



AMERICAN INTERNATIONAL UNIVERSITY-BANGLADESH (AIUB)

Where leaders are created

Lecture # 1 (Final)

Serial Communications Interfaces

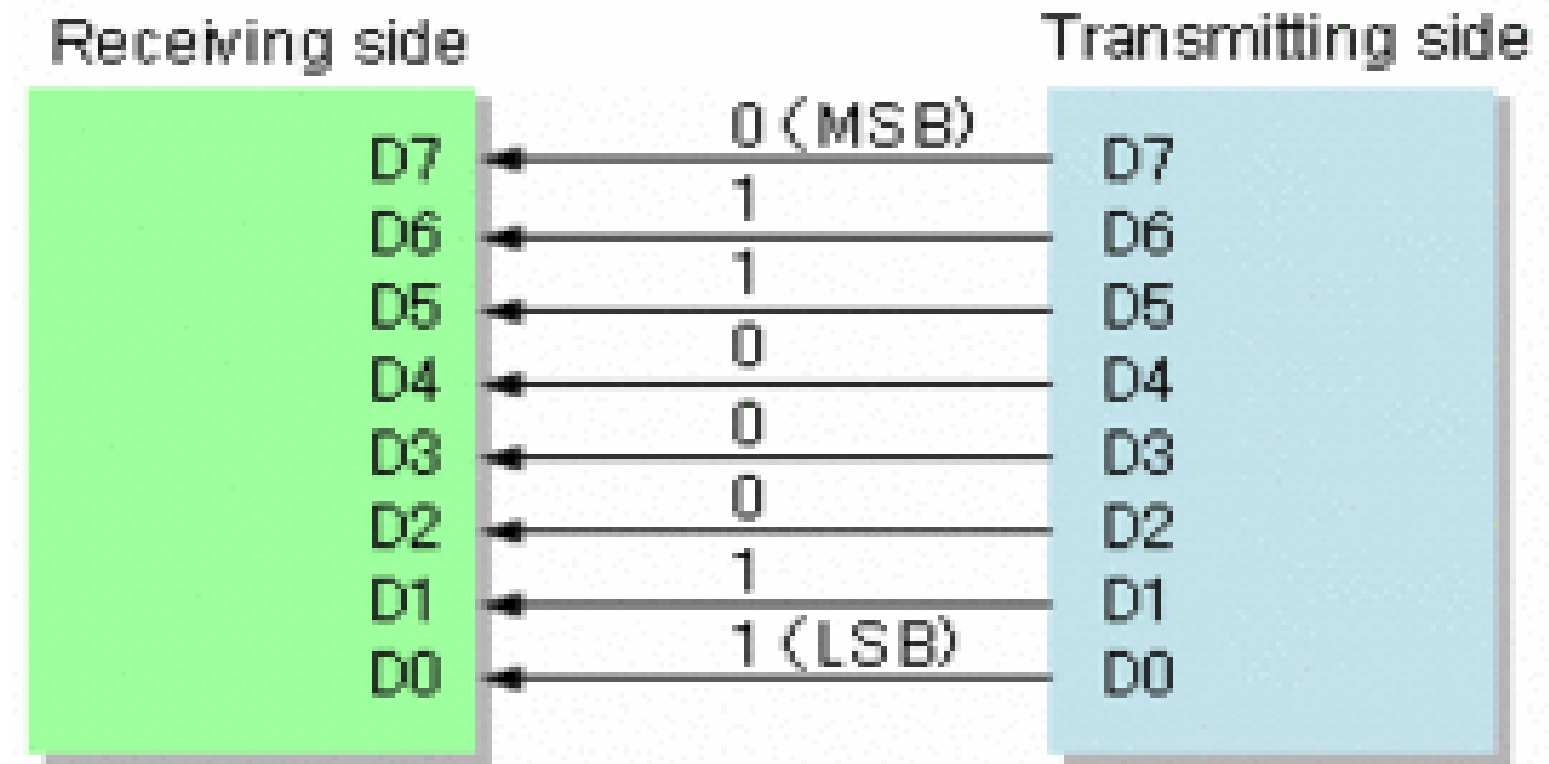
Prepared by: Ms. Tahmida Islam, Lecturer, EEE Department, AIUB
Modified by the Course Teacher: Prof. Dr. Engr. Muhibul Haque Bhuyan,
EEE Department, AIUB

Data Transmission

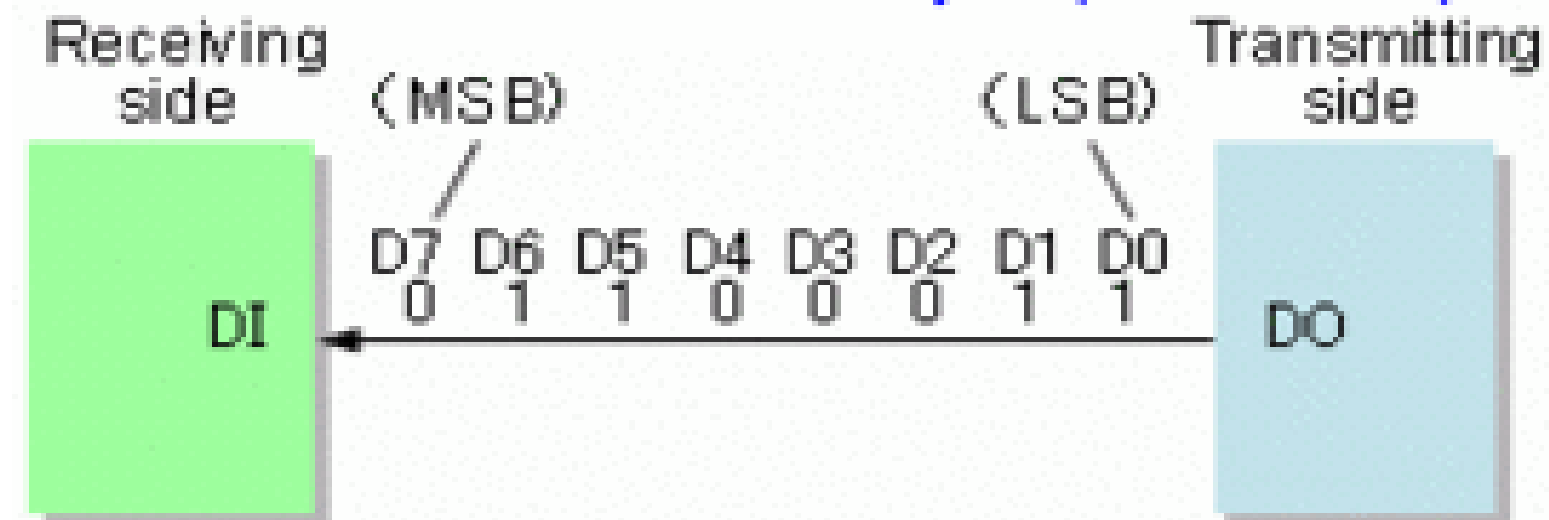
- Data transmission can be performed two ways.

- Parallel Communications**, where **several bits** of data are transmitted/received, on a link with **several parallel channels**.
- Serial Communications**, where data is transmitted/received **bit by bit** through **a single** channel.

Parallel interface example



Serial interface example (MSB first)



Serial Data Communication

❑ Advantage of serial communication:

- ***Smaller number of communication lines*** is required compared to parallel communication.
 - 2 lines (transmit & receive) are required in ***asynchronous full duplex*** serial comm.
 - 3 lines (transmit, receive & clock) are required in ***synchronous*** serial communication.


❑ Disadvantage of serial communication:

- ***More time*** is required to transmit/receive compared to parallel communication.

Types of Serial Communication

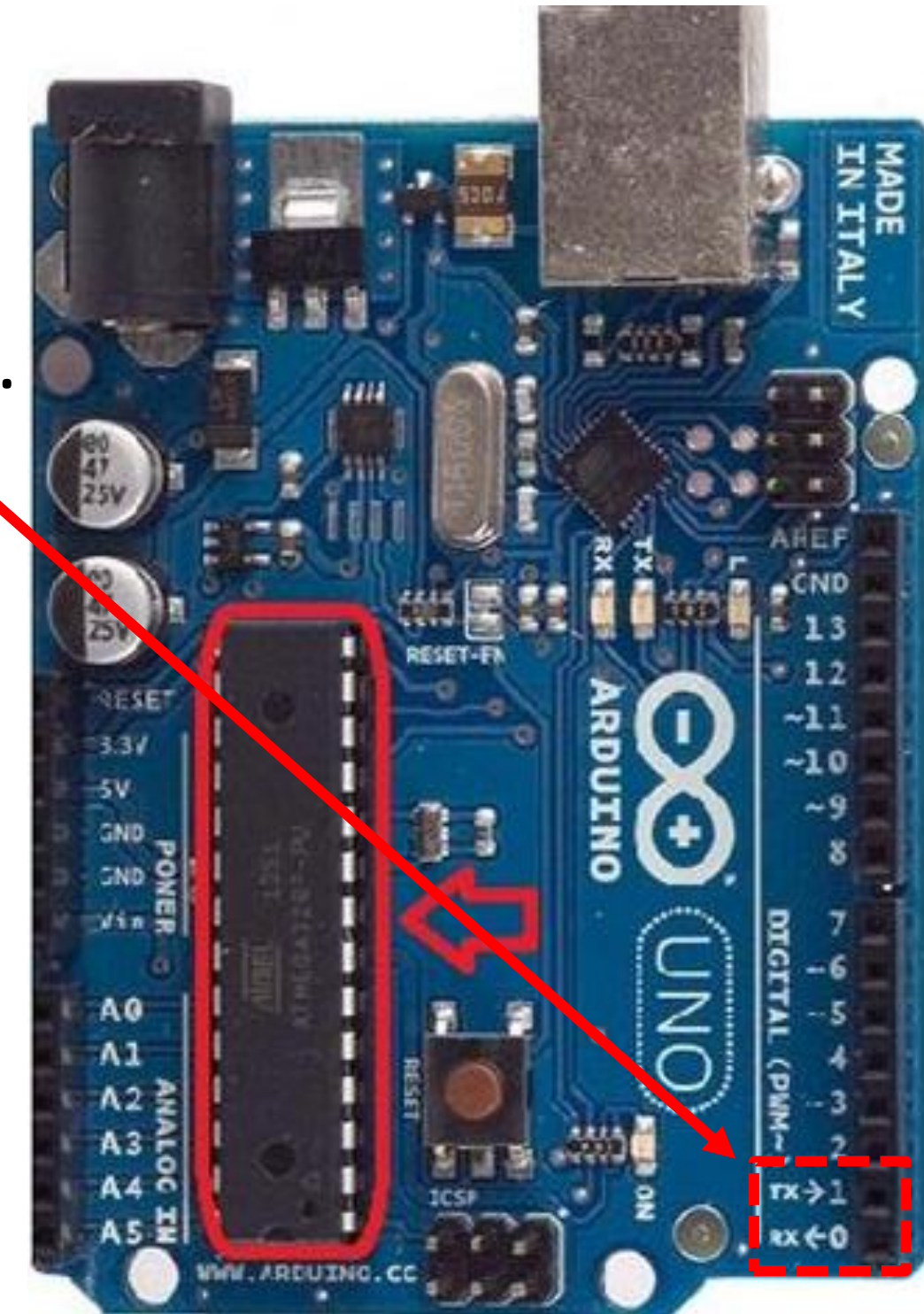
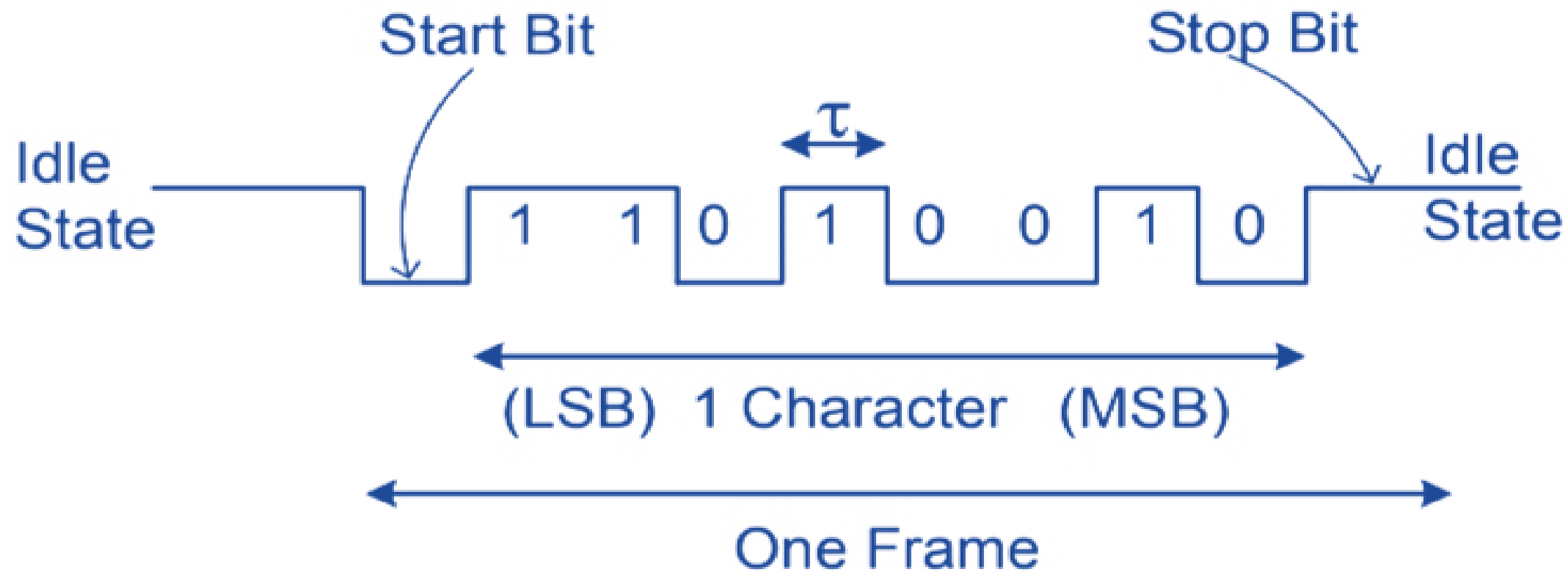
ATmega328 has 3 types of serial communication interfaces:

1. Universal Synchronous Asynchronous Receiver & Transmitter (USART).
2. Serial Peripheral Interface (SPI).
3. Two Wire Interface (TWI)/ Inter-Integrated Circuit (I2C).

 <div> <div>AMERICAN INTERNATIONAL UNIVERSITY- BANGLADESH</div> <div>Where leaders are created</div> </div>	<div>USART vs. UART</div>	<div>Microprocessor and Embedded Systems</div>
	UART	USART
Full Name	Universal Asynchronous Receiver/Transmitter	Universal Synchronous/Asynchronous Receiver/Transmitter
Data type and rate	It generates asynchronous data, hence has low data rate .	It generates clocked/synchronous data, hence has higher data rate .
Baud rate	Receiver need to know baud rate of the transmitter before communication to be established so that UART can generate clock internally and synchronize it with data stream with the help of transition of start bit.	Receiver need not be required to know the baud rate of the transmitter. This is derived from the clock signal and data line.
Data Structure	It uses start bit (before data word), stop bits (one or two, after data word), parity bit (even or odd) in its base format for data formatting.	USART can also generate data like UART. Hence USART can be used as UART, but reverse is not possible .
Protocol	UART is simple protocol to generate data.	USART is complex and uses many different protocols to generate the data for transmissions.
<div> <div>8 May 2025</div> <div>Course Teacher: Prof. Dr. Engr. Muhibul Haque Bhuyan</div> <div>5</div> </div>		

UART/USART

- It is an asynchronous/synchronous serial communication.
- It uses 2 pins in Port D:
 1. TXD/PD1 – The serial data transmission line.
 2. RXD/PD0 – The serial data reception line.
- Data is transmitted/received in a serial frame as follows:



USART

- Each bit is sent with a specific time duration τ , called **bit-time**. The smaller is τ , the faster is data transmission. The rate of data transmission/reception is called the **Baud rate**.
- **Standard Baud rates are:** 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200, 230400, ... bps
- In the ATmega328, the Baud rate is generated from internal clock. The Baud rates at the transmitter and receptor **must be the same** to avoid communication error.
- **The baud error should be $< \pm 2\%$ to avoid communication error.**
- **Baud error rate** =
$$\frac{\text{Standard baud rate} - \text{Calculated baud rate}}{\text{Standard baud rate}} \times 100\%$$

USART: Internal Clock Generation – The Baud Rate Generator

- Internal clock generation is used for the asynchronous and the synchronous master modes of operation.
- The **USART Baud Rate Register (UBRRn)** and the down-counter connected to it functions as a baud rate generator.
- The down-counter, running at **system oscillator clock frequency (f_{osc})**, is loaded with the **UBRRn value**, **each time the counter has counted down to zero**, thus a clock is generated.
- The **Transmitter divides the baud rate generator clock output by 2, 8, or 16** depending on mode.
- The **baud rate generator output is used directly by the Receiver's clock and data recovery units.**

Calculation of the Baud Rate

Table 17.1 Equations to calculate Baud Rate Register Setting

Operating Mode	Baud Rate Equations	Equations for UBRRn Values
Asynchronous Normal Mode (U2Xn = 0)	$Baud\ rate = \frac{f_{osc}}{16(UBRRn + 1)}$	$UBRRn = \frac{f_{osc}}{16 \times Baud\ rate} - 1$
Asynchronous Double Speed Mode (U2Xn = 1)	$Baud\ rate = \frac{f_{osc}}{8(UBRRn + 1)}$	$UBRRn = \frac{f_{osc}}{8 \times Baud\ rate} - 1$
Synchronous Master Mode	$Baud\ rate = \frac{f_{osc}}{2(UBRRn + 1)}$	$UBRRn = \frac{f_{osc}}{2 \times Baud\ rate} - 1$

Note:

1. The *Baud rate* is defined as the data transfer rate in bits per second (**bps**)
2. System oscillator clock frequency (*f_{osc}*) should be set in **Hz**
3. **UBRRn** means contents of the UBRRnH and UBRRnL registers, and their values may vary from 0 to 4095; there are 12 bits data, so the total values can be $2^{12} = 4096$.

Example to Practice

Find the baud rate for the three operating modes when $f_{osc} = 1$ MHz and $UBRRn = 25$. Calculate the baud error and comment whether there will be any communication error or not.

Solution:

For asynchronous normal mode:

$$\text{Baud rate} = \frac{f_{osc}}{16(UBRRn+1)} = \frac{1 \times 10^6}{16(25+1)} = 2404 \text{ bps}$$

$$\begin{aligned} \text{Baud error rate} &= \frac{\text{Standard baud rate} - \text{Calculated baud rate}}{\text{Standard baud rate}} \times 100\% \\ &= \frac{2400 - 2404}{2400} \times 100\% = -0.167\% < \pm 2\% \end{aligned}$$

So, there will be no communication error for the given information.

Continuation...

Solution:

For asynchronous double speed mode:

$$\text{Baud rate} = \frac{f_{osc}}{8(UBRRn+1)} = \frac{1 \times 10^6}{8(25+1)} = 4808 \text{ bps}$$

$$\begin{aligned} \text{Baud error rate} &= \frac{\text{Standard baud rate} - \text{Calculated baud rate}}{\text{Standard baud rate}} \times 100\% \\ &= \frac{4800 - 4808}{4800} \times 100\% = -0.167\% < \pm 2\% \end{aligned}$$

So, there will be no communication error for the given information.

Continuation...

Solution:

For synchronous master mode:

$$\text{Baud rate} = \frac{f_{osc}}{2(UBRRn+1)} = \frac{1 \times 10^6}{2(25+1)} = 19231 \text{ bps}$$

$$\begin{aligned} \text{Baud error rate} &= \frac{\text{Standard baud rate} - \text{Calculated baud rate}}{\text{Standard baud rate}} \times 100\% \\ &= \frac{19200 - 19231}{19200} \times 100\% = -0.161\% < \pm 2\% \end{aligned}$$

So, there will be no communication error for the given information.

Practice Example # 1

Compute the baud rate for the synchronous master operating mode when the oscillator frequency, $f_{OSC} = 16$ MHz, and register data is, $UBRRn = 010110101101$. **Compute** the baud error and comment on whether there will be any communication errors. Standard Baud rates are 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200, 230400, ... bps.

Solution:

$$UBRRn = 010110101101$$

$$= 0 \times 2^{11} + 1 \times 2^{10} + 0 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$= 0 + 1024 + 0 + 256 + 128 + 0 + 32 + 0 + 8 + 4 + 0 + 1 = 1453$$

For the asynchronous normal operating mode,

$$Baud\ Rate = \frac{f_{osc}}{2(UBRRn+1)} = \frac{16 \times 10^6}{2(1453+1)} = 5502\ bps$$

$$Baud\ Error\ Rate, \varepsilon = \frac{Standard\ baud\ rate - calculated\ baud\ rate}{Standard\ baud\ rate} \times 100\%$$

$$= \frac{4800 - 5502}{4800} \times 100\% = 14.625\%$$

This value is $\gg 2\%$, therefore, there will be communication errors.

Practice Example # 2

Compute the baud rate for the asynchronous double-speed mode when the oscillator frequency, $f_{OSC} = 16$ MHz, and the baud rate register data is, $UBRRn = 000110101110$. **Compute** the baud error and comment on whether there will be any communication errors. Standard Baud rates are 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200, 230400, ... bps.

Solution:

$$UBRRn = 000110101110$$

$$= 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

$$= 256 + 128 + 32 + 8 + 4 + 2 = 430$$

For the asynchronous double-speed mode,

$$\text{Baud Rate} = \frac{f_{osc}}{16(UBRRn+1)} = \frac{16 \times 10^6}{8(430+1)} = 4640 \text{ bps}$$

$$\text{Baud Error Rate, } \varepsilon = \frac{\text{Standard baud rate} - \text{calculated baud rate}}{\text{Standard baud rate}} \times 100\%$$

$$= \frac{4800 - 4640}{4800} \times 100\% = 3.33\%$$

This value is $> 2\%$, therefore, there will be communication errors.

USART- Arduino Libraries

- USART functions can be used with **Serial Monitor** of the Arduino.
 1. **serial.begin(baud)** – to enable input/output to serial monitor with baud speed or rate in bps. **Must be written in setup()**.
 2. **serial.available()** – Get the number of bytes (characters) available for reading from the serial port.
 3. **serial.println(val)** – to display **val** value to serial monitor **with newline** added.
 4. **serial.print(val)** – as above but **without newline**.
 5. **serial.print("Error")** – display message "Error" without **newline**.
 6. **serial.read()** – Reads incoming serial data.
- For other functions – please refer to [arduino.cc](https://www.arduino.cc).

USART- Arduino Libraries: Camera shutter speed example

- This example detects how long a camera shutter is open by using a change interrupt. At the first transition, it gets the time and at the second one, it gets the new time. Then the main loop shows the difference.
- This is tested down to a 50 μ s pulse, but it could probably go a bit shorter, as it takes around 5 μ s to enter and leave an ISR.

```
volatile boolean started;  
volatile unsigned long startTime;  
volatile unsigned long endTime;  
  
// interrupt service routine  
void shutter () {  
  if (started)  
    endTime = micros ();
```

```
else  
  startTime = micros ();  
started = !started; } // end of the shutter  
  
void setup () {  
  Serial.begin (115200);  
  Serial.println ("Shutter test ...");  
  attachInterrupt (digitalPinToInterrupt (2), shutter, CHANGE);  
} // end of the setup  
  
void loop () {  
  if (endTime) {  
    Serial.print ("Shutter open for ");  
    Serial.print (endTime - startTime);  
    Serial.println (" microseconds.");  
    endTime = 0; } // end of if statement  
} // end of the loop
```

Practice Example # 2

Compute the output for 20 seconds and the SPI clock frequency when the oscillator frequency, $f_{osc} = 16 \text{ MHz}$ for the following codes. **Determine** the slave state at the end of the program.

```
#include <SPI.h>

void setup (void) {
    Serial.begin(57600);
    digitalWrite(SS, HIGH);
    SPI.begin();
    SPI.setClockDivider(SPI_CLOCK_DIV8);
}

void loop (void) {
    char c;
    digitalWrite(SS, LOW);
    // send test string
    for (const char * p = "Hello, Engineers! \r" ; c = *p; p++) {
        SPI.transfer(c);
        Serial.print(c);
    }
    Serial.println("+++++");
    digitalWrite(SS, HIGH);
    delay(4000);
}
```

Practice Example # 2

Solution:

From the command in the code, `delay(4000);`

We find that the string sending repetition rate is 4 seconds per string. So, within 20 seconds, the string will be sent $20/4 = 5$ times. So, the serial monitor output will look as follows:

```
Hello, Engineers!+++++  
Hello, Engineers!+++++  
Hello, Engineers!+++++  
Hello, Engineers!+++++  
Hello, Engineers!+++++
```

Since “+++++” signs are in the `Serial.println()` function so, the next characters will be printed on the serial monitor in a new line.

From the command in the code, `Serial.begin(57600);`

We find that the baud rate is 57600 bps.

From the command in the code, `SPI.setClockDivider(SPI_CLOCK_DIV8);`

We find that the SPI clock frequency is $16/8 = 2$ MHz.

From the command in the code, `digitalWrite(SS, HIGH);`

We find that the slave state (SS) at the end of the program is HIGH. Therefore, the slave or controller device is deactivated after each cycle.

Advantages and Disadvantages of USART

❑ Advantages

- Hardware complexity is low.
- As this is one to one connection between two devices, software addressing is not required.
- Due to its simplicity, it is widely used in the **devices having 9 pin connectors**.

❑ Disadvantages

- It is suitable for **communication between only two devices**.
- It supports fixed data rate between devices wanting to communicate otherwise data will be **garbled (distorted,unclear)**.

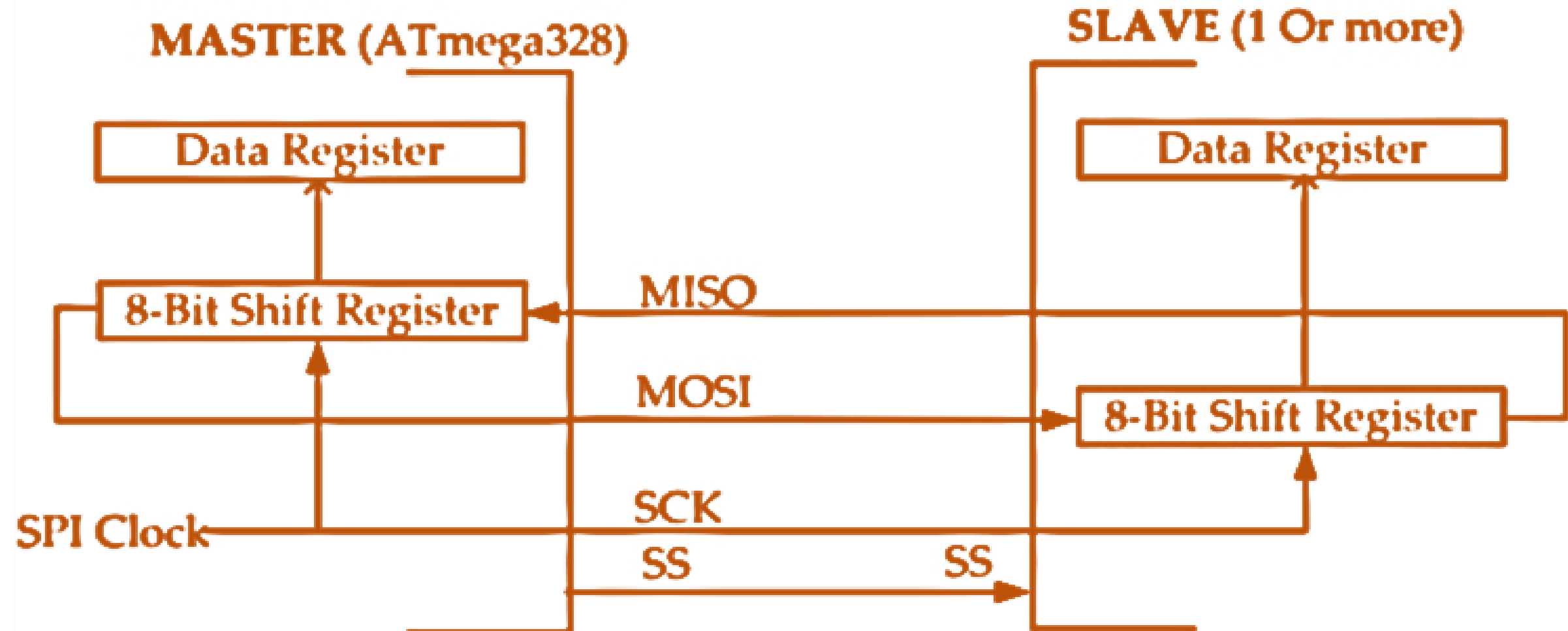
Serial Peripheral Interfaces (SPI)

- SPI is a synchronous data communication.
- SPI uses 4 pins in Port B:
- **SS/PB2** – **Slave (Peripheral)** Selection pin, this pin on each peripheral enables the Master to enable and disable a slave or peripheral device.
- **MOSI/PB3** – **Master (Controller)** Out Slave In, the Master line for sending data to the **Peripherals**, this pin enables the Master to drive a slave.
- **MISO/PB4** – Master In Slave Out, the Slave line for sending data to the master, this pin enables the Master to receive any slave data.
- **SCK/PB5** – The clock pulses which synchronize data transmission generated by the **Master (Controller)**.

19	<input type="checkbox"/>	PB5 (SCK/PCINT5)
18	<input type="checkbox"/>	PB4 (MISO/PCINT4)
17	<input type="checkbox"/>	PB3 (MOSI/OC2A/PCINT3)
16	<input type="checkbox"/>	PB2 ($\overline{\text{SS}}$ /OC1B/PCINT2)

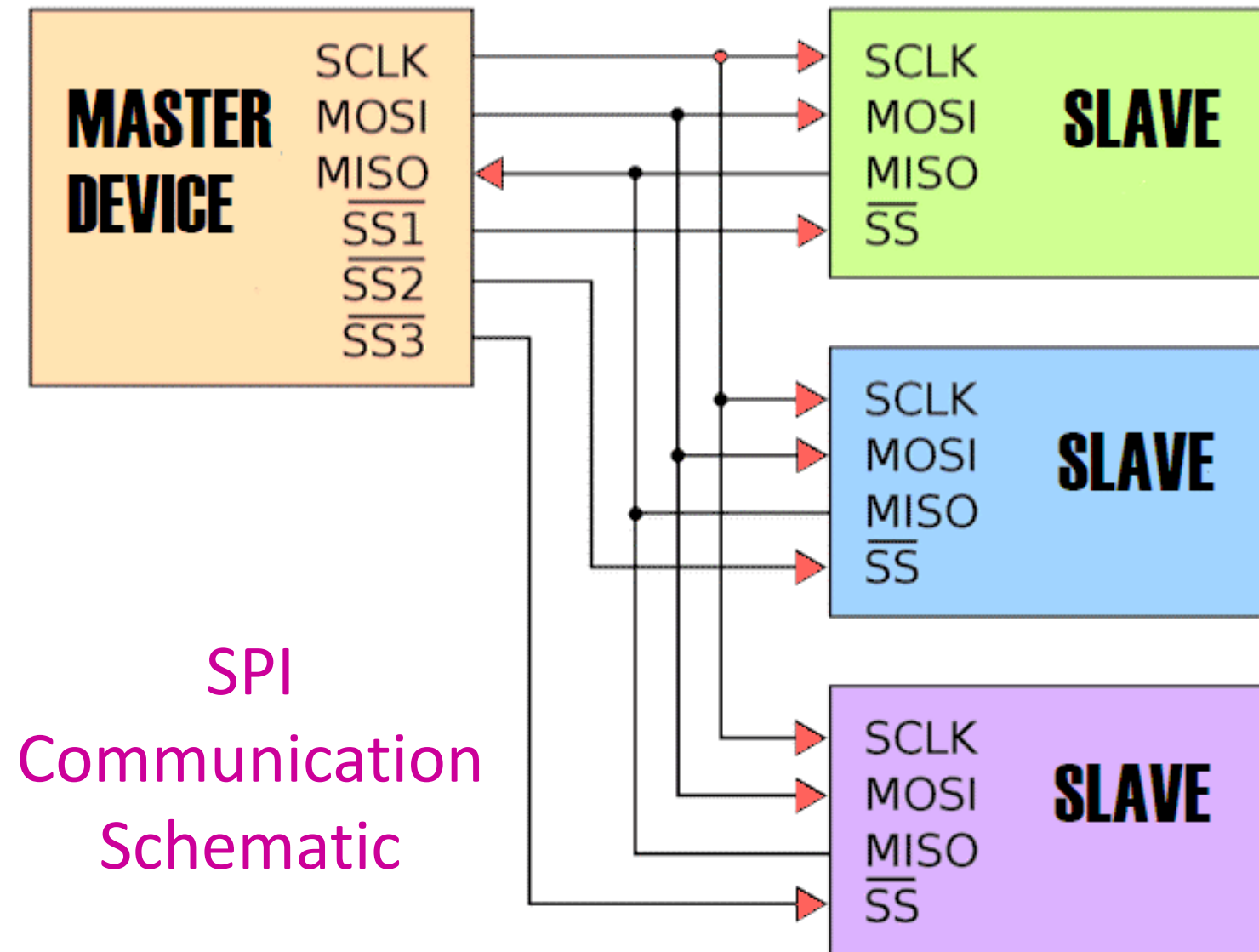
Serial Peripheral Interfaces (SPI)

- Connection using SPI is in the Master-Slave configuration.
- **Master** – Normally, is the ATmega328. **Master (Controller) initiates** the data transfer. SPI clock is also generated by master.
- **Slave** – Consists of 1 or more SPI I/O **Slaves (Peripherals)**. The slave transfers data as a **reaction** to master.



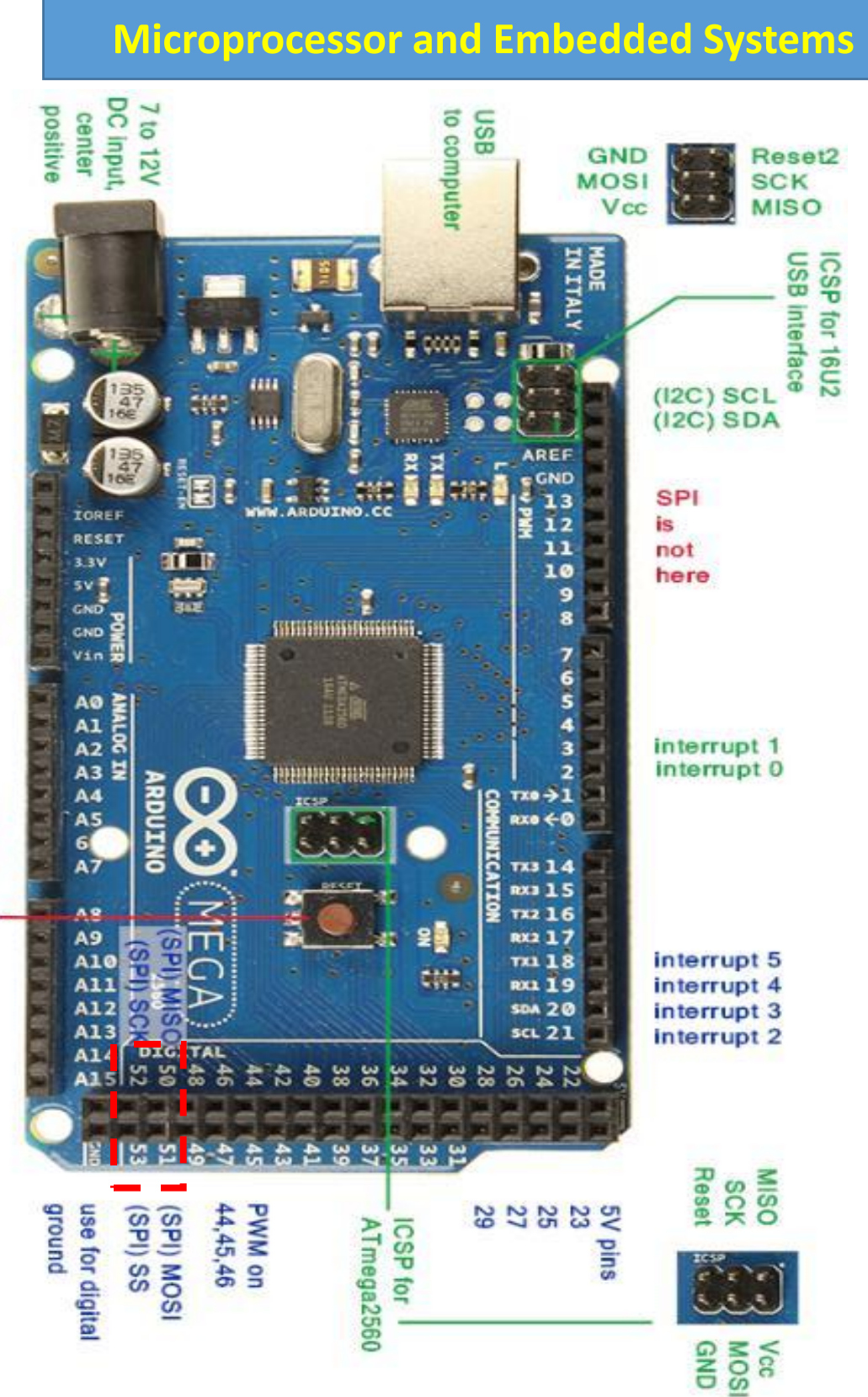
Serial Peripheral Interfaces (SPI)

- Connection using SPI is in the **Master (Controller)-Slave (Peripheral)** configuration.



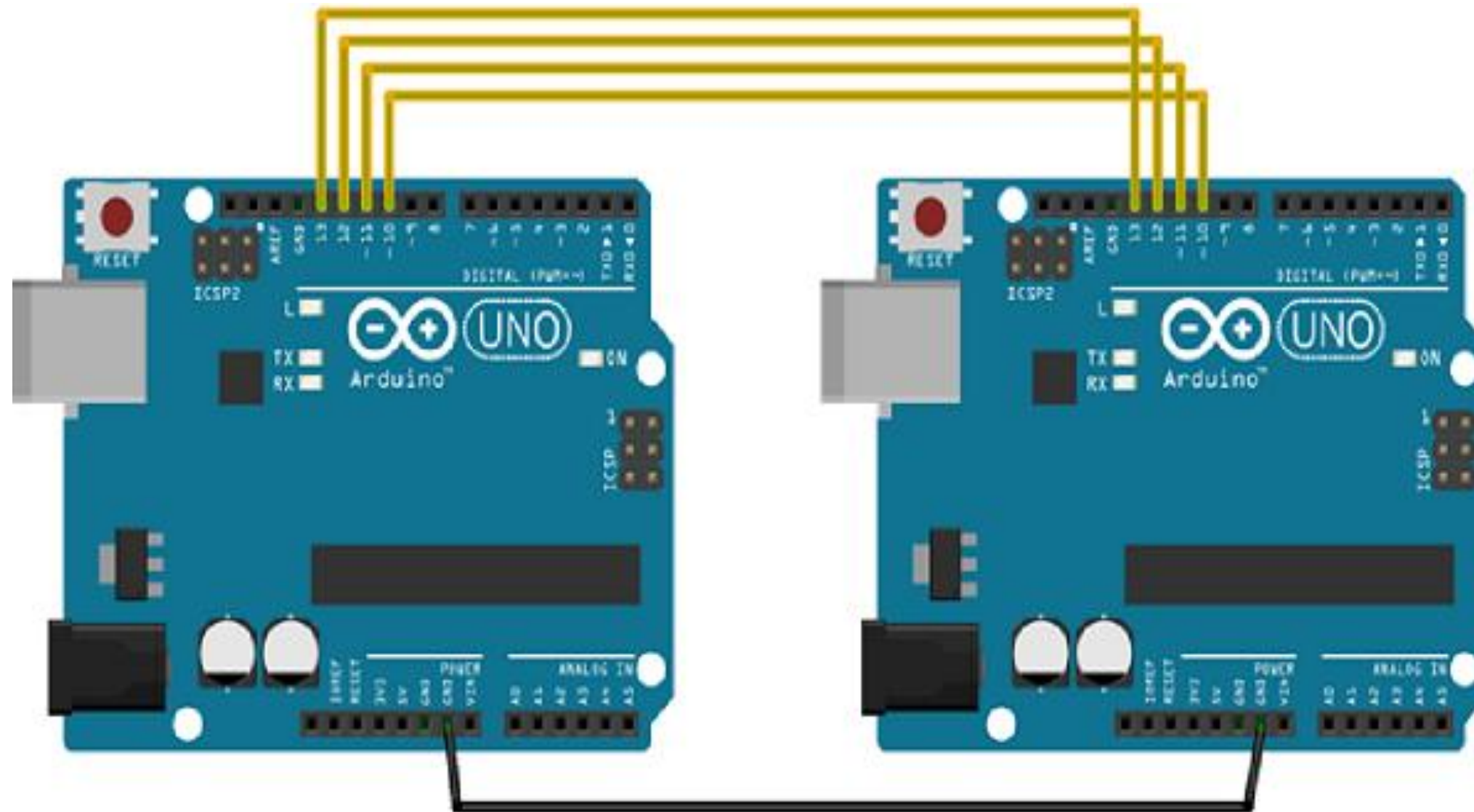
SPI
Communication
Schematic

SPI
Communication
Schematic Ports
on Arduino
Mega Board



Serial Peripheral Interfaces (SPI)

- Let's make an example with Arduino. In this example, we are going to let the two Arduinos to communicate with each other.



Pin connections of these two Arduinos

We will connect two **Arduino UNO** boards; one as a master and the other as a slave.

- (SS): pin 10; **Slave Selection**
 - (MOSI): pin 11
 - (MISO): pin 12
 - (SCK): pin 13
- Communication**

- 19 □ PB5 (SCK/PCINT5)
- 18 □ PB4 (MISO/PCINT4)
- 17 □ PB3 (MOSI/OC2A/PCINT3)
- 16 □ PB2 ($\overline{\text{SS}}$ /OC1B/PCINT2)

SPI Arduino Libraries

- **SPI.begin()** – Initializes the SPI bus by setting SCK, MOSI, and SS to outputs, set SCK & MOSI low, & SS high. Must be written in **setup()**
- **SPI.end()** – Disables the SPI bus.
- **SPI.setBitOrder(order)** – Sets the order of the bits shifted out of and into the SPI bus, either LSBFIRST or MSBFIRST.
- **SPI.setClockDivider(divider)** – Sets the SPI clock divider (**SPI_CLOCK_DIVn**, $n = 2, 4, 8, 16, 32, 64, \text{ or } 128$). The **default setting** is **SPI_CLOCK_DIV4**, which sets the **SPI clock to 4 MHz** for Uno
- **SPI.setDataMode(mode)** – Sets the SPI data mode: clock polarity and phase. **Available modes: SPI_MODE0 – SPI_MODE3**. refer to arduino.cc
- **SPI.transfer(val)** – Transfers **one byte** over the SPI bus, both sending and receiving. **val: the byte to send out**.
- **Returns:** the byte read from the bus.
- **SPI.beginTransaction(SPISettings(speedMaximum, dataOrder, dataMode))** – speedMaximum is the clock, dataOrder(MSBFIRST or LSBFIRST), dataMode(SPI_MODE0, SPI_MODE1, SPI_MODE2, or SPI_MODE3).

SPI Examples

SPI as MASTER

```
#include <SPI.h>

void setup (void) {
    Serial.begin(115200); //set baud rate to 115200 for USART
    digitalWrite(SS, HIGH); // disable Slave Select
    SPI.begin ();
    SPI.setClockDivider(SPI_CLOCK_DIV8); //divide the clock by 8
}

void loop (void) {
    char c;
    digitalWrite(SS, LOW); // enable Slave Select
    // send test string
    for (const char * p = "Hello, world!\r" ; c = *p; p++) {
        SPI.transfer (c);
        Serial.print(c);
    }
    digitalWrite(SS, HIGH); // disable Slave Select
    delay(2000);
}
```

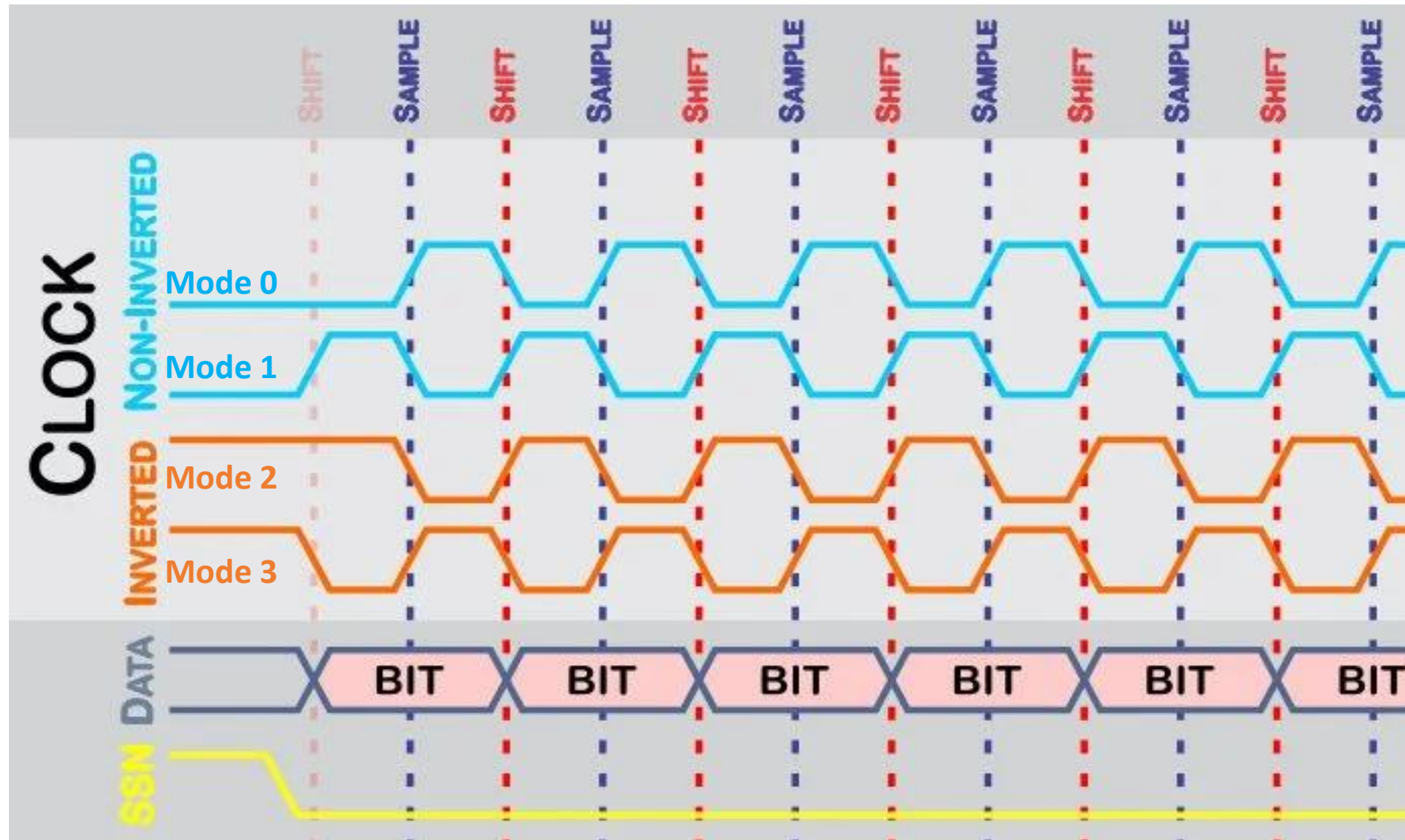
SPI as SLAVE

Microprocessor and Embedded Systems

```
#include <SPI.h>
char buff [50];
volatile byte indx;
volatile boolean process;
void setup (void) {
    Serial.begin (115200);
    pinMode(MISO, OUTPUT); // have to send on master in so it set as output
    SPCR |= _BV(SPE); // turn on SPI in slave mode
    indx = 0; // buffer empty
    process = false;
    SPI.attachInterrupt(); // turn on interrupt
}
ISR (SPI_STC_vect) // SPI interrupt routine {
    byte c = SPDR; // read byte from SPI Data Register
    if (indx < sizeof buff) {
        buff [indx++] = c; // save data in the next index in the array buff
        if (c == '\r') //check for the end of the word
            process = true;
    }
}
void loop (void) {
    if (process) {
        process = false; //reset the process
        Serial.println (buff); //print the array on serial monitor
        indx= 0; //reset button to zero
    }
}
```

SPI Modes

There are **four modes (0, 1, 2, 3)** of operation in SPI. These modes correspond to the four possible clocking configurations. Clock transitions govern the shifting and sampling of data.



CPOL: Clock Polarity: This governs the initial logic state of the clock signal.

CPHA: Clock Phase: This governs the relationship between the data transitions and the clock transitions.

Bits that are sampled on the rising edge of the clock cycle are shifted out on the falling edge of the clock cycle, and vice versa.

SPI Modes

Each transaction begins when the slave-select line is driven to logic LOW (slave select is typically an active-LOW signal). The exact relationship between the slave-select, data, and clock lines depends on how the **Clock Polarity (CPOL)** and **Clock Phase (CPHA)** are configured.

With the **Non-inverted Clock Polarity** (i.e., the clock is at logic **LOW** when slave select transitions to logic LOW):

- **Mode 0 (the default)** – Clock is normally LOW (**CPOL = 0**), and the data is sampled on the transition from LOW to HIGH (leading/rising edge) and shifted out on the trailing/falling edge of the clock pulse. (CPHA = 0).
- **Mode 1** – Clock is normally LOW (**CPOL = 0**), and the data is sampled on the transition from HIGH to LOW (trailing/falling edge) and shifted out on the (leading/rising edge of the clock pulse (CPHA = 1).

SPI Modes

With the **Inverted Clock Polarity** (i.e., the clock is at logic **HIGH** when slave select transitions to logic LOW):

- **Mode 2** – Clock is normally HIGH (**CPOL = 1**), and the data is sampled on the transition from HIGH to LOW (trailing/falling edge) and shifted out on the leading/rising edge of the clock pulse (CPHA = 0).
- **Mode 3** – Clock is normally HIGH (**CPOL = 1**), and the data is sampled on the transition from LOW to HIGH (leading/rising edge) and shifted out on the trailing/falling edge of the clock pulse. (CPHA = 1).
- **SPI.attachInterrupt(handler)** – Function to be called when a slave device receives data from the master.

SPI Hardware Structures

The SPI Control Register (SPCR) has 8 bits, each bit position may take values.

7	6	5	4	3	2	1	0
SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

- **SPIE** - Enables the SPI interrupt when 1
- **SPE** - Enables the SPI when 1
- **DORD** - Sends the data and decides the order of data: Least (LSB) and Most (MSB) Significant Bit first when 1 and 0, respectively
- **MSTR** - Sets the Arduino in Master mode when 1, Slave mode when 0
- **CPOL** - Sets the data clock to be idle when high if set to 1, idle when low if set to 0
- **CPHA** - Samples the data on the clock's falling edge when 1, rising edge when 0
- **SPR1 and SPR0** - Sets the SPI speed: 00 = fastest (4 MHz), 11 = slowest (250 kHz)

SPI Examples

SPI as MASTER

```
//SPI Master Device
//We need to import SPI.h library first
#include <SPI.h>
//Our Slave Selection pin
#define SlaveSelection 10

int count = 0;

void setup()
{
    //Set SlaveSelection pin as output.
    pinMode(SlaveSelection, OUTPUT); //and Make it
    //HIGH to prevent to start communication right away
    digitalWrite(SlaveSelection, HIGH); //Start the SPI
    //communication.
    SPI.begin();
}
```

SPI as MASTER

Microprocessor and Embedded Systems

```
void loop() {
    for(count=0; count<255; count++){
        sendSerialData(count, SlaveSelection);
        delay(2000);
    }
    delay(500);
}

void sendSerialData(char data, int SlaveSelection) {
    //Enable slave Arduino with setting the Slave Selection pin to 0 V
    digitalWrite(SlaveSelection, LOW);
    // Wait for a moment
    delay(10);
    //We sent the data here and wait for the response from device
    char receivedValue = SPI.transfer(data);
    //And then write the answer to the serial port
    Serial.println(receivedValue);
    //Disable slave Arduino with setting the Slave Selection pin to 5 V
    digitalWrite(SlaveSelection, HIGH);
}
```

SPI Examples

SPI as SLAVE

```
//Slave device of the SPI communication
#include <SPI.h>

char i = 0;

#define SlaveSelection 10

void setup() {
  //Start the Serial Communication
  Serial.begin(9600);
  // initialize SPI :
  pinMode(SlaveSelection , INPUT); // Set Slave Selection as input
  pinMode(13,OUTPUT); // Set clock as output
  pinMode(11,OUTPUT); // Set MOSI as output
  pinMode(12,INPUT); // Set MISO as input
  // SPCR - SPI Control Register
  // According to the structure of table we, enable the SPI and Interface
  SPCR |= 0b11000000;
  // SPSR - SPI Status Register
  SPSR |= 0x00;
}
```

SPI as SLAVE

Microprocessor and Embedded Systems

```
void loop()
{
  delay(1000);
}

//SPI Interrupt function

ISR(SPI_STC_vect) {
  //Here we read the SPI lines, this line will check data for every ASCII codes
  //for 8-bit received data SPDR -> SPI Data Read bit
  SPDR = i;
  i ++;
  if ( i > 255)
    i = 0;
  while(!(SPSR & (1 << SPIF)));
  //Load the received data to the variable
  char received = SPDR;
  //And send it to the serial communication bus
  Serial.println(received);
}
```

Advantages and Disadvantages of SPI

□ Advantages

- It is a simple protocol and hence does not require processing overheads.
- Supports **full duplex** communication.
- Due to the separate use of CS lines, the same kind of multiple chips can be used in the circuit design.
- SPI uses push-pull and hence higher data rates, and longer ranges are possible.
- SPI uses less power compared to I2C

□ Disadvantages

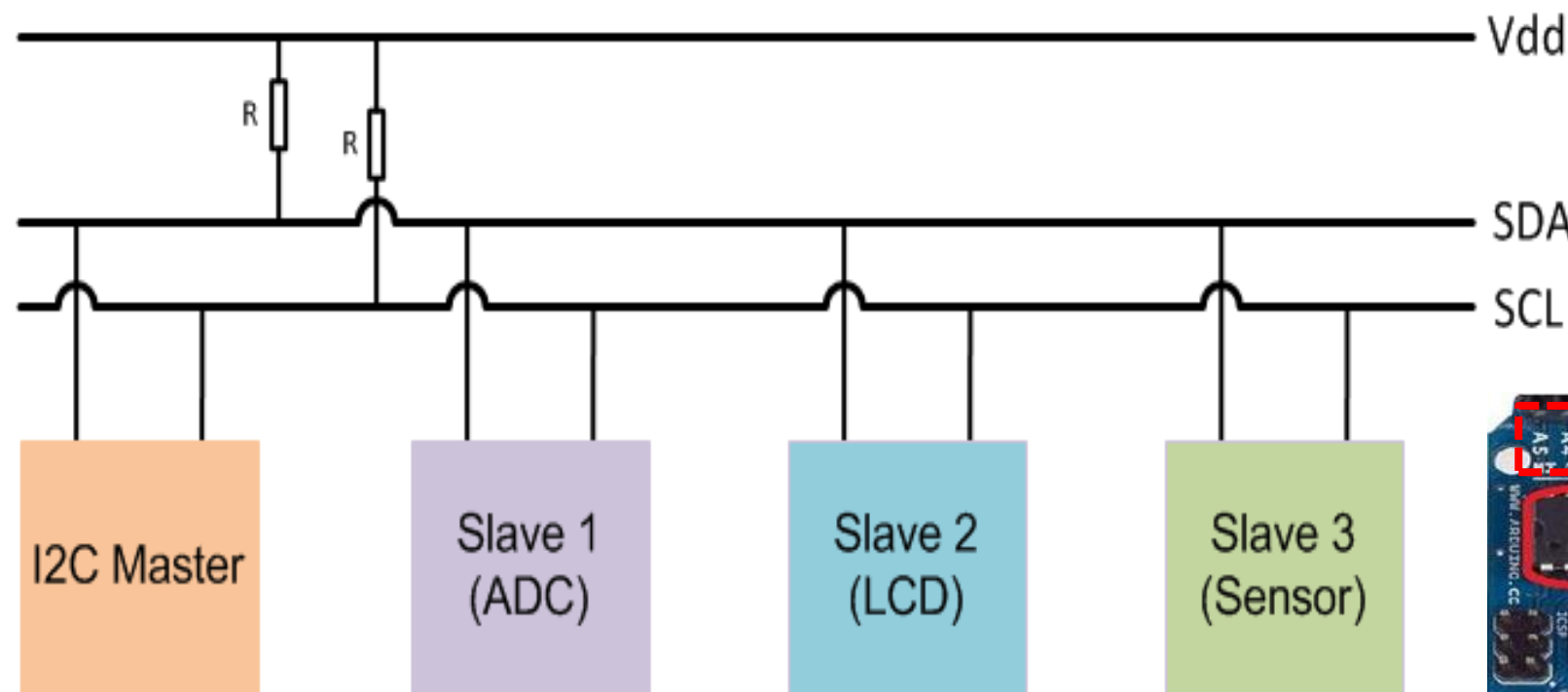
- As the number of slaves increases, the number of CS lines increases, this results in **hardware complexity** as the number of pins required will increase.
- To add a device in SPI requires one to **add an extra CS line** and changes in software for the particular device addressing is concerned.
- Master and slave relationship can not be changed as usually done in the I2C interface.
- No flow control available in SPI.

I2C (Inter-Integrated Circuit): What is it?

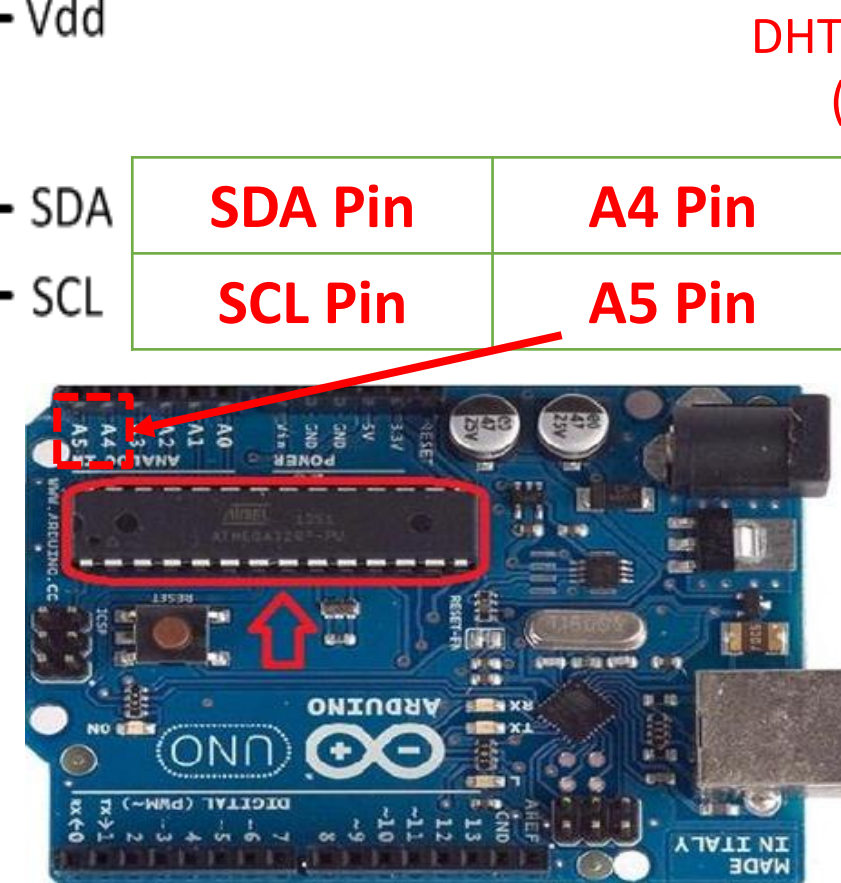
- An inter-integrated circuit (I2C) or two-wire interface (TWI) is a synchronous serial protocol originally developed by Philips Semiconductors (now NXP).
- It's a **multi-master, multi-slave serial bus for low-speed devices** that only requires two wires among **multiple devices**. It can easily be implemented with two digital input/output channels on a device.
- An I2C bus has **just two wires** over which hundreds of devices communicate serially.
- As a **master-slave type communication** standard, **at least one device** connected to the bus should be the **master** that generates a clock signal for synchronous serial data communication.
- The **slave devices can transfer data** to and from the master device(s), which access slave devices by their **I2C addresses**. The address of each slave device on an I2C bus must be unique. The **I2C slave devices still must obtain their addresses from NXP**.

I2C (Inter-Integrated Circuit)

- A chip-to-chip protocol for communicating with low-speed peripherals.
- The I2C bus drivers are **open drain**, which means the devices can pull the I2C signal line low but cannot drive it high. **By default, both the lines are pulled high** by pull-up resistors **until the bus is accessed by a master device** to avoid **bus contention**.

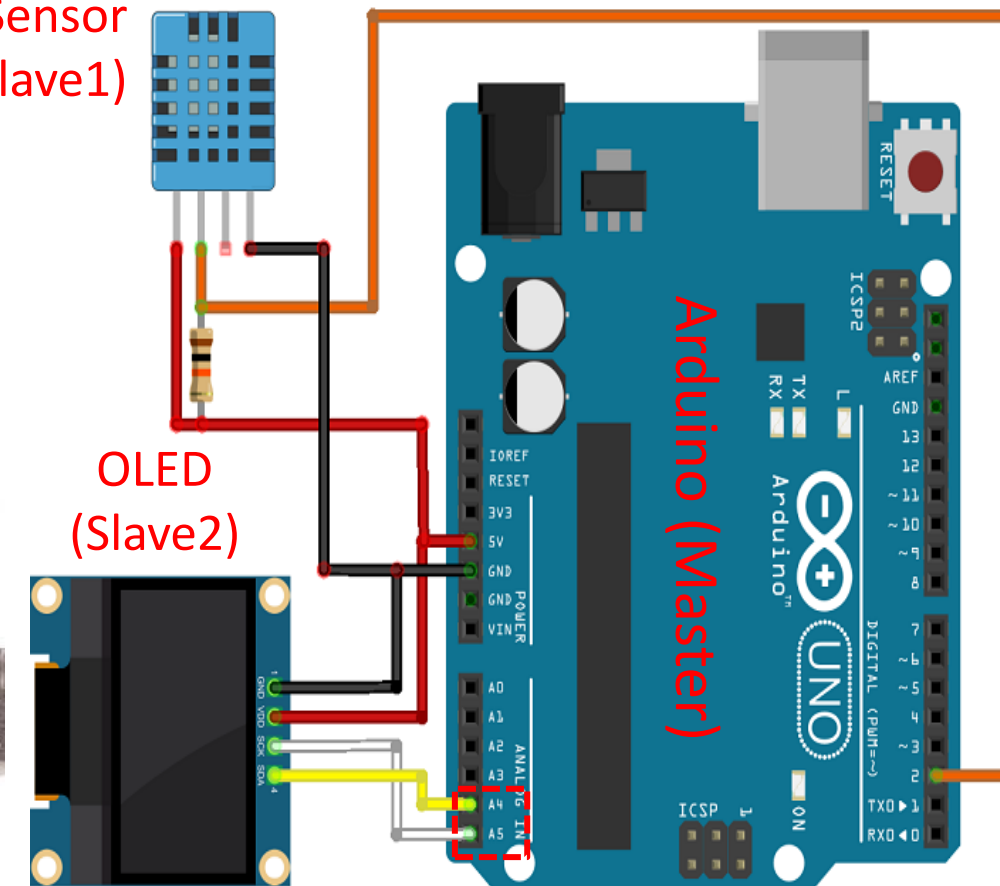


Block Diagram of an I2C protocol for communicating with multiple low-speed peripheral devices (Slaves) using a single controller device (Master)



DHT Sensor (Slave1)

OLED (Slave2)



I2C (Inter-Integrated Circuit)

- I2C is another serial protocol for **two-wire interface** to connect to **low-speed devices** like Micro-controller, EEPROMs, I/O Interfaces, and other similar devices used in embedded systems.
- **I2C is a bus** for communication between a master (or can be multiple masters) and a single or multiple slave devices.
- I2C uses only **two wires- SCL (Serial Clock) and SDA (Serial Data)**.
- **SCL (Serial Clock)**: The clock line used to **synchronize all data transfers** over the I2C bus, the line over which **master device(s)** generate the clock signal.
- **SDA (Serial Data)**: The data line used to **transmit the data** between devices, the line over which the **master and slave devices** communicate serial data
- Each I2C Slave devices have a 7-bit/10-bit addressing.
- The **data transfer rate depends on the clock frequency**. In the **standard mode**, the clock frequency is **100-400 kHz with 7 bit addressing** and **data transfer of 100 kbps**.

I2C Addresses

- The clock frequencies for the following three modes are with **10-bit addressing**:
 - 1 MHz in fast mode I2C (data rate 400 kbps)
 - 3.4 MHz in high-speed mode (data rate 1 Mbps)
 - 5 MHz in ultra-fast mode (data rate 3.4 Mbps)
- **Addresses need to be unique** on the bus to determine the slave that were to transmit the data.
- The **master device needs no address** since it generates the clock (using SCL) and addresses individual I2C Slave devices.
- The **maximum number of Slave devices** that can be used while using **7-bit addressing are 112 devices** The I2C specification has reserved 2 sets of 8 addresses, **1111XXX and 0000XXX**. and the maximum number of Slave devices used in **10-bit addressing are 1008 devices. The remaining 16 are reserved.**

↑
Extension to standard
mode I2C

Working of I2C in Arduino

- The I2C is a **half-duplex type of communication**. A master (controller) device can only read or write data to the slave (peripheral) at a time. All operations are controlled by master device(s).
- In I2C data transfer occurs in **Message Frames** which are then divided into **Frames of Data**. A message contains the various number of Frames in which one frame contain the address of the slave, and remaining frames for data to be transmitted.
- In the I2C protocol, writing to and reading from the target (peripheral/slave) device requires the use of an I2C address. The I2C address identifies which device the controller (master) wants to communicate with. Typically, this address is written over a single byte, where the address itself is 7 bits, and an eighth additional bit is used to indicate a read from or write to the device.
- The **message includes** START/STOP Conditions, READ/WRITE Bits, Address Bits, ACK/NACK (Acknowledgement/No-acknowledgement) Bits, and Data Bits between each **Frame of Data**.
- Working sequence described in the following slides:

Working of I2C in Arduino

- **Start Condition:** The SDA line switches from **high to low** voltage level before SCL switches from **high to low**.
 - **Stop Condition:** The SDA line switches from **low to high** voltage level after SCL switches from **low to high**.
 - **Address Frame:** **7 or 10-bit sequence** unique to each slave that identifies the slave when the master wants to talk.
 - **Read/Write Bit:** **A bit specifying** whether the **master is sending data to the slave or requesting data** from it.
 - **ACK/NACK Bit:** **Each frame** in a message **follows an ACK/NACK Bit**.
- ☐ **7-bit Addressing:**
- In 7-bit addressing procedure, the slave address is transferred in the first byte after the Start condition. The **first seven bits** of the byte comprise the **slave address**. The **eighth bit is the read/write flag** where **0** indicates a **write** and **1** indicates a **read**.

Working of I2C in Arduino



Figure 1: 7-bit addressing. The I2C bus specification specifies that in standard-mode I2C, the slave address is 7-bits long followed by the read/write bit.

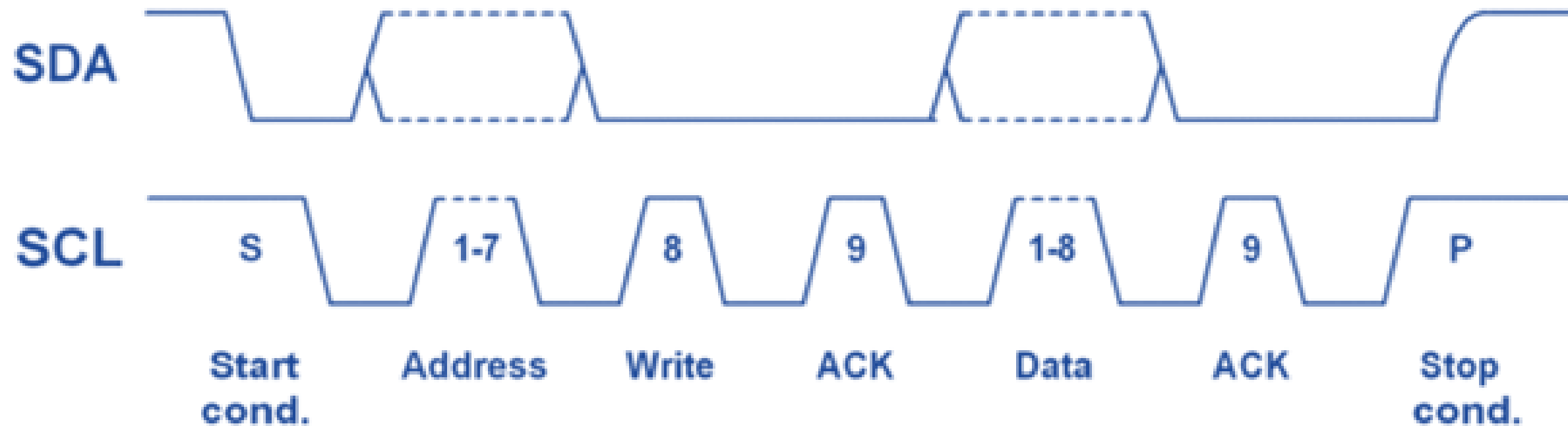
- All I2C products from Total Phase, follow this standard convention. The slave address used should only be the top seven bits. In the case of the Aardvark I2C/SPI Host Adapter, the software will automatically append the correct read/write bit depending on the transaction to be performed. In the case of the Beagle I2C/SPI Protocol Analyzer, the slave address and the type of transaction are displayed in two different columns.
- **Reserved Addresses**
- The I2C has several sets of reserved addresses that are limited for use based on specific applications. The functions called with these reserved addresses are options for devices and are not necessarily available in all I2C devices. The I2C's specific **reserved two sets of eight addresses are 1111XXX and 0000XXX**. These addresses are used for special purposes.

Working of I2C in Arduino



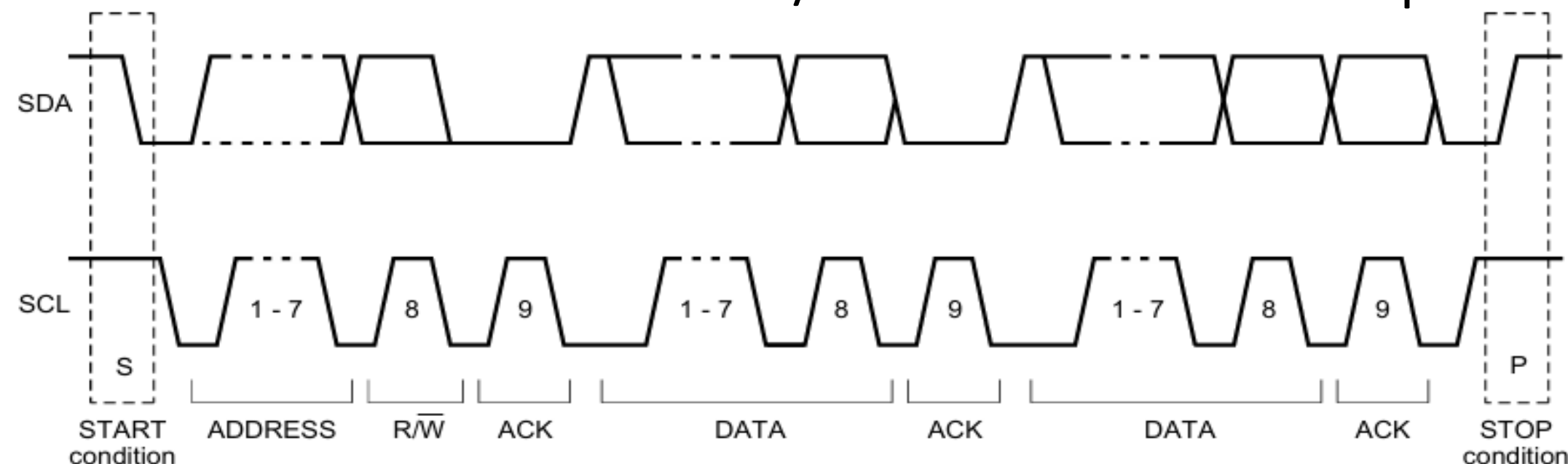
Figure 1: 7-bit addressing. The I2C bus specification specifies that in standard-mode I2C, the slave address is 7-bits long followed by the read/write bit.

- The first byte of an I2C transfer contains the slave address and the data direction.
- The address is 7 bits long, followed by the **direction bit (read or write operation)**. Like all data bytes, **the address is transferred with the most significant bit first**.



Working of I2C in Arduino

- Data transfers follow the format shown in Fig. 2. After the START condition (S), a target address is sent. This address is seven bits long followed by an **eighth bit** which is a data direction bit (**R/W**), a '0' indicates a **transmission of data (WRITE)**, a '1' indicates a **request for data (READ)**.
- A data transfer is always terminated by a STOP condition (P) generated by the controller. However, if a controller still wishes to communicate on the bus, it can generate **a repeated START condition (Sr)** and address another target without first generating a STOP condition. Various combinations of read/write formats are then possible within such a transfer.



The total number of bits transferred in frame one is $7 + 1 + 1 + 8 + 1 = 18$.

If the clock frequency is 200 kHz with 7 bit addressing mode, the clock period is $1/200 \text{ ms} = 5 \mu\text{s}$. Thus, the data transfer time = $18 \times 5 = 90 \mu\text{s}$.

So, the data transfer rate is $18 \times 10^6 / 90 = 200 \text{ kbps}$.

The frame rate is $10^6 / 90 \cong 11 \text{ kbps}$.

Figure 2: A complete data transfer.

AMERICAN INTERNATIONAL UNIVERSITY- BANGLADESH

Where leaders are created

AMERICAN INTERNATIONAL UNIVERSITY- BANGLADESH

Where leaders are created

AMERICAN INTERNATIONAL UNIVERSITY- BANGLADESH

Where leaders are created

AMERICAN INTERNATIONAL UNIVERSITY- BANGLADESH

Where leaders are created

Microprocessor and Embedded Systems

Working of I2C in Arduino

Possible data transfer formats are:

Controller-transmitter transmits data to target-receiver. The data transfer direction is not changed (Fig. 3). The target receiver acknowledges each byte.

Controller reads target immediately after first byte (Fig. 4). At the moment of the first acknowledge, the controller-transmitter becomes a controller-receiver and the target-receiver becomes a target-transmitter. This first acknowledge is still generated by the target. The controller generates subsequent acknowledges. The STOP condition is generated by the controller, which sends a not-acknowledge (\bar{A}) just before the STOP condition.

S

TARGET ADDRESS

R/ \bar{W}

A

DATA

A

DATA

A/ \bar{A}

P

'0' (write)

data transferred
(n bytes + acknowledge)

from controller to target

from target to controller

A = acknowledge (SDA LOW)

\bar{A} = not acknowledge (SDA HIGH)

S = START condition

P = STOP condition

S

TARGET ADDRESS

R/ \bar{W}

A

DATA

A

DATA

\bar{A}

P

(read)

data transferred
(n bytes + acknowledge)

Figure 4: A controller reads a target immediately after the first byte.

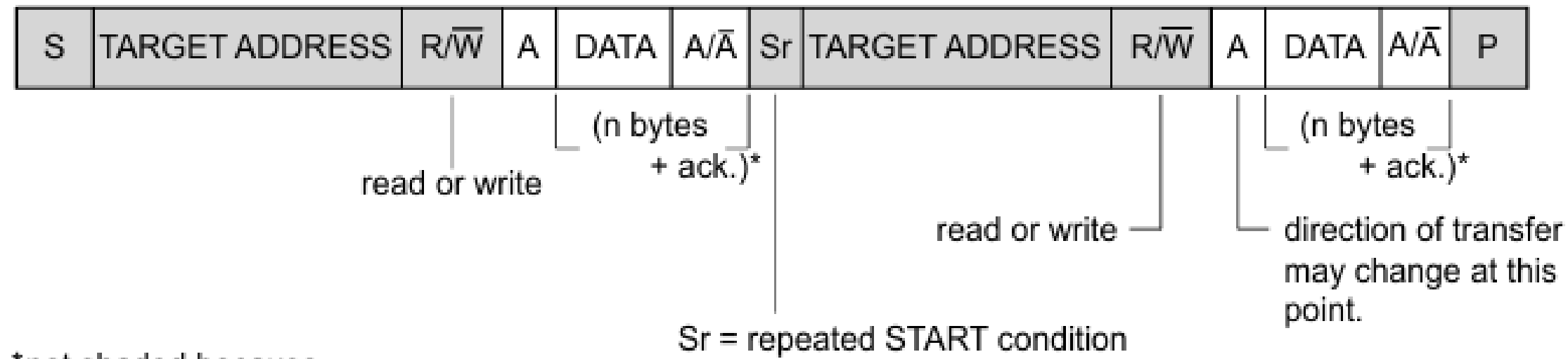
Figure 3: A controller-transmitter addressing a target receiver with a 7-bit address (the transfer direction is not changed).

11 May 2025

Course Teacher: Prof. Dr. Engr. Muhibul Haque Bhuyan

42

- ❑ Possible data transfer formats are:
- Combined format (Fig. 5): During a change of direction within a transfer, the START condition and the target address are both repeated, but with the R/W bit reversed. If a controller-receiver sends a repeated START condition, it sends a not acknowledge (\bar{A}) just before the repeated START condition.



*not shaded because transfer direction of data and acknowledge bits depends on R/ \bar{W} bits.

Figure 5: A controller-transmitter and a target-receiver both are exchanging data.

- Combined formats can be used, for example, to control a serial memory. The internal memory location must be written during the first data byte. After the START condition and target address is repeated, and data can be transferred.
- Each byte is followed by an acknowledgment bit as indicated by the A or A blocks in the sequence.

Working of I2C in Arduino

- The following table has been taken from the [I2C Specifications \(2000\)](#).

List of Reserved I2C addresses

Target Address	R/W Bit	Description
000 0000	0	General call address
000 0000	1	START byte
000 0001	X	C-Bus address
000 0010	X	Reserved for different bus format
000 0011	X	Reserved for future purposes
000 01XX	X	Hs-mode controller code
111 11XX	1	Device ID
111 10XX	X	10-bit target address


- (1) No device is allowed to acknowledge at the reception of the START byte.
- (2) The CBUS address has been reserved to enable the inter-mixing of CBUS compatible and I2C-bus compatible devices in the same system. I2C-bus compatible devices are not allowed to respond on reception of this address.
- (3) The address reserved for a different bus format is included to enable I2C and other protocols to be mixed. Only I2C-bus compatible devices that can work with such formats and protocols are allowed to respond to this address.

- **10-bit Addressing**
- One of the reasons that Total Phase decided to use 7-bit addressing for all of its products was to ensure that 10-bit addressing could be properly handled.
- 10-bit addressing was designed to be compatible with 7-bit addressing, allowing developers to mix two types of devices on a single bus. When communicating with a 10-bit addressed device, the special reserved address is used to indicate that 10-bit addressing is being used.



Figure 6: 10-bit addressing. In 10-bit addressing, the slave address is sent in the first two bytes. The first byte begins with the special reserved address of 1111 0XX which indicates that 10-bit addressing is being used. The 10 bits of the address is encoded in the last 2 bits of the first byte and the entire 8 bits of the second byte. The 8th bit of the first byte remains the read/write flag.

- This target stays in communication until the controller sends a STOP condition or until the controller sends a repeated START condition to communicate with a different target address.



The diagram illustrates the timing sequence for a successful I2C write operation. It shows a sequence of bits on the SDA line: a Start (S) bit, followed by the 7-bit target address, a Write (W) bit, an Acknowledge (A) bit, the 8-bit data byte, another Acknowledge (A) bit, and finally a Stop (P) bit. The clock (SCL) line shows the timing of these bits. The sequence is labeled as '78 to 7B' for the address and 'XXXX XXXX' for the data. The diagram is divided into sections: 'START', 'Write ACK', '8 bits for second byte', 'ACK', and 'STOP'.

Signal	Bit	Value	Label
SDA	1	S	START
	2	1	Target address 1 st 7 bits
	3	1	
	4	1	
	5	0	
	6	X	
	7	X	
	8	0	
	9	A	
	10	X	Target address 2 nd byte
11	X		
12	X		
13	X		
14	X		
15	X		
16	X		
17	X		
18	A	ACK	
19	A		
20	Data	8 bits for second byte	
21	Data		
22	A	ACK	
23	A		
24	Data	Data	
25	Data		
26	A	A	
27	A		
28	P	STOP	

Working of I2C in Arduino

- **10-bit Addressing (Data Read)**

- Reading from a 10-bit addressed device is like a write but with added steps. At the beginning of the communication, there is a START followed by the reserved address. A 0 for a write bit is then written. This is followed by an ACK from all devices that use the reserved address. The target address second byte is then sent. An ACK from the addressed device is then received. Until this point, the communication is the same as a 10-bit address write (Fig. 8).
- To read from this device, the controller then sends a repeated START. This step is followed by the reserved address that was just used. Then the read bit is sent followed by an ACK. Because the read bit is sent (and not the write bit), the device that previously ACKed this communication interprets that this is a read. Other devices with the same reserved address do not respond. The addressed device then ACKs this repeat of the reserved target address.

AMERICAN INTERNATIONAL UNIVERSITY- BANGLADESH

Where leaders are created

1984

BAKU

STATUS

AMERICAN INTERNATIONAL UNIVERSITY- BANGLADESH

Where leaders are created

Microprocessor and Embedded Systems

Working of I2C in Arduino

- **10-bit Addressing (Data Read)**
- After the addressed device sends the ACK, data is transmitted by the device, and after each byte, the controller ACKs the data. This data transmission continues until the controller sends a STOP condition or a repeated START followed by a different target address (Fig. 8).

78 to 7B
1 1 1 1 0 X X

0

X X X X X X X X

78 to 7B
1 1 1 1 0 X X

1

S	Target address 1 st 7 bits	W	A	Target address 2 nd byte	A	Sr	Target address 1 st 7 bits	R	A	Data	A		Data	\bar{A}	P
START		Write ACK		8 bits for second address byte		Repeated START		STOP							

Figure 8: I2C 10-bit addressing read.

11 May 2025

Course Teacher: Prof. Dr. Engr. Muhibul Haque Bhuyan

48

Program for I2C: I2C_SCANNER

```
#include <Wire.h>

void setup()
{
    Wire.begin();

    serial.begin(9600);
    while (!Serial); // Leonardo: wait for serial monitor
    serial.println("\nI2C Scanner");
}

void loop()
{
    byte error, address;
    int nDevices;

    serial.println("Scanning...");

    nDevices = 0;
```

```
for(address = 1; address < 127; address++ )
{
    // The i2c_scanner uses the return value of the
    // Write.endTransmission to see if a device did acknowledge to
    // the address.
    Wire.beginTransmission(address);
    error = Wire.endTransmission();

    if (error == 0)
    {
        serial.print("I2C device found at address 0x");
        if (address<16)
            serial.print("0");
        serial.print(address ,HEX);
        serial.println(" !");

        nDevices++;
    }
}
```

Program for I2C: I2C_SCANNER

```
else if (error==4)
{
    serial.print("Unknown error at address 0x");
    if (address<16)
        serial.print("0");
    serial.print(address,HEX);
    serial.println(" !");
}
}
if (nDevices == 0)
    serial.println("No I2C devices found\n");
else
    serial.println("done\n");

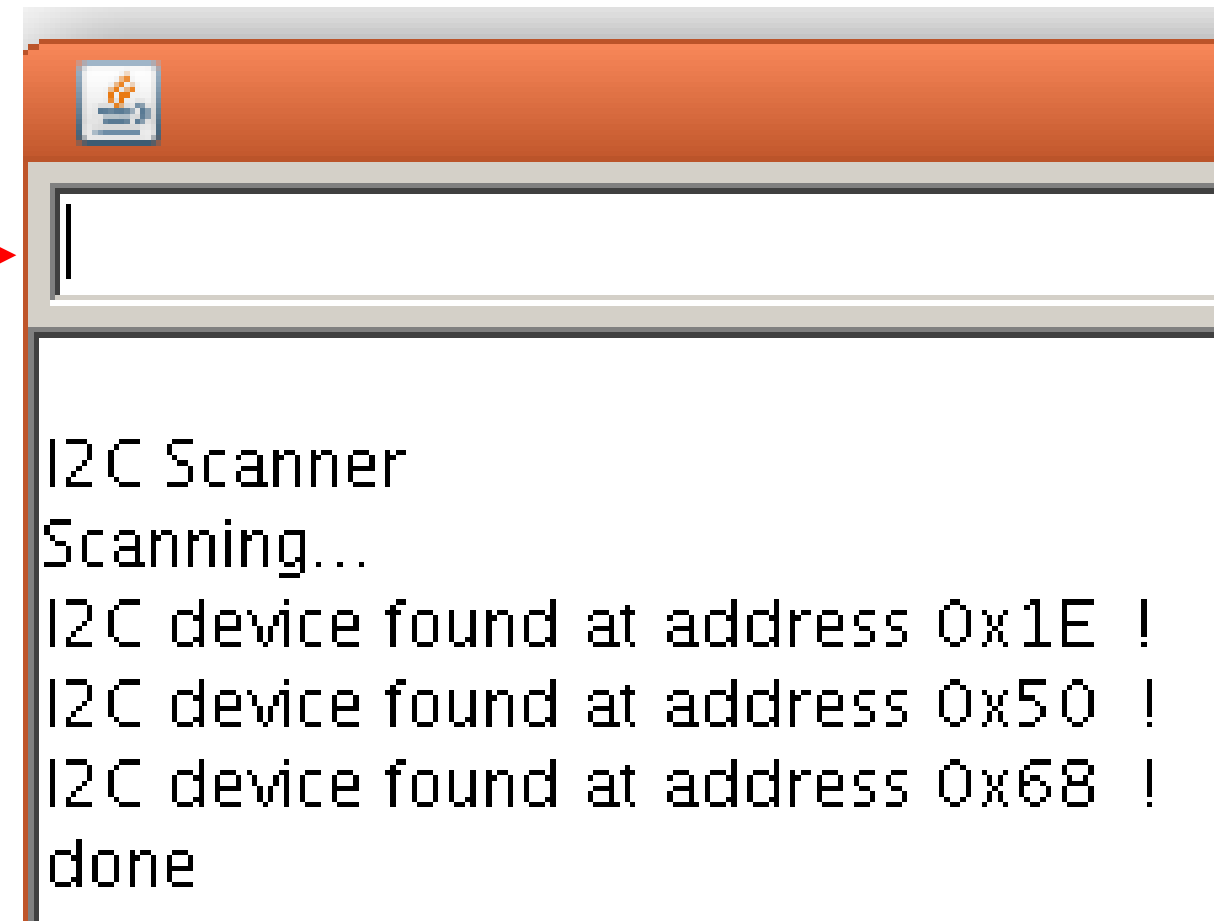
delay(5000);    // wait 5 seconds for next scan
}
```

**Find addresses of different I2C devices
connected to Arduino**

<https://playground.arduino.cc/Main/I2cScanner/>

Serial Monitoring

- Upload it to the Arduino and open the serial monitor. Every found device on the I2C-bus is reported.
- You can change the wires, and plug-in I2C devices while the I2C_scanner is running.
- The output of the serial monitor looks like this:



```
I2C Scanner
Scanning...
I2C device found at address 0x1E !
I2C device found at address 0x50 !
I2C device found at address 0x68 !
done
```

Advantages and Disadvantages of I2C

□ Advantages

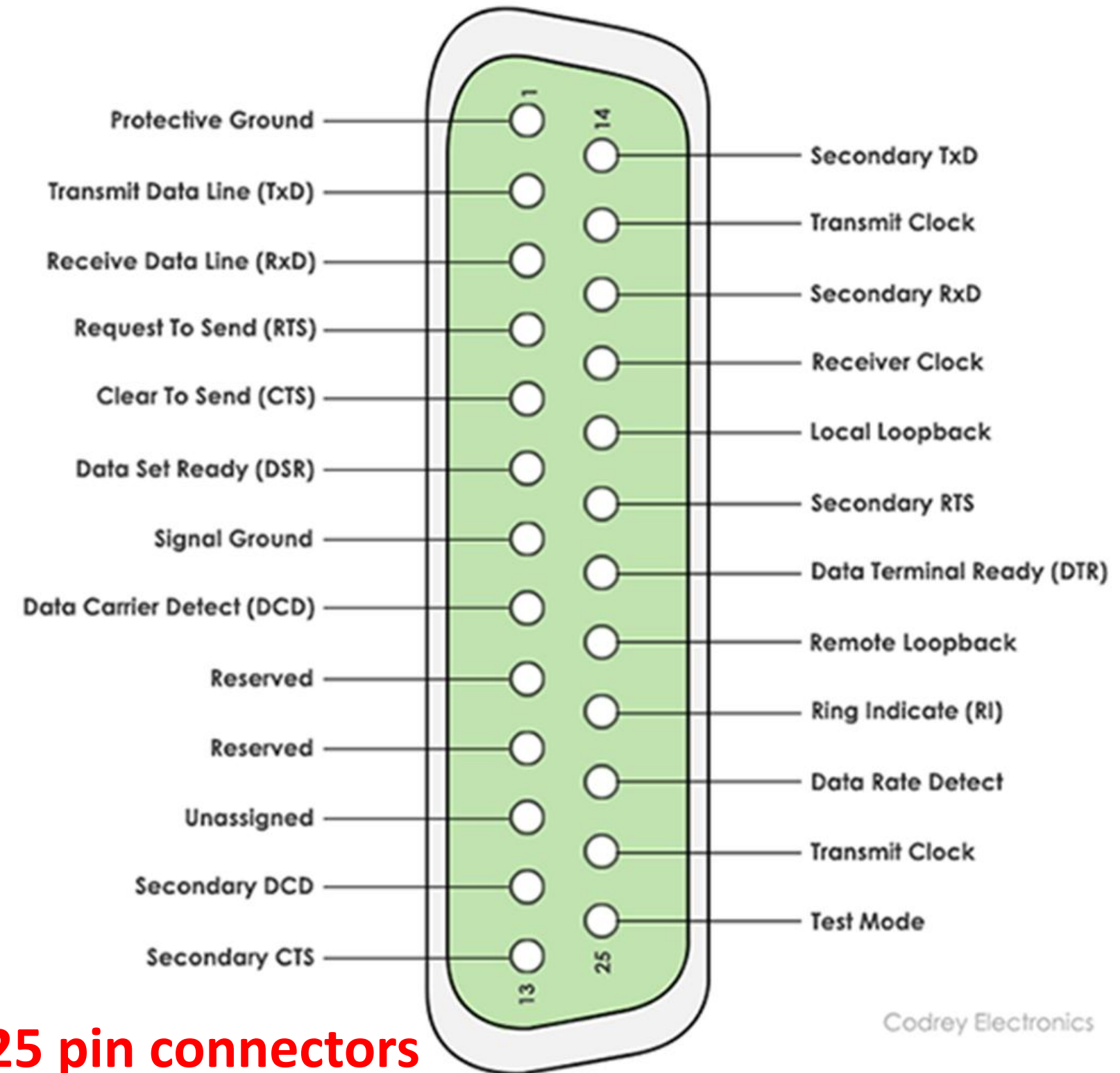
- Due to open collector design, limited slew rates can be achieved.
- More than one masters can be used in the electronic circuit design.
- Needs fewer i.e., only 2 wires for communication.
- I2C addressing is simple which does not require any CS lines used in SPI and it is easy to add extra devices on the bus.
- It uses open collector bus concept. Hence there is bus voltage flexibility on the interface bus.
- Uses flow control.

• Disadvantages

- Increases complexity of the circuit when number of slaves and masters increases.
- I2C interface is **half duplex**.
- Requires software stack to control the protocol and hence it needs some processing overheads on microcontroller/ microprocessor

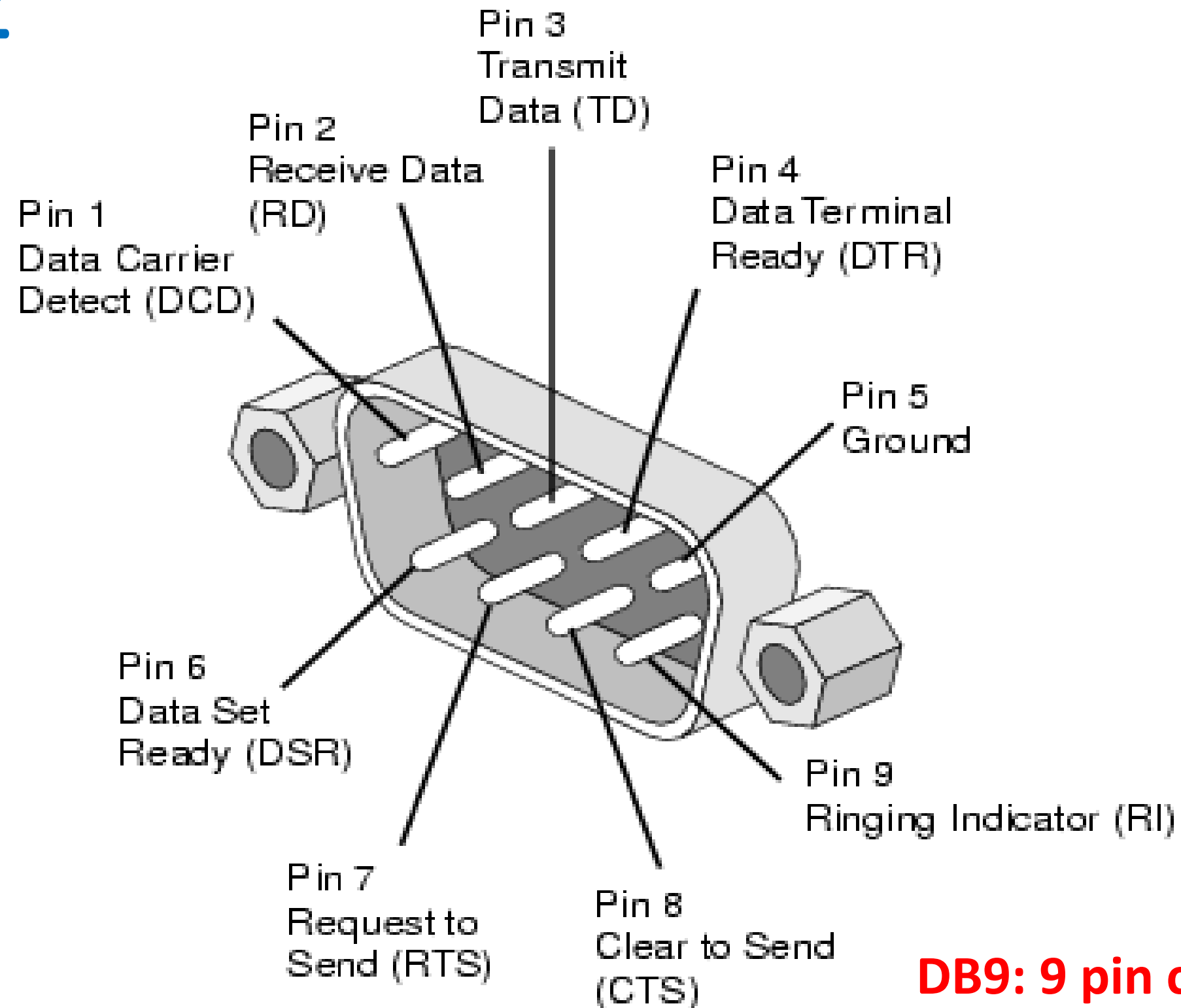
RS232

- **RS232** is the **interface** mainly used for **serial data communication**.
- It supports **data transfer rate from about 110 bps to about 115200 bps**.
- Hyper terminal is the application mainly used to check serial communication port of the computer, often referred as **COM port**.
- The interface is of **two types**-
 - **DB9 pin connector** and
 - **DB25 pin connector**.
- The interface is **mainly used for one-to-one serial communication**.



DB25: 25 pin connectors

RS232



DB9: 9 pin connectors

Summary

- ❑ The **UART** is good for **basic, full-duplex** data communication between **two devices** with a similar clock.
- ❑ An **SPI** is good for **full-duplex, high-speed** data communication with **two or more peripherals (slave) with a single controller (master) device**.
- ❑ An **I2C** is good for **slow-speed data communication** with **multiple devices, among multiple masters** over a 2-wire bus, but it is a **half-duplex communication channel**.

Thanks for attending....

