PT. SOLUSI INTEK INDONESIA

# TYPES OF SERIAL COMMUNICATION

BASIC ELECTRONICS TUTORIAL #03

COMPILED BY: M. LATIEF & TEAM

I2180622082

DEPARTMENT OF RESEARCH AND DEVELOPMENT

MECHATRONICS DIVISION

JUNE 2024

# LIST OF CHAPTER

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# OVERVIEW COMMUNICATION SERIAL

## I. OVERVIEW COMMUNICATION SERIAL

Serial communication is a widely used method for transferring data between computers, microcontrollers, and various peripherals one bit at a time over a communication channel. Unlike parallel communication, which transfers multiple bits simultaneously, serial communication sends data sequentially, making it more cost-effective for longer distances and less prone to signal degradation.

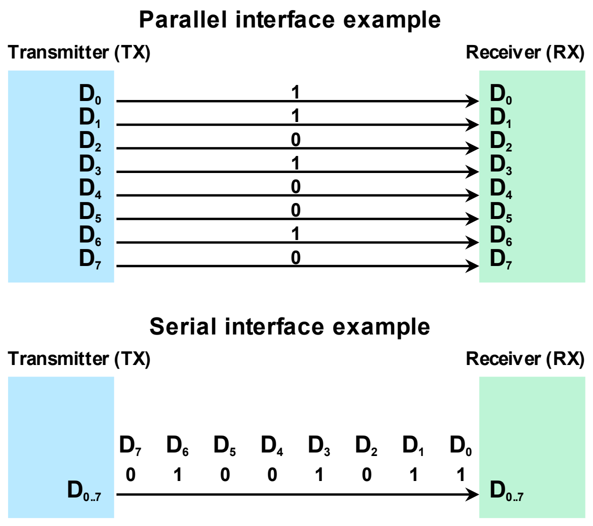

*Figure 1. 1. Parallel Interface Example and Serial Interface Example*

## I.1. The Differences Between Serial and Parallel Interface

***Table 1. 1***

| Feature | Serial Communication | Parallel Communication |
|---|---|---|
| Data Transmission | One bit at a time sequentially | Multiple bits (typically 8, 16, or 32) simultaneously |
| Number of Wires/Lines | Fewer wires (typically 2-4) | More wires (one for each bit plus control lines) |
| Distance | Suitable for long-distance communication | Suitable for short-distance communication |
| Speed | Generally slower (depends on baud rate) | Generally faster (depends on clock speed) |
| Complexity | Simpler and cheaper wiring | More complex and expensive wiring |
| Synchronization | Can be asynchronous or synchronous | Typically synchronous |
| Data Integrity | Less crosstalk and interference over long distances | More prone to crosstalk and signal degradation over long distances |
| Common Protocols | UART, RS-232, SPI, I2C, CAN, USB | PCI, IEEE 1284 (parallel port), memory buses |
| Use Cases | Microcontrollers, sensors, modems, peripheral devices | Internal computer buses, parallel ports (legacy printers) |

## I.2. Key Characteristics of Serial Communication

1. **Data Transfer**: Data is transmitted one bit at a time, either synchronously or asynchronously.
2. **Simplicity**: Requires fewer wires compared to parallel communication, reducing complexity and cost.
3. **Versatility**: Used in a variety of applications from microcontroller interfacing to long-distance data communication.
4. **Types**: Includes various protocols like RS-232, UART, SPI, I2C, CAN, and USB, each with unique features and applications.

## I.3. Types of Serial Communication

A. **UART (Universal Asynchronous Receiver/Transmitter)**

B. **RS-232 (Recommended Standard 232)**

C. **RS-485 (Recommended Standard 232)**

D. **SPI (Serial Peripheral Interface)**

E. **I2C (Inter-Integrated Circuit)**

F. **CAN (Controller Area Network)**

G. **USB (Universal Serial Bus)**

### I.3.A. UART (Universal Asynchronous Receiver/Transmitter)

**Description:** UART is a hardware communication protocol that uses asynchronous serial communication. It converts parallel data from a microcontroller into serial form and vice versa.

**Key Features:**

- Simple and cost-effective.
- No clock signal needed.
- Commonly used in microcontrollers and embedded systems.

**Applications:**

- Microcontroller communication.
- GPS modules.
- Bluetooth modules.

**Example:** An Arduino board communicating with a GPS module.

**Pinout:**

- TX (Transmit)
- RX (Receive)
- GND (Ground)

## I.3.B. RS-232 (Recommended Standard 232)

**Description:** RS-232 is one of the oldest serial communication standards used for transmitting data over short distances. It uses a single-ended signal for data transmission and supports asynchronous communication.

**Key Features:**

- Typically used for communication between computers and peripherals.
- Supports full-duplex communication (simultaneous two-way communication).
- Standard baud rates include 9600, 19200, 38400, 57600, and 115200.

**Applications:**

- Serial ports on computers.
- Modems.
- Industrial equipment.

**Example:** A computer communicating with a serial printer.

**Pinout:**

- TX (Transmit)
- RX (Receive)
- GND (Ground)
- Control signals like RTS (Request to Send) and CTS (Clear to Send).

## I.3.C. RS-485 (Recommended Standard 485)

**Description:** RS-485 (Recommended Standard 485) is a communication standard for serial data transmission. It is widely used in industrial and building automation for robust and reliable data communication over long distances and in electrically noisy environments. RS-485 supports multipoint connections, allowing multiple devices to communicate on the same bus.

**Key Features:**

- Differential signaling for improved noise immunity.
- Supports long-distance communication (up to 1200 meters).
- Allows multipoint (multi-drop) network configurations with up to 32 devices.
- High data transfer rates (up to 10 Mbps for short distances).
- Half-duplex and full-duplex communication modes.

**Applications:**

- Industrial automation and control systems.
- Building automation (HVAC, lighting control).
- Long-distance data communication in noisy environments.
- Communication between multiple devices on the same bus.

**Example:** An industrial control system where multiple sensors and actuators communicate with a central controller via an RS-485 bus.

**Pinout:**

- **A (Data +):** Positive differential signal.
- **B (Data -):** Negative differential signal.
- **GND (Ground, optional):** Common ground reference.

### I.3.D. SPI (Serial Peripheral Interface)

**Description:** SPI is a synchronous serial communication protocol used for short-distance communication, primarily in embedded systems.

**Key Features:**

- Uses a master-slave architecture.
- Supports high-speed data transfer.
- Uses separate lines for data in, data out, clock, and chip select.

**Applications:**

- Sensors.
- SD cards.
- Display modules.

**Example:** A microcontroller communicating with an SD card module.

**Pinout:**

- MOSI (Master Out Slave In)
- MISO (Master In Slave Out)
- SCLK (Serial Clock)
- CS (Chip Select)

### I.3.E. I2C (Inter-Integrated Circuit)

**Description:** I2C is a multi-master, multi-slave, packet-switched, single-ended, serial communication bus. It is used for attaching low-speed peripherals to processors and microcontrollers.

**Key Features:**

- Two-wire interface.
- Supports multiple masters and slaves.
- Each device has a unique address.

**Applications:**

- Temperature sensors.
- EEPROMs.
- Real-time clocks.

**Example:** A Raspberry Pi communicating with a temperature sensor.

**Pinout:**

- SDA (Serial Data Line)
- SCL (Serial Clock Line)

### I.3.F. CAN (Controller Area Network)

**Description:** CAN is a robust serial communication protocol designed for automotive and industrial applications to allow microcontrollers and devices to communicate with each other without a host computer.

**Key Features:**

- Supports multi-master and multi-slave configuration.
- High immunity to noise.
- Reliable data transfer.

**Applications:**

- Automotive electronics.
- Industrial automation.
- Medical equipment.

**Example:** An ECU (Electronic Control Unit) in a car communicating with various sensors and actuators.

**Pinout:**

- CAN_H (CAN High)
- CAN_L (CAN Low)

### I.3.G. USB (Universal Serial Bus)

**Description:** USB is an industry standard for short-distance digital data communications. It connects peripherals such as keyboards, mice, printers, and storage devices to computers.

**Key Features:**

- High data transfer rates.
- Supports plug-and-play.
- Provides power to connected devices.

**Applications:**

- Computer peripherals.
- External storage devices.
- Mobile device charging and data transfer.

**Example:** A computer communicating with a USB flash drive.

**Pinout:**

- VBUS (Power)
- D- (Data -)
- D+ (Data +)
- GND (Ground)

# CHAPTER 2

UART

# II. UART (UNIVERSAL ASYNCHRONOUS RECEIVER/TRANSMITTER)

## II.1.A. Overview

UART is a hardware communication protocol used for asynchronous serial communication between devices. It is widely used in embedded systems and microcontrollers for communication with peripheral devices such as sensors, GPS modules, and Bluetooth modules.

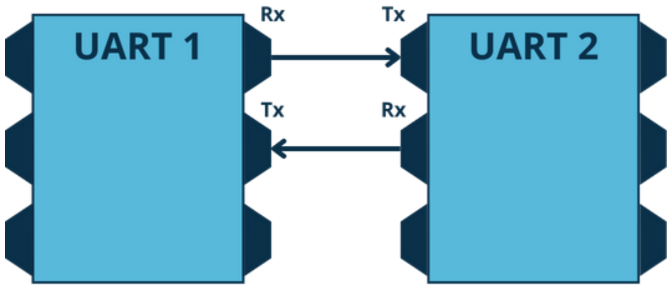

*Figure 2. 1 Two UARTs communication with each other*

## II.1.B. Key Components

1. **Transmitter (TX):** Converts parallel data from the device into serial data and transmits it bit by bit.
2. **Receiver (RX):** Receives serial data bit by bit and converts it back into parallel data for the device.

## II.1.C. Working Principle

***Table 2. 1***

| Data Transmission | Data Reception |
|---|---|
| Start Bit: A low signal indicates the start of a data transmission. The start bit signals the receiver to prepare for an incoming byte. | The receiver waits for a start bit. |
| Data Bits: Typically 5 to 9 bits representing the actual data. Common configurations are 7 or 8 data bits. | Once the start bit is detected, the receiver reads the incoming bits at the preconfigured baud rate. |
| Parity Bit (optional): Used for error checking. It can be even, odd, or none. | After receiving the data bits, it checks the parity bit (if used) for any transmission errors. |
| Stop Bit: A high signal indicating the end of a data transmission. There can be one or two stop bits. | Finally, it looks for the stop bit(s) to ensure the end of the transmission. |

## II.1.D. Data Framing

A UART frame consists of:

- 1 Start Bit
- 5-9 Data Bits
- 1 Parity Bit (optional)
- 1 or 2 Stop Bits

**Example of a UART frame:**

| Start Bit | Data Bits (7-8 bits) | Parity Bit (optional) | Stop Bit(s) |

## II.1.E. Baud Rate

The baud rate defines the number of signal changes (bits per second) transmitted or received. Both the transmitter and receiver must operate at the same baud rate for successful communication.

Common baud rates include:

- 9600
- 19200
- 38400
- 57600
- 115200

## II.1.F. Parity and Error Checking

- **Even Parity:** The number of 1s in the data bits plus the parity bit is even.
- **Odd Parity:** The number of 1s in the data bits plus the parity bit is odd.
- **No Parity:** No parity bit is used.

Error checking mechanisms can include parity bits and, in more sophisticated systems, additional error-checking protocols like CRC (Cyclic Redundancy Check).

## II.1.G. Flow Control

Flow control ensures that the sender does not overwhelm the receiver with data. Common methods include:

- **Hardware Flow Control (RTS/CTS):** Ready to Send (RTS) and Clear to Send (CTS) lines manage the data flow.
- **Software Flow Control (XON/XOFF):** Uses special characters to control data flow.

## II.1.H. Advantages of UART

- **Simplicity:** Easy to implement and use.
- **Cost-effective:** Requires fewer pins and lines than parallel communication.
- **Asynchronous:** Does not require a clock signal.
- **Error Detection:** Parity bit allows for simple error detection.

## II.1.I. Common Use Cases

- **Microcontroller Communication:** Connecting microcontrollers to peripherals like sensors and modules.

- **GPS Modules:** Receiving location data from GPS devices.
- **Bluetooth Modules:** Wireless data transfer between devices.
- **Serial Consoles:** Debugging and monitoring embedded systems.

## II.1.J. Example Code

**Arduino Example:**

```
void setup() {
 Serial.begin(9600); // Initialize UART with 9600 baud rate
}
void loop() {
 if (Serial.available() > 0) {
   int incomingByte = Serial.read(); // Read the incoming byte
   Serial.print("Received: ");
   Serial.println(incomingByte, DEC); // Print the received byte as a decimal number
 }
}
```

**Python Example:**

```
import serial
def read_from_serial(port='COM3', baud_rate=9600):
    ser = serial.Serial(port, baud_rate)
    while True:
        if ser.in_waiting > 0:
            line = ser.readline().decode('utf-8').rstrip()
            print(f"Received: {line}")

if __name__ == "__main__":
    read_from_serial()
```

## II.1.K. Detailed UART Communication Steps

1. **Configuration:**
   - Both devices configure their UART modules with the same baud rate, data bits, parity, and stop bits.
2. **Idle State:**
   - The line remains high (logic 1) during the idle state.
3. **Start Bit:**
   - The transmitter pulls the line low (logic 0) to indicate the start of a transmission.
4. **Data Bits:**
   - Data bits are transmitted sequentially, starting with the least significant bit (LSB).
5. **Parity Bit:**
   - An optional parity bit is transmitted for error checking.
6. **Stop Bit:**
   - The line is pulled high (logic 1) to signal the end of the transmission.
7. **Reception:**
   - The receiver detects the start bit, reads the data bits at the configured baud rate, checks the parity bit, and waits for the stop bit.

## II.1.L. Conclusion

UART is a fundamental communication protocol widely used in embedded systems for serial communication. Its simplicity, cost-effectiveness, and flexibility make it a popular choice for interfacing microcontrollers with various peripherals. Understanding UART's working principles, data framing, and configuration is crucial for designing reliable serial communication systems.

# CHAPTER 3

## RS-232

# III. RS-232 (RECOMMENDED STANDARD 232)

## III.1.A. Overview

RS-232, also known as Recommended Standard 232, is a standard for serial communication transmission of data. It is used for serial communication between computers and peripheral devices such as modems, mice, and printers. Established by the Electronic Industries Alliance (EIA), RS-232 is widely used due to its simplicity and reliability.

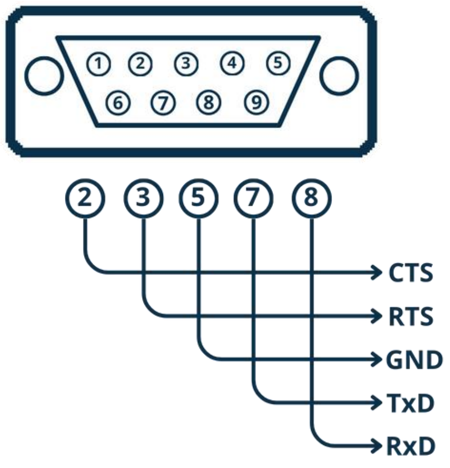

*Figure 3. 1. Male pinout of a 9-pin (D-subminiature, DE-9) serial port commonly found on 1990s computers*

## III.1.B. Key Concepts

- Serial Communication: Data is transmitted one bit at a time over a single channel.
- Asynchronous Communication: Each data byte is framed with start and stop bits, allowing the receiver to synchronize with the sender.
- Voltage Levels: RS-232 uses specific voltage levels for signaling:
- Logic 1 (Mark): -3V to -15V
- Logic 0 (Space): +3V to +15V
- Baud Rate: The speed of data transmission, typically measured in bits per second (bps).

## III.1.C. RS-232 Connectors

- DB-25: A 25-pin connector commonly used in early RS-232 implementations.
- DB-9: A 9-pin connector that became more common due to its smaller size.

## III.1.D. RS-232 Pinout (DE – 9 pin and DB – 25 pin)

***Table 3. 1 Data and Control Signal***

| Circuit | | | Direction | | DB – 25 pin | DE – 9 pin (TIA – 574) |
|---|---|---|---|---|---|---|
| Name | Typical purpose | Abbreviation | DTE | DCE | | |
| Data Terminal Ready | DTE is ready to receive, initiate, or continue a call | DTR | Out | In | 20 | 4 |
| Data Carrier Detect | DCE is receiving a carrier from a remote DCE | DCD | In | Out | 8 | 1 |
| Data Set Ready | DCE is ready to receive and send data | DSR | In | Out | 6 | 6 |
| Ring Indicator | DCE has detected an incoming ring signal on the telephone line. | RI | In | Out | 22 | 9 |
| Request To Send | DTE requests the DCE prepare to transmit data | RTS | Out | In | 4 | 7 |
| Ready To Receive | DTE is ready to receive data from | RTR | Out | In | 4 | 7 |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
|  | DCE. If in use, RTS is assumed to be always asserted |  |  |  |  |  |
| Clear To Send | DCE is ready to accept data from the DTE. | CTS | In | Out | 5 | 8 |
| Transmitted Data | Carries data from DTE to DCE. | TxD | Out | In | 2 | 3 |
| Received Data | Carries data from DCE to DTE. | RxD | In | Out | 3 | 2 |
| Common Ground | Zero voltage reference for all of the above. | GND | Common |  | 7 | 5 |
| Protective Ground | Connected to chassis ground. | PG | Common |  | 1 | - |

## III.1.E. RS-232 Communication

- Full-Duplex: Allows simultaneous transmission and reception of data.
- Flow Control: Manages data flow between devices to prevent data loss. Common methods include:
- Hardware Flow Control: Uses RTS/CTS signals.
- Software Flow Control: Uses special control characters (XON/XOFF).

## III.1.F. Example

Let's walk through an example of using RS-232 to communicate between an Arduino and a computer.

### III.1.F.i. Wiring Diagram

1. Connect the RS-232 shield or module to the Arduino.
2. Connect the RS-232 cable from the shield/module to the computer's serial port.

### III.1.F.ii. Arduino Code

**Arduino Sketch:**

```
void setup() {

    // Start the serial communication at 9600 baud

    Serial.begin(9600);

}


void loop() {

    // Send data to the computer

    Serial.println("Hello from Arduino");


     // Wait for a second

    delay(1000);

}
```

### III.1.F.iii. Computer Side Code

You can use a serial terminal program like PuTTY, or you can write a Python script to read data from the Arduino.

**Python Script:**

```
import serial


# Open the serial port

ser = serial.Serial('COM3', 9600)  # Replace 'COM3' with your Arduino's port


while True:

    # Read a line of data from the Arduino

    line = ser.readline().decode('utf-8').rstrip()

    print(line)
```

### III.1.G. Advanced RS-232 Topics

### III.1.G.i. Null Modem

A null modem cable is used to connect two devices directly without a modem. It crosses the transmit and receive lines to enable communication.

### III.1.G.ii. RS-232 Signal Levels

RS-232 signals operate at higher voltage levels than typical TTL logic levels, which is why level shifters (e.g., MAX232) are often used to interface RS-232 with microcontrollers.

### III.1.G.iii. RS-232 Variants

- RS-422: Extends the distance and speed capabilities of RS-232 by using differential signaling.
- RS-485: Allows multiple devices to share the same bus using differential signaling and supports longer distances and higher speeds than RS-232.

## III.1.H. Common Use Cases

- Modem Communication: Connecting computers to modems for internet and dial-up connections.
- Peripheral Communication: Interfacing with serial mice, printers, and other peripherals.
- Embedded Systems: Debugging and communication with microcontrollers and other embedded devices.
- Industrial Automation: Communication between industrial equipment and control systems.

## III.1.I. Debugging RS-232

- Check Connections: Ensure all pins are correctly connected and secure.
- Verify Baud Rate: Make sure both devices are set to the same baud rate.
- Use Serial Analyzers: Tools like serial analyzers or oscilloscopes can capture and analyze RS-232 signals.
- Check Voltage Levels: Ensure the RS-232 voltage levels are within the specified range.

## III.1.J. Conclusion

RS-232 is a versatile and reliable standard for serial communication, widely used in various applications. Its simplicity, combined with robust error-checking mechanisms, makes it ideal for many communication tasks. Understanding how to configure and use RS-232 enables you to build systems that leverage this standard for effective and reliable data transmission.

By mastering RS-232 communication, you can create projects that integrate various peripherals and devices, enhancing functionality and enabling more sophisticated system designs. Despite the emergence of newer communication standards, RS-232 remains a valuable tool in the toolbox of engineers and developers.

# CHAPTER 4

# RS-485

# IV. RS-485 (RECOMMENDED STANDARD 485)

## IV.1.A. Overview

RS-485, also known as Recommended Standard 485, is a standard for serial communication used in industrial and commercial applications. It supports higher data rates and longer distances compared to RS-232 and allows multiple devices to communicate over a single pair of wires. RS-485 is widely used in environments where reliable data transmission is required over extended distances and in noisy environments.

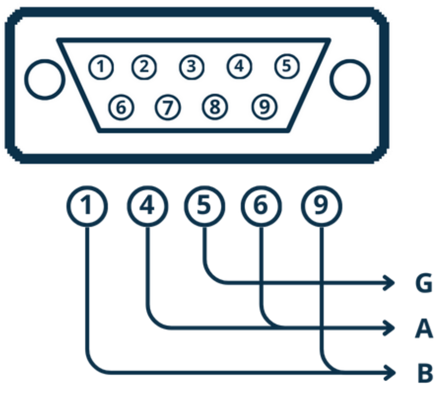

*Figure 4. 1. RS – 485 Pin*

## IV.1.B. Comparison Table

*Table 4. 1 Comparison Table*

| Feature | RS-232 | RS-485 |
|---|---|---|
| Communication Type | Point-to-Point | Multi-Point |
| Voltage Level | $\pm 12\,V$ | $0\,V\ to\ 5\,V\ (Differential)$ |
| Maximum Distance | 15 meters (50 feet) | 1200 meters (4000 feet) |
| Maximum Data Rate | 20 kbps | 10 Mbps |
| Number of Devices | 2 | 32 (standard) |
| Common Aplications | Computer peripherals, modems | Industrial automation |

### IV.1.C. Key Concepts

- Differential Signaling: RS-485 uses differential signaling, which improves noise immunity and allows for longer transmission distances.
- Multipoint Communication: RS-485 supports multiple devices on the same bus, enabling multi-drop or multi-point communication.
- Half-Duplex and Full-Duplex: RS-485 can operate in both half-duplex (two-wire) and full-duplex (four-wire) modes.

### IV.1.D. RS-485 Pinout

- A (Data+ or Non-Inverting): Positive data line.
- B (Data- or Inverting): Negative data line.
- GND: Ground (optional, for common reference).

### IV.1.E. RS-485 Communication

- Bus Topology: RS-485 uses a bus topology, where devices are connected in parallel along a single pair of wires.
- Termination Resistors: To prevent signal reflections, termination resistors (typically 120 ohms) are placed at both ends of the bus.
- Biasing Resistors: Biasing resistors ensure that the bus remains in a known state when no devices are actively transmitting.

### IV.1.F. Example

Let's walk through an example of using RS-485 to communicate between two Arduino boards.

#### IV.1.F.i. Wiring Diagram

1. Connect RS-485 modules to both Arduino boards.
2. Connect the A and B lines of the RS-485 modules to each other.
3. Add termination resistors (120 ohms) at both ends of the bus.

### IV.1.F.ii. Arduino Code for Transmitter

**Transmitter Arduino Sketch:**

```
#include <SoftwareSerial.h>
// Define the pins for the RS-485 module
#define TX_PIN 10
#define RX_PIN 11
#define DE_PIN 2
#define RE_PIN 3
// Create a software serial object
SoftwareSerial rs485Serial(RX_PIN, TX_PIN);
void setup() {
  // Set the control pins as outputs
  pinMode(DE_PIN, OUTPUT);
  pinMode(RE_PIN, OUTPUT);
  // Initialize the RS-485 communication
  rs485Serial.begin(9600);

  // Enable transmission mode
  digitalWrite(DE_PIN, HIGH);
  digitalWrite(RE_PIN, HIGH);
}

void loop() {
  // Send data to the RS-485 bus
  rs485Serial.println("Hello from Transmitter");

  // Wait for a second
  delay(1000);
}
```

## IV.1.F.iii. Arduino Code for Receiver

**Receiver Arduino Sketch:**

```
#include <SoftwareSerial.h>

// Define the pins for the RS-485 module
#define TX_PIN 10
#define RX_PIN 11
#define DE_PIN 2
#define RE_PIN 3

// Create a software serial object
SoftwareSerial rs485Serial(RX_PIN, TX_PIN);

void setup() {
 // Set the control pins as outputs
 pinMode(DE_PIN, OUTPUT);
 pinMode(RE_PIN, OUTPUT);

 // Initialize the RS-485 communication
 rs485Serial.begin(9600);

 // Enable reception mode
 digitalWrite(DE_PIN, LOW);
 digitalWrite(RE_PIN, LOW);
}

void loop() {
 // Check if data is available
 if (rs485Serial.available()) {
    // Read and print the incoming data
    String data = rs485Serial.readString();
    Serial.println(data);
  }
}
```

## IV.1.G. Advanced RS-485 Topics

### IV.1.G.i. Network Configuration

- Node Addressing: Each device on the RS-485 bus can be assigned a unique address for identification and communication.
- Collision Avoidance: Implement protocols to avoid data collisions on the bus, such as Master-Slave or Token Passing protocols.

### IV.1.G.ii. RS-485 Protocols

- Modbus: A popular protocol used in industrial automation systems for communication over RS-485.
- Profibus: Another protocol commonly used in industrial applications, providing robust and reliable communication.

## IV.1.H. Common Use Cases

- Industrial Automation: Communication between PLCs (Programmable Logic Controllers), sensors, and actuators.
- Building Automation: Controlling HVAC (Heating, Ventilation, and Air Conditioning) systems, lighting, and security systems.
- Remote Monitoring: Monitoring and controlling remote equipment and systems.
- Data Acquisition: Collecting data from various sensors and transmitting it to a central controller.

## IV.1.I. Debugging RS-485

- Check Connections: Ensure all connections are secure and correctly wired.
- Verify Termination: Ensure termination resistors are correctly placed at both ends of the bus.
- Use Diagnostic Tools: Use oscilloscopes or logic analyzers to capture and analyze RS-485 signals.
- Check Voltage Levels: Ensure the differential voltage levels are within the specified range.

## IV.1.J. Conclusion

RS-485 is a powerful and reliable standard for serial communication in industrial and commercial applications. Its ability to support long-distance communication and multiple devices on the same bus makes it ideal for various use cases. Understanding how to configure and use RS-485 enables you to build systems that leverage this standard for efficient and robust data transmission.

By mastering RS-485 communication, you can create projects that integrate various devices and sensors, enhancing functionality and enabling more sophisticated system designs. Despite the emergence of newer communication standards, RS-485 remains a valuable tool in the toolbox of engineers and developers, particularly in industrial and automation contexts.

# CHAPTER 5

SPI

## V. SPI (SERIAL PERIPHERAL INTERFACE)

### V.1.A. Overview

SPI (Serial Peripheral Interface) is a synchronous serial communication protocol used to transfer data between microcontrollers and peripheral devices such as sensors, SD cards, and shift registers. SPI is widely used in embedded systems due to its simplicity and high-speed data transfer capabilities.

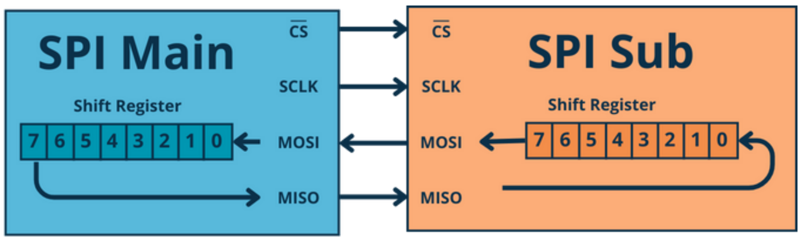

**Figure 5. 1 Basic SPI Configuration Using a Single Main and a Single Sub**

### V.1.B. Key Concepts

1. Master-Slave Architecture: SPI operates in a master-slave configuration where the master device controls the clock and initiates data transfers, while the slave device responds to the master's commands.
2. Full-Duplex Communication: SPI supports full-duplex communication, meaning data can be transmitted and received simultaneously.
3. Clock Polarity and Phase: SPI allows configuring the clock polarity (CPOL) and clock phase (CPHA) to support different timing requirements.

## V.1.C. SPI Signals

***Table 5. 1***

| Short Name | Long Name | Description (historical terms in parens) |
|---|---|---|
| $\overline{CS}$ | Chip Select | Active-low select signal from main (master) to enable communication with a specific sub (slave) device |
| SCLK | Serial Clock | Clock Signal from main (master) transitions for each serial data bit. |
| MOSI | Main Out, Sub In (master out, slave in) | Serial Data from main (master), highest bit first. |
| MISO | Main In, Sub Out (Master in, Slave Out) | Serial data from sub (slave), highest bit first. |

## V.1.D. Wiring and Connections

SPI typically requires four wires:

- MISO (connected between master MISO and slave MISO)
- MOSI (connected between master MOSI and slave MOSI)
- SCLK (connected between master SCLK and slave SCLK)
- CS (connected between master CS and slave CS)

### V.1.E. SPI Modes

SPI has four possible modes based on the CPOL and CPHA values:

***Table 5. 2***

| SPI Modes | CPOL | CPHA | Clock Polarity in Idle State | Clock Phase Used to Sample and/or Shift the Data |
|---|---|---|---|---|
| 0 | 0 | 0 | Logic Low | Data sampled on rising edge and shifted out on the falling edge |
| 1 | 0 | 1 | Logic Low | Data sampled on the falling edge and shifted out on the rising edge |
| 2 | 1 | 0 | Logic High | Data sampled on the falling edge and shifted out on the rising edge |
| 3 | 1 | 1 | Logic High | Data sampled on the rising edge and shifted out on the falling edge |

### V.1.F. Example: Using SPI with Arduino

Let's walk through an example of using SPI to communicate between an Arduino (as master) and an SPI-based sensor (as slave).

#### V.1.F.i. Wiring Diagram:

1. Connect the MISO pin on the Arduino to the MISO pin on the sensor.
2. Connect the MOSI pin on the Arduino to the MOSI pin on the sensor.
3. Connect the SCLK pin on the Arduino to the SCLK pin on the sensor.
4. Connect the SS pin on the Arduino to the CS pin on the sensor.

## V.1.F.ii. Arduino Code

**Arduino Sketch:**

```
#include <SPI.h>

const int CS_PIN = 10;

void setup() {

      // Initialize the SPI library

       SPI.begin();

       // Set the SS pin as an output

       pinMode(CS_PIN, OUTPUT);

       // Set the SS pin high to start

       digitalWrite(CS_PIN, HIGH);

       // Set the SPI clock speed and data order

       SPI.beginTransaction(SPISettings(4000000, MSBFIRST, SPI_MODE0));

}

void loop() {

       // Select the slave device by setting the SS pin low

       digitalWrite(CS_PIN, LOW);

       // Send and receive data

       byte receivedData = SPI.transfer(0x42); // Send 0x42 and receive a byte

       // Deselect the slave device by setting the SS pin high

       digitalWrite(CS_PIN, HIGH);

       // Use the received data (e.g., print it to the serial monitor)

       Serial.println(receivedData);

       // Delay for a while before the next transfer

       delay(1000);

}
```

## V.1.G. Common Use Cases

- Sensor Interfacing: Reading data from sensors like accelerometers, gyroscopes, and temperature sensors.
- Memory Devices: Communicating with EEPROMs, flash memory, and SD cards.
- Display Modules: Controlling SPI-based LCD and OLED displays.
- Shift Registers: Expanding the number of I/O pins using shift registers.

## V.1.H. Conclusion

SPI is a powerful and flexible communication protocol widely used in embedded systems. Understanding how to configure and use SPI allows you to interface with a variety of peripheral devices, enabling you to build complex and responsive systems.

# CHAPTER 6

# I2C

# VI. I2C (INTER-INTEGRATED CIRCUIT)

## VI.1.A. Overview

I²C (Inter-Integrated Circuit) is a synchronous, multi-master, multi-slave, packet-switched, single-ended, serial communication bus widely used for connecting lower-speed peripheral ICs to processors and microcontrollers. Developed by Philips Semiconductor (now NXP Semiconductors), I²C is commonly utilized in embedded systems for communication between microcontrollers and peripherals.

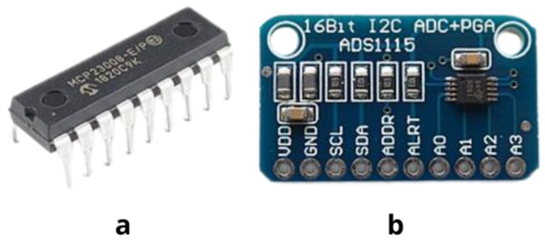

*Figure 6. 1. (a). Microchip MCP23008 8-bit I²C I/O expander in DIP-18 package, (b). a 16-bit ADC board with I²C interface*

## VI.1.B. Key Concepts

- Multi-Master and Multi-Slave: Multiple master and slave devices can coexist on the same bus.
- Addressing: Each device on the I²C bus has a unique address, enabling the master to communicate with multiple slaves.
- SDA and SCL Lines: I²C employs two bidirectional open-drain lines, SDA (Serial Data Line) and SCL (Serial Clock Line), pulled up with resistors.

## VI.1.C. I²C Signals

1. SDA (Serial Data Line): Carries the data.
2. SCL (Serial Clock Line): Carries the clock signal.

## VI.1.D. Wiring and Connections

I²C requires two wires:

- SDA: Connected between the SDA pins of all devices.
- SCL: Connected between the SCL pins of all devices.
- Pull-Up Resistors: Typically 4.7kΩ resistors are connected between SDA and Vcc, and SCL and Vcc.

## VI.1.E. I²C Communication

4. Start Condition: A high-to-low transition on SDA while SCL is high.
5. Address Frame: The master sends the 7-bit address followed by a read/write bit.
6. Acknowledge Bit (ACK): The receiver pulls the SDA line low to acknowledge the receipt of data.
7. Data Frames: Data is transferred in 8-bit bytes, each followed by an ACK bit.
8. Stop Condition: A low-to-high transition on SDA while SCL is high.

## VI.1.F. Addressing

- 7-bit Addressing: Commonly used, supporting 127 unique addresses (some addresses are reserved).
- 10-bit Addressing: Less common, supporting 1024 unique addresses.

## VI.1.G. I²C Modes of Operation

***Table 6. 1.*** ***$I^2C$ Modes***

| Mode | Maximum Speed (kbit/s) | Maximum Capacitance (pF) | Drive | Direction |
|---|---|---|---|---|
| Standard mode (Sm) | 100 | 400 | Open Drain | Bidirectional |
| Fast mode (Fm) | 400 | 400 | Open Drain | Bidirectional |
| Fast mode plus (Fm+) | 1000 | 550 | Open Drain | Bidirectional |
| High-speed mode (Hs) | 1700 | 400 | Open Drain | Bidirectional |
| High-speed mode (Hs) | 3400 | 100 | Open Drain | Bidirectional |
| Ultra-Fast mode (UFm) | 5000 | ? | Push-pull | Unidirectional |

## VI.1.H. Example: Using I²C with Arduino

Let's walk through an example of using I²C to communicate between an Arduino (as master) and an I²C-based sensor (as slave).

### VI.1.H.i. Wiring Diagram

1. Connect the SDA pin on the Arduino to the SDA pin on the sensor.
2. Connect the SCL pin on the Arduino to the SCL pin on the sensor.
3. Connect pull-up resistors (typically 4.7kΩ) between SDA and Vcc, and SCL and Vcc.

### VI.1.H.ii. Arduino Code

**Arduino Sketch:**

```
#include <Wire.h>

const int sensorAddress = 0x68; // Replace with your sensor's I2C address

void setup() {

        // Initialize the I2C communication

        Wire.begin();

        Serial.begin(9600);

}

void loop() {

        // Request data from the sensor

        Wire.beginTransmission(sensorAddress);

        Wire.write(0x00); // Register address to read from

        Wire.endTransmission();

        // Request 2 bytes of data from the sensor

        Wire.requestFrom(sensorAddress, 2);

        if (Wire.available() == 2) {

                // Read the two bytes of data
```

```
                    int highByte = Wire.read();
                     int lowByte = Wire.read();
                // Combine the two bytes into a single 16-bit value
              int sensorValue = (highByte << 8) | lowByte;


               // Print the sensor value to the serial monitor
              Serial.println(sensorValue);
    }
    // Delay before the next loop
    delay(1000);
}
```

### VI.1.I. Multi-Master Communication

In a multi-master configuration, more than one master device can initiate communication on the I²C bus. Arbitration is used to ensure that only one master controls the bus at a time. If two masters start communication simultaneously, the one that sends the higher priority signal (lower address) wins arbitration and continues, while the other master stops.

### VI.1.J. Clock Stretching

Clock stretching allows a slave device to hold the clock line low to indicate that it needs more time to process data. The master must wait until the slave releases the clock line before continuing communication.

### VI.1.K. I²C Bus Speed

Different devices on the I²C bus may support different speeds. It's crucial to configure the I²C clock speed according to the slowest device on the bus to ensure proper communication. The bus speed is typically set in the initialization function of the microcontroller's I²C library.

## VI.1.L. Common Use Cases

- Sensor Interfacing: Reading data from sensors like accelerometers, gyroscopes, temperature sensors, and light sensors.
- Real-Time Clocks (RTCs): Communicating with RTC modules for timekeeping.
- EEPROM: Reading and writing data to EEPROM chips.
- Display Modules: Controlling I$^2$C-based LCD and OLED displays.
- ADC/DAC: Interfacing with analog-to-digital converters and digital-to-analog converters.

## VI.1.M. Debugging I$^2$C

- Check Connections: Ensure all SDA and SCL connections are correct and pull-up resistors are in place.
- Check Addresses: Verify that the correct addresses are being used for communication.
- Use Logic Analyzer: Use a logic analyzer to capture and analyze the I$^2$C signals.
- Software Libraries: Use well-documented and tested I$^2$C libraries provided by the microcontroller's manufacturer or community.

## VI.1.N. Conclusion

I$^2$C is a versatile and widely used communication protocol in embedded systems. Its simplicity and ability to connect multiple devices on the same bus make it a popular choice for interconnecting sensors, displays, and other peripherals with microcontrollers. Understanding how to configure and use I$^2$C enables you to build complex and responsive systems efficiently.

# CHAPTER 7

CAN

# VII. CAN (CONTROLLER AREA NETWORK)

## VII.1.A. Overview

Controller Area Network (CAN) is a robust, high-speed, and reliable communication protocol designed for automotive and industrial applications. It enables microcontrollers and devices to communicate with each other without a host computer, making it ideal for distributed systems. Developed by Bosch in the 1980s, CAN has become a standard in automotive networks and various other applications.

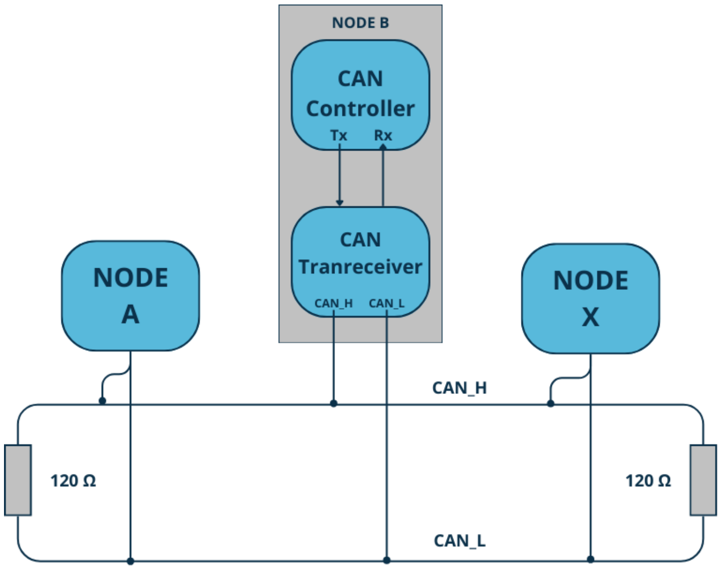

*Figure 7. 1. CAN Interface Hardware*

## VII.1.B. Key Concepts

- Multi-Master: Multiple devices (nodes) can initiate communication.
- Priority-Based Arbitration: Messages are assigned priorities, and the highest priority message gets transmitted first.
- Error Detection and Handling: CAN includes mechanisms for detecting and handling errors to ensure reliable communication.
- Differential Signaling: Uses two wires (CAN_H and CAN_L) for noise immunity and robustness.

## VII.1.C. CAN Bus Signals

1. CAN_H (CAN High): The high voltage line of the differential pair.
2. CAN_L (CAN Low): The low voltage line of the differential pair.

## VII.1.D. Wiring and Connections

CAN uses a two-wire twisted pair cable:

- CAN_H: Connected to the CAN_H pin of all nodes.
- CAN_L: Connected to the CAN_L pin of all nodes.
- Termination Resistors: Typically 120Ω resistors are connected at each end of the bus to prevent signal reflections.

## VII.1.E. CAN Communication

1. Frames: CAN communication is based on frames, which are structured data packets.
2. Identifiers: Each message has an identifier that determines its priority and content.
3. Data Length Code (DLC): Specifies the number of data bytes in the message.
4. CRC: Cyclic Redundancy Check for error detection.
5. ACK: Acknowledgment bit for confirming the receipt of the message.

## VII.1.F. Types of CAN Frames

1. Data Frame: Contains data to be transmitted.
2. Remote Frame: Requests data from another node.
3. Error Frame: Indicates an error has occurred.
4. Overload Frame: Used to inject a delay between data or remote frames.

## VII.1.G. CAN Frame Format

1. Start of Frame (SOF): Indicates the start of a message.
2. Identifier: Defines the message priority and type.
3. Control Field: Contains the DLC.
4. Data Field: Contains the data (0-8 bytes).
5. CRC Field: Contains the CRC for error checking.
6. ACK Field: Indicates successful receipt of the message.
7. End of Frame (EOF): Indicates the end of the message.

## VII.1.H. Example

Let's walk through an example of using CAN to communicate between two Arduino boards using CAN transceivers.

### VII.1.H.i. Wiring Diagram

1. Connect the CAN_H pin of the transceiver to the CAN_H pins of both Arduinos.
2. Connect the CAN_L pin of the transceiver to the CAN_L pins of both Arduinos.
3. Connect 120Ω termination resistors at both ends of the CAN bus.

### VII.1.H.ii. Arduino Code

**Transmitter:**

```
#include <SPI.h>

#include <mcp2515.h>

struct can_frame canMsg;

void setup() {

        Serial.begin(9600);

        // Initialize MCP2515 CAN Controller

        mcp2515.reset();

        mcp2515.setBitrate(CAN_500KBPS, MCP_8MHZ);

        mcp2515.setNormalMode();

        // Configure CAN message

        canMsg.can_id = 0x123; // Identifier

        canMsg.can_dlc = 8;    // Data length code

        canMsg.data[0] = 0x01; // Data bytes

        canMsg.data[1] = 0x02;

        canMsg.data[2] = 0x03;

        canMsg.data[3] = 0x04;

        canMsg.data[4] = 0x05;

        canMsg.data[5] = 0x06;

        canMsg.data[6] = 0x07;

        canMsg.data[7] = 0x08;

}

void loop() {

  // Send CAN message
  mcp2515.sendMessage(&canMsg);
  Serial.println("Message Sent");
  delay(1000);
}
```

**Receiver:**

```
#include <SPI.h>
#include <mcp2515.h>

struct can_frame canMsg;

void setup() {
  Serial.begin(9600);

  // Initialize MCP2515 CAN Controller
  mcp2515.reset();
  mcp2515.setBitrate(CAN_500KBPS, MCP_8MHZ);
  mcp2515.setNormalMode();
}
void loop() {
  // Check for received CAN message
  if (mcp2515.readMessage(&canMsg) == MCP2515::ERROR_OK) {
    Serial.print("Message Received: ");
    for (int i = 0; i < canMsg.can_dlc; i++) {
      Serial.print(canMsg.data[i], HEX);
      Serial.print(" ");
    }
    Serial.println();
  }
}
```

## VII.1.I. Advanced CAN Topics

### VII.1.I.i. CAN FD (Flexible Data Rate)

CAN FD is an extension of the original CAN protocol, providing:

- Higher data rates (up to 8 Mbps).
- Longer data fields (up to 64 bytes).
- Improved efficiency for high-speed communication.

### VII.1.I.ii. CANopen

CANopen is a higher-layer protocol based on CAN, used primarily in industrial automation. It defines:

- Communication objects.
- Device profiles.
- Network management.

### VII.1.I.iii. J1939

J1939 is a higher-layer protocol based on CAN, used primarily in heavy-duty vehicles and agricultural equipment. It defines:

- Standard messages for vehicle components.
- Diagnostic communication.

## VII.1.J. Common Use Cases

- Automotive: Engine control units, airbags, antilock braking systems, and infotainment systems.
- Industrial Automation: Machine control, sensors, and actuators.
- Aerospace: Avionics and control systems.
- Medical Equipment: Patient monitoring systems and diagnostic devices.

## VII.1.K. Debugging CAN

- Check Connections: Ensure all CAN_H and CAN_L connections are correct and termination resistors are in place.
- Check Bus Speed: Verify that all devices on the bus are configured to the same speed.
- Use CAN Analyzer: Use a CAN analyzer tool to capture and analyze CAN messages.
- Error Handling: Monitor and handle error frames to identify and resolve communication issues.

### VII.1.L. Conclusion

CAN is a robust and reliable communication protocol widely used in automotive and industrial applications. Understanding how to configure and use CAN enables you to build distributed systems with multiple microcontrollers and devices efficiently. By mastering CAN communication, you can create systems with enhanced functionality, reliability, and scalability.

With the increasing adoption of CAN FD and higher-layer protocols like CANopen and J1939, CAN continues to be a vital technology in modern embedded systems, providing the necessary performance and reliability for complex and critical applications.

# CHAPTER 8

USB

# VIII. USB (UNIVERSAL SERIAL BUS)

## VIII.1.A. Overview

Universal Serial Bus (USB) is a widely used interface for connecting peripherals to computers and other devices. It supports data transfer, power supply, and device communication. Introduced in the mid-1990s, USB has become the standard for connecting a wide range of devices, from keyboards and mice to storage devices and printers.

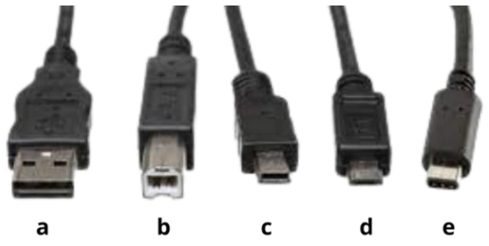

*Figure 8. 1. (a). USB Type A, (b). USB Type B, (c). USB Mini, (d). USB Micro, (e). USB Type C*

## VIII.1.B. Key Concepts

- Plug-and-Play: USB devices are automatically recognized and configured by the host system.
- Hot Swappable: USB devices can be connected and disconnected without powering down the host system.
- Power Supply: USB can provide power to connected devices, eliminating the need for separate power adapters.

## VIII.1.C. USB Versions and Features

***Table 8. 1.***

| USB Versions | Maximum Data Rate | Connector Types | Key Features |
|---|---|---|---|
| USB 1.0/1.1 | Low-speed: 1.5 Mbps <br> Full-speed: 12 Mbps | Type-A, Type-B, Mini-USB | Basic peripheral connectivity, low data rate suitable for keyboards and mice |
| USB 2.0 | High-speed: 480 Mbps | Type-A, Type-B, Mini-USB, Micro-USB | Enhanced data transfer rate, backward compatible with USB 1.x, commonly used for storage devices |
| USB 3.0 | SuperSpeed: 5 Gbps | Type-A, Type-B, Type-C | Significantly higher data transfer rate, additional pins for increased bandwidth, backward compatible with USB 2.0 |
| USB 3.1 | SuperSpeed+: 10 Gbps | Type-A, Type-B, Type-C | Even higher data transfer rate, improved power delivery, backward compatible with USB 3.0 and USB 2.0 |
| USB 3.2 | Up to 20 Gbps | Type-C | Dual-lane operation for double data rate, uses USB Type-C connector exclusively |
| USB 4 | Up to 40 Gbps | Type-C | Integrates Thunderbolt 3, supports multiple data and display protocols, backward compatible with USB 3.x and USB 2.0 |

## VIII.1.D. USB Connectors

- Type-A: The most common rectangular connector found on computers.
- Type-B: Typically used for printers and other large peripherals.
- Mini-USB: Smaller connector used for older portable devices.
- Micro-USB: Smaller connector used for mobile devices.
- USB-C: Reversible connector that supports USB 3.x and USB4.

## VIII.1.E. USB Architecture

1. Host: The computer or device that controls the communication.
2. Device: The peripheral connected to the host.
3. Hub: An intermediate device that expands a single USB port to multiple ports.
4. Endpoints: Logical channels within a USB device for data transfer.

## VIII.1.F. USB Communication

1. Control Transfers: Used for configuration, command, and status operations.
2. Bulk Transfers: Used for large, non-time-critical data transfers (e.g., file transfers).
3. Interrupt Transfers: Used for small, time-critical data transfers (e.g., mouse and keyboard inputs).
4. Isochronous Transfers: Used for time-sensitive data transfers (e.g., audio and video streams).

## VIII.1.G. USB Power Delivery

- USB 2.0: Up to 2.5W (5V, 500mA).
- USB 3.0/3.1: Up to 4.5W (5V, 900mA).
- USB Power Delivery (USB PD): Up to 100W (20V, 5A).

## VIII.1.H. Example

Let's walk through an example of using USB to communicate between an Arduino and a computer.

### VIII.1.H.i. Wiring Diagram

1. Connect the Arduino to the computer using a USB cable.
2. Ensure the Arduino drivers are installed on the computer.

### VIII.1.H.ii. Arduino Code

```
void setup() {
  // Start the serial communication
  Serial.begin(9600);
}

void loop() {
  // Send data to the computer
  Serial.println("Hello from Arduino");

  // Wait for a second
  delay(1000);
}
```

### VIII.1.H.iii. Computer Side Code

You can use a serial terminal program like PuTTY, or you can write a Python script to read data from the Arduino.

**Python Script:**

```
import serial

# Open the serial port
ser = serial.Serial('COM3', 9600)  # Replace 'COM3' with your Arduino's port

while True:
    # Read a line of data from the Arduino
    line = ser.readline().decode('utf-8').rstrip()
    print(line)
```

## VIII.1.I. Advanced USB Topics

### VIII.1.I.i. USB Descriptors

USB devices use descriptors to provide information about their capabilities and configurations. Key descriptors include:

- Device Descriptor: Contains general information about the device.
- Configuration Descriptor: Describes the device's power requirements and configuration options.
- Interface Descriptor: Describes individual interfaces within a configuration.
- Endpoint Descriptor: Describes endpoints within an interface.

#### VIII.1.I.ii. USB Classes

USB classes define standard protocols for common device types, enabling interoperability between devices and hosts. Examples include:

- Human Interface Device (HID): Keyboards, mice, game controllers.
- Mass Storage: USB flash drives, external hard drives.
- Communications Device Class (CDC): Modems, network adapters.
- Audio: USB microphones, speakers.

#### VIII.1.I.iii. USB On-The-Go (OTG)

USB OTG allows devices to act as both host and device, enabling direct communication between two USB devices without a computer. OTG is commonly used in mobile devices.

#### VIII.1.I.iv. USB Type-C and Power Delivery

USB Type-C is a versatile connector that supports USB 3.x and USB4 data rates, as well as USB Power Delivery (PD) for higher power output. USB PD allows dynamic power negotiation, enabling devices to request and receive varying levels of power.

### VIII.1.J. Common Use Cases

- Peripheral Connectivity: Connecting keyboards, mice, printers, and storage devices to computers.
- Mobile Device Charging: Charging smartphones, tablets, and other portable devices.
- Embedded Systems: Communication between microcontrollers and sensors, actuators, and other peripherals.
- Audio and Video: Transferring audio and video data between devices.

### VIII.1.K. Debugging USB

- Check Connections: Ensure all USB connections are secure.
- Check Drivers: Verify that the correct drivers are installed on the host system.
- Use USB Analyzers: Use USB analyzers to capture and analyze USB traffic.
- Monitor Power Usage: Ensure that the USB port provides sufficient power for connected devices.

### VIII.1.L. Conclusion

USB is a versatile and widely adopted communication protocol for connecting peripherals to computers and other devices. Its plug-and-play capability, combined with power delivery and high-speed data transfer, makes it ideal for a wide range of applications. Understanding how to configure and use USB enables you to build systems that leverage this powerful and flexible interface.

By mastering USB communication, you can create projects that seamlessly integrate a variety of peripherals, enhancing functionality and enabling more sophisticated embedded system designs. With the ongoing advancements in USB standards and connectors, USB continues to be a crucial technology in modern electronics.