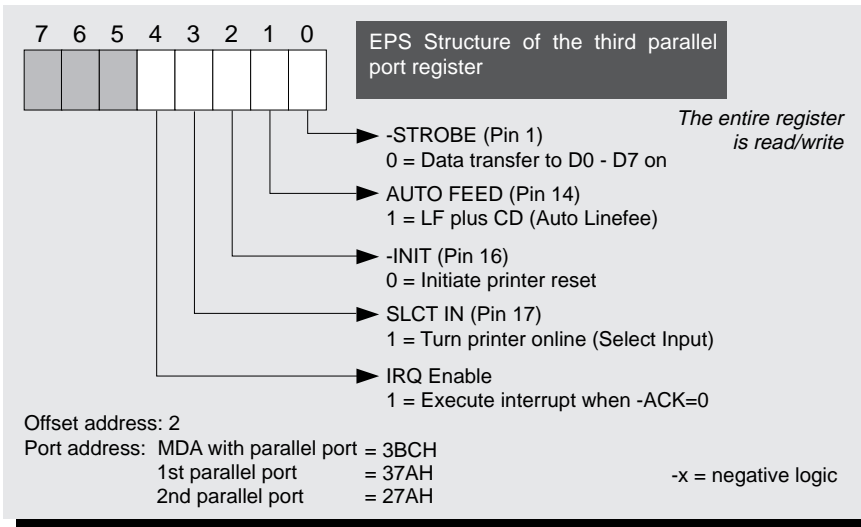
The second register is responsible for the current printer status and is read-only. This register reflects the condition of the status lines coming from the printer. The following illustration names the pins from the host's point of view.



### Printer control

The third register controls the printer and its hardware, and plays an important role in transferring characters. Except for bit 4, all bits are connected to corresponding pins of the parallel port.

A bit hidden in this register can execute a hardware interrupt as soon as the ACK signal switches to low, which indicates that the printer has received the last character. You can usually determine which interrupt will be executed by setting some DIP switches on the port. You can choose between IRQ 7 and 5, which are associated with interrupts 0FH and 0DH. Unlike serial ports, this option is rarely used with parallel ports because these ports work on the polling principle instead of the interrupt principle. This also applies to the BIOS, which doesn't use this interrupt vector.



### Communication between printer and host

The assignment of each pin in the port and the meanings of the corresponding register bits become apparent when you take a "behind the scenes" look at communication between host and printer. First, the byte sent out by the host passes to the first parallel port register and is transmitted through data lines D0 through D7. This signal immediately arrives at the printer but more information is needed before this first byte can be processed. Since there is always some sort of signal coming in on data lines D0 through D7, the printer doesn't know whether this is the first character to print or simply a stray byte from the last transmission.

#### The -STROBE line

The -STROBE line is important for keeping track of data. When the host sets this bit to 0 (and thus setting the current in the corresponding line to low), the printer knows that a character is coming over the data lines. The host must then disable the -STROBE signal quickly; otherwise the printer may read the character twice. The printer hardware needs only a microsecond to read the character from the data lines.

#### The BUSY line

Since a microsecond isn't a very long time, the printer would never be able to keep up with this kind of data transfer rate, even if it stored the characters in an internal buffer. The BUSY line pauses the communication long enough to process the character it has just received. A BUSY signal is generally sent immediately after a -STROBE signal.

The software or ROM BIOS must then wait until the printer removes the BUSY signal before it can send the next character. The BUSY line is the only pin in the parallel port that inverts the signal when it's received. In order for the host to receive 0, the printer must send a value of 1 over the BUSY line.

#### The -ACKnowledge line

The printer must also send an -ACK signal of 0 on the -ACKnowledge line. Because of the negative logic of this line, the host will receive this as a value of 1, which indicates that the printer received the character that was sent.

The durations of all signals needed to transmit one character add up to about 10 microseconds. Theoretically, this would produce a data transfer rate of 100,000 characters per second. However, in reality, processor overhead adds a lot of extra time. Real transfer rates are actually about 1/100th of this (1000 characters per second), even if the printer has its own buffer for storing characters as they are received.

### The printer responds

Although communication between a host and a printer is mostly unilateral, the printer does offer feedback to the host. The printer uses three pins to send information back to the host: -ERROR, SLCT, and PE. All three of these pins have their corresponding bits in the first parallel port register.

SLCT represents "Select". This corresponds to the ONLINE switch found on the front of your printer. If you turn the printer offline, the printer will signal the host using the SLCT line.

PE represents "Paper Error". This allows the printer to tell the host that it's out of paper or that the paper feed is jammed. This type of error is separated from normal data transfer errors, which are transmitted through the ERROR line. This is done because paper errors can be immediately corrected by the user but data transfer errors are more serious. Data transfer errors are usually caused by cable failures or electrical disturbances.

### Host control

Obviously, the host has some control signals that it uses to command the printer. These signals are -AUTO FEED, -INIT, and -SLCT IN. The bits that receive these signals are found in the third parallel port register, where the values can be read or manipulated by software.

-AUTO FEED tells the printer to add a linefeed to every CR (carriage return) character (ASCII code 13) it receives as long as this signal is set to high (1). This line is included because all printers don't react the same way when they receive a CR (carriage return) character. Many printers simply return to the start of the current line without adding the linefeed, which moves the print head down to the next line. So, the LF (linefeed) character must be added separately.

The host can use the -SLCT IN line to turn the printer OFFLINE by sending a signal of 1. Normally, this line will be set to low so the printer stays online.

The host can use the -INIT line to reset the printer. To execute a reset, set the corresponding bit briefly to 0, and then immediately back to 1. If you don't set the value back to 1, the printer will reset itself repeatedly.

### Using the proper cable

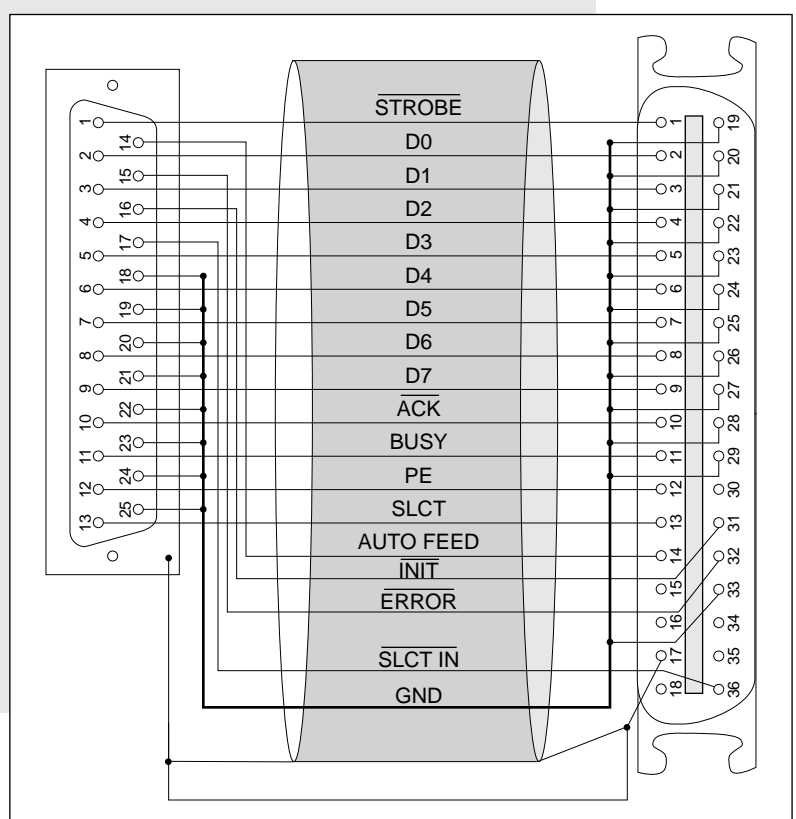
The entire transfer of data between host and printer will only work if the correct pins on the two ports are connected by a proper cable. Which signals are found at which pins and the way in which the pins are connected is standardized. The Centronics standard describes both the pin assignments at each port and the lines in the cable.

The illustration at the top of the following page shows how the pins of the host and printer ports are connected. The second illustration shows the structure of a parallel cable with the ground line.

*Cable connections between parallel port and printer*

Computer pin	Printer pin	Signal name	Meaning
1	1	-STROBE	Indicates data transfer
2	2	D0	Data line - bit 0
3	2	D1	Data line - bit 1
4	2	D2	Data line - bit 2
5	2	D3	Data line - bit 3
6	2	D4	Data line - bit 4
7	2	D5	Data line - bit 5
8	2	D6	Data line - bit 6
9	2	D7	Data line - bit 7
10	10	-ACK	Last character received
11	11	-BUSY	Printer busy
12	12	PE	Printer has no paper
13	13	SLCT	Printer is online
14	14	-AUTO FEED	Automatic CR after LF
15	32	-ERROR	Data transfer error
16	31	-INIT	Reset printer
17	36	SLCT IN	Turn printer online
18-25	19-30	GND	Ground

*Structure of a parallel cable*



### A do-it-yourself parallel transfer cable

If you want to "misuse" your parallel port for transferring data between two computers, a normal parallel cable won't work. One problem is that the parallel ports on both computers have identical female connectors. So, one end of the parallel cable won't plug into a second computer.

Another problem is that normal parallel communications travel in only one direction. One computer can use data lines D0 to D7 to send data, but it cannot receive data over these same lines and the other computer cannot send data over them.

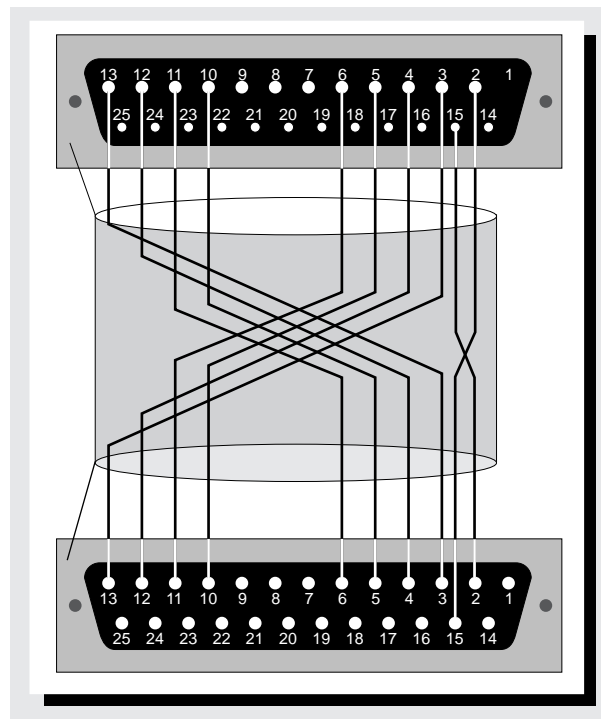
Usually data transfer between two computers requires a bidirectional connection. For example, the receiver will return a checksum of the data it received, so the sender will know whether the data was received without error.

The status lines used by a printer to return status information to the computer can provide a solution. These are the -ERROR, SCLT, PE, -ACK, and BUSY lines, which are associated with the second parallel port register. These lines are connected to data lines D0 to D4. This means that the receiver reads output from the sender through the status lines previously listed. Conversely, data lines D0 to D4 from the receiver are connected to the status lines of the sender, which enables two-way communication.

So, basically we're simply crossing data lines D0 - D4 with the -ERROR, SLCT, PE, -ACK, and -BUSY status lines. So, the following rule applies to both sender and receiver: Any data sent out via the first five bits of the first parallel port register will be received by bits 3 to 7 in the second register of the other communication partner. It doesn't matter which end of the cable is connected to the sender and which end is connected to the receiver.

The following illustration shows which pins to connect at each end of the cable to make a parallel transfer cable.

*Pins to connect when you want to make a parallel transfer cable*



This type of cable is very difficult to find commercially unless you own a LapLink or similar cable. If you want to make your own, you need the following information:

You'll need two male DB-25 connectors and a shielded single-pole cable less than 10 feet in length. Parallel cables longer than this cause data transfer problems.

As the illustration on the right shows, you must connect pins 2 to 6 on one connector with pins 15 and 13 to 10 (leave pin 14 free) on the other. Solder any five lines from the cable to pins 2 to 6 on the first connector. Then connect the other end of each wire to the proper pin on the other connector. Be sure to follow the proper order. For example, D0 must be connected with -ERROR, not SLCT or BUSY.

Now you must repeat the entire procedure for the other side of the cable to cross the connections properly. Don't forget to solder the cable shielding to the connectors as a ground.

*Pin connections  
for a parallel  
transfer cable*

Pin		Pin	
2	15	15	2
3	13	13	3
4	12	12	4
5	10	11	6
6	11	10	5

This type of cable can be used with commercial data transfer programs, such as LapLink. These programs usually work with the same type of cable as our parallel transfer cable. If the cable doesn't work, check your pin connections again. It's also possible that the program assumes the data and status lines are connected in a different order. A parallel transfer cable can be used to connect two PCs and transfer data between them. You can also use this type of cable to control a slave PC from a master PC.

### Sample programs

The following file transfer programs listed are named PLINKP.PAS and PLINKC.C. Both can act as a sender or receiver in parallel file transfer. The basic syntax for calling either program is as follows:

```
PLINKP
PLINKC
```

The operating mode depends on how you start the program. The previous syntax sets the program in receive mode. The receiver waits until the sender begins transmitting or until the user presses **Esc** to exit. If a sender doesn't appear, the receiver waits until a time out error occurs, then exits.

The following examples start the program in sender mode, and try to send FILENAME.EXE to the receiving computer:

```
PLINKP FILENAME.EXE
PLINKC FILENAME.EXE
```

The sender waits until it senses a receiver. If a receiver exists, file transfer begins. If it doesn't, the sender waits until it recognizes a receiver or until the user presses **Esc** to exit. If a receiver still doesn't appear, the sender waits until a time out error occurs, then exits. You can also use wildcards to specify entire groups of files for transfer. The following examples start the program in sender mode, and try to send all EXE files to the receiving computer:

```
PLINKP *.EXE
PLINKC *.EXE
```

The two optional switches /P and /T can also be entered when you start the program. The /P switch specifies the parallel port through which you want information sent other than the default (LPT1). A number between 1 and 4 must follow the /P. The following examples start the program in receive mode and configure LPT3 as the parallel port for receiving data:

```
PLINKP /P3
PLINKC /P3
```

The /T switch specifies the number of time out intervals. A single time out is 10 seconds; you can enter any group of 10 second intervals. Enter a number after the /T switch. The program multiplies this by 10. The following examples start the program in sender mode, request a 30 second time out, and attempt to send all TXT files:

```
PLINKP /T3 * .TXT
PLINKC /T3 * .TXT
```

You can get help by typing PLINKC or PLINKP and the /? parameter. Enter the following to see the command syntax:

```
PLINKP /?
PLINKC /?
```

The program lists the switches for interface and time out intervals. Since these programs are coded, we omitted some features, such as checking for existing files. If you try to transfer a file to a receiver containing a file of the same name, the programs overwrite the existing filename, then time out. Check your receiver before sending files or add your own code to check for files.

### Transferring data over the parallel transfer cable

With a parallel transfer cable like the one we described, you can simultaneously send five bits through data lines D0 to D4. You can transmit data in both directions simultaneously because the connections are crossed. However, some kind of communications protocol is needed so data can be transferred systematically. We need something similar to the -STROBE line to keep track of the data transfer pace. One of our five data lines must be used for this purpose.

The BUSY bit seems to be best suited for this job. The two outer bits (-ERROR and BUSY) are actually the only possibilities because the four data bits must be next to one another. Of these two, BUSY will be a better -STROBE line because it has no real application of its own for data transfer. Also, this bit is automatically inverted by the hardware. If you used this bit for data, it would have to be inverted after the transfer, which would be time-consuming. The inversion doesn't affect a -STROBE bit. Actually, it's important for the communications protocol. A 0 at one end of the cable must come out as a 1 at the other end. We'll discuss this in more detail later.

Communications protocol refers to the two demo programs described in this section. Although there can be numerous communications protocols, in this instance we're being quite specific. The protocol used here works on two levels, the byte level and the data block level. Each of these levels is handled separately. The data block level is on top of the byte level. The byte level is hardware-oriented, but the block level works with the software.

### Data transfer at byte level

First let's discuss the byte level from the sender's point of view. The first thing we must consider is that an entire byte cannot be sent at once. A byte consists of eight bits and we can only send four bits in one direction at one time. So, each byte is divided into two halves called *nibbles*, which are sent one after the other.

First, the low nibble of the byte is written to bits 0 to 3 of the first parallel port register. From here it is sent out through data lines D0 to D3. The bit for data line D4 is set to 0 so the receiver will receive a value of 1 at its BUSY pin. This will tell the receiver that the low nibble of the next byte is ready to be read. This means that the receiver simply waits and reads the status line until the BUSY bit contains a value of 1.

The BUSY bit is then no longer useful, so the receiver proceeds by reading the nibble from the corresponding bits of the second port register. The contents of these four bits are stored in a variable and sent back to the sender through the data lines. The bit for data line D4 is set to 0, so a value of 1 is received back on the other end. This indicates that the returned nibble can be read and saved.

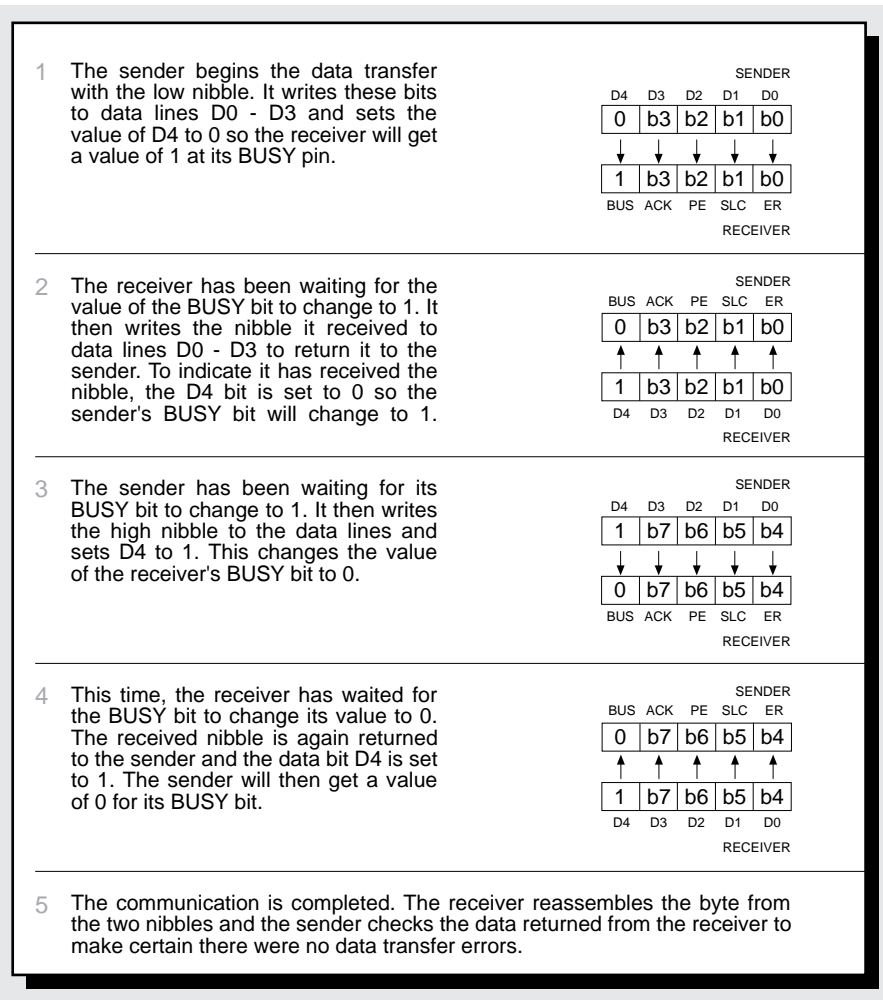
When both nibbles have been returned in this way, the sender can determine whether the byte was properly transferred. The communication is therefore verified at the byte level. However, this isn't the usual procedure. Since it takes too much time to check each byte individually, this type of verification is normally performed only at the block level. In our case, this argument doesn't apply, since the BUSY line must be used to send a -STROBE signal back to the sender with each nibble anyway. It doesn't take any more time to send the entire nibble back.

The transmission of the second nibble is basically the same, except that the value of the status bit is changed to 1. So, the receiver will receive a value of 0 at the BUSY pin. The receiver has been waiting for this as the signal to read the high nibble.

The nibble is then returned to the sender and the sender's BUSY bit is reset to 0. The two nibbles can then be combined to form the complete byte, and the transfer is complete from the receiver's point of view. The sender combines the two nibbles sent back by the receiver and checks the complete byte for data transfer errors. The program's send routine alerts its caller of any errors so the appropriate action can be taken and the data block can be resent if necessary. Most data transfer error can be detected in this way; exceptions include unusual types of cable interference.

As we saw with normal communication to a printer, successful communication between two computers requires the proper switching of the -STROBE signal over the BUSY line. Remember that because of the way the lines in the cable are crossed, we're dealing with two separate BUSY lines. The same applies to both sender and receiver: For output, the BUSY bit is data line D4. The signal is received, however, at the BUSY status line on the other side.

*The communications protocol at the hardware oriented byte level*



Sender and receiver both use the data lines D0 - D4 for sending information and each has its own separate -STROBE line. This is shown in the previous illustration.

### Time out problems

Communications protocols usually function without error as long as the electrical current isn't interrupted. If an error occurs, either the sender or receiver will be left waiting for the other end to respond to its last message. To prevent a situation in which

one of the communications partners is waiting forever for an answer that may never come, a time out value is set. The time out value determines how long one of the systems will wait for an answer from its partner before terminating the connection.

We mentioned in the first section the BIOS also uses a time out counter for communicating with the parallel port. The time out interval is usually measured by executing a read loop a certain number of times. For a program that must run on different PC systems, this isn't easy to manage. The time needed to process the read loop can vary with the system's processor speed.

Here is an example of how the time out counter works. Suppose that the sender has just sent the low nibble of a byte. It sets the time out counter to its maximum value and then waits for its BUSY bit to be set to 1 by the receiver. The read loop then begins to execute. It will continue to run until either the value of the BUSY bit changes to 1 or the time out variable reaches 0. The time out variable will continue to count down as long as its value isn't equal to 0.

This would look as follows in pseudocode:

```

TimeOutCount = MAXVALUE
WHILE ( BUSY-Bit = 0 ) AND ( TimeOutCount > 0 ) DO
  BEGIN
  END

IF TimeOutCount = 0 THEN
  error
ELSE
  o.k.
END

```

The communications protocol is activated along with the time out reading in both demo programs with routines called SendAByte and ReceiveAByte.

### Synchronization

Once the communications protocol is activated, it will work without any problems. However, sometimes getting it started can be a problem. Before communications begin, both sender and receiver must have a value of 0 in the BUSY bit. If this is not the case, the receiver will immediately assume that a nibble has already been sent and it will try to read it, although the sender hasn't even sent anything yet.

The sender and receiver must be synchronized before communications can begin. Determining whether the sender or the receiver should be started first is complicated. For the demo programs presented here, we'll use the receiver as our starting point.

For initialization, the receiver waits for the sender to set its BUSY bit to 0. It then sets the sender's BUSY bit to 0. The synchronization is complete when the BUSY bits on both sides are set to 0. In the demo programs, this is done within the PortInit routine. A time out limit is also used in the initialization procedure. It works according to the principle previously described. In addition, the programs enable the user to quit at any time by pressing the **Esc** key. Otherwise, you may have to wait several minutes for the timeout interval to be reached.

### Stopping the program

Both programs include a keyboard interrupt handler that is activated by pressing the **Esc** key. As with the timer interrupt handler, the program also uses a variable to communicate with the keyboard interrupt handler. In this case, it's a variable of type BOOL, which is set to TRUE when the **Esc** key is pressed. We can avoid having to read this variable separately in the read loop by coupling it with the time out variable. This is done by setting both the escape variable to TRUE and the time out variable to 0 when the **Esc** key is pressed.

The program will then simply respond as though the time out interval has been reached. A quick check of the escape variable will then allow you to determine the cause of the interrupt.

### Block level protocol

The block level is above the byte level. As the name suggests, the block level is used for transferring entire data blocks from sender to receiver. This is strictly a software protocol. It's independent of the hardware because it relies on the send and receive routines from the byte level. In our demo programs, these are the SendABlock and ReceiveABlock routines.

A block always contains the following information: A token that precedes the block and describes its contents, the length of the block, and the block itself. The token is used so the receiver can immediately recognize what is being sent without having to read the data block. According to this convention, the SendABlock routine expects to be passed the token, the number of bytes in the block, and a pointer to the data block itself. Both the token and the number of bytes are handled as a sort of header and kept separate from the actual data block. The receiver must first correctly receive the header before the first byte of the data block will be sent. Imagine what would happen if the sender wanted to send a 120 byte data block but the receiver was expecting a block of 200 bytes. The receiver would count the next 80 bytes as part of the first data block, and the communication would be hopelessly tangled.

Remember, at the byte level, the receiver is sending every byte it receives back to the sender. So, the block level protocol will immediately know whether the header was properly transferred. Unfortunately, this doesn't let the receiver know whether it received the header correctly. The sender therefore notifies the receiver by sending a standard character. So, the receiver doesn't have to assume that there weren't errors in the header.

As feedback, the sender will send an ACK character (Acknowledge) character if the transfer was successful, or a NAK character (Non-Acknowledge) character if there was an error. The ACK character isn't related to the port pin of the same name; it simply fulfills the same function. The ACK and NAK characters are represented by the codes 00H and FFH in the demo programs, but you can use any codes. These characters also play a part in the communications protocol, since they are also checked at the byte level for successful transfer.

When the receiver has received the header and the subsequent ACK character without errors, the sender can begin to transfer the actual data block. If there was a problem, the sender repeats the transfer of the header. The receiver will know that the header is being sent again because it would have received a NAK character from the previous attempt. Once the receiver has the header and the ACK character, it can concentrate on receiving the data block. As long as you use very different bit patterns for these two characters, it's unlikely that an ACK character could become a NAK character because of a data transfer error.


If it continues to encounter errors, the sender won't keep trying to send the header forever. The constant MAXTRY is set to tell the sender how many errors to count before aborting the attempt to send the current data block. The ACK and NAK characters are also used to confirm receipt of the data block. The feedback character is sent to the receiver only after the entire data block has been sent.

With this method, every type of communication error can be detected. This protocol eliminates the need for checksums, which is a common way of checking for data transfer errors in other communications software.

### Reading key status

Once the data block is transferred, a final byte is sent to complete the process. But this byte is sent from the receiver to the sender. This also gives the receiver the ability to communicate an ESCAPE signal to the sender. However, this isn't really necessary because the receiver could simply exit the communications software with ESCAPE and then let the sender wait for a time out error.

Since this isn't the best solution, the protocol has the sender wait for an "escape byte" from the receiver after the data block has been completely transferred. If the sender receives a value of TRUE, it exits the program with an appropriate message. Otherwise, the sender continues with normal program execution.

Communicating this type of message must be allowed in both directions, since the sender could also decide to terminate the communication at any time. The procedure for this is different than with the receiver. When it starts, the SendABlock routine determines whether the  key has been pressed. If it has, it sends a special escape token instead of the actual data block header. This special escape token is known to the receiver. When the receiver recognizes this token, it considers it as a signal to exit the program.

By building this escape mechanism into the block protocol, we can avoid having to make a permanent escape query at a higher level. Both of the demo programs also deal with a file level above the block level. The file level uses the block level protocol to transfer entire files piece by piece. We won't go into detail here about the file level, since the program listings at the end of this section are well documented.

Remember that the routines at the byte and block level can be used to transfer any data between sender and receiver. Also, both the sender and receiver are able to abort the communication at any point. These routines could serve as the basis for your own data transfer programs, which can compete with commercial packages such as LapLink. They may not be quite as fast, however, because this would require all of the byte level routines to be written completely in assembly language.

### The higher levels

If an error, such as a time out error at the byte level or the receipt of an escape token at the block level, occurs, this error should be communicated to the highest level as quickly as possible. The various levels (byte, block, and file) in these demo programs use procedures and functions to communicate with each other. Each time a routine ends with an error, it returns to the routine that called it, working its way back to the top level by level.

Modern C compilers support the `setjmp()` and `longjmp()` functions in these instances. These functions allow jumps across several program levels. The `setjmp()` function sets the location in the program code that will be the destination of the jump. If an error occurs, you can change program control to this location by using the `longjmp()` function. For more information about these functions, refer to your C compiler documentation.

Unfortunately, Turbo Pascal doesn't have similar functions. However, you can implement these commands yourself.

***You'll find the following program(s) on the companion CD-ROM***



PLINKP.PAS (Pascal listing)  
PLINKPA.ASM (Assembler listing)  
PLINKC.C (C listing)  
PLINKCA.ASM (Assembler listing)