# Embedded System on AVR Microcontroller (ATMEGA32)

## Exp9A: UART Communication on ATMEGA32

Submitted by
Ronit Dutta, MS in IOT and Signal Processing
Department of Electrical Engineering, IIT Kharagpur

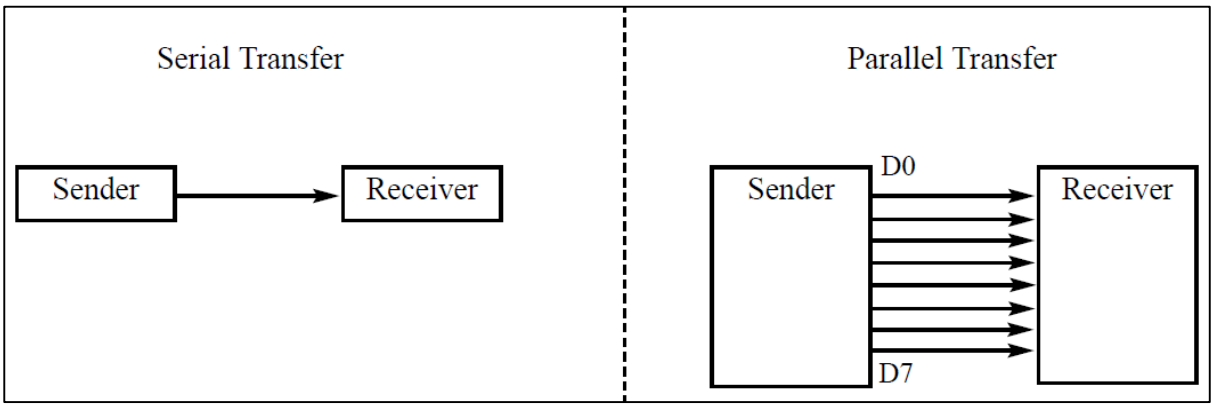
*Under the guidance of*
Aurobinda Routray, Professor, Electrical Engineering


Department of Electrical Engineering
Indian Institute of Technology Kharagpur
March, 2025

# • Introduction to Parallel and Serial Communication

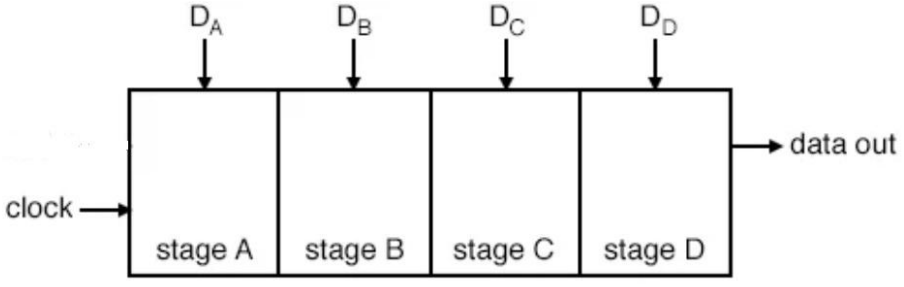
Computers transfer data in two ways: parallel and serial. In parallel data transfers, often eight or more lines (wire conductors) are used to transfer data to a device that is only a few feet away. Devices that use parallel transfers include monitors and hard disks; each uses cables with many wires. 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. The AVR has serial communication capability built into it, thereby making possible fast data transfer using only a few wires. Serial communication of the AVR will be discussed in this module.

The parallel communication 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.




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.




Parallel-in serial-out shift register with 4-stages

Fig. Parallel to Serial Data Conversion at Transmitter

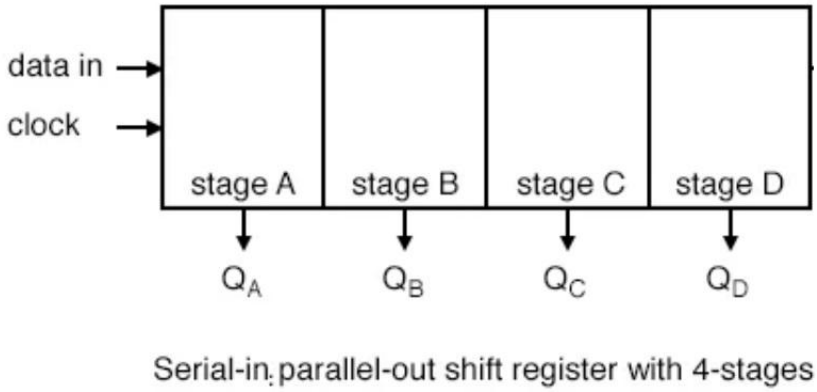



Fig. Serial to Parallel Data Conversion at Receiver

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. 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) for long-distance data transfers using communication lines such as a telephone.

## • Asynchronous and Synchronous Serial Communication

Serial data communication uses two methods, asynchronous and synchronous.

1. Asynchronous data communication relies on individual data packets sent independently, without a shared clock signal between sender and receiver.
2. Synchronous data communication synchronizes transmission with a common clock signal, ensuring data is sent and received in coordinated intervals.
3. Asynchronous communication is typically simpler and more flexible, suitable for varying data rates and distances, but may introduce overhead due to start and stop bits.
4. Synchronous communication offers higher efficiency and reliability for continuous data streams, but requires precise timing alignment between sender and receiver.

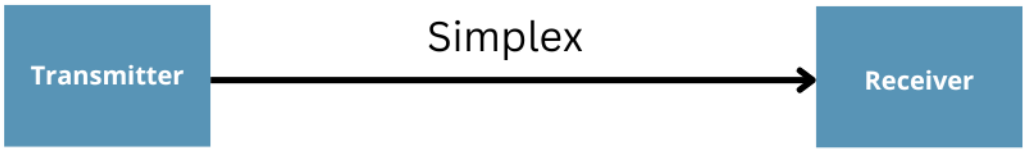
**The AVR chip has a built-in USART (Universal Synchronous Asynchronous Receiver and Transmitter).**

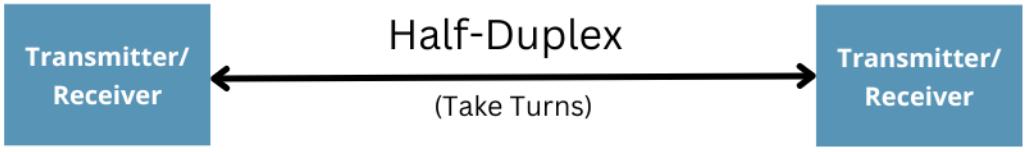
## • Simplex, Half-Duplex and Full-Duplex transmission

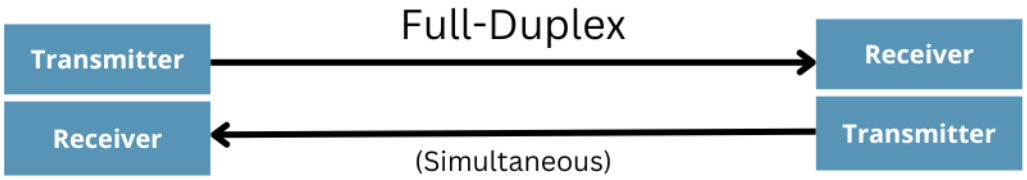
In data transmission, if the data can be both transmitted and received, it is a duplex transmission. This is in contrast to simplex transmissions such as with printers, in which the computer only sends data. Duplex transmissions can be half or full duplex, depending on whether or not the data transfer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

**Examples of Simplex Transmission:** Television, Radio broadcasts, Monitors etc.

**Examples of Half-Duplex Transmission:** Walkie-talkie
**Examples of Full-Duplex Transmission:** Telephone conversation, Internet video conferencing, Ethernet networks etc.




- # Asynchronous Serial Communication and Data Framing

The data coming at receiver from transmitter through serial data transfer is all 0s and 1s, hence 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**

  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).
  **Example:** The ASCII character "A" (8-bit binary 0100 0001 and hex 0x41) is framed between the start bit and a single stop bit. Notice that the LSB is sent out first.
  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".

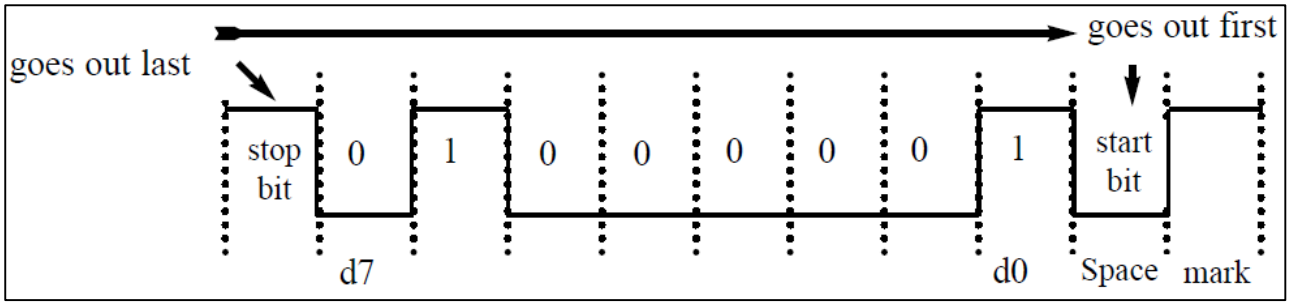
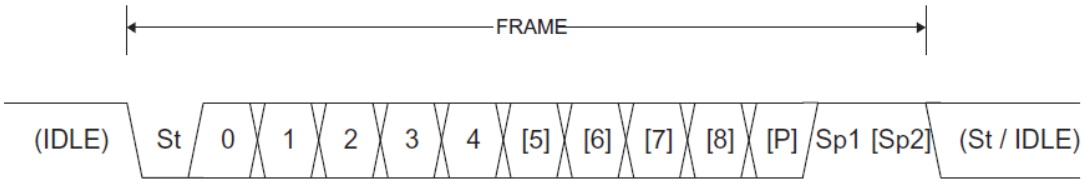



Fig. UART Transmission of ASCII 'A'

In asynchronous serial communications, ATMEGA32 can be programmed for data that is 5, 6, 7, 8 or 9 bits wide. This is in addition to the number of stop bits, 1 or 2.

The parity bit of the character byte can be included in the data frame in order to maintain data integrity. The parity bit can be 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 of ATMEGA32 allows programming of the parity bit for odd, even, and no-parity options.




| | |
|---|---|
| **St** | Start bit, always low. |
| **(n)** | Data bits (0 to 8). |
| **P** | Parity bit. Can be odd or even. |
| **Sp** | Stop bit, always high. |
| **IDLE** | No transfers on the communication line (RxD or TxD). An IDLE line must be high. |

- ## 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 text 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 ATMEGA32 can transfer data at the rate of 9600 bps (depending upon UBRR register settings). Normally 9600 bps is used where speed is not a critical issue.

- # Hardware and Software Requirements

Normally in USART, Tx (Transmitter), Rx(Receiver), and GND wires are required.

- AVR ATMEGA32 USART has a TTL voltage level which is 0 v for logic 0 and 5 v for logic 1.
- In computers and most of the old devices, RS232 protocol is used for serial communication, where normally 9 pin 'D' shape connector is used. RS232 serial communication has different voltage levels than ATMEGA32 serial communication i.e. +3 v to +25 v for logic zero and -3 v to -25 v for logic 1.
- So to communicate with RS232 protocol, a voltage level converter like MAX232 IC is used.
- With a new PC and laptops, there is no RS232 protocol and DB9 connector. Then USB to UART TTL converter is used. There is various USB to UART TTL converter available e.g. CP2102, FT232RL, CH340 etc.

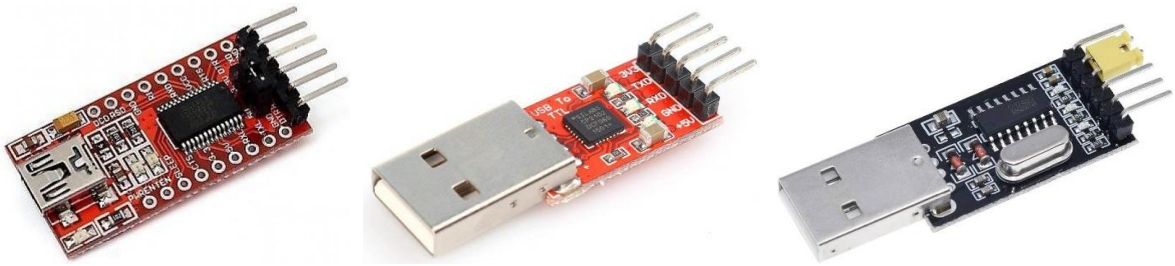
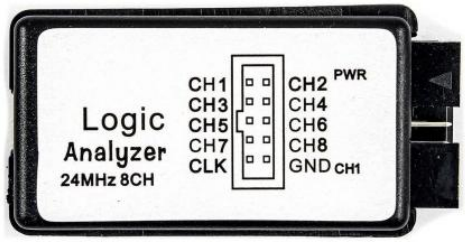
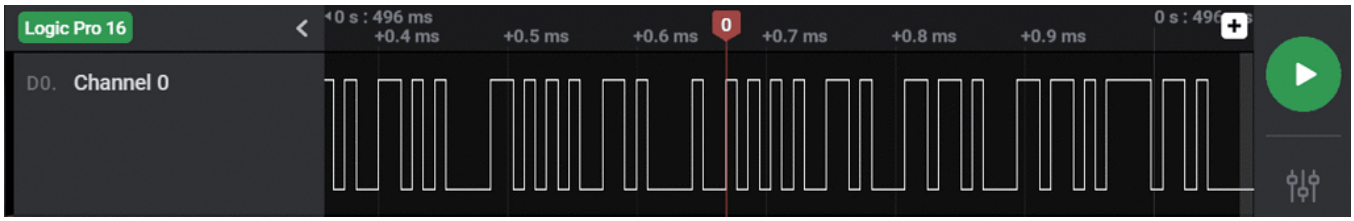


Fig. (a) FTDI FT232RL, (b) Silicon Lab CP2102, (c) Win Chip Head (WCH) CH340

- To see serial communication, a serial terminal like Realterm will be used and to see serial plot, a serial plotter will be used. By selecting the serial port number (COM port in windows) and baud rate, the serial communication will be established.
- A Logic Analyzer is used to check the UART data frame, allowing for precise analysis of signal transmission. **Saleae** driver software is utilized for this purpose, providing a user-friendly interface to capture and interpret data. After installing the software, the application icon will appear as **"Logic"**, enabling quick access to the tool for efficient debugging and signal verification.

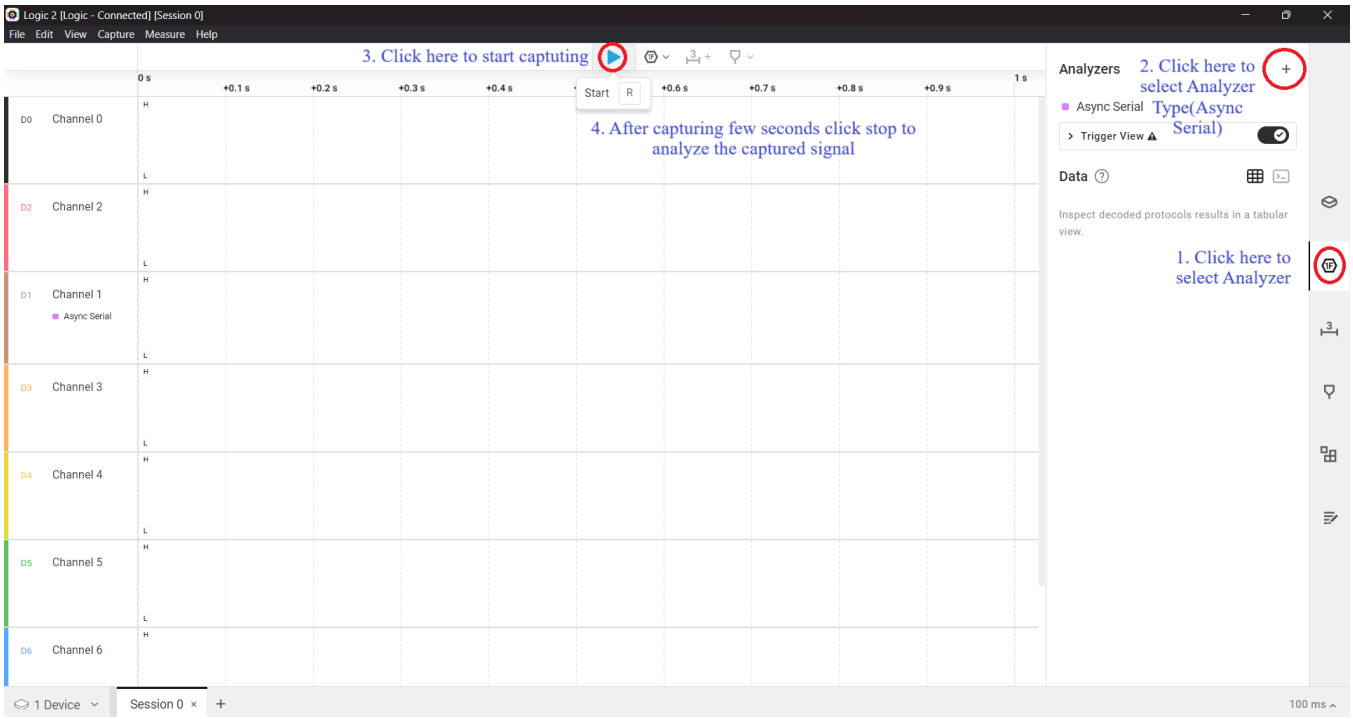



The Logic Analyzer view will appear as shown below:

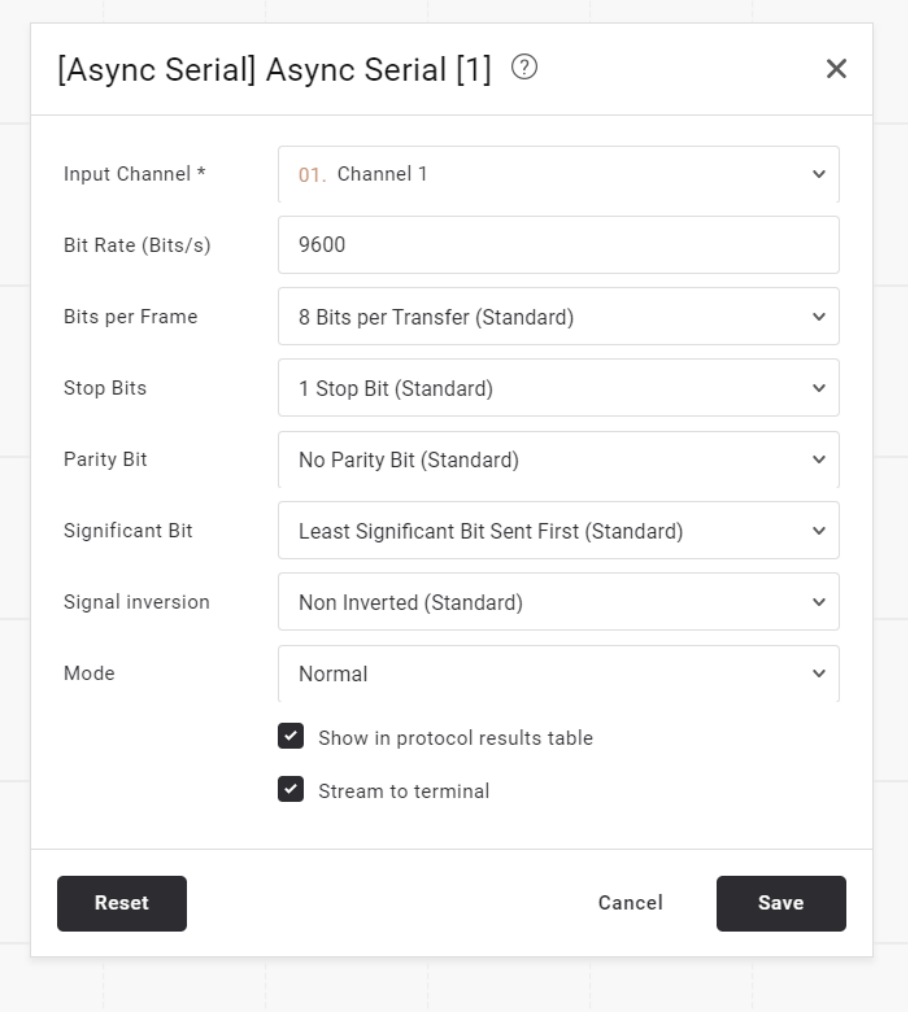


- The necessary software link for the experiment: [Exp9A UART Software]

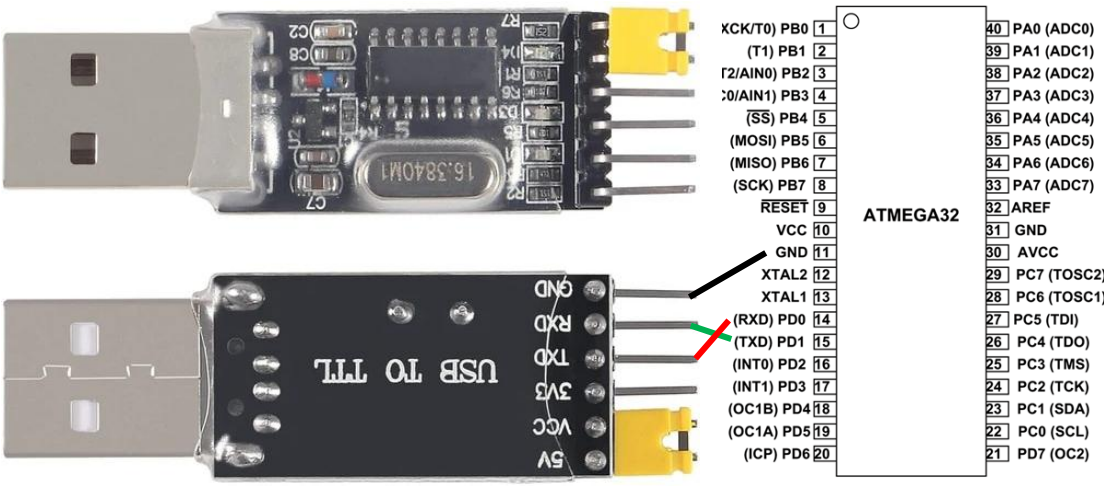
**Settings of Logic Analyzer:**




Choose the appropriate settings when selecting the Analyzer Type.



- # ATMEGA32 connection to CH340




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.

- # USART Registers

In this section the serial communication registers of the ATmega32 and how to program them to transfer & receive data using asynchronous mode will be discussed. 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 used to connect the AVR-based system to the x86 PC serial port, sensors for the purpose of full-duplex serial data transfer. In this section the asynchronous mode will be examined only.

In the AVR microcontroller five registers are associated with the USART. They are **UDR (USART Data Register), UCSRA, UCSRB, UCSRC (USART Control Status Register)** and **UBRR (USART Baud Rate Register)**.

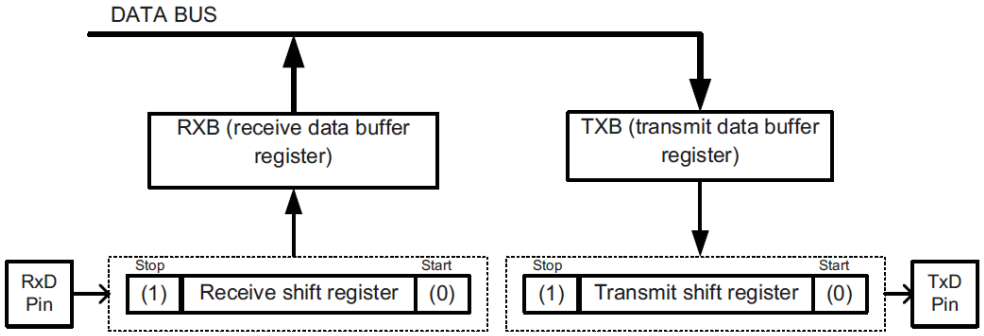
### i. UDR (USART Data Register)

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | RXB[7:0] | | | | | | | | UDR (Read) |
| | TXB[7:0] | | | | | | | | UDR (Write) |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

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 (TXB) and Receive Data Buffer Register

(RXB). 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 data is written to UDR, it will be transferred to the Transmit Data Buffer Register (TXB), and when data is read data UDR, it will return the contents of the Receive Data Buffer Register (RXB).




### ii. UCSRA (USART Control and Status Register A)

As the name suggests, it is used for controlling serial communication and status flags.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | RXC | TXC | UDRE | FE | DOR | PE | U2X | MPCM | UCSRA |
| Read/Write | R | R/W | R | R | R | R | R/W | R/W | |
| Initial Value | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | |

**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. When the MPCM bit is written to one, all the incoming frames received by the USART receiver that do not contain address information will be ignored. The transmitter is unaffected by the MPCM setting.

### iii. UCSRB (USART Control and Status Register B)

As the name suggests, it is also used for controlling serial communication and status flags.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | RXCIE | TXCIE | UDRIE | RXEN | TXEN | UCSZ2 | RXB8 | TXB8 | UCSRB |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

**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, UCSZ0 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. This bit is not used in this experiment module.

**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. This bit is not used in this experiment module.

**iv. UCSRC (USART Control and Status Register B)**

As the name suggests, it is also used for controlling serial communication and status flags.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| | URSEL | UMSEL | UPM1 | UPM0 | USBS | UCSZ1 | UCSZ0 | UCPOL | UCSRC |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | |

**The UCSRC Register shares the same I/O location as the UBRRH Register.**

**URSEL (Bit 7): Register Select**

This bit selects between accessing the UCSRC or the UBRRH Register. It is read as one when reading UCSRC. The URSEL must be one when writing the UCSRC.

**UMSEL (Bit 6): USART Mode Select**

This bit selects between Asynchronous and Synchronous mode of operation.

| UMSEL | Mode |
|---|---|
| 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.

| UPM1 | UPM0 | Parity Mode |
|---|---|---|
| 0 | 0 | Disabled |
| 0 | 1 | Reserved |
| 1 | 0 | Enabled, Even Parity |
| 1 | 1 | Enabled, Odd Parity |

**USBS (Bit 3): Stop Bit Select**

This bit selects the number of stop bits to be transmitted.

| USBS | Stop Bit(s) |
|---|---|
| 0 | 1-bit |
| 1 | 2-bit |

**UCSZ1:0 (Bit 2:1): Character Size**

These bits combined with the UCSZ2 bit in UCSRB set the character size in a frame.

| UCSZ2 | UCSZ1 | UCSZ0 | Character Size |
|---|---|---|---|
| 0 | 0 | 0 | 5-bit |
| 0 | 0 | 1 | 6-bit |
| 0 | 1 | 0 | 7-bit |
| 0 | 1 | 1 | 8-bit |
| 1 | 0 | 0 | Reserved |
| 1 | 0 | 1 | Reserved |
| 1 | 1 | 0 | Reserved |
| 1 | 1 | 1 | 9-bit |

**UCPOL (Bit 0): Clock Polarity**

This bit is used for synchronous mode only and will not be covered in this section.

**v. UBRR (USART Baud Rate Register)**

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 16-bit register called UBRR (UBRRH and UBRRL).

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | |
|---|---|---|---|---|---|---|---|---|---|
| | URSEL | – | – | – | UBRR[11:8] | | | | UBRRH |
| | UBRR[7:0] | | | | | | | | UBRRL |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R/W | R | R | R | R/W | R/W | R/W | R/W | |
| | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

The UBRRH Register shares the same I/O bus as the UCSRC Register.

**Bit 15 – URSEL: Register Select**

This bit selects between accessing the UBRRH or the UCSRC Register. It is read as zero when reading UBRRH. The URSEL must be zero when writing the UBRRH.

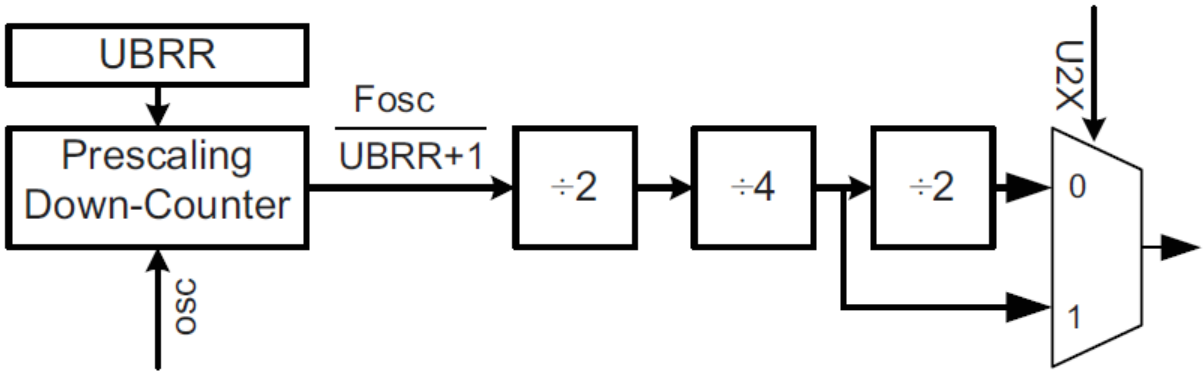
**Bit 14:12 – Reserved Bits**

These bits are reserved for future use. For compatibility with future devices, these bit must be written to zero when UBRRH is written.

**Bit 11:0 – UBRR11:0: USART Baud Rate Register**

This is a 12-bit register which contains the USART baud rate. The UBRRH contains the four most significant bits, and the UBRRL contains the 8 least significant bits of the USART baud rate. Ongoing transmissions by the

transmitter and receiver will be corrupted if the baud rate is changed. Writing UBRRL will trigger an immediate update of the baud rate prescaler.




For a given crystal frequency and U2X=0, the value loaded into the UBRR decides the baud rate. The relation between the value loaded into UBBR and the Fosc (frequency of oscillator) is dictated by the following formula:

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

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:

$$X = \frac{Fosc}{16(\text{Desired Baud Rate})} - 1$$

| Baud Rate (bps) | $f_{osc}$ = 1.0000MHz | | | | $f_{osc}$ = 1.8432MHz | | | | $f_{osc}$ = 2.0000MHz | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | U2X = 0 | | U2X = 1 | | U2X = 0 | | U2X = 1 | | U2X = 0 | | U2X = 1 | |
| | UBRR | Error | UBRR | Error | UBRR | Error | UBRR | Error | UBRR | Error | UBRR | Error |
| 2400 | 25 | 0.2% | 51 | 0.2% | 47 | 0.0% | 95 | 0.0% | 51 | 0.2% | 103 | 0.2% |
| 4800 | 12 | 0.2% | 25 | 0.2% | 23 | 0.0% | 47 | 0.0% | 25 | 0.2% | 51 | 0.2% |
| 9600 | 6 | -7.0% | 12 | 0.2% | 11 | 0.0% | 23 | 0.0% | 12 | 0.2% | 25 | 0.2% |
| 14.4k | 3 | 8.5% | 8 | -3.5% | 7 | 0.0% | 15 | 0.0% | 8 | -3.5% | 16 | 2.1% |
| 19.2k | 2 | 8.5% | 6 | -7.0% | 5 | 0.0% | 11 | 0.0% | 6 | -7.0% | 12 | 0.2% |
| 28.8k | 1 | 8.5% | 3 | 8.5% | 3 | 0.0% | 7 | 0.0% | 3 | 8.5% | 8 | -3.5% |
| 38.4k | 1 | -18.6% | 2 | 8.5% | 2 | 0.0% | 5 | 0.0% | 2 | 8.5% | 6 | -7.0% |
| 57.6k | 0 | 8.5% | 1 | 8.5% | 1 | 0.0% | 3 | 0.0% | 1 | 8.5% | 3 | 8.5% |
| 76.8k | – | – | 1 | -18.6% | 1 | -25.0% | 2 | 0.0% | 1 | -18.6% | 2 | 8.5% |
| 115.2k | – | – | 0 | 8.5% | 0 | 0.0% | 1 | 0.0% | 0 | 8.5% | 1 | 8.5% |
| 230.4k | – | – | – | – | – | – | 0 | 0.0% | – | – | – | – |
| 250k | – | – | – | – | – | – | – | – | – | – | 0 | 0.0% |
| Max (1) | 62.5 Kbps | | 125 Kbps | | 115.2 Kbps | | 230.4 Kbps | | 125 Kbps | | 250 Kbps | |

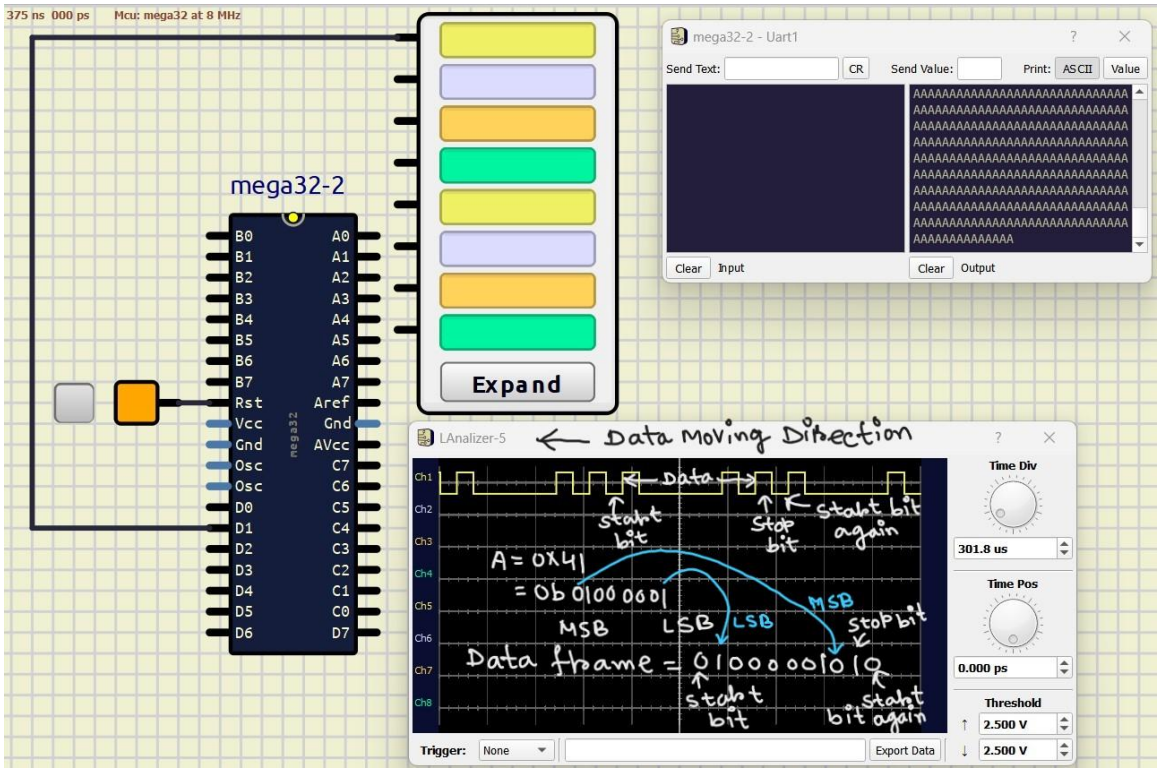
| Baud Rate (bps) | $f_{osc}$ = 3.6864MHz | | | | $f_{osc}$ = 4.0000MHz | | | | $f_{osc}$ = 7.3728MHz | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | U2X = 0 | | U2X = 1 | | U2X = 0 | | U2X = 1 | | U2X = 0 | | U2X = 1 | |
| | UBRR | Error | UBRR | Error | UBRR | Error | UBRR | Error | UBRR | Error | UBRR | Error |
| 2400 | 95 | 0.0% | 191 | 0.0% | 103 | 0.2% | 207 | 0.2% | 191 | 0.0% | 383 | 0.0% |
| 4800 | 47 | 0.0% | 95 | 0.0% | 51 | 0.2% | 103 | 0.2% | 95 | 0.0% | 191 | 0.0% |
| 9600 | 23 | 0.0% | 47 | 0.0% | 25 | 0.2% | 51 | 0.2% | 47 | 0.0% | 95 | 0.0% |
| 14.4k | 15 | 0.0% | 31 | 0.0% | 16 | 2.1% | 34 | -0.8% | 31 | 0.0% | 63 | 0.0% |
| 19.2k | 11 | 0.0% | 23 | 0.0% | 12 | 0.2% | 25 | 0.2% | 23 | 0.0% | 47 | 0.0% |
| 28.8k | 7 | 0.0% | 15 | 0.0% | 8 | -3.5% | 16 | 2.1% | 15 | 0.0% | 31 | 0.0% |
| 38.4k | 5 | 0.0% | 11 | 0.0% | 6 | -7.0% | 12 | 0.2% | 11 | 0.0% | 23 | 0.0% |
| 57.6k | 3 | 0.0% | 7 | 0.0% | 3 | 8.5% | 8 | -3.5% | 7 | 0.0% | 15 | 0.0% |
| 76.8k | 2 | 0.0% | 5 | 0.0% | 2 | 8.5% | 6 | -7.0% | 5 | 0.0% | 11 | 0.0% |
| 115.2k | 1 | 0.0% | 3 | 0.0% | 1 | 8.5% | 3 | 8.5% | 3 | 0.0% | 7 | 0.0% |
| 230.4k | 0 | 0.0% | 1 | 0.0% | 0 | 8.5% | 1 | 8.5% | 1 | 0.0% | 3 | 0.0% |
| 250k | 0 | -7.8% | 1 | -7.8% | 0 | 0.0% | 1 | 0.0% | 1 | -7.8% | 3 | -7.8% |
| 0.5M | – | – | 0 | -7.8% | – | – | 0 | 0.0% | 0 | -7.8% | 1 | -7.8% |
| 1M | – | – | – | – | – | – | – | – | – | – | 0 | -7.8% |
| Max (1) | 230.4Kbps | | 460.8Kbps | | 250Kbps | | 0.5Mbps | | 460.8Kbps | | 921.6Kbps | |

| Baud Rate (bps) | $f_{osc}$ = 8.0000MHz | | | | $f_{osc}$ = 11.0592MHz | | | | $f_{osc}$ = 14.7456MHz | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | U2X = 0 | | U2X = 1 | | U2X = 0 | | U2X = 1 | | U2X = 0 | | U2X = 1 | |
| | UBRR | Error | UBRR | Error | UBRR | Error | UBRR | Error | UBRR | Error | UBRR | Error |
| 2400 | 207 | 0.2% | 416 | -0.1% | 287 | 0.0% | 575 | 0.0% | 383 | 0.0% | 767 | 0.0% |
| 4800 | 103 | 0.2% | 207 | 0.2% | 143 | 0.0% | 287 | 0.0% | 191 | 0.0% | 383 | 0.0% |
| 9600 | 51 | 0.2% | 103 | 0.2% | 71 | 0.0% | 143 | 0.0% | 95 | 0.0% | 191 | 0.0% |
| 14.4k | 34 | -0.8% | 68 | 0.6% | 47 | 0.0% | 95 | 0.0% | 63 | 0.0% | 127 | 0.0% |
| 19.2k | 25 | 0.2% | 51 | 0.2% | 35 | 0.0% | 71 | 0.0% | 47 | 0.0% | 95 | 0.0% |
| 28.8k | 16 | 2.1% | 34 | -0.8% | 23 | 0.0% | 47 | 0.0% | 31 | 0.0% | 63 | 0.0% |
| 38.4k | 12 | 0.2% | 25 | 0.2% | 17 | 0.0% | 35 | 0.0% | 23 | 0.0% | 47 | 0.0% |
| 57.6k | 8 | -3.5% | 16 | 2.1% | 11 | 0.0% | 23 | 0.0% | 15 | 0.0% | 31 | 0.0% |
| 76.8k | 6 | -7.0% | 12 | 0.2% | 8 | 0.0% | 17 | 0.0% | 11 | 0.0% | 23 | 0.0% |
| 115.2k | 3 | 8.5% | 8 | -3.5% | 5 | 0.0% | 11 | 0.0% | 7 | 0.0% | 15 | 0.0% |
| 230.4k | 1 | 8.5% | 3 | 8.5% | 2 | 0.0% | 5 | 0.0% | 3 | 0.0% | 7 | 0.0% |
| 250k | 1 | 0.0% | 3 | 0.0% | 2 | -7.8% | 5 | -7.8% | 3 | -7.8% | 6 | 5.3% |
| 0.5M | 0 | 0.0% | 1 | 0.0% | – | – | 2 | -7.8% | 1 | -7.8% | 3 | -7.8% |
| 1M | – | – | 0 | 0.0% | – | – | – | – | 0 | -7.8% | 1 | -7.8% |
| Max (1) | 0.5Mbps | | 1Mbps | | 691.2Kbps | | 1.3824Mbps | | 921.6Kbps | | 1.8432Mbps | |

| Baud Rate (bps) | $f_{osc}$ = 16.0000MHz | | | | $f_{osc}$ = 18.4320MHz | | | | $f_{osc}$ = 20.0000MHz | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | U2X = 0 | | U2X = 1 | | U2X = 0 | | U2X = 1 | | U2X = 0 | | U2X = 1 | |
| | UBRR | Error | UBRR | Error | UBRR | Error | UBRR | Error | UBRR | Error | UBRR | Error |
| 2400 | 416 | -0.1% | 832 | 0.0% | 479 | 0.0% | 959 | 0.0% | 520 | 0.0% | 1041 | 0.0% |
| 4800 | 207 | 0.2% | 416 | -0.1% | 239 | 0.0% | 479 | 0.0% | 259 | 0.2% | 520 | 0.0% |
| 9600 | 103 | 0.2% | 207 | 0.2% | 119 | 0.0% | 239 | 0.0% | 129 | 0.2% | 259 | 0.2% |
| 14.4k | 68 | 0.6% | 138 | -0.1% | 79 | 0.0% | 159 | 0.0% | 86 | -0.2% | 173 | -0.2% |
| 19.2k | 51 | 0.2% | 103 | 0.2% | 59 | 0.0% | 119 | 0.0% | 64 | 0.2% | 129 | 0.2% |
| 28.8k | 34 | -0.8% | 68 | 0.6% | 39 | 0.0% | 79 | 0.0% | 42 | 0.9% | 86 | -0.2% |
| 38.4k | 25 | 0.2% | 51 | 0.2% | 29 | 0.0% | 59 | 0.0% | 32 | -1.4% | 64 | 0.2% |
| 57.6k | 16 | 2.1% | 34 | -0.8% | 19 | 0.0% | 39 | 0.0% | 21 | -1.4% | 42 | 0.9% |
| 76.8k | 12 | 0.2% | 25 | 0.2% | 14 | 0.0% | 29 | 0.0% | 15 | 1.7% | 32 | -1.4% |
| 115.2k | 8 | -3.5% | 16 | 2.1% | 9 | 0.0% | 19 | 0.0% | 10 | -1.4% | 21 | -1.4% |
| 230.4k | 3 | 8.5% | 8 | -3.5% | 4 | 0.0% | 9 | 0.0% | 4 | 8.5% | 10 | -1.4% |
| 250k | 3 | 0.0% | 7 | 0.0% | 4 | -7.8% | 8 | 2.4% | 4 | 0.0% | 9 | 0.0% |
| 0.5M | 1 | 0.0% | 3 | 0.0% | – | – | 4 | -7.8% | – | – | 4 | 0.0% |
| 1M | 0 | 0.0% | 1 | 0.0% | – | – | – | – | – | – | – | – |
| Max (1) | 1Mbps | | 2Mbps | | 1.152Mbps | | 2.304Mbps | | 1.25Mbps | | 2.5Mbps | |

- **Programming the ATMEGA32 to PC Serial Monitor (Serially Transfer Polling Method)**

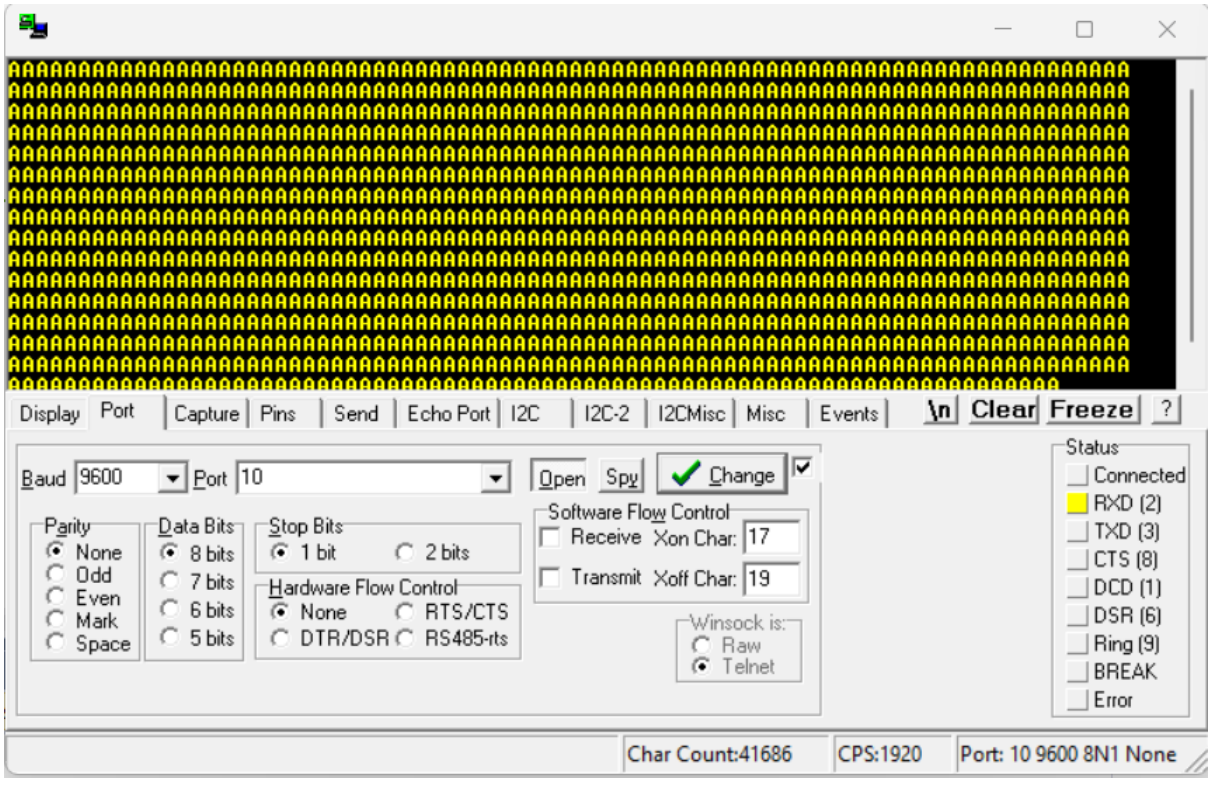
```
Experiment 1: /* Write a program for the AVR to transfer the letter 'A' serially at
9600 baud, continuously. Oscillator Frequency=8MHz, U2X=0, StopBit=1, No Parity Bit.
*/
.INCLUDE "M32DEF.INC"

LDI R16,HIGH(RAMEND)
OUT SPH,R16
LDI R16,LOW(RAMEND)
OUT SPL,R16

// Tx Rx pins Data Direction Register
CBI DDRD,PIND0 // Set Rx as Input Pin
SBI PORTD,PIND0 //Enable Internal Pull-Up Register

SBI DDRD,PIND1 // Set Tx as Output Pin
SBI PORTD,PIND1

LDI R16,(1<<TXEN) //Enable Transmitter
OUT UCSRB,R16
LDI R16,0x86
OUT UCSRC,R16
LDI R16,0x00
OUT UBRRH,R16
LDI R16,0x33
OUT UBRRL,R16

AGAIN:      SBIS UCSRA, UDRE
            RJMP AGAIN
            LDI R16,'A'
            OUT UDR,R16
            RJMP AGAIN
```



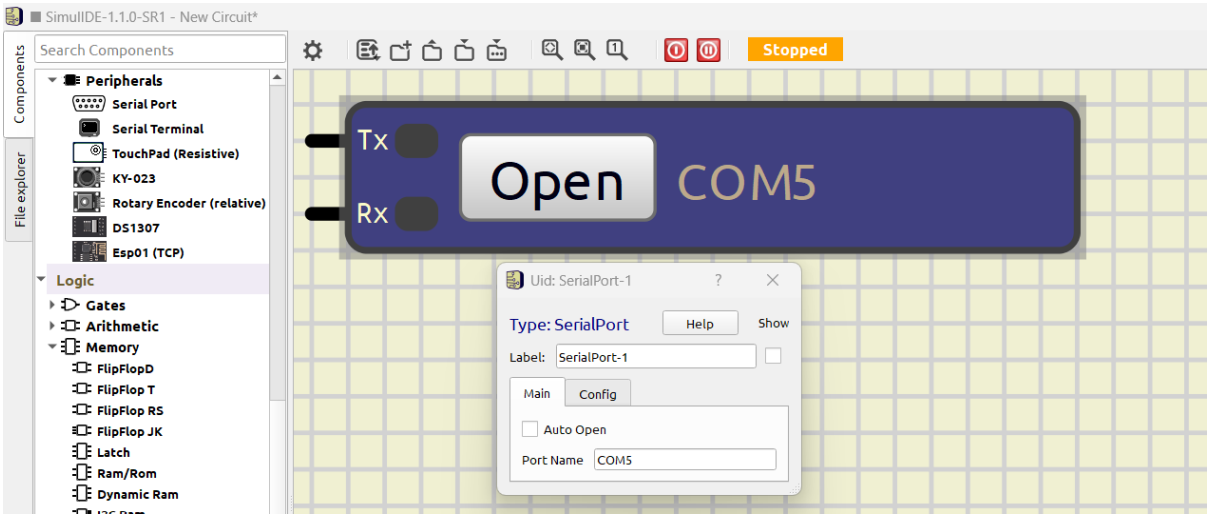

**Make the above circuit on SimulIDE to realize the UART transmitter and verify the logic transfer bits with logic analyzer. Make the hardware and verify with Realterm serial monitor.**

**Set Realterm serial monitor settings as below:**

Click on Port then select Baud: 9600, Parity: None, Data Bits: 8 bits, Stop bits: 1 bit, Hardware Flow Control: None, Select Port as showing at device manager then click open. Press reset button on AVR development board.



**Alternatively, you can use the SimulIDE serial port to view hardware data:**




Select the below setting to view the data

i. Right click on the **Serial Port** to select properties and enter USB to TTL converter COM Port number

Example: COM5

ii. Then click on **Config** to enter proper baud rate, data bits, stop bits.

iii. Then click on **Open** and start the simulation

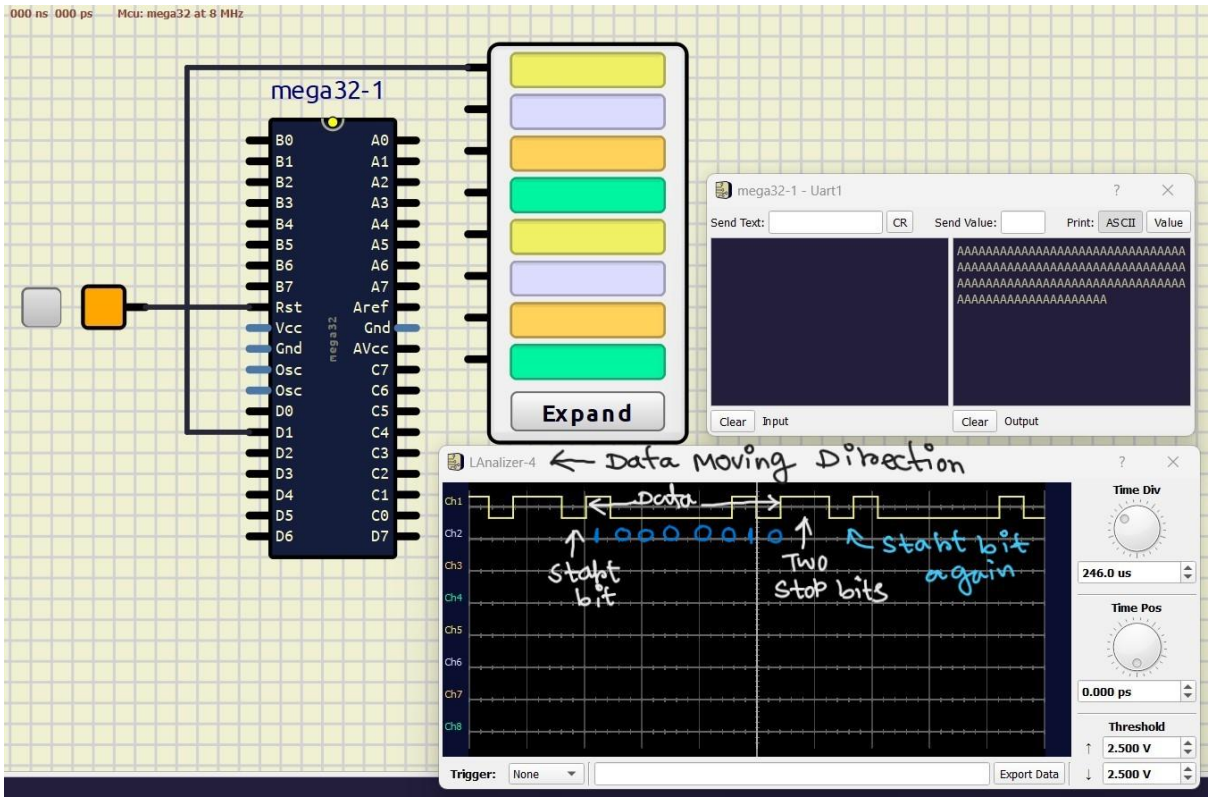
iv. At last right click on the serial port to open serial monitor.

v. Then click on clear and press reset button of the development board to view the exact data on the serial monitor.

**Experiment 2:** /* Write a program for the AVR to transfer the letter 'A' serially at 9600 baud, continuously. Assume Oscillator Frequency=8MHz, No Parity bit, 2 stop bits. */

```
.INCLUDE "M32DEF.INC"

LDI R16,HIGH(RAMEND)
OUT SPH,R16
LDI R16,LOW(RAMEND)
OUT SPL,R16

// Tx Rx pins Data Direction Register
CBI DDRD,PIND0 // Set Rx as Input Pin
SBI PORTD,PIND0 //Enable Internal Pull-Up Register

SBI DDRD,PIND1 // Set Tx as Output Pin
SBI PORTD,PIND1

LDI R16,(1<<TXEN) //Enable Transmitter
OUT UCSRB,R16
LDI R16,0x8E
OUT UCSRC,R16
LDI R16,0x00
OUT UBRRH,R16
LDI R16,0x33
OUT UBRRL,R16

AGAIN:      SBIS UCSRA, UDRE
            RJMP AGAIN
```

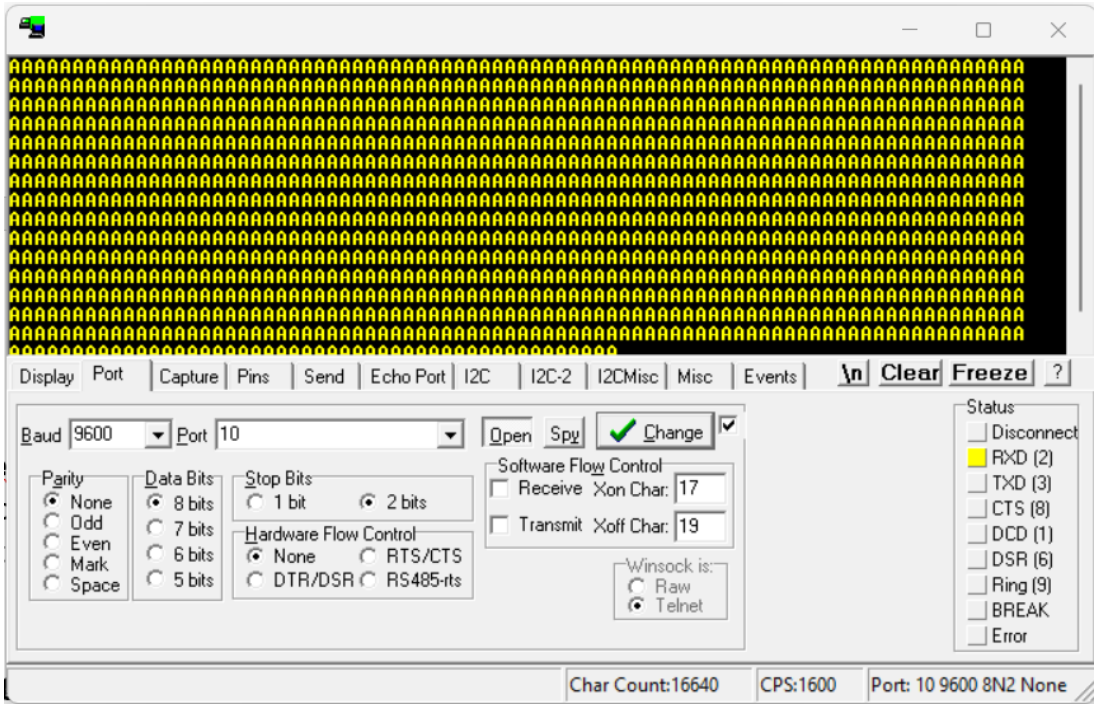
```
LDI R16,'A'
OUT UDR,R16
RJMP AGAIN
```




**Make the above circuit on SimulIDE to realize the UART transmitter and verify the logic transfer bits with logic analyzer. Make the hardware and verify with Realterm serial monitor.**

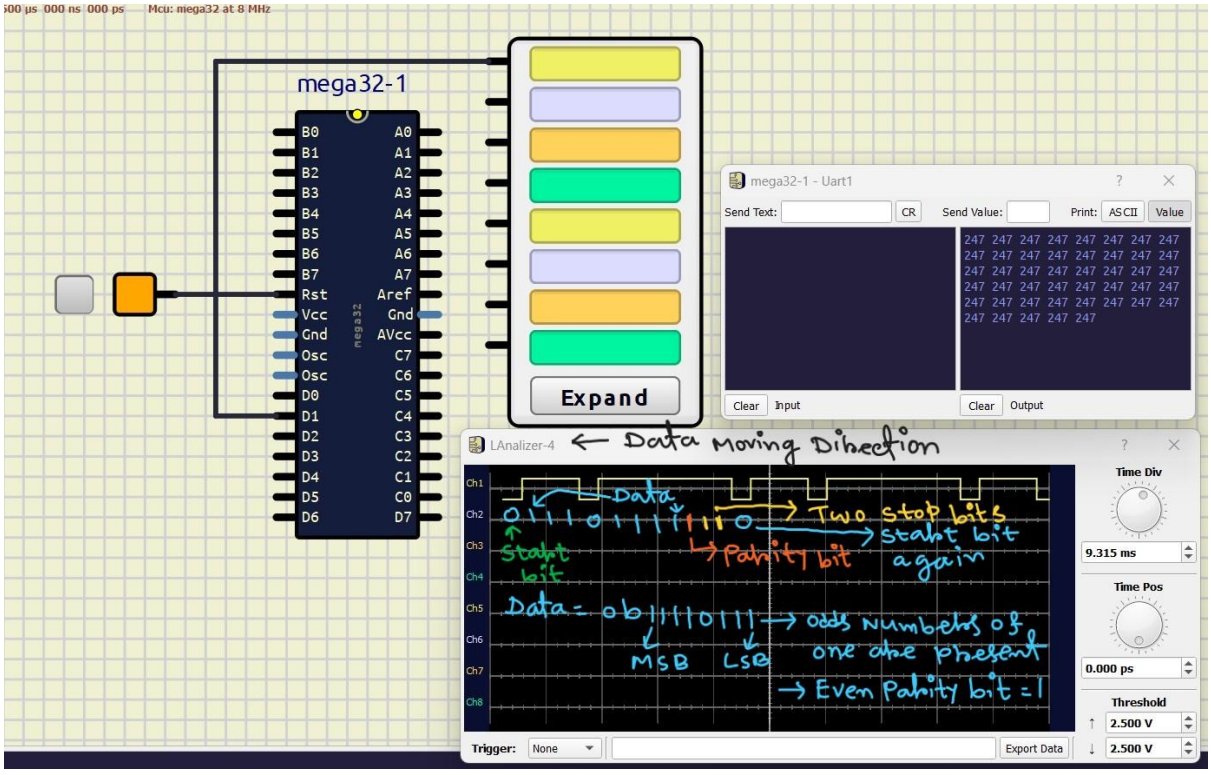
**Set Realterm serial monitor settings as below:**

Click on Port then select Baud: 9600, Parity: None, Data Bits: 8 bits, Stop bits: 2 bits, Hardware Flow Control: None, Select Port as showing at device manager then click open. Press reset button on AVR development board.



**Experiment 3:** /* Write a program for the AVR to transfer the 8-bit data '0b11110111' serially at 9600 baud, continuously. Assume Oscillator Frequency=8MHz, Even Parity Mode, 2 stop bits. */

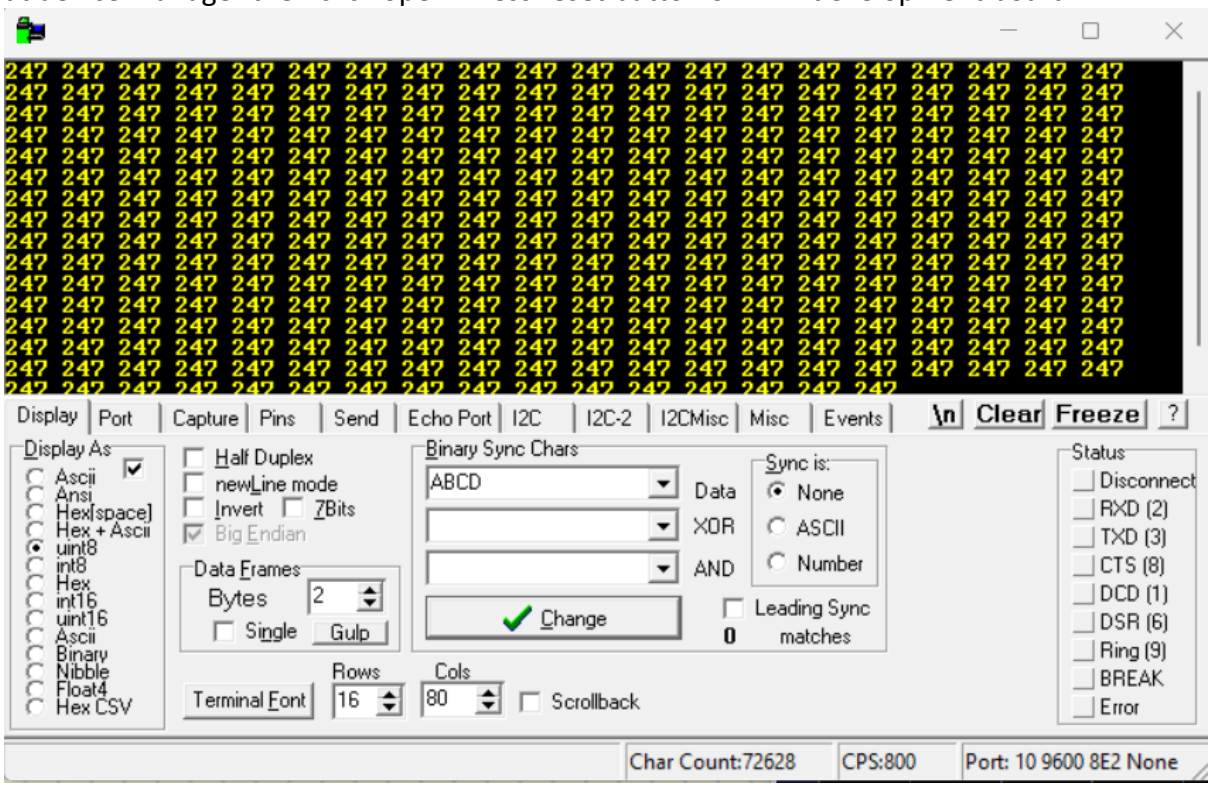
```
.INCLUDE "M32DEF.INC"
.ORG 0x0000

LDI R16,HIGH(RAMEND)
OUT SPH,R16
LDI R16,LOW(RAMEND)
OUT SPL,R16
// Tx Rx pins Data Direction Register
CBI DDRD,PIND0 // Set Rx as Input Pin
SBI PORTD,PIND0 //Enable Internal Pull-Up Register
SBI DDRD,PIND1 // Set Tx as Output Pin
SBI PORTD,PIND1

LDI R16,(1<<TXEN) //Enable Transmitter
OUT UCSRB,R16
LDI R17,0xAE
OUT UCSRC,R17
LDI R16,0x33
OUT UBRRL,R16

AGAIN:      SBIS UCSRA, UDRE
            RJMP AGAIN
            LDI R16,0b11110111
            OUT UDR,R16
            RJMP AGAIN
```




**Make the above circuit on SimulIDE to realize the UART transmitter and verify the logic transfer bits with logic analyzer. Make the hardware and verify with Realterm serial monitor.**

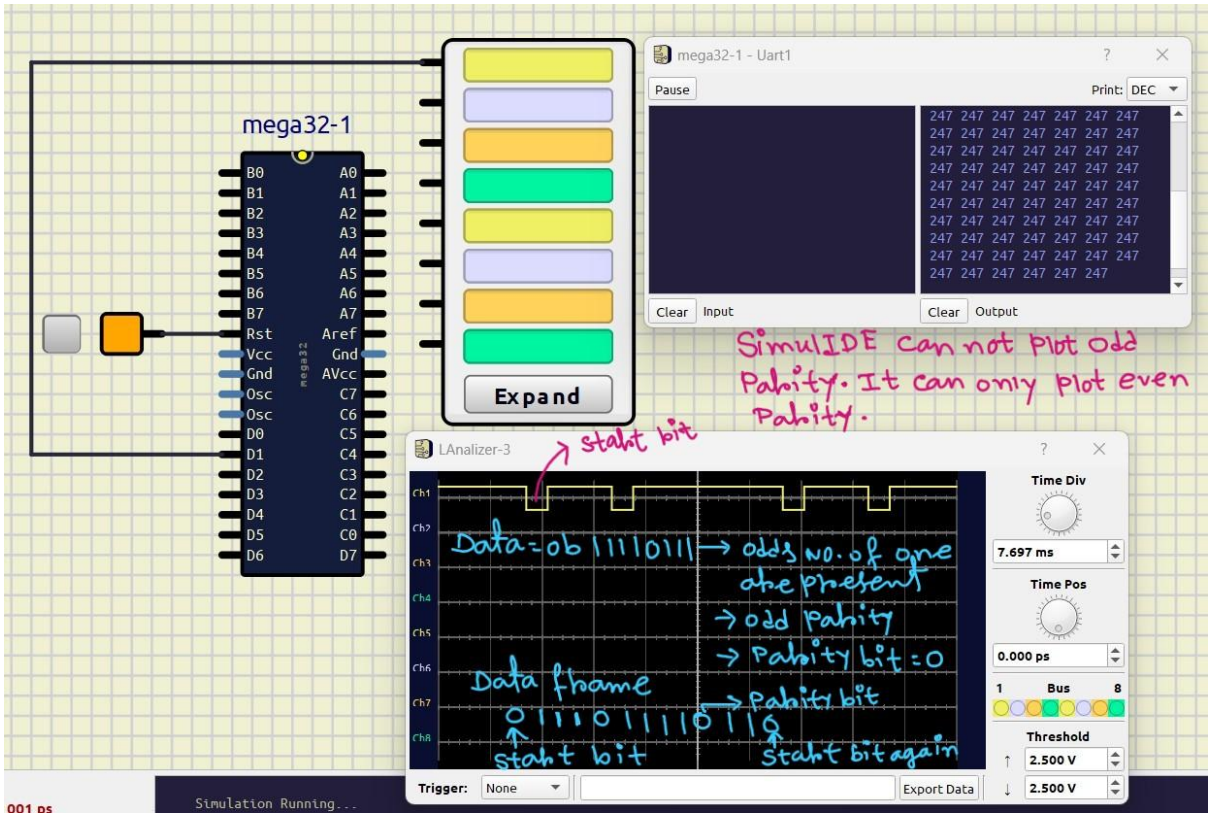
**Set Realterm serial monitor settings as below:**

First click on display select Display as unit8. Then click on Port then select Baud: 9600, Parity: Even, Data Bits: 8 bits, Stop bits: 2 bits, Hardware Flow Control: None, Select Port as showing at device manager then click open. Press reset button on AVR development board.

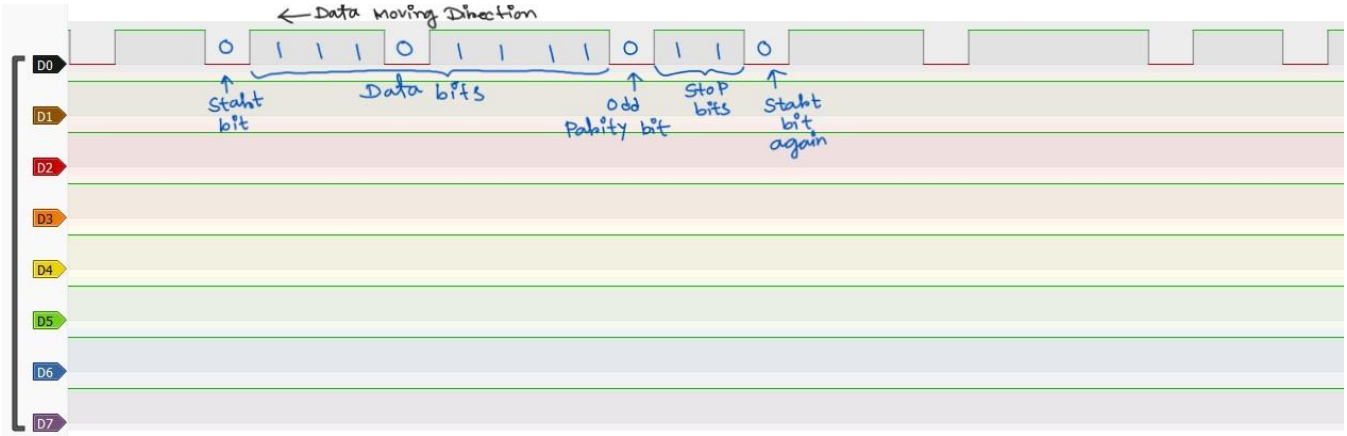



**Experiment 4:** /* Write a program for the AVR to transfer the 8-bit data '0b11110111' serially at 9600 baud, countinuously. Assume Oscillator Frequency=8MHz, Odd Parity Mode, 2 Stop bits */

```
.INCLUDE "M32DEF.INC"

LDI R16,HIGH(RAMEND)
OUT SPH,R16
LDI R16,LOW(RAMEND)
OUT SPL,R16

// Tx Rx pins Data Direction Register
CBI DDRD,PIND0 // Set Rx as Input Pin
SBI PORTD,PIND0 //Enable Internal Pull-Up Register

SBI DDRD,PIND1 // Set Tx as Output Pin
SBI PORTD,PIND1

LDI R16,(1<<TXEN) //Enable Transmitter
OUT UCSRB,R16
LDI R16,0xBE
OUT UCSRC,R16
LDI R16,0x33
OUT UBRRL,R16

AGAIN:      SBIS UCSRA, UDRE
            RJMP AGAIN
            LDI R16,0b11110111
            OUT UDR,R16
            RJMP AGAIN
```



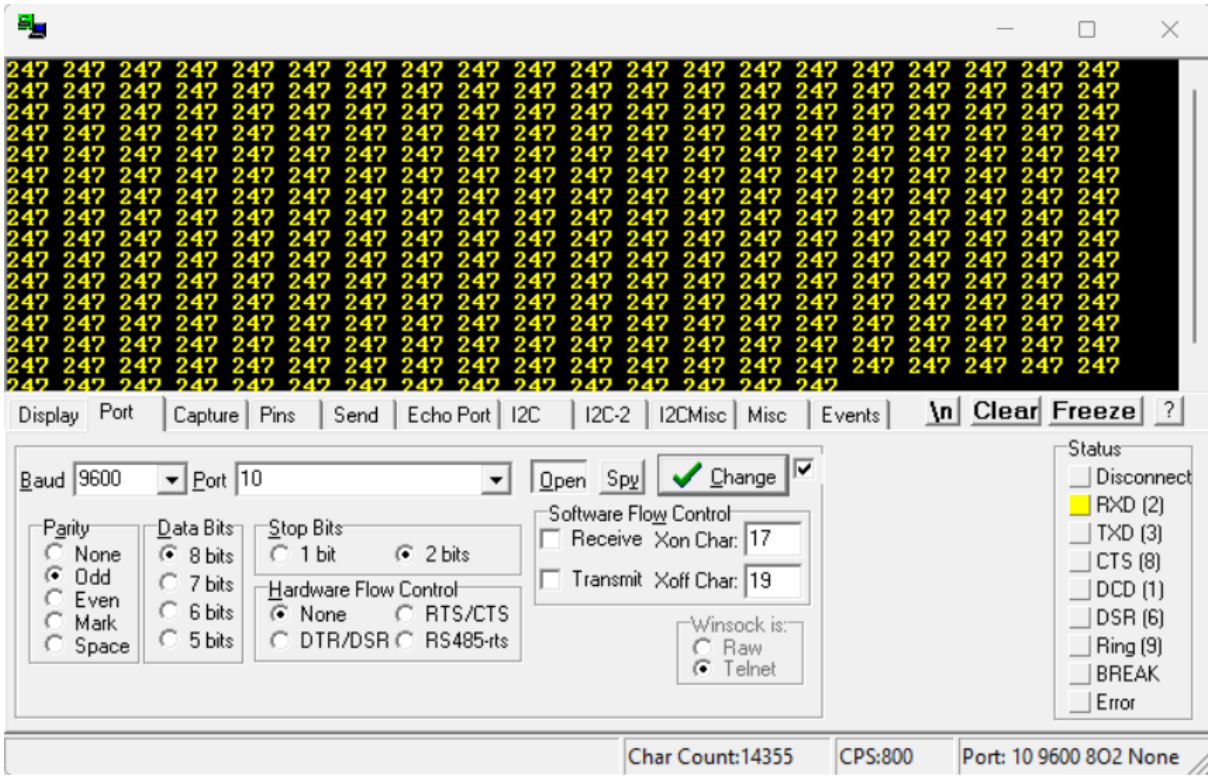

**To verify the data frame for Odd Parity Mode, logic analyzer hardware will be used to capture the data frame**




**Make the hardware and verify with Realterm serial monitor.**

**Set Realterm serial monitor settings as below:**

First click on display select Display As unit8. Then click on Port then select Baud: 9600, Parity: Odd, Data Bits: 8 bits, Stop bits: 2 bits, Hardware Flow Control: None, Select Port as showing at device manager then click open. Press reset button on AVR development board.



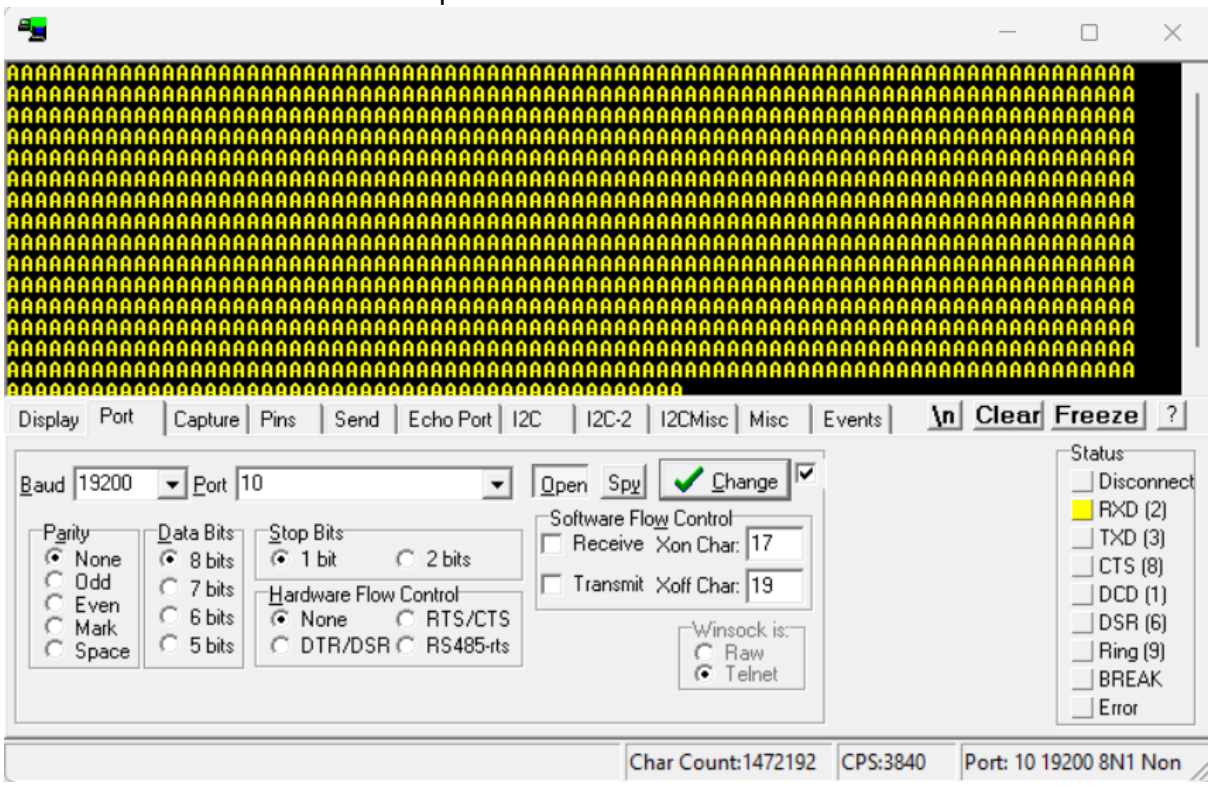

**Experiment 5:** /* Write a program for the AVR to transfer the letter 'A' serially at 9600 baud, continuously with U2X=0. Assume Oscillator Frequency=8MHz, No Parity bit, 1 stop bit.
Set U2X=1 at UCSRA register and don’t modify UBRR, then the baud rate will be 19.2K which is double of 9600. */

```
.INCLUDE "M32DEF.INC"

LDI R16,HIGH(RAMEND)
OUT SPH,R16
LDI R16,LOW(RAMEND)
OUT SPL,R16

// Tx Rx pins Data Direction Register
CBI DDRD,PIND0 // Set Rx as Input Pin
SBI PORTD,PIND0 //Enable Internal Pull-Up Register

SBI DDRD,PIND1 // Set Tx as Output Pin
SBI PORTD,PIND1

LDI R16,(1<<TXEN) //Enable Transmitter
OUT UCSRB,R16
LDI R16,0x86
OUT UCSRC,R16
LDI R16,0x00
OUT UBRRH,R16
LDI R16,0x33
OUT UBRRL,R16
SBI UCSRA,U2X

AGAIN:      SBIS UCSRA, UDRE
            RJMP AGAIN
            LDI R16,'A'
            OUT UDR,R16
            RJMP AGAIN
```

**Make the hardware and verify with Realterm serial monitor.**

**Set Realterm serial monitor settings as below:**

Click on Port then select Baud: 19200, Parity: None, Data Bits: 8 bits, Stop bits: 1 bits, Hardware Flow Control: None, Select Port as showing at device manager then click open. Press reset button on AVR development board.




- ## Programming the ATMEGA32 to PC Serial Monitor (Serially Transfer Interrupt Method)

  To program the serial port to transmit data using the interrupt method, we need to set HIGH the USART Data Register Empty Interrupt Enable (UDRIE) bit in UCSRB. Setting this bit enables the interrupt on the UDRE flag in UCSRA. When the UDR register is ready to accept new data, the UDRE (USART Data Register Empty flag) becomes HIGH. If UDRIE = 1, changing UDRE to one will force the CPU to jump to the interrupt vector.

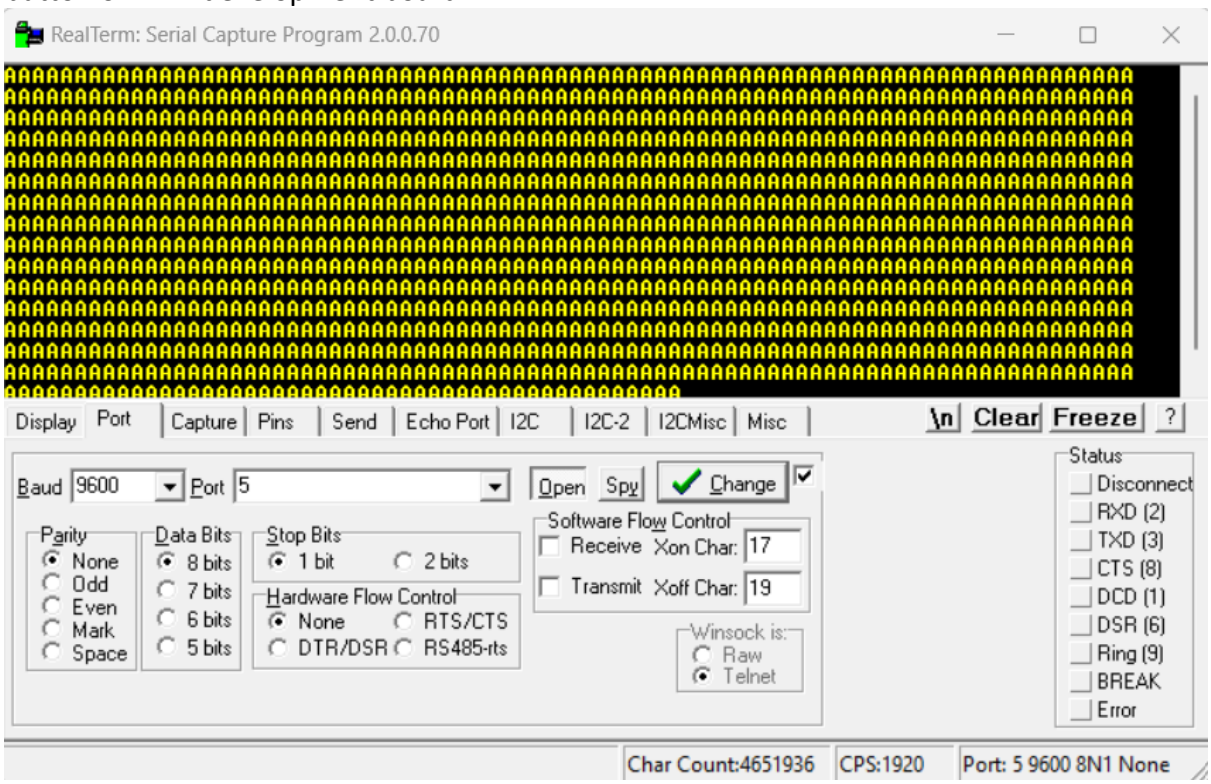
```
Experiment 6: /* Write a program for the AVR to transmit the letter 'A' serially at
9600 baud, continuously.
Assume Oscillator Frequency=8MHz, No Parity bit, 1 stop bit.
Use interrupts instead of the polling method. */

.INCLUDE "M32DEF.INC"
.ORG 0x0000
       JMP MAIN
.ORG 0x001C
       JMP Tx

MAIN:         LDI R16,HIGH(RAMEND)
              OUT SPH,R16
              LDI R16,LOW(RAMEND)
```

```
                OUT SPL,R16

                // Tx Rx pins Data Direction Register
                CBI DDRD,PIND0 // Set Rx as Input Pin
                SBI PORTD,PIND0 //Enable Internal Pull-Up Register

                SBI DDRD,PIND1 // Set Tx as Output Pin
                SBI PORTD,PIND1

                LDI R16,0x28
                OUT UCSRB,R16
                LDI R16,0x86
                OUT UCSRC,R16
                LDI R16,0x00
                OUT UBRRH,R16
                LDI R16,0x33
                OUT UBRRL,R16
                SEI
                AGAIN: NOP
                       RJMP AGAIN

Tx:     LDI R16,'A'
        OUT UDR,R16
        RETI
```

**Make the hardware and verify with Realterm serial monitor.**

**Set Realterm serial monitor settings as below:**

Click on Port then select Baud: 9600, Parity: None, Data Bits: 8 bits, Stop bits: 1 bits, Hardware Flow Control: None, Select Port as showing at device manager then click open. Press reset button on AVR development board.

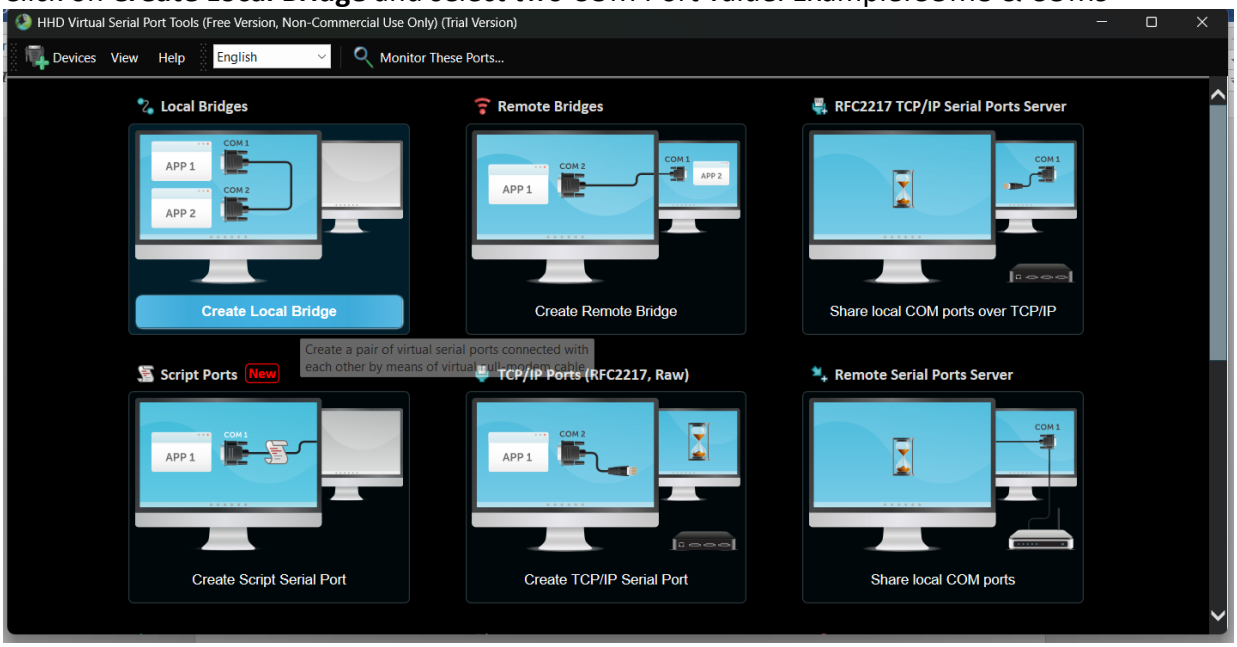


- # Programming the ATMEGA32 to PC Serial Plotter

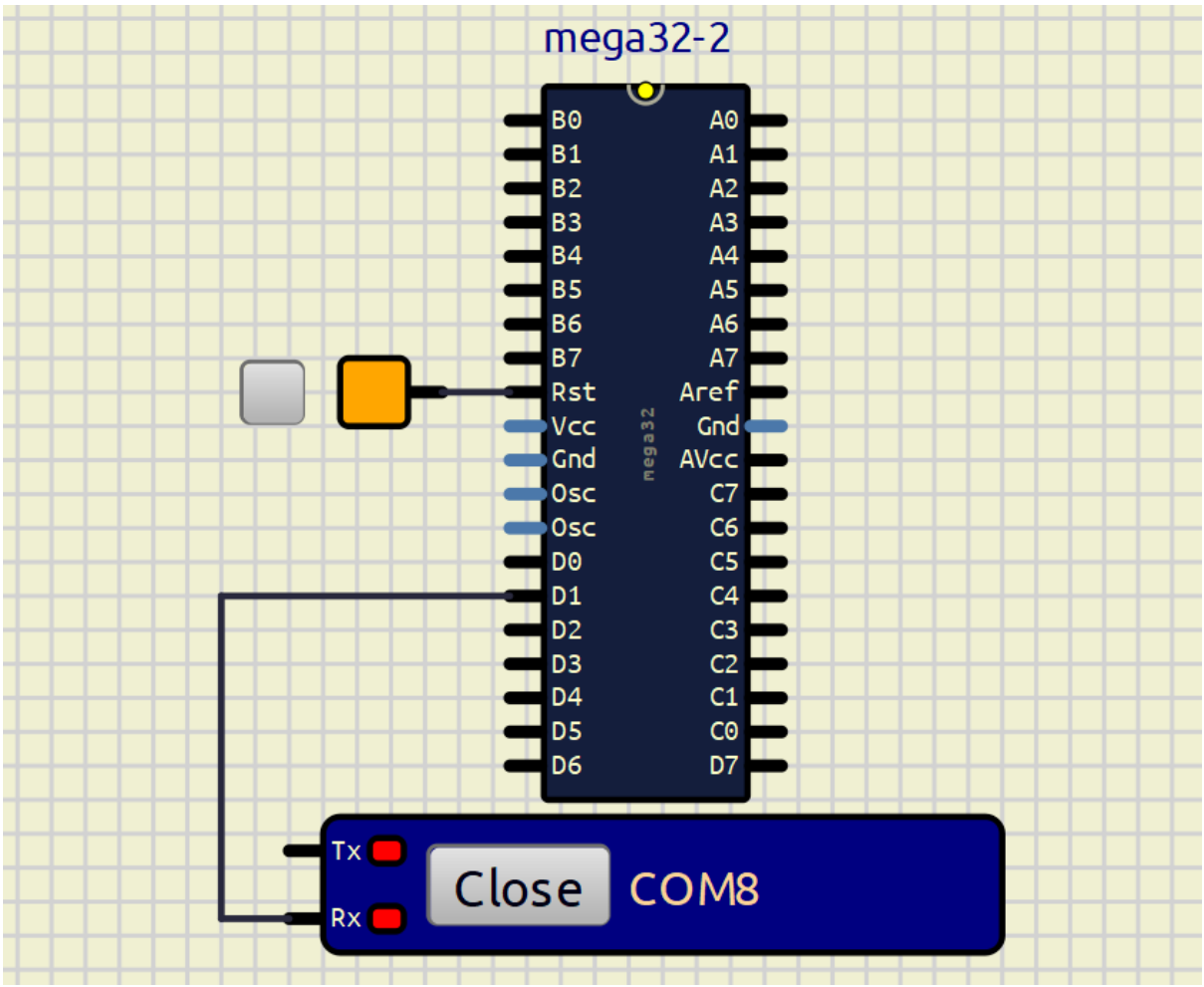
```
Experiment 7: /* Sawtooth wave plotting through UART on serial plotter.
Set the baud rate at 9600, oscillator frequency 8Mhz, 8-bit data, and 1 stop bit. */

.INCLUDE "M32DEF.INC"
.ORG 0x0000
       JMP MAIN
.ORG 0x001C
       JMP Tx_Data

MAIN:  LDI R16,HIGH(RAMEND)
       OUT SPH,R16
       LDI R16,LOW(RAMEND)
       OUT SPL,R16

       //Data Direction Register for transmitter
       SBI DDRD,PIND1
       //UART Initialization
       LDI R16,0x28
       OUT UCSRB,R16
       LDI R16,0x86
       OUT UCSRC,R16
       LDI R16,51
       OUT UBRRL,R16
       SEI
       LDI R17,0x00
       Infinite_Loop:       NOP
                            JMP Infinite_Loop

Tx_Data:      OUT UDR,R17
              INC R17
              RETI
```

**How to simulate the serial plotter on simulIDE:**

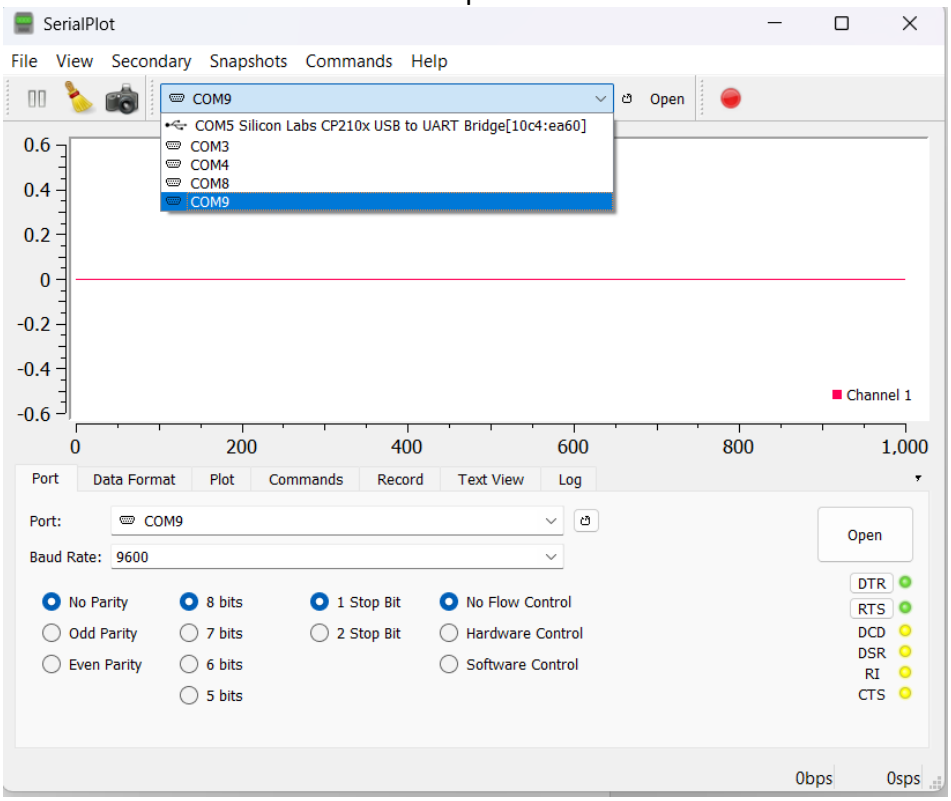
1. Install the software **Virtual Serial Port Tools**
2. Open the software and click on **Continue evaluation**
3. Click on **Create Local Bridge** and select two COM Port value. Example:COM8 & COM9



4. Put COM8 at Serial Port on SimulIDE and run the simulation

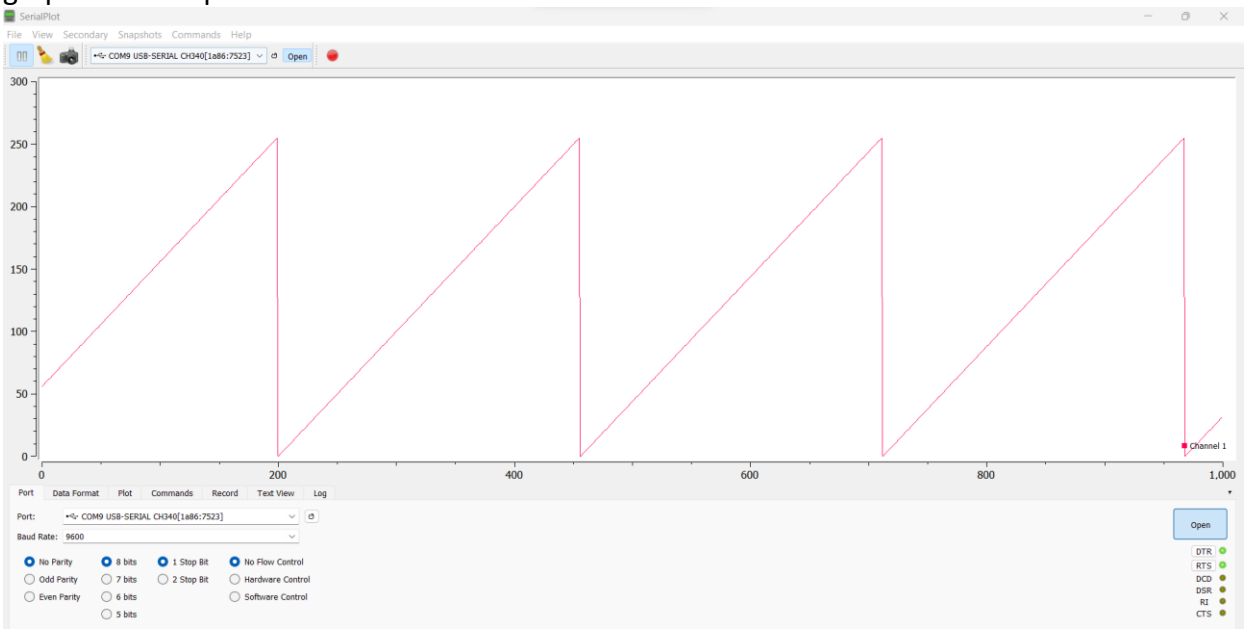



Put COM9 at Serial Plotter and then click Open

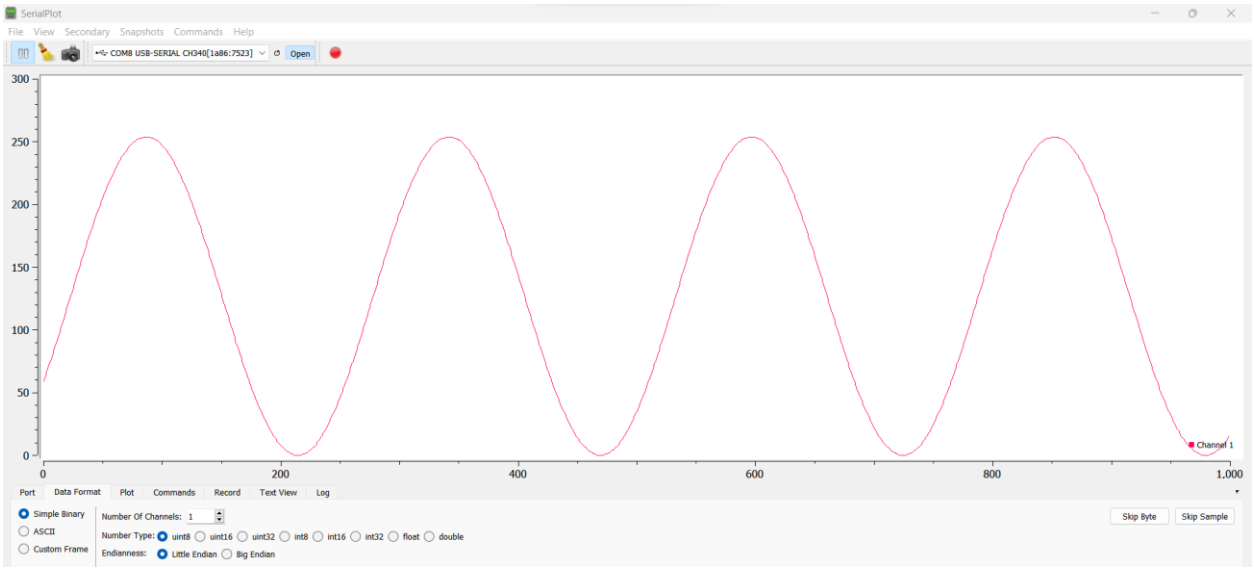


**How to view the waveform on Serial Plotter from Hardware:**

1. Connect USB to TTL Converter at USB and find out the COM Port number. If the COM Port was used in Realterm then close it from the Realterm software.
2. Select that COM Port inside Serial Plotter to view the waveform.
3. Select proper Baud Rate, Parity, Data Bits, Stop Bits with respect to your code.
4. Press reset button on AVR development board and then finally click open to view the graph on serial plotter.




**Class Assignment 1: Write Assembly Code for UART transmission ATMEGA32 to PC serial plotter in 8bit Mode to plot the below sine function continuously. Set Fosc=4MHz, Baud Rate=9600, No Parity bit and 1 stop bit.**



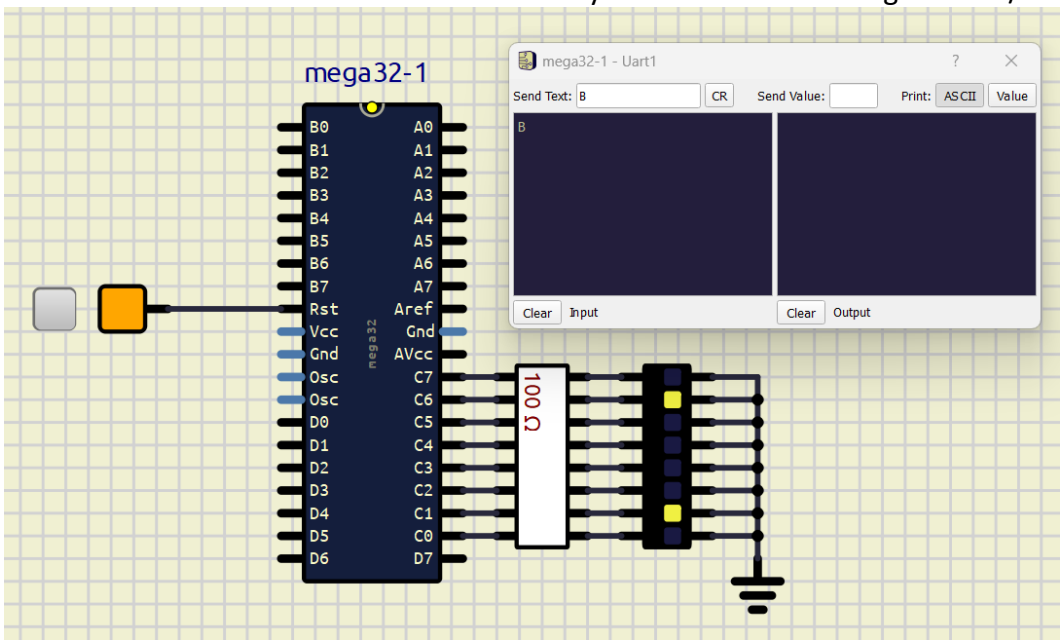

**Simulate it through virtual port creation to display the graph on serial plotter. Then make the hardware accordingly to display the same on serial plotter through USB to TTL converter.**

- # Programming the PC Serial Monitor to ATMEGA32 (Serially Receive Polling Method)

**Experiment 8:** /* Program the ATMEGA32 to receive bytes of data serially and put them on PORTC. Set the baud rate at 9600, oscillator frequency 8Mhz, 8-bit data, and 1 stop bit. */

```
.INCLUDE "M32DEF.INC"
.ORG 0x0000

LDI R16,HIGH(RAMEND)
OUT SPH,R16
LDI R16,LOW(RAMEND)
OUT SPL,R16

//Data Direction Register Initialization
CBI DDRD,PIND0 // PORTD0 as Input Pin
SBI PORTD,PIND0 //Enabling internal Pull-up register

LDI R16,0xFF
OUT DDRC,R16 // PORTC as data receive status pins

// UART Initialization
LDI R16,0x10
OUT UCSRB,R16
LDI R16,0x86
OUT UCSRC,R16
LDI R16,51
OUT UBRRL,R16

AGAIN:      SBIS UCSRA,RXC
            RJMP AGAIN
            IN R17,UDR
            OUT PORTC,R17
            RJMP AGAIN
```

Make the below circuit on SimulIDE and verify LED Dots with sending number/character.




**Class Assignment 2: Modify the above code to receive the data through interrupt.**

**** These below experiments can be completed after learning about the LCD 16x2.**

**Experiment 9:** /* Program the ATMEGA32 to receive bytes of data serially and put them on LCD 16x2 at PORTB. Set the baud rate at 9600, oscillator frequency 8Mhz, 8-bit data, and 1 stop bit. */

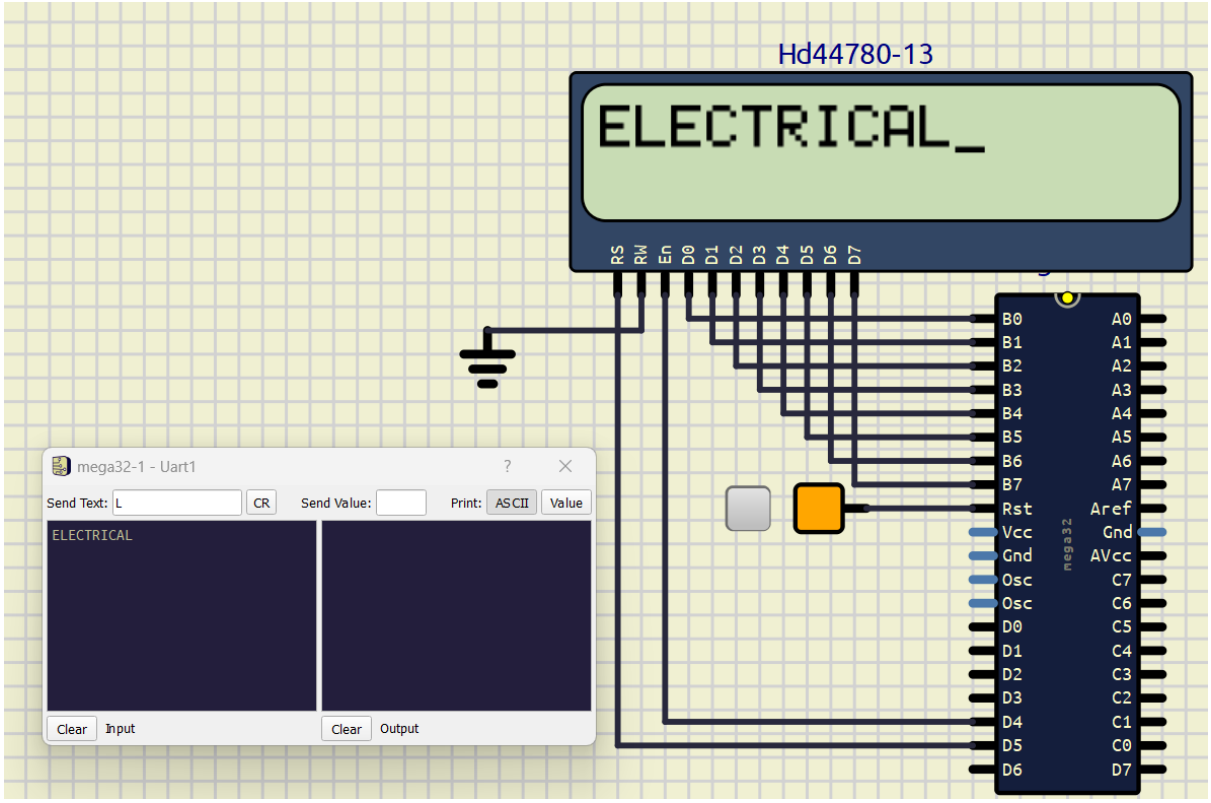
```
.INCLUDE "M32DEF.INC"
.ORG 0x0000

LDI R16,HIGH(RAMEND)
OUT SPH,R16
LDI R16,LOW(RAMEND)
OUT SPL,R16

// Data Direction Register Settings for LCD
LDI R16,0xFF
OUT DDRB,R16

SBI DDRD,PIND4 //Falling Edged Enable
SBI DDRD,PIND5 //Register Select

// LCD Initialization
CBI PORTD,PIND5 // Command Register Enable
LDI R16,0x38 //2 lines and 5x7 matrix
OUT PORTB,R16
CALL ENABLE
LDI R16,0x02 // Return Home
OUT PORTB,R16
CALL ENABLE
LDI R16,0x01 // Clear display screen
OUT PORTB,R16
CALL ENABLE
LDI R16,0x0E //Display on, cursor blinking
OUT PORTB,R16
CALL ENABLE
LDI R16,0x06 // Shift Cursor to right after print on LCD
OUT PORTB,R16
CALL ENABLE

//Set Cursor Coordinate
LDI R16,0x80 //Set Cursor at begining of 1st Line
OUT PORTB,R16
CALL ENABLE

// Tx Rx pins Data Direction Register
CBI DDRD,PIND0 // Set Rx as Input Pin
SBI PORTD,PIND0 //Enable Internal Pull-Up Register

SBI DDRD,PIND1 // Set Tx as Output Pin
SBI PORTD,PIND1

//UART Initialization
LDI R16,0x18// Enabling Tx Rx
OUT UCSRB,R16
LDI R17,0x86// 8 bit data mode
OUT UCSRC,R17
//9600bps Baud-Rate Settings for 8MHz Oscillator
LDI R16,0x00
OUT UBRRH,R16
LDI R16,51
OUT UBRRL,R16
```

```
SBI PORTD,PIND5 //Data Register Select Mode for LCD

Infinite_Loop:              SBIS UCSRA,RXC
                            RJMP Infinite_Loop
                            IN R20,UDR
                            OUT PORTB,R20
                            CALL ENABLE
                            JMP Infinite_Loop

ENABLE :        SBI PORTD,PIND4
                LDI R18,0xFF
                LOOP2:  LDI R17,0xFF
                        LOOP1:      NOP
                                    DEC R17
                                    BRNE LOOP1
                        DEC R18
                        BRNE LOOP2
                CBI PORTD,PIND4
                RET
```

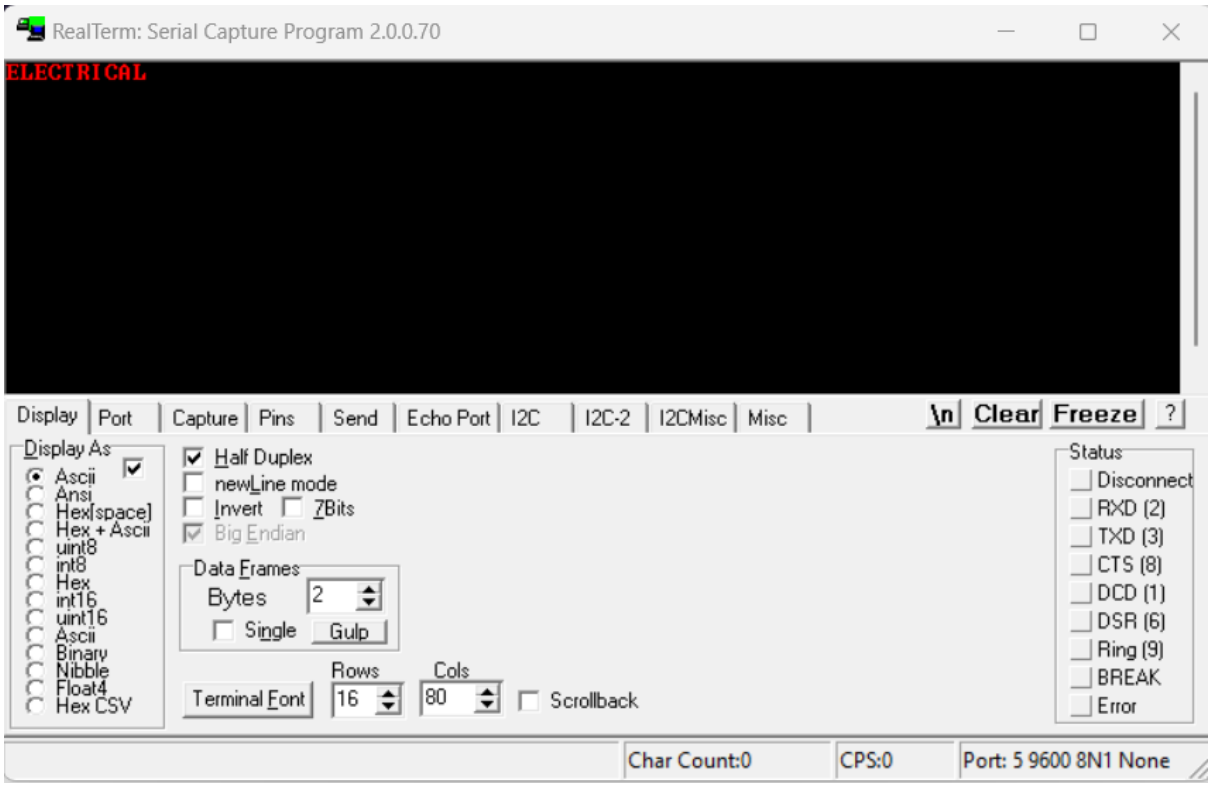
**Make the below circuit on SimulIDE to realize the UART receiver and verify the logic receive bits with LCD 16x2. Make the hardware and send data from Realterm serial monitor and verify with LCD 16x2.**

**Set Realterm serial monitor settings as below:**

First click on display select Half Duplex. Then click on Port then select Baud: 9600, Parity: None, Data Bits: 8 bits, Stop bits: 1 bits, Hardware Flow Control: None, Select Port as showing at device manager then click open. Press reset button on AVR development board and then type on the screen at Realterm.



Realterm serial monitor:




Demo Video Link: https://youtu.be/KF-QmtLMSok

**Verify the above send data on the LCD 16x2 which is connected with ATMEGA32.**

**Class Assignment 3: Program the ATMEGA32 to receive bytes of data serially through interrupt and display on LCD 16x2. Set the baud rate at 9600, oscillator frequency 8Mhz, 8-bit data, and 1 stop bit.**

**Class Assignment 4: Write Assembly Code for UART transmission PC through ATMEGA32 to LCD 16x2 to display text and if enter hits the line will be changed to next line. Set 8bit mode, Fosc=8MHz, Baud Rate=9600, U2X=0 and 1 stopbits.**