به نام خدا

ارتباط سریال در AVR

آشنایی با ارتباط سریال

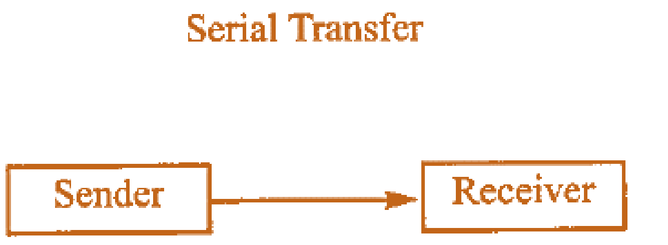
Dr. Aref Karimiafshar
A.karimiafshar@ec.iut.ac.ir

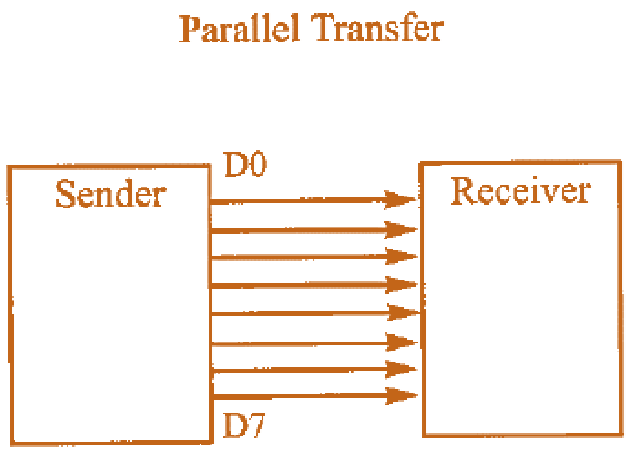
Transferring Data

- Two ways of data transferring
  - Parallel
    - Eight or more lines are used to transfer data
      - A few meters away (short distance)
        - » Printers and IDE hard disks
  - Serial
    - Send one bit at a time
      - Many meters away
        - » keyboards

Data Transferring

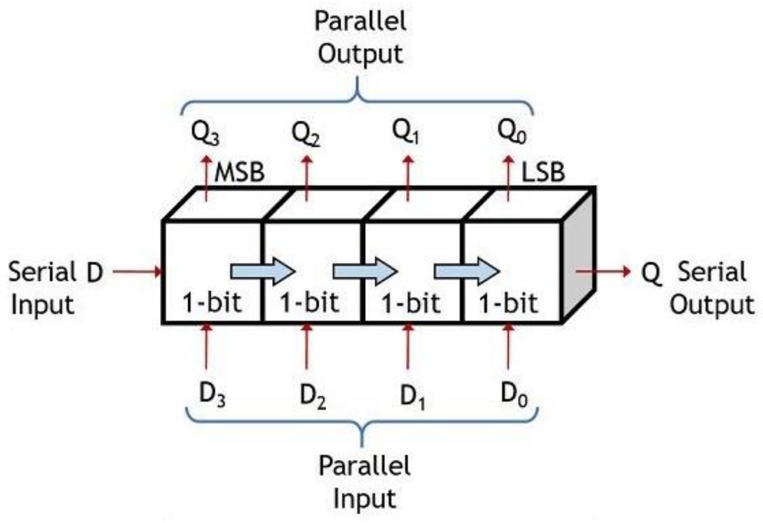
When a microprocessor communicates with the outside world, it provides the data in byte-sized chunks. For some devices, such as printers, the information is simply grabbed from the 8-bit data bus and presented to the 8-bit data bus of the device. This can work only if the cable is not too long, because long cables diminish and even distort signals. Furthermore, an 8-bit data path is expensive. For these reasons, serial communication is used for transferring data between two systems located at distances of hundreds of feet to millions of miles apart.





Serial Data Communication

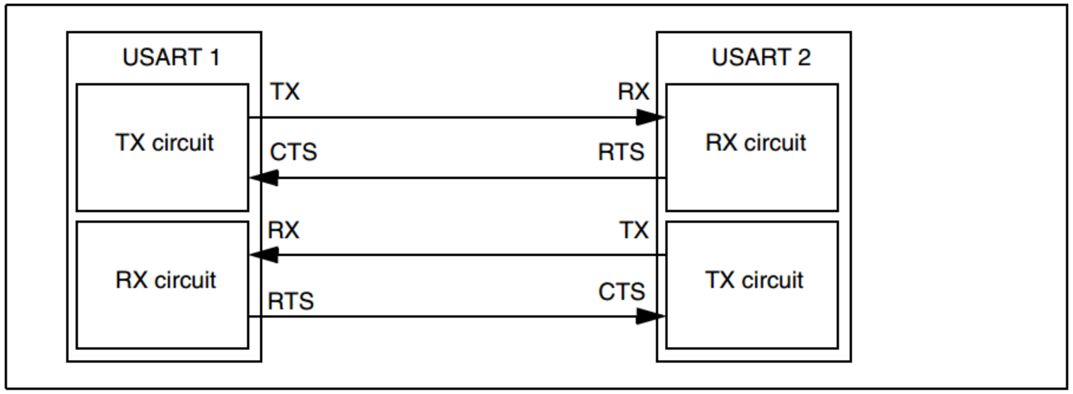
- For serial data communication to work
  - Byte of data must be converted to serial bits
    - Parallel-in-Serial-out shift register
      - Then it can be transmitted over a single data line
  - At the receiving end
    - Serial-in-Parallel-out shift register
      - Then pack them into a byte



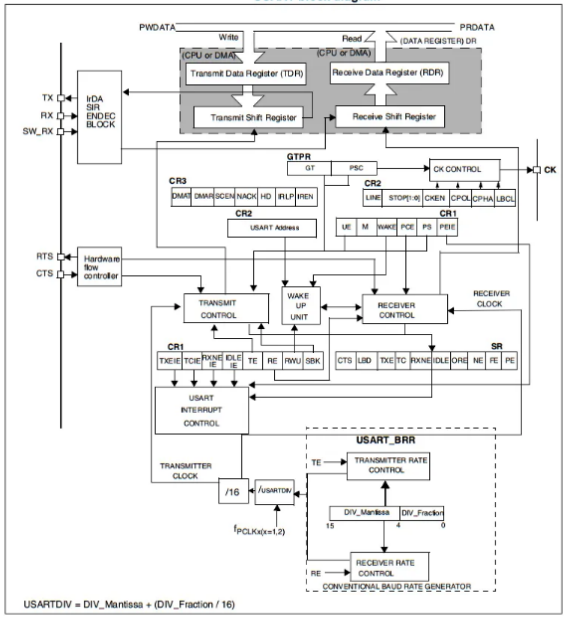
Methods Serial Data Communication

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.

Hardware flow control between two USARTs

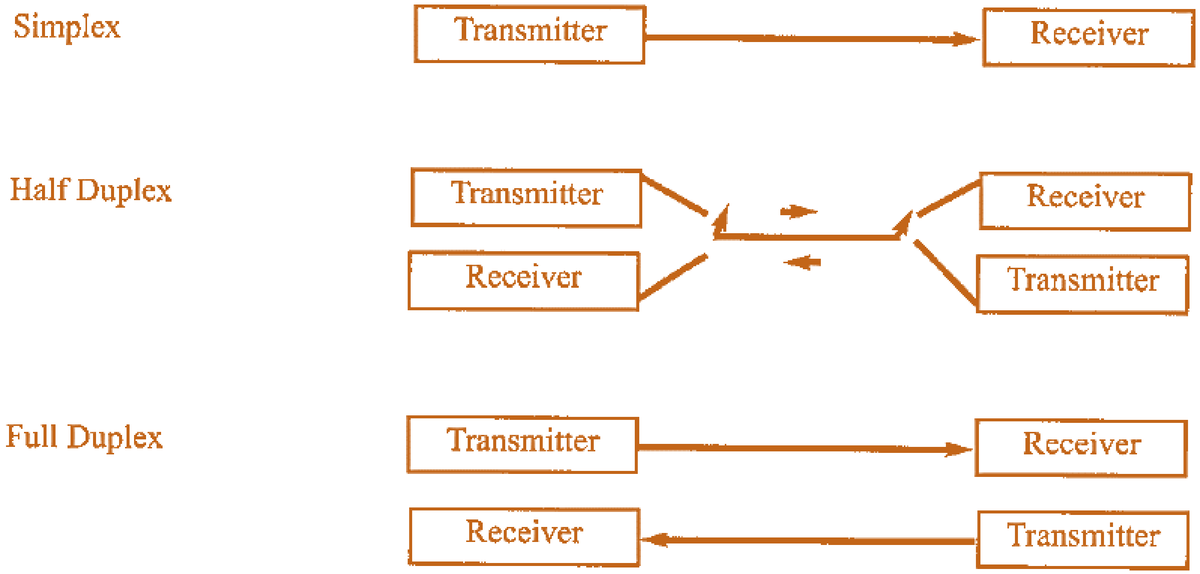


USART block diagram



Methods Serial Data Communication

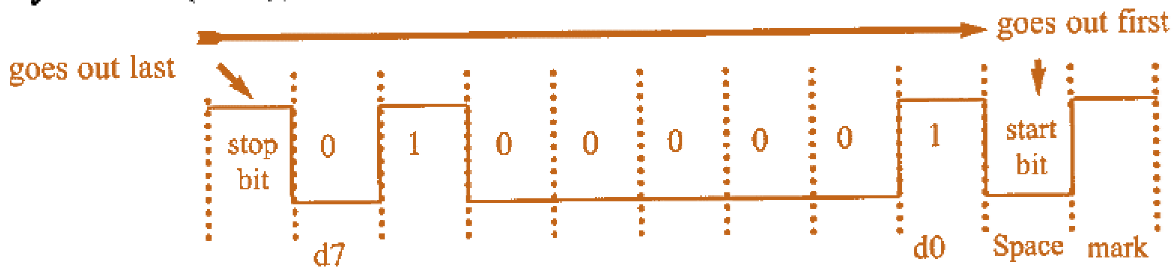
In data transmission, if the data can be both transmitted and received, it is a *duplex* transmission. This is in contrast to *simplex* transmissions such as with printers, in which the computer only sends data. Duplex transmissions can be half or full duplex, depending on whether or not the data transfer 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.



Asynchronous Serial Data Communication

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.

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).



Notice in the Figure 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”.

Asynchronous Serial Data Communication

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 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.

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.

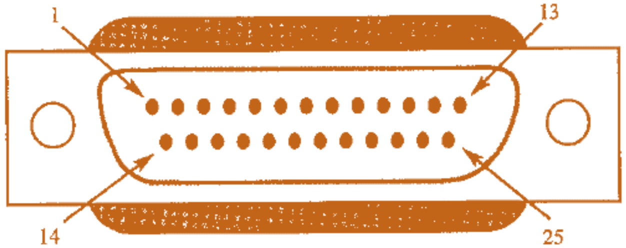
RS232 Standards

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.

RS232 is 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.

RS232 Pins

The Table shows the pins for the original RS232 cable and their labels, commonly referred to as the DB-25 connector. In labeling, DB-25P refers to the plug connector (male), and DB-25S is for the socket connector (female).

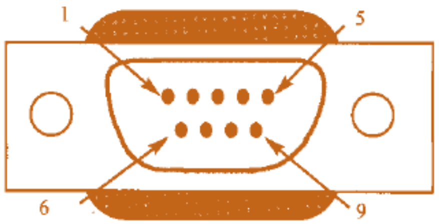


The Original RS232 Connector DB-25 (No longer in use)

Because not all the pins were used in PC cables, IBM introduced the DB-9 version of the serial I/O standard, which uses only 9 pins, as shown in

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)



9-Pin Connector for DB-9

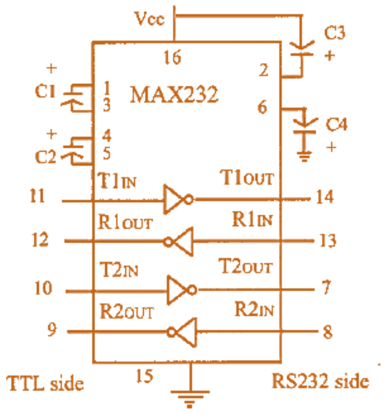
RS232 Pins (DB-25)

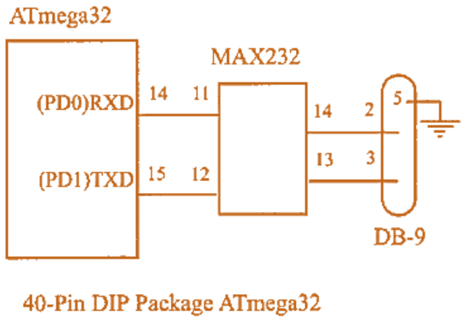
Pin	Description
1	Protective ground
2	Transmitted data (TxD)
3	Received data (RxD)
4	Request to send (RTS)
5	Clear to send (CTS)
6	Data set ready (DSR)
7	Signal ground (GND)
8	Data carrier detect (DCD)
9/10	Reserved for data testing
11	Unassigned
12	Secondary data carrier detect
13	Secondary clear to send
14	Secondary transmitted data
15	Transmit signal element timing
16	Secondary received data
17	Receive signal element timing
18	Unassigned
19	Secondary request to send
20	Data terminal ready (DTR)
21	Signal quality detector
22	Ring indicator
23	Data signal rate select
24	Transmit signal element timing
25	Unassigned

X86 PC COM Ports

The x86 PCs (based on 8086, 286, 386, 486, and all 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, one of these has been replaced with the USB port, and COM 1 is the only serial port available, if any. We can connect the AVR serial port to the COM 1 port of a PC for serial communication experiments. In the absence of a COM port, we can use a COM-to-USB converter module.

The ATmega32 has two pins that are used specifically for transferring and receiving data serially. These two pins are called TX and RX and are part of the Port D group (PD0 and PD1) of the 40-pin package. Pin 15 of the ATmega32 is assigned to TX and pin 14 is designated as RX. These pins are TTL compatible; therefore, they require a line driver to make them RS232 compatible. One such line driver is the MAX232 chip.

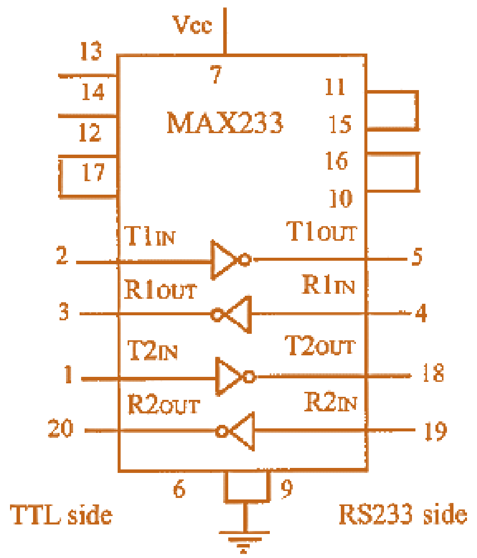


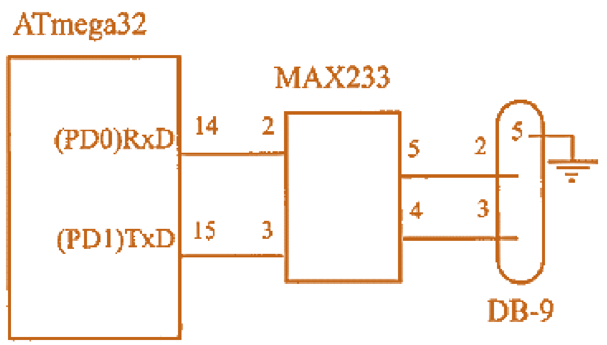


Line Driver (MAX232, MAX233)

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

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 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.





40-Pin DIP Package ATmega32

AVR Serial Port Programming

The USART (universal synchronous asynchronous receiver/ transmitter) in the AVR has normal asynchronous, double-speed asynchronous, master synchronous, and slave synchronous mode features.

In he AVR microcontroller five registers are associated with the USART that we deal with in this part. They are UDR (USART Data Register), UCSRA, UCSRB, UCSRC (USART Control Status Register), and UBRR (USART Baud Rate Register).

UBRR Register and Baud Rate in the AVR

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 UBRR and Fosc is dictated by the following formula:

UBRR Values for Various Baud Rates

Baud Rate	UBRR (Decimal Value)
38400	12
19200	25
9600	51
4800	103
2400	207
1200	415

Note: For Fosc = 8 MHz

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

Assuming that Fosc = 8 MHz, we have the following:

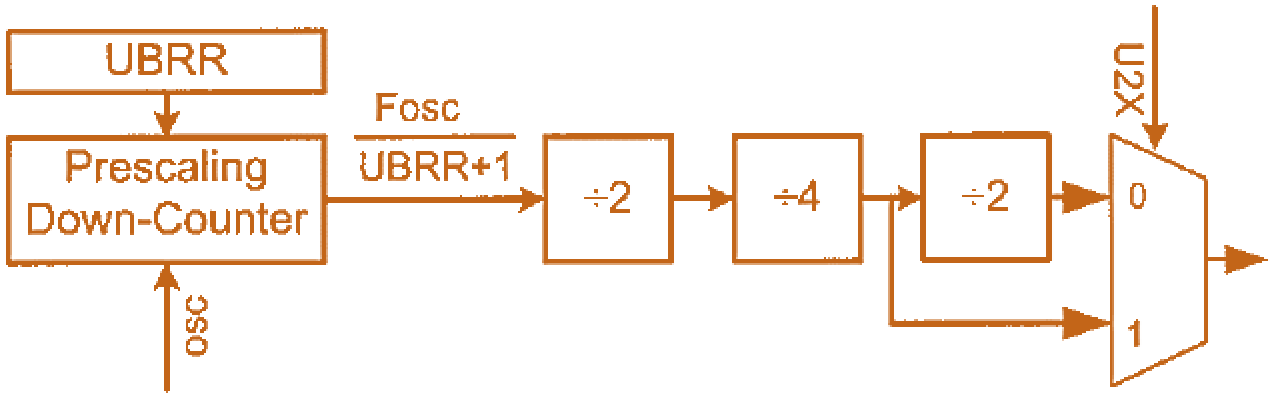
Desired Baud Rate = Fosc/ (16(X + 1)) = 8 MHz/16(X + 1) = 500 kHz/(X + 1)

X = (500 kHz/ Desired Baud Rate) – 1

UBRR Register and Baud Rate in the AVR

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.

we can choose to bypass the last divider and double the baud rate.



Example

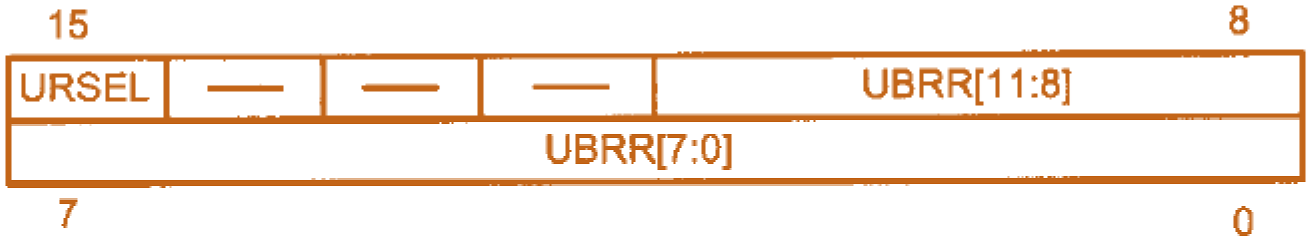
With Fosc = 8 MHz, find the UBRR value needed to have the following baud rates:
(a) 9600 (b) 4800 (c) 2400 (d) 1200

Fosc = 8 MHz => X = (8 MHz/16(Desired Baud Rate)) – 1
=> X = (500 kHz/(Desired Baud Rate)) – 1

(a) (500 kHz/ 9600) – 1 = 52.08 – 1 = 51.08 = 51 = 33 (hex) is loaded into UBRR
(b) (500 kHz/ 4800) – 1 = 104.16 – 1 = 103.16 = 103 = 67 (hex) is loaded into UBRR
(c) (500 kHz/ 2400) – 1 = 208.33 – 1 = 207.33 = 207 = CF (hex) is loaded into UBRR
(d) (500 kHz/ 1200) – 1 = 416.66 – 1 = 415.66 = 415 = 19F (hex) is loaded into UBRR

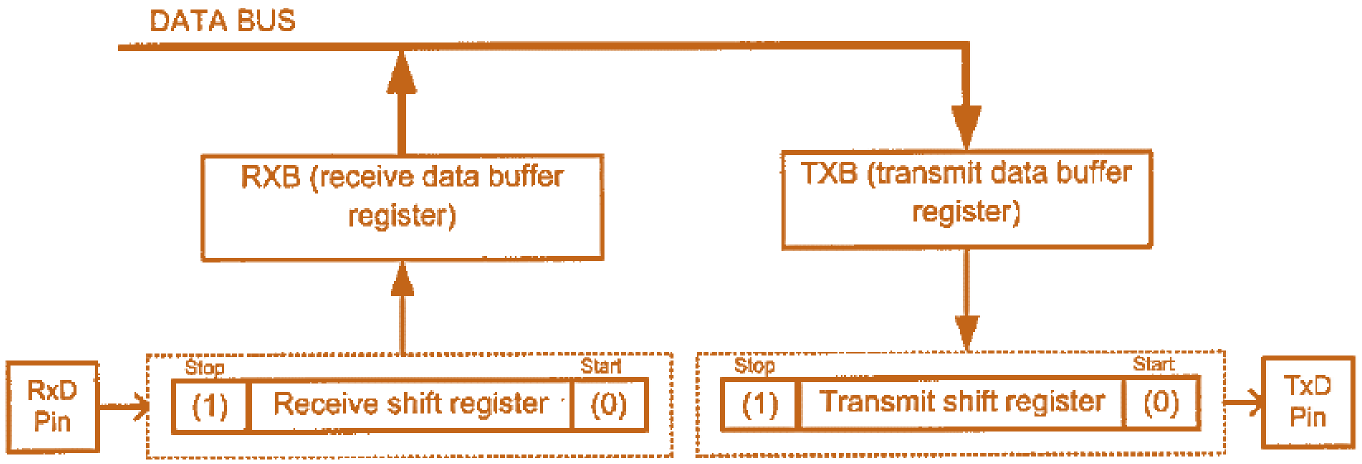
UBRR Register

As you see in the Figure , UBRR is a 16-bit register but only 12 bits of it are used to set the USART baud rate. Bit 15 is URSEL and, as we will see in the next section, selects between accessing the UBRRH or the UCSRC register. The other bits are reserved.



UDR Registers and USART Data IO 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).



UCSR Registers and USART Configuration in the AVR

- UCSRs (USART Control Status Registers)
  - are 8-bit control registers
    - UCSRA
    - UCSRB
    - UCSRC
  - Used for controlling serial communication in AVR

UCSRA

RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM

RXC (Bit 7): USART Receive Complete
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.

TXC (Bit 6): USART Transmit Complete
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.

UDRE (Bit 5): USART Data Register Empty
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 UDR because it overrides your last data. The UDRE flag can generate a data register empty interrupt.

FE (Bit 4): Frame Error
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.

DOR (Bit 3): Data OverRun
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.

PE (Bit 2): Parity Error
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.

U2X (Bit 1): Double the USART Transmission Speed
Setting this bit will double the transfer rate for asynchronous communication.

MPCM (Bit 0): Multi-processor Communication Mode
This bit enables the multi-processor communication mode.

Notice that FE, DOR, and PE are valid until the receive buffer (UDR) is read. Always set these bits to zero when writing to UCSRA.

UCSRB

RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8

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

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

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

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

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

UCSZ2 (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.

RXB8 (Bit 1): Receive data bit 8
This is the ninth data bit of the received character when using serial frames with nine data bits.

TXB8 (Bit 0): Transmit data bit 8
This is the ninth data bit of the transmitted character when using serial frames with nine data bits.

UCSRC

URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL

URSEL (Bit 7): Register Select
This bit selects to access either the UCSRC or the UBRRH register

UMSEL (Bit 6): USART Mode Select
This bit selects to operate in either the asynchronous or synchronous mode of operation.
0 = Asynchronous operation
1 = Synchronous operation

UPM1:0 (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

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

UCSZ1:0 (Bit 2:1): Character Size
These bits combined with the UCSZ2 bit in UCSRB set the character size in a frame

UCPOL (Bit 0): Clock Polarity
This bit is used for synchronous mode only

Character Size

To set the number of data bits (character size) in a frame you must set the values of the UCSZ1 and USCZ0 bits in the UCSRB and UCSZ2 bits in UCSRC. The Table shows the values of UCSZ2, UCSZ1, and UCSZ0 for different character sizes.

Values of UCSZ2:0 for Different Character Sizes

UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5
0	0	1	6
0	1	0	7
0	1	1	8
1	1	1	9

Because of some technical considerations, the UCSRC register shares the same I/O location as the UBRRH, and therefore some care must be taken when accessing these I/O locations. When you write to UCSRC or UBRRH, the high bit of the written value (URSEL) controls which of the two registers will be the target of the write operation. If URSEL is zero during a write operation, the UBRRH value will be updated; otherwise, UCSRC will be updated.

Example

(a) What are the values of UCSRB and UCSRC 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 UCSRB and UCSRC for this configuration.

(a) RXEN and TXEN have to be 1 to enable receive and transmit. UCSZ2:0 should be 011 for 8-bit data, UMSEL should be 0 for asynchronous operating mode, UPM1:0 have to be 00 for no parity, and USBS should be 0 for one stop bit.

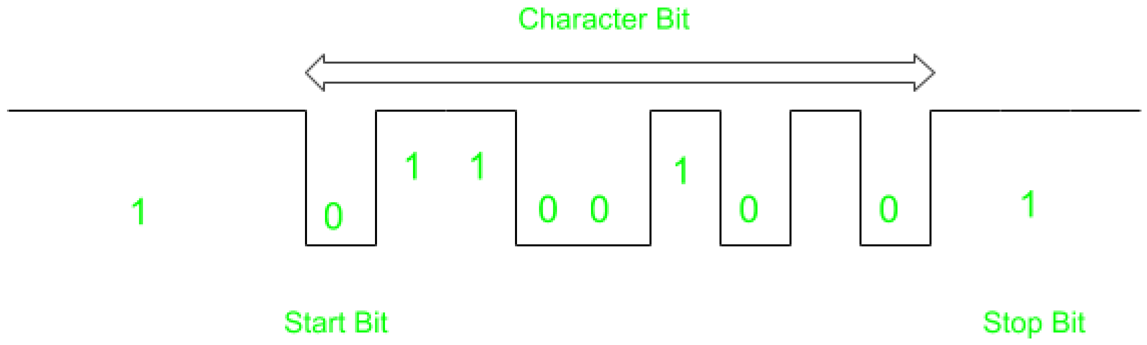
(b)

```
        .INCLUDE "M32DEF.INC"

        LDI   R16,(1<<RXEN)|(1<<TXEN)
        OUT   UCSRB, R16
;In the next line URSEL = 1 to access UCSRC. Note that instead
;of using shift operator, you can write "LDI R16, 0b10000110"
        LDI   R16,(1<<UCSZ1)|(1<<UCSZ0)|(1<<URSEL)
        OUT   UCSRC, R16
```

FE and PE 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 PE to one, indicating that an 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 FE 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 FE and PE are valid until the receive buffer (UDR) is read. So we have to read FE and PE bits before reading UDR.



Programming the AVR to transfer data Serially

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

1. The UCSRB 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 UCSRC register is loaded with the value 06H, indicating asynchronous mode with 8-bit data frame, no parity, and one stop bit.
3. The UBRR is loaded with one of the values in the Table (if Fosc = 8 MHz) to set the baud rate for serial data transfer.
4. The character byte to be transmitted serially is written into the UDR register.
5. Monitor the UDRE bit of the UCSRA register to make sure UDR is ready for the next byte.
6. To transmit the next character, go to Step 4.

Example

Write a program for the AVR to transfer the letter 'G' serially at 9600 baud, continuously. Assume XTAL = 8 MHz.

```
 .INCLUDE "M32DEF.INC"
      LDI   R16,(1<<TXEN)                    ;enable transmitter
      OUT   UCSRB, R16
      LDI   R16,(1<<UCSZ1)|(1<<UCSZ0)|(1<<URSEL);8-bit data
      OUT   UCSRC, R16                       ;no parity, 1 stop bit
      LDI   R16,0x33                         ;9600 baud rate
      OUT   UBRRL,R16                        ;for XTAL = 8 MHz
AGAIN:
      SBIS  UCSRA,UDRE                       ;is UDR empty
      RJMP  AGAIN                            ;wait more
      LDI   R16,'G'                          ;send 'G'
      OUT   UDR,R16                          ;to UDR
      RJMP  AGAIN                            ;do it again
```

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

Example

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

```
        .INCLUDE "M32DEF.INC"

        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<<TXEN)                ;enable transmitter
        OUT    UCSRB, R16
        LDI    R16,(1<<UCSZ1)|(1<<UCSZ0)|(1<<URSEL); 8-bit data
        OUT    UCSRC, R16                   ;no parity, 1 stop bit
        LDI    R16,0x33                     ;9600 baud rate
        OUT    UBRRL,R16
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:
        SBIS   UCSRA,UDRE                   ;is UDR empty?
        RJMP   TRNSMT                       ;wait more
        OUT    UDR,R17                      ;send R17 to UDR
        RET
```

Programming the AVR to receive data Serially

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

1. The UCSRB 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 UCSRC register is loaded with the value 06H, indicating asynchronous mode with 8-bit data frame, no parity, and one stop bit.
3. The UBRR is loaded with one of the values in the Table (if Fosc = 8 MHz) to set the baud rate for serial data transfer.
4. The RXC flag bit of the UCSRA register is monitored for a HIGH to see if an entire character has been received yet.
5. When RXC is raised, the UDR register has the byte. Its contents are moved into a safe place.
6. To receive the next character, go to Step 5.

Example

Program the ATmega32 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.

```
.INCLUDE "M32DEF.INC"
      LDI   R16,(1<<RXEN)                     ;enable receiver
      OUT   UCSRB, R16
      LDI   R16,(1<<UCSZ1)|(1<<UCSZ0)|(1<<URSEL);8-bit data
      OUT   UCSRC, R16                        ;no parity, 1 stop bit
      LDI   R16,0x33                          ;9600 baud rate
      OUT   UBRRL,R16
      LDI   R16,0xFF                          ;Port B is output
      OUT   DDRB,R16
RCVE:
      SBIS  UCSRA,RXC                         ;is any byte in UDR?
      RJMP  RCVE                              ;wait more
      IN    R17,UDR                           ;send UDR to R17
      OUT   PORTB,R17                         ;send R17 to PORTB
      RJMP  RCVE                              ;do it again
```

Example

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 A and transmit it via the serial port to the PC's screen, and (c) receive any key press sent by HyperTerminal and put it on LEDs. The programs must do parts (b) and (c) repeatedly.

```
        LDI   R21,0x00
        OUT   DDRA,R21                   ;Port A is input
        LDI   R21,0xFF
        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<<TXEN)|(1<<RXEN)    ;enable transmitter
        OUT   UCSRB, R16                 ;and receiver
        LDI   R16,(1<<UCSZ1)|(1<<UCSZ0)|(1<<URSEL) ;8-bit data
        OUT   UCSRC, R16                 ;no parity, 1 stop bit
        LDI   R16,0x33                   ;9600 baud rate
        OUT   UBRRL,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:
        SBIS  UCSRA,RXC                  ;is there new data?
        RJMP  SKIP_RX                    ;skip receive cmnds
        IN    R17,UDR                    ;move UDR to R17
        OUT   PORTB,R17                  ;move R17 TO PORTB
SKIP_RX:
        SBIS  UCSRA,UDRE                 ;is UDR empty?
        RJMP  SKIP_TX                    ;skip transmit cmnds
        IN    R17,PINA                   ;move Port A to R17
        OUT   UDR,R17                    ;send R17 to UDR
SKIP_TX:
        RJMP  AGAIN                      ;do it again
TRNSMT:
        SBIS  UCSRA,UDRE                 ;is UDR empty?
        RJMP  TRNSMT                     ;wait more
        OUT   UDR,R17                    ;send R17 to UDR
        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 UCSRA 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 U2X bit of the UCSRA register. When the AVR is powered up, the U2X bit of the UCSRA register is zero. We can set it to high by software and thereby double the baud rate. If we set the U2X bit to HIGH, the third frequency divider will be bypassed. In the case of XTAL = 8 MHz and U2X bit set to HIGH, we would have:

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

UBRR Values for Various Baud Rates (XTAL = 8 MHz)

	U2X = 0		U2X = 1	
Baud Rate	**UBRR**	**UBR (HEX)**	**UBRR**	**UBR (HEX)**
38400	12	C	25	19
19200	25	19	51	33
9,600	51	33	103	67
4,800	103	67	207	CF
UBRR = (500 kHz / Baud rate) – 1			*UBRR = (1 kHz / 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 = 8 MHz and U2X = 0 we have the following for the 9600 baud rate:

UBRR value = (500,000/ 9600) – 1 = 52.08 – 1 = 51.08 = 51

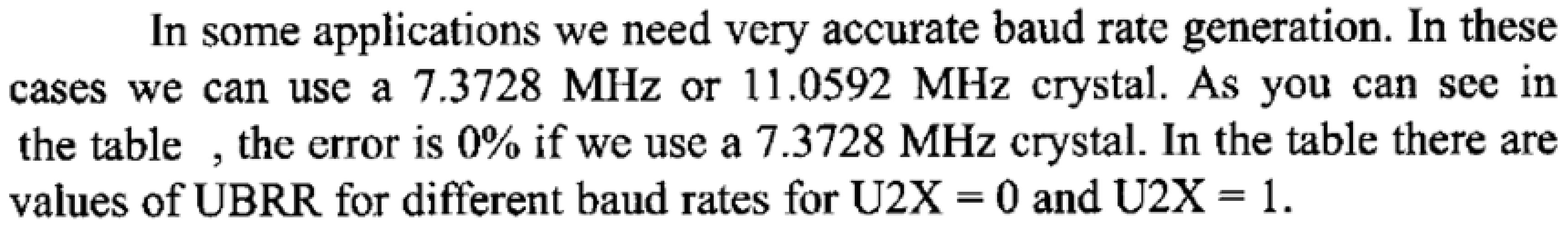
Error = (51.08 – 51)/ 51 = 0.16%

UBRR Values for Various Baud Rates (XTAL = 8 MHz)

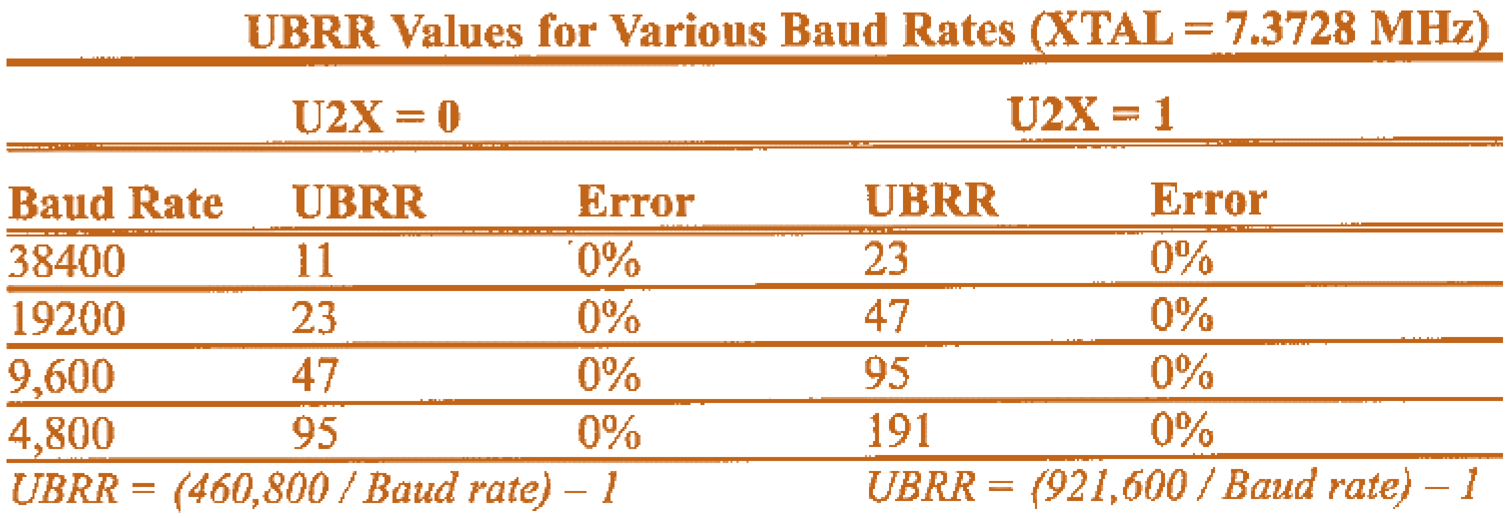
	U2X = 0		U2X = 1	
Baud Rate	**UBRR**	**Error**	**UBRR**	**Error**
38400	12	0.2%	25	0.2%
19200	25	0.2%	51	0.2%
9,600	51	0.2%	103	0.2%
4,800	103	0.2%	207	0.2%

UBRR = (500,000 / Baud rate) – 1   *UBRR = (1,000,000 / Baud rate) – 1*

Baud Rate Error Calculation



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 the table , the error is 0% if we use a 7.3728 MHz crystal. In the table there are values of UBRR for different baud rates for U2X = 0 and U2X = 1.



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*

AVR Serial Port Programming in C

all the special function registers of the AVR are accessible directly in C compilers by using the appropriate header file.

Example

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

```
void usart_init (void)
{
   UCSRB = (1<<TXEN);
   UCSRC = (1<< UCSZ1)|(1<<UCSZ0)|(1<<URSEL);
   UBRRL = 0x33;
}
```

Example

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 = 8 MHz.

```
#include <avr/io.h>                               //standard AVR header
void usart_init (void)
{
  UCSRB = (1<<TXEN);
  UCSRC = (1<< UCSZ1)|(1<<UCSZ0)|(1<<URSEL);
  UBRRL = 0x33;
}
void usart_send (unsigned char ch)
{                                                 //wait until UDR
  while (! (UCSRA & (1<<UDRE)));                  //is empty
  UDR = ch;                                       //transmit 'G'
}

int main (void)
{
  usart_init();                                   //initialize the USART
  while(1)                                        //do forever
    usart_send ('G');                             //transmit 'G' letter
  return 0;
}
```

Example

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.

```
#include <avr/io.h>                    //standard AVR header

void usart_init (void)
{  UCSRB  =  (1<<TXEN);
   UCSRC  =  (1<< UCSZ1)|(1<<UCSZ0)|(1<<URSEL);
   UBRRL  =  0x33;
}
void usart_send (unsigned char ch)
{  while (! (UCSRA & (1<<UDRE)));
   UDR = 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;
   }
   return 0;
}
```

AVR Serial Port Programming using Interrupts

- Polling TXIF and RXIF is Waste of ‘s µC time
  - Instead, we can use interrupts

- Using interrupts
  - Interrupt Enable bit(s) become HIGH and Force the CPU to Jump to the interrupt service routine
    - Upon completion of receive
    - When UDR is ready to accept new data

Example (interrupt-based data receive)

Program the ATmega32 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.

```
.INCLUDE "M32DEF.INC"
.CSEG                                   ;put in code segment
        RJMP  MAIN                      ;jump main after reset
.ORG  URXCaddr                          ;int-vector of URXC int.
        RJMP  URXC_INT_HANDLER          ;jump to URXC_INT_HANDLER
.ORG  40                                ;start main after
                                        ;interrupt vector

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<<RXEN)|(1<<RXCIE)    ;enable receiver
      OUT   UCSRB, R16                  ;and RXC interrupt
      LDI   R16,(1<<UCSZ1)|(1<<UCSZ0)|(1<<URSEL);sync,8-bit data
      OUT   UCSRC, R16                  ;no parity, 1 stop bit
      LDI   R16,0x33                    ;9600 baud rate
      OUT   UBRRL,R16
      LDI   R16,0xFF                    ;set Port B as an
      OUT   DDRB,R16                    ;input
      SEI                               ;enable interrupts
WAIT_HERE:
      RJMP  WAIT_HERE                   ;stay here
URXC_INT_HANDLER:
      IN    R17,UDR                     ;send UDR to R17
      OUT   PORTB,R17                   ;send R17 to PORTB
      RETI
```

Example (interrupt-based data transmit)

Write a program for the AVR to transmit the letter 'G' serially at 9600 baud, continuously. Assume XTAL = 8 MHz. Use interrupts instead of the polling method.

```
.INCLUDE "M32DEF.INC"

.CSEG                                   ;put in code segment
      RJMP  MAIN                        ;jump main after reset
.ORG  UDREaddr                          ;int. vector of UDRE int.
      RJMP  UDRE_INT_HANDLER            ;jump to UDRE_INT_HANDLER
.ORG  40                                ;start main after
                                        ;interrupt vector
;*****************************
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<<TXEN)|(1<<UDRIE)    ;enable transmitter
      OUT   UCSRB, R16                  ;and UDRE interrupt
      LDI   R16,(1<<UCSZ1)|(1<<UCSZ0)|(1<<URSEL); sync., 8-bit
      OUT   UCSRC, R16              ;data no parity, 1 stop bit
      LDI   R16,0x33                    ;9600 baud rate
      OUT   UBRRL,R16
      SEI                               ;enable interrupts
WAIT_HERE:
      RJMP  WAIT_HERE                   ;stay here
;*****************************
UDRE_INT_HANDLER:
      LDI   R26,'G'                     ;send 'G'
      OUT   UDR,R26                     ;to UDR
      RETI
```

Example (Programming in C)

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.

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

ISR(USART_RXC_vect)
{
   PORTB = UDR;
}

int main (void)
{
   DDRB = 0xFF;                           //make Port B an output
   UCSRB = (1<<RXEN)|(1<<RXCIE);  //enable receive and RXC int.
   UCSRC = (1<< UCSZ1)|(1<<UCSZ0)|(1<<URSEL);
   UBRRL = 0x33;
   sei();                                 //enable interrupts
   while (1);                             //wait forever
   return 0;
}
```

Example (Programming in C)

Write a C program to transmit the letter 'G' serially at 9600 baud, continuously. Assume XTAL = 8 MHz. Use interrupts instead of the polling method.

```
#include <avr\io.h>
#include <avr\interrupt.h>
ISR(USART_UDRE_vect)
{
    UDR = 'G';
}

int main (void)
{
   UCSRB = (1<<TXEN)|(1<<UDRIE);
   UCSRC = (1<< UCSZ1)|(1<<UCSZ0)|(1<<URSEL);
   UBRRL = 0x33;

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

پایان

موفق و پیروز باشید