



---

## Parallel Port

---

### 17.1 Introduction

---

This chapter discusses parallel communications. The Centronics printer interface transmits 8 bits of data at a time to an external device, normally a printer. Normally it uses a 25-pin D-type connector to connect to a 36-pin Centronics printer interface. In the past, it has been one of the most under used parts of a PC and was not normally used to interface to other equipment. This was because, as a standard, it could only transmit data in one direction (from the PC to the external device). Some interface devices overcame this by using four of the input handshaking lines to input data and then multiplexing using an output handshaking line to multiplex them to produce 8 output bits.

As technology has improved there has been a great need for an inexpensive, external bi-directional port to connect to devices such as tape backup drives, CD-ROMs, and so on. The Centronics interface unfortunately lacks speed (150 Kbps), has limited length of lines (2 m) and very few computer manufacturers have complied with an electrical standard.

Thus, in 1991, several manufacturers (including IBM and Texas Instruments) formed a group called NPA (National Printing Alliance). Their original objective was to develop a standard for control printers over a network. To achieve this a bi-directional standard was developed which was compatible with existing software. This standard was submitted to the IEEE and was published as the IEEE 1284-1994 Standard (as it was released in 1994 and was developed by the IEEE 1284 committee).

With this standard all parallel ports use a bi-directional link in either a compatible, nibble or byte mode. These modes are relatively slow, as the software must monitor the handshaking lines (and gives rates up to 100 Kbps). To allow high speed the EPP (Enhanced Parallel Port) and ECP (Extended Capabilities Port Protocol) modes have been developed to allow high-speed data transfer using automatic hardware handshaking. In addition to the previous three modes, EPP and ECP are being implemented on the latest I/O controllers by most of the Super I/O chip manufacturers. These modes use hardware to assist in the data transfer. For example, in EPP mode, a byte of data can be transferred to the peripheral by a simple OUT instruction and the I/O controller handles all the handshaking and data transfer to the peripheral.

### 17.2 Data handshaking

---

Figure 17.1 shows the pin connections on the PC connector. The data lines (D0–D7) output data from the PC and each of the data lines has an associated ground line (GND).

The main handshaking lines are  $\overline{\text{ACK}}$ ,  $\text{BUSY}$  and  $\overline{\text{STROBE}}$ . Initially the computer places the data on the data bus, then it sets the  $\overline{\text{STROBE}}$  line low to inform the external device that the data on the data bus is valid. When the external device has read the data it

sets the  $\overline{\text{ACK}}$  lines low to acknowledge that it has read the data. The PC then waits for the printer to set the BUSY line inactive, that is, low. Figure 17.2 shows a typical handshaking operation and Table 17.1 outlines the definitions of the pins.

The parallel interface can be accessed either by direct reads to and writes from the I/O memory addresses or from a program which uses the BIOS printer interrupt. This interrupt allows a program either to get the status of the printer or to write a character to it. Table 17.2 outlines the interrupt calls.

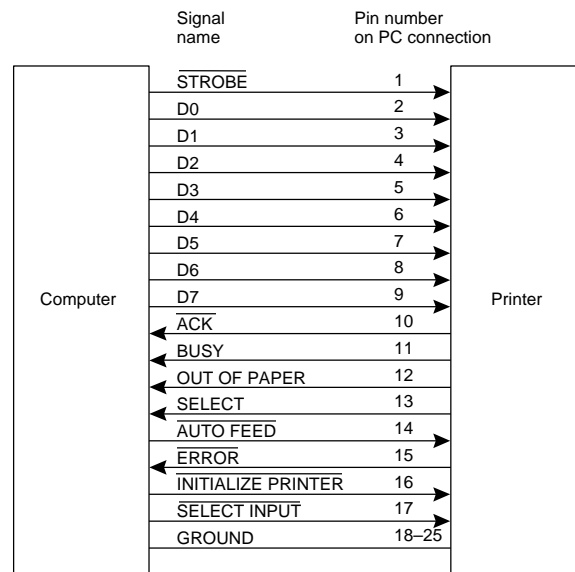


Figure 17.1 Centronics parallel interface showing pin numbers on PC connector.

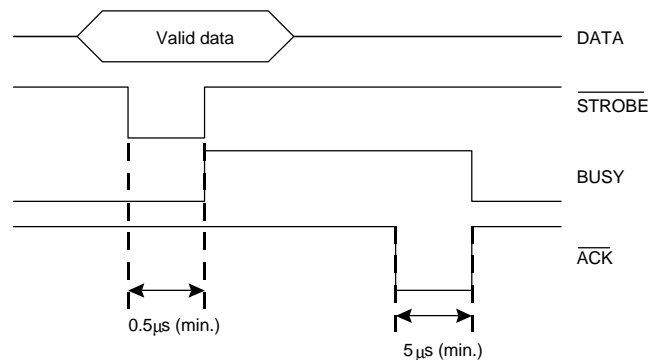


Figure 17.2 Data handshaking with the Centronics parallel printer interface.

### 17.2.1 BIOS printer

Program 17.1 uses the BIOS printer interrupt to test the status of the printer and outputs characters to the printer.

**Table 17.1** Signal definitions.

<i>Signal</i>	<i>In/out</i>	<i>Description</i>
$\overline{\text{STROBE}}$	Out	Indicates that valid data is on the data lines (active low)
$\overline{\text{AUTOFEED}}$	Out	Instructs the printer to insert a line feed for every carriage return (active low)
$\overline{\text{SELECT INPUT}}$	Out	Indicates to the printer that it is selected (active low)
INIT	Out	Resets the printer
$\overline{\text{ACK}}$	In	Indicate that the last character was received (active low)
BUSY	In	Indicates that the printer is busy and thus cannot accept data
OUT OF PAPER	In	Out of paper
SELECT	In	Indicates that the printer is on-line and connected
$\overline{\text{ERROR}}$	In	Indicates that an error exists (active low)

**Table 17.2** BIOS printer interrupt.

<i>Description</i>	<i>Input registers</i>	<i>Output registers</i>
Initialize printer port	AH = 01h DX = printer number (00h–02h)	AH = printer status bit 7: not busy bit 6: acknowledge bit 5: out of paper bit 4: selected bit 3: I/O error bit 2: unused bit 1: unused bit 0: timeout
Write character to printer	AH = 00h AL = character to write DX = printer number (00h–02h)	AH = printer status
Get printer status	AH = 02h DX = printer number (00h–02h)	AH = printer status

**Program 17.1**

```
#include <dos.h>
#include <stdio.h>
#include <conio.h>

#define PRINTERR -1

void print_character(int ch);
int init_printer(void);

int main(void)
```

```

{
int  status,ch;

status=init_printer();
if (status==PRINTERR) return(1);

do
{
printf("Enter character to output to printer");
ch=getch();
print_character(ch);
} while (ch!=4);
return(0);
}

int  init_printer(void)
{
union REGS inregs,outregs;

inregs.h.ah=0x01; /* initialize printer */
inregs.x.dx=0; /* LPT1: */
int86(0x17,&inregs,&outregs);
if (inregs.h.ah & 0x20)
{ puts("Out of paper"); return(PRINTERR); }
else if (inregs.h.ah & 0x08)
{ puts("I/O error"); return(PRINTERR); }
else if (inregs.h.ah & 0x01)
{ puts("Printer timeout"); return(PRINTERR); }

return(0);
}

void print_character(int ch)
{
union REGS inregs,outregs;

inregs.h.ah=0x00; /* print character */
inregs.x.dx=0; /* LPT1: */
inregs.h.al=ch;

int86(0x17,&inregs,&outregs);
}

```

## 17.3 I/O addressing

---

### 17.3.1 Addresses

The printer port has three I/O addresses assigned for the data, status and control ports. These addresses are normally assigned to:

Printer	Data register	Status register	Control register
LPT1	378h	379h	37ah
LPT2	278h	279h	27ah

The DOS debug program is used to display the base addresses for the serial and parallel ports by displaying the 32 memory location starting at 0040:0008. For example:

```

-d 40:00
0040:0000  F8 03 F8 02 00 00 00 00-78 03 00 00 00 00 29 02

```

The first four 16-bit addresses give the serial communications ports. In this case there are two COM ports at address 03F8h (COM1) and 02F8h (for COM2). The next four 16-bit addresses give the parallel port addressees. In this case, there are two parallel ports; one at 0378h (LPT1) and one at 0229h (LPT4).

### 17.3.2 Output lines

Figure 17.3 shows the bit definitions of the registers. The Data port register links to the output lines. Writing a 1 to the bit position in the port sets the output high, while a 0 sets the corresponding output line to a low. Thus to output the binary value 1010 1010b (AAh) to the parallel port data then using Borland C:

```
outportb(0x378,0xAA); /* in Visual C this is _outp(0x378,0xAA); */
```

The output data lines are each capable of sourcing 2.6 mA and sinking 24 mA; it is thus essential that the external device does not try to pull these lines to ground.

The Control port also contains five output lines, of which the lower four bits are  $\overline{\text{STROBE}}$ ,  $\overline{\text{AUTO FEED}}$ , INIT and  $\overline{\text{SELECT INPUT}}$ , as illustrated in Figure 17.3. These lines can be used as either control lines or as data outputs. With the data line, a 1 in the register gives an output high, while the lines in the Control port have inverted logic. Thus, a 1 to a bit in the register causes an output low.

Program 17.2 outputs the binary pattern 0101 0101b (55h) to the data lines and sets  $\overline{\text{SELECT INPUT}}=0$ , INIT=1,  $\overline{\text{AUTO FEED}}=1$ , and  $\overline{\text{STROBE}}=0$ , the value of the Data port will be 55h and the value written to the Control port will be XXXX 1101 (where X represents don't care). The value for the control output lines must be inverted, so that the  $\overline{\text{STROBE}}$  line will be set to a 1 so that it will be output as a LOW.

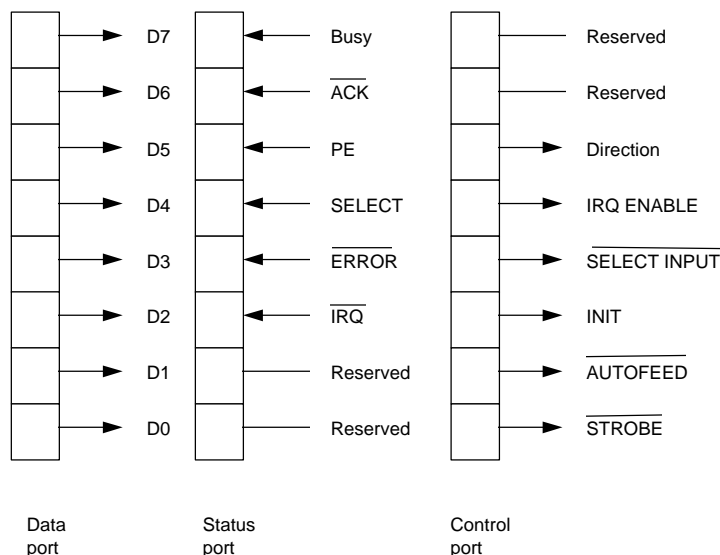



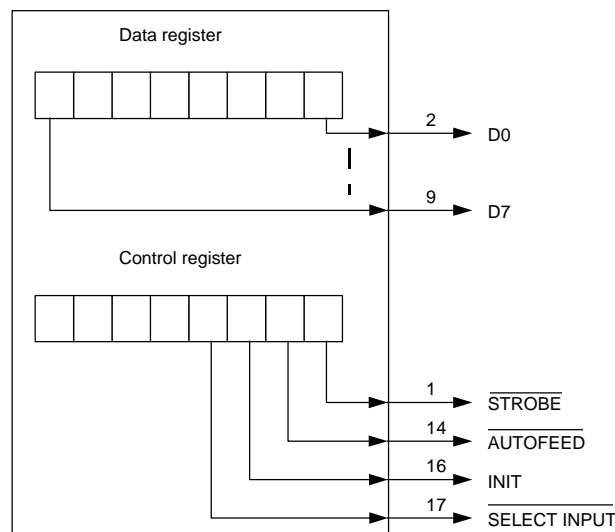
Figure 17.3 Port assignments.

 **Program 17.2**

```

#define DATA      0x378
#define STATUS     DATA+1
#define CONTROL    DATA+2
int main(void)
{
int out1,out2;
  out1 = 0x55;          /* 0101 0101 */
  outportb(DATA, out1);
  out2 = 0x0D;          /* 0000 1101 */
  outportb(CONTROL, out2); /* STROBE=LOW, AUTOFEED=HIGH, etc */
  return(0);
}

```



**Figure 17.4** Output lines.

The setting of the output value (in this case, `out2`) looks slightly confusing, as the output is the inverse of the logical setting (that is, a 1 sets the output low). An alternative method is to exclusive-OR (EX-OR) the output value with `$B` which inverts the 1st, 2nd and 4th least significant bits (`SELECT INPUT=0`, `AUTOFEED=1`, and `STROBE=0`), while leaving the 3rd least significant bit (`INIT`) untouched. Thus, the following will achieve the same as the previous program:

```

out2 = 0x06;          /* 0000 0110 */
outportb(CONTROL, out2 ^ 0xb); /* STROBE=LOW, AUTOFEED=HIGH, etc */

```

If the 5th bit on the control register (IRQ Enable) is written as 1 then the output on this line will go from a high to a low which will cause the processor to be interrupted.

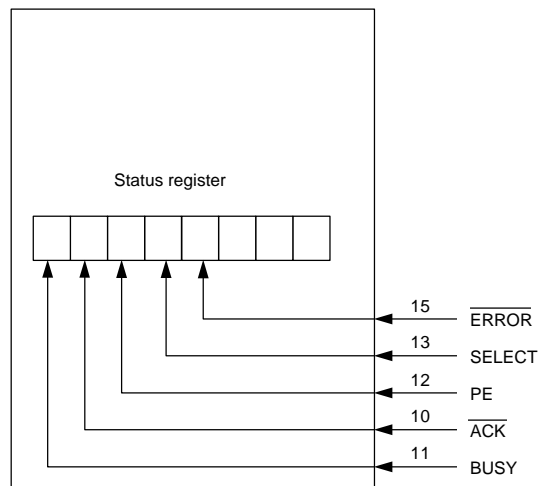
The control lines are driven by open collector drivers pulled to +5 Vdc through 4.7 k $\Omega$  resistors. Each can sink approximately 7 mA and maintain 0.8 V down-level.

### 17.3.3 Inputs

There are five inputs from the parallel port (`BUSY`, `ACK`, `PE`, `SELECT` and `ERROR`).

The status of these lines can be found by simply reading the upper 5 bits of the Status register, as illustrated in Figure 17.5.

Unfortunately, the BUSY line has an inverted status. Thus when a LOW is present on BUSY, the bit will actually be read as a 1. For example, Program 17.3 reads the bits from the Status register, inverts the BUSY bit and then shifts the bits three places to the right so that the 5 inputs bits are in the 5 least significant bits.



**Figure 17.5** Input lines.



### Program 17.3

```
#include <stdio.h>
#define DATA 0x378
#define STATUS DATA+1
int main(void)
{
    unsigned int in1;

    in1 = inportb(STATUS); /* read from status register */

    in1 = in1 ^ 0x80 /* invert BUSY bit */
    in1 = in1 >> 3; /* move bits so that the inputs are the least
                    significant bits */

    printf("Status bits are %d\n",in1);
    return(0);
}
```

#### 17.3.4 Electrical interfacing

The output lines can be used to drive LEDs. Figure 17.6 shows an example circuit where a LOW output will cause the LED to be ON while a HIGH causes the output to be OFF. For an input an open push button causes a HIGH input on the input.

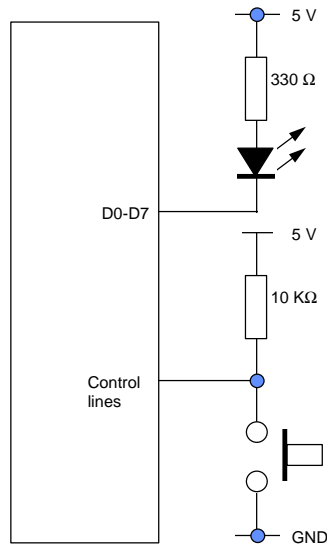


Figure 17.6 Interfacing to inputs and outputs.

### 17.3.5 Simple example

Program 17.4 uses a push button connected to pin 11 (BUSY). When the button is open then the input to BUSY will be a HIGH and the most significant bit in the status register will thus be a 0 (as the BUSY signal is inverted). When the button is closed then this bit will be a 1. This is tested with:

```
if (in1&0x80)==1)
```

When this condition is TRUE (that is, when the button is closed) then the output data lines (D0–D7) will flash on and off with a delay of 1 second between flashes. An output of all 1s to the data lines causes the LEDs to be off and all 0s cause the LEDs to be on.



#### Program 17.4

```
/* Flash LEDs on and off when the push button connected to BUSY */
/* is closed */
#include <stdio.h>
#include <dos.h>

#define DATA 0x378
#define STATUS DATA+1
#define CONTROL DATA+2

int main(void)
{
int in1;
do
{
in1 = inportb(STATUS);

if (in1&0x80)==1) /* if switch closed this is TRUE */
{
outportb(DATA,0x00); /* LEDs on */
```

```

        delay(1000);
        outportb(DATA, 0xff); /* LEDs off */
        delay(1000);
    }
    else
        outportb(DATA,0x01); /* switch open */
    } while (!kbhit());
    return(0);
}

```

## 17.4 Exercises

---

- 17.4.1** Write a program that sends a ‘walking-ones’ code to the parallel port. The delay between changes should be 1 second. A ‘walking-ones’ code is as follows:

```

00000001
00000010
00000100
00001000
:
:
10000000
00000001
00000010
and so on.

```

*Hint:* Use a `do...while` loop with either the shift left operators (`<<`) or output the values `0x01`, `0x02`, `0x04`, `0x08`, `0x10`, `0x20`, `0x40`, `0x80`, `0x01`, `0x02`, and so on.

- 17.4.2** Write separate programs which output the patterns in (a) and (b). The sequences are as follows:

(a)	00000001	(b)	10000001
	00000010		01000010
	00000100		00100100
	00001000		00011000
	00010000		00100100
	00100000		01000010
	01000000		10000001
	10000000		01000010
	01000000		00100100
	00100000		00011000
	00010000		00100100
	::		<i>and so on.</i>
	00000001		
	00000010		
	<i>and so on.</i>		

- 17.4.3** Write separate programs which output the following sequences:

(a)	1010 1010	(b)	1111 1111
	0101 0101		0000 0000
	1010 1010		1111 1111
	0101 0101		0000 0000
	<i>and so on.</i>		<i>and so on.</i>

```
(c) 0000 0001      (d) 0000 0001
    0000 0011      0000 0011
    0000 1111      0000 0111
    0001 1111      0000 1111
    0011 1111      0001 1111
    0111 1111      0011 1111
    1111 1111      0111 1111
    0000 0001      1111 1111
    0000 0011      0111 1111
    0000 0111      0011 1111
    0000 1111      0001 1111
    0001 1111      0000 1111
    and so on.      and so on.
```

(e) The inverse of (d) above.

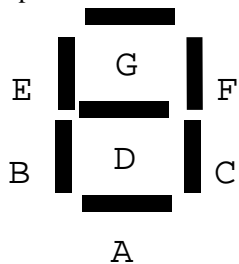
**17.4.4** Binary coded decimal (BCD) is used mainly in decimal displays and is equivalent to the decimal system where a 4-bit code represents each decimal number. The first 4 bits represent the units, the next 4 the tens, and so on. Write a program that outputs to the parallel port a BCD sequence with a 1-second delay between changes. A sample BCD table is given in Table 17.3. The output should count from 0 to 99.

*Hint:* One possible implementation is to use two variables to represent the units and tens. These would then be used in a nested loop. The resultant output value will then be  $(tens \ll 4) + units$ . An outline of the loop code is given next.

```
for (tens=0;tens<10;tens++)
  for (units=0;units<10;units++)
  {
  }
}
```

**17.4.5** Write a program which interfaces to an 7-segment display and displays an incremented value every second. Each of the segments should be driven from one of the data lines on the parallel port. For example:

Value	Segment							Hex Value
	A	B	C	D	E	F	G	
0	1	1	1	0	1	1	1	77h
1	0	0	1	0	0	1	0	12h
2	1	1	0	1	0	1	1	6Bh
:				:	:			
9	0	0	1	1	1	1	1	1Fh



Two ways of implementing this is either to determine the logic for each segment or to have a basic lookup table, such as:

```
int seg_val[8]={0x77, 0x12, 0x6B, ... 0x1F};

val=seg_val(count % 10);
/* mask-off the least-significant digit */
outportb(0x378,seg_val[val]);
```

**Table 17.3** BCD conversion.

<i>Digit</i>	<i>BCD</i>
00	00000000
01	00000001
02	00000010
03	00000011
04	00000100
05	00000101
06	00000110
07	00000111
08	00001000
09	00001001
10	00010000
11	00010001
.	.
.	.
.	.
97	10010111
98	10011000
99	10011001