Device Drivers I: Programmed I/O and Printers

Contents

20.1 I/O ports and instructions
20.2 Programming parallel printers
20.3 Asynchronous serial communications
20.4 Programming an asynchronous serial interface
   Summary
   Self-check quiz

The theme of the next three chapters is how to use assembly language programs to drive some of the I/O devices connected to a microcomputer, namely: a printer, the keyboard, and the display. There will be a chapter which discusses driving each of these three devices. But also, each chapter will have as an equally important theme one of the three main ways in which it is possible to drive hardware devices: programmed I/O through I/O ports; interrupt I/O using (hardware) interrupts; and direct memory access.

To begin, then, programmed I/O, which involves driving hardware devices through I/O ports, will be considered.

20.1 I/O ports and instructions

Some devices which are external to the main microcomputer are capable of both receiving and sending data. Thus a disk drive is, by design, external to the main computer's circuits, but may receive data to write on to a disk and may
also send data read from a disk to an area of memory. If the external device is a printer, communication is one way only as far as our data is concerned. But if we are to be able to take 'intelligent' actions (such as stopping the computer from sending out data to be printed when the printer has run out of paper), some primitive form of two-way communication is essential.

External devices are almost always connected not directly to the system bus but to an **interface**. Most interfaces are designed so that a range of such possibilities can be catered for, leaving the programmer to specify his or her requirements. This is done by setting certain registers in the peripheral **interface chip** used by the interface. Because of the possible confusion with the main registers of the 8086 family, peripheral interface chip registers are usually referred to as **ports**.

A typical interface may have three or more ports associated with it:

- A **control port**, the setting of which will determine if the interface is to send or receive.
- A **data port** for the data element to be transmitted or to hold a data element received.
- A **status port** which can be used to obtain information such as 'printer out of paper, don't send any more data' or, for a serial transmission, 'all the bits of the data element haven't yet been received'.

Any interface will have a least a data port, but the functions of status port and control port may be combined into one port for a simple interface. On the other hand, sophisticated interfaces may have several control and status ports. Interfaces usually come on a circuit board which plugs into the main computer circuit board (the **motherboard** and, while the actual port numbers and port settings may differ from machine to machine, the techniques involved will always be similar.

To illustrate how to use I/O ports to drive printers, it will be assumed that your computer has a parallel printer interface compatible with the IBM Parallel Printer Adapter and a serial interface compatible with the IBM Serial Adapter. If it does not, it may be that you have to modify the example programs given slightly, in order for them to run exactly as specified on your computer. The necessary changes may range from specifying different addresses from those given for the I/O ports to different arrangements for the control and status ports.

### 20.1.1 Input and output instructions

Just as locations in memory are referred to by their numeric addresses, so the various ports attached to an 8086-family based microcomputer are numbered. There is provision for ports numbered 0 to 0FFFFH. Which port is given which number is more or less determined by the designer of a particular microcomputer.
Setting a port is accomplished by the 8086-family **OUT** instruction. It has the general formats:

1. \texttt{OUT DX, accumulator}
2. \texttt{OUT port number, accumulator}

where \texttt{accumulator} denotes AL or AX depending on whether an 8-bit or 16-bit port is involved. If the port number is between 0 and 0FFH inclusive then format (2) can be used and the contents of the named register are copied thereby into the specified port. In all other cases format (1) must be used, in which case the contents of the named register are copied into the port whose number is given in register DX.

The contents of a port are obtained by using the 8086-family **IN** instruction, which has two general formats corresponding to those for the **OUT** instruction:

\texttt{IN accumulator, DX} \quad /
\texttt{IN accumulator, port number}

## 20.2 Programming parallel printers

Parallel printers require data to be transmitted to them in parallel. The parallel transmission of the letter A (ASCII code 01000001) to a printer is illustrated in Figure 20.1.

This section will describe how to program the transmission of data in parallel through a parallel printer interface. In our case, the parallel printer interface has three 8-bit I/O ports:

![Parallel printer interface diagram](image)

**Figure 20.1** Parallel transmission of the ASCII code for 'A' – eight bits are transmitted at a time, so eight data lines are necessary.
<table>
<thead>
<tr>
<th>Port function</th>
<th>Port number</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>3BCH</td>
</tr>
<tr>
<td>output control</td>
<td>3BEH</td>
</tr>
<tr>
<td>printer status</td>
<td>3BDH</td>
</tr>
</tbody>
</table>

Port number 3BCH is the data port. If this port is set to the ASCII code for a character, and the output control and printer status registers are set correctly, then that character will be printed wherever printing last left off. Thus, the sequence

```
MOV AL,42H
MOV DX,3BCH
OUT DX,AL
```

would set the data port of the parallel printer interface to the ASCII code for the letter B.

Having set up the data port we must next check the printer status before telling the printer to print the data. The sequence

```
MOV DX,3BDH
IN AL,DX
```

copies the contents of the printer status port into AL and the setting of the eight bits in the resulting value signifies various printer conditions, as shown in Figure 20.2 for both the output control and the printer status ports.

To test if the printer is busy we can use the TEST instruction. Remembering that 80H = 10000000B:

```
TEST AL,80H
```

sets the Z-flag if the printer is busy but does not change the contents of AL. If the printer is busy we simply have to wait until it is ready:

```
MOV DX,3BDH
TEST_IF_BUSY: IN AL,DX
TEST AL,80H
JZ TEST_IF_BUSY
```

Repeatedly checking to see if the printer is ready like this is known as polling the printer. (Generally, polling is the programmed testing of one or more bits which indicate whether or not a device is ready to undertake some action or has completed some action.)

Each character is sent to the printer following a timing pulse. Once we know that the printer is ready, we use the output control port to send signals which we can therefore regard as being to start and stop the printing of the character in the data port. Since the printer can capture the character from the data port very quickly, the stop and start instructions can come one after another. To start printing, the right-most bit of the contents of the output control port must be set to 1; to stop printing it is set back to 0. This must be
done for each and every character printed. Lastly, bits 2 and 3 must always be set to 1 during printing.

Thus, given that \( 0\text{DH} = 00001101\text{B} \) and that \( 0\text{CH} = 00001100\text{B} \), the following instruction sequence prints the character whose ASCII code is in the data port, provided the printer is not busy:

\[
\begin{align*}
\text{MOV } DX, \text{3BEH} \\
\text{MOV } AL, 0\text{DH} \\
\text{OUT } DX, AL \quad ; \text{start printing} \\
\text{MOV } AL, 0\text{CH} \\
\text{OUT } DX, AL \quad ; \text{stop printing}
\end{align*}
\]

Before beginning a session, the printer must be initialized by setting bit 2 of the output control register to 0 for at least 50 microseconds \((0.000 \, 050\, \text{sec})\).
seconds) and then setting it back to 1. This wait can be timed by executing 50D
NOP instructions (NOP - No Operation - is an 8086-family instruction provided for
just such timing purposes. It has no effect on the state of either memory or
registers, but since its execution nevertheless takes a certain amount of time,
repeated execution of NOP can be used to synchronize operations.)

Putting all this together we can easily construct (Figure 20.3) a program
which turns the computer into a typewriter. Those printable characters typed at
the keyboard are sent to a printer connected via the parallel interface until the
letter X is typed, when a return is made to DOS.

```
CR EQU ODH
LF EQU OAH

DATA SEGMENT
DATA ENDS

WORKING_STORAGE SEGMENT STACK
    DW 100H DUP(?)
WORKING_STORAGE ENDS

CODE SEGMENT
ASSUME CS:CODE,DS:DATA,SS:WORKING_STORAGE
;set up data segment addressability
START: MOV AX,DATA
        MOV DS,AX
;initialize the printer
        MOV DX,3BEH
;set bit 6 of output control port to 0
        MOV AL,08H
        OUT DX,AL
        MOV AL,0CH
        MOV CX,50D
;wait at least 50 microseconds
WAIT:  NOP
        LOOP WAIT
;repeat read a character from the keyboard
        READ: CALL READCHAR
;print it
        CALL SEND_TO_PRINTER
;if char = CR
        CMP AL,CR
        JNZ XCHECK
;then begin
;print a LF
        MOV AL,LF
        CALL SEND_TO_PRINTER
;LF to display
        CALL DISPLAY_NEWLINE
;end
```

**Figure 20.3**
Parallel printer control program.
Programming parallel printers

; until char=X
XCHECK: CMP AL,'X'
    JZ DONE
    JMP READ
; return to DOS
DONE: MOV AX,4C00H
        INT 21H

SEND_TO_PRINTER PROC NEAR
; save char on stack
    PUSH AX
; transfer character to data port
    MOV DX,3BCH
    OUT DX,AL
; check if the printer is busy
    MOV DX,3BDH
TEST_IF_BUSY:
    IN AL,DX
    TEST AL,80H
    JZ TEST_IF_BUSY
; now print the character
    MOV DX,3BEH
    MOV AL,ODH
; start printing
    OUT DX,AL
    MOV AL,0CH
; stop printing
    OUT DX,AL
; restore character from stack
    POP AX
    RET
SEND_TO_PRINTER ENDP

READCHAR PROC NEAR
    MOV AH,1
    INT 21H
    RET
READCHAR ENDP

DISPLAY_NEWLINE PROC NEAR
    MOV AH,2
    MOV DL,LF
    INT 21H
DISPLAY_NEWLINE ENDP

CODE ENDS
END START

Figure 20.3 (cont.)

EXERCISES

20.1 Enhance the program in Figure 20.3 so that if the printer runs out of paper a suitable message is displayed on the screen and the program
waits until more paper has been inserted. Once this has been done the program should return to normal execution.

20.2 The bit in position 4 of the printer status port is set to 1 if the printer is on-line (that is, it is switched on and capable of receiving data from the computer and not left in some printer test mode). If the printer is not on-line this bit is set to 0. Amend the program in Figure 20.3 so that a suitable message is displayed on the screen if the printer is not on-line and the program waits until it is.

20.3 Asynchronous serial communications

Dealing with parallel printers is, in many respects, easier than dealing with printers which expect data to arrive from a serial interface. This is because although most serial transmission of data conforms to one of a family of electrical standards usually referred to informally as the RS232 standard, considerable choice is available in the format in which data is transmitted, in the speed at which it is to be transmitted, and in the electrical connections used to join the two communicating devices together. This section examines some of these various possibilities from the point of view of asynchronous serial communications: that is, serial communications in which the characters transferred are separated by special bit patterns, so that the transmitter and receiver need not work at precisely the same speed. Section 20.4 will then discuss how to write 8086-family assembly language software to carry out the serial transmission of data.

20.3.1 Serial data transmission

To transmit the letter A serially, the ASCII code 01000001 has to be transmitted. Electronically, a 1 is represented by a high voltage and a 0 by a low voltage. (From the point of view of our current discussion, the exact voltages do not matter.) Thus to transmit a letter A the pulse train shown in Figure 20.4 must pass along the transmit wire.

While no transmission is taking place, the signal on the wire between a computer and a serial printer is kept at a high voltage. On its own, therefore,

\[
\begin{align*}
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{align*}
\]

\[
\begin{align*}
\text{Equal intervals of time}
\end{align*}
\]

\text{Figure 20.4}
Serial transmission of the ASCII code for the letter A.
this pulse train is inadequate because it does not allow for the possibility that the letter A might begin a transmission, as shown in Figure 20.5.

To solve the problem a convention is adopted. In order to begin transmission without missing any data a start bit is added to the front of each data character. Thus A is transmitted as 01000001 (see Figure 20.6).

**Parity bits**

Any scheme adopted for serially transmitted data must, if it is not to be severely limited in its scope, allow for sending data along telephone lines and between machines in environments which can corrupt electrical signals flowing along a wire. To give protection against the corruption of a ‘0’ into a ‘1’ (or vice versa) by outside electrical interference and to assist in synchronization of the transmission and reception of signals, one or two bits are added to the actual data bits for each character. These extra bits are known as parity bits.

One or more parity bits are added at the end of the data bits to guard against interference accidentally changing one letter into another. For example, the ASCII code for B is 01000010 and the ASCII code for C is 01000011. Thus, if electrical interference corrupted the right-most 0 bit of the code for B into a 1 bit, the receiver would think that C had been transmitted, not B.

At the heart of the problem is how very ‘close’ the code for B is to the code for C. One solution then would be to change the code, but there are several objections to this. In particular, changing the code might mean that B and C no longer have similar codes, but that L and X do. Theory shows that the only way you can represent all the upper- and lower-case letters and digits and punctuation marks without any two codes being very similar is to have a longer code. This is, in effect, what we do with parity bits. However, there are two systems for adding parity bits to create a longer code: the odd parity system and the even parity system.

**Figure 20.5**

Why the pulse train for the bits in the ASCII code for the letter A is not adequate on its own.

**Figure 20.6**

The use of a start bit.
Let us first consider odd parity. In this system, since the ASCII for B – 01000010 – contains two 1-bits and two is even we put a 1 after the last data bit to be transmitted to make the total number of 1s odd. Thus we transmit B as 101000010. The ASCII for C on the other hand (01000011) contains three 1-bits; three is odd so we put a 0 after the last bit (001000011) to keep the total number of 1s transmitted odd.

Now, if B (101000010) gets corrupted to C (001000011), two bits will have been corrupted. Since this is much less likely to happen, we have achieved a degree of protection. And the hardware or software can be designed to check for oddities so that if

101000010

was corrupted to

101000011

then an error would be signaled because 101000011 cannot be the code of any real data item under our odd parity system.

The system we have described is called odd parity because we have arranged always to transmit an odd number of 1s. Here are a few more examples:

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII code</th>
<th>With odd parity</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>01001000</td>
<td>101001000</td>
</tr>
<tr>
<td>Z</td>
<td>01011011</td>
<td>001011011</td>
</tr>
<tr>
<td>;</td>
<td>00111011</td>
<td>000111011</td>
</tr>
</tbody>
</table>

Under the alternative, even parity, system a 0 or 1 bit is added to the code to ensure that the number of 1s transmitted is always even. These days it is more likely that a system uses even parity than odd parity if it uses parity bits at all.

**Stop bits**

Stop bits are added to the data bits (and parity bit if there is one) quite simply to assist in the synchronization of transmitter and receiver. Just one stop bit is added usually. Thus the letter A, ASCII code 01000001, with even parity and one stop bit is actually transmitted as 00010000010 (Figure 20.7). Eleven bits for a single character! On slower printers, particularly the heavier old-fashioned models like teletypes, it was found that it was a good idea, practically speaking, to have a couple of stop bits and dispense with parity. The two stop bits aided electrical synchronization and gave the printer’s mechanism a chance to keep up with the speed of the electrical signals. (When the print head gets to the end of a line, for example, there will be an appreciable delay whilst the print head returns to the start of the next line.)
3 - bit - one bit is the least length of a character as well as the shortest stop bit. The 1-bit stop bit is the total number of bits per character. While it will be used for parity, it will not be used for stop bit. The character will be transmitted in the form shown in Figure 20.8.

With two stop bits and no parity the letter F (ASCII code 01000110) will be transmitted in the form shown in Figure 20.8.

All serially connected devices require one start bit and at least one stop bit. That is for sure. Thereafter, more or less any combination is to be found on some device or other. It is rare, in fact, for the full 8-bit ASCII codes to be used. A 7-bit ASCII code allows 128 characters, which is often as many as less sophisticated peripherals and software will handle. (128 characters will include a,b,, . . . , z, A, B, . . . , Z, 0, 1, . . . 9, and punctuation marks for example.) However, if there were such a thing as an average device with an RS232 interface, my guess is that it would as likely as not transmit data in the format:

1 start bit: 1 stop bit: 7 data bits: even parity (7-bit ASCII)

(Of course, you may go through the whole of your life and never meet such a device; that is the nature of the business. But if one had to guess because of inadequate, missing or lost documentation, the above format would be a sensible first try.)

20.3.2 Speed of transmission

For the speed of transmission in serial communications, the basic unit is not a mile per hour but a baud, which in our context is a bit per second. (Baudot was an early pioneer in data communications.) Since the earliest computer terminals were made by US-based Teletype Corporation, and they used the data format 1 start bit, 7 data bits, even parity and 2 stop bits (that is, 11 bits in all), and printed 10 characters per second, this meant that the actual
transmission speed between computer and terminal was 110 baud (110 bits per second). No-one would want a printer slower than a teletype, so 110 baud has become the slowest transmission rate used in the microcomputer environment by historical accident.

Close on the heels of the Teletype Corporation, General Electric produced a 300 baud terminal. This used 1 start bit, 1 stop bit, 7 data bits and even parity, so that in this case 300 baud was exactly equivalent to 30 characters per second. As speeds of printers and the like increased, other manufacturers tended to produce devices operating at speeds which were multiples of 300 baud. Speeds of 1200, 2400 and 9600 baud are particularly common, 9600 baud being mostly used for transmission between a computer and its display.

Notice then, that the operating speed of a printer which the manufacturer says can be fed at, say, 300 baud will effectively depend on the data transmission format. If 11 bits are used to transmit one character, such a printer will operate at $300 / 11 = 27$ characters per second whereas if only 9 bits are used, the effective rate is $300 / 9 = 33$ characters per second.

### 20.3.3 Serial and parallel data transmission compared

For communication over lines with a given maximum bit rate, parallel data communication is quicker than serial because of the use of several lines. The disadvantage of parallel communication is the cost of the extra lines. These costs increase with the distance over which data has to be transmitted or received, and consequently parallel communication is used over long distances only if very high data transfer rates are required.

### 20.4 Programming an asynchronous serial interface

In this section it will be assumed that a correct electrical connection has been made between the serial interface of the computer and that of the printer. Serial interfaces can be constructed in many different ways, but perhaps the most popular of them is based on a support chip called the 8250 asynchronous communications element. This section will illustrate how to program serial interfaces using assembly language with reference to such an interface.

From the programmer’s point of view, a serial interface for an 8086-family microprocessor based on the 8250 chip will have two data ports (one for data to be transmitted and one for data received), two status ports, and six control ports, as shown in Table 20.1.

Five of these ports must be programmed to initialize the interface before any communication begins, but thereafter can be ignored. These are:
Table 20.1 8250 registers.

<table>
<thead>
<tr>
<th>I/O port address</th>
<th>Register selected</th>
<th>Input or output</th>
</tr>
</thead>
<tbody>
<tr>
<td>3F8H</td>
<td>Bit 7 of line control register = 0:</td>
<td>Output</td>
</tr>
<tr>
<td></td>
<td>Transmitter holding register</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bit 7 of line control register = 0:</td>
<td>Input</td>
</tr>
<tr>
<td></td>
<td>Receiver data register</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bit 7 of line control register = 1:</td>
<td>Output</td>
</tr>
<tr>
<td>3F9H</td>
<td>Baud rate divisor (LSB)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bit 7 of line control register = 1:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Baud rate divisor (MSB)</td>
<td>Output</td>
</tr>
<tr>
<td></td>
<td>Bit 7 of line control register = 0:</td>
<td></td>
</tr>
<tr>
<td>3FAH</td>
<td>Interrupt-identification register</td>
<td>Input</td>
</tr>
<tr>
<td>3FBH</td>
<td>Line control register</td>
<td>Output</td>
</tr>
<tr>
<td>3FCH</td>
<td>Modem control register</td>
<td>Output</td>
</tr>
<tr>
<td>3FDH</td>
<td>Line status register</td>
<td>Input</td>
</tr>
<tr>
<td>3FEH</td>
<td>Modem status register</td>
<td>Input</td>
</tr>
</tbody>
</table>

- Baud rate divisor (LSB)
- Baud rate divisor (MSB)
- Line control register
- Modem control register
- Interrupt enable register

In the complete program example it will be assumed that the printer in use has an interface which expects to receive data transmitted from the computer at 1200 baud in the format 7 data bits, 1 stop bit and odd parity. To initialize the computer's interface to 1200 baud the baud rate divisor registers must be set to an appropriate value as given in Table 20.2.

Table 20.2 Baud rate divisor values.

<table>
<thead>
<tr>
<th>Desired baud rate</th>
<th>Value for baud rate divisor registers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSB</td>
</tr>
<tr>
<td>110</td>
<td>04H</td>
</tr>
<tr>
<td>300</td>
<td>01H</td>
</tr>
<tr>
<td>600</td>
<td>00H</td>
</tr>
<tr>
<td>1200</td>
<td>00H</td>
</tr>
<tr>
<td>2400</td>
<td>00H</td>
</tr>
<tr>
<td>4800</td>
<td>00H</td>
</tr>
<tr>
<td>9600</td>
<td>00H</td>
</tr>
</tbody>
</table>
Having initialized the baud rate divisor registers, the next step is to initialize the **line control register**. This determines the character length, number of stop bits and the type of parity to be used in a serial communication (see Figure 20.9).

Normally, bits 5, 6, and 7 of the line control register should be set to 0. Bit 7 is set to 1 to enable access to the baud rate divisor registers but is otherwise set to 0, while non-zero settings of bits 5 and 6 have a special purpose beyond the scope of this book.

After the line control register, the next register to be initialized is the **modem control register** (see Figure 20.10). It has only two settings of interest to us here, determined respectively by bits 0 and 1.

Interrupts will be discussed in detail in Chapter 21. However, while the **interrupt enable register** allows the programmer selectively to enable or disable four different types of interrupts, for our present purposes we do not wish to confuse the issue by allowing interrupts of any kind during our serial transmissions, and so the interrupt enable register must be initialized to zero. This completes all the necessary initializations.

Before a character can be transmitted we must first check the **line status register** (Figure 20.11) to see if the transmitter holding register is empty. (Many of the status indications provided are beyond our current terms of reference and will not be discussed further here.) This involves waiting until all
the bits in the transmitter holding register are zero except for bit 5. Once this is the case, then the ASCII code for a character can be placed in the transmitter holding register, and the character will then be transmitted in the form determined by the initializations of the five control registers.

A complete example program constructed along these lines and showing serial transmission of characters typed at the keyboard to a printer is given in Figure 20.12. When the letter X is typed at the keyboard, control returns to DOS.

**SUMMARY**

Programmed I/O relies on the `in` and `out` instructions. These allow us to write device drivers which depend entirely upon the construction of an interface which is plugged into the main computer. We saw this in the context of both a parallel printer interface and an asynchronous serial printer interface.

In order to give illustrative programs we had to be specific as far as certain details of the underlying hardware are concerned. However, it is the principles involved which are important here, not a particular set of port addresses or the individual control, status and data register configuration of a given chip around which a specific interface has been designed. Indeed, the basic ideas in this chapter will also permeate the next two chapters, which deal with different aspects of writing device drivers in 8086-family assembly language.
CR EQU ODH
LF EQU OAH

DATA SEGMENT
DATA ENDS

WORKING STORAGE SEGMENT STACK
    DW 100H DUP(?)
WORKING STORAGE ENDS

CODE SEGMENT
ASSUME CS:CODE, DS:DATA, SS:WORKING STORAGE
;set up data segment addressability
START: MOV AX, DATA
    MOV DS, AX
;access 8250 baud rate divisor registers
    MOV DX, 3FBH
    MOV AL, 80H
    OUT DX, AL
;set the baud rate to 1200 baud
    MOV DX, 3F8H
    MOV AL, 60H
    OUT DX, AL
    MOV DX, 3F9H
    MOV AL, 0
    OUT DX, AL
;initialize the line control register to give data
;format of 7 data bits, 1 stop bit, odd parity
    MOV DX, 3BFH
    MOV AL, OAH
    OUT DX, AL
;initialize the modem control register
;to give RTS and DTR signals
    MOV DX, 3FCH
    MOV AL, 3H
    OUT DX, AL
;disable all interrupts during transmission of data
    MOV DX, 3F9H
    MOV AL, 0
    OUT DX, AL
;repeat read a character from the keyboard
    READ: CALL READCHAR
;print it
    CALL SEND_TO_PRINTER
;if char = CR
    CMP AL, CR
    JNZ XCHECK
;then begin
    ;print a LF
    MOV AL, LF
    CALL SEND_TO_PRINTER
    ;LF to display
    CALL DISPLAY_NEWLINE
;end

Figure 20.12
Serial printer control program.
;until char = X
XCHECK: CMP AL,'X'
    JZ DONE
    JMP READ
;return to DOS
DONE: MOV AX,4C00H
    INT 21H
SEND_TO_PRINTER PROC NEAR
;save char on stack
    PUSH AX
;repeat
;get status register contents
READY_CHECK: MOV DX,3FDH
    IN AL,DX
;until transmitter holding register empty
    TEST AL,20H
    JZ READY_CHECK
;put char in transmitter holding register
    MOV DX,3F8H
    OUT DX,AL
;restore character from stack
    POP AX
    RET
SEND_TO_PRINTER ENDP
READCHAR PROC NEAR
    MOV AH,1
    INT 21H
    RET
READCHAR ENDP
DISPLAY_NEWLINE PROC NEAR
    MOV AH,2
    MOV DL,LF
    INT 21H
DISPLAY_NEWLINE ENDP
CODE ENDS
END START

Figure 20.12
(cont.)

SELF-CHECK QUIZ

1. Why do you think 8086-family microprocessors make provision for so many I/O ports (numbered from 0 to 0FFFFH inclusive)?

2. What is wrong with:
   
   MOV AL,01100101B
   OUT 38ABH,AL

3. If the printer has run out of paper and is not online, what will be the contents of the printer status port in Figure 20.2? How would you test this with an 8086-family assembly language program fragment?

4. Draw a diagram showing the serial