The original IBM PC design provided support for three parallel printer ports that IBM designated LPT1:, LPT2:, and LPT3:\(^1\). IBM probably envisioned machines that could support a standard dot matrix printer, a daisy wheel printer, and maybe some other auxiliary type of printer for different purposes, all on the same machine (laser printers were still a few years in the future at that time). Surely IBM did not anticipate the general use that parallel ports have received or they would probably have designed them differently. Today, the PC's parallel port controls keyboards, disk drives, tape drives, SCSI adapters, ethernet (and other network) adapters, joystick adapters, auxiliary keypad devices, other miscellaneous devices, and, oh yes, printers. This chapter will not attempt to describe how to use the parallel port for all these various purposes - this book is long enough already. However, a thorough discussion of how the parallel interface controls a printer and one other application of the parallel port (cross machine communication) should provide you with enough ideas to implement the next great parallel device.

21.1 Basic Parallel Port Information

There are two basic data transmission methods modern computers employ: parallel data transmission and serial data transmission. In a serial data transmission scheme (see "The PC Serial Ports" on page 1223) one device sends data to another a single bit at a time across one wire. In a parallel transmission scheme, one device sends data to another several bits at a time (in parallel) on several different wires. For example, the PC's parallel port provides eight data lines compared to the serial port's single data line. Therefore, it would seem that the parallel port would be able to transmit data eight times as fast since there are eight times as many wires in the cable. Likewise, it would seem that a serial cable, for the same price as a parallel cable, would be able to go eight times as far since there are fewer wires in the cable. And these are the common trade-offs typically given for parallel vs. serial communication methods: speed vs. cost.

In practice, parallel communications is not eight times faster than serial communications, nor do parallel cables cost eight times as much. In generally, those who design serial cables (.e.g, ethernet cables) use higher materials and shielding. This raises the cost of the cable, but allows devices to transmit data, still a bit at a time, much faster. Furthermore, the better cable design allows greater distances between devices. Parallel cables, on the other hand, are generally quite inexpensive and designed for very short connections (generally no more than about six to ten feet). The real world problems of electrical noise and cross-talk create problems when using long parallel cables and limit how fast the system can transmit data. In fact the original Centronics printer port specification called for no more than 1,000 characters/second data transmission rate, so many printers were designed to handle data at this transmission rate. Most parallel ports can easily outperform this value; however, the limiting factor is still the cable, not any intrinsic limitation in a modern computer.

Although a parallel communication system could use any number of wires to transmit data, most parallel systems use eight data lines to transmit a byte at a time. There are a few notable exceptions. For example, the SCSI interface is a parallel interface, yet newer versions of the SCSI standard allow eight, sixteen, and even thirty-two bit data transfers. In this chapter we will concentrate on byte-sized transfers since the parallel port on the PC provides for eight-bit data.

A typical parallel communication system can be one way (or unidirectional) or two way (bidirectional). The PC's parallel port generally supports unidirectional communications (from the PC to the printer), so we will consider this simpler case first.

In a unidirectional parallel communication system there are two distinguished sites: the transmitting site and the receiving site. The transmitting site places its data on the data lines and informs the receiving site that data is available; the receiving site then reads the data lines and informs the transmitting site that it

---

1. In theory, the BIOS allows for a fourth parallel printer port, LPT4:, but few (if any) adapter cards have ever been built that claim to work as LPT4.
has taken the data. Note how the two sites synchronize their access to the data lines - the receiving site does not read the data lines until the transmitting site tells it to, the transmitting site does not place a new value on the data lines until the receiving site removes the data and tells the transmitting site that it has the data. Handshaking is the term that describes how these two sites coordinate the data transfer.

To properly implement handshaking requires two additional lines. The strobe (or data strobe) line is what the transmitting site uses to tell the receiving site that data is available. The acknowledge line is what the receiving site uses to tell the transmitting site that it has taken the data and is ready for more. The PC’s parallel port actually provides a third handshaking line, busy, that the receiving site can use to tell the transmitting site that it is busy and the transmitting site should not attempt to send data. A typical data transmission session looks something like the following:

Transmitting site:
1) The transmitting site checks the busy line to see if the receiving is busy. If the busy line is active, the transmitter waits in a loop until the busy line becomes inactive.
2) The transmitting site places its data on the data lines.
3) The transmitting site activates the strobe line.
4) The transmitting site waits in a loop for the acknowledge line to become active.
5) The transmitting site sets the strobe inactive.
6) The transmitting site waits in a loop for the acknowledge line to become inactive.
7) The transmitting site repeats steps one through six for each byte it must transmit.

Receiving site:
1) The receiving site sets the busy line inactive (assuming it is ready to accept data).
2) The receiving site waits in a loop until the strobe line becomes active.
3) The receiving site reads the data from the data lines (and processes the data, if necessary).
4) The receiving site activates the acknowledge line.
5) The receiving site waits in a loop until the strobe line goes inactive.
6) The receiving site sets the acknowledge line inactive.
7) The receiving site repeats steps one through six for each additional byte it must receive.

By carefully following these steps, the receiving and transmitting sites carefully coordinate their actions so the transmitting site doesn’t attempt to put several bytes on the data lines before the receiving site consumes them and the receiving site doesn’t attempt to read data that the transmitting site has not sent.

Bidirectional data transmission is often nothing more than two unidirectional data transfers with the roles of the transmitting and receiving sites reversed for the second communication channel. Some PC parallel ports (particularly on PS/2 systems and many notebooks) provide a bidirectional parallel port. Bidirectional data transmission on such hardware is slightly more complex than on systems that implement bidirectional communication with two unidirectional ports. Bidirectional communication on a bidirectional parallel port requires an extra set of control lines so the two sites can determine who is writing to the common data lines at any one time.
21.2 The Parallel Port Hardware

The standard unidirectional parallel port on the PC provides more than the 11 lines described in the previous section (eight data, three handshake). The PC’s parallel port provides the following signals:

Table 79: Parallel Port Signals

<table>
<thead>
<tr>
<th>Pin Number on Connector</th>
<th>I/O Direction</th>
<th>Active Polarity</th>
<th>Signal Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>output</td>
<td>0</td>
<td>Strobe (data available signal).</td>
</tr>
<tr>
<td>2-9</td>
<td>output</td>
<td>-</td>
<td>Data lines (bit 0 is pin 2, bit 7 is pin 9).</td>
</tr>
<tr>
<td>10</td>
<td>input</td>
<td>0</td>
<td>Acknowledge line (active when remote system has taken data).</td>
</tr>
<tr>
<td>11</td>
<td>input</td>
<td>0</td>
<td>Busy line (when active, remote system is busy and cannot accept data).</td>
</tr>
<tr>
<td>12</td>
<td>input</td>
<td>1</td>
<td>Out of paper (when active, printer is out of paper).</td>
</tr>
<tr>
<td>13</td>
<td>input</td>
<td>1</td>
<td>Select. When active, the printer is selected.</td>
</tr>
<tr>
<td>14</td>
<td>output</td>
<td>0</td>
<td>Autofeed. When active, the printer automatically inserts a line feed after every carriage return it receives.</td>
</tr>
<tr>
<td>15</td>
<td>input</td>
<td>0</td>
<td>Error. When active, there is a printer error.</td>
</tr>
<tr>
<td>16</td>
<td>output</td>
<td>0</td>
<td>Init. When held active for at least 50 µsec, this signal causes the printer to initialize itself.</td>
</tr>
<tr>
<td>17</td>
<td>output</td>
<td>0</td>
<td>Select input. This signal, when inactive, forces the printer off-line</td>
</tr>
<tr>
<td>18-25</td>
<td>-</td>
<td>-</td>
<td>Signal ground.</td>
</tr>
</tbody>
</table>

Note that the parallel port provides 12 output lines (eight data lines, strobe, autofeed, init, and select input) and five input lines (acknowledge, busy, out of paper, select, and error). Even though the port is unidirectional, there is a good mixture of input and output lines available on the port. Many devices (like disk and tape drives) that require bidirectional data transfer use these extra lines to perform bidirectional data transfer.

On bidirectional parallel ports (found on PS/2 and laptop systems), the strobe and data lines are both input and output lines. There is a bit in a control register associated with the parallel port that selects the transfer direction at any one given instant (you cannot transfer data in both direction simultaneously).

There are three I/O addresses associated with a typical PC compatible parallel port. These addresses belong to the data register, the status register, and the control register. The data register is an eight-bit read/write port. Reading the data register (in a unidirectional mode) returns the value last written to the data register. The control and status registers provide the interface to the other I/O lines. The organization of these ports is as follows:

```
7 6 5 4 3 2 1 0
```

Unused
Printer ackon PS/2 systems (active if zero)
Device error (active if zero)
Device selected (selected if one)
Device out of paper (out of paper if one)
Printer acknowledge (ack if zero)
Printer busy (busy if zero)

Parallel Port Status Register (read only)
Bit two (printer acknowledge) is available only on PS/2 and other systems that support a bidirectional printer port. Other systems do not use this bit.

The parallel port control register is an output register. Reading this location returns the last value written to the control register except for bit five that is write only. Bit five, the data direction bit, is available only on PS/2 and other systems that support a bidirectional parallel port. If you write a zero to this bit, the strobe and data lines are output bits, just like on the unidirectional parallel port. If you write a one to this bit, then the data and strobe lines are inputs. Note that in the input mode (bit 5 = 1), bit zero of the control register is actually an input. Note: writing a one to bit four of the control register enables the printer IRQ (IRQ 7). However, this feature does not work on all systems so very few programs attempt to use interrupts with the parallel port. When active, the parallel port will generate an int 0Fh whenever the printer acknowledges a data transmission.

Since the PC supports up to three separate parallel ports, there could be as many as three sets of these parallel port registers in the system at any one time. There are three parallel port base addresses associated with the three possible parallel ports: 3BCh, 378h, and 278h. We will refer to these as the base addresses for LPT1:, LPT2:, and LPT3:, respectively. The parallel port data register is always located at the base address for a parallel port, the status register appears at the base address plus one, and the control register appears at the base address plus two. For example, for LPT1:, the data register is at I/O address 3BCh, the status register is at I/O address 3BDh, and the control register is at I/O address 3BEh.

There is one minor glitch. The I/O addresses for LPT1:, LPT2:, and LPT3: given above are the physical addresses for the parallel ports. The BIOS provides logical addresses for these parallel ports as well. This lets users remap their printers (since most software only writes to LPT1:). To accomplish this, the BIOS reserves eight bytes in the BIOS variable space (40:8, 40:10, 40:12, and 40:14). Location 40:8 contains the base address for logical LPT1:, location 40:10 contains the base address for logical LPT2:, etc. When software accesses LPT1:, LPT2:, etc., it generally accesses the parallel port whose base address appears in one of these locations.

### 21.3 Controlling a Printer Through the Parallel Port

Although there are many devices that connect to the PC’s parallel port, printers still make up the vast number of such connections. Therefore, describing how to control a printer from the PC’s parallel port is probably the best first example to present. As with the keyboard, your software can operate at three different levels: it can print data using DOS, using BIOS, or by writing directly to the parallel port hardware. As with the keyboard interface, using DOS or BIOS is the best approach if you want to maintain compatibility with other devices that plug into the parallel port.

---

2. Many devices connect to the parallel port with a pass-through plug allowing you to use that device and still use the parallel port for your printer. However, if you talk directly to the parallel port with your software, it may conflict with that device’s operation.
device, going directly to the hardware is your only choice. However, the BIOS provides good printer support, so going directly to the hardware is rarely necessary if you simply want to send data to the printer.

### 21.3.1 Printing via DOS

MS-DOS provides two calls you can use to send data to the printer. DOS function 05h writes the character in the d1 register directly to the printer. Function 40h, with a file handle of 04h, also sends data to the printer. Since the chapter on DOS and BIOS fully describes these functions, we will not discuss them any further here. For more information, see “MS-DOS, PC-BIOS, and File I/O” on page 699.

### 21.3.2 Printing via BIOS

Although DOS provides a reasonable set of functions to send characters to the printer, it does not provide functions to let you initialize the printer or obtain the current printer status. Furthermore, DOS only prints to LPT1. The PC’s int 17h BIOS routine provides three functions, print, initialize, and status. You can apply these functions to any supported parallel port on the system. The print function is roughly equivalent to DOS’ print character function. The initialize function initializes the printer using system dependent timing information. The printer status returns the information from the printer status port along with time-out information. For more information on these routines, see “MS-DOS, PC-BIOS, and File I/O” on page 699.

### 21.3.3 An INT 17h Interrupt Service Routine

Perhaps the best way to see how the BIOS functions operate is to write a replacement int 17h ISR for a printer. This section explains the handshaking protocol and variables the printer driver uses. It also describes the operation and return results associated with each machine.

There are eight variables in the BIOS variable space (segment 40h) the printer driver uses. The following table describes each of these variables:

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>40:08</td>
<td>Base address of LPT1: device.</td>
</tr>
<tr>
<td>40:0A</td>
<td>Base address of LPT2: device.</td>
</tr>
<tr>
<td>40:0C</td>
<td>Base address of LPT3: device.</td>
</tr>
<tr>
<td>40:0E</td>
<td>Base address of LPT4: device.</td>
</tr>
<tr>
<td>40:78</td>
<td>LPT1: time-out value. The printer port driver software should return an error if the printer device does not respond in a reasonable amount of time. This variable (if non-zero) determines how many loops of 65,536 iterations each a driver will wait for a printer acknowledge. If zero, the driver will wait forever.</td>
</tr>
<tr>
<td>40:79</td>
<td>LPT2: time-out value. See description above.</td>
</tr>
<tr>
<td>40:7B</td>
<td>LPT4: time-out value. See description above.</td>
</tr>
</tbody>
</table>

You will notice a slight deviation in the handshake protocol in the following code. This printer driver does not wait for an acknowledge from the printer after sending a character. Instead, it checks to see if
the printer has sent an acknowledge to the previous character before sending a character. This saves a small amount of time because the program printer then characters can continue to operating in parallel with the receipt of the acknowledge from the printer. You will also notice that this particular driver does not monitor the busy lines. Almost every printer in existence leaves this line inactive (not busy), so there is no need to check it. If you encounter a printer than does manipulate the busy line, the modification to this code is trivial. The following code implements the int 17h service:

; INT17.ASM
;
; A short passive TSR that replaces the BIOS' int 17h handler.
; This routine demonstrates the function of each of the int 17h
; functions that a standard BIOS would provide.
;
; Note that this code does not patch into int 2Fh (multiplex interrupt)
; nor can you remove this code from memory except by rebooting.
; If you want to be able to do these two things (as well as check for
; a previous installation), see the chapter on resident programs. Such
; code was omitted from this program because of length constraints.
;
; cseg and EndResident must occur before the standard library segments!

cseg segment para public 'code'
cseg ends

; Marker segment, to find the end of the resident section.
EndResident segment para public 'Resident'
EndResident ends

.xlist
include stdlib.a
includelib stdlib.lib
.list

byp equ <byte ptr>
cseg segment para public 'code'
assume cs:cseg, ds:cseg

OldInt17 dword ?

; BIOS variables:
PrtrBase equ 8
PrtrTimeOut equ 78h

; This code handles the INT 17H operation. INT 17H is the BIOS routine
; to send data to the printer and report on the printer's status. There
; are three different calls to this routine, depending on the contents
; of the AH register. The DX register contains the printer port number.
;
; DX=0 -- Use LPT1:
; DX=1 -- Use LPT2:
; DX=2 -- Use LPT3:
; DX=3 -- Use LPT4:
;
; AH=0 -- Print the character in AL to the printer. Printer status is
; returned in AH. If bit #0 = 1 then a timeout error occurred.
;
; AH=1 -- Initialize printer. Status is returned in AH.
;
; AH=2 -- Return printer status in AH.
;
; The status bits returned in AH are as follows:
### Bit Function

<table>
<thead>
<tr>
<th>Bit</th>
<th>Function</th>
<th>Non-error values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1=time out error</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>unused</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>unused</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>1=I/O error</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1=selected, 0=deselected.</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1=out of paper</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1=acknowledge</td>
<td>x</td>
</tr>
<tr>
<td>7</td>
<td>1=not busy</td>
<td>x</td>
</tr>
</tbody>
</table>

Note that the hardware returns bit 3 with zero if an error has occurred, with one if there is no error. The software normally inverts this bit before returning it to the caller.

### Printer port hardware locations:

There are three ports used by the printer hardware:

- **PrtrPortAdrs** --- Output port where data is sent to printer (8 bits).
- **PrtrPortAdrs+1** --- Input port where printer status can be read (8 bits).
- **PrtrPortAdrs+2** --- Output port where control information is sent to the printer.

#### Data output port
8-bit data is transmitted to the printer via this port.

#### Input status port:

- bit 0: unused.
- bit 1: unused.
- bit 2: unused.

- bit 3: -Error, normally this bit means that the printer has encountered an error. However, with the P101 installed this is a data return line for the keyboard scan.

- bit 4: +SLCT, normally this bit is used to determine if the printer is selected or not. With the P101 installed this is a data return line for the keyboard scan.

- bit 5: +PE, a 1 in this bit location means that the printer has detected the end of paper. On many printer ports, this bit has been found to be inoperative.

- bit 6: -ACK, A zero in this bit position means that the printer has accepted the last character and is ready to accept another. This bit is not normally used by the BIOS as bit 7 also provides this function (and more).

- bit 7: -Busy, When this signal is active (0) the printer is busy and cannot accept data. When this bit is set to one, the printer can accept another character.

### Output control port:

- **Bit 0**: +Strobe, A 0.5 us (minimum) active high pulse on this bit clocks the data latched into the printer data output port to the printer.

- **Bit 1**: +Auto FD XT - A 1 stored at this bit causes the printer to line feed after a line is printed. On some printer interfaces (e.g., the Hercules Graphics Card) this bit is inoperative.

- **Bit 2**: -INIT, a zero on this bit (for a minimum of 50 us) will cause the printer to (re)init-
Chapter 21

; Initialize itself.

; Bit 3: +SLCT IN, a one in this bit selects the printer. A zero will cause the printer to go off-line.

; Bit 4: +IRQ ENABLE, a one in this bit position allows an interrupt to occur when -ACK changes from one to zero.

; Bit 5: Direction control on BI-DIR port. 0=output, 1=input.

; Bit 6: reserved, must be zero.

; Bit 7: reserved, must be zero.

MyInt17 proc far
assume ds:nothing
push ds
push bx
push cx
push dx
mov bx, 40h ;Point DS at BIOS vars.
mov ds, bx
cmp dx, 3 ;Must be LPT1..LPT4.
ja InvalidPrtr
cmp ah, 0 ;Branch to the appropriate code for
jz PrtChar ; the printer function
cmp ah, 2
jb PrtrInit
je PrtrStatus

; If they passed us an opcode we don’t know about, just return.

InvalidPrtr: jmp ISR17Done

; Initialize the printer by pulsing the init line for at least 50 us.
; The delay loop below will delay well beyond 50 usec even on the fastest machines.

PrtrInit: mov bx, dx ;Get printer port value.
         shl bx, 1 ;Convert to byte index.
mov dx, PrtrBase[ bx] ;Get printer base address.
test dx, dx ;Does this printer exist?
je InvalidPrtr ;Quit if no such printer.
add dx, 2 ;Point dx at control reg.
in al, dx ;Read current status.
and al, 11011011b ;Clear INIT/BIDIR bits.
out dx, al ;Reset printer.
mov cx, 0 ;This will produce at least
                     ; a 50 usec delay.
PIDelay: loop PIDelay ;Stop resetting printer.
or al, 100b
out dx, al
jmp ISR17Done

; Return the current printer status. This code reads the printer status port and formats the bits for return to the calling code.

PrtrStatus: mov bx, dx ;Get printer port value.
         shl bx, 1 ;Convert to byte index.
mov dx, PrtrBase[ bx] ;Base address of printer port.
mov al, 00101001b ;Dflt: every possible error.
test dx, dx ;Does this printer exist?
je InvalidPrtr ;Quit if no such printer.
in dx ;Point at status port.
in al, dx ;Read status port.
and al, 11111000b ;Clear unused/timeout bits.
jmp ISR17Done
; Print the character in the accumulator!

PrtChar:    mov    bx, dx
            mov    cl, PrtrTimeOut[bx] ;Get time out value.
            shl    bx, 1 ;Convert to byte index.
            mov    dx, PrtrBase[bx] ;Get Printer port address
            or     dx, dx ;Non-nil pointer?
            jz     NoPrtr2 ; Branch if a nil ptr

; The following code checks to see if an acknowlege was received from
; the printer. If this code waits too long, a time-out error is returned.
; Acknowlege is supplied in bit #7 of the printer status port (which is
; the next address after the printer data port).

            push   ax
            inc    dx ;Point at status port
            mov    bl, cl ;Put timeout value in bl
            mov    bh, cl ; and bh.
            WaitLp1: xor    cx, cx ;Init count to 65536.
            WaitLp2: in     al, dx ;Read status port
                        mov    ah, al ;Save status for now.
                        test   al, 80h ;Printer acknowledge?
                        jnz    GotAck ;Branch if acknowledge.
                        loop   WaitLp2 ;Repeat 65536 times.
                        dec    bl ;Decrement time out value.
                        jnz    WaitLp1 ;Repeat 65536*TimeOut times.
            got Ack: mov    cx, 16 ;Short delay if crazy prtr
                        GALp:  loop   GALp
                        pop     ax
                        push    ax
                        dec     dx
                        pushf
                        cli
                        out     dx, al ;Output data to the printer.
                        ; The following short delay gives the data time to travel through the
                        ; parallel lines. This makes sure the data arrives at the printer before
                        ; the strobe (the times can vary depending upon the capacitance of the
                        ; parallel cable’s lines).
            DataSettleLp: loop  DataSettleLp
            jmp     ISR17Done

; See if the user has selected no timeout:
            cmp     bh, 0
            je     WaitLp1

; TIMEOUT ERROR HAS OCCURRED!

; A timeout - I/O error is returned to the system at this point.
; Either we fall through to this point from above (time out error) or
; the referenced printer port doesn’t exist. In any case, return an error.

            NoPrtr2: or     ah, 9 ;Set timeout-I/O error flags
                        and    ah, 0F9h ;Turn off unused flags.
                        xor     ah, 40h ;Flip busy bit.

; Okay, restore registers and return to caller.

            pop     cx ;Remove old ax.
            mov     al, cl ;Restore old al.
            jmp     ISR17Done

; If the printer port exists and we’ve received an acknowlege, then it’s
; okay to transmit data to the printer. That job is handled down here.

            GotAck: mov     cx, 16 ;Give data time to settle
                        GALp:  loop   GALp
                        pop     ax
                        push    ax
                        dec     dx
                        pushf
                        cli
                        out     dx, al ;Output data to the printer.
                        ; Now that the data has been latched on the printer data output port, a
                        ; strobe must be sent to the printer. The strobe line is connected to
Chapter 21

; bit zero of the control port. Also note that this clears bit 5 of the 
; control port. This ensures that the port continues to operate as an 
; output port if it is a bidirectional device. This code also clears bits 
; six and seven which IBM claims should be left zero.

inc dx ; Point DX at the printer
inc dx ; control output port.
in al, dx ; Get current control bits.
and al, 01eh ; Force strobe line to zero and
out dx, al ; make sure it’s an output port.

mov cx, 16 ; Short delay to allow data
Delay0: loop Delay0 ; to become good.
or al, 1 ; Send out the (+) strobe.
out dx, al ; Output (+) strobe to bit 0
mov cx, 16 ; Short delay to lengthen strobe
StrobeDelay: loop StrobeDelay
and al, 0FEh ; Clear the strobe bit.
out dx, al ; Output to control port.
popf ; Restore interrupts.
pop dx ; Get old AX value
mov al, dl ; Restore old AL value
ISR17Done: pop dx
pop cx
pop bx
pop ds
iret
MyInt17 endp

Main proc

mov ax, cseg
mov ds, ax

print
byte "INT 17h Replacement",cr,lf
byte "Installing....",cr,lf,0

; Patch into the INT 17 interrupt vector. Note that the 
; statements above have made cseg the current data segment, 
; so we can store the old INT 17 value directly into 
; the OldInt17 variable.

cli ; Turn off interrupts!
mov ax, 0
mov es, ax
mov ax, es:[17h*4]
mov word ptr OldInt17, ax
mov ax, es:[17h*4 + 2]
mov word ptr OldInt17+2, ax
mov es:[17h*4], offset MyInt17
mov es:[17h*4+2], cs
sti ; Okay, ints back on.

; We’re hooked up, the only thing that remains is to terminate and 
; stay resident.

print
byte "Installed.",cr,lf,0
mov ah, 62h ; Get this program’s PSP
ti 21h ; value.
mov dx, EndResident; Compute size of program.
sub dx, bx
mov ax, 3100h ; DOS TSR command.
21.4 Inter-Computer Communications on the Parallel Port

Although printing is, by far, the most popular use for the parallel port on a PC, many devices use the parallel port for other purposes, as mentioned earlier. It would not be fitting to close this chapter without at least one example of a non-printer application for the parallel port. This section will describe how to get two computers to transmit files from one to the other across the parallel port.

The Laplink™ program from Travelling Software is a good example of a commercial product that can transfer data across the PC’s parallel port; although the following software is not as robust or feature laden as Laplink, it does demonstrate the basic principles behind such software.

Note that you cannot connect two computer’s parallel ports with a simple cable that has DB25 connectors at each end. In fact, doing so could damage the computers’ parallel ports because you’d be connecting digital outputs to digital outputs (a real no-no). However, you purchase “Laplink compatible” cables (or buy real Laplink cables for that matter) that provide proper connections between the parallel ports of two computers. As you may recall from the section on the parallel port hardware, the unidirectional parallel port provides five input signals. A Laplink compatible cable routes four of the data lines to four of these input lines in both directions. The connections on a Laplink compatible cable are as follows:

<table>
<thead>
<tr>
<th>Transmitting Site</th>
<th>Receiving Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data bit 4</td>
<td>Busy (inverted)</td>
</tr>
<tr>
<td>Data bit 3</td>
<td>Acknowledge</td>
</tr>
<tr>
<td>Data bit 2</td>
<td>Paper Empty</td>
</tr>
<tr>
<td>Data bit 1</td>
<td>Select</td>
</tr>
<tr>
<td>Data bit 0</td>
<td>Error</td>
</tr>
</tbody>
</table>

Connections on a Laplink Compatible Cable

Data written on bits zero through three of the data register at the transmitting site appear, unchanged, on bits three through six of the status port on the receiving site. Bit four of the transmitting site appears, inverted, at bit seven of the receiving site. Note that Laplink compatible cables are bidirectional. That is, you can transmit data from either site to the other using the connections above. However, since there are only five input bits on the parallel port, you must transfer the data four bits at a time (we need one bit for the data strobe). Since the receiving site needs to acknowledge data transmissions, we cannot simultaneously transmit data in both directions. We must use one of the output lines at the site receiving data to acknowledge the incoming data.
Since the two sites cooperating in a data transfer across the parallel cable must take turns transmitting and receiving data, we must develop a protocol so each participant in the data transfer knows when it is okay to transmit and receive. Our protocol will be very simple - a site is either a transmitter or a receiver, the roles will never switch. Designing a more complex protocol is not difficult, but this simple protocol will suffice for the example you are about to see. Later in this section we will discuss ways to develop a protocol that allows two-way transmissions.

The following example programs will transmit and receive a single file across the parallel port. To use this software, you run the transmit program on the transmitting site and the receive program on the receiving site. The transmit program fetches a file name from the DOS command line and opens that file for reading (generating an error, and quitting, if the file does not exist). Assuming the file exists, the transmit program then queries the receiving site to see if it is available. The transmitter checks for the presence of the receiving site by alternately writing zeros and ones to all output bits then reading its input bits. The receiving site will invert these values and write them back when it comes on-line. Note that the order of execution (transmitter first or receiver first) does not matter. The two programs will attempt to handshake until the other comes on-line. When both sites cycle through the inverting values three times, they write the value 05h to their output ports to tell the other site they are ready to proceed. A time-out function aborts either program if the other site does not respond in a reasonable amount of time.

Once the two sites are synchronized, the transmitting site determines the size of the file and then transmits the file name and size to the receiving site. The receiving site then begins waiting for the receipt of data.

The transmitting site sends the data 512 bytes at a time to the receiving site. After the transmission of 512 bytes, the receiving site delays sending an acknowledgment and writes the 512 bytes of data to the disk. Then the receiving site sends the acknowledge and the transmitting site begins sending the next 512 bytes. This process repeats until the receiving site has accepted all the bytes from the file.

Here is the code for the transmitter:

```assembly
; TRANSMIT.ASM
;
; This program is the transmitter portion of the programs that transmit files across a Laplink compatible parallel cable.
;
; This program assumes that the user want to use LPT1: for transmission.
; Adjust the equates, or read the port from the command line if this
; is inappropriate.

.286
.xlist
.include stdlib.a
.include stdlib.lib
.list

dseg segment para public 'data'

TimeoutConst equ 4000 ;About 1 min on 66Mhz 486.
PrtrBase equ 10 ;Offset to LPT1: adrs.
MyPortAdrs word ? ;Holds printer port address.
FileHandle word ? ;Handle for output file.
FileBuffer byte 512 dup (?) ;Buffer for incoming data.
FileSize dword ? ;Size of incoming file.
FileNamePtr dword ? ;Holds ptr to filename

dseg ends

cseg segment para public 'code'
assume cs:cseg, ds:dseg

; TestAbort- Check to see if the user has pressed ctrl-C and wants to
; abort this program. This routine calls BIOS to see if the
; user has pressed a key. If so, it calls DOS to read the
; key (function AH=8, read a key w/o echo and with ctrl-C
; checking).

TestAbort proc near
    push ax
    push cx
    push dx
    mov ah, 1
    int 16h ;See if keypress.
    je NoKeyPress ;Return if no keypress.
    mov ah, 8 ;Read char, chk for ctrl-C.
    int 21h ;DOS aborts if ctrl-C.

NoKeyPress: pop dx
            pop cx
            pop ax
            ret

TestAbort endp

; SendByte- Transmit the byte in AL to the receiving site four bits
; at a time.

SendByte proc near
    push cx
    push dx
    mov ah, al ;Save byte to xmit.
    mov dx, MyPortAdrs ;Base address of LPT1: port.

    ; First, just to be sure, write a zero to bit #4. This reads as a one
    ; in the busy bit of the receiver.
    mov al, 0
    out dx, al ;Data not ready yet.

    ; Wait until the receiver is not busy. The receiver will write a zero
    ; to bit #4 of its data register while it is busy. This comes out as a
    ; one in our busy bit (bit 7 of the status register). This loop waits
    ; until the receiver tells us its ready to receive data by writing a
    ; one to bit #4 (which we read as a zero). Note that we check for a
    ; ctrl-C every so often in the event the user wants to abort the
    ; transmission.

    inc dx ;Point at status register.
    mov cx, 10000
    Wait4NotBusy: in al, dx ;Read status register value.
                  test al, 80h ;Bit 7 = 1 if busy.
                  loopne Wait4NotBusy ;Repeat while busy, 10000 times.
                  je ItsNotBusy ;Leave loop if not busy.
                  call TestAbort ;Check for Ctrl-C.
                  jmp Wait4NotBusy

    ; Okay, put the data on the data lines:

    ItsNotBusy: dec dx ;Point at data register.
                mov al, ah ;Get a copy of the data.
                and al, 0Fh ;Strip out H.O. nibble
                out dx, al ;"Prime" data lines, data not avail.
                or al, 10h ;Turn data available on.
                out dx, al ;Send data w/data available strobe.

    ; Wait for the acknowledge from the receiving site. Every now and then
    ; check for a ctrl-C so the user can abort the transmission program from
    ; within this loop.

    inc dx ;Point at status register.
    mov cx, 10000 ;Times to loop between ctrl-C checks.
    Wait4Ack: in al, dx ;Read status port.
               test al, 80h ;Ack = 1 when rcvr acknowledges.
               loopne Wait4Ack ;Repeat 10000 times or until ack.
               jne GotAck ;Branch if we got an ack.
               call TestAbort ;Every 10000 calls, check for a
jmp W4ALp ; ctrl-C from the user.

; Send the data not available signal to the receiver:

GotAck:   dec  dx      ;Point at data register.
mov al, 0      ;Write a zero to bit 4, this appears
out dx, al     ; as a one in the rcvr’s busy bit.

; Okay, on to the H.O. nibble:

W4NB2:   mov cx, 10000 ;10000 calls between ctrl-C checks.
Wait4NotBsy2:  in al, dx ;Read status register.
test al, 80h ;Bit 7 = 1 if busy.
loopne Wait4NotBsy2 ;Loop 10000 times while busy.
je NotBusy2 ;H.O. bit clear (not busy)?
call TestAbort ;Check for ctrl-C.
jmp W4NB2

; Okay, put the data on the data lines:

NotBusy2:  dec dx      ;Point at data register.
mov al, ah ;Retrieve data to get H.O. nibble.
shr al, 4    ;Move H.O. nibble to L.O. nibble.
out dx, al   ;Prime” data lines.
or al, 10h    ;Data + data available strobe.
out dx, al    ;Send data w/data available strobe.

; Wait for the acknowledge from the receiving site:

W4A2Lp:   mov cx, 10000 ;Read status port.
Wait4Ack2:  in al, dx ;Read status port.
test al, 80h ;Ack = 1
loope Wait4Ack2 ;While while no acknowledge
jne GotAck2 ;H.O. bit = 1 (ack)?
call TestAbort ;Check for ctrl-C
jmp W4A2Lp

; Send the data not available signal to the receiver:

GotAck2:  dec  dx      ;Point at data register.
mov al, 0      ;Output a zero to bit #4 (that
out dx, al     ; becomes busy=1 at rcvr).
mov al, ah ;Restore original data in AL.
pop dx
pop cx
ret

SendByte endp

; Synchronization routines:
;
; Send0s- Transmits a zero to the receiver site and then waits to
; see if it gets a set of ones back. Returns carry set if
; this works, returns carry clear if we do not get a set of
; ones back in a reasonable amount of time.

Send0s proc near
push cx
push dx
mov dx, MyPortAdrs
mov al, 0 ;Write the initial zero
out dx, al ; value to our output port.
xor cx, cx ;Checks for ones 10000 times.
Wait41s: inc dx, cx ;Point at status port.
in al, dx ;Read status port.
dec dx ;Point back at data port.

The PC Parallel Ports

; Send0s— Transmits all ones to the receiver site and then waits to see if it gets a set of zeros back. Returns carry set if this works, returns carry clear if we do not get a set of zeros back in a reasonable amount of time.

Send0s proc near
    push cx
    push dx
    mov dx, MyPortAdrs ;LPT1: base address.
    mov al, 0Fh ;Write the “all ones” value to our output port.
    out dx, al ; value to our output port.

    mov cx, 0
    Wait40s: inc dx ;Point at input port.
        in al, dx ;Read the status port.
        dec dx ;Point back at data port.
        and al, 78h ;Mask input bits.
        loopne Wait40s ;Loop until we get zero back.
        je Got0s ;All zeros? If so, branch.
        clc ;Return failure.
        pop dx
        pop cx
        ret

Got0s: stc ;Return success.
        pop dx
        pop cx
        ret
Send0s endp

; Send1s— Transmits all ones to the receiver site and then waits to see if it gets a set of zeros back. Returns carry set if this works, returns carry clear if we do not get a set of zeros back in a reasonable amount of time.

Send1s proc near
    push cx
    push dx
    mov dx, MyPortAdrs ;LPT1: base address.
    mov al, 0Fh ;Write the “all ones” value to our output port.
    out dx, al ; value to our output port.

    mov cx, 0
    Wait44s: inc dx ;Point at input port.
        in al, dx ;Read the status port.
        dec dx ;Point back at data port.
        and al, 78h ;Mask input bits.
        loopne Wait44s ;Loop until we get zero back.
        je Got1s ;Branch if success.
        clc ;Return failure.
        pop dx
        pop cx
        ret

Got1s: stc ;Return success.
        pop dx
        pop cx
        ret
Send1s endp

; Synchronize—This procedure slowly writes all zeros and all ones to its output port and checks the input status port to see if the receiver site has synchronized. When the receiver site is synchronized, it will write the value 05h to its output port. So when this site sees the value 05h on its input port, both sites are synchronized. Returns with the carry flag set if this operation is successful, clear if unsuccessful.

Synchronize proc near
    print byte "Synchronizing with receiver program"
    print byte cr,lf,0

    mov dx, MyPortAdrs

    mov cx, TimeOutConst ;Time out delay.
SyncLoop: call Send0s ;Send zero bits, wait for
    jc Got0s ;ones (carry set=got ones).
        ; If we didn’t get what we wanted, write some ones at this point and see if we’re out of phase with the receiving site.
Retry0: call Send1s ;Send ones, wait for zeros.
        jc  SyncLoop ;Carry set = got zeros.

; Well, we didn’t get any response yet, see if the user has pressed ctrl-C
; to abort this program.

DoRetry:  call   TestAbort

; Okay, the receiving site has yet to respond. Go back and try this again.

            loop  SyncLoop

; If we’ve timed out, print an error message and return with the carry
; flag clear (to denote a timeout error).

            print
            byte  “Transmit: Timeout error waiting for receiver”
            byte  cr,lf,0
            clo
            ret

; Okay, we wrote some zeros and we got some ones. Let’s write some ones
; and see if we get some zeros. If not, retry the loop.

Got1s:  call   Send1s ;Send one bits, wait for
        jnc  DoRetry ; zeros (carry set=got zeros).

; Well, we seem to be synchronized. Just to be sure, let’s play this out
; one more time.

          call  Send0s ;Send zeros, wait for ones.
          jnc  Retry0
          call  Send1s ;Send ones, wait for zeros.
          jnc  DoRetry

; We’re synchronized. Let’s send out the 05h value to the receiving
; site to let it know everything is cool:

          mov  al, 05h ;Send signal to receiver to
          out  dx, al ; tell it we’re sync’d.
           xor  cx, cx ;Long delay to give the rcvr

FinalDelay:  loop  FinalDelay ; time to prepare.

            print
            byte  “Synchronized with receiving site”
            byte  cr,lf,0
            stc
            ret

Synchronize endp

; File I/O routines:
;
; GetFileInfo—Opens the user specified file and passes along the file
; name and file size to the receiving site. Returns the
; carry flag set if this operation is successful, clear if
; unsuccessful.

Get FileInfo proc near

; Get the filename from the DOS command line:

          mov  ax, 1
          argv
          mov  word ptr FileNamePtr, di
          mov  word ptr FileNamePtr+2, es

            printf
            byte  “Opening %“s
            byte cr,lf,0
            dword FileNamePtr

Get FileInfo endp
; Open the file:
    push ds
    mov ax, 3D00h ;Open for reading.
    lds dx, FileNamePtr
    int 21h
    pop ds
    jc BadFile
    mov FileHandle, ax

; Compute the size of the file (do this by seeking to the last position
; in the file and using the return position as the file length):
    mov bx, ax ;Need handle in BX.
    mov ax, 4202h ;Seek to end of file.
    xor cx, cx ;Seek to position zero
    xor dx, dx ; from the end of file.
    int 21h
    jc BadFile

; Save final position as file length:
    mov word ptr FileSize, ax
    mov word ptr FileSize+2, dx

; Need to rewind file back to the beginning (seek to position zero):
    mov bx, FileHandle ;Need handle in BX.
    mov ax, 4200h ;Seek to beginning of file.
    xor cx, cx ;Seek to position zero
    xor dx, dx
    int 21h
    jc BadFile

; Okay, transmit the good stuff over to the receiving site:
    mov al, byte ptr FileSize ;Send the file
    call SendByte ; size over.
    mov al, byte ptr FileSize+1
    call SendByte
    mov al, byte ptr FileSize+2
    call SendByte
    mov al, byte ptr FileSize+3
    call SendByte

    les bx, FileNamePtr ;Send the characters
    SendName:
        mov al, es:[bx] ; in the filename to
        call SendByte ; the receiver until
        inc bx ; we hit a zero byte.
        cmp al, 0
        jne SendName
        stc ;Return success.
        ret

BadFile:    print
byte "Error transmitting file information:",0
puti
putcr
clc
ret
GetFileInfo endp

; GetFileData-This procedure reads the data from the file and transmits
; it to the receiver a byte at a time.
GetFileData proc near
    mov ah, 3Fh ;DOS read opcode.
    mov cx, 512 ;Read 512 bytes at a time.
    mov bx, FileHandle ;File to read from.
    lea dx, FileBuffer ;Buffer to hold data.
    int 21h ;Read the data
jc GFDError ; Quit if error reading data.
mov cx, ax ; Save # of bytes actually read.
jcxz GFDDone ; quit if at EOF.
lea bx, FileBuffer ; Send the bytes in the file

XmitLoop: mov al, [bx] ; buffer over to the rcvr
call SendByte ; one at a time.
in bx
loop XmitLoop
jmp GetFileData ; Read rest of file.

GFDError: print byte "DOS error ", 0
puti print byte " while reading file", cr, lf, 0
GFDDone: ret GetFileData endp

; Okay, here’s the main program that controls everything.
Main proc
mov ax, dseg
mov ds, ax
meminit

; First, get the address of LPT1: from the BIOS variables area.
mov ax, 40h
mov es, ax
mov ax, es:[PrtrBase]
mov MyPortAdrs, ax

; See if we have a filename parameter:
argc cmp cx, 1
je GotName
print byte "Usage: transmit <filename>", cr, lf, 0
jmp Quit

GotName: call Synchronize ; Wait for the transmitter program.
jnc Quit

call GetFileInfo ; Get file name and size.
jnc Quit

call GetFileData ; Get the file’s data.

Quit: ExitPgm ; DOS macro to quit program.
Main endp

cseg ends

sseg segment para stack 'stack'
stk byte 1024 dup ("stack ")
sseg ends

zzzzzzseg segment para public 'zzzzzz'
LastBytes byte 16 dup (?)
zzzzzzseg ends
end Main
Here is the receiver program that accepts and stores away the data sent by the program above:

```assembly
; RECEIVE.ASM
;
; This program is the receiver portion of the programs that transmit files
; across a Laplink compatible parallel cable.
;
; This program assumes that the user want to use LPT1: for transmission.
; Adjust the equates, or read the port from the command line if this
; is inappropriate.

; 286
.xlist
include stdlib.a
include lib stdlib.lib
.list

dseg segment para public 'data'

timeOutConst equ 100 ;About 1 min on 66Mhz 486.
prtrBase equ 8 ;Offset to LPT1: adrs.
myPortAdrs word ? ;Holds printer port address.
fileHandle word ? ;Handle for output file.
fileBuffer byte 512 dup (?) ;Buffer for incoming data.

filesize dword ? ;Size of incoming file.
fileName byte 128 dup (0) ;Holds filename

dseg ends

cseg segment para public 'code'

assume cs:cseg, ds:dseg

; TestAbort- Reads the keyboard and gives the user the opportunity to
; hit the ctrl-C key.

TestAbort proc near

push ax
mov ah, 1
int 16h ;See if keypress.
je NoKeyPress
mov al, 8 ;Read char, chk for ctrl-C
int 21h

NoKeyPress: pop ax
ret
TestAbort endp

; GetByte- Reads a single byte from the parallel port (four bits at
; at time). Returns the byte in AL.

GetByte proc near

push cx
push dx

mov dx, MyPortAdrs
mov al, 10h ;Signal not busy.
out dx, al

inc dx ;Point at status port

wait4Dlp: mov cx, 10000

wait4Data: in al, dx ;See if data available.

; Receive the L.O. Nibble.

mov dx, MyPortAdrs
mov al, 10h ;Signal not busy.
out dx, al

inc dx ;Point at status port

mov cx, 10000

wait4Data: in al, dx ;See if data available.

; (bit 7=0 if data available).

loope wait4Data
je DataIsAvail ;Is data available?
call TestAbort ;if not, check for ctrl-C.
```

Page 1217
jmph W4DLp

DataIsAvail: shr al, 3 ;Save this four bit package
and al, 0Fh ; (This is the L.O. nibble
mov ah, al ; for our byte).
dec dx ;Point at data register.
mov al, 0 ;Signal data taken.
out dx, al

inc dx ;Point at status register.

W4ALp:
mov cx, 10000
Wait4Ack: in al, dx ; Wait for transmitter to
test al, 80h ; retract data available.
loope Wait4Ack ; Loop until data not avail.
jne NextNibble ; Branch if data not avail.
call TestAbort ; Let user hit ctrl-C.
jmph W4ALp

; Receive the H.O. nibble:

NextNibble: dec dx ; Point at data register.
mov al, 10h ; Signal not busy
out dx, al
inc dx ; Point at status port

W4D2Lp:
mov cx, 10000
Wait4Data2: in al, dx ; See if data available.
test al, 80h ; (bit 7=0 if data available).
loope Wait4Data2 ; Loop until data available.
jne DataAvail2 ; Branch if data available.
call TestAbort ; Check for ctrl-C.
jmph W4D2Lp

DataAvail2: shl al, 1 ; Merge this H.O. nibble
and al, 0F0h ; with the existing L.O.
or ah, al ; nibble.
dec dx ; Point at data register.
mov al, 0 ; Signal data taken.
out dx, al

inc dx ; Point at status register.

W4A2Lp:
mov cx, 10000
Wait4Ack2: in al, dx ; Wait for transmitter to
test al, 80h ; retract data available.
loope Wait4Ack2 ; Wait for data not available.
jne ReturnData ; Branch if ack.
call TestAbort ; Check for ctrl-C
jmp W4A2Lp

ReturnData: mov al, ah ; Put data in al.
pop dx
pop cx
ret

GetByte endp

; Synchronize- This procedure waits until it sees all zeros on the input
; bits we receive from the transmitting site. Once it receives
; all zeros, it writes all ones to the output port. When
; all ones come back, it writes all zeros. It repeats this
; process until the transmitting site writes the value 05h.
Synchronize proc near

print
byte "Synchronizing with transmitter program"
byte cr,lf,0

mov dx, MyPortAdrs
mov al, 0 ; Initialize our output port
out dx, al ; to prevent confusion.
mov bx, TimeOutConst ; Time out condition.

mov cx, 10000
Wait4Ack: test al, 80h ; (bit 7=0 if data available).
loope Wait4Ack ; Wait for data not available.
jne Synchronize ; Branch if ack.
call TestAbort ; Check for ctrl-C
jmp Synchronize

Synchronize endp
The PC Parallel Ports

SyncLoop:      mov    cx, 0        ;For time out purposes.
SyncLoop0:     inc    dx          ;Point at input port.
in        al, dx          ;Read our input bits.
dec    dx
and       al, 78h       ;Keep only the data bits.
cmp    al, 78h          ;Check for all ones.
je     Got1s            ;Branch if all ones.
cmp    al, 0           ;See if all zeros.
loopne   SyncLoop0

; Since we just saw a zero, write all ones to the output port.
    mov    al, 0FFh        ;Write all ones
    out    dx, al

; Now wait for all ones to arrive from the transmitting site.
SyncLoop1:     inc    dx          ;Point at status register.
in       al, dx          ;Read status port.
dec     dx           ;Point back at data register.
and       al, 78h       ;Keep only the data bits.
cmp    al, 78h          ;Are they all ones?
loopne   SyncLoop1      ;Repeat while not ones.
je     Got1s            ;Branch if got ones.

; If we've timed out, check to see if the user has pressed ctrl-C to
; abort.
call    TestAbort        ;Check for ctrl-C.
dec     bx           ;See if we've timed out.
jne    SyncLoop        ;Repeat if time-out.

print
byte    "Receive: connection timed out during synchronization"
byte    cr,lf,0         ;Signal time-out.
clc
ret

; Jump down here once we've seen both a zero and a one. Send the two
; in combinations until we get a 05h from the transmitting site or the
; user presses Ctrl-C.
Got1s:       inc    dx          ;Point at status register.
in      al, dx          ;Just copy whatever appears
dec     dx           ;in our input port to the
shr     al, 3         ;output port until the
and     al, 0Fh       ;transmitting site sends
cmp    al, 05h          ;us the value 05h
je     Synchronized    ;Keep inverting what we get
not    al                ;and send it to xmitter.
call    TestAbort     ;Check for CTRL-C here.
jmp     Got1s

; Okay, we're synchronized. Return to the caller.
Synchronized:
and     al, 0Fh        ;Make sure busy bit is one
out       dx, al        ;(bit 4=0 for busy=1).
print
byte    "Synchronized with transmitting site"
byte    cr,lf,0
stc
ret

Synchronize   endp

; GetFileInfo - The transmitting program sends us the file length and a
; zero terminated filename. Get that data here.
GetFileInfo   proc   near
mov   dx, MyPortAdrs
mov   al, 10h         ;Set busy bit to zero.
out dx, al ;Tell xmit pgm, we’re ready.

; First four bytes contain the filesize:
call GetByte
mov byte ptr FileSize, al
call GetByte
mov byte ptr FileSize+1, al
call GetByte
mov byte ptr FileSize+2, al
call GetByte
mov byte ptr FileSize+3, al

; The next n bytes (up to a zero terminating byte) contain the filename:
mov bx, 0
GetFileName: call GetByte
mov FileName[bx], al
call TestAbort
inc bx
cmp al, 0
jne GetFileName
ret
GetFileInfo endp

; GetFileData- Receives the file data from the transmitting site
; and writes it to the output file.
GetFileData proc near

; First, see if we have more than 512 bytes left to go
 cmp word ptr FileSize+2, 0 ;If H.O. word is not
 jne MoreThan512 ; zero, more than 512.
cmp word ptr FileSize, 512 ;If H.O. is zero, just
 jbe LastBlock ; check L.O. word.

; We’ve got more than 512 bytes left to go in this file, read 512 bytes
; at this point.
MoreThan512: mov cx, 512 ;Receive 512 bytes
 lea bx, FileBuffer ; from the xmitter.
ReadLoop: call GetByte ;Read a byte.
mov [bx], al ;Save the byte away.
inc bx ;Move on to next
loop ReadLoop ; buffer element.

; Okay, write the data to the file:
mov ah, 40h ;DOS write opcode.
mov bx, FileHandle ;Write to this file.
mov cx, 512 ;Write 512 bytes.
lea dx, Filebuffer ;From this address.
int 21h ;Quit if error.

; Decrement the file size by 512 bytes:
sub word ptr FileSize, 512 ;32-bit subtraction
sbb word ptr FileSize, 0 ; of 512.
jmp GetFileData

; Process the last block, that contains 1..511 bytes, here.
LastBlock:
mov cx, word ptr FileSize ;Receive the last
lea bx, FileBuffer ; 1..511 bytes from
ReadLB:
call GetByte ; the transmitter.
mov [bx], al
inc bx
loop ReadLB
mov ah, 40h ;Write the last block
mov bx, FileHandle ; of bytes to the
mov cx, word ptr FileSize ; file.
lea dx, Filebuffer
int 21h
jnc Closefile

BadWrite: print
byte "DOS error ",0
puti
print byte " while writing data.",cr,lf,0

; Close the file here.

CloseFile: mov bx, FileHandle ;Close this file.
mov ah, 3Eh ;DOS close opcode.
int 21h
ret

GetFileData endp

; Here’s the main program that gets the whole ball rolling.

Main proc
mov ax, dseg
mov ds, ax
meminit

; First, get the address of LPT1: from the BIOS variables area.

mov ax, 40h ;Point at BIOS variable segment.
mov es, ax
mov ax, es:[PrtrBase]
mov MyPortAdrs, ax

call Synchronize ;Wait for the transmitter program.
jnc Quit

call GetFileInfo ;Get file name and size.
printf byte "Filename: %s
File size: %ld
",0
dword Filename, FileSize

mov ah, 3Ch ;Create file.
mov cx, 0 ;Standard attributes
lea dx, Filename
int 21h
jnc GoodOpen

print byte "Error opening file",cr,lf,0
jmp Quit

GoodOpen: mov FileHandle, ax
call GetFileData ;Get the file’s data.

Quit: ExitPgm ;DOS macro to quit program.

Main endp

end Main
21.5 Summary

The PC's parallel port, though originally designed for controlling parallel printers, is a general purpose eight bit output port with several handshaking lines you can use to control many other devices in addition to printers.

In theory, parallel communications should be many times faster than serial communications. In practice, however, real world constraints and economics prevent this from being the case. Nevertheless, you can still connect high performance devices to the PC's parallel port.

The PC's parallel ports come in two varieties: unidirectional and bidirectional. The bidirectional versions are available only on PS/2s, certain laptops, and a few other machines. Whereas the eight data lines are output only on the unidirectional ports, you can program them as inputs or outputs on the bidirectional port. While this bidirectional operation is of little value to a printer, it can improve the performance of other devices that connect to the parallel port, such as disk and tape drives, network adapters, SCSI adapters, and so on.

When the system communicates with some other device over the parallel port, it needs some way to tell that device that data is available on the data lines. Likewise, the devices needs some way to tell the system that it is not busy and it has accepted the data. This requires some additional signals on the parallel port known as handshaking lines. A typical PC parallel port provides three handshaking signals: the data available strobe, the data taken acknowledge signal, and the device busy line. These lines easily control the flow of data between the PC and some external device.

In addition to the handshaking lines, the PC's parallel port provides several other auxiliary I/O lines as well. In total, there are 12 output lines and five input lines on the PC's parallel port. There are three I/O ports in the PC's address space associated with each I/O port. The first of these (at the port's base address) is the data register. This is an eight bit output register on unidirectional ports, it is an input/output register on bidirectional ports. The second register, at the base address plus one, is the status register. The status register is an input port. Five of those bits correspond to the five input lines on the PC's parallel port. The third register (at base address plus two) is the control register. Four of these bits correspond to the additional four output bits on the PC, one of the bits controls the IRQ line on the parallel port, and a sixth bit controls the data direction on the bidirectional ports.

For more information on the parallel port's hardware configuration, see:

- “Basic Parallel Port Information” on page 1199
- “The Parallel Port Hardware” on page 1201

Although many vendors use the parallel port to control lots of different devices, a parallel printer is still the device most often connected to the parallel port. There are three ways application programs commonly send data to the printer: by calling DOS to print a character, by calling BIOS' int 17h ISR to print a character, or by talking directly to the parallel port. You should avoid this last technique because of possible software incompatibilities with other devices that connect to the parallel port. For more information on printing data, including how to write your own int 17h ISR/printer driver, see:

- “Controlling a Printer Through the Parallel Port” on page 1202
- “Printing via DOS” on page 1203
- “Printing via BIOS” on page 1203
- “An INT 17h Interrupt Service Routine” on page 1203

One popular use of the parallel port is to transfer data between two computers; for example, transferring data between a desktop and a laptop machine. To demonstrate how to use the parallel port to control other devices besides printers, this chapter presents a program to transfer data between computers on the unidirectional parallel ports (it also works on bidirectional ports). For all the details, see

- “Inter-Computer Communications on the Parallel Port” on page 1209