# Lecture 3’s sequence

- **3.1** Serial communication – The basics
- **3.2** Serial communication in ATmega16
- **3.3** Example application & Debugging tool

# Serial communication– The basics

- **Computers transfer data in two ways: parallel and serial:**
  - **Parallel:** **Several data bits are transferred simultaneously, e.g., to printers and hard disks**
  - **Serial:** **A single data bit is transferred at one time**

- **Advantages of serial communication: longer distances, easier to synchronize, fewer IO pins, and lower cost.**

- **Serial communication often requires:**
  - **Shift registers:** **convert a byte to serial bits and vice versa**
  - **Modems:** **modulate/demodulate serial bits to/from audio tones**

# Synchronous versus asynchronous

- **Synchronous serial communication:**
  - **The clocks of the sender and receiver are synchronized**
  - **A block of characters, enclosed by synchronizing bytes, is sent at a time**
  - **Faster transfer and less overhead**
  - **Examples: serial peripheral interface (SPI) by Motorola, binary synchronous communication (BISYNC) by IBM.**

- **Asynchronous serial communication:**
  - **The clocks of the sender and receiver are not synchronized**
  - **One character (usually 8 bits) is sent at a time, enclosed between a start bit and one or two stop bits. A parity bit may be included**
  - **Examples: RS-232 by Electronic Industries Alliance, USART of ATmega16**

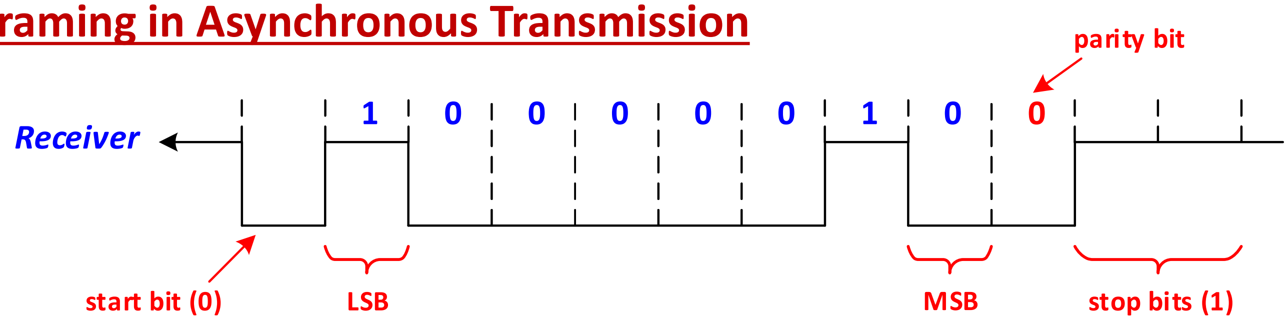
# Data framing examples

## Data Framing in Synchronous BISYNC

| | | | | | | |
|---|---|---|---|---|---|---|
| SYN | SYN | STX | DATA FIELD | ETX | BCC | PAD |

| BISYNC Control Characters | |
|---|---|
| SYN (16h): | synchronization |
| STX (02h): | start of text |
| ETX (03h): | end of text |
| BCC: | block checksum char |
| PAD (FFh): | end of frame block |

## Data Framing in Asynchronous Transmission




Sending character "A" (41h = 0100 0001)
8-bit data, 1 start bit, 2 stop bits, even-parity

# Serial communication terminology

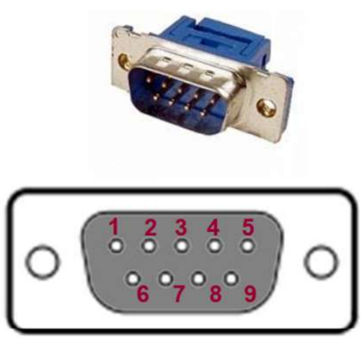
- **Baud rate:** The number of bits sent per second (bps). Strictly speaking, it is the number of signal changes per second.
- **Parity bit:** A single bit for error checking, sent with data bits to make the total number of 1's.
  - even (for even parity), or
  - odd (for odd parity)
- **Start bit:** to indicate the start of a character. Its typical value is 0
- **Stop bit:** to indicate the end of a character. Its typical value is 1

# RS-232 standard

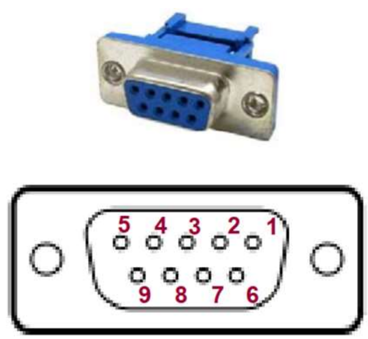
- **The RS-232 (latest revision RS-232E) is a widely used standard for serial interfacing.**
- **It covers four main aspects:**
  - **Electrical**: **voltage level, rise and fall time, data rate, distance**
  - **Functional**: **function of each signal**
  - **Mechanical**: **number of pins, shape & dimension of connectors**
  - **Procedural**: **sequence of events for transmitting data**

# RS-232 standard

- **It defines 25-pin D connectors, but 9-pin connectors are often used.**
- **RS-232 specifies the baud rate up to 20Kbps, and the cable length up to 15m. In practice, it supports up to 56Kbps & 30m of shielded cables.**




*Male DB9 connector*



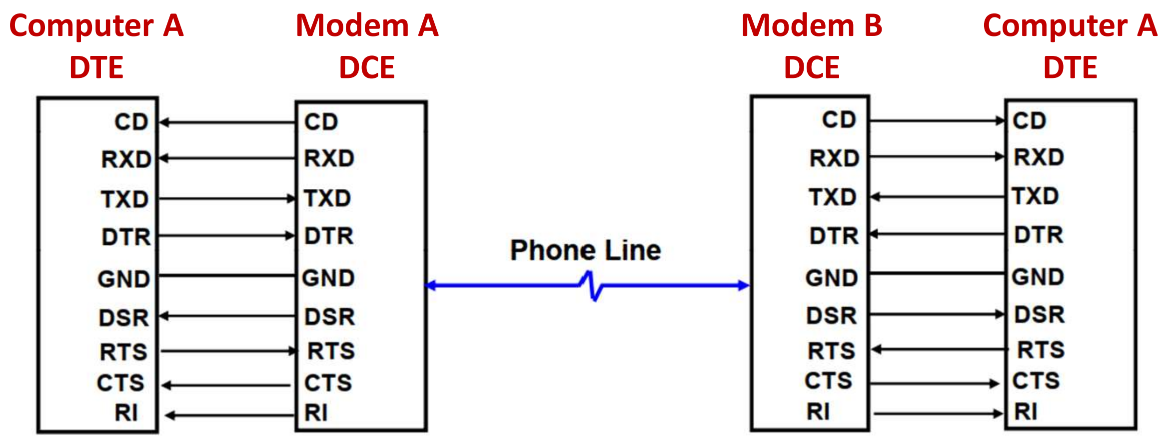

*Female DB9 connector*

# RS-232 standard

| Pin | Name | | Description |
|---|---|---|---|
| 1 | $\overline{\text{CD}}$ | Carrier Detect | DCE has detected a carrier tone |
| 2 | RXD | Received Data | Incoming data from DCE |
| 3 | TXD | Transmit Data | Outgoing data to DCE |
| 4 | DTR | Data Terminal Ready | DTE is connected and turned on |
| 5 | GND | Ground | |
| 6 | $\overline{\text{DSR}}$ | Data Set Ready | DCE is connected and turned on |
| 7 | $\overline{\text{RTS}}$ | Request To Send | DTE has data to send |
| 8 | $\overline{\text{CTS}}$ | Clear To Send | DCE can receive data |
| 9 | RI | Ring Indicator | Synchronized with the phone’s ringing tone |

- **Data Terminal Equipment (DTE) essentially refers to the computer.**
- **Data Communication Equipment (DCE) essentially refers to a remote device or modem.**
- **These terms are needed to explain the pin functions.**

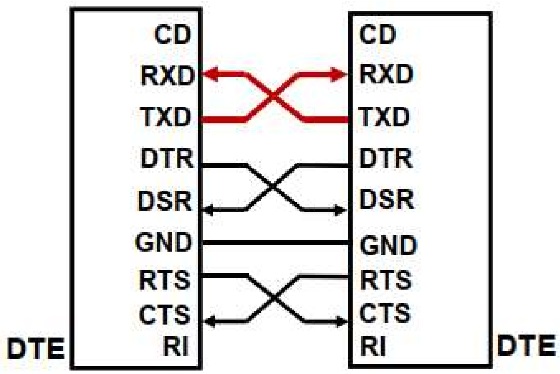
# Modem connection



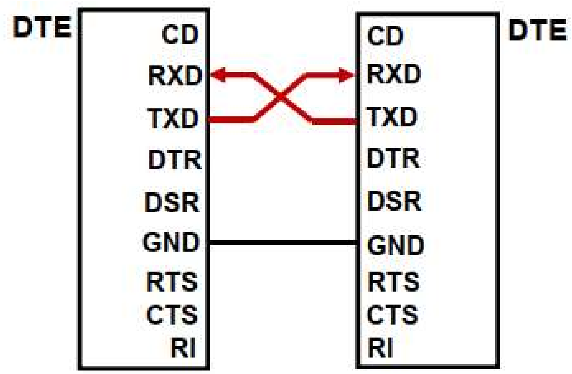

- **RS-2322 was originally used with modems to connect two PCs over the public phone lines**
- **When Computer A has data to send, it assert its RTS pin**
- **Modem A will assert its CTS when it is ready to receive**
- **Computer A transmits data through its TXD**

# Null-modem connection

**Full handshaking cable**




**Simple cable**

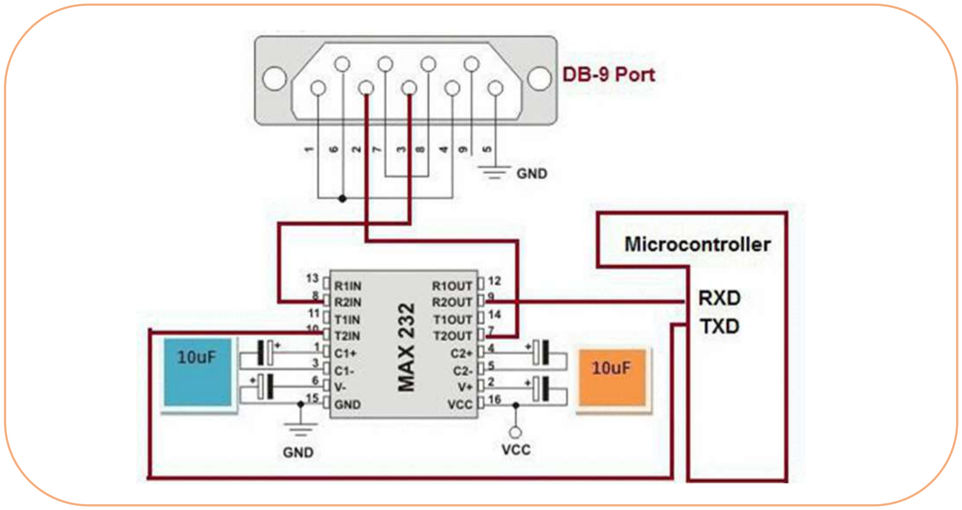


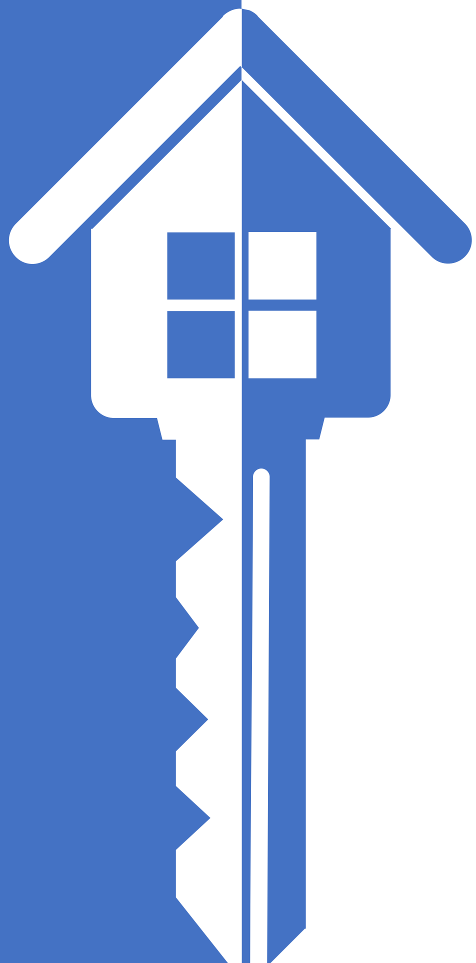

- **RS-232 is now mainly used to connect a microcontroller with PC or peripheral devices (e.g., GPS receiver, infrared range finder, camera).**
- **This configuration is known as null-modem:**
  - **Connect pin TXD of a DTE with pin RXD of the other DTE**
  - **Wire other pins to support flow control**

# RS-232 interface and MAX232 chip

- **Compared to TTL in computer electronics, RS-232 interface uses different voltage levels**
- **A level converter is required between RS-232 interface and TXD/RXD pins of microcontroller**
- **MAX232 chip is often used for this purpose**

| Logic | RS-232 levels | TTL levels |
|---|---|---|
| 1 | [-12V, -3V] | [+2V, +5V] |
| 0 | [+3V, +12V] | [0V, +0.8V] |





# Lecture 3's sequence

- **3.1** Serial communication – The basics
- **3.2** Serial communication in ATmega16
- **3.3** Example application & Debugging tool

# Serial communication in ATmega16

- **ATmega16 has 3 subsystems for serial communication:**
  - **Universal Synchronous & Asynchronous Receiver & Transmitter (USART)**
  - **Serial Peripheral Interface (SPI)**
  - **Two-wire Serial Interface (TWI)**

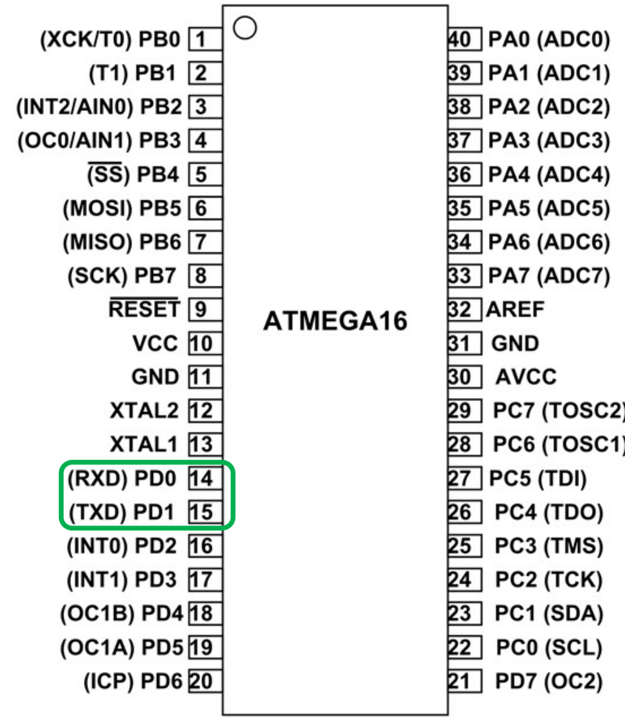
- **USART:**
  - **We focus on this subsystem in this lecture**
  - **Supports full-duplex mode between two devices**
  - **Typically used in asynchronous communication**
  - **Start bit and stop bit are used for each byte of data**

# Serial communication in ATmega16

- **Serial Peripheral Interface (SPI):**
  - The receiver and transmitter share a common clock line
  - The transmitter is treated as the master, the receiver as the slave
  - SPI supports higher data rates
  - **Examples:** liquid crystal display, high-speed ADC
- **Two-wire Serial Interface (TWI):**
  - For connecting several devices, e.g., microcontrollers and display boards, using a two-wire bus
  - Each device has a unique address and can exchange data with other devices in a small network
  - Up to 128 devices are supported

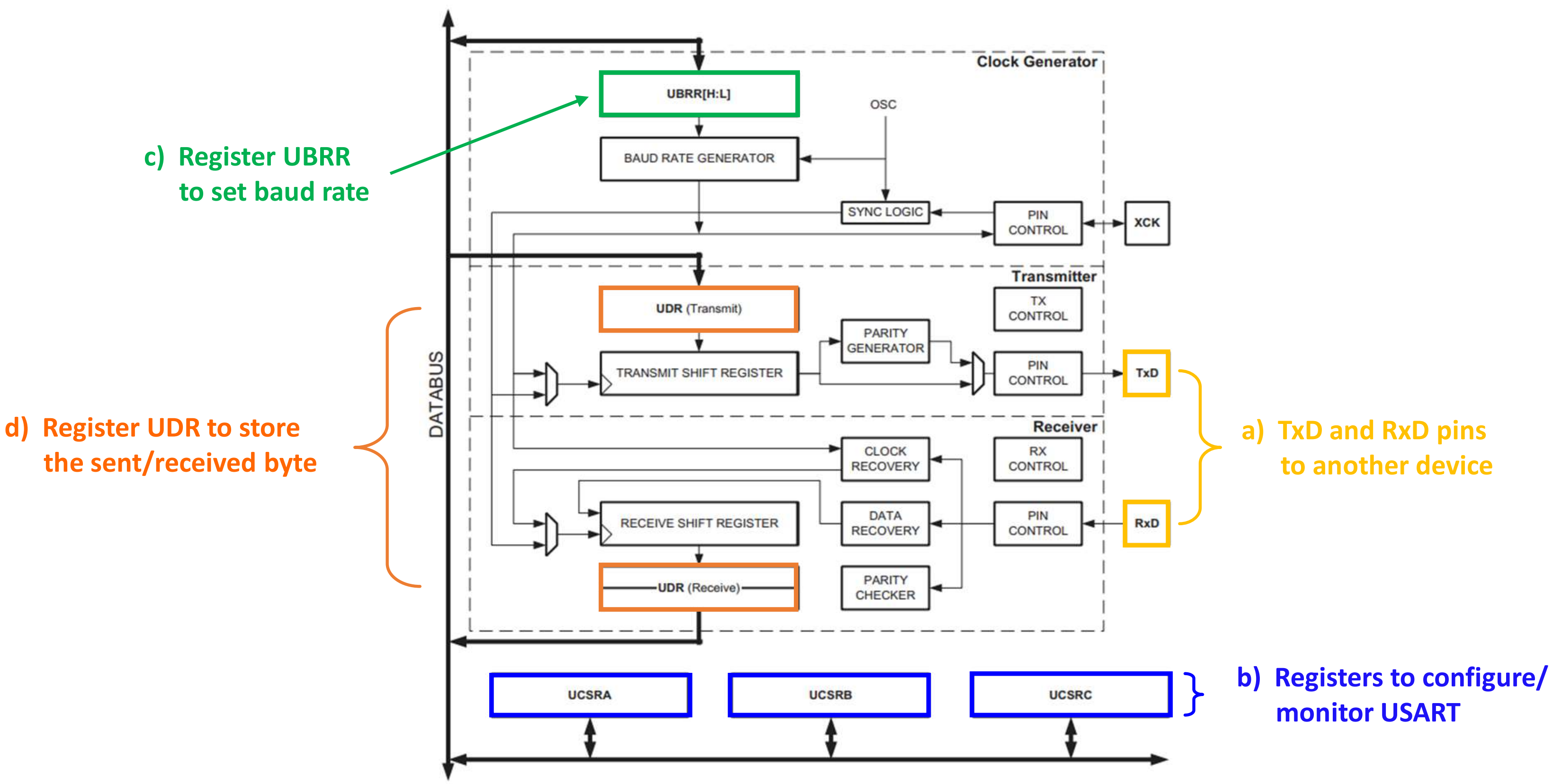
# Serial USART – An overview

- **USART of the ATmega16 supports:**
  - baud rates from 960bps to 57.6kbps
  - character size: 5 to 9 bits
  - 1 start bit
  - 1 or 2 stop bits
  - optional parity bit (even or odd parity)
- **Common baud rates are 19200, 9600, 4800, 2400, and 1200 bps**




ATmega16 chip

# Serial USART – An overview



# Serial USART – Hardware elements

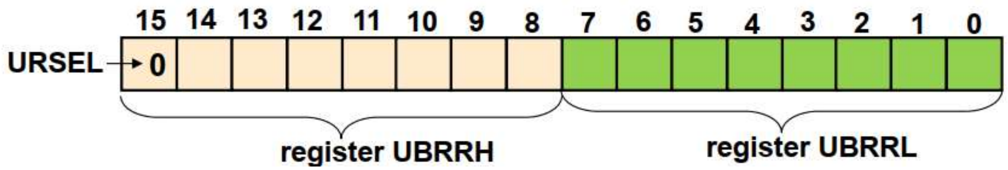
- **USART Clock Generator:**
  - to provide clock source
  - to set baud rate using UBRR register
- **USART Transmitter:**
  - to send a character through TxD pin
  - to handle start/stop bit framing, parity bit, shift register
- **USART Receiver:**
  - to receive a character through RxD pin
  - to perform the reverse operation of the transmitter
- **USART Registers:**
  - to configure, control and monitor the serial USART

# Serial USART – Three groups of registers

- **USART Baud Rate Registers:**
  - **UBRRH and UBRRL**
- **USART Control and Status Registers:**
  - **UCSRA**
  - **UCSRB**
  - **UCSRC**
- **USART Data Registers:**
  - **UDR**
- **Understanding these registers is essential in using the serial port. Therefore, we'll study these registers in depth.**

# USART Baud Rate Registers : UBRR

- **Two 8-bit registers together define the baud rate:**




$$\text{baud rate} = \frac{\text{system clock frequency (Hz)}}{16 \text{ x } (\text{UBRR} + 1)}$$

$$\Rightarrow \text{UBRR} = \frac{\text{system clock frequency (Hz)}}{16 \text{ x baud rate}} - 1$$

- **Example: Find UBRR registers for baud rate of 1200 bps assuming system clock is 1MHz:**
  - **UBRR = 1000000/(16 × 1200) – 1 = $51_d$ = $0033_H$**
  - **Therefore, UBRRH = $00_H$ and UBRRL = $33_H$**
  - **C code:** `UBRRH = 0x00; UBRRL = 0x33;`

# USART Control and Status Register A: UCSRA

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| RXC | TXC | UDRE | FE | DOR | PE | U2X | MPCM |

- 1 to enable multi-processor com mode
- 1 to double the transmission speed
- 1 when there is Parity Error
- 1 when there is Data Over Run
- 1 when there is Frame Error
- 1 when USART Data Register is Empty
- 1 when no new data in transmit buffer (TX Complete)
- 1 when receive buffer has unread data (RX Complete)

# USART Control and Status Register B: UCSRB

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| RXCIE | TXCIE | UDRIE | RXEN | TXEN | UCSZ2 | RXB8 | TXB8 |

- TXB8: Tx extra data bit for 9-bit character size
- RXB8: Rx extra data bit for 9-bit character size
- UCSZ2: bit UCSZ2 to decide character size
- TXEN: 1 to enable USART transmitter: Pin D.1 = TXD pin
- RXEN: 1 to enable USART receiver: Pin D 0 = RXD pin
- UDRIE: 1 to enable USART Data Register Empty Interrupt
- TXCIE: 1 to enable TX Complete Interrupt, valid only if Global Interrupt Flag = 1 and TXC = 1
- RXCIE: 1 to enable RX Complete Interrupt, valid only if Global Interrupt Flag = 1 and RXC = 1

# USART Control and Status Register C: UCSRC

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| URSEL | UMSEL | UPM1 | UPM0 | USBS | UCSZ1 | UCSZ0 | UCPOL |

- UCPOL: Clock polarity, used with synchronous
- UCSZ1, UCSZ0: Used with UCSZ2 to select character size
- USBS: To select stop bit modes: 0 -> 1 stop bit, 1 -> 2 stop bits
- UPM1, UPM0: To select parity mode: 00 no parity, 10 - even party, 11 - odd parity
- UMSEL: To select USART modes: 0 - asynchronous, 1 - synchronous
- URSEL: Must be set to 1 to write to UCSRC. (Note: UCSRC and UBRRH share same location)

# Setting character size

- **Character size (5 to 9) is determined by three bits: UCSZ2, UCSZ1, UCSZ0**

| RXCIE | TXCIE | UDRIE | RXEN | TXEN | UCSZ2 | RXB8 | TXB8 | UCSRB |
|---|---|---|---|---|---|---|---|---|
| URSEL | UMSEL | UPM1 | UPM0 | USBS | UCSZ1 | UCSZ0 | UCPOL | UCSRC |

| 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 |

# USART Data Register : UDR

- **Register UDR is the buffer for characters sent or received through the serial port.**

- **To start sending a character, we write it to UDR:**

```
UDR = 'a'; // start sending character ‘a’
```

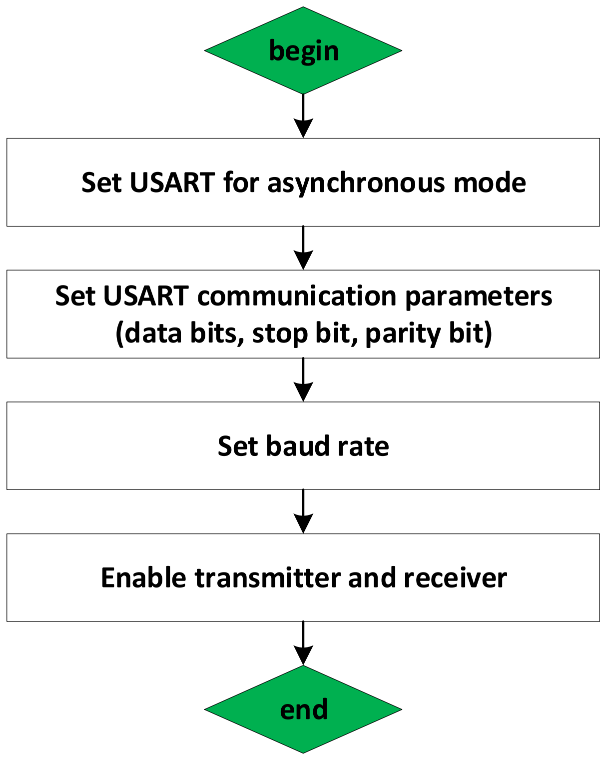
- **To process a received character, we read it from UDR:**

```
unsigned char data;
data = UDR; // this read will clear UDR
```

# Serial USART – Main tasks

- **There are 4 main tasks in using the serial port:**
  - **Initializing the serial port**
  - **Sending a character**
  - **Receiving a character**
  - **Sending/receiving a formatted string**

# Initializing serial port



# Initializing serial port – Example

**Initialize serial port of ATmega16 to baud rate 1200 bps, no parity, 1 stop bit 8 data bits. Assume a clock speed of 1MHz.**

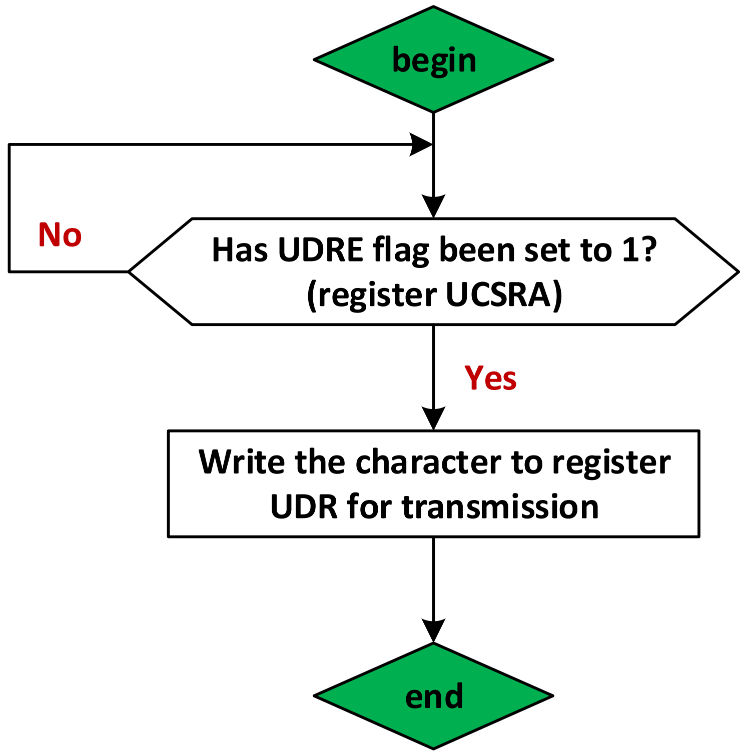
```
void USART_init(void) {
    // Asynchronous mode, no parity, 1 stop bit, 8 data bits
    UCSRC = 0b10000110;

    // Normal speed, disable multi-proc
    UCSRA = 0b00000000;

    // Baud rate 1200bps, assuming 1MHz clock
    UBRRL = 0x33;
    UBRRH = 0x00;

    // Enable Tx and Rx, disable interrupts
    UCSRB = 0b00011000;
}
```

# Sending a character



# Sending a character – Example

**Write a C function to send a character through ATmega16 serial port.**

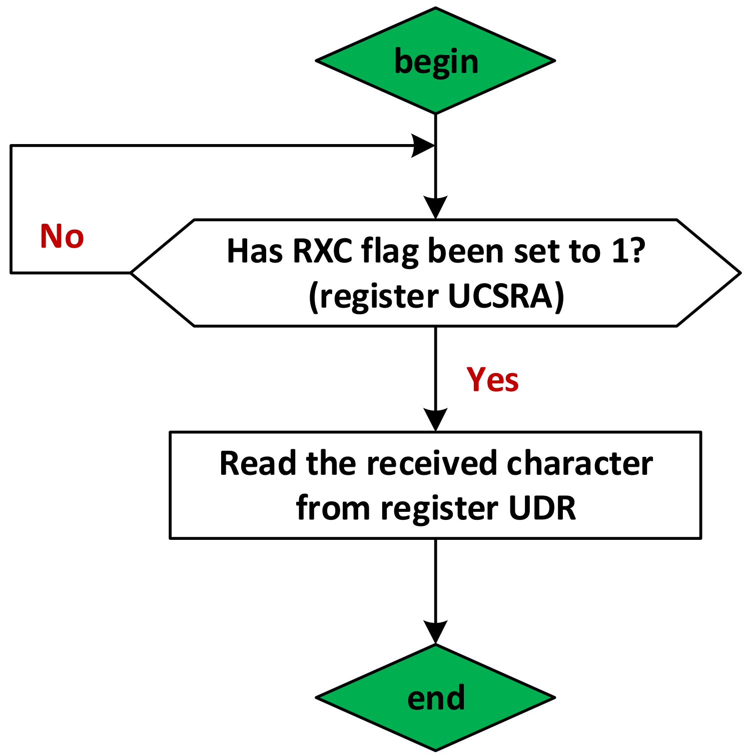
```
void USART_send(unsigned char data) {
     // Wait until UDRE flag = 1
     while ((UCSRA & (1<<UDRE)) == 0x00){;}
     // Write char to UDR for transmission
     UDR = data;
}
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| UCSRA | RXC | TXC | UDRE | PE | DOR | PE | U2X | MPCM |
| 1<<UDRE | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| bit-wise AND | 0 | 0 | UDRE | 0 | 0 | 0 | 0 | 0 |

Constant UDRE is defined in `avr/io.h`
`#define UDRE 5`

Bit-wise AND returns zero if bit at position UDRE is 0

# Receiving a character



# Receiving a character – Example

**Write a C function to receive a character via ATmega16 serial port.**

```
unsigned char USART_receive(void) {
     // Wait until RXC flag = 1
     while ((UCSRA & (1<<RXC)) == 0x00) {;}
     // Read the received char from UDR
     return (UDR);
}
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **UCSRA** | **RXC** | **TXC** | **UDRE** | **PE** | **DOR** | **PE** | **U2X** | **MPCM** |
| **1<<RXC** | **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **bit-wise AND** | **RXC** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |

**Constant RXC is defined in `avr/io.h`**

```
#define RXC 7
```

**Bit-wise AND returns zero if bit at position RXC is 0**

# Sending/receiving a formatted string

- In ANSI C, the header file *<stdio.h>* has two functions for formatted strings: *printf* and *scanf*.

- Function *printf* sends a formatted string to the standard output device, which is usually the video display.

```
unsigned char a, b;
a = 2; b = 3;
printf("first = %d, second = %d, sum = %d", a, b, a + b);
```

first = 2, second = 3, sum = 5

- Function *scanf* reads a formatted string from the standard input device, which is usually the keyboard.

```
unsigned char a, b;
scanf("%d %d", &a, &b); // get integers a, b from input string
```

# Sending/receiving a formatted strings

- **Being able to send/receive formatted strings through a serial port is useful in microcontroller applications.**
- **To this end, we configure the serial port as the standard input and output devices.**
- **General steps:**
  1) **Write two functions to send and receive a character via serial port.**
  2) **In *main()*, call *fdevopen()* to set the two functions as the handlers for standard output and input devices.**
  3) **Use *printf*/*scanf* as usual. Formatted strings will be sent/received via serial port.**

# Sending/receiving a formatted strings – Example

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

int serial_send(char c, FILE *stream) {
     // wait until UDRE flag is set to logic 1
     while ((UCSRA & (1<<UDRE)) == 0x00){;}
     UDR = c; // Write character to UDR for transmission
}

int serial_receive(FILE *stream) {
     // wait until RXC flag is set to logic 1
     while ((UCSRA & (1<<RXC)) == 0x00) {;}
     return (UDR); // Read the received character from UDR
}

int main(void) {
     unsigned char a;
     // ... Code to initialize baud rate, TXD, RXD, and so on is not shown here
     // Initialize the standard IO handlers
     stdout = fdevopen(serial_send, NULL);
     stdin = fdevopen(NULL, serial_receive);

     // Start using printf, scanf as usual
     while (1) {
          printf("\n\rEnter a = ");
          scanf("%d", &a); printf("%d", a);
     }
}
```

# Lecture 3's sequence

- **3.1** Serial communication – The basics
- **3.2** Serial communication in ATmega16
- **3.3** Example application & Debugging tool

# Example application: Programing

```c
#include <avr/io.h>
#include <stdio.h>
#include <util/delay.h>

int serial_int(void) {
     UCSRA = 0b00000010; // double speed, disable multi-proc
     UCSRB = 0b00011000; // enable Tx and Rx, disable interrupts
     UCSRC = 0b10000110; // asyn mode, no parity, 1 stop bit, 8 data bits
     // In double-speed mode, UBRR = Fclock/(8xbaud rate) – 1
     UBRRH = 0; UBRRL = 12; // baud rate 9600bps, assuming 1MHz clock
}

void serial_send(unsigned char data){
    while ((UCSRA & (1 << UDRE)) == 0){;} // wait until UDRE flag = 1
    UDR = data; // write character to UDR for transmission
}

int main(void) {
    serial_init(); // initialize USART

    while (1) {
        for (int i = 0; i < 10; i++) { // rotate left 10 times
            serial_send('4’);
            _delay_ms(1000);
        }
        for (int i = 0; i < 10; i++) { // rotate right 10 times
            serial_send('6’);
            _delay_ms(1000);
        }
    }
}
```

# Example application: Debugging

- **Sending/receiving data through serial port is useful for debugging.**
- **A program for monitoring serial data is Hyper Terminal:**
  **https://vnueduvn-my.sharepoint.com/:u:/g/personal/manhhv87_vnu_edu_vn/EZslb5W3Wf5IkZpI05Vm0scB0Bxdg0mToH-rIQgRADamMA?e=QdbCv8**
- **Hyper Terminal is used to:**
  - **create a serial connection between the PC and the microcontroller**
  - **send a text string to the microcontroller**
  - **receive a text string sent from the microcontroller**