CHAPTER 11

AVR SERIAL PORT PROGRAMMING IN ASSEMBLY AND C

OBJECTIVES

Upon completion of this chapter, you will be able to:

- >> **Contrast and compare serial versus parallel data transfer**
- >> **List the advantages of serial communication over parallel**
- >> **Explain serial communication protocol**
- >> **Contrast synchronous versus asynchronous communication**
- >> **Contrast half- versus full-duplex transmission**
- >> **Explain the process of data framing**
- >> **Describe data transfer rate and bps rate**
- >> **Define the RS232 standard**
- >> **Explain the use of the MAX232 and MAX233 chips**
- >> **Interface the AVR with an RS232 connector**
- >> **Discuss the baud rate of the AVR**
- >> **Describe serial communication features of the AVR**
- >> **Describe the main registers used by serial communication of the AVR**
- >> **Program the ATmega328 serial port in Assembly and C**

Computers transfer data in two ways: parallel and serial. In parallel data transfers, often eight or more lines (wire conductors) are used to transfer data to a device that is only a few feet away. Although a lot of data can be transferred in a short amount of time by using many wires in parallel, the distance cannot be great. To transfer to a device located many meters away, the serial method is used. In serial communication, the data is sent one bit at a time, in contrast to parallel communication, in which the data is sent a byte or more at a time. As technology advances, the data rate of serial communication may exceed parallel communication while parallel communication still retains the disadvantages of the size and cost of cable and connector, and the crosstalk between the data lines at longer distance.

Serial communication of the AVR is the topic of this chapter. The AVR has serial communication capability built into it, thereby making possible fast data transfer using only a few wires.

In this chapter we first discuss the basics of serial communication. In Section 11.2, AVR interfacing to RS232 connectors via MAX232 line drivers is discussed. Serial port programming of the AVR is discussed in Section 11.3. Section 11.4 covers AVR C programming for the serial port using the Win AVR compiler. In Section 11.5 interrupt-based serial port programming is discussed.

SECTION 11.1: BASICS OF SERIAL COMMUNICATION

When a microprocessor communicates with the outside world it usually provides the data in byte-sized chunks. For parallel transfer, 8-bit data is transferred at the same time. For serial transfer, 8-bit data is transferred one bit at a time. Figure 11-1 diagrams serial versus parallel data transfers.

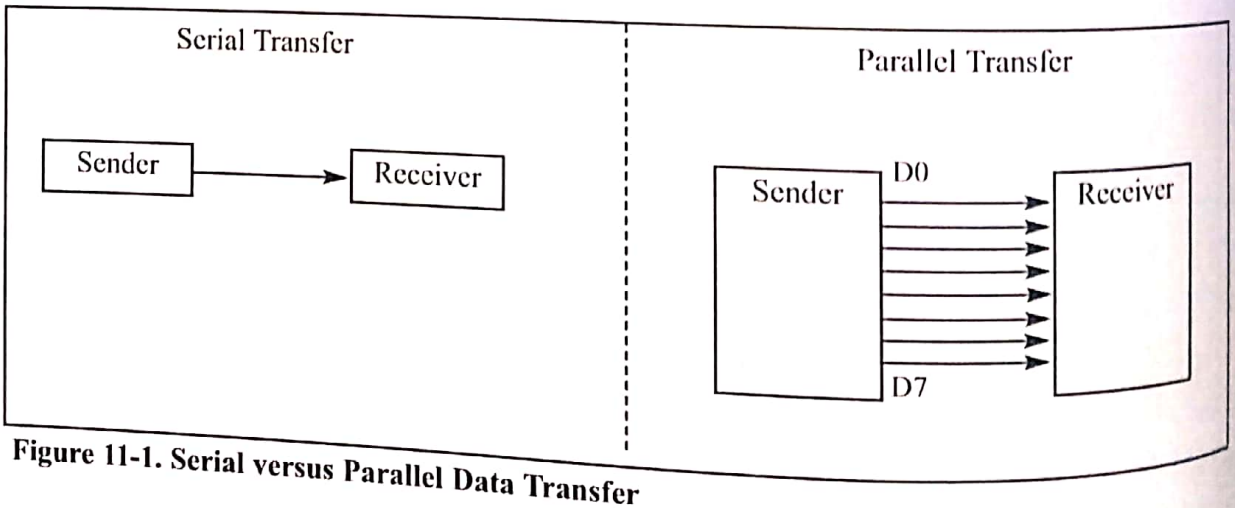

Figure 11-1. Serial versus Parallel Data Transfer

The fact that a single data line is used in serial communication instead of the 8-bit data line of parallel communication makes serial transfer not only much cheaper but also enables two computers located in two different cities to communicate over the telephone.

For serial data communication to work, the byte of data must be converted to serial bits using a parallel-in-serial-out shift register; then it can be transmitted over a single data line. This also means that at the receiving end there must be a serial-in-parallel-out shift register to receive the serial data and pack them into a

byte. Of course, if data is to be transmitted on the telephone line, it must be converted from 0s and 1s to audio tones, which are sinusoidal signals. This conversion is performed by a peripheral device called a *modem*, which stands for "modulator/demodulator."

When the distance is short, the digital signal can be transmitted as it is on a simple wire and requires no modulation. This is how x86 PC keyboards transfer data to the motherboard. For long-distance data transfers using communication lines such as a telephone, however, serial data communication requires a modem to *modulate* (convert from 0s and 1s to audio tones) and *demodulate* (convert from audio tones to 0s and 1s).

Serial data communication uses two methods, asynchronous and synchronous. The *synchronous* method transfers a block of data (characters) at a time, whereas the *asynchronous* method transfers a single byte at a time. It is possible to write software to use either of these methods, but the programs can be tedious and long. For this reason, special IC chips are made by many manufacturers for serial data communications. These chips are commonly referred to as UART (universal asynchronous receiver-transmitter) and USART (universal synchronous-asynchronous receiver-transmitter). The AVR chip has a built-in USART, which is discussed in detail in Section 11.3.

Half- and full-duplex transmission

In data transmission, if the data can be both transmitted and received, it is a *duplex* transmission. This is in contrast to *simplex* transmissions such as with printers, in which the computer only sends data. Duplex transmissions can be half or full duplex, depending on whether or not the data transfer can be simultaneous. If data is transmitted one way at a time, it is referred to as *half duplex*. If the data can go both ways at the same time, it is *full duplex*. Of course, full duplex requires two wire conductors for the data lines (in addition to the signal ground), one for transmission and one for reception, in order to transfer and receive data simultaneously. See Figure 11-2.

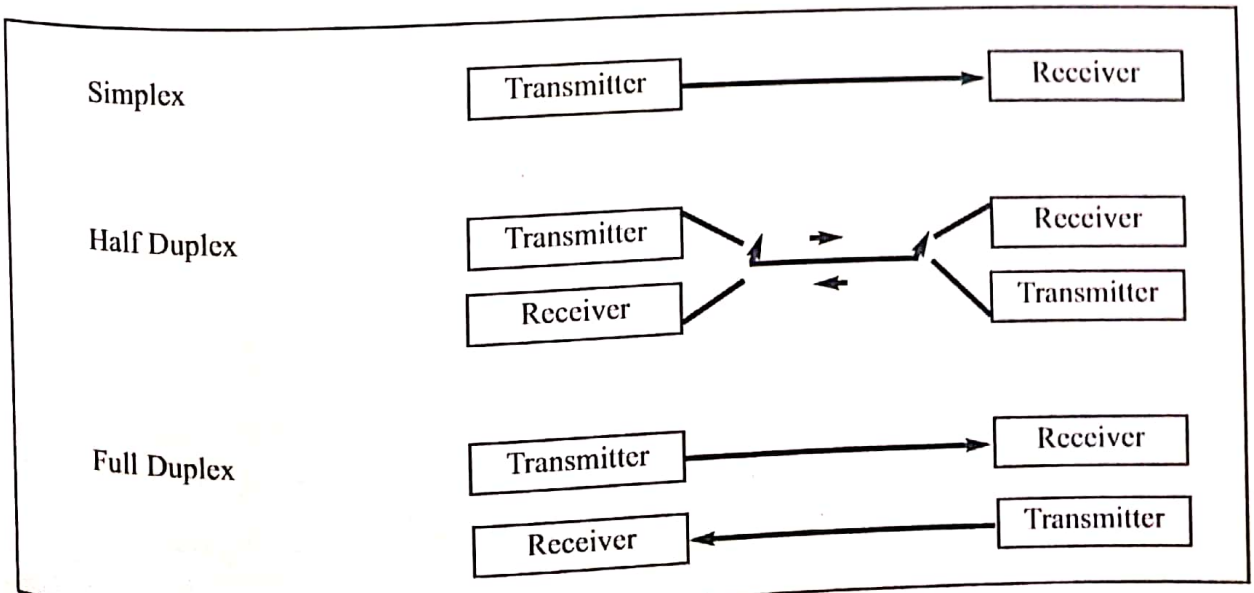

Figure 11-2. Simplex, Half-, and Full-Duplex Transfers

Asynchronous serial communication and data framing

The data coming in at the receiving end of the data line in a serial data transfer is all 0s and 1s; it is difficult to make sense of the data unless the sender and receiver agree on a set of rules, a *protocol*, on how the data is packed, how many bits constitute a character, and when the data begins and ends.

Start and stop bits

Asynchronous serial data communication is widely used for character-oriented transmissions, while block-oriented data transfers use the synchronous method. In the asynchronous method, each character is placed between start and stop bits. This is called *framing*. In data framing for asynchronous communications, the data, such as ASCII characters, are packed between a start bit and a stop bit. The start bit is always one bit, but the stop bit can be one or two bits. The start bit is always a 0 (low), and the stop bit(s) is 1 (high). For example, look at Figure 11-3 in which the ASCII character "A" (8-bit binary 0100 0001) is framed between the start bit and a single stop bit. Notice that the LSB is sent out first.

Notice in Figure 11-3 that when there is no transfer, the signal is 1 (high), which is referred to as *mark*. The 0 (low) is referred to as *space*. Notice that the transmission begins with a start bit (space) followed by D0, the LSB, then the rest of the bits until the MSB (D7), and finally, the one stop bit indicating the end of the character "A".

In asynchronous serial communications, peripheral chips and modems can be programmed for data that is 7 or 8 bits wide. This is in addition to the number of stop bits, 1 or 2. While in older systems ASCII characters were 7-bit, in recent years, 8-bit data has become common due to the extended ASCII characters. In some older systems, due to the slowness of the receiving mechanical device, two stop bits were used to give the device sufficient time to organize itself before transmission of the next byte. In modern PCs, however, the use of one stop bit is standard. Assuming that we are transferring a text file of ASCII characters using 1 stop bit, we have a total of 10 bits for each character: 8 bits for the ASCII code, and 1 bit each for the start and stop bits. Therefore, each 8-bit character has an extra 2 bits, which gives 25% overhead.

In some systems, the parity bit of the character byte is included in the data frame in order to maintain data integrity. This means that for each character (7- or 8-bit, depending on the system) we have a single parity bit in addition to start and stop bits. The parity bit is odd or even. In the case of an odd parity bit the number of 1s in the data bits, including the parity bit, is odd. Similarly, in an even parity

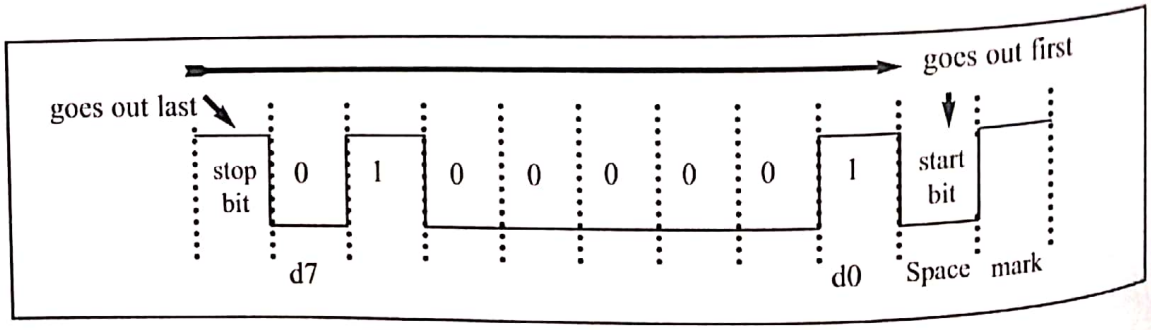

Figure 11-3. Framing ASCII 'A' (41H)

bit system the total number of bits, including the parity bit, is even. For example, the ASCII character "A", binary 0100 0001, has 0 for the even parity bit. UART chips allow programming of the parity bit for odd-, even-, and no-parity options.

Data transfer rate

The rate of data transfer in serial data communication is stated in *bps* (bits per second). Another widely used terminology for bps is *baud rate*. However, the baud and bps rates are not necessarily equal. This is because baud rate is the modem terminology and is defined as the number of signal changes per second. In modems, sometimes a single change of signal transfers several bits of data. As far as the conductor wire is concerned, the baud rate and bps are the same, and for this reason in this book we use the terms bps and baud interchangeably.

The data transfer rate of a given computer system depends on communication ports incorporated into that system. For example, the early IBM PC/XT could transfer data at the rate of 100 to 9600 bps. In recent years, however, Pentium-based PCs transfer data at rates as high as 56K. Notice that in asynchronous serial data communication, the baud rate is generally limited to 100,000 bps.

RS232 standards

To allow compatibility among data communication equipment made by various manufacturers, an interfacing standard called RS232 was set by the Electronics Industries Association (EIA) in 1960. In 1963 it was modified and called RS232A. RS232B and RS232C were issued in 1965 and 1969, respectively. In this book we refer to it simply as RS232. Today, RS232 is one of the most widely used serial I/O interfacing standards. This standard is used in PCs and numerous types of equipment. Because the standard was set long before the advent of the TTL logic family, however, its input and output voltage levels are not TTL compatible. In RS232, a 1 is represented by –3 to –25 V, while a 0 bit is +3 to +25 volts, making –3 to +3 undefined. For this reason, to connect any RS232 to a microcontroller system we must use voltage converters such as MAX232 to convert the TTL logic levels to the RS232 voltage levels, and vice versa. MAX232 IC chips are commonly referred to as line drivers. Original RS232 connection to MAX232 is discussed in Section 11.2.

Table 11-1: IBM PC DB-9 Signals

Pin	Description
1	Data carrier detect (DCD)
2	Received data (RxD)
3	Transmitted data (TxD)
4	Data terminal ready (DTR)
5	Signal ground (GND)
6	Data set ready (DSR)
7	Request to send (RTS)
8	Clear to send (CTS)
9	Ring indicator (RI)

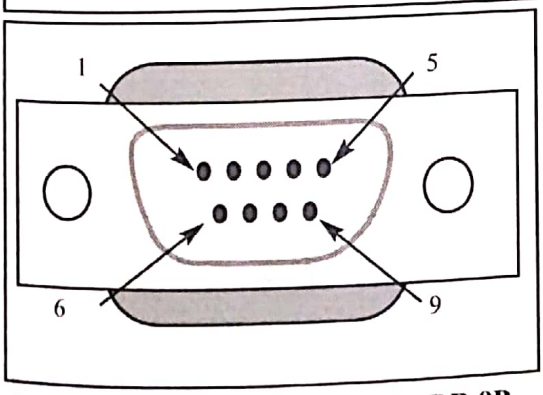

Figure 11-4. 9-Pin Connector for DB-9P

RS232 pins

Table 11-1 shows the pins for the RS232 cable and their labels, commonly referred to as the DB-9 connector. In labeling, DB-9P refers to the plug connector (male), and DB-9S is for the socket connector (female). See Figure 11-4.

Data communication classification

Current terminology classifies data communication equipment as DTE (data terminal equipment) or DCE (data communication equipment). DTE refers to terminals and computers that send and receive data, while DCE refers to communication equipment, such as modems, that are responsible for transferring the data. Notice that all the RS232 pin function definitions of Tables 11-1 and 11-2 are from the DTE point of view.

The simplest connection between a PC and a microcontroller requires a minimum of three pins, TX, RX, and ground, as shown in Figure 11-5. Notice in that figure that the RX and TX pins are interchanged.

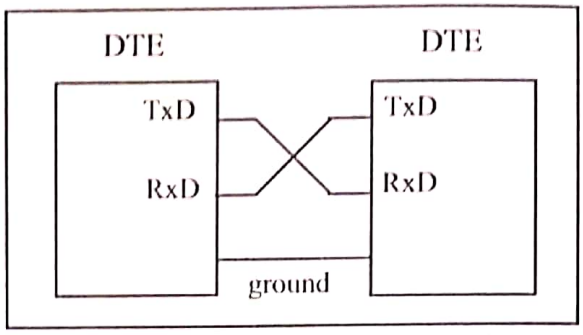

Figure 11-5. Null Modem Connection

Examining RS232 handshaking signals

To ensure fast and reliable data transmission between two devices, the data transfer must be coordinated. Just as in the case of the printer, because the receiving device may have no room for the data in serial data communication, there must be a way to inform the sender to stop sending data. Many of the pins of the RS-232 connector are used for handshaking signals. Their description is provided below only as a reference, and they can be bypassed because they are not supported by the AVR UART chip.

1. DTR (data terminal ready). When the terminal (or a PC COM port) is turned on, after going through a self-test, it sends out signal DTR to indicate that it is ready for communication. If there is something wrong with the COM port, this signal will not be activated. This is an active-LOW signal and can be used to inform the modem that the computer is alive and kicking. This is an output pin from DTE (PC COM port) and an input to the modem.
2. DSR (data set ready). When the DCE (modem) is turned on and has gone through the self-test, it asserts DSR to indicate that it is ready to communicate. Thus, it is an output from the modem (DCE) and an input to the PC (DTE). This is an active-LOW signal. If for any reason the modem cannot make a connection to the telephone, this signal remains inactive, indicating to the PC (or terminal) that it cannot accept or send data.
3. RTS (request to send). When the DTE device (such as a PC) has a byte to transmit, it asserts RTS to signal the modem that it has a byte of data to transmit. RTS is an active-LOW output from the DTE and an input to the modem.
4. CTS (clear to send). In response to RTS, when the modem has room to store the data it is to receive, it sends out signal CTS to the DTE (PC) to indicate that it can receive the data now. This input signal to the DTE is used by the DTE to start transmission.
5. DCD (data carrier detect). The modem asserts signal DCD to inform the DTE (PC) that a valid carrier has been detected and that contact between it and the other modem is established. Therefore, DCD is an output from the modem and an input to the PC (DTE).
6. RI (ring indicator). An output from the modem (DCE) and an input to a PC

(DTE) indicates that the telephone is ringing. RI goes on and off in synchronization with the ringing sound. Of the six handshake signals, this is the least often used because modems take care of answering the phone. If in a given system the PC is in charge of answering the phone, however, this signal can be used.

From the above description, PC and modem communication can be summarized as follows: While signals DTR and DSR are used by the PC and modem, respectively, to indicate that they are alive and well, it is RTS and CTS that actually control the flow of data. When the PC wants to send data it asserts RTS, and in response, the modem, if it is ready (has room) to accept the data, sends back CTS. If, for lack of room, the modem does not activate CTS, the PC will deassert DTR and try again. RTS and CTS are also referred to as hardware control flow signals.

This concludes the description of the most important pins of the RS232 handshake signals plus TX, RX, and ground. Ground is also referred to as SG (signal ground).

PC COM ports

The x86 PCs (based on 8086, 286, 386, 486, and Pentium microprocessors) used to have two COM ports. Both COM ports were RS232-type connectors. The COM ports were designated as COM 1 and COM 2. In recent years, COM ports have been replaced with the USB port. In the absence of COM ports, we can use a COM-to-USB converter module, to connect the AVR serial port to the USB port of a PC. Some trainer boards, including the Arduino boards, have on-board Serial-to-USB converters and connect directly to USB ports.

With this background in serial communication, we are ready to look at the AVR. In the next section we discuss the physical connection of the AVR and RS232 connector, and in Section 11.3 we see how to program the AVR serial communication port.

Review Questions

1. The transfer of data using parallel lines is _____ (more, less) expensive.
2. True or false. Sending data from a radio station is duplex.
3. True or false. In full duplex we must have two data lines, one for transfer and one for receive.
4. The start and stop bits are used in the _____ (synchronous, asynchronous) method.
5. Assuming that we are transmitting the ASCII letter "E" (0100 0101 in binary) with no parity bit and one stop bit, show the sequence of bits transferred serially.
6. In Question 5, find the overhead due to framing.
7. Calculate the time it takes to transfer 10,000 characters as in Question 5 if we use 9600 bps. What percentage of time is wasted due to overhead?
8. True or false. RS232 is not TTL compatible.
9. What voltage levels are used for binary 0 in RS232?
10. True or false. The AVR has a built-in UART.

SECTION 11.2: ATMEGA328 CONNECTION TO RS232

In this section, the details of the physical connections of the ATmega328 to RS232 connectors are given. As stated in Section 11.1, the RS232 standard is not TTL compatible; therefore, a line driver such as the MAX232 chip is required to convert RS232 voltage levels to TTL levels, and vice versa. The interfacing of ATmega328 with RS232 connectors via the MAX232 chip is the main topic of this section.

RX and TX pins in the ATmega328

The ATmega328 has two pins that are used specifically for transferring and receiving data serially. These two pins are called RXD and TXD (or simply RX and TX) and are part of Port D (PD0 and PD1). Pin PD1 of the ATmega328 is assigned to TXD and pin PD0 is designated as RXD. These pins are TTL compatible; therefore, they require a line driver to make them RS232 compatible. One such line driver is the MAX232 chip. This is discussed next.

MAX232

Because the RS232 is not compatible with today's microprocessors and microcontrollers, we need a line driver (voltage converter) to convert the RS232's signals to TTL voltage levels that will be acceptable to the AVR's TX and RX pins. One example of such a converter is MAX232 from Maxim Corp. (www.maxim-ic.com). The MAX232 converts from RS232 voltage levels to TTL voltage levels, and vice versa. One advantage of the MAX232 chip is that it uses a +5 V power source, which is the same as the source voltage for the AVR. In other words, with a single +5 V power supply we can power both the AVR and MAX232, with no need for the dual power supplies that are common in many older systems.

The MAX232 has two sets of line drivers for transferring and receiving data, as shown in Figure 11-7. The line drivers used for TX are called T1 and T2, while the line drivers for RX are designated as R1 and R2. In many applications only one of each is used. For example, T1 and R1 are used together for TX and RX of the AVR, and the second set is left unused. Notice in MAX232 that the T1 line driver has a designation of T1in and T1out on pin numbers 11 and 14, respectively. The T1in pin is the TTL side and is connected to TX of the microcontroller, while T1out is the RS232 side that is connected to the RX pin of the RS232 DB connector. The R1 line driver has a designation of R1in and R1out on pin numbers 13 and 12, respectively. The R1in (pin 13) is the RS232 side that is connected to the TX pin of the RS232 DB connector, and R1out (pin 12) is the TTL side that is connected to the RX pin of the microcontroller. See Figure 11-7. Notice the null modem connection where RX for one is TX for the other.

MAX232 requires four capacitors ranging from 0.1 to 22 μF. The most widely used value for these capacitors is 22 μF.

MAX233

To save board space, some designers use the MAX233 chip from Maxim. The MAX233 performs the same job as the MAX232 but eliminates the need for

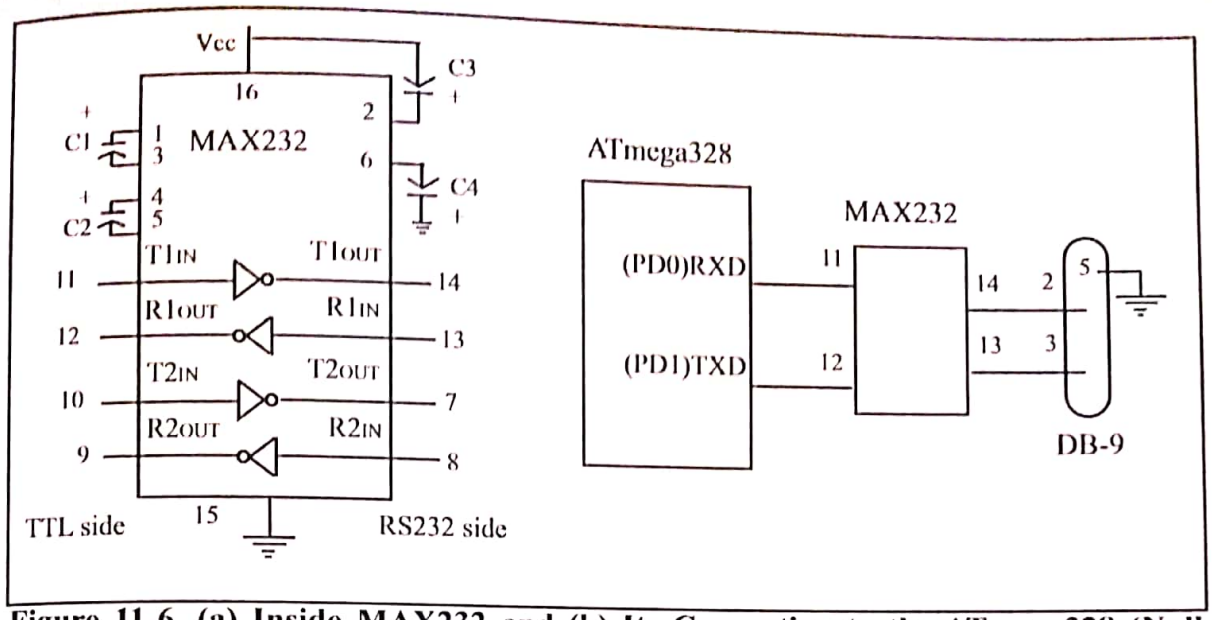

Figure 11-6. (a) Inside MAX232 and (b) Its Connection to the ATmega328 (Null Modem)

capacitors. However, the MAX233 chip is much more expensive than the MAX232. Notice that MAX233 and MAX232 are not pin compatible. You cannot take a MAX232 out of a board and replace it with a MAX233. See Figure 11-8 for MAX233 with no capacitor used.

Review Questions

1. True or false. The PC COM port connector is the RS232 type.
2. Which pins of the ATmega328 are set aside for serial communication, and what are their functions?
3. What are line drivers such as MAX 232 used for?
4. MAX232 can support ____ lines for TX and ____ lines for RX.
5. What is the advantage of the MAX233 over the MAX232 chip?

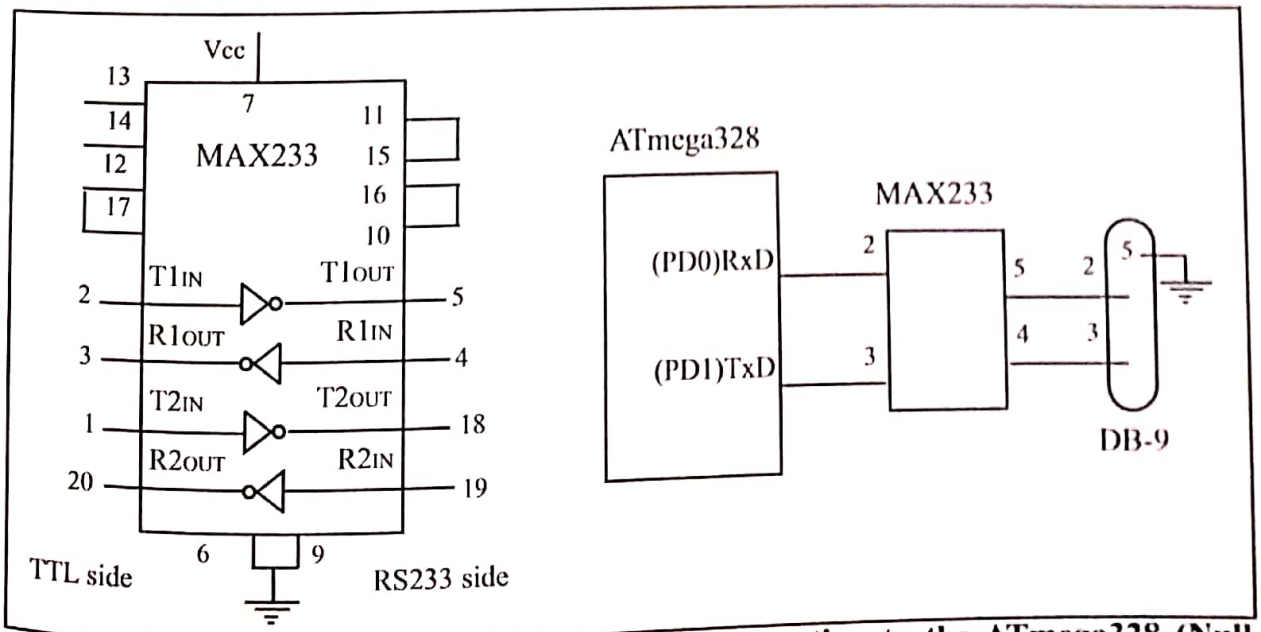

Figure 11-7. (a) Inside MAX233 and (b) Its Connection to the ATmega328 (Null Modem)

SECTION 11.3: AVR SERIAL PORT PROGRAMMING IN ASSEMBLY

In this section we discuss the serial communication registers of the ATmega328 and show how to program them to transfer and receive data using asynchronous mode. The USART (universal synchronous asynchronous receiver/transmitter) in the AVR has normal asynchronous, double-speed asynchronous, master synchronous, and slave synchronous mode features. The synchronous mode can be used to transfer data between the AVR and external peripherals such as ADC and EEPROMs. The asynchronous mode is the one we will use to connect the AVR-based system to the x86 PC serial port for the purpose of full-duplex serial data transfer. In this section we examine the asynchronous mode only.

In the Arduino Uno, the UART0 port of the Atmega328 is connected to the on-board Serial-to-USB converter, which is connected to a USB connector. This converter has two functions:

a) the programming (downloading),

b) the use as a virtual COM port for serial communications.

When the USB cable connects the PC to the Arduino board, the device driver at the host PC establishes a virtual connection between the PC and the UART of the AVR microcontroller. On the host PC, it appears as a COM port and will work with communication software on the PC such as a terminal emulator.

Examining the datasheet of the Atmega328, we see the UART0 uses PD0 and PD1 pins as alternate functions for RXD and TXD, respectively. See Figure 11-8.

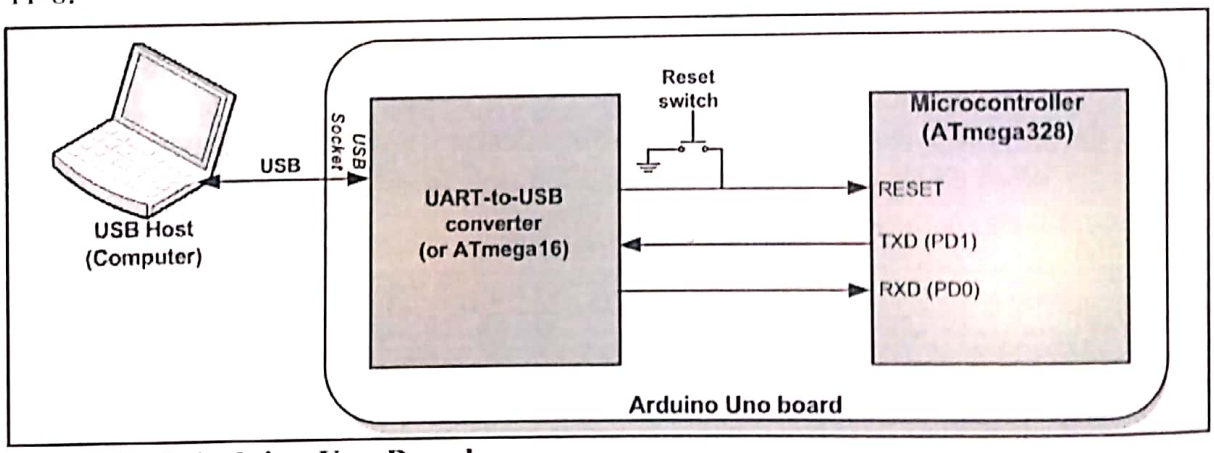

Figure 11-8. Arduino Uno Board

AVRs can have up to 4 USART ports. They are designated as USART0, USART1, USART2, and USART3. In Atmega328, there is only one USART. Figure 11-9 shows the simplified block diagram of the USART units.

In all microcontrollers, there are 3 groups of registers in UART peripherals:

1. **Configuration (Control) registers:** Before using the UART peripheral the configuration registers must be initialized. This sets some parameters of the communication including: Baud rate, word length, stop bit, interrupts (if needed). In ATmega328 microcontroller, some of the configuration registers are: UBRR, UCSR0A, UCSR0B, and UCSR0C.

2. **Transmit and receive register:** To send data, we simply write to the

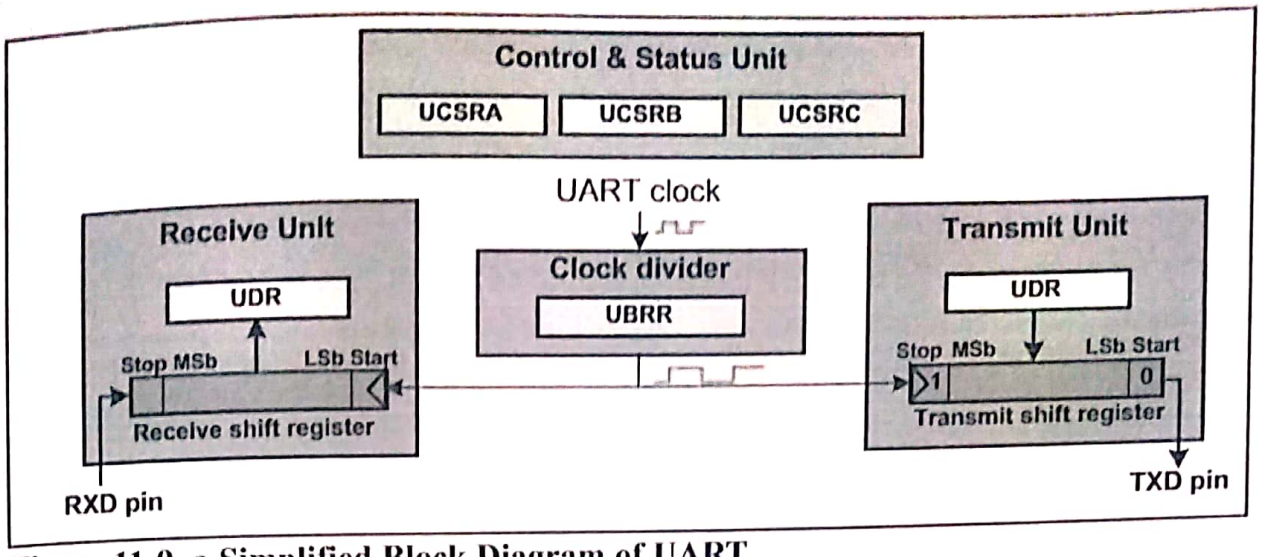

Figure 11-9. a Simplified Block Diagram of UART

transmit register. The UART peripheral sends out the contents of the transfer register through the serial transmit pin (TXD). The received data is stored in the receive register. In AVR, the transfer and receive registers are named as UDR

3. **Status register:** the status register contains some flags which show the state of sending and receiving data including: the transmitter sent out the entire byte, the transmitter is ready for another byte of data, the receiver received a whole byte of data, etc. The status register is named as UCSRA in AVR.

Next, we examine each of the registers and show how they are used in full-duplex serial data communication.

UBRR register and baud rate in the AVR

Some of the standard baud rates are listed in Table 11-2. The AVR transfers and receives data serially at many different baud rates. The baud rate in the AVR is programmable. This is done with the help of the 8-bit register called UBRR. For a given crystal frequency, the value loaded into the UBRR decides the baud rate. The relation between the value loaded into UBBR and the Fosc (frequency of oscillator connected to the XTAL1 and XTAL2 pins) is dictated by the following formula:

Table 11-2: Some Standard Baud Rates

1,200
2,400
4,800
9,600
19,200
38,400
57,600
115,200

$$\textbf{Desired Baud Rate} = \textbf{Fosc/ (16(X + 1))}$$

where X is the value we load into the UBRR register. To get the X value for different baud rates we can solve the equation as follows:

$$\textbf{X} = \textbf{(Fosc/ (16(Desired Baud Rate))) – 1}$$

Assuming that Fosc = 16 MHz, we have the following:

$$\textbf{Desired Baud Rate} = \textbf{Fosc/ (16(X + 1))} = \textbf{16 MHz/16(X + 1)} = \textbf{1000 kHz/(X + 1)}$$

$$\textbf{X} = \textbf{(1000 kHz/ Desired Baud Rate) – 1}$$

Table 11-3: UBRR Values for Various Baud Rates (Fosc=16 MHz, U2X=0)		
Baud Rate	**UBRR (Decimal Value)**	**UBRR (Hex Value)**
38400	25	19
19200	51	33
9600	103	67
4800	207	CF
2400	416	1A0
1200	832	340
Note: For Fosc = 16 MHz we have UBRR = (1000000/BaudRate) – 1		

Table 11-3 shows the X values for the different baud rates if Fosc = 16 MHz. Another way to understand the UBRR values listed in Table 11-3 is to look at Figure 11-10. The UBRR is connected to a down-counter, which functions as a programmable prescaler to generate baud rate. The system clock (Fosc) is the clock input to the down-counter. The down-counter is loaded with the UBRR value each time it counts down to zero. When the counter reaches zero, a clock is generated. This makes a frequency divider that divides the OSC frequency by UBRR+1. Then the frequency is divided by 2, 4, and 2. See Example 11-1. As you

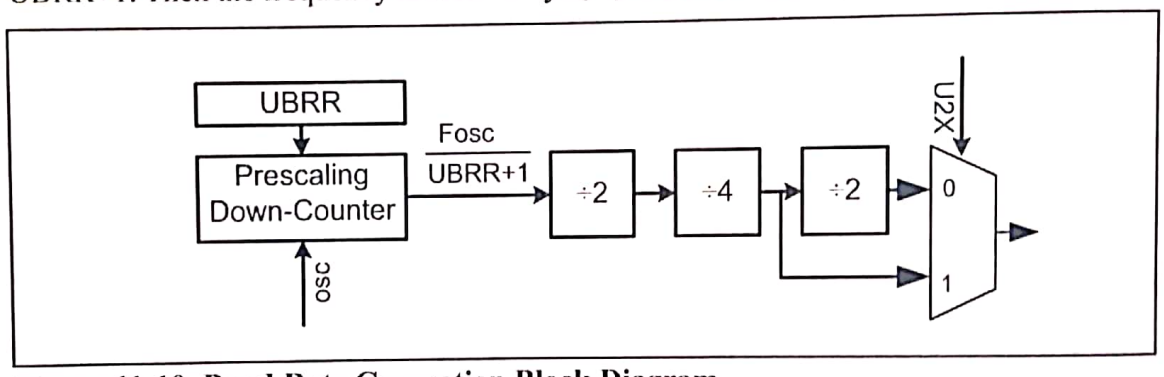

Figure 11-10. Baud Rate Generation Block Diagram

Example 11-1

With Fosc = 16 MHz, find the UBRR value needed to have the following baud rates:
(a) 9600 (b) 4800 (c) 2400 (d) 1200

Solution:

Fosc = 16 MHz => X = (16 MHz/16(Desired Baud Rate)) – 1
=> X = (1000 kHz/(Desired Baud Rate)) – 1

(a) (1000 kHz/ 9600) – 1 = 104.16 – 1 = 103.16 = 103 = 67 (hex) is loaded into UBRR
(b) (1000 kHz/ 4800) – 1 = 208.33 – 1 = 207.33 = 207 = CF (hex) is loaded into UBRR
(c) (1000 kHz/ 2400) – 1 = 416.66 – 1 = 415.66 = 416 = 1A0 (hex) is loaded into UBRR
(d) (1000 kHz/ 1200) – 1 = 833.33 – 1 = 832.33 = 832 = 340 (hex) is loaded into UBRR

Notice that dividing the output of the prescaling down-counter by 16 is the default setting upon Reset. We can get a higher baud rate with the same crystal by changing this default setting. This is explained at the end of this section.

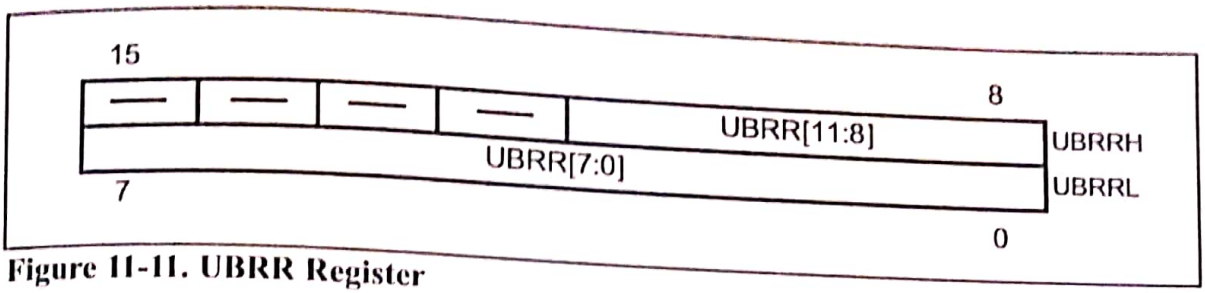

Figure 11-11. UBRR Register

can see in Figure 11-10, we can choose to bypass the last divider and double the baud rate. In the next section we learn more about it.

As you see in Figure 11-11, UBRR is a 16-bit register but only 12 bits of it are used to set the USART baud rate. The other bits are reserved.

UDR registers and USART data I/O in the AVR

In the AVR, to provide a full-duplex serial communication, there are two shift registers referred to as *Transmit Shift Register* and *Receive Shift Register*. Each shift register has a buffer that is connected to it directly. These buffers are called *Transmit Data Buffer Register* and *Receive Data Buffer Register*. The USART Transmit Data Buffer Register and USART Receive Data Buffer Register share the same I/O address, which is called *USART Data Register* or *UDR*. When you write data to UDR, it will be transferred to the Transmit Data Buffer Register (TXB), and when you read data from UDR, it will return the contents of the Receive Data Buffer Register (RXB). See Figures 11-12A and 11-12B.

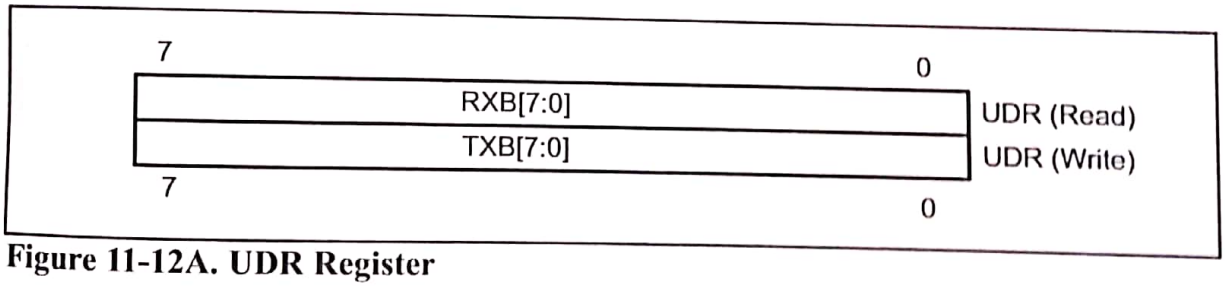

Figure 11-12A. UDR Register

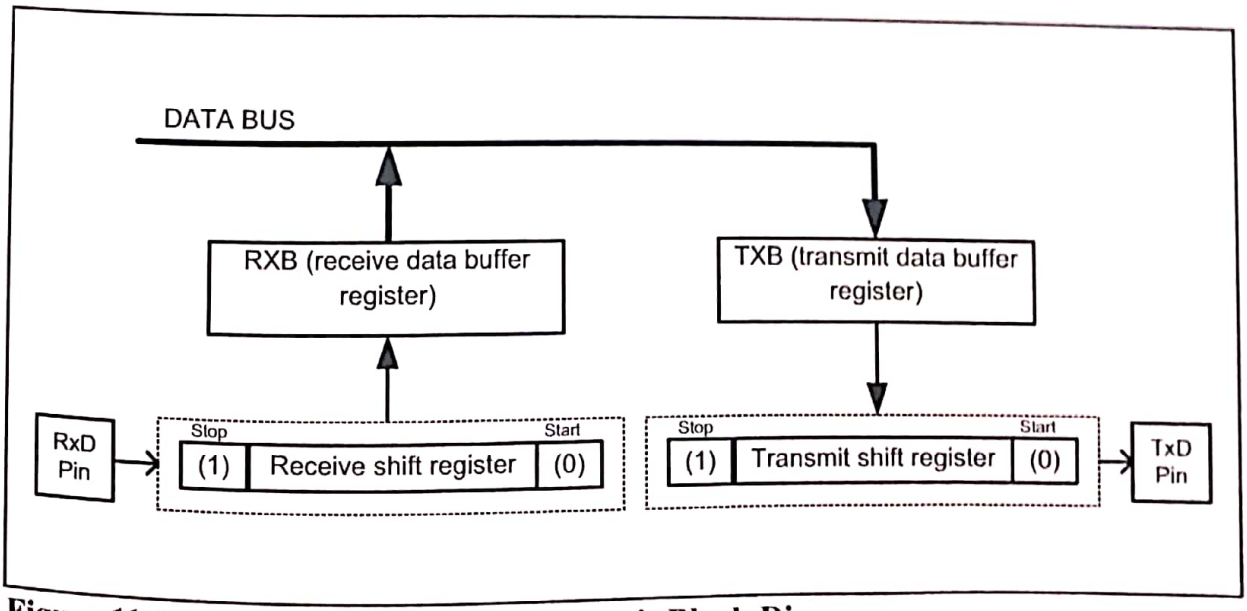

Figure 11-12B. Simplified USART Transmit Block Diagram

UCSR registers and USART configurations in the AVR

UCSRs are 8-bit control registers used for controlling serial communication in the AVR. There are three USART Control Status Registers in the AVR. They are UCSR0A, UCSR0B, and UCSR0C. In Figures 11-13 to 11-15 you can see the role of each bit in these registers. Examine these figures carefully before continuing to read this chapter. As you see in Figure 11-3, UCSR0A contains both control bits and

RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0

RXC0 (Bit 7): USART Receive Complete 0
This flag bit is set when there are new data in the receive buffer that are not read yet. It is cleared when the receive buffer is empty. It also can be used to generate a receive complete interrupt.

TXC0 (Bit 6): USART Transmit Complete 0
This flag bit is set when the entire frame in the transmit shift register has been transmitted and there are no new data available in the transmit data buffer register (TXB). It can be cleared by writing a one to its bit location. Also it is automatically cleared when a transmit complete interrupt is executed. It can be used to generate a transmit complete interrupt.

UDRE0 (Bit 5): USART Data Register Empty 0
This flag is set when the transmit data buffer is empty and it is ready to receive new data. If this bit is cleared you should not write to UDR0 because it overrides your last data. The UDRE0 flag can generate a data register empty interrupt.

FE0 (Bit 4): Frame Error 0
This bit is set if a frame error has occurred in receiving the next character in the receive buffer. A frame error is detected when the first stop bit of the next character in the receive buffer is zero.

DOR0 (Bit 3): Data OverRun 0
This bit is set if a data overrun is detected. A data overrun occurs when the receive data buffer and receive shift register are full, and a new start bit is detected.

PE0 (Bit 2): Parity Error 0
This bit is set if parity checking was enabled (UPM1 = 1) and the next character in the receive buffer had a parity error when received.

U2X0 (Bit 1): Double the USART Transmission Speed 0
Setting this bit will double the transfer rate for asynchronous communication.

MPCM0 (Bit 0): Multi-processor Communication Mode 0
This bit enables the multi-processor communication mode. The MPCM0 feature is not discussed in this book.

Notice that FE0, DOR0, and PE0 are valid until the receive buffer (UDR0) is read. Always set these bits to zero when writing to UCSR0A.

Figure 11-13. UCSR0A: USART Control and Status Register 0 A

status flags. U2X0 and MPCM0 are control bits and the other bits are status flags.

Three of the UCSR0B register bits are related to interrupt. They are RXCIE0, TXCIE0, and UDRIE0. See Figure 11-14. In Section 11-5 we will see how these flags are used with interrupts. In this section we monitor (poll) the UDRE0 flag bit to make sure that the transmit data buffer is empty and it is ready to receive new data. By the same logic, we monitor the RXC0 flag to see if a byte of data has come in yet.

Before you start serial communication you have to enable the USART receiver or USART transmitter by writing one to the RXEN0 or TXEN0 bit of UCSR0B. As we mentioned before, in the AVR you can use either synchronous or asynchronous operating mode. The UMSEL01:00 bits of the UCSR0C register select the USART operating mode. Since we want to use asynchronous USART operating mode, we have to set the UMSEL bits to zeros. Also you have to set an identical character size for both transmitter and the receiver. If the character size of the receiver device does not match the character size of the transmitter, data transfer would fail. Parity mode and number of stop bits are other factors that the receiver and transmitter must agree on before starting USART communication.

RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80

RXCIE0 (Bit 7): Receive Complete Interrupt Enable
To enable the interrupt on the RXC0 flag in UCSR0A you should set this bit to one.

TXCIE0 (Bit 6): Transmit Complete Interrupt Enable
To enable the interrupt on the TXC0 flag in UCSR0A you should set this bit to one.

UDRIE0 (Bit 5): USART Data Register Empty Interrupt Enable
To enable the interrupt on the UDRE0 flag in UCSR0A you should set this bit to one.

RXEN0 (Bit 4): Receive Enable
To enable the USART receiver you should set this bit to one.

TXEN0 (Bit 3): Transmit Enable
To enable the USART transmitter you should set this bit to one.

UCSZ02 (Bit 2): Character Size
This bit combined with the UCSZ1:0 bits in UCSRC sets the number of data bits (character size) in a frame.

RXB80 (Bit 1): Receive data bit 8
This is the ninth data bit of the received character when using serial frames with nine data bits. This bit is not used in this book.

TXB80 (Bit 0): Transmit data bit 8
This is the ninth data bit of the transmitted character when using serial frames with nine data bits. This bit is not used in this book.

Figure 11-14. UCSR0B: USART Control and Status Register 0 B

UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOL0

UMSEL01:00 (Bits 7:6): USART Mode Select
These bits select the operation mode of the USART:
00 = Asynchronous USART operation
01 = Synchronous USART operation
10 = Reserved
11 = Master SPI (MSPIM)

UPM01:00 (Bit 5:4): Parity Mode
These bits disable or enable and set the type of parity generation and check.
00 = Disabled
01 = Reserved
10 = Even Parity
11 = Odd Parity

USBS0 (Bit 3): Stop Bit Select
This bit selects the number of stop bits to be transmitted.
0 = 1 bit
1 = 2 bits

UCSZ01:00 (Bit 2:1): Character Size
These bits combined with the UCSZ02 bit in UCSR0B set the character size in a frame and will be discussed more in this section.

UCPOL0 (Bit 0): Clock Polarity
This bit is used for synchronous mode only and will not be covered in this section.

Figure 11-15. UCSR0C: USART Control and Status Register 0 C

To set the number of data bits (character size) in a frame you must set the values of the UCSZ01 and USCZ00 bits in the UCSR0B and UCSZ02 bits in UCSR0C. Table 11-4 shows the values of UCSZ02, UCSZ01, and UCSZ00 for different character sizes. In this book we use the 8-bit character size because it is the most common in serial communications. If you want to use 9-bit data, you have to use the RXB80 and TXB80 bits in UCSR0B as the 9th bit of UDR0 (USART Data Register). See Examples 11-2 and 11-3.

Table 11-4: Values of UCSZ02:00 for Different Character Sizes

UCSZ02	UCSZ01	UCSZ00	Character Size
0	0	0	5
0	0	1	6
0	1	0	7
0	1	1	8
1	1	1	9

Note: Other values are reserved. Also notice that UCSZ00 and UCSZ01 belong to UCSR0C and UCSZ02 belongs to UCSR0B

Example 11-2

(a) What are the values of UCSR0B and UCSR0C needed to configure USART for asynchronous operating mode, 8 data bits (character size), no parity, and 1 stop bit? Enable both receive and transmit.
(b) Write a program for the AVR to set the values of UCSR0B and UCSR0C for this configuration.

Solution:

(a) RXEN0 and TXEN0 have to be 1 to enable receive and transmit. UCSZ02:00 should be 011 for 8-bit data, UMSEL01:00 should be 00 for asynchronous operating mode, UPM01:00 have to be 00 for no parity, and USBS0 should be 0 for one stop bit.
(b)

Assembly:

```
      LDI   R16,(1<<RXEN0)|(1<<TXEN0)
      STS   UCSR0B, R16
;Note that, in the next line, instead of using shift operator,
;you can write "LDI R16, 0b00000110"
      LDI   R16,(1<<UCSZ01)|(1<<UCSZ00)
      STS   UCSR0C, R16
```

C:

```
      UCSR0B = (1<<RXEN0)|(1<<TXEN0);
      UCSR0C = (1<<UCSZ01)|(1<<UCSZ00);
```

Example 11-3

In Example 11-2, set the baud rate to 1200 and write a program for the AVR to set up the values of UCSR0B, UCSR0C, and UBRR. (Focs = 16 MHz)

Solution in Assembly:

```
      LDI   R16,(1<<RXEN0)|(1<<TXEN0)
      STS   UCSR0B, R16
;Note that, in the next line, instead of using shift operator,
;you can write "LDI R16, 0b00000110"
      LDI   R16,(1<<UCSZ01)|(1<<UCSZ00)
      STS   UCSR0C, R16       ;move R16 to UCSR0C
      LDI   R16,0x40          ;UBRR0 = 0x340 (see Table 11-3)
      STS   UBRR0L,R16        ;UBRR0L = 0x40
      LDI   R16,0x3           ;R16 = 0x3
      STS   UBRR0H,R16        ;UBRR0H = 0x3
```

Solution in C:

```
      UCSR0B = (1<<RXEN0)|(1<<TXEN0);
      UCSR0C = (1<<UCSZ01)|(1<<UCSZ00);
      UBRR0L = 0x40;
      UBRR0H = 0x3;
```

FE0 and PE0 flag bits

When the AVR USART receives a byte, we can check the parity bit and stop bit. If the parity bit is not correct, the AVR will set PE0 to one, indicating that a parity error has occurred. We can also check the stop bit. As we mentioned before, the stop bit must be one, otherwise the AVR would generate a stop bit error and set the FE0 flag bit to one, indicating that a stop bit error has occurred. We can check these flags to see if the received data is valid and correct. Notice that FE0 and PE0 are valid until the receive buffer (UDR0) is read. So we have to read FE0 and PE0 bits before reading UDR0. You can explore this on your own.

Programming the AVR to transfer data serially

In programming the AVR to transfer character bytes serially, the following steps must be taken:

1. The UCSR0B register is loaded with the value 08H, enabling the USART transmitter. The transmitter will override normal port operation for the TxD pin when enabled.
2. The UCSR0C register is loaded with the value 06H, indicating asynchronous mode with 8-bit data frame, no parity, and one stop bit.
3. The UBRR0 is loaded with one of the values in Table 11-3 (if Fosc = 16 MHz) to set the baud rate for serial data transfer.
4. The character byte to be transmitted serially is written into the UDR0 register.
5. Monitor the UDRE0 bit of the UCSR0A register to make sure UDR0 is ready for the next byte.
6. To transmit the next character, go to Step 4.

Example 11-4 (Assembly Version of Example 11-10)

Write a program for the AVR to transfer the letter 'G' serially at 9600 baud, continuously. Assume XTAL = 16 MHz.

Solution:

```
      LDI   R16,(1<<TXEN0)   ;enable transmitter
      STS   UCSR0B, R16
      LDI   R16,(1<<UCSZ01)|(1<<UCSZ00) ;8-bit data
      STS   UCSR0C, R16      ;no parity, 1 stop bit
      LDI   R16,103          ;9600 baud rate
      STS   UBRR0L,R16       ;for XTAL = 16 MHz
AGAIN:
      LDS   R20, UCSR0A
      SBRS  R20, UDRE0       ;is UDR0 empty?
      RJMP  AGAIN            ;wait more
      LDI   R16,'G'          ;send 'G'
      STS   UDR0,R16         ;to UDR0
      RJMP  AGAIN            ;do it again
```

Importance of monitoring the UDRE0 flag

By monitoring the UDRE0 flag, we make sure that we are not overloading the UDR0 register. If we write another byte into the UDR0 register before it is empty, the old byte could be lost before it is transmitted.

We conclude that by checking the UDRE0 flag bit, we know whether or not the AVR is ready to accept another byte to transmit. The UDRE0 flag bit can be checked by the instruction "SBRS R20,UDRE0" or we can use an interrupt, as we will see in Section 11.5. In Section 11.5 we will show how to use interrupts to transfer data serially, and avoid tying down the microcontroller with instructions such as "SBRS R20,UDRE". Example 11-4 shows the program to transfer 'G' serially at 9600 baud.

Example 11-5 shows how to transfer "YES " continuously.

Example 11-5

Write a program to transmit the message "YES " serially at 9600 baud, 8-bit data, and 1 stop bit. Do this forever.

Solution:

```
      LDI   R21,HIGH(RAMEND)            ;initialize high
      OUT   SPH,R21                     ;byte of SP
      LDI   R21,LOW(RAMEND)             ;initialize low
      OUT   SPL,R21                     ;byte of SP
      LDI   R16,(1<<TXEN0)              ;enable transmitter
      STS   UCSR0B, R16
      LDI   R16,(1<<UCSZ01)|(1<<UCSZ00) ;8-bit data
      STS   UCSR0C, R16                 ;no parity, 1 stop bit
      LDI   R16,103                     ;9600 baud rate
      STS   UBRR0L,R16                  ;for XTAL = 16 MHz

AGAIN:
      LDI   R17,'Y'                     ;move 'Y' to R17
      CALL  TRNSMT                      ;transmit r17 to TxD
      LDI   R17,'E'                     ;move 'E' to R17
      CALL  TRNSMT                      ;transmit r17 to TxD
      LDI   R17,'S'                     ;move 'S' to R17
      CALL  TRNSMT                      ;transmit r17 to TxD
      LDI   R17,' '                     ;move ' ' to R17
      CALL  TRNSMT                      ;transmit space to TxD
      RJMP  AGAIN                       ;do it again

TRNSMT:
      LDS   R20, UCSR0A
      SBRS  R20, UDRE0        ;is UDR0 empty?
      RJMP  TRNSMT            ;wait more
      STS   UDR0,R17          ;to UDR0
      RET                     ;return
```

Programming the AVR to receive data serially

In programming the AVR to receive character bytes serially, the following steps must be taken:

1. The UCSR0B register is loaded with the value 10H, enabling the USART receiver. The receiver will override normal port operation for the RxD pin when enabled.
2. The UCSR0C register is loaded with the value 06H, indicating asynchronous mode with 8-bit data frame, no parity, and one stop bit.
3. The UBRR0 is loaded with one of the values in Table 11-3 (if Fosc = 16 MHz) to set the baud rate for serial data transfer.
4. The RXC0 flag bit of the UCSR0A register is monitored for a HIGH to see if an entire character has been received yet.
5. When RXC0 is raised, the UDR0 register has the byte. Its contents are moved into a safe place.
6. To receive the next character, go to Step 5.

Example 11-6 shows the coding of the above steps.

Example 11-6 (Assembly Version of Example 11-12)

Program the ATmega328 to receive bytes of data serially and put them on Port B. Set the baud rate at 9600, 8-bit data, and 1 stop bit.

Solution:

```
        LDI   R16,(1<<RXEN0)    ;enable receiver
        STS   UCSR0B, R16
        LDI   R16,(1<<UCSZ01)|(1<<UCSZ00) ;8-bit data
        STS   UCSR0C, R16       ;no parity, 1 stop bit
        LDI   R16,103           ;9600 baud rate
        STS   UBRR0L,R16
        LDI   R16,0xFF          ;Port B is output
        OUT   DDRB,R16
RCVE:   LDS   R20,UCSR0A
        SBRS  R20,RXC0          ;is any byte in UDR0?
        RJMP  RCVE              ;wait more
        LDS   R17,UDR0          ;send UDR0 to R17
        OUT   PORTB,R17         ;send R17 to PORTB
        RJMP  RCVE              ;do it again
```

Transmit and receive

In previous examples we showed how to transmit or receive data serially. Next we show how do both send and receive at the same time in a program. Assume that the AVR serial port is connected to the COM port of the PC, and we are using a terminal program on the PC to send and receive data serially. Ports B and C of the AVR are connected to switches and LEDs, respectively. Example 11-7 shows an AVR program with the following parts: (a) sends the message "YES" once to the PC screen, (b) gets data on switches and transmits it via the serial port to the PC's screen, and (c) receives any key press sent by the terminal program and puts it on LEDs.

Example 11-7

Write an AVR program with the following parts: (a) send the message "YES" once to the PC screen, (b) get data from switches on Port C and transmit it via the serial port to the PC's screen, and (c) receive any key press sent by a terminal and put it on LEDs. The programs must do parts (b) and (c) repeatedly.

Solution:

```
      LDI   R21,0x00
      OUT   DDRC,R21                    ;Port C is input
      LDI   R21,0xFF
      OUT   PORTC,R21                   ;enable pull-up
      OUT   DDRB,R21                    ;Port B is output

      LDI   R21,HIGH(RAMEND)            ;initialize high
      OUT   SPH,R21                     ;byte of SP
      LDI   R21,LOW(RAMEND)             ;initialize low
      OUT   SPL,R21                     ;byte of SP

      LDI   R16,(1<<TXEN0)|(1<<RXEN0)   ;enable transmitter
      STS   UCSR0B, R16                 ;and receiver
      LDI   R16,(1<<UCSZ01)|(1<<UCSZ00) ;8-bit data
      STS   UCSR0C, R16                 ;no parity, 1 stop bit
      LDI   R16,103                     ;9600 baud rate
      STS   UBRR0L,R16

      LDI   R17,'Y'                     ;move 'Y' to R17
      CALL  TRNSMT                      ;transmit r17 to TxD
      LDI   R17,'E'                     ;move 'E' to R17
      CALL  TRNSMT                      ;transmit r17 to TxD
      LDI   R17,'S'                     ;move 'S' to R17
      CALL  TRNSMT                      ;transmit r17 to TxD
AGAIN:
      LDS   R16,UCSR0A
      SBRS  R16,RXC0                    ;is there new data?
      RJMP  SKIP_RX                     ;skip receive cmnds
      LDS   R17,UDR0                    ;move UDR0 to R17
      OUT   PORTB,R17                   ;move R17 TO PORTB

SKIP_RX:
      LDS   R16,UCSR0A
      SBRS  R16,UDRE0                   ;is UDR0 empty?
      RJMP  SKIP_TX                     ;skip transmit cmnds
      IN    R17,PINC                    ;move Port C to R17
      STS   UDR0,R17                    ;send R17 to UDR0
SKIP_TX:
      RJMP  AGAIN                       ;do it again

TRNSMT:
      LDS   R16,UCSR0A
      SBRS  R16,UDRE0                   ;is UDR0 empty?
      RJMP  TRNSMT                      ;wait more
      STS   UDR0,R17                    ;send R17 to UDR0
      RET
```

Doubling the baud rate in the AVR

There are two ways to increase the baud rate of data transfer in the AVR:

1. Use a higher-frequency crystal.
2. Change a bit in the UCSR0A register, as shown below.

Option 1 is not feasible in many situations because the system crystal is fixed. Therefore, we will explore option 2. There is a software way to double the baud rate of the AVR while the crystal frequency stays the same. This is done with the U2X0 bit of the UCSR0A register. When the AVR is powered up, the U2X0 bit of the UCSR0A register is zero. We can set it to high by software and thereby double the baud rate.

To see how the baud rate is doubled with this method, look again at Figure 11-9. If we set the U2X0 bit to HIGH, the third frequency divider will be bypassed. In the case of XTAL = 16 MHz and U2X0 bit set to HIGH, we would have:

Desired Baud Rate = Fosc / (8 (X + 1)) = 16 MHz / 8 (X + 1) = 2 MHz / (X + 1)

To get the X value for different baud rates we can solve the equation as follows:

X = (2 MHz / Desired Baud Rate) – 1

In Table 11-5 you can see values of UBRR in hex and decimal for different baud rates for U2X0 = 0 and U2X0 = 1. (XTAL = 16 MHz).

Table 11-5: UBRR0 Values for Various Baud Rates (XTAL = 16 MHz)

	U2X0 = 0		U2X0 = 1	
Baud Rate	**UBRR0**	**UBRR (HEX)**	**UBRR0**	**UBR (HEX)**
38400	25	19	51	33
19200	51	33	103	67
9,600	103	67	207	CF
4,800	207	CF	415	19F
UBRR = (1 MHz / Baud rate) – 1			*UBRR = (2 MHz / Baud rate) – 1*	

Baud rate error calculation

In calculating the baud rate we have used the integer number for the UBRR register values because AVR microcontrollers can only use integer values. By dropping the decimal portion of the calculated values we run the risk of introducing error into the baud rate. There are several ways to calculate this error. One way would be to use the following formula.

Error = (Calculated value for the UBRR – Integer part) / Integer part

For example, with XTAL = 16 MHz and U2X0 = 0 we have the following for the 9600 baud rate:

UBRR value = (1000,000/ 9600) – 1 = 104.16 – 1 = 103.16 = 103

⇒ Error = (103.16 – 103)/ 103 = 0.16%

Another way to calculate the error rate is as follows:

Error = (Calculated baud rate – desired baud rate) / desired baud rate

Where the desired baud rate is calculated using X = (Fosc / (16(Desired Baud rate))) – 1, and then the integer X (value loaded into UBRR0 register) is used for the calculated baud rate as follows:

Calculated baud rate = Fosc / (16(X + 1)) (for U2X = 0)

For XTAL = 16 MHz and 9600 baud rate, we got X = 103. Therefore, we get the calculated baud rate of 16 MHz/(16(103 + 1)) = 9615. Now the error is calculated as follows:

Error = (9615 – 9600) / 9600 = 0.16%

which is the same as what we got earlier using the other method. Table 11-6 provides the error rates for UBRR0 values of 16 MHz crystal frequencies.

Table 11-6: UBRR Values for Various Baud Rates (XTAL = 16 MHz)

	U2X = 0		U2X = 1	
Baud Rate	**UBRR0**	**Error**	**UBRR0**	**Error**
38400	25	0.2%	51	0.2%
19200	51	0.2%	103	0.2%
9,600	103	0.2%	207	0.2%
4,800	207	0.2%	415	0.2%
UBRR = (1,000,000 / Baud rate) – 1			*UBRR = (2,000,000 / Baud rate) – 1*	

In some applications we need very accurate baud rate generation. In these cases we can use a 7.3728 MHz or 11.0592 MHz crystal. As you can see in Table 11-7, the error is 0% if we use a 7.3728 MHz crystal. In the table there are values of UBRR0 for different baud rates for U2X = 0 and U2X = 1.

Table 11-7: UBRR Values for Various Baud Rates (XTAL = 7.3728 MHz)

	U2X = 0		U2X = 1	
Baud Rate	**UBRR**	**Error**	**UBRR**	**Error**
38400	11	0%	23	0%
19200	23	0%	47	0%
9,600	47	0%	95	0%
4,800	95	0%	191	0%
UBRR = (460,800 / Baud rate) – 1			*UBRR = (921,600 / Baud rate) – 1*	

See Example 11-8 to see how to calculate the error for different baud rates.

Example 11-8

Assuming XTAL = 10 MHz, calculate the baud rate error for each of the following:
(a) 2400 (b) 9600 (c) 19,200 (d) 57,600
Use the U2X = 0 mode.

Solution:

UBRR Value = (Fosc / 16(baud rate)) – 1 , Fosc = 10 MHz =>

(a) UBRR Value = (625,000 / 2400) – 1 = 260.41 – 1 = 259.41 = 259

Error = (259.41c259) / 259 = 0.158%

(b) UBRR Value (625,000 / 9600) – 1 = 65.104 – 1 = 64.104 = 64

Error = (64.104 – 64)/64 = 0.162%

(c) UBRR Value (625,000 / 19,200) – 1 = 32.55 – 1 = 31.55 = 31

Error = (31.55 – 31) / 31 = 1.77%

(d) UBRR Value (625,000 / 57,600) – 1 = 10.85 – 1 = 9.85 = 9

Error = (9.85 – 9) / 9 = 9.4%

Review Questions

1. Which registers of the AVR are used to set the baud rate?
2. If XTAL = 10 MHz, what value should be loaded into UBRR0 to have a 14,400 baud rate? Give the answer in both decimal and hex.
3. What is the baud rate error in the last question?
4. With XTAL = 7.3728 MHz, what value should be loaded into UBRR0 to have a 9600 baud rate? Give the answer in both decimal and hex.
5. To transmit a byte of data serially, it must be placed in register _____.
6. UCSR0A stands for _____ .
7. Which bits are used to set the data frame size?
8. True or false. UCSR0A and UCSR0B share the same I/O address.
9. What parameters should the transmitter and receiver agree on before starting a serial transmission?
10. Which register has the U2X0 bit, and why do we use the U2X0 bit?

SECTION 11.4: AVR SERIAL PORT PROGRAMMING IN C

As we have seen in previous chapters, all the special function registers of the AVR are accessible directly in C compilers by using the appropriate header file. Examples 11-9 through 11-14 show how to program the serial port in C. Connect

your AVR trainer to the PC's COM port and use a terminal program to test the operation of these examples.

Example 11-9

Write a C function to initialize the USART to work at 9600 baud, 8-bit data, and 1 stop bit. Assume XTAL = 16 MHz.

Solution:

```
void usart_init (void)
{
  UCSRB = (1<<TXEN0);
  UCSRC = (1<< UCSZ01)|(1<<UCSZ00);
  UBRRL = 103;
}
```

Example 11-10 (C Version of Example 11-4)

Write a C program for the AVR to transfer the letter 'G' serially at 9600 baud, continuously. Use 8-bit data and 1 stop bit. Assume XTAL = 16 MHz.

Solution:

```
#include <avr/io.h>

void usart_init (void)
{
      UCSR0B = (1<<TXEN0);
      UCSR0C = (1<< UCSZ01)|(1<<UCSZ00);
      UBRR0L = 103;
}

void usart_send (unsigned char ch)
{
      while (!(UCSR0A&(1<<UDRE0))); //wait until UDR0 is empty
      UDR0 = ch;                    //transmit ch
}

int main (void)
{
  usart_init();             //initialize the USART

  while(1)                  //do forever
    usart_send ('G');       //transmit 'G' letter

  return 0;
}
```

Example 11-11

Write a program to send the message "The Earth is but One Country. " to the serial port continuously. Using the settings in the last example.

Solution:

```
#include <avr/io.h>
void usart_init (void)
{
      UCSR0B = (1<<TXEN0);
      UCSR0C = (1<< UCSZ01)|(1<<UCSZ00);
      UBRR0L = 103;
}
void usart_send (unsigned char ch)
{
      while (!(UCSR0A&(1<<UDRE0))); //wait until UDR0 is empty
      UDR0 = ch;                    //transmit ch
}

int main (void)
{ unsigned char str[30]= "The Earth is but One Country. ";
  unsigned char strLenght = 30;
  unsigned char i = 0;
  usart_init();
  while(1)
  {
    usart_send(str[i++]);
    if (i >= strLenght)
      i = 0;
  }
}
```

Example 11-12 (C Version of Example 11-6)

Program the AVR in C to receive bytes of data serially and put them on Port B. Set the baud rate at 9600, 8-bit data, and 1 stop bit.

Solution:

```
#include <avr/io.h>                 //standard AVR header
int main (void)
{
  DDRB = 0xFF;                      //Port B is output
  UCSR0B = (1<<RXEN0);              //initialize USART
  UCSR0C = (1<< UCSZ01)|(1<<UCSZ00);
  UBRR0L = 103;
  while(1)
  {
    while (! (UCSR0A & (1<<RXC0)));   //wait until new data
    PORTB = UDR0;
  }
  return 0;
}
```

Example 11-13

Write an AVR C program to receive a character from the serial port. If it is 'a' – 'z' change it to capital letters and transmit it back. Use the settings in the last example.

Solution:

```
#include <avr/io.h>

int main (void)
{
      UCSR0B = (1<<TXEN0)|(1<<RXEN0);     //initialize USART0
      UCSR0C = (1<< UCSZ01)|(1<<UCSZ00);
      UBRR0L = 103;

      unsigned char ch;

      while(1)
      {
            while (!(UCSR0A&(1<<RXC0))); //wait until new data

            ch = UDR0;

            if (ch >= 'a' && ch <= 'z')
            {
                  ch += ('A'-'a');
                  while (! (UCSR0A & (1<<UDRE0)));
                  UDR0 = ch;
            }
      }
      return 0;
}
```

Example 11-14

In the last five examples, what is the baud rate error?

Solution:

According to Table 11-6, for 9600 baud rate and XTAL = 16 MHz, the baud rate error is about 0.2%.

Review Questions

1. True or false. All the SFR registers of AVR are accessible in the C compiler.
2. True or false. The FE0 flag is cleared the moment we read from the UDR0 register.

SECTION 11.5: AVR SERIAL PORT PROGRAMMING IN ASSEMBLY AND C USING INTERRUPTS

By now you might have noticed that it is a waste of the microcontroller's time to poll the UDRE0 and RXC0 flags. In order to avoid wasting the microcontroller's time we use interrupts instead of polling. In this section, we will show how to use interrupts to program the AVR's serial communication port.

Interrupt-based data receive

To program the serial port to receive data using the interrupt method, we need to set HIGH the Receive Complete Interrupt Enable (RXCIE0) bit in UCSR0B. Setting this bit enables the interrupt on the RXC0 flag in UCSR0A. Upon completion of the receive, the RXC0 (USART receive complete flag) becomes HIGH. If RXCIE0 = 1, changing RXC0 to one will force the CPU to jump to the interrupt vector. Example 11-15 shows how to receive data using interrupts.

Example 11-15 (Assembly Version of Example 11-17)

Program the ATmega328 to receive bytes of data serially and put them on Port B. Set the baud rate at 9600, 8-bit data, and 1 stop bit. Use Receive Complete Interrupt instead of the polling method.

Solution:

```
      RJMP  MAIN                          ;jump main after reset
.ORG  0x24                                ;int-vector of URXC int.
      RJMP  URXC_INT_HANDLER              ;jump to URXC_INT_HANDLER

MAIN: LDI   R16, HIGH(RAMEND)             ;initialize high byte of
      OUT   SPH, R16                      ;stack pointer
      LDI   R16, LOW(RAMEND)              ;initialize low byte of
      OUT   SPL,R16                       ;stack pointer
      LDI   R16,(1<<RXEN0)|(1<<RXCIE0)    ;enable receiver
      STS   UCSR0B, R16                   ;and RXC0 interrupt
      LDI   R16,(1<<UCSZ01)|(1<<UCSZ00);async, 8-bit data
      STS   UCSR0C, R16                   ;no parity, 1 stop bit
      LDI   R16,103                       ;9600 baud rate
      STS   UBRR0L,R16
      LDI   R16,0xFF
      OUT   DDRB,R16                      ;set Port B as an
      SEI                                 ;output
WAIT_HERE:                                ;enable interrupts
      RJMP  WAIT_HERE
                                          ;stay here
URXC_INT_HANDLER:
      LDS   R17,UDR0
      OUT   PORTB,R17                     ;send UDR0 to R17
      RETI                                ;send R17 to PORTB
```

Interrupt-based data transmit

To program the serial port to transmit data using the interrupt method, we need to set HIGH the USART Data Register Empty Interrupt Enable (UDRIE0) bit in UCSRB. Setting this bit enables the interrupt on the UDRE0 flag in UCSR0A. When the UDR0 register is ready to accept new data, the UDRE (USART Data Register Empty flag) becomes HIGH. If UDRIE = 1, changing UDRE0 to one will force the CPU to jump to the interrupt vector. Example 11-16 shows how to transmit data using interrupts. To transmit data using the interrupt method, there is another source of interrupt; it is Transmit Complete Interrupt. Try to clarify the difference between these two interrupts for yourself. Can you provide some examples that the two interrupts can be used interchangeably?

Examples 11-17 and 11-18 are the C versions of Examples 11-15 and 11-16, respectively.

Example 11-16 (Assembly Version of Example 11-18)

Write a program for the AVR to transmit the letter 'G' serially at 9600 baud, continuously. Assume XTAL = 16 MHz. Use interrupts instead of the polling method.

Solution:

```
.ORG  0x00
      RJMP  MAIN                          ;jump main after reset
.ORG  0x26                                ;int. vector of UDRE int.
      RJMP  UDRE_INT_HANDLER              ;jump to UDRE_INT_HANDLER

;*****************************
MAIN:
      LDI   R16, HIGH(RAMEND)             ;initialize high byte of
      OUT   SPH, R16                      ;stack pointer
      LDI   R16, LOW(RAMEND)              ;initialize low byte of
      OUT   SPL,R16                       ;stack pointer

      LDI   R16,(1<<TXEN0)|(1<<UDRIE0)    ;enable transmitter
      STS   UCSR0B, R16                   ;and UDRE interrupt
      LDI   R16,(1<<UCSZ01)|(1<<UCSZ00)   ;async., 8-bit
      STS   UCSR0C, R16                   ;no parity, 1 stop bit
      LDI   R16,103                       ;9600 baud rate
      STS   UBRR0L,R16
      SEI                                 ;enable interrupts
WAIT_HERE:                                ;stay here
      RJMP  WAIT_HERE

;*****************************
UDRE_INT_HANDLER:                         ;send 'G'
      LDI   R26,'G'                       ;to UDR0
      STS   UDR0,R26
      RETI
```

Example 11-17 (C Version of Example 11-15)

Write a C program to receive bytes of data serially and put them on Port B. Use Receive Complete Interrupt instead of the polling method.

Solution:

```
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(USART_RX_vect)
{
      PORTB = UDR0;
}

int main (void)
{
      DDRB = 0xFF;                          //make Port B an output
      UCSR0B = (1<<RXEN0)|(1<<RXCIE0);//enable rec and RXC int.
      UCSR0C = (1<< UCSZ01)|(1<<UCSZ00);
      UBRR0L = 103;

      sei();                  //enable interrupts

      while (1);              //wait forever
}
```

Example 11-18 (C Version of Example 11-16)

Write a C program to transmit the letter 'G' serially at 9600 baud, continuously. Assume XTAL = 16 MHz. Use interrupts instead of the polling method.

Solution:

```
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(USART_UDRE_vect)
{
      UDR0 = 'G';
}

int main (void)
{
      UCSR0B = (1<<TXEN0)|(1<<UDRIE0);
      UCSR0C = (1<< UCSZ01)|(1<<UCSZ00);
      UBRR0L = 103;

      sei();
      while (1);                    //enable interrupts
      return 0;               //wait forever
}
```

Review Questions

1. What is the advantage of interrupt-based programming over polling?
2. How do you enable transmit or receive interrupts in AVR?

SUMMARY

This chapter began with an introduction to the fundamentals of serial communication. Serial communication, in which data is sent one bit a time, is used in situations where data is sent over significant distances. (In parallel communication, where data is sent a byte or more at a time, great distances can cause distortion of the data.) Serial communication has the additional advantage of allowing transmission over phone lines. Serial communication uses two methods: synchronous and asynchronous. In synchronous communication, data is sent in blocks of bytes; in asynchronous, data is sent one byte at a time. Data communication can be simplex (can send but cannot receive), half duplex (can send and receive, but not at the same time), or full duplex (can send and receive at the same time). RS232 is a standard for serial communication connectors.

The AVR's UART was discussed. We showed how to interface the ATmega328 with an RS232 connector and change the baud rate of the ATmega328. In addition, we described the serial communication features of the AVR, and programmed the ATmega328 for serial data communication. We also showed how to program the serial port of the ATmega328 chip in Assembly and C.

PROBLEMS

SECTION 11.1: BASICS OF SERIAL COMMUNICATION

1. Which is more expensive, parallel or serial data transfer?
2. True or false. 0- and 5-V digital pulses can be transferred on the telephone without being converted (modulated).
3. Show the framing of the letter ASCII 'Z' (0101 1010), no parity, 1 stop bit.
4. If there is no data transfer and the line is high, it is called _____ (mark, space).
5. True or false. The stop bit can be 1, 2, or none at all.
6. Calculate the overhead percentage if the data size is 7, 1 stop bit, and no parity bit.
7. True or false. The RS232 voltage specification is TTL compatible.
8. What is the function of the MAX232 chip?
9. How many pins of the RS232 are used by the IBM serial cable, and why?
10. True or false. The longer the cable, the higher the data transfer baud rate.
11. State the absolute minimum number of signals needed to transfer data between two PCs connected serially. What are those signals?
12. If two PCs are connected through the RS232 without a modem, both are configured as a _____ (DTE, DCE) -to- _____ (DTE, DCE) connection.
13. Calculate the total number of bits transferred if 200 pages of ASCII data are sent using asynchronous serial data transfer. Assume a data size of 8 bits, 1

stop bit, and no parity. Assume each page has 80 × 25 of text characters.

14. In Problem 13, how long will the data transfer take if the baud rate is 9600?

SECTION 11.2: ATMEGA328 CONNECTION TO RS232

15. The MAX232 DIP package has _____ pins.
16. For the MAX232, indicate the V_{CC} and GND pins.
17. The MAX233 DIP package has _____ pins.
18. For the MAX233, indicate the V_{CC} and GND pins.
19. Is the MAX232 pin compatible with the MAX233?
20. State the advantages and disadvantages of the MAX232 and MAX233.
21. MAX232/233 has _____ line driver(s) for the RX wire.
22. MAX232/233 has _____ line driver(s) for the TX wire.
23. Show the connection of pins TX and RX of the ATmega328 to a DB-9 RS232 connector via the second set of line drivers of MAX232.
24. Show the connection of the TX and RX pins of the ATmega328 to a DB-9 RS232 connector via the second set of line drivers of MAX233.
25. What is the advantage of the MAX233 over the MAX232 chip?
26. Which pins of the ATmega328 are set aside for serial communication, and what are their functions?

SECTION 11.3: AVR SERIAL PORT PROGRAMMING IN ASSEMBLY

27. Which of the following baud rates are supported by the HyperTerminal program in PC?
    (a) 4800 (b) 3600 (c) 9600
    (d) 1800 (e) 1200 (f) 19,200
28. Which register of ATmega328 is used for baud rate programming?
29. Which bit of the UCSR0A is used for baud rate speed?
30. What is the role of the UDR0 register in serial data transfer?
31. UDR is a(n) _____-bit register.
32. For XTAL = 10 MHz, find the UBRR0 value (in both decimal and hex) for each of the following baud rates.
    (a) 9600 (b) 4800 (c) 1200
33. What is the baud rate if we use UBRR = 15 to program the baud rate? Assume XTAL = 10 MHz.
34. Write an AVR program to transfer serially the letter 'Z' continuously at 9600 baud rate. Assume XTAL = 10 MHz.
35. When is the PE0 flag bit raised?
36. When is the RXC0 flag bit raised or cleared?
37. When is the UDRE0 flag bit raised or cleared?
38. To which register do RXC0 and UDRE0 belong?
39. Find the UBRR0 for the following baud rates if XTAL = 8 MHz and U2X = 0.
    (a) `9600` (b) `19200` (c) `38400` (d) `57600`
40. Find the UBRR0 for the following baud rates if XTAL = 8 MHz and U2X = 1.
    (a) `9600` (b) `19200` (c) `38400` (d) `57600`

41. Find the UBRR0 for the following baud rates if XTAL = 11.0592 MHz and U2X = 0.
    (a) 9600 (b) 19200
    (c) 38400 (d) 57600
42. Find the UBRR0 for the following baud rates if XTAL = 11.0592 MHz and U2X = 1.
    (a) 9600 (b) 19200
    (c) 38400 (d) 57600
43. Find the baud rate error for Problem 39.
44. Find the baud rate error for Problem 40.
45. Find the baud rate error for Problem 41.
46. Find the baud rate error for Problem 42.

SECTION 11.4: AVR SERIAL PORT PROGRAMMING IN C

47. Write an AVR C program to transmit serially the letter 'Z' continuously at 9600 baud rate.
48. Write an AVR C program to transmit serially the message "The earth is but one country and mankind its citizens" continuously at 57,600 baud rate.

ANSWERS TO REVIEW QUESTIONS

SECTION 11.1: BASICS OF SERIAL COMMUNICATION

1. more expensive
2. False; it is simplex.
3. True
4. Asynchronous
5. With 0100 0101 binary the bits are transmitted in the sequence:
   (a) 0 (start bit) (b) 1 (c) 0 (d) 1 (e) 0 (f) 0 (g) 0 (h) 1 (i) 0 (j) 1 (stop bit)
6. 2 bits (one for the start bit and one for the stop bit). Therefore, for each 8-bit character, a total of 10 bits is transferred.
7. 10,000 × 10 = 100,000 total bits transmitted. 100,000 / 9600 = 10.4 seconds; 2 / 10 = 20%.
8. True
9. +3 to +25 V
10. True

SECTION 11.2: ATMEGA328 CONNECTION TO RS232

1. True
2. PD0 (RxD) and PD1 (TXD).
3. They convert different voltage levels to each other to make two different standards compatible.
4. 2, 2
5. It has a built-in capacitor.

SECTION 11.3: AVR SERIAL PORT PROGRAMMING IN ASSEMBLY

1. UBRRL and UBRRH.
2. (Fosc / 16 (baud rate)) – 1 = (10M / 16 (14400)) – 1 = 42.4 = 42 or 2AH

3. (42.4 – 42) / 42 = 0.95%
4. (Fosc / 16 (baud rate)) – 1 = (7372800 / 16 (9600)) – 1 = 47 or 2FH
5. UDR
6. USART Control Status Register A
7. UCSZ0 and UCSZ1 bits in UCSRB and UCSZ2 in UCSRC
8. False
9. Baud rate, frame size, stop bit, parity
10. U2X is bit1 of UCSRA and doubles the transfer rate for asynchronous communication.

SECTION 11.4 : AVR SERIAL PORT PROGRAMMING IN C

1. True
2. True

SECTION 11.5 : AVR SERIAL PORT PROGRAMMING IN ASSEMBLY AND C USING INTERRUPTS

1. In interrupt-based programming, CPU time is not wasted.
2. By writing the value of 1 to the UDRIE and RXCIE bits