1. SERIAL PORT

This laboratory work presents the principles of serial communication and describes the controllers used by the serial ports of personal computers. First, the serial communication model is introduced, the parameters of serial communication are presented, and the two basic types of serial communication, asynchronous and synchronous, are explained. Next, the serial port signals are presented, the methods for controlling the data flow are explained, and the types of connectors and cables are described. After presenting the main features of several UART chips, the registers of the 16x50 family of UART chips are described in detail. The proposed applications aim to communicate with an FPGA development board through the serial port.

1.1. Serial Communication Model

An example of serial communication system is illustrated in Figure 1.1.

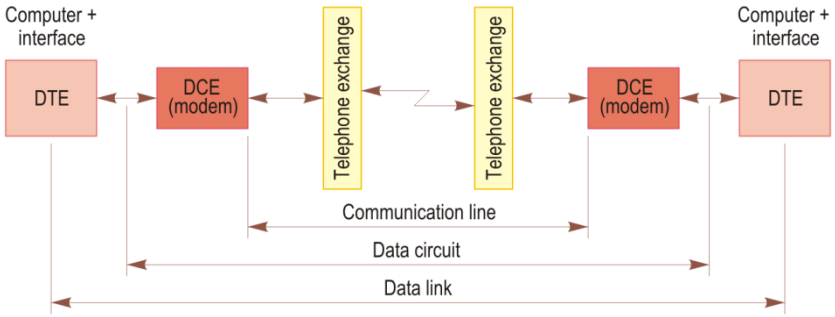

Figure 1.1. Serial communication system.

The components of a serial communication system are the following:

1. DTE – *Data Terminal Equipment* (computer, data terminal). It also includes the serial interfaces or the communication controller.
2. DCE – *Data Communication Equipment*. This equipment is called modem and allows the computer to send information over an analog telephone line. The main functions accomplished by a modem are the following:
   - Digital-to-analog conversion of the computer information and analog-to-digital conversion of the signals on the analog telephone line.
   - Modulation/demodulation of a carrier signal. For transmission, the modem superposes (modulates) the digital signals of the computer over the carrier signal of the telephone line. For reception, the modem extracts (demodulates) the information carried by the carrier signal and it transfers them to the computer.
3. The *communication line* represents a physical line or a telephone line. The telephone line can be a switched line (connected to a telephone exchange) or a leased line (dedicated).

4. The *data circuit* consists of the segment between two data terminal equipment, that is, the modems and the communication line. For short distances, it is possible a direct serial communication between two data terminal equipment via physical lines, without using modems. In this case, the data circuit is represented by these lines.
5. The *data link* contains the data circuit and the serial interfaces of the data terminal equipment.

Depending on the number of interconnected computers or devices, serial links can be classified into *point-to-point* links or *multi-point* links.

1.2. Parameters of Serial Communication

The *communication speed* (also referred to as *binary rate*) is measured in bits/s (bps):

$$BR = \frac{1}{T} \text{ [Bits/s]} \tag{1}$$

where T is the time period required for transmitting a single bit.

The modem represents the data signals by various electrical states, depending on the modulation type used: frequency, amplitude, or phase. Each electrical state is maintained at the output of the modem for a time interval called *modulation period* (Δ). The *modulation speed* is the reciprocal of the modulation period, representing the number of changes per second of the electrical state of the modem:

$$MS = \frac{1}{\Delta} \text{ [Baud]} \tag{2}$$

The unit of measure for modulation speed is *baud*, after the name of French engineer and telegrapher Jean-Maurice Baudot. The relationship between communication speed BR and modulation speed MS is the following:

$$BR = MS \cdot \log_2 n \text{ [Bits/s]} \tag{3}$$

where n is the number of distinct electrical states of the modem. In the particular case when there are only two distinct states of the modem, the communication speed is equal to the modulation speed. However, in general, there are a larger number of electrical states of the modem, so that the communication speed is a multiple of the modulation speed.

Modulation speed (baud rate) and communication speed (binary rate) are often confused. Modulation speed is the rate at which the electrical states of the modem change in one second. For instance, if frequency modulation is used, and the carrier signal frequency can be changed by the modem at a rate of 2,400 times per second, the modulation speed is 2,400 bauds. Early modems encoded a bit of 0 by a certain frequency and a bit of 1 by another frequency. In this particular case, the modulation speed has the same value as the communication speed. In general, though, modems encode several bits of information by an electrical state. For instance, if the modem encodes 4 bits of information by a certain frequency, for the previous example the communication speed would be $4 \times 2{,}400 = 9{,}600$ bits/s.

1.3. Types of Serial Communication

Considering the *direction of transfer*, the following types of communication may be distinguished:

- Simplex;
- Half duplex;
- Full duplex.

In a *simplex* communication, data are always sent in the same direction, from the transmitter equipment to the receiver. In a *half duplex* communication, each data terminal

equipment operates alternatively as transmitter, and then as receiver. For this type of connection, a single transmission line (two wires) is sufficient. In a *full duplex* communication, data are transferred simultaneously in both directions. Early full duplex connections required two transmission lines (four wires), but later connections require a single line.

From the point of view of the *synchronization* between the transmitter and the receiver, there are two types of serial communication:

- Asynchronous;
- Synchronous.

1.3.1. Asynchronous Communication

To ensure synchronization between the transmitter and the receiver, each character sent is preceded by a START bit, with a logical 0 value ("*space*"), and is followed by at least one STOP bit, with a logical 1 value ("*mark*"). Therefore, the START and STOP bits surround each character sent; the character sent between these two bits represents a data *frame*. This frame represents the basic digital information in a serial communication system. With asynchronous communication, the time interval between sending two successive characters is variable, and during this time the communication line is in the logical 1 state. This communication mode is also called *start-stop*.

Synchronization at the bit level is achieved with the aid of local clock signals of the same frequency. When the receiver detects the beginning of a character indicated by the START bit, it sets off a local clock oscillator, which enables correct sampling of individual bits of the character. Bit sampling is performed approximately in the middle of the interval corresponding to each bit.

Figure 1.2 illustrates the transmission of the character with ASCII code 0x61. After the START bit, with duration T corresponding to one bit, character transmission starts with the least significant bit b_0. After transmitting the most significant bit b_7, a parity bit p is transmitted; in this example, the parity is odd. The parity bit is optional, and when it is appended to the character sent, the parity can be selected to be even or odd. It is also possible to set the parity bit to 0 or 1, irrespective of the actual parity of the character. In the example illustrated, at the end of the character two STOP bits s_1 and s_2 are transmitted, and then the line remains in the logical 1 state for an undefined time. This time corresponds to an interval of silence.

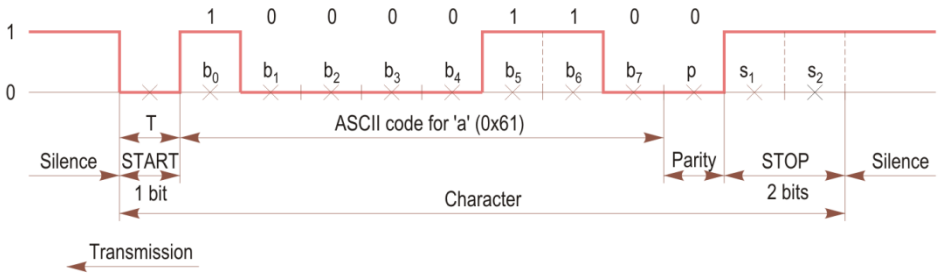

Figure 1.2. Asynchronous communication.

With asynchronous communication, bit synchronization is assured only during the actual transmission of each character. Such a communication is character-oriented and has the disadvantage that it requires extra information of at least 25% to identify each character.

1.3.2. Synchronous Communication

With synchronous communication, a frame does not contain a single character, but a *block* of characters or a *message*. Bit synchronization must be permanently assured, not only during actual transmission, but also during periods of silence. Therefore, time is continuously divided into elementary intervals at the transmitter, intervals which must then be retrieved at the receiver. This could cause some problems. If the local clock of the receiver is slightly

different in frequency from that of the receiver, errors could occur in recognizing characters, because of the length of transmitted blocks of characters.

In order to avoid such errors, the receiver clock must be resynchronized frequently with that of the transmitter. This can be done by ensuring that there are sufficient transitions from 1 to 0 and from 0 to 1 in the transmitted message. If the data to be transmitted consists of long strings of 1's and 0's, suitable transitions must be inserted to resynchronize the clocks. Such techniques are difficult to implement, and therefore a technique called synchronized-asynchronous communication (simply called synchronous communication) is used instead.

This communication type is characterized by the fact that, even though the message is transmitted in a synchronous manner, there is no synchronization during the time interval between two messages. The information is transmitted as character blocks or successive bits, without START and STOP bits. In order to readjust the local oscillator at the start of a message, each message is preceded by a number of special synchronization characters, for instance, the SYN character (0x16h). To maintain synchronization, additional synchronization characters may be inserted into the transmitted message, at certain time intervals.

At the receiver there are three levels of synchronization:

- Bit synchronization, using PLL (*Phase–Locked Loop*)[1] circuits, based on the existing transitions in the received signal;
- Character synchronization, ensured by recognizing certain synchronization characters;
- Block or message synchronization, depending on the data protocol used.

1.4. The RS-232C Standard

The electrical specifications of the serial port used on the IBM PC computers have been defined in the RS-232C (*Reference Standard No. 232, Revision C*) standard, developed in 1969 by the USA Standards Committee, today known as the *Electronic Industries Alliance* (EIA). The standard has been developed for digital communication between a computer and a remote terminal or between two terminals, without using a computer. The terminals were connected through telephone lines, so that modems were needed on both ends of the communication line.

The RS-232C standard has suffered various changes, and several revisions of it have been developed. For instance, in 1987 a new revision of the standard has been developed, named EIA RS-232D. In 1991, the EIA and the *Telecommunications Industry Association* (TIA) have developed revision E of the standard (EIA/TIA RS-232E). The current revision is EIA RS-232F, released in 1997. Nevertheless, in spite of its revision, usually the standard is named RS-232C or RS-232.

In Europe, the equivalent version of the RS-232C standard is V.24, developed by the CCITT (*Comité Consultatif International pour Téléphonie et Télégraphie*) committee. The name of this committee has been changed in the early 1990's to *International Telecommunications Union* (ITU). Both standards specify the signals used for communication, the voltage levels, the protocol used for data flow control, and the connectors of the serial interface.

The RS-232C standard defines both an asynchronous and a synchronous communication. Details such as character encoding (ASCII, Baudot, EBCDIC), character framing (character length, number of stop bits, parity) are not defined, nor the communication speeds, although the standard is intended for speeds lower than 20,000 bits/s. However, current equipment allows higher communication speeds, using voltage levels that are compatible with those specified by the standard. Serial ports of the computers usually allow to select one of the following communication speeds: 150; 300; 600; 1,200; 2,400; 4,800; 9,600; 19,200; 38,400; 57,600; 115,200 bits/s.

[1] A PLL circuit represents a closed loop system for controlling the frequency of an oscillator. Its operation is based on detecting the phase difference between the input and output signals of the controlled oscillator.

A basic RS-232C link only requires three connections: one for transmit, one for receive, and one for the common signal ground. However, most of the serial links also use signals for data flow control.

Unlike other types of serial communication that are differential[2], the RS-232C communication is single-ended, using a single wire for each signal. Although this simplifies the circuitry required by the interface, at the same time the maximum communication distance is also reduced in the case of a direct link, without using modems. The RS-232C standard specifies a maximum distance of 15 m. The distance can be increased if lower communication speeds are used.

The electrical voltages specified by the RS-232C standard are the following:

- The logical 0 value corresponds to a positive voltage between +3 V and +25 V;
- The logical 1 value corresponds to a negative voltage between –3 V and –25 V.

1.5. Serial Interface Signals

The serial interface uses signals for transmitting and receiving data, as well as signals for controlling the flow of data between a data terminal equipment and a data communication equipment. Figure 1.3 presents the serial interface signals according to the RS-232C standard. The link illustrated is the one for which the serial port was originally developed, that is, connecting a modem to the computer. The figure shows the pin numbers of the DB-25 connectors used for such connections via modems; the numbers in parentheses represent the signals according to the V.24 standard.

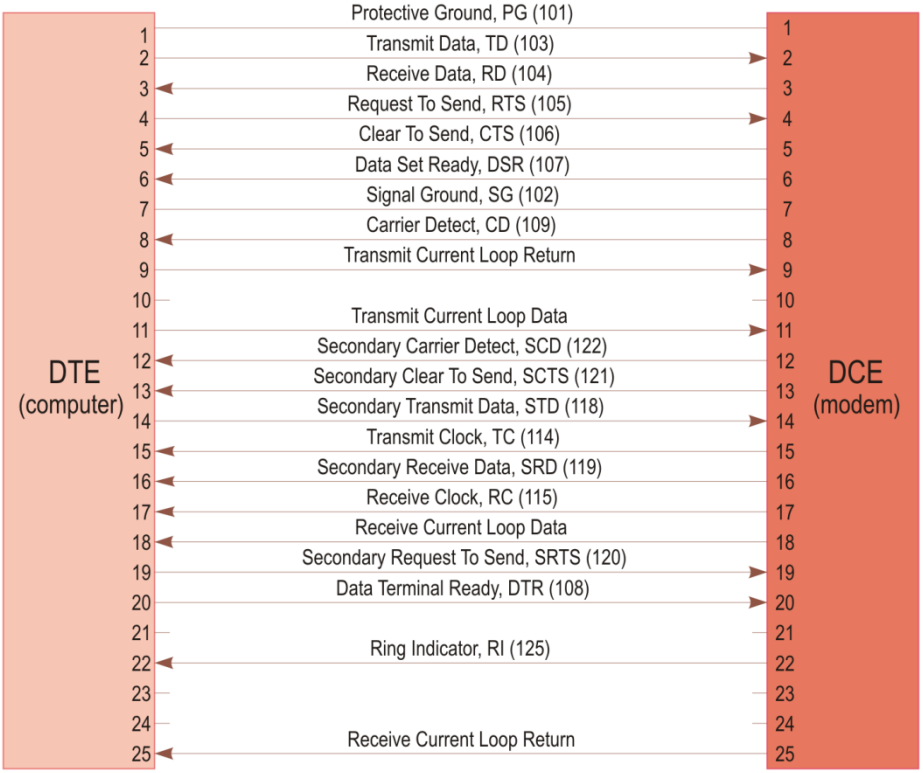

Figure 1.3. Serial interface signals.

The most important signals are described next.

[2] A differential communication uses a pair of wires for each signal. Examples of interfaces that use differential communication are the RS-422 and RS-485 interfaces, as well as the USB and IEEE 1394 buses.

Transmit Data, TD

Data are sent serially on this line by the computer. After the start bit, the least significant bit of a character is sent. In general, for data transmission the *RTS*, *CTS*, *DTR*, and *DSR* signals need to be asserted. These signals are asserted in a sequence of operations for establishing the link with the modem.

Receive Data, RD

This line is used by the computer to receive data from the modem or from an external equipment.

Data Terminal Ready, DTR

When the computer is turned on and is ready for the data communication, it asserts the *DTR* signal. The modem will answer to the *DTR* signal by asserting the *DSR* signal.

Data Set Ready, DSR

When the modem or the external equipment is turned on and is ready for the data communication, it asserts the *DSR* signal. This signal is asserted by the modem as an answer to the assertion of the *DTR* signal by the computer. The computer will send data to the modem only when the *DSR* signal is asserted.

Request To Send, RTS

When the computer is ready for data transmission, it asserts the *RTS* signal. This signal indicates to the modem that it can send data to the computer. A de-asserted *RTS* signal will prevent the modem from sending data to the computer. This allows the computer to control the flow of data sent by the modem. The answer to the *RTS* signal is received by the computer on the *CTS* line.

Clear To Send, CTS

By asserting this signal, the modem or the external equipment indicates that it is ready to receive data from the computer. The *CTS* signal is asserted by the modem as an answer to the assertion of the *RTS* signal by the computer. A de-asserted *CTS* signal will prevent the computer from sending data to the modem. This allows the modem to control the flow of data sent by the computer.

Carrier Detect, CD

By asserting this signal, the modem signals to the computer that it has detected the carrier signal of another modem on the telephone line, that is, there is a connection with a remote modem. In a serial link without modems, the assertion of the *CD* signal indicates that it is possible to communicate with an equipment at the other end of the line. Often, this signal is ignored by the computer.

Ring Indicator, RI

When the modem detects on the telephone line the ringing signal from another modem, it asserts the *RI* signal. This signal allows the program running on the computer to automatically answer a remote telephone call.

Transmit Clock, TC

Represents the transmit clock signal provided to the computer by the modem in a synchronous communication. It is not used for asynchronous communication.

Receive Clock, RC

Represents the receive clock signal provided to the computer by the modem in a synchronous communication. It is not used for asynchronous communication.

Transmit Current Loop Data
Transmit Current Loop Return

Receive Current Loop Data
Receive Current Loop Return

These signals allow communication between equipment close to one another, without using modems. The method used is referred to as *current-loop* transmission. The logical 0 level is indicated by a current of 20 mA, and a logical 1 level is indicated by the absence of this current. Connecting an equipment that uses current-loop communication to an RS-232C compatible serial port requires a voltage-level shifter.

Secondary Transmit Data, STD
Secondary Receive Data, SRD

Represent the data signals for a secondary serial link. Although theoretically two duplex serial links are possible through a single cable, in practice the second link is only rarely implemented.

Secondary Request To Send, SRTS
Secondary Clear To Send, SCTS
Secondary Carrier Detect, SCD

Represent the *RTS*, *CTS*, and *CD* signals, respectively, for the secondary serial link.

The sequence for establishing the link between the computer and the modem consists of the following operations: software asserts the *DTR* signal and waits the answer from the modem, represented by assertion of the *DSR* signal. Next, software asserts the *RTS* signal and waits the answer from the modem, represented by assertion of the *CTS* signal.

1.6. Data Flow Control

To be possible the communication between devices with different speeds, the designers of the serial interface provided special signals for data flow control. These signals allow a device to stop and then to resume the data transmission when requested by the device located at the other end of the serial communication line. Besides this *hardware method* for data flow control, there is a *software method* as well, based on the transmission of special characters between the two devices. When the receiving device (e.g., a printer) cannot accept more data because its buffer is full, it sends a particular control character to the transmitting device (e.g., to the computer). When the receiving device can accept new data, it sends another control character that signals to the transmitting device that it may resume the data transmission.

Usually, the control method that will be used by the computer may be selected via the software driver of the serial controller. Some programs may use by default a particular method. For the peripherals, the control method may be selected either via the software or with a switch. It is important to use the same control method for both the computer and the peripheral in order to avoid data loss.

1.6.1. Hardware Control

The hardware control method assumes to use a communication protocol by means of the serial interface control signals. The protocol used is based on a serial communication via modems and a telephone line, for which the original serial interface was developed. This protocol implies establishing the connection between two modems via the telephone line and maintaining the data flow between them while the connection is active. The phases of this protocol are described next. In a simplified form, this protocol may also be used for a direct serial communication between two devices, without using modems and a telephone line.

1. When a remote modem desires to establish connection with the local modem, it sends the ring signal over the telephone line. This signal is detected by the local modem, which asserts the *RI* signal to inform the local computer on the existence of a telephone call.

2. When the assertion of *RI* signal is detected, a communication program is launched on the local computer. This program indicates the readiness of the computer to start the communication by asserting the *DTR* signal.
3. When the local modem detects that the data terminal (the computer) is ready, it answers the telephone call and waits the activation of the carrier signal by the remote modem. When the local modem detects the carrier signal, it asserts the *CD* signal.
4. The local modem negotiates with the remote modem a connection with certain parameters. For instance, the two modems may determine the optimal communication speed based on the telephone link quality. After this negotiation, the local modem asserts the *DSR* signal.
5. When detecting the assertion of the *DSR* signal, the local computer software asserts the *RTS* signal to indicate to the modem that it may send data to the computer.
6. When the modem detects the assertion of the *RTS* signal, it asserts the *CTS* signal to indicate that it is ready for receiving data from the computer.
7. Next, the data are transferred in both directions between the remote devices, on the *TD* and *RD* lines.
8. Since the speed of telephone line is lower than that of the link between the computer and the local modem, the modem buffer will fill up. The local modem requests the computer to stop transmitting data by de-asserting the *CTS* signal. When the buffer empties, the modem re-asserts the *CTS* signal.
9. If the computer cannot accept more data from the modem, it de-asserts the *RTS* signal. When the computer can accept again data from the modem, it re-asserts the *RTS* signal.
10. At the end of the communication session, the carrier signal is de-activated, and the local modem de-asserts the *CD*, *CTS*, and *DSR* signals.
11. When detecting the de-assertion of the *CD* signal, the local computer de-asserts the *RTS* and *DTR* signals.

From the above protocol, it follows that:

- The computer must detect the assertion of *DSR* and *CTS* signals before transmitting data to the modem. De-assertion of any of these signals will usually stop the flow of data from the computer.
- The modem must detect the assertion of the *DTR* and *RTS* signals before transmitting data over the serial line or to the computer. De-assertion of the *DTR* signal will stop the transmission of data over the serial line, and de-assertion of the *RTS* signal will stop the transmission of data to the computer.

The status of the *CD* signal is not interpreted by all of the serial communication systems. In some systems, the *CD* signal must be asserted before the data terminal will start to transmit data. In other systems, the status of the *CD* signal is simply ignored.

1.6.2. Software Control

The software method for data flow control involves sending some control characters between the two devices. For instance, the peripheral will send a particular control character to indicate that it cannot accept more data from the computer and will send another control character to indicate that transmission of data may be resumed by the computer. There are two variants of this method. The first variant uses the XON/XOFF control characters, and the second variant uses the ETX/ACK control characters.

When using the XON/XOFF variant, the peripheral sends the XOFF character to indicate that its buffer is full and data transmission should be stopped by the computer. This

character is also denoted DC1 (*Device Control* 1) and has an ASCII code of 0x13, equivalent to the Ctrl-S character. The Ctrl-S character may also be entered by the user in some communication programs to stop the data transmission by a device connected to the computer. When the peripheral is ready to receive new data, it sends the XON character to the computer. This character is also denoted DC3 (*Device Control* 3) and has an ASCII code of 0x11, equivalent to the Ctrl-Q character. In some communication programs, entering the Ctrl-Q character cancels the effect of the Ctrl-S character.

When using the ETX/ACK variant, sending the ETX (*End of TeXt*) character by the peripheral indicates that data transmission should be stopped by the computer. This character has an ASCII code of 0x03 and is equivalent to the Ctrl-C character. Sending the ACK (*ACKnowledge*) character indicates the possibility to resume the data transmission by the computer. This character has an ASCII code of 0x06 and is equivalent to the Ctrl-F character.

1.7. Connectors

Serial ports may use either of two types of connectors. The 25-pin DB-25 connector has been used by previous-generation computers. Newer computers use the 9-pin DB-9 connector. For the serial ports of computers male connectors are used, and for the serial ports of peripheral devices female connectors are used.

Figure 1.4 illustrates the DB-25 connector of the serial port.

Figure 1.4. The DB-25 connector used for the serial ports of previous-generations personal computers.

Out of the 25 signals of the DB-25 connector, at most 10 signals are used for a common serial connection. Table 1.1 shows the names of these signals and their assignments to the DB-25 connector pins.

Table 1.1. Signal assignments to the DB-25 connector pins of the serial port.

Pin	Signal	Meaning	← In → Out
1	PG	Protective Ground	
2	TD	Transmit Data	→
3	RD	Receive Data	←
4	RTS	Request To Send	→
5	CTS	Clear To Send	←
6	DSR	Data Set Ready	←
7	SG	Signal Ground	
8	CD	Carrier Detect	←
20	DTR	Data Terminal Ready	→
22	RI	Ring Indicator	←

To reduce the space occupied by the serial port connector, the DB-25 connector has been replaced with a smaller connector, the 9-pin DB-9 connector (Figure 1.5).

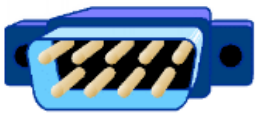

Figure 1.5. The DB-9 connector used for the serial ports of personal computers.

Table 1.2 shows the serial port signal assignments to the DB-9 connector pins.

Table 1.2. Signal assignments to the DB-9 connector pins of the serial port.

Pin	Signal	Meaning	← In → Out
1	CD	Carrier Detect	←
2	RD	Receive Data	←
3	TD	Transmit Data	→
4	DTR	Data Terminal Ready	→
5	SG	Signal Ground	
6	DSR	Data Set Ready	←
7	RTS	Request To Send	→
8	CTS	Clear To Send	←
9	RI	Ring Indicator	←

1.8. Cables

There are several variants of cables that can be used for serial communication. For low communication speeds and shorts lengths, normal cables can be used, that are not shielded. To reduce interferences with other devices, shielded cables should be used, which contain a coat of aluminum foil. Ideally, the shield of the cable should be connected to the protective ground of the connector, if it is a DB-25 connector. The DB-9 connector does not include a pin for the protective ground. When using this type of connectors, the shield of the cable can be connected to the signal ground.

Notes

- With a serial cable that uses DB-25 connectors, the electrical ground or *signal ground SG* is separated from the chassis ground or *protective ground PG*. The protective ground is directly connected to the shield of the connector (and of the device), and it has a protective function. By making this connection, the metallic cases of the two devices connected through the serial cable will be at the same potential, avoiding voltage differences to build between the two devices, voltages that can be dangerous for them. Often, the protective ground connection is missing from the serial cables.
- The protective ground *PG* should never be connected to the signal ground *SG*.

The signals of the serial interface were provided in order to connect a data terminal equipment (DTE) to a data communication equipment (DCE). When connecting two such equipment, e.g., a computer to a modem, a cable that connects pins with the same numbers of the connectors at the two ends is required. This is called a *straight-through cable*. When connecting two equipment with different connectors, an *adapter cable* is required. When connecting two data terminal equipment, e.g., two computers, the data sent on the *TD* pin of one of the equipment must be received on the *RD* pin of the other equipment. Therefore, the connections of these pins have to be inverted at the two ends of the cable; this is called a *crossover cable*.

1.8.1. Straight-Through Cables

In a straight-through cable, each pin corresponding to a particular signal at one end is connected to the pin corresponding to the same signal at the other end. Frequently, not all of the serial interface signals are used. Even when the data flow is controlled with the hardware method, only nine signals need to be connected between a computer and a modem, assuming an asynchronous communication. For instance, if DB-25 connectors are used at both ends of the cable, the pins that must be connected are: 2, 3, 4, 5, 6, 7, 8, 20, and 22. For a synchronous communication, two additional connections are needed, representing the clock signals for transmit and receive, which are generated by the synchronous modem. If DB-9 connectors are used, all the pins should be connected, except for pin 9 (*RI* signal).

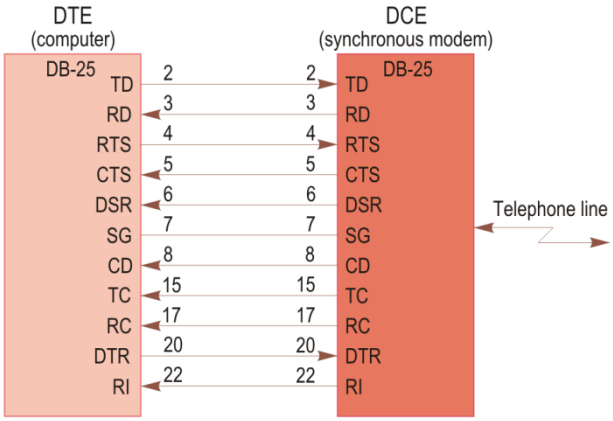

Figure 1.6. Connecting a synchronous modem to a computer (DB-25 connectors).

Figure 1.6 illustrates the connections required for a serial cable when connecting a synchronous modem to a computer, assuming that DB-25 connectors are used by both devices.

1.8.2. Adapter Cables

When the two connectors of a serial link are of different types, for instance, DB-25 at one end and DB-9 at the other end, an adapter cable is required. Although present-day IBM PC computers use only DB-9 connectors, some peripherals, such as serial printers, plotters, or modems, use DB-25 connectors. The adapter can take the form of a small assembly with two connectors of different types, or of an adapter cable with different types of connectors at the two ends.

Table 1.3 presents the connections required for an adapter cable between a DB-25 connector and a DB-9 connector.

Table 1.3. Connections for an adapter cable between a DB-25 connector and a DB-9 connector.

DB-25 Pin	DB-9 Pin	Signal	Meaning
2	3	TD	Transmit Data
3	2	RD	Receive Data
4	7	RTS	Request To Send
5	8	CTS	Clear To Send
6	6	DSR	Data Set Ready
7	5	SG	Signal Ground
8	1	CD	Carrier Detect
20	4	DTR	Data Terminal Ready
22	9	RI	Ring Indicator

1.8.3. Crossover Cables

Crossover cables are needed for connecting two data terminal equipment, such as two computers. These cables are also needed for connecting some types of peripherals to the computer, such as serial printers and plotters, because when the serial interface was designed, these peripherals were considered data terminals.

Next, two types of crossover cables are presented. The first type can be used when the data flow control is accomplished with the software method, while the second type can be used when the data flow control is accomplished with the hardware method.

Many serial communication systems do not use all the signals for data flow control, so that for these systems the connections may be simplified. In the simplest case, when the software method is used for data flow control, only three connections are required, for transmitted data, received data, and signal ground. For instance, if DB-25 connectors are used, the pins that need to be connected are 2, 3, and 7, and if DB-9 connectors are used, these pins are 2, 3, and 5. Such a cable is called *null-modem* cable.

A *null-modem* cable can be used for directly connecting two data terminal equipment, e.g., two computers. The connections are made so that, from the point of view of the computer, the communication is carried on as though at the other end were a modem, and not another computer. The data sent by the first computer must be received by the second computer, so that the *TD* pin on the first computer is connected to the *RD* pin on the second computer, and vice versa. The two pins for the signal ground *SG* must be connected together. In both connectors, the *DTR* pin is connected to the *DSR* and *CD* pins at the same end of the link. Therefore, when the *DTR* signal is asserted, the *DSR* and *CD* signals are also asserted. The *DSR* and *CD* signals are thus asserted the same way as though at the other end of the link were a modem. Similarly, in both connectors the *RTS* pin is connected to the *CTS* pin at the same end of the link. Since the computers communicate at the same speed, the data flow control is not needed.

Figure 1.7 illustrates the connections required for a *null-modem* cable, assuming that DB-9 connectors are used at both ends.

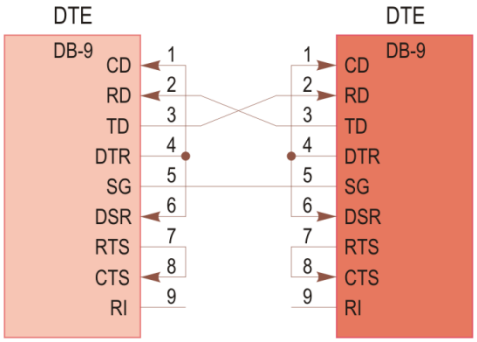

Figure 1.7. Connecting two data terminal equipment via a *null-modem* cable (DB-9 connectors).

When the hardware method is used for data flow control, the three lines of the *null-modem* cable presented earlier are not sufficient. In this case, a crossover cable with additional connections for this control method must be used. The data pins and the signal ground pin are connected in the same way as for the *null-modem* cable. In both connectors, the *DTR* pin is connected to the *DSR* and *CD* pins at the other end of the link. By this connection, each of the two data terminals may determine when the other data terminal equipment is ready. The signals used for data flow control are *RTS* and *CTS*. In both connectors, the *RTS* pin is connected to the *CTS* pin at the other end of the link. This connection ensures the data flow control with the hardware method, as described in Section 1.6.1.

Figure 1.8 illustrates the connections required for a crossover cable that allows a hardware data flow control, assuming that DB-9 connectors are used at both ends.

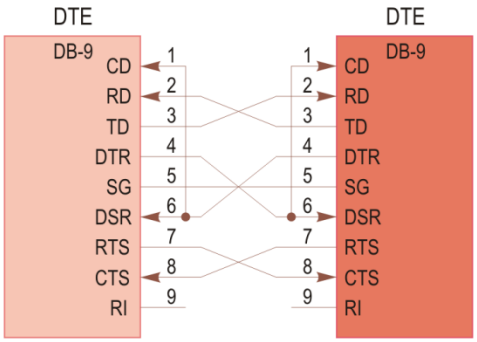

Figure 1.8. Connecting two data terminal equipment via a crossover cable with hardware data flow control (DB-9 connectors).

Note

- Often, the crossover cable with the connections illustrated in Figure 1.8 is mistakenly called *null-modem* cable.

1.9. UART Chips

The main component of a serial port is an UART (*Universal Asynchronous Receiver/Transmitter*) chip. This chip performs the conversion of parallel data from the computer into the format required for the serial transmission and the conversion of serial data received into the parallel format used by the computer. The chip appends the start bit, the stop bit, and the parity bit to the serial data transmitted and it detects these bits in the serial data received.

In the original IBM PC and IBM PC/XT computers, UART chips from the 8250 family have been used. Starting with the first 16-bit systems, the UART 16450 chip, manufactured by *National Semiconductor*, has been used. This chip is compatible with the 8250-family chips at the register level, but allows communication at higher speeds. In the IBM PS/2 computers, the UART 16550 chip has been used, which has also been adopted for systems based on the 80386 and later processors. The 16550 chip works similarly to the 8250 and 16450 chips, but contains in addition two 16-byte FIFO memories for transmit and receive. This memory allows communication at higher speeds compared to the speeds allowed by previous chips. Starting with the Pentium processor-based systems, the UART 16550 chip (or a later version of this) has been included into the motherboard chipset. Improved versions of the UART 16550 chips are the 16650, 16750, and 16850 chips.

In the following, the main features of some UART chips are described.

8250

The 8250 chip has been used in the first IBM PC computers. The chip did not include a transmit buffer nor a receive buffer, and for this reason the communication speed provided was low. The chip had several bugs. The IBM PC and PC/XT computers' ROM BIOS were designed to take into account these bugs.

8250A

In this version, some bugs of the UART 8250 chip have been corrected, including one bug in the interrupt enable register. Because the ROM BIOS of IBM PC and PC/XT computers expected this bug, the 8250A chip did not work correctly with those computers. The 8250A chip did work with the IBM PC/AT computers, but did not work properly at speeds of 9,600 bits/s and higher, because of the lack of transmit and receive buffers.

8250B

In this last version of the 8250-family chips, the bugs found in the two previous versions have been corrected. The interrupt enable register bug in the original 8250 version has been put back into the chip, in order to operate in the way expected by the ROM BIOS of IBM PC and PC/XT computers. This chip did not work properly at speeds of 9,600 bits/s and higher.

16450

This chip has been used in the first IBM PC/AT computers. The chip allowed higher communication speeds due to transmit and receive buffers of one byte each. In this chip, a working register of one byte has been added to the register set, used as temporary storage.

16550

Represents an improved version of the 16450 chip, containing transmit and receive buffers of 16 bytes each, organized as FIFO memories. The chip also allows transfers by multiple DMA channels. The initial version of this chip did not allow to use the FIFO memories, bug that has been corrected in the 16550A version. The last version manufactured by *National Semiconductor* is 16550D. By using the FIFO memories, the communication speeds can be increased significantly, eliminating the possibility to loose characters at higher speeds. The maximum communication speed allowed by the 16550 chip is 115,200 bits/s.

16650, 16750, and 16850

Several improved versions of the 16550 chip have been produced, which are compatible with it, but they contain larger FIFO memories:

- The 16650 chip includes two FIFO memories of 32 bytes each;
- The 16750 chip includes two FIFO memories of 64 bytes each;
- The 16850 chip includes two FIFO memories of 128 bytes each.

These chips allow higher communication speeds, of 230.4 Kbits/s (16650), 460.8 Kbits/s (16750), and 921.6 Kbits/s (16850). The use of these chips is recommended with high-speed external serial links, such as those carried out with an ISDN adapter.

1.10. The Serial Ports of Personal Computers

The original IBM PC computers' BIOS software allowed to use two serial ports, named COM1 and COM2. Later on, the number of ports has been extended with other two ports, named COM3 and COM4. Starting with the *Windows* 95 operating system, the number of serial ports has been extended to 128. These ports are managed with the device drivers that control them.

Access to the serial ports can be accomplished with BIOS functions (interrupt 0x14), with operating system functions, or directly via the registers of the UART chips. Each UART chip associated with a serial port contains a number of eight I/O registers starting from the serial port's base address. The BIOS stores the base addresses of COM1..COM4 serial ports into four successive 16-bit words, from address 0x0000:0x0400 corresponding to port COM1.

The base addresses of COM1..COM4 serial ports are shown in Table 1.4. In general, the addresses of COM1 and COM2 ports are fixed, and have the values shown in this table. The addresses of COM3 and COM4 ports may be different from those shown. In Table 1.4 the interrupt levels used by COM1..COM4 serial ports are also shown.

Table 1.4. Standard assignment of base addresses and interrupt levels to the serial ports.

Serial Port	Base Address	Interrupt
COM1	0x3F8	IRQ 4
COM2	0x2F8	IRQ 3
COM3	0x3E8	IRQ 4
COM4	0x2E8	IRQ 3

In principle, each serial port requires its own interrupt level. When there are more than two serial ports in the computer, it might be necessary to share some interrupt levels. For instance, the COM3 port shares the level-4 interrupt (IRQ 4) with the COM1 port, and the COM4 port shares the level-3 interrupt (IRQ 3) with the COM2 port. The PCI bus allows to share the interrupt levels, so that for expansion boards based on the PCI or PCI Express bus it is possible to use a single interrupt level without conflicts.

1.11. The Registers of UART 16x50 Chips

The registers of UART chips are accessible with I/O instructions. For the first serial port, the registers have addresses between 0x3F8 and 0x3FF, while for the second port the registers have addresses between 0x2F8 and 0x2FF. The first two addresses allow access to several registers of the chip. There are some registers that are not accessible by software.

The addresses of software-accessible registers for the first two serial ports, their access modes (R – read, W – write, R/W – read/write), abbreviations and names are presented in Table 1.5. The column labeled *Offset* indicates the displacement of each register's address relative to the serial port's base address. The column labeled DLAB (*Divisor Latch Access Bit*) represents the value of bit 7 of the line control register (LCR). When set to 1, this bit allows access to two registers used for setting the communication speed.

Table 1.5. The registers of UART 16x50 chips.

COM1	COM2	Offset	DLAB	Access	Abbrev.	Name
0x3F8	0x2F8	+ 0	0	W	THR	Transmitter Holding Register
			0	R	RBR	Receiver Buffer Register
			1	R/W	–	Divisor Latch Register LSB
0x3F9	0x2F9	+ 1	0	R/W	IER	Interrupt Enable Register
			1	R/W	–	Divisor Latch Register MSB
0x3FA	0x2FA	+ 2	X	R	IIR	Interrupt Identification Register
			X	W	FCR	FIFO Control Register
0x3FB	0x2FB	+ 3	X	R/W	LCR	Line Control Register
0x3FC	0x2FC	+ 4	X	R/W	MCR	Modem Control Register
0x3FD	0x2FD	+ 5	X	R	LSR	Line Status Register
0x3FE	0x2FE	+ 6	X	R	MSR	Modem Status Register
0x3FF	0x2FF	+ 7	X	R/W	–	Scratch Register

Note

- The 16650, 16750, and 16850 chips contain additional registers in comparison with those indicated in Table 1.5. The structure of these registers may vary from one manufacturer to another, reason for which these registers are not described in this laboratory work.

THR - Transmitter Holding Register

Represents the transmit buffer, which is selected if the DLAB bit of LCR register is 0. The character to be sent must be written to this register. If the *Transmitter Shift Register* (TSR) is emptied (that is, the chip may begin sending a new character), the contents of THR register (or, when the FIFO memories are enabled, a byte from the transmit FIFO memory) is transferred into the TSR register and transmission of the character begins.

If the THR register is emptied (that is, may be written with a new character), the chip generates an interrupt request if generation of this type of interrupt is enabled. The status of this register can be determined by testing bit 5 of the LSR register.

TSR - Transmitter Shift Register

It is an internal register not accessible by software. When the transmission of a character is finished, the contents of THR register (or, when the FIFO memories are enabled, a byte from the transmit FIFO memory) is transferred automatically into the TSR register and its transmission begins.

If the TSR register is emptied (that is, transmission of a new character may begin), but the THR register is empty (or, when the FIFO memories are enabled and the transmit FIFO memory is emptied below a certain threshold level), the chip generates an interrupt request when this type of interrupt is enabled. The status of this register can be determined by testing bit 6 of the LSR register.

RBR - Receiver Buffer Register

Represents the receive buffer, which is selected if the DLAB bit of the LCR register is 0. A received character is placed into this register. The presence of a character in the RBR register may be determined by testing bit 0 of the LSR register. If the FIFO memories are disabled, the received character must be read by the software from the RBR register before a new character is received; otherwise, an overrun error will occur. The reception a character generates an interrupt request, if this type of interrupt is enabled. If the FIFO memories are enabled, it is possible that the receive interrupt will be generated only after receiving a certain number of characters into the receive FIFO memory. This threshold value can be programmed by setting bits 7..6 of the FCR register.

RSR - Receiver Shift Register

It is an internal register not accessible by software. Each character is received into this register. When the reception of a character is finished, the contents of the RSR register is

transferred automatically into the RBR register if the FIFO memories are disabled. If these memories are enabled, the characters received are placed into the receive FIFO memory.

Divisor Latch Register LSB & MSB

These registers contain the value by which the 16x50 chip's internal clock frequency (1.8432 MHz) should be divided to get the desired binary rate. The LSB register contains the least significant byte of the divisor, and the MSB register contains the most significant byte of the divisor. The two divisor registers are accessible if the DLAB bit of the LCR register is 1. For computing the value of the divisor, the clock factor of the 16x50 chip should be taken into account. Usually, this factor is 16 (the binary rate is 16 times lower than the frequency obtained by division). The following formula may be used:

$$Divisor = 1{,}843{,}200 / (BinaryRate * 16)$$

Table 1.6 contains the divisors corresponding to various binary rates.

Table 1.6. Divisors of the 1.8432 MHz frequency for various binary rates.

Binary Rate (Bits/s)	Divisor	Binary Rate (Bits/s)	Divisor
50	0x0900	4,800	0x0018
150	0x0300	9,600	0x000C
300	0x0180	19,200	0x0006
600	0x00C0	38,400	0x0003
1,200	0x0060	57,600	0x0002
2,400	0x0030	115,200	0x0001

IER - Interrupt Enable Register

This register is accessible if the DLAB bit of the LCR register is 0. The UART chip may generate five types of interrupt requests, with different priority levels. The IER register allows to individually enable these interrupts. The structure of this register is illustrated in Figure 1.9.

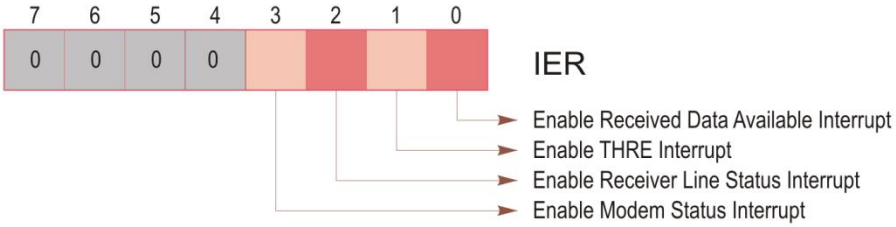

Figure 1.9. Interrupt Enable Register (IER).

The bits of the IER register are described next.

- Bits 7..4 are not used (set to 0).
- Bit 3 (Enable Modem Status Interrupt) enables by value 1 the generation of an interrupt request when the modem status register (MSR) changes.
- Bit 2 (Enable Receiver Line Status Interrupt) enables by value 1 the generation of an interrupt request when the line status register (LSR) changes, usually, when receive errors occur.
- Bit 1 (Enable Transmitter Holding Register Empty Interrupt) enables by value 1 the generation of an interrupt request when the THR register is empty, that is, after transferring the contents of this register to the TSR register. When this interrupt occurs, a new character can be written into the THR register.
- Bit 0 (Enable Received Data Available Interrupt) enables by value 1 the generation of an interrupt request when a character is received. If the FIFO memories are enabled,

this bit also enables the generation of the time-out interrupt. This interrupt is described in the next section.

IIR - Interrupt Identification Register

This register is read-only. Although the interrupts generated by the 16x50 chip occur on a single interrupt level (IRQ 4 for the first serial port and IRQ 3 for the second port), these interrupts may have four types of causes, with different priority levels. Identification of the interrupt cause can be performed by testing bits 3..0 of the interrupt identification register (IIR). The other bits of this register indicate the status of FIFO memories.

The meanings of IIR register bits that allow to identify the cause of an interrupt are shown in Table 1.7.

Table 1.7. Bits of IIR register used to identify the cause of an interrupt.

Bits 3..0	Priority	Interrupt Type	Interrupt Cause	Interrupt Reset
0 0 0 1	–	–	No interrupts	–
0 0 0 0	3 (lowest)	Change of modem status	The CTS or DSR or RI or CD signal has changed	Read the modem status register (MSR)
0 0 1 0	2	End of character transmission	The THR register is empty	Read IIR or write a character to THR
0 1 0 0	1	Character received	The RBR register contains a character received or the receive FIFO memory is filled above the threshold level	Read the RBR register or empty the receive FIFO memory below the threshold level
1 1 0 0	1	Time-out	No characters have been read or placed from/into the receive FIFO during four characters and there is at least one character in the FIFO memory	Read the RBR register
0 1 1 0	0 (highest)	Change of line status	Overrun, framing, or parity error, or Break interrupt	Read the LSR register

The UART chip can also be used with software polling, testing periodically bit 0 of the interrupt identification register. If this bit is 0, an interrupt occurred.

The receive FIFO memory threshold level refers to the number of characters that must be received before the chip will generate a receive interrupt, if this type of interrupt is enabled. Setting this threshold level is described in the section dedicated to the FIFO memory control register (FCR).

On a receive operation with the FIFO memories enabled, the UART chip will generate an interrupt request even though the receive FIFO memory contains a lower number of characters than the threshold level, but no characters have been received in a time equal to the period needed to transfer four characters. This represents the time-out interrupt and it has been provided for cases when the transmitter stops transmitting characters in order to wait for an answer from the receiver. Without this interrupt, it would be possible that the receiver does not detect that characters have been received, since the receive FIFO memory does not contain a sufficient number of characters to generate a receive interrupt. The bit combination in the IIR register that indicates this interrupt cause may be ignored, because the UART chip will also indicate that characters are available in the receive buffer.

Bits 7..4 of the IIR register are described next.

- Bits 7..6 indicate the status of FIFO memories:

  00: There are no FIFO memories (the chip is an 8250 or 16450).
  01: Reserved combination.
  10: The FIFO memories are enabled, but unusable. This situation may occur with the 16550 chip, because of the bug in using the FIFO memories.
  11: The FIFO memories are enabled and operational.

- Bits 5..4 are reserved (set to 0).

FCR - FIFO Control Register

This register is available on the 16550 and later UART chips. The structure of the FCR Register is illustrated in Figure 1.10.

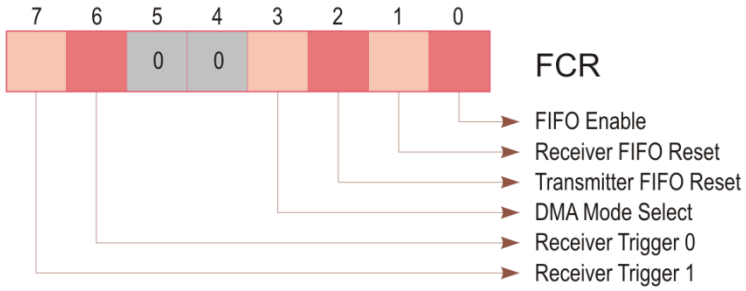

Figure 1.10. FIFO Control Register (FCR).

The bits of the FCR register are described next.

- Bits 7..6 (Receiver Trigger) allow to set the number of characters received in the receive FIFO memory after which the chip will generate a receive interrupt, if this type of interrupt is enabled. If this number is set to a value greater than 1, the chip will not generate an interrupt request after each character received, which will decrease the time needed for interrupt handling. The meaning of bits 7..6 is the following:

  00: The interrupt request is generated after each character received;
  01: The interrupt request is generated after 4 characters received;
  10: The interrupt request is generated after 8 characters received;
  11: The interrupt request is generated after 14 characters received.

- Bits 5..4 are reserved (set to 0).

- Bit 3 (DMA Mode Select), present on the 16550 and later chips, allows to select the mode of DMA transfers. When the FIFO memories are enabled, there are two DMA transfer modes that can be selected, mode 0 and mode 1. Mode 0, selected if bit 3 of the FCR register is 0, allows single-word DMA transfers. Mode 1, selected if bit 3 of the FCR register is 1, allows multi-word DMA transfers, which are performed continuously until the receive FIFO memory is emptied or the transmit FIFO memory is filled. On the 16450 chip, only mode 0 is allowed.

- Bit 2 (Transmitter FIFO Reset) allows to clear the transmit FIFO memory by setting this bit to 1.

- Bit 1 (Receiver FIFO Reset) allows to clear the receive FIFO memory by setting this bit to 1.

- Bit 0 (FIFO Enable) enables by value 1 the transmit and receive FIFO memories. By default, these memories are disabled, for compatibility with the 8250 and 16450 UART chips. By setting this bit to 0, the FIFO memories will be disabled, and the data stored to them will be lost.

Note

- When the FIFO memories are not used, or bits 7..6 of the FCR register are set so that an interrupt request will be generated after each character received, in the receive interrupt handling routine is enough to read the RBR receive register only once. If the FIFO memories are used, in the receive interrupt handling routine a program loop should be implemented to read characters from the RBR receive register one by one, as long as there is at least one character in this register. The presence of a character in the RBR receive register is indicated by the fact that bit 0 of the LSR register is set to

1. After reading a character from the RBR register, the chip will load automatically the next character from the FIFO memory, if there is any character in this memory.

LCR - Line Control Register

By writing the line control register it is possible to set the parameters of the serial communication. The structure of the LCR register is illustrated in Figure 1.11.

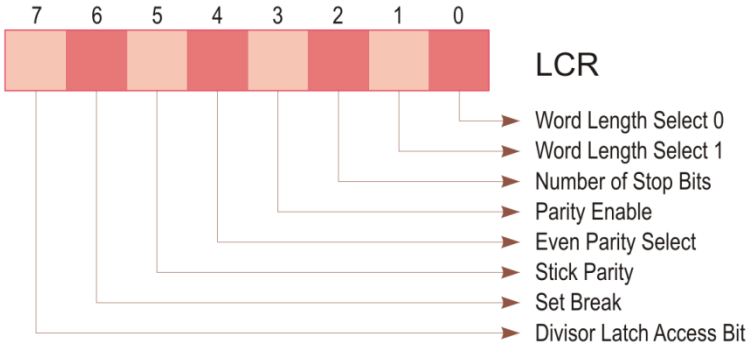

Figure 1.11. Line Control Register (LCR).

The meaning of LCR register bits is the following:

- Bit 7 (Divisor Latch Access Bit) changes the function of registers accessible through addresses 0x3F8 (0x2F8) and 0x3F9 (0x2F9). If this bit is 0, the accessible registers are *Transmitter / Receiver Buffer Register* and *Interrupt Enable Register*, respectively. If bit 7 is 1, the accessible registers are the divisor latch registers (LSB and MSB, respectively).
- Bit 6 (Set Break). If set to 1, the chip forces the communication line to the logical 0 level (space). This corresponds to the "*break*" state of the line, which allows to draw a remote terminal's attention by an interrupt generated when this state of the line is detected. The line may be brought to the normal state by setting bit 6 to 0.
- Bit 5 (Stick Parity) allows to send or wait for parity bits with a fixed value, 0 or 1:

  0: The parity is checked normally, according to the Parity Enable and Even Parity Select bits;
  1: If the Parity Enable bit is 1, fixed-value bits are sent or checked in place of the parity bit, according to the Even Parity Select bit. If the Even Parity Select bit is 0, the parity bit is always 1, and if the Even Parity Select bit is 1, the parity bit is always 0.
- Bit 4 (Even Parity Select) indicates the type of parity used, if parity generation and parity checking is enabled by the Parity Enable bit:

  0: Odd parity;
  1: Even parity.
- Bit 3 (Parity Enable) enables or disables parity generation and parity checking:

  0: Parity generation and parity checking is disabled;
  1: Parity generation and parity checking is enabled.
- Bit 2 (Number of Stop Bits) indicates the number of stop bits generated or expected by the UART chip:

  0: 1 stop bit;
  1: 2 stop bits (1.5 bits if character length is 5 bits).

The receiver only checks the first stop bit, regardless of the number of stop bits selected.

- Bits 1..0 (Word Length Select) specify the length of characters sent or received:

  00: 5 bits/character;
  01: 6 bits/character;
  10: 7 bits/character;
  11: 8 bits/character.

MCR - Modem Control Register

The MCR register (Figure 1.12) is used to control the communication with the modem.

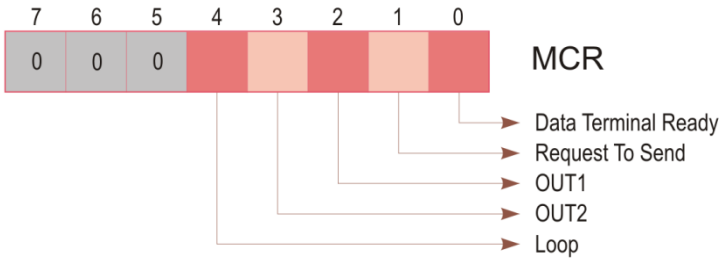

Figure 1.12. Modem Control Register (MCR).

The meaning of MCR register bits is the following:

- Bits 7..5 are not used (set to 0).
- Bit 4 (Loop) allows to test the UART chip and the communication programs. By setting this bit to 1, the following operations will be performed:
  1. The transmitter serial output is placed into the logical 1 state.
  2. The receiver serial input is disconnected.
  3. The data on the TSR register output will be received in the receive buffer RBR.
  4. The input lines for modem control *DSR*, *CTS*, *RI*, and *DCD* are disconnected, and they can be controlled with bits 0..3 of the MCR register (Data Terminal Ready, Request To Send, OUT1, and OUT2, respectively). If the interrupts are enabled, changes of these bits will generate interrupt requests as if the signals had been activated by the modem.
- Bits 2 and 3 (OUT1 and OUT2) can be used to implement a user-defined communication.
- Bit 1 (Request To Send) asserts by value 1 the *RTS* signal of the interface.
- Bit 0 (Data Terminal Ready) asserts by value 1 the *DTR* signal of the interface.

Note

- The *DTR*, *RTS*, *OUT1*, and *OUT2* signals are active in the logical 0 state.

LSR - Line Status Register

This register indicates the communication line status. Bits 6..5 refer to the transmitter, and bits 4..0 refer to the receiver (Figure 1.13).

The meaning of LSR register bits is the following:

- Bit 7 (Error in Receiver FIFO) is set to 1 when the FIFO memories are enabled and at least a parity error, a framing error, or a "*break*" condition occurred while receiving the characters present in the receive FIFO memory.

- Bit 6 (Transmitter Shift Register Empty) is set to 1 if both the THR and TSR registers are empty. If the FIFO memories are enabled, this bit is set to 1 when both the transmit FIFO memory and the TSR register are empty.

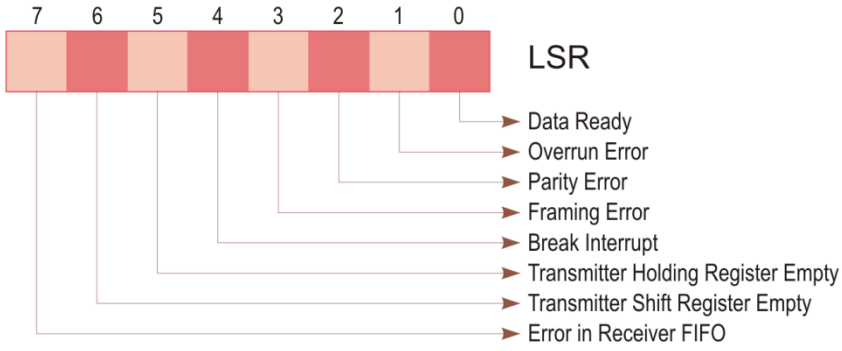

Figure 1.13. Line Status Register (LSR).

- Bit 5 (Transmitter Holding Register Empty) is set to 1 when the contents of the THR register is transferred into the TSR register and transmission of the character begins. This indicates that the UART chip is ready to accept a new character for transmission. When the THR register is empty, the UART chip generates an interrupt request if this type of interrupt is enabled. This bit is reset to 0 when a new character is written into the THR register. If the FIFO memories are enabled, this bit is set to 1 when the transmit FIFO memory is empty and it is reset to 0 when at least one character is written into the transmit FIFO memory.
- Bit 4 (Break Interrupt) is set to 1 if spaces (logical 0) are detected on the line for a longer period than that needed to send one character. In this case, a byte with value 0 is written into the receive buffer and an interrupt request is generated. This bit is reset to 0 by reading the LSR register.
- Bit 3 (Framing Error) is set to 1 if a character is received without the corresponding stop bits. The receiver only checks the first stop bit, regardless of the number of stop bits programmed. When detects this error, the chip tries to resynchronize. This bit is reset to 0 by reading the LSR register.
- Bit 2 (Parity Error) is set to 1 if a character with a parity different than that expected is received. This bit is reset to 0 by reading the LSR register.
- Bit 1 (Overrun Error) is set to 1 if a new character is received before the program reads the character from the RBR register. In this case, one or more characters will be lost. The overrun error, like other errors, generates an interrupt request. This bit is reset to 0 by reading the LSR register. If the FIFO memories are enabled and the receive FIFO memory is filled above the threshold level, an overrun error will be signaled only after the FIFO memory is full and the next character has been received in the RSR register.
- Bit 0 (Data Ready) is set to 1 when a character has been received and it has been transferred into the RBR register or into the FIFO memory. This bit is reset to 0 by reading the character from the RBR register or after reading all the characters from the receive FIFO memory. By receiving a character, an interrupt request will be generated if this type of interrupt is enabled.

MSR - Modem Status Register

This register contains information about the modem status (Figure 1.14). The meaning of MSR register bits is the following:

- Bits 7..4 indicates the current status of the *CD*, *RI*, *DSR*, and *CTS* signals, respectively. An asserted signal is indicated by the corresponding bit in the MSR register set to 1. If the Loop bit of the MCR register is 1, the state of bits 7, 6, 5, 4 in the MSR register is equivalent to the state of bits OUT2, OUT1, Data Terminal Ready, and Request To Send, respectively, in the MCR register.
- Bits 3..0 indicates the change of signals *CD*, *RI*, *DSR*, and *CTS*, respectively, from the last read of the MSR register. These bits are reset when the MSR register is read.

Note

- The *CTS*, *DSR*, *RI*, and *CD* signals are active in the logical 0 state.

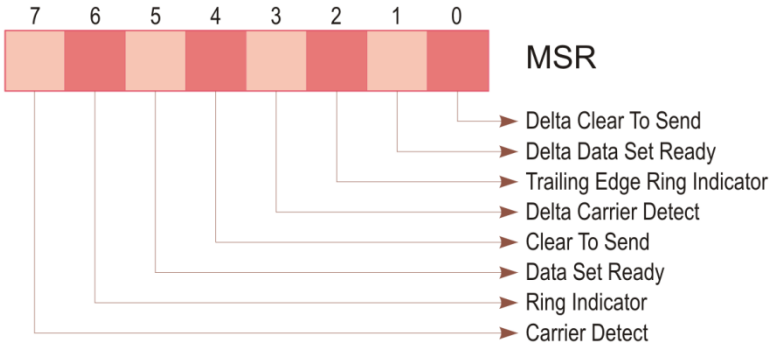

Figure 1.14. Modem Status Register (MSR).

Scratch Register

This register is not used for communication; it may be used for temporary storage of a byte.

1.12. Port Access in Windows Operating Systems

1.12.1. The I/O Port Access Problem

Windows operating systems, starting with the NT version, will generate a privileged instruction exception if an attempt is made to access an I/O port from a program running in user mode. This is due to the restrictions imposed by the processor running in protected mode.

In protected mode, accessing I/O ports is controlled by the I/O privilege level (IOPL) in the flags register (EFLAGS) and the I/O permission bitmap of a *Task State Segment* (TSS).

The processor recognizes four privilege levels for the programs, from 0 to 3, where 0 is the highest privilege level. The two IOPL bits indicate the privilege level that a process should have to be able to execute the privileged instructions; such instructions are the I/O instructions IN and OUT. For instance, if both IOPL bits are 1, the privileged instructions can only be executed by processes with a privilege level of 0 or 1.

Under *Windows* operating systems, only two privilege levels are used out of the processor's four privilege levels. Programs in user mode will run with privilege level 3, while the operating system's kernel and device drivers will run with privilege level 0. Consequently, privileged instructions can be executed by the user programs only through the operating system or device drivers.

The I/O permission bitmap contains one bit for each I/O address. If the bit corresponding to an I/O address is set, an I/O instruction with that address will generate an exception, otherwise the I/O operation will be allowed. This bitmap can be used to allow certain user programs to access certain I/O ports. The processor checks the I/O permission bitmap when an I/O instruction is executed in a process and the process is not privileged enough to execute the instruction.

In addition to restrictions imposed by the operating systems on port accesses, another problem is that most of today's programming environments for high-level languages do not pro-

vide functions for direct I/O port access, functions that existed in previous versions of these programming environments.

There are two solutions to the problem of accessing I/O ports under *Windows* operating systems. The first solution is to use a device driver that runs with privilege level 0 and which performs port accesses. Data are transferred between a user program and the device driver via calls to the `DeviceIoControl` system function; the operation to be performed by the driver is specified by an IOCTL (*I/O Control*) code. To simplify user programs, the driver may provide some I/O functions similar to the `inportb()` and `outportb()` functions provided by earlier versions of programming environments for the C language. For instance, these functions might be provided as dynamic link libraries (DLLs). Consequently, there is no need to directly call the `DeviceIoControl` system function.

This solution is recommended for accessing the I/O ports. However, the disadvantage of using such a device driver is that the efficiency of data transfers will be quite low. At each call of a function to read or write a byte or word, the processor must switch from privilege level 3 to level 0, and after performing the operation must switch back. Nevertheless, the driver may also provide functions for reading and writing a data block, instead of reading and writing a single byte.

The second solution to the problem of accessing I/O ports is to modify the I/O permission bitmap to allow a particular process to access certain I/O ports. Although this method is not recommended, it allows running existing applications under the *Windows* operating systems.

1.12.2. The Marvin HW Driver

There are several drivers available for accessing I/O ports under the *Windows* operating systems. For the laboratory applications, the *Marvin HW* driver developed by Marvin Test Solutions Inc. will be used. The current version (in 2021) of the driver is 4.9.8.

The *Marvin HW* driver can be used on 32-bit and 64-bit *Windows* platforms. It includes 32-bit and 64-bit DLL files that use IOCTL calls to access the kernel-mode drivers. For interfacing with the DLL files, a C/C++ header file (Hw.h) is available. The header file defines functions for reading a byte, a 16-bit word, or a 32-bit double-word from an input port, and for writing a byte, a word, or a double-word to an output port. Besides these functions, the header file also defines functions for reading and writing a byte, a word, or a double-word from/to the memory. The memory functions are useful for accessing the physical memory. For instance, they can be used for accessing the configuration space of a PCI device or the memory-mapped registers of an I/O controller.

In the Hw.h header file, the functions for reading a byte, a word, and a double-word from an input port are defined as follows:

```
INT    __inp(WORD wPort);
WORD   __inpw(WORD wPort);
DWORD  __inpd(WORD wPort);
```

These functions return the value read from the specified port. The single parameter of these functions (`wPort`) represents the port address, which is always a 16-bit value. The `__inp()` function reads a byte from the port, the `__inpw()` function reads a word from the port, and the `__inpd()` function reads a double-word from the port.

Notes

- Although the `__inp()` function returns an `INT` value, the function is used for reading a single byte from an 8-bit port.
- The names of these functions start with two underline characters. These functions should not be confused with the `_inp()`, `_inpw()`, and `_inpd()` functions, whose names start with a single underline character and which are available when the conio.h header file is included in the source code. The functions defined in the conio.h

header file cannot be called in user-mode programs, because each call would generate a privileged instruction exception.

In the Hw.h header file, the functions for writing a byte, a word, and a double-word to an output port are defined as follows:

```
INT    __outp(WORD wPort, INT iData);
WORD   __outpw(WORD wPort, WORD wData);
DWORD  __outpd(WORD wPort, DWORD dwData);
```

The first parameter of these functions (`wPort`) represents the port address, which is always a 16-bit value. The second parameter represents the data to be written to the specified port. The `__outp()` function writes a byte to the port, the `__outpw()` function writes a word to the port, and the `__outpd()` function writes a double-word to the port.

Notes

- Although the second parameter of the `__outp()` function is specified with the type `INT`, this function is used for writing a single byte to an 8-bit port.
- Each of these functions returns the data written to the port. However, the return value is usually ignored.
- The names of these functions start with two underline characters. These functions should not be confused with the `_outp()`, `_outpw()`, and `_outpd()` functions, which have restrictions similar to those of the `_inp()`, `_inpw()`, and `_inpd()` functions.

The following functions are available for reading a byte, a word, or a double-word from a memory location:

```
BYTE   _inm(DWORD_PTR dwAddress);
WORD   _inmw(DWORD_PTR dwAddress);
DWORD  _inmdw(DWORD_PTR dwAddress);
```

These functions return the value read from the memory location with the specified address (`dwAddress`). The parameter of these functions is defined with the type `DWORD_PTR`, which is a 64-bit unsigned integer.

For writing a byte, a word, or a double-word to a memory location the following functions are available:

```
BOOL   _outm(DWORD_PTR dwAddress, INT iData);
BOOL   _outmw(DWORD_PTR dwAddress, WORD wData);
BOOL   _outmdw(DWORD_PTR dwAddress, DWORD dwData);
```

The first parameter of these functions (`dwAddress`) represents the memory address, which is defined as a 64-bit unsigned integer. The second parameter represents the data to be written to the specified address. These functions return `TRUE` if the execution was successful.

Note

- Although the second parameter of the `__outm()` function is specified with the type `INT`, this function is used for writing a single byte to a memory location.

Assuming that the 64-bit *Marvin HW* driver is installed, for using the functions provided by this driver in a 64-bit application the following operations are required:

1. After creating the application project, copy the Hw.h and Hw64.lib files into the folder of the application project. These files are available in the installation folder of the *Marvin HW* driver (C:\Program Files (x86)\Marvin Test Solutions\HW).
2. Change the active solution platform to x64. For this, select *Build* → *Configuration Manager*... in the main menu. In the *Configuration Manager* dialog window, click on

the *x86* line under the *Active Solution Platform* field, select x64, and then select the *Close* button.

3. Insert the following line into the source file in which the *Marvin HW* driver functions will be used:

```
#include "Hw.h"
```

4. Before using the driver functions, call the `HwOpen()` function. If this function returns `FALSE`, display an error message indicating that the driver is not installed correctly. At the end of the application, call the `HwClose()` function:

```
if (!HwOpen()) { ... };
...
HwClose();
```

5. Before building the application, specify the Hw64.lib file as an additional dependency for the linker. For this, right-click on the project name in the *Solution Explorer* tab and select the *Properties* option. In the *Property Pages* dialog window, expand the *Configuration Properties* option, expand the *Linker* option, and select the *Input* line. On the right tab, select the *Additional Dependencies* line, click on the arrow at the end of the line, and select *<Edit…>*. In the *Additional Dependencies* dialog box, enter **Hw64.lib** and select the *OK* button. Close the *Property Pages* dialog window.

1.13. Applications

1.13.1. Answer the following questions:

a. What is the difference between the unit of measure for modulation speed and the unit of measure for communication speed?

b. How can the synchronization between the receiver clock and the transmitter clock be ensured in the case of a synchronous communication?

c. How can the correctness of data blocks transmitted be ensured in the case of a synchronous communication?

d. What is the function of each of the following signals of the serial interface: *DTR*, *DSR*, *RTS*, and *CTS*?

e. What is the reason for which I/O ports cannot be accessed directly from user programs under the *Windows* operating systems?

1.13.2. Consider an 8-bit read/write port with the address defined by the `PORT` constant and an 8-bit mask defined by the `BIT4` constant as `(1 << 4)`. Use the `__inp()` and `__outp()` functions of the *Marvin HW* driver to write the sequences of instructions in the C language that perform the following operations:

- Wait until the bit of the port defined by the `BIT4` mask becomes set;
- Set the bit of the port defined by the `BIT4` mask;
- Clear the bit of the port defined by the `BIT4` mask;
- Complement (toggle) the bit of the port defined by the `BIT4` mask.

Other bits of the port should not be altered by the sequences of instructions.

1.13.3. Build and test the TestCom1DT-e application, whose source files are available on the laboratory page in the TestCom1DT-e.zip archive. This application checks the existence of the serial port with the base address 0x3F8. The check is performed by writing the values 0xAA and 0x55 into the port's LCR register and then reading them back. If the values read match the values written, the application assumes that the port exists. Use the *Visual Studio 2019* programming environment to build this application, by performing the following operations:

1. Launch the *Visual Studio 2019* programming environment.

2. Select *File* → *New* → *Project…*. In the *Create a new project* dialog window, select the *C++* language, select *Windows Desktop Wizard*, and then select the *Next* button. Enter the name of the project into the *Project name* field. In the *Location* field, select the path to the folder in which the project will be created. Check the *Place solution and project in the same directory* option to avoid creating another folder for the solution. Select the *Create* button.
3. In the *Windows Desktop Project* dialog box, select the *Desktop Application (.exe)* option for the application type, check the *Empty project* option, and then select the *OK* button.
4. Change the active solution platform by selecting *Build* → *Configuration Manager…* in the main menu. In the *Configuration Manager* dialog window, click on the *x86* line under the *Active Solution Platform* field, select *x64*, and then select the *Close* button.
5. In the *Solution Explorer* tab, right-click on the project name and select the *Properties* option. In the *TestCom1DT-e Property Pages* dialog window, expand the *Configuration Properties* option, expand the *Advanced* option, select the *Character Set* line in the right tab, and choose the *Not Set* option. Select the *OK* button.
6. Copy to the project folder the files contained in the TestCom1DT-e.zip archive.
7. In the main menu, select *Project* → *Add Existing Item…*. Select the files copied to the project folder, and then select the *Add* button.
8. Copy to the project folder the Hw.h and Hw64.lib files. These files are available in the installation folder of the driver (C:\Program Files (x86)\Marvin Test Solutions\HW).
9. Add to the project the Hw.h file.
10. Specify the Hw64.lib file as an additional dependency for the linker, as described in Step 5 of the sequence of operations presented in Section 1.12.2.
11. Select *Build* → *Build Solution*, make sure that the application builds without errors, and then run the application by selecting *Debug* → *Start Without Debugging*.

Note

- The TestCom1DT-e application uses the `DrawText()` function for displaying the contents of the application window. The TestCom1TO-e application, also available on the laboratory page, performs the same operations as the TestCom1DT-e application, but uses the `TextOut()` function for displaying the contents of the application window.

1.13.4. Create a Windows application for initializing the COM1 serial port. As template for the Windows application, use the AppScroll-e application, whose source files are available on the laboratory web page in the AppScroll-e.zip archive. Perform the following operations to create the application project:

1. In the *Visual Studio 2019* programming environment, create a new empty *Windows Desktop* project with the *Windows Desktop Wizard*. Check the *Place solution and project in the same directory* option to avoid creating another folder for the solution.
2. Change the active solution platform to x64.
3. Change the *Character Set* project property to *Not Set*, as described in Step 5 of the sequence of operations presented in Section 1.13.3.
4. Copy to the project folder the files contained in the AppScroll-e.zip archive and add these files to the project.
5. Copy to the project folder the Hw.h, Hw64.lib, and ComDef-e.h files from the folder of the project created for Application 1.13.3.

6. Add to the project the Hw.h and ComDef-e.h files.
7. Specify the Hw64.lib file as an additional dependency for the linker.
8. Open the AppScroll-e.cpp source file and add a `#include` directive to include the ComDef-e.h header file.
9. Select *Build* → *Build Solution* and make sure that the application builds correctly.

In the AppScroll-e.cpp source file, write a function to initialize the COM1 serial port with the following parameters: binary rate of 115,200 bits/s; character length of 8 bits; no parity bit; 1 stop bit. Use the ComDef-e.h definition file for the serial port. Perform the following steps to initialize the serial port:

1. Set bit 7 of the LCR register to 1; the mask of this bit is `LCR_DLAB`. Setting this bit is needed to access the divisor registers for specifying the value by which the serial controller's clock frequency should be divided to get the desired binary rate.
2. Write the least significant byte of the divisor to the DLR_LSB register and the most significant byte of the divisor to the DLR_MSB register. The divisors are defined in Table 1.6.
3. Using the description of the LCR register (Section 1.11), write to the LCR register a byte conforming to the required communication parameters. Bit 7 of this byte should be 0 in order to access the registers normally after accessing the divisor registers.
4. Set the following bits of the MCR register: Data Terminal Ready; Request To Send; OUT2.

In the `AppScroll()` function, after initializing the *Marvin HW* driver with the `HwOpen()` function, call the `CreateFile()` function to open the COM1 port (similarly to the call in the TestCom1DT-e application), call the function for initializing the COM1 port, and then call the `CloseHandle()` function to close the COM1 port.

1.13.5. Extend Application 1.13.4 by writing a function that sends a single character through the COM1 serial port. In the `AppScroll()` function, after initializing the COM1 port, use this function to send a command for turning on/off the LEDs or changing the color of the RGB LEDs on a development board connected to the computer through the serial port. The commands consist of character strings that are interpreted by an application running on a microprocessor implemented on the development board. Example commands are presented in the Board-Commands.txt file. For testing the application on the virtual machine, launch the Oracle VM VirtualBox, select *Settings* → *Serial Ports* → *Port 1*, select the *Raw File* option for *Port Mode*, and in the *Path/Address* field enter the full path and filename on the host system to which the characters sent through the serial port of the virtual machine will be written.

1.13.6. Create a new application for sending commands to the same development board of Application 1.13.5 for reading the status of switches or buttons and receiving the results returned by the board. The commands are presented in the Board-Commands.txt file. First, create an empty *Windows Desktop* project with the *Windows Desktop Wizard*. Then, copy to the project folder the source, header, and resource files from the folder of the project created for Application 1.13.4, and perform the same operations that are specified for creating that application. In the AppScroll-e.cpp source file, after initializing the COM1 port, use the `WriteFile()` function for sending to the development board the same command specified in Application 1.13.5 for turning on/off the LEDs or changing the color of the RGB LEDs. For details on the function parameters, access the Windows Developer documentation by placing the cursor inside the function name and pressing the F1 key. Test the application on the virtual machine. Next, use the `WriteFile()` function for sending to the development board the command for reading the status of switches. Finally, receive the results returned by the development board using the `ReadFile()` function, and display the characters received.

1.13.7. Extend Application 1.13.4 to send a character string through the COM1 serial port. First, write a function that sends a single character through the COM1 serial port. In the `AppScroll()` function, call the function written for Application 1.13.4 for initializing the COM1 serial port, and then, in a loop, call the function for sending a single character to send a string of characters. Connect a crossover serial cable between the COM1 ports of two computers. Launch the *HyperTerminal PE* or the *Tera Term* application on the computer used as receiver and create a connection with the same parameters as those of the transmitter serial port. Then, launch the transmitter application and verify its operation watching the characters displayed in the *HyperTerminal PE* or *Tera Term* window.

1.13.8. Extend Application 1.13.7 to receive a character string through the COM1 serial port. First, write a function that receives a single character through the COM1 serial port. In the `AppScroll()` function, call the function written for Application 1.13.4 for initializing the COM1 serial port, and then, in a loop, call the function for receiving a single character to receive a string of characters. The application will display each character received and will complete when the ESC (0x1B) character is received. Connect a crossover serial cable between the COM1 ports of two computers. Launch the *HyperTerminal PE* or the *Tera Term* application on the computer used as transmitter and create a connection with the same parameters as those of the receiver serial port. Then, launch the receiver application and verify its operation entering text lines in the *HyperTerminal PE* or *Tera Term* window.

1.13.9. Modify Application 1.13.8 so that it will send back (in echo) each character received on the COM1 serial port. Verify the application similarly to the procedure described for Application 1.13.8.

1.13.10. Connect two computers through a crossover serial cable. Use the transmitter application 1.13.7 and the receiver application 1.13.8 to send a character string from one of the computers to the other. The receiver computer should display the character string received.

Bibliography

[1] Baruch, Z., *Sisteme de intrare/ieşire, Îndrumător de lucrări de laborator*, U.T.PRES, Cluj-Napoca, 1998.

[2] National Semiconductor Corp., “PC16550D Universal Asynchronous Receiver/ Transmitter with FIFOs”, 1995, www.national.com/ds.cgi/PC/PC16550D.pdf.

[3] Peacock, C., “Interfacing the Serial / RS232 Port”, Beyond Logic, 2010, http://beyondlogic.org/serial/serial.htm.

[4] Rosch, W. L., *Hardware Bible*, Sixth Edition, Que Publishing, 2003.

[5] Strangio, C. E., “The RS232 Standard”, CAMI Research Inc., Lexington, Massachusetts, 1993-2015, http://www.camiresearch.com/Data_Com_Basics/RS232_standard.html.

[6] Wikimedia Foundation, Inc., “RS-232”, Wikipedia, The Free Encyclopedia, 2015, http://en.wikipedia.org/wiki/RS-232.