# EMBEDDED SYSTEMS

## Contents



# DIGITAL ASSIGNMENT 1

**1.Build an embedded system to monitor the temperature and humidity and display the humidity and temperature value to LCD configure to Arduino uno**

**1.** ***Embedded System for Temperature and Humidity Monitoring***

- **Components**: Arduino Uno, DHT11 Sensor, LCD 16x2

**Code**

```
#include <DHT.h>
#include <LiquidCrystal.h>

#define DHTPIN 2
#define DHTTYPE DHT11

DHT dht(DHTPIN, DHTTYPE);
LiquidCrystal lcd(12, 11, 5, 4, 3, 2); //rs, e, d4, d5, d6, d7

void setup() {
  lcd.begin(16, 2);
  dht.begin();
}

void loop() {
  delay(2000);
  float h = dht.readHumidity();
  float t = dht.readTemperature();

  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Humidity: ");
  lcd.print(h);
  lcd.print("%");

  lcd.setCursor(0, 1);
  lcd.print("Temp: ");
  lcd.print(t);
  lcd.print("C");
}
```

**2. Write a short note on watchdog timer register and write a program to configure watchdog timer to schedule programs.**

## WATCHDOG TIMER (WDT) IN EMBEDDED SYSTEMS

A **Watchdog Timer (WDT)** is a ***hardware*** timer that helps **detect and recover from system failures**. It automatically **resets the microcontroller** if the program hangs or gets stuck due to software bugs, infinite loops, or external interference.

### How Watchdog Timer Works

1. **Timer Initialization**
    - The microcontroller starts a countdown timer (e.g., from a preset value).
    - If the program is running correctly, it must **reset (refresh) the WDT** before it expires.
2. **Normal Operation**
    - If the program executes normally, it **resets the WDT periodically** to prevent a reset.
3. **System Failure**
    - If the software **hangs or crashes**, the program fails to reset the WDT.
    - The timer expires, and the WDT **forces a system reset**, restarting the microcontroller.

### Watchdog Timer in Arduino

**Enabling Watchdog Timer**

- The **Arduino AVR-based boards** (like Uno, Mega) have a built-in WDT.
- It is controlled using the **avr/wdt.h** library.

**Example Code (Using Watchdog Timer in Arduino)**

```
#include <avr/wdt.h>  // Include watchdog timer library

void setup() {
  Serial.begin(9600);
  wdt_enable(WDTO_2S);  // Enable watchdog timer with a 2-second timeout
}

void loop() {
  Serial.println("Running...");
  delay(1000);  // Simulate normal operation
  wdt_reset();  // Reset watchdog timer to prevent reset
}
```

- If the program **fails to reset the WDT within 2 seconds**, the system **restarts**.

### Key Features of Watchdog Timer

- **Self-recovery mechanism** → Prevents microcontrollers from getting stuck.
- **Configurable timeout period** → Can be set from **milliseconds to seconds**.
- **Independent operation** → Works even if the main program crashes.

**Use Cases of Watchdog Timer**

- **Industrial Automation** → Ensures robots and machines recover from unexpected failures.
- **IoT Devices** → Prevents smart devices from freezing.
- **Automotive Systems** → Critical for safety in car engine control units (ECUs).

**3. Write a short note on UART and configure respective registers to transmit and receive a serial data "India". write a program.**

**3. UART Communication**

**Short Note**

**Universal Asynchronous Receiver-Transmitter (UART)** is a hardware communication protocol that allows for serial communication between devices. It uses two lines: one for transmitting data and another for receiving.

**Code to Transmit "India"**

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

void loop() {
  Serial.println("India"); // Transmit "India"
  delay(1000); // Wait for 1 second
}
```

**4. Write a program to obtain analogue sensor value by configuring ADC register in arduino UNO board and if the analogue threshold value reaches a certain threshold value greater than 500 configure led to blame else LED off.**

***Analog Sensor Value with ADC***

**Code**

```
#define LED_PIN 9
#define THRESHOLD 500

void setup() {
  pinMode(LED_PIN, OUTPUT);
  Serial.begin(9600);
}

void loop() {
  int sensorValue = analogRead(A0); // Read analog value from pin A0

  if (sensorValue > THRESHOLD) {
    digitalWrite(LED_PIN, HIGH); // Turn LED ON
  } else {
    digitalWrite(LED_PIN, LOW); // Turn LED OFF
  }
  delay(500);
}
```

**5. write a short note on timers, registers, server motor. write a program to configure timer to enable server motor.**

**Timers, Registers, and Servo Motor**

**Short Note**

**Timers** are used in microcontrollers to perform tasks at specific intervals. **Registers** control timer settings, while a **servo motor** is a rotary actuator that allows for precise control of angular position.

## SERVO MOTOR: EXPLANATION & WORKING

A **servo motor** is a **rotary actuator** that allows for **precise control of angular position, velocity, and acceleration**. It consists of a **motor, a position sensor, and a control circuit**.

### How a Servo Motor Works?

1. **Control Signal (PWM Input)**
    - The servo motor is controlled by a **Pulse Width Modulation (PWM) signal**.
    - A typical **50 Hz PWM signal (20ms period)** is used.
    - The **duty cycle** of the PWM determines the position of the motor shaft.
2. **Internal Feedback System**
    - A **position sensor (usually a potentiometer)** provides feedback on the actual position of the shaft.
    - The control circuit adjusts the motor's movement to reach the desired angle.
3. **Movement & Holding Position**
    - The motor moves to the instructed angle and **holds its position** using feedback control.

### PWM Control for Servo Motor

| PWM Signal | Servo Position |
|---|---|
| **1ms pulse (5% duty cycle)** | 0° (Minimum Position) |
| **1.5ms pulse (7.5% duty cycle)** | 90° (Middle Position) |
| **2ms pulse (10% duty cycle)** | 180° (Maximum Position) |

### Types of Servo Motors

1. **Positional Rotation Servo**
    - Rotates within a fixed range (0° to 180° or 0° to 270°).
    - Used in robotic arms, RC cars, etc.
2. **Continuous Rotation Servo**
    - Can rotate **continuously in either direction** like a DC motor.
    - Used in conveyor belts, wheels of robots.

3. **Linear Servo**
    - Converts rotational motion into **linear motion.**
    - Used in **robotic actuators.**

**Applications of Servo Motors**

✅ **Robotics** – Used for precise joint movements.
✅ **RC Vehicles** – Steering control in cars, airplanes.
✅ **Industrial Automation** – Conveyor belts, CNC machines.
✅ **3D Printers** – Used for accurate positioning.
✅ **Automatic Doors** – Opening and closing mechanisms.

**Servo Motor Control with Arduino**

A servo motor can be controlled using the **Servo library** in Arduino:

```
#include <Servo.h>

Servo myServo; // Create servo object

void setup() {
  myServo.attach(9); // Attach servo to pin 9
}

void loop() {
  myServo.write(0);   // Move to 0 degrees
  delay(1000);        // Wait 1 sec
  myServo.write(90);  // Move to 90 degrees
  delay(1000);
  myServo.write(180); // Move to 180 degrees
  delay(1000);
}
```

**Code to Control Servo Motor by configuring timer**

```
#include <Servo.h>

Servo myServo;

void setup() {
  myServo.attach(9); // Attach servo on pin 9
  Timer1.initialize(1000000); // 1 second
  Timer1.attachInterrupt(moveServo);
}

void moveServo() {
  myServo.write(90); // Move servo to 90 degrees
}

void loop() {
  // Main code
}
```

**6. Build an embedded system to detect the object and display distance by configuring LCD 16 x 2 displaying the status in the line one as "object detected" and on the second line displaying "distance in cm".**

***Object Detection System***

**Block Diagram**

- **Components**: Arduino Uno, Ultrasonic Sensor (HC-SR04), LCD 16x2
- **Connections**:
    - Ultrasonic:
        - VCC to 5V
        - GND to GND
        - Trigger to Digital Pin 9
        - Echo to Digital Pin 10

**Code**

```
#include <LiquidCrystal.h>

LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

void setup() {
  lcd.begin(16, 2);
  pinMode(9, OUTPUT);
  pinMode(10, INPUT);
}

void loop() {
  long duration, distance;
  digitalWrite(9, LOW);
  delayMicroseconds(2);
  digitalWrite(9, HIGH);
  delayMicroseconds(10);
  digitalWrite(9, LOW);
  duration = pulseIn(10, HIGH);
  distance = duration * 0.034 / 2;

  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Object Detected");
  lcd.setCursor(0, 1);
  lcd.print("Distance: ");
  lcd.print(distance);
  lcd.print(" cm");
  delay(1000);
}
```

**7. Build an embedded system to connect a photo transistor sensor and detect the Lux value units for brightness and based on the LUX value rotate the servo motor**

**(threshold value greater than 180) and rotate by 90 degree else the servo motor should be off.**

***Photo Transistor Sensor for Brightness***

**Block Diagram**

- **Components**: Arduino Uno, Photo Transistor, Servo Motor
- **Connections**:
    - Photo Transistor:
        - Collector to A0
        - Emitter to GND
        - Base to VCC
    - Servo Motor: Same as above.

**Code**

```
#include <Servo.h>

Servo myServo;
#define THRESHOLD 180

void setup() {
  myServo.attach(9);
  pinMode(A0, INPUT);
}

void loop() {
  int luxValue = analogRead(A0); // Read LUX value

  if (luxValue > THRESHOLD) {
    myServo.write(90); // Rotate by 90 degrees
  } else {
    myServo.write(0); // Turn off
  }
  delay(500);
}
```

# MODULE 2

**1. Types of Memory**

**RAM (Random Access Memory)**

- **Definition**: RAM is a type of ***volatile*** memory that allows both ***reading and writing*** of data. It is used for ***temporary*** storage while a system is running.
- **Characteristics**:
    - **Volatility**: Data is lost when power is turned off.
    - **Speed**: ***Fast*** access times, making it ideal for active processes.

- **Usage**: Stores data that the CPU needs immediately, such as ***variables*** and ***program instructions***.

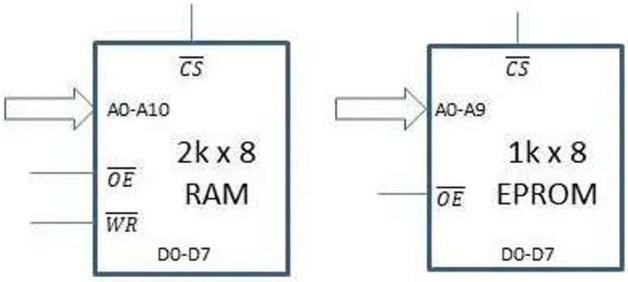
- **Control Signals**:
  - **Memory Read (RD)**: Enables the microprocessor to read data from RAM.
  - **Memory Write (WR)**: Enables the microprocessor to write data to RAM.

**ROM (Read Only Memory)**

- **Definition**: ROM is a type of ***non-volatile*** memory that is used primarily for ***permanent storage*** of data. Once data is written to ROM, it typically ***cannot be modified***.
- **Characteristics**:
  - **Non-volatility**: Retains data even when power is turned off.
  - **Speed**: Generally ***slower*** than RAM but sufficient for its purpose.
  - **Usage**: Stores firmware or software that does not change frequently (e.g., BIOS).
- **Control Signals**:
  - **Memory Read (RD)**: Used to read data from ROM.
  - No write capability in standard ROM.

**2. Memory Interfacing Problem Statement**

- **Task**: Interface a **1kB EPROM** and a **2kB RAM** with the **8085 microprocessor**.
- **Address Allocation**:
  - **1kB EPROM**: Assigned address range from 2000H to 22FFH.
  - **2kB RAM**: Address range can be assigned based on design choice (e.g., 2300H to 23FFH).




**3. Address and Data Pins**

**Data Pins**

- **Data Lines**: Each memory location typically stores 8 bits (1 byte), so:
  - **8 Data Lines**: D0, D1, D2, D3, D4, D5, D6, D7 are connected to the data bus. This allows the transfer of 8 bits of data simultaneously.

**Address Pins**

- **Address Lines**: The number of address lines required depends on the size of the memory.
  - **1kB EPROM**:

- **Memory Locations**: $2^{10}$=1024 locations.
- **Address Lines Needed**: 10 address lines (A0 to A9).

- **2kB RAM**:
  - **Memory Locations**: $2^{11}$=2048 locations.
  - **Address Lines Needed**: 11 address lines (A0 to A10).

**4. Control Signals**

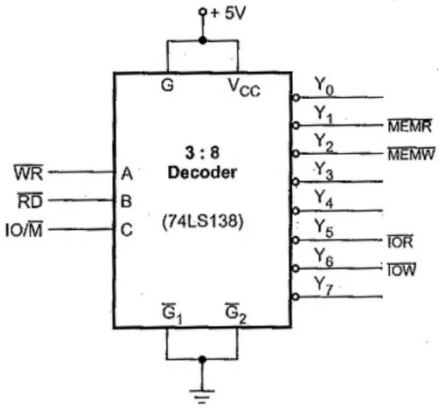
Control signals are crucial for coordinating operations between the microprocessor and the memory chips.

**CS (Chip Select) Pin**

- **Function**: Activates the memory chip, indicating that the microprocessor wants to communicate with it.
- **Operation**: The CS pin is driven low to select the chip. Only the selected chip responds to read/write commands.

**OE (Output Enable) Pin**

- **Function**: Allows the memory chip to output data onto the data bus.
- **Operation**: This pin is active-low, meaning it must be set to low for the data to be read from the memory.




**WR (Write) Pin**

- **Function**: Enables writing data to the memory chip.
- **Operation**: This pin is also active-low; when activated, the data present on the data bus is stored in the specified memory location.

**5. Power Pins**

- **VCC and GND Pins**:
  - These pins are essential for powering the memory chips.
  - **VCC**: Connects to the positive supply voltage.
  - **GND**: Connects to the ground.

**6. Control Signal Generation**

The control signals for memory operations are derived from the microprocessor's signals.

**Control Signals from 8085**

- **WR (Write)**: Activates the WR pin on the RAM when writing data.
- **RD (Read)**: Activates the RD pin on the ROM when reading data.
- **MEMR (Memory Read)**: Signal generated to read from memory.
- **MEMW (Memory Write)**: Signal generated to write to memory.

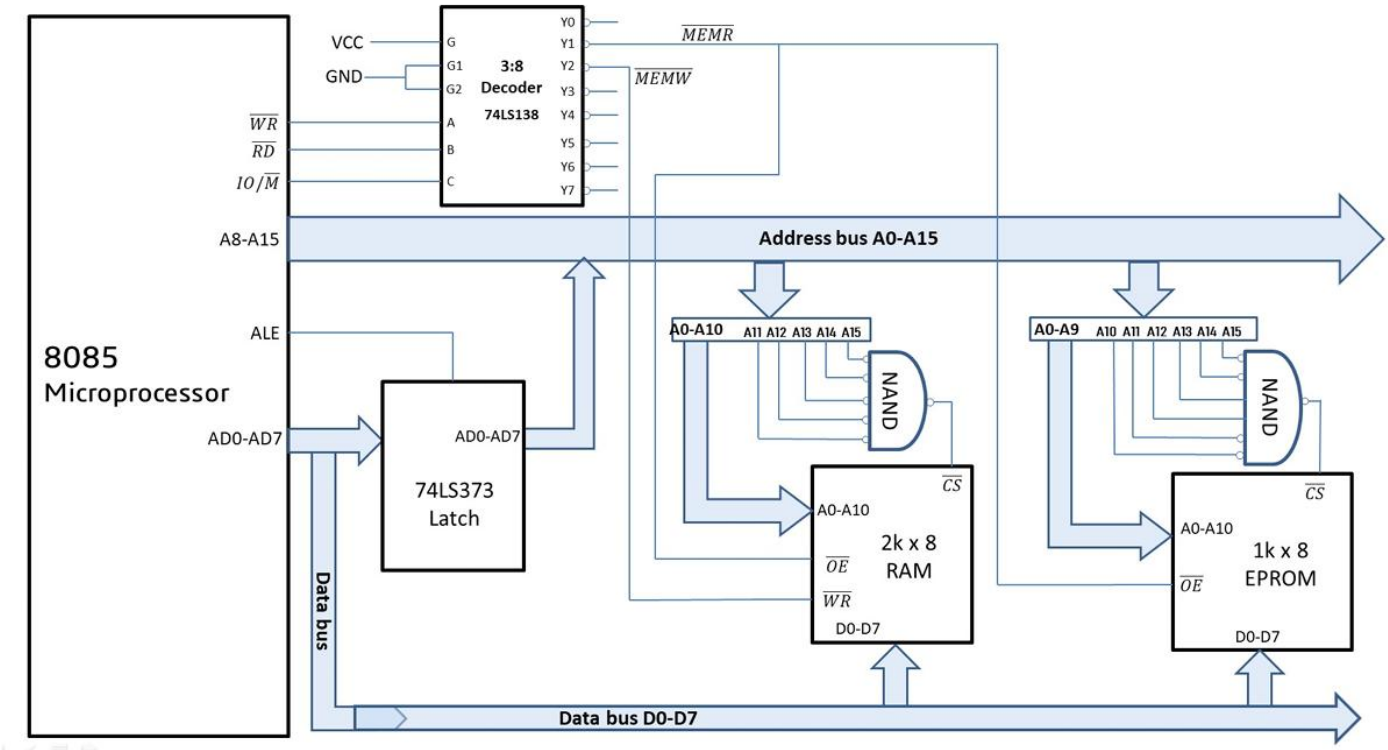
**7. Connections**

**Reading and Writing Operations**

- **Reading from Memory**:
  - **MEMR** is connected to the **OE** pin of the memory chip (RAM or ROM). When MEMR is activated, the data stored in memory is placed on the data bus.
- **Writing to Memory**:
  - **MEMW** is connected to the **WR** pin of the RAM. When MEMW is activated, the data on the data bus is written to the specified memory address.

**8. Address Bus Connections**

- The address bus of the 8085 microprocessor consists of 16 lines (A0 to A15), allowing access to 64KB of memory.
- **Connections**:
  - **2kB RAM**: The first 11 address lines (A0 to A10) are connected to the RAM.
  - **1kB EPROM**: The first 10 address lines (A0 to A9) are connected to the EPROM.
- **Generating CS Signals**:
  - The remaining address lines are used to generate the Chip Select (CS) signals for both memory chips. This ensures that only one chip is selected at any time based on the address being accessed.



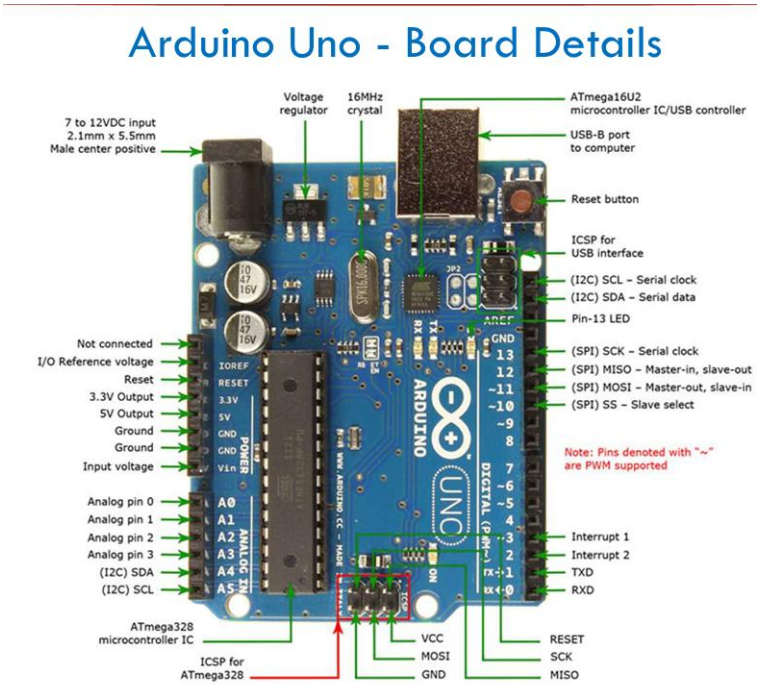

## INTRODUCTION TO ARDUINO UNO

**Features**

- **An open-source microcontroller board**
- **Microcontroller**: ATmega328

- **Operating Voltage**: 5V
- **Input Voltage (external)**: 6-20V (7-12V recommended)
- **Digital I/O Pins**: 14 (6 PWM output)
- **Analog Input Pins**: 6
- **DC Current per I/O Pin**: 40 mA
- **DC Current for 3.3V Pin**: 50 mA
- **Flash Memory**: 32 KB (0.5 KB for bootloader)
- **Clock Frequency**: 16 MHz (Crystal Oscillator)
- **Programming Interface**: USB (ICSP)

---




## Arduino Uno - Board Details

### Pin Details

- **Digital Pins**: 0-13
- **Analog Pins**: A0-A5
- **Power Pins**: GND, 5V, 3.3V, Vin
- **PWM Pins**: 3, 5, 6, 9, 10, 11

### INTRODUCTION TO ATMEGA328

### Features

- **Instruction Set Architecture**: RISC
- **Data Bus Size**: 8-bit
- **Address Bus Size**: 16-bit
- **Program Memory (ROM)**: 32 KB
- **Data Memory (RAM)**: 2 KB
- **Data Memory (EEPROM)**: 1 KB

- **Input/Output Ports**: 3 (B, C, D)
- **General Purpose Registers**: 32
- **Timers**: 3 (Timer0, Timer1, Timer2)
- **Interrupts**: 26 (2 External interrupts)
- **ADC Modules**: 6 channels (10-bit)
- **PWM Modules**: 6

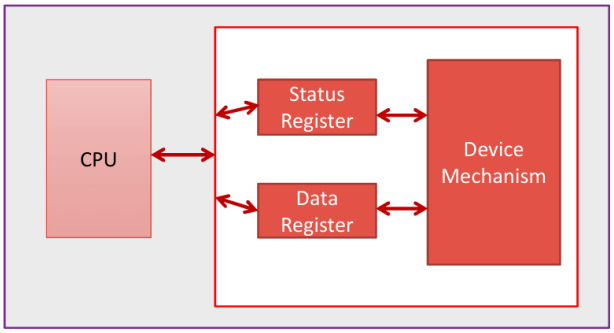
**Additional Features**

- **Analog Comparators**: 1
- **Serial Communication**: SPI, USART, TWI (I2C)

## PROGRAMMING INPUT & OUTPUT

**CPU Communication**

- The CPU communicates with devices by reading and writing to registers.
- **Data Register**: Read and write operations.
- **Status Register**: Read-only.
- **Examples**: UART, Timer, and many more.




**Microcontroller Programming Input and Output**

1. **I/O Mapped I/O**
2. **Memory Mapped I/O** (widely used)
    - In memory-mapped I/O, addresses are assigned for the registers in each I/O device.
    - Read and write instructions are used to communicate with the devices.

**Example: In 8051 Timer Register:** MOV TMOD, #01H

**Busy-Wait I/O**

- Checking an I/O device to see if it has completed a task by reading its status register is often called **polling**.
- This method is very inefficient.

  **Example:** AGAIN: JNB TF0, AGAIN

**Interrupts**

- Interrupts enable I/O devices to signal completion or status changes and force execution of a specific piece of code.

  **Example:** MOV IE, #82H

## PROGRAMMING LED & SWITCHES

**Digital I/O**

- **Read the state of an input pin**
- **Produce a logical high or low at an output pin**
- Each bit of a port can be programmed independently:
  - As input
  - As output
  - All for the same purpose

**ATmega328P**

- Two 8-bit I/O ports and one 7-bit I/O port.

**1.Input**

- Reads voltage applied to the pins.
- A small amount of current can change an input pin's state.
- State can change due to electronic interference (e.g., static discharges) when an input pin is unconnected.
- Use a **pull-down resistor** (connected to ground) when connecting a switch to an input pin.

**2.Output**

- Up to 40 mA power can be delivered to output circuits.
- Can power LEDs but not motors (requires a driver circuit).
- Cannot read sensors.
- Connecting an output pin directly to 5V or 0V is not advised.

**3.Input Pull-up**

- Sets a pin as input and connects an internal resistor.
- Inverts the behavior of the input mode:
  - HIGH means the sensor is OFF.
  - LOW means the sensor is ON.
- Internal resistor is at least **20 kilohms**.

**Pin Configuration**

- In an Arduino-based embedded system, use:

  **pinMode(pin, mode)**

  - **pin**: Number or variable (e.g., 1 to 13 or A0 to A5).

- ***Digital*** pins are ***input*** by default.

**Parameters**

- **Returns**: None
- **pin**: The number of the pin whose mode you wish to set.
- **mode**: INPUT, OUTPUT, or INPUT_PULLUP.

**Writing and Reading Pins**

- **Output Pin**:

  **digitalWrite(pin, value)**

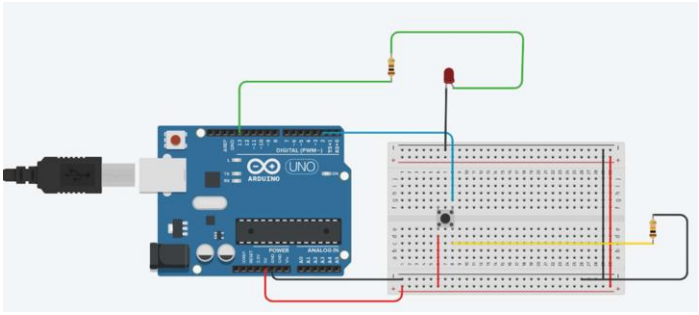
  - **Parameters**:
    - **pin**: The number of the pin you want to write.
    - **value**: HIGH or LOW.
  - **Returns**: None
- **Input Pin**:

  **digitalRead(pin)**

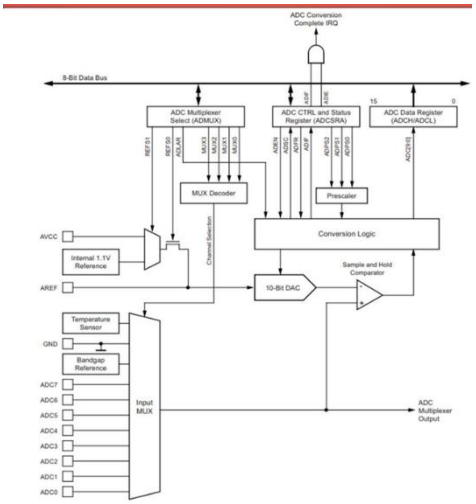
  - **Parameters**:
    - **pin**: The number of the pin you want to read (int).
  - **Returns**: HIGH or LOW.

**Example: Code to Read a Pushbutton and Control an LED**

```
const int buttonPin = 2;     // the number of the pushbutton pin
const int ledPin =  13;      // the number of the LED pin
int buttonState = 0;         // variable for reading the pushbutton status

void setup() {
  pinMode(ledPin, OUTPUT);   // initialize the LED pin as an output
  pinMode(buttonPin, INPUT); // initialize the pushbutton pin as an input
}

void loop() {
  buttonState = digitalRead(buttonPin); // read the state of the pushbutton value
  // check if the pushbutton is pressed. If it is, the buttonState is HIGH
  if (buttonState == HIGH) {
    digitalWrite(ledPin, HIGH); // turn LED on
  } else {
    digitalWrite(ledPin, LOW);  // turn LED off
  }
}
```



## ANALOG TO DIGITAL (ADC) CONVERTER




- An ADC is an electronic circuit whose digital output is proportional to its analog input.
- It effectively "measures" the input voltage and provides a binary output number proportional to its size.
- The list of possible analog input signals is vast (e.g., audio, video, medical, or climatic variables).
- The ADC almost always operates within a larger environment, often called a **data acquisition system**.

**ADC Conversion Formula:**

$$D = \frac{V_i}{V_r} \times 2^n$$

Where:

- *Vi* = Input voltage
- *Vr* = Reference voltage
- *n* = Number of bits in the converter output
- *D* = Digital output value

The output binary number *D* is an integer. For an n-bit number, it can take any value from 0 to (2^*n* −1).

**ADC Characteristics**

- ADC has maximum and minimum permissible input values.
- **Resolution** is a measure of how precisely an ADC can convert and represent a given input voltage:

$$Resolution = \frac{V_r}{2^n}$$

- Arduino ADC is **10-bit**, leading to a resolution of 5/1024 or approximately **4.88 mV**.

**Microcontroller ADC Pins**

- Not all pins on a microcontroller can perform analog to digital conversions.

- In an Arduino board, analog pins are labeled with an ‘A’ (e.g., A0 through A5).
- A 10-bit device can yield a total of 1024 different values:
  - An input of **0 volts** results in a decimal **0**.
  - An input of **5 volts** gives the maximum value of **1023**.

**ADC Value to Voltage Conversion**

- For example, if the ADC value is **434**:

$$434 = \frac{V_i}{5} \times 1023$$

$$V_i = 2.12\ \text{V}$$

**Analog I/O:**

**AREF pin and analogReference(type)**

- AREF pin and analogReference(type) allow setting an ADC reference voltage other than 5V.
- Increases the resolution available to analog inputs.
- Useful for operating at lower voltage ranges below +5V.
- **Must be declared before analogRead().**

**analogReference(type)**

**Parameters:**

- **type:** Specifies the reference type to use (DEFAULT, INTERNAL, INTERNAL1V1, INTERNAL2V56, or EXTERNAL).

**Returns:**

- None

**Analog I/O - analogRead(pin)**

- Reads the value from a specified analog pin with **10-bit resolution.**
- Works for analog pins **A0–A5.**
- Returns a value between **0 and 1023.**
- Takes **100 microseconds per reading.**
- Reading rate: **10,000 readings per second.**
- No need to declare **INPUT** or **OUTPUT** before use.

**analogRead(pin)**

**Parameters:**

- **pin:** The number of the analog input pin to read from (0-5).

**Returns:**

- **int (0 to 1023)**

**Analog I/O - analogWrite(pin, value)**

- Writes an **analog output** on a **digital pin** using **Pulse Width Modulation (PWM).**

- **PWM** uses digital signals to control analog devices.

**analogWrite(pin, value)**

**Parameters:**

- **pin:** The digital pin number you want to write to.
- **value:** The duty cycle between **0 (always off, 0%)** and **255 (always on, 100%).**

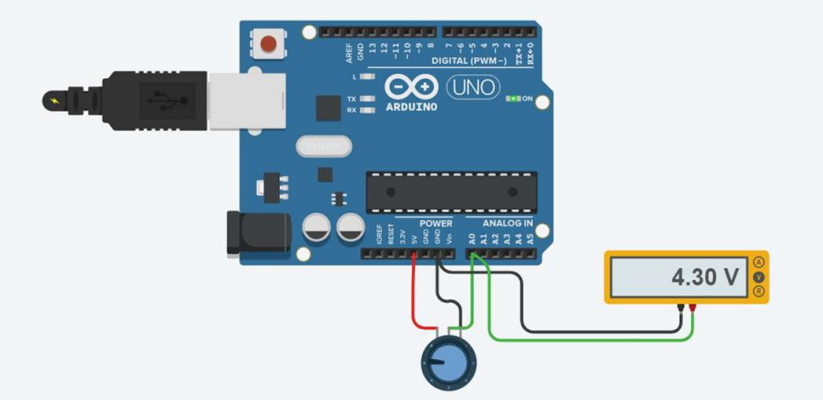
**Returns:**

- None
- Digital pins **#3, #5, #6, #9, #10, and #11** support **PWM output.**
- **No need to configure the pin as OUTPUT before using analogWrite().**

**EXAMPLE 1**

**Program an Arduino board to read an analog value from an analog input pin, convert the value into a digital value, and display it in a serial window.**

```
int sensorValue = 0;

void setup() {
  // Monitor output in serial port
  Serial.begin(9600); // Initiating serial communication with 9600 baud rate
}

void loop() {
  sensorValue = analogRead(A0);
  // Analog input is read from A0 pin
  Serial.println(sensorValue);
  // Printing the output in serial port
  delay(100); // Simple delay of 100ms
}
```

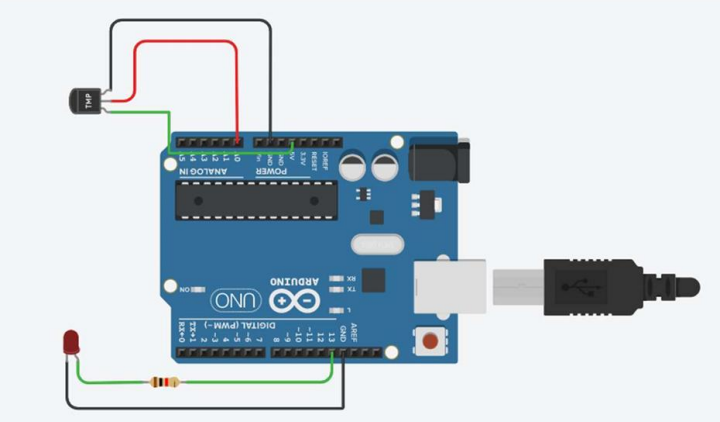



**EXAMPLE 2**

**Code an Arduino board to read a temperature sensor connected to an analog input A0 and display the temperature value in the serial window. Each time the value goes beyond 40 degrees Celsius, an LED connected to a digital output glows as a warning.**

```
const int lm35_pin = A0; /* LM35 O/P pin */
```

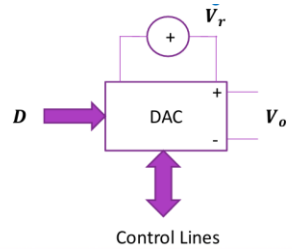
```
const int ledPin = 13;

void setup() {
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);
}

void loop() {
  int temp_adc_val;
  float temp_val;

  temp_adc_val = analogRead(lm35_pin);
  /* Read Temperature */
  temp_val = (temp_adc_val * 4.88); /* Convert ADC value to equivalent voltage */
  temp_val = (temp_val / 10); /* LM35 gives output of 10mV/°C */

  if (temp_val > 40) {
    digitalWrite(ledPin, HIGH); // Turn LED on
  } else {
    digitalWrite(ledPin, LOW); // Turn LED off
  }

  Serial.print("Temperature = ");
  Serial.print(temp_val);
  Serial.println(" Degree Celsius");

  delay(1000);
}
```



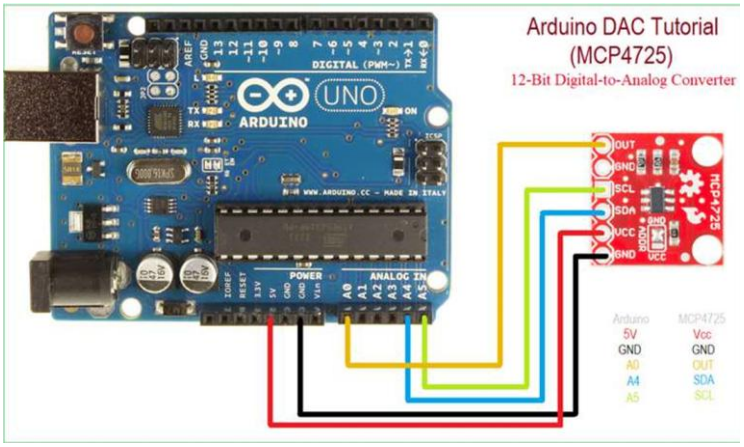
## DIGITAL TO ANALOG (DAC) CONVERTER

- A digital-to-analog converter (DAC) is a circuit that converts a binary input number into an analog output.
- DAC has a digital input, represented by **D**, and an analog output, represented by $\mathbf{V_o}$.

$$V_o = \frac{D}{2^n} \times V_r$$




- For each input digital value, there is a corresponding analog output.
- The number of possible output values is given by **$2^n$**.
- The step size is given by **$V_r$ / $2^n$**, which is called the resolution.
- The maximum possible output value occurs when **D = $(2^n - 1)$ $V_r$**, though this value is never quite reached.




## PWM GENERATION

**Introduction to PWM**

- **Pulse Width Modulation (PWM)** is a technique used to create a simple square wave by varying the **duty cycle.**
- It is used to control **analog variables**, such as **voltage** or **current**, using digital signals.
- PWM is widely applied in areas such as **telecommunications, robotic control, motor speed control**, and more.
- The **Arduino Uno** has **six PWM output pins: Pin 11, Pin 10, Pin 9, Pin 6, Pin 5, and Pin 3.**

**PWM Concepts**

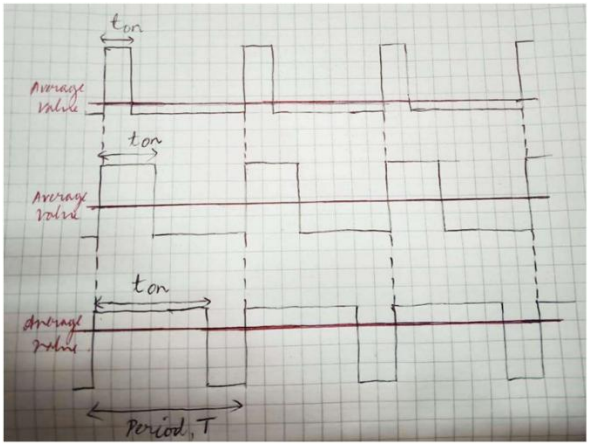
**1. Pulse Width and Duty Cycle**

- **Pulse Width (Period):** The duration of one complete cycle of the PWM signal.
- **Duty Cycle:** The percentage of time the signal remains **ON (high)** during a single cycle. It is calculated as:

$$Duty\ cycle = \frac{pulse\ on\ time}{pulse\ period} \times 100\% = \frac{t_{on}}{T} \times 100\%$$

- **100% Duty Cycle:** The signal is **always ON** (constant high voltage).
- **0% Duty Cycle:** The signal is **always OFF** (no voltage output).

## Applications of PWM




- **DC Motor Speed Control:**
  - The speed of a **DC motor** is proportional to the applied **DC voltage.**
  - Instead of using a **Digital-to-Analog Converter (DAC)** and a **power amplifier,** which are **expensive and bulky,** PWM can be used.
  - A **PWM signal** can directly drive a **power transistor,** which then controls the motor speed.

**PWM IN ARDUINO**

**Using the analogWrite() Function**

**analogWrite(pin, value);**

- This function generates a **PWM signal (analog output)** on a **digital pin.**
- The **value** parameter defines the **duty cycle,** ranging from:
  - 0 (always OFF, **0%**)
  - 255 (always ON, **100%**)

**Parameters**

- pin: The **PWM-capable digital pin** to output the signal.
- value: The **duty cycle value** between 0 and 255.

**Notes**

- PWM **simulates** an **analog output** using a **digital signal.**
- The following **Arduino Uno digital pins** can be used for **PWM output**:
  - **Pin 3, Pin 5, Pin 6, Pin 9, Pin 10, Pin 11**
- **No need to configure the pin as OUTPUT manually**—the analogWrite() function handles it automatically.

**Example Code: PWM for Motor Speed Control**

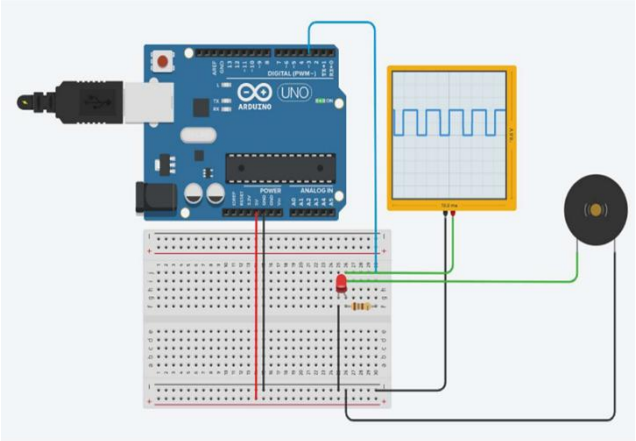
```
int pwmPin = 9;  // PWM output pin

void setup() {
  pinMode(pwmPin, OUTPUT); // Set pin as output
}

void loop() {
  analogWrite(pwmPin, 128); // Set 50% duty cycle (half speed)
  delay(2000); // Wait for 2 seconds

  analogWrite(pwmPin, 255); // Set 100% duty cycle (full speed)
  delay(2000); // Wait for 2 seconds
```

```
  analogWrite(pwmPin, 0); // Set 0% duty cycle (motor off)
  delay(2000); // Wait for 2 seconds
}
```

**EXAMPLE 1**

**Code an Arduino board to control the brightness of an LED. Monitor the PWM waveform in an oscilloscope and interface a buzzer to listen to different tones.**

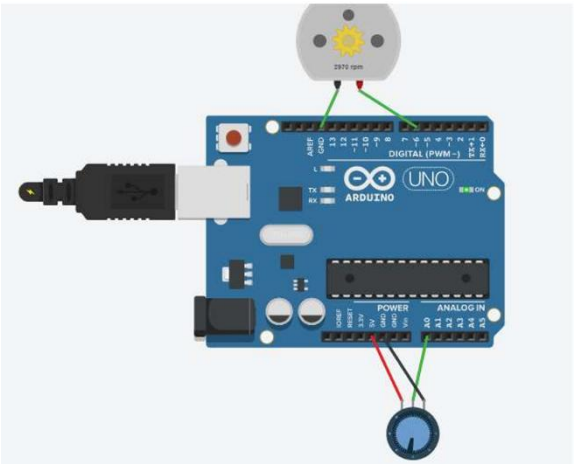
```
int brilho = 0;

void setup() {
  pinMode(3, OUTPUT);
}

void loop() {
  for (brilho = 0; brilho <= 255; brilho += 5) {
    analogWrite(3, brilho);
    delay(30); // Wait for 30 milliseconds
  }
  for (brilho = 255; brilho >= 0; brilho -= 5) {
    analogWrite(3, brilho);
    delay(30); // Wait for 30 milliseconds
  }
}
```



**EXAMPLE 2**

**Code an Arduino board to control the speed of a DC motor using a potentiometer.**

```
int potvalue;

void setup() {
  pinMode(A0, INPUT);
  pinMode(6, OUTPUT);
}

void loop() {
  potvalue = analogRead(A0) / 4;
  analogWrite(6, potvalue);
}
```



**TIMER/COUNTER**

Timers and counters play a crucial role in embedded systems by providing **time- or count-related events** with minimal processor and software overhead.

**Applications of Timers and Counters:**

- **Timing References**: Used for control sequences.
- **System Ticks**: Provides periodic interrupts for operating systems.
- **Waveform Generation**: Helps in generating PWM and other signal waveforms.

- **Baud Rate Generation**: Used in serial communication.
- **Audible Tone Generation**: Produces sound signals.

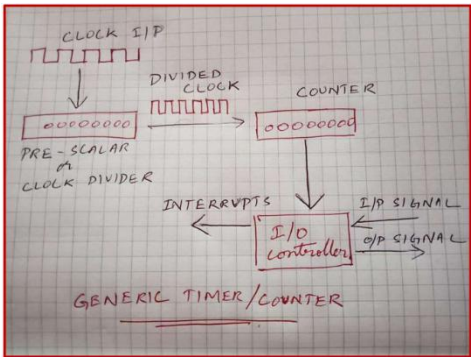
**Timers vs. Counters:**

Timers and counters are differentiated based on their functionality rather than their logic.

| Type | Function |
|---|---|
| **Timer** | Works with periodic signals. |
| **Counter** | Works with aperiodic signals (event-driven). |

**Clock Source for Timers/Counters:**

- The timing signal is derived from a **clock input**, which can be:
  - **Internal** (generated by the microcontroller).
  - **External** (connected via a separate pin).
- **Clock Division:**
  - A **divider** or **pre-scaler** is used to slow down the clock.
  - The **pre-scaler** divides the clock frequency based on the value written into its register.
  - The divided clock is passed to a counter, which operates in either **count-up** or **count-down** mode.




---

**Counter Operations:**

- When the counter reaches **zero**, an **event** occurs.
- Events may include:
  - **Interrupt generation** (triggering an external event).
  - **External signal change** (such as toggling an I/O pin).

**I/O Control Block in Timers/Counters:**

- **Generates Interrupts**: Triggers an interrupt when a specific condition is met.
- **Controls Counters**: Can start, stop, or modify counter operations based on external signals.

**TIMERS IN EMBEDDED SYSTEMS**

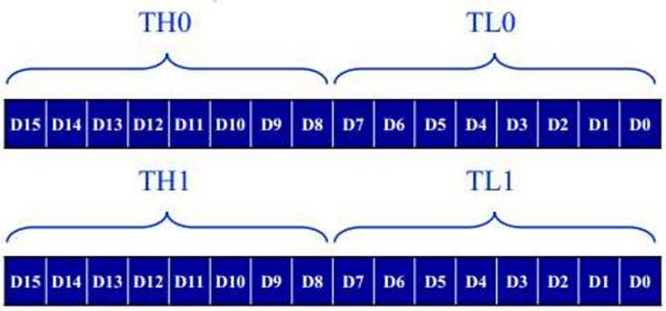
Timers play a critical role in embedded systems by managing time-related tasks efficiently.

**8051 Timer Registers**

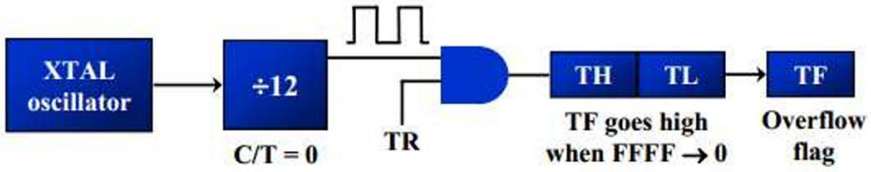
The **8051 microcontroller** includes the following registers for timer operations:

1. **TH & TL (Timer/Counter Register)**
   - Holds the value required for generating a time delay.

- Accessed as **low byte (TL0/TL1)** and **high byte (TH0/TH1).**




2. **TMOD (Timer Mode Register)**
   - Defines the mode of timer operation.
3. **TCON (Timer Control Register)**
   - Controls the timer operation.




**Calculating Values for TH & TL Registers**

To determine the values to be loaded into the **TH** and **TL** registers:

1. **Divide the desired time delay by 1.085 µs** (if the clock frequency is **11.0592 MHz**).
2. **Perform (65536 - n)**, where **n** is the decimal result from Step 1.
3. **Convert the result from Step 2 into a hexadecimal value** (yyxx format).
4. **Set TL = xx and TH = yy** to generate the time delay.

**Example: 500 µs Time Delay Calculation**

- **Step 1**: 500 µs÷1.085 µs = 461 pulses
- **Step 2**: P = 65536 – 461 = 65075
- **Step 3**: Convert **65075** to hexadecimal → **FE33**
- **Step 4**: Set **TH1 = 0xFE** and **TL1 = 0x33**

**TIMERS IN ARDUINO**

The **Arduino Uno** features **three timers** that control different functions:

1. **Timer 0 (8-bit timer)**: Used for functions like delay(), millis().
2. **Timer 1 (16-bit timer)**: Used in the **Servo library.**
3. **Timer 2 (8-bit timer)**: Used for functions like tone().

**Arduino Timer Details**

- The **internal clock** runs at **16 MHz.**
- Each count takes **62 nanoseconds.**
- **Pre-scalars** can be updated to change frequency and counting modes.

- **Interrupts** can be generated when the timer reaches a specific count.

**Arduino Timer Registers**

1. **Timer/Counter Control Registers (TCCRnA/B)**
    - Controls the **timer mode** and **pre-scalar settings.**
    - Available **pre-scalars: 1, 8, 64, 256, 1024.**
2. **Timer/Counter Register (TCNTn)**
    - Holds the counter value.
    - Can be **pre-loaded** with a specific value.

**Example: Pre-loader Calculation for Timer 1 (2s Delay)**

TCNT1 = 65535 – ($16\times10^6\times2$ / 1024) = 34285

**Step 2: Calculate the Timer Overflow Time**

Each tick of Timer1 takes:

$$\text{Timer Tick Time} = \frac{1}{\text{Clock Frequency}/\text{Pre-scaler}}$$

$$= \frac{1}{16,000,000/1024}$$

$$= \frac{1024}{16,000,000}$$

$$= 64\,\mu s \text{ per tick}$$

The maximum count for a **16-bit timer** is **65536 ticks**.

The **total time for a full cycle** of Timer1 (from 0 to 65535) is:

$$65536 \times 64\,\mu s = 4.096\,s$$

Since **4.096 seconds is too long**, we need to preload the timer so that it reaches **overflow** exactly at **2 seconds** instead.

**Step 3: Find the Required Number of Ticks**

To create a **2-second delay**, we need:

$$\frac{2 \text{ sec}}{64\,\mu s} = 31250 \text{ ticks}$$

Since Timer1 **overflows at 65536**, the preloader value is:

$$\text{TCNT1} = 65536 - 31250$$

$$= 34285$$

**Arduino Timer Functions**

1. **delay(ms)**
    - Pauses the program execution for a given time in milliseconds.

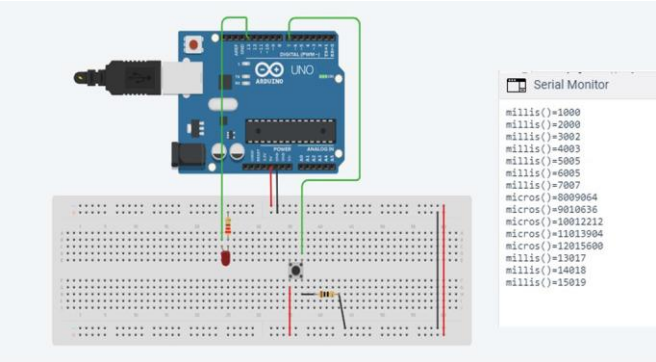
- **Example**: delay(1000); // Delays for 1 second
- **Returns**: None

2. **delayMicroseconds(us)**
   - Provides a shorter delay in **microseconds.**
   - **Example**: delayMicroseconds(1000); // Delays for 1 millisecond
   - **Returns**: None
3. **millis()**
   - Returns the number of **milliseconds** since the microcontroller started running.
   - Uses **Timer 1** for tracking time.
   - **Example**: unsigned long time = millis();
   - **Returns**: unsigned long (milliseconds since start).
4. **micros()**
   - Returns the number of **microseconds** since the program started.
   - **Example**: unsigned long time = micros();
   - **Returns**: unsigned long (microseconds since start).

**Q) Code an Arduino board to read a pushbutton connected to a digital input to display millis() when ON and display micros() when OFF. Blink an LED connected to a digital output with 500 ms delay.**

```
#define LED 13

int buttonState = 0;

void setup() {
  Serial.begin(9600);        // Initialize serial communication
  pinMode(LED, OUTPUT);      // Set LED as output
  pinMode(7, INPUT_PULLUP);  // Use internal pull-up resistor for button
}

void loop() {
  // Blink LED every 500ms
  digitalWrite(LED, HIGH);
  delay(500);
  digitalWrite(LED, LOW);
  delay(500);

  // Read button state
  buttonState = digitalRead(7);

  if (buttonState == LOW) {  // Button pressed
    Serial.print("micros() = ");
    Serial.println(micros());  // Print time in microseconds
  } else {  // Button released
```

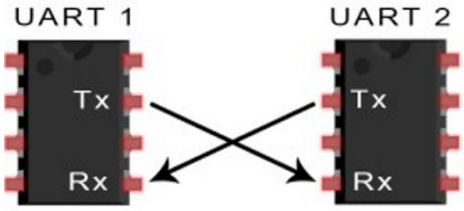
```
    Serial.print("millis() = ");
    Serial.println(millis());  // Print time in milliseconds
  }
}
```




## UART

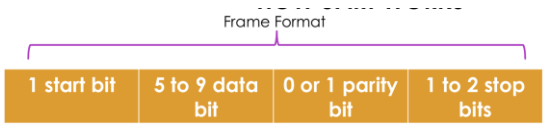
### What is UART?

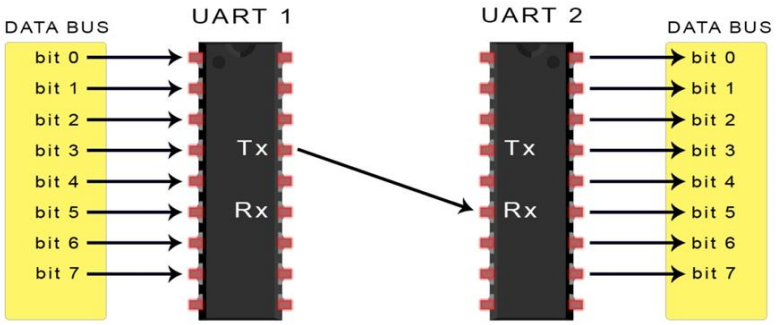
- **UART** stands for **Universal Asynchronous Receiver/Transmitter.**
- It is a hardware communication protocol used for serial communication.
- **It does not require a clock signal,** making it an **asynchronous** communication protocol.
- Only **two wires** are needed for communication:
    - **Tx (Transmit)** – Sends data.
    - **Rx (Receive)** – Receives data.



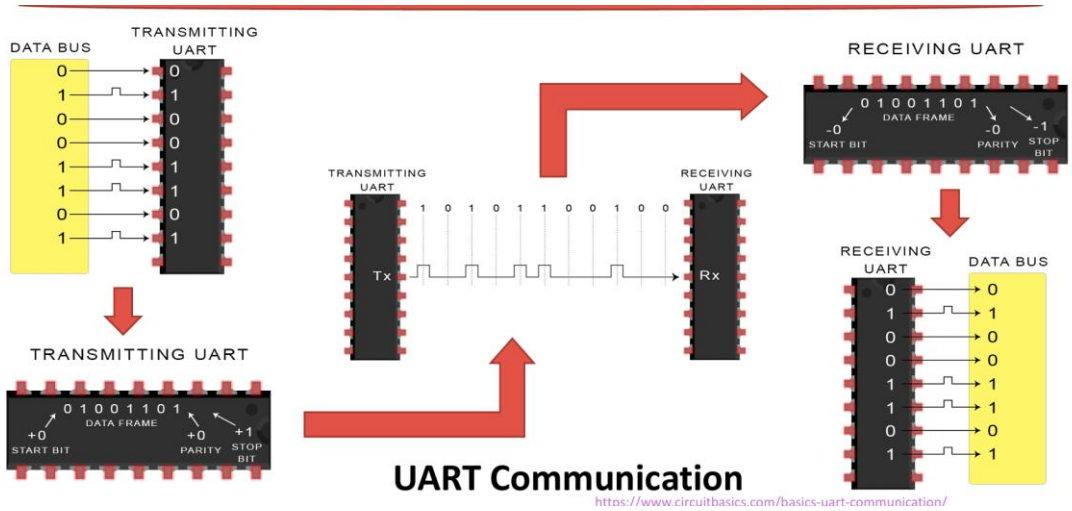

### How UART Works

1. **Data Transmission Process**
    - The sender takes data in **parallel format** from a data bus.
    - It converts this parallel data into **serial** format for transmission.
    - A **start bit** and a **stop bit** are added to indicate the beginning and end of data transmission.
    - A **parity bit** may also be added for **error checking.**
    - The UART sends data **one bit at a time** over the Tx line.






2. **Data Reception Process**
    - The receiving UART detects the **start bit**.
    - It reads the incoming **serial** data and removes the **start bit, stop bit, and parity bit**.
    - The data is then **converted back to parallel** format for processing.
    - The receiver verifies the **baud rate** and data integrity before accepting the data.




## Key Characteristics of UART

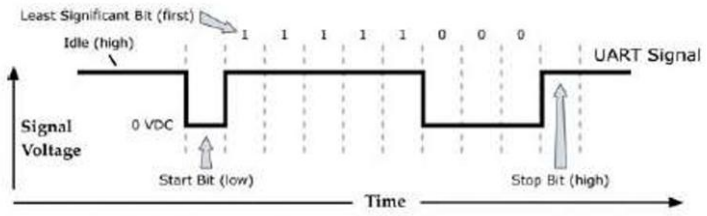
| Parameter | Description |
|---|---|
| **Wires Used** | 2 (Tx, Rx) |
| **Maximum Speed** | Up to **115200 baud rate** (depends on hardware) |
| **Synchronous?** | **No** (It is asynchronous, meaning it does not use a clock signal) |
| **Serial?** | **Yes** (Data is transmitted serially, one bit at a time) |
| **Max Number of Masters** | 1 (Only one device can control communication) |
| **Max Number of Slaves** | 1 (One device at a time receives the data) |

---

**Baud Rate**

- **Baud rate** refers to the speed of data transmission.
- It is measured in **bits per second (bps)**.

- Common baud rates: **9600, 19200, 38400, 57600, 115200 bps.**
- **Both transmitting and receiving UARTs must have the same baud rate** to communicate properly.




**A common asynchronous serial data format**

**Advantages of UART**

✅ **Simple & Easy to Use** – Only two wires required.
✅ **Error Detection** – Parity bits help detect transmission errors.
✅ **No Clock Required** – Saves hardware resources and reduces complexity.

**Disadvantages of UART**

❌ **Limited Speed** – Maximum baud rate is much lower than SPI or I2C.
❌ **One-to-One Communication** – Cannot handle multiple devices at the same time.

**UART IN ARDUINO**

**UART (Universal Asynchronous Receiver/Transmitter)** allows the Atmega chip on Arduino to perform serial communication while executing other tasks, using a *64-byte serial buffer*.

**Key Points about UART:**

- **Serial Communication**: All Arduino boards have at least one serial port, also known as UART or USART, to facilitate communication between the board and a computer or other devices.
- **Pins**: The Arduino uses digital pins 0 (RX) and 1 (TX) for communication. These pins are also used for communication with a computer via USB. If you use the serial functions, you cannot use pins 0 and 1 for other digital inputs or outputs.

**USART Registers for Arduino Serial Communication:**

1. **UCSRnA (USART Control and Status Register A):**
    - This register provides status information and controls certain aspects of UART operation.
2. **UCSRnB (USART Control and Status Register B):**
    - This register is used to enable or disable USART features like interrupts and transmitter/receiver functionality.
3. **UCSRnC (USART Control and Status Register C):**
    - This register configures the communication mode, frame format, and parity settings.
4. **UDRn (USART Data Register):**
    - The UDR0 register is used for both transmitting and receiving data.

- Writing to UDR0 places data into the transmit buffer.
- Reading from UDR0 retrieves data from the receive buffer.

**Arduino Serial Functions:**

- **if (Serial)**: Checks if the USB serial connection is open.
- **Serial.available()**: Returns the number of bytes available for reading from the serial port.
- **Serial.begin()**: Initializes the serial communication at a specified baud rate.
- **Serial.end()**: Disables serial communication, freeing RX/TX for use as general I/O pins.
- **Serial.find()**: Reads data from the serial buffer until a specified target is found.
- **Serial.read()**: Reads incoming serial data.
- **Serial.println()**: Prints data as ASCII text followed by a carriage return and newline character.
- **Serial.readString()**: Reads characters from the serial buffer into a string.
- **Serial.write()**: Writes binary data to the serial port.

**EXAMPLE-1: Print data received through serial communication on the serial monitor of Arduino**

```
void setup() {
  Serial.begin(9600); // Set up serial library baud rate to 9600
}

void loop() {
  if (Serial.available()) { // If number of bytes available for reading from serial port
    Serial.print("I received: "); // Print "I received:"
    Serial.write(Serial.read()); // Send back what was read
  }
}
```

**EXAMPLE-2: Arduino code for serial interface to blink LED when 'a' is received**

```
int inByte; // Stores incoming command

void setup() {
  Serial.begin(9600);
  pinMode(13, OUTPUT); // LED pin
  Serial.println("Ready"); // Ready to receive commands
}

void loop() {
  if (Serial.available() > 0) { // A byte is ready to receive
    inByte = Serial.read();
    if (inByte == 'a') { // If received byte is 'a'
      digitalWrite(13, HIGH);
```

```
      Serial.println("LED - On");
    } else { // If received byte isn't 'a'
      digitalWrite(13, LOW);
      Serial.println("LED - Off");
    }
  }
}
```

**Additional Serial Function**

**This function can also be used:**

```
if (Serial.find("a")) {
  // Executes when 'a' is found in the serial input buffer
}
```

**ARDUINO SOFTWARESERIAL LIBRARY**

The **SoftwareSerial** library allows serial communication on digital pins other than the default hardware **RX (pin 0) and TX (pin 1)** of the Arduino. This is achieved using software to replicate the functionality of hardware serial communication, hence the name **"SoftwareSerial."**

**Key Features:**

- Enables serial communication on **additional digital pins**.
- Supports **multiple software serial ports** with speeds up to **115200 bps**.
- Useful for **communicating with multiple serial-enabled devices**.

**Limitations:**

- **Maximum RX speed** is **57600 bps**.
- **RX does not work** on **pin 13**.
- When using **multiple software serial ports, only one can receive data at a time.**

**SoftwareSerial Library Functions:**

| Function | Description |
|---|---|
| **SoftwareSerial(rxPin, txPin)** | Creates a software serial object using the specified RX and TX pins. |
| **available()** | Returns the number of bytes available for reading from the serial port. |
| **begin(speed)** | Initializes serial communication with the specified baud rate. |
| **overflow()** | Checks if a buffer overflow has occurred. |
| **peek()** | Returns the next character received without removing it from the buffer. |
| **print()** | Works the same as Serial.print(), sending data as ASCII text. |

| | |
|---|---|
| **read()** | Reads and returns the next character received on the RX pin. |
| **listen()** | Enables the selected serial port to receive data. |
| **write(data)** | Sends binary data to the TX pin. |

**Example Code: Using SoftwareSerial Library**

```
#include <SoftwareSerial.h>

SoftwareSerial mySerial(10, 11);  // RX on pin 10, TX on pin 11

void setup() {
    Serial.begin(9600);       // Start hardware serial communication
    mySerial.begin(9600);     // Start software serial communication
}

void loop() {
    if (mySerial.available()) {
        char received = mySerial.read();  // Read incoming data
        Serial.print("Received: ");
        Serial.println(received);  // Print to Serial Monitor
    }

    if (Serial.available()) {
        char sendData = Serial.read();  // Read from Serial Monitor
        mySerial.write(sendData);  // Send data via software serial
    }
}
```

**EXAMPLE-3: Arduino code to Receives from the hardware serial, sends to software serial**

```
#include <SoftwareSerial.h>

SoftwareSerial mySerial(10, 11); // RX, TX

void setup() {
  Serial.begin(57600); // Open serial communications and wait for port to open:
  while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
  }
  mySerial.begin(4800); // Set the data rate for the SoftwareSerial port
  mySerial.println("Hello, world?");
}

void loop() { // Run over and over
  if (mySerial.available()) {
    Serial.write(mySerial.read());
  }
  if (Serial.available()) {
    mySerial.write(Serial.read());
  }
}
```

# MEMORY INTERFACE

## EEPROM INTERFACE USING SPI

### What is EEPROM?

EEPROM (**Electrically Erasable Programmable Read-Only Memory**) is a type of **non-volatile memory**, meaning that it retains stored data even when power is lost. It is useful for storing settings, logs, or any data that needs to persist across power cycles.

### 1. Why Use SPI for EEPROM Communication?

Many EEPROM chips, such as the **AT25HP512 Atmel EEPROM**, use the **SPI (Serial Peripheral Interface)** protocol for communication. SPI is a high-speed, synchronous **serial communication protocol** that allows microcontrollers (like Arduino) to **efficiently read and write** to external memory devices like EEPROM.

### Advantages of SPI for EEPROM

✓ **Fast data transfer** compared to I2C.
✓ **Simple wiring** (only 4 main connections required).
✓ **Supports multiple devices** using unique **Chip Select (CS) pins.**

### 2. How SPI Works with EEPROM

SPI requires **one master (e.g., Arduino)** and one or more **slave devices (e.g., EEPROM).**

### SPI Communication Pins

| SPI Signal | Description |
|---|---|
| **MISO (Master In Slave Out)** | Used by the **EEPROM** to send data **to Arduino.** |
| **MOSI (Master Out Slave In)** | Used by **Arduino** to send data **to EEPROM.** |
| **SCK (Serial Clock)** | Synchronizes data transfer, **generated by the master (Arduino).** |
| **CS (Chip Select or Slave Select - SS)** | Used to **activate a specific SPI device** (EEPROM in this case). |

📌 **Key Rule:** The master (Arduino) generates the **clock signal (SCK),** and data flows based on this timing.

### 3. SPI Control Registers in Arduino

**Types of Registers Used**

1. **Control Registers (SPCR)**
    - Configures SPI settings (**speed, polarity, etc.**).
    - Each **bit in SPCR controls a different setting.**
    - Example: SPCR |= (1 << SPE); enables SPI.
2. **Data Register (SPDR)**
    - Holds the **byte to be sent or received** via SPI.
    - When SPI.transfer(byte) is called, the **data is stored in SPDR** and sent out **via MOSI.**
3. **Status Register (SPSR)**

- Indicates **when data transfer is complete.**
- The **7th bit of SPSR (SPI Interrupt Flag - SPIF)** is **set to 1** when data is shifted.

**SPCR Register**

Each bit in the **SPCR** register has a specific function:


| Bit | Name | Function |
|---|---|---|
| **7** | **SPIE (SPI Interrupt Enable)** | Enables SPI interrupt when set to **1** |
| **6** | **SPE (SPI Enable)** | Enables the SPI module when set to **1** |
| **5** | **DORD (Data Order)** | **1** → Sends data **Least Significant Bit (LSB) first**<br>**0** → Sends data **Most Significant Bit (MSB) first** |
| **4** | **MSTR (Master/Slave Select)** | **1** → **Master mode** (Arduino controls communication)<br>**0** → **Slave mode** (Arduino listens for communication) |
| **3** | **CPOL (Clock Polarity)** | **1** → Clock is **idle HIGH**<br>**0** → Clock is **idle LOW** |
| **2** | **CPHA (Clock Phase)** | **1** → Samples data on **falling edge** of the clock<br>**0** → Samples data on **rising edge** of the clock |
| **1-0** | **SPR1, SPR0 (SPI Clock Rate Select)** | Sets SPI **clock speed**:<br>**00** → **Fastest (4MHz)**<br>**11** → **Slowest (250kHz)** |

---

**Understanding CPOL and CPHA (Clock Polarity & Phase)**

CPOL and CPHA determine **when data is sampled** relative to the clock signal.

| CPOL | CPHA | Mode | Clock Idle State | Data Sampling |
|---|---|---|---|---|
| **0** | **0** | Mode 0 | **Low (0)** | **Rising edge** |
| **0** | **1** | Mode 1 | **Low (0)** | **Falling edge** |
| **1** | **0** | Mode 2 | **High (1)** | **Falling edge** |
| **1** | **1** | Mode 3 | **High (1)** | **Rising edge** |

---

**Example: Configuring SPI on Arduino**

```
SPCR = 0b01010000; // Set SPI Control Register
```

This **binary value (01010000)** means:

- **SPIE = 0** (Disable SPI interrupt)
- **SPE = 1** (Enable SPI)
- **DORD = 0** (MSB first)
- **MSTR = 1** (Master mode)
- **CPOL = 0** (Clock idle LOW)
- **CPHA = 0** (Sample on rising edge)
- **SPR1 = 0, SPR0 = 0** (Fastest speed, 4MHz)

## 1. Introduction to AT25HP512 EEPROM

The **AT25HP512** is a **65,536-byte (512Kb)** serial EEPROM that communicates using the **SPI (Serial Peripheral Interface) protocol**. It is organized into **512 pages**, with each page containing **128 bytes**.

### Key Features of AT25HP512

- Supports **SPI Modes 0 and 3**.
- Operates at **1.8V to 5V**, with speeds up to **10MHz at 5V**.
- **Writes only in 128-byte pages**, but **reads from 1 to 128 bytes** at a time.
- Includes **write protection features**.
- Has a **Hold pin** that can pause operations.

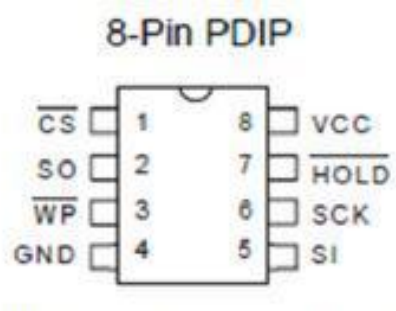
## 2. SPI Communication with EEPROM

The **AT25HP512 EEPROM** uses **SPI (Serial Peripheral Interface)**, which requires four main connections:

| SPI Signal | Description | EEPROM Pin | Arduino Pin |
|---|---|---|---|
| **MISO (Master In Slave Out)** | Data from EEPROM to Arduino | Pin 2 | Pin 12 |
| **MOSI (Master Out Slave In)** | Data from Arduino to EEPROM | Pin 5 | Pin 11 |
| **SCK (Serial Clock)** | Clock generated by Arduino | Pin 6 | Pin 13 |
| **CS (Chip Select / SS)** | Enables EEPROM (Active LOW) | Pin 1 | Pin 10 |

📌 **Other Connections:**

- **VCC (Pins 3, 7, 8)** → Connected to **5V**.
- **GND (Pin 4)** → Connected to **GND**.




---

### Pin Descriptions

| Pin Name | Function |
|---|---|
| **CS (Chip Select)** | Enables the EEPROM for communication when pulled **LOW** (active-low). |
| **SCK (Serial Data Clock)** | Provides the clock signal to synchronize data transfer between the microcontroller and EEPROM. |
| **SI (Serial Data Input)** | Also known as **MOSI (Master Out Slave In)**, this pin receives data from the master (microcontroller). |

| | |
|---|---|
| **SO (Serial Data Output)** | Also known as **MISO (Master In Slave Out)**, this pin sends data from EEPROM to the microcontroller. |
| **GND (Ground)** | Common ground connection. |
| **VCC (Power Supply)** | Provides power to the EEPROM (typically **1.8V - 5V**). |
| **WP (Write Protect)** | Enables or disables write protection. When LOW, it **prevents writing** to certain memory areas. |
| **HOLD** | (Low) Pauses communication without resetting the operation. Useful for multi-device SPI setups. |

**Typical Arduino Connection**

If interfacing with an **Arduino** using SPI:

| **EEPROM Pin** | **Arduino Pin** |
|---|---|
| **CS** | Digital Pin 10 (SS - Slave Select) |
| **SCK** | Digital Pin 13 (SCK - Serial Clock) |
| **SI (MOSI)** | Digital Pin 11 (MOSI - Master Out Slave In) |
| **SO (MISO)** | Digital Pin 12 (MISO - Master In Slave Out) |
| **GND** | GND |
| **VCC** | 5V or 3.3V (as required) |
| **WP** | GND (to allow writing) |
| **HOLD** | HIGH (unless pausing data transfer) |

## 3. EEPROM SPI Communication Process

### A. Writing to EEPROM

1. **Enable Write Operation**
   - EEPROM **must be write-enabled** before writing data.
   - Use the **WREN (Write Enable) opcode (0x06).**
2. **Send Write Instruction**
   - Pull **CS LOW** to enable EEPROM.
   - Send **WRITE opcode (0x02).**
   - Send **16-bit memory address** (**MSB first**).
   - Send **128 bytes of data (page write).**
   - Pull **CS HIGH** and **wait 10ms** for the write cycle to complete.

### B. Reading from EEPROM

1. **Send Read Instruction**
   - Pull **CS LOW** to enable EEPROM.
   - Send **READ opcode (0x03).**
   - Send **16-bit memory address** (**MSB first**).

- Send a **dummy byte (0x00)** to shift data out.
- EEPROM **sends back the requested data byte** on **MISO.**
- Pull **CS HIGH** to release EEPROM.

---

```
#define DATAOUT 11//MOSI
#define DATAIN  12//MISO
#define SPICLOCK  13//sck
#define SLAVESELECT 10//ss

//opcodes
#define WREN  6
#define WRDI  4
#define RDSR  5
#define WRSR  1
#define READ  3
#define WRITE 2
```

```
byte eeprom_output_data;
byte eeprom_input_data=0;
byte clr;
int address=0;
//data buffer
char buffer [128];
```

```
void setup()
{
  Serial.begin(9600);
  pinMode(DATAOUT, OUTPUT);
  pinMode(DATAIN, INPUT);
  pinMode(SPICLOCK,OUTPUT);
  pinMode(SLAVESELECT,OUTPUT);
  digitalWrite(SLAVESELECT,HIGH); //disable device
```

```
// SPCR = 01010000
  //interrupt disabled,spi enabled,msb 1st,master,clk low when idle,
  //sample on leading edge of clk,system clock/4 rate (fastest)
  SPCR = (1<<SPE)|(1<<MSTR);
  clr=SPSR;
  clr=SPDR;
  delay(10);
```

```
//fill buffer with data
  fill_buffer();
  //fill eeprom w/ buffer
  digitalWrite(SLAVESELECT,LOW);
  spi_transfer(WREN); //write enable
  digitalWrite(SLAVESELECT,HIGH);
```

```
delay(10);
  digitalWrite(SLAVESELECT,LOW);
  spi_transfer(WRITE); //write instruction
  address=0;
  spi_transfer((char)(address>>8));   //send MSByte address first
  spi_transfer((char)(address));      //send LSByte address
  //write 128 bytes
  for (int I=0;I<128;I++)
  {
    spi_transfer(buffer[I]); //write data byte
  }
  digitalWrite(SLAVESELECT,HIGH); //release chip
  //wait for eeprom to finish writing
  delay(3000);
```

```
void loop()
{
  eeprom_output_data = read_eeprom(address);
  Serial.print(eeprom_output_data,DEC);
  Serial.print('\n',BYTE);
  address++;
  delay(500); //pause for readability
}
```

The spi_transfer function loads the output data into the data transmission register, thus starting the SPI transmission. It polls a bit to the SPI Status register (SPSR) to detect when the transmission is complete using a bit mask, SPIF. It then returns any data that has been shifted in to the data register by the EEPROM

```
char spi_transfer(volatile char data)
{
  SPDR = data;                    // Start the transmission
  while (!(SPSR & (1<<SPIF)))     // Wait for the end of the transmission
  {
  };
  return SPDR;                    // return the received byte
}
```

```
byte read_eeprom(int EEPROM_address)
{
  //READ EEPROM
  int data;
  digitalWrite(SLAVESELECT,LOW);
  spi_transfer(READ); //transmit read opcode
  spi_transfer((char)(EEPROM_address>>8));   //send MSByte address first
  spi_transfer((char)(EEPROM_address));      //send LSByte address
  data = spi_transfer(0xFF); //get data byte
  digitalWrite(SLAVESELECT,HIGH); //release chip, signal end transfer
  return data;
}
```

## PROGRAMMING KEYPAD AND DISPLAY

**7-SEGMENT DISPLAY IN EMBEDDED SYSTEMS**

**Introduction**

A **7-segment display** is an electronic display device used to represent numerical and some alphabetical information. It is widely used in applications such as **digital clocks, electronic meters, microwave ovens, and other consumer electronics**.

**Structure of a 7-Segment Display**

A **7-segment display** consists of **7 LEDs (Light Emitting Diodes)**, labeled **A to G**, arranged in the shape of the number "8". Additionally, there is an optional **8th segment (DP or H)**, which represents a decimal point. Each of these LEDs can be turned ON or OFF in different combinations to display numbers (0-9) and some characters (A-F in hexadecimal systems).

**Pin Configuration**

A **7-segment display** typically comes in a **10-pin package**:

- **8 pins** correspond to the **individual segments (A to G and DP).**
- **2 pins** are common terminals, which depend on the type of display used (Common Cathode or Common Anode).

**Types of 7-Segment Displays**

There are two configurations of **7-segment displays** based on their electrical connection:

1. **Common Cathode (CC)**
    - In this type, all the negative terminals (cathodes) of the LEDs are **connected to ground (GND).**

- To turn on a segment, a **HIGH (1)** signal must be applied to the corresponding segment pin.
- **Example:** To display "1", the **B and C segments** need to be activated (set HIGH).

2. **Common Anode (CA)**
   - In this type, all the positive terminals (anodes) of the LEDs are **connected to a common HIGH (Vcc).**
   - To turn on a segment, a **LOW (0)** signal must be applied to the corresponding segment pin.
   - **Example:** To display "1", the **B and C segments** must be set to LOW.

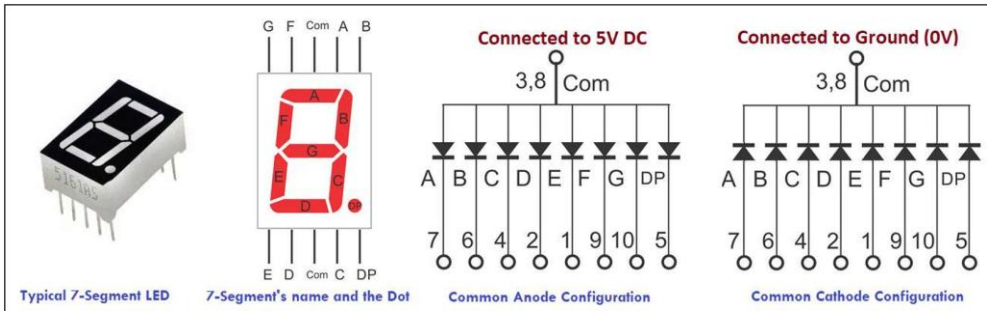
## Working Principle

- Each segment of the **7-segment display** is an LED that **glows** when current flows through it.
- By controlling which segments are turned ON/OFF, different numbers and characters can be displayed.
- A **microcontroller** or **Arduino** can be used to send signals to the display to control which segments are illuminated.

## Applications

7-segment displays are used in various applications such as:

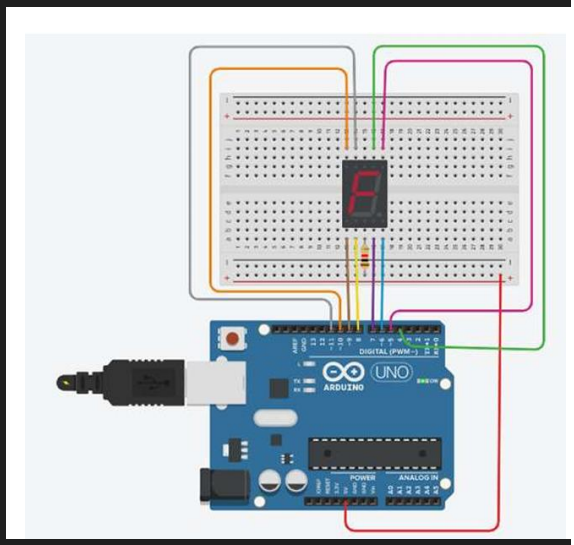
- **Digital clocks and watches**
- **Microwave ovens and stoves**
- **Electronic meters (voltmeter, ammeter, etc.)**
- **Elevator floor indicators**
- **Fuel dispensers at gas stations**



| Decimal Digit | Individual Segments Illuminated for Common Anode configuration | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | a | b | c | d | e | f | g | dp |
| . | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| 4 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 5 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 6 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 7 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 9 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |

**EXAMPLE: Write a program to display hexa decimal values from 0 to 9 on the 7-segment display interfaced with digital pins of Arduino Uno board.**

```
// Displays hexadecimal values from digit 0 - 9 on a 7-segment display

int segPins[] = {4, 5, 7, 8, 9, 11, 10, 6}; // { a b c d e f g . }
int segCode[10][8] = {
  //  a  b  c  d  e  f  g  .
  { 0, 0, 0, 0, 0, 0, 1, 1},  // 0
  { 1, 0, 0, 1, 1, 1, 1, 1},  // 1
  { 0, 0, 1, 0, 0, 1, 0, 1},  // 2
  { 0, 0, 0, 0, 1, 1, 0, 1},  // 3
  { 1, 0, 0, 1, 1, 0, 0, 1},  // 4
  { 0, 1, 0, 0, 1, 0, 0, 1},  // 5
  { 0, 1, 0, 0, 0, 0, 0, 1},  // 6
  { 0, 0, 0, 1, 1, 1, 1, 1},  // 7
  { 0, 0, 0, 0, 0, 0, 0, 1},  // 8
  { 0, 0, 0, 0, 1, 0, 0, 1},  // 9
};

void displayDigit(int digit) {
  for (int i = 0; i < 8; i++) {
    digitalWrite(segPins[i], segCode[digit][i]);
  }
}

void setup() {
  for (int i = 0; i < 8; i++) {
    pinMode(segPins[i], OUTPUT);
  }
}

void loop() {
  for (int n = 0; n < 10; n++) { // display digits 0 - 9
    displayDigit(n);
    delay(1000);
  }
}
```



**LIQUID CRYSTAL DISPLAY (LCD) IN EMBEDDED SYSTEMS**

## Introduction

A **Liquid Crystal Display (LCD)** is one of the most frequently used output devices for displaying data in embedded systems. It is widely used in **digital meters, calculators, clocks, mobile phones, and many other electronic devices** due to its *low power consumption* and *clear visibility*.

The name **Liquid Crystal Display** is derived from the **thin film of liquid crystal** inside the screen, which plays a crucial role in controlling the passage of light to create visible images.

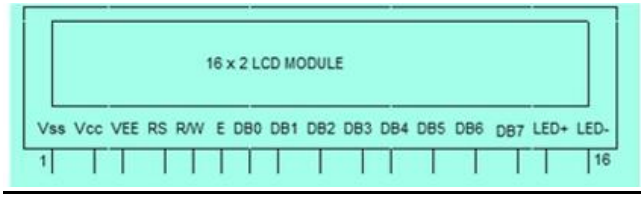
## Working Principle of LCD

- **In its normal state**, the **liquid crystals** inside the display are **twisted**, allowing light to pass through the screen.
- **When an electrical current is applied**, the liquid crystals **untwist**, blocking the light, which makes that portion of the screen appear black.
- This controlled blocking of light is used to **form characters and numbers** on the screen.

## LCD Screen Characteristics

- Unlike modern LED or OLED screens, **basic LCD screens do not have a specific resolution** but are instead categorized based on the **number of characters they can display**.
- **Example:** A **16x2 LCD** means the display **has 16 characters per line and 2 lines in total**.
- These types of LCDs **cannot display graphics**, only alphanumeric characters.

## Pin Configuration of LCD




| Pin | Name | Function |
|---|---|---|
| **Vss** | Ground (GND) | Connects to the ground of the circuit. |
| **Vcc** | Supply (+5V) | Provides power to the LCD module. |
| **VEE** | Contrast Control | Adjusts screen contrast (typically 0.4V to 0.9V). |
| **RS** | Register Select | **0** = Command Mode, **1** = Data Mode. |
| **R/W** | Read/Write | **0** = Write Data, **1** = Read Data. |
| **E** | Enable | A **High to Low** pulse enables the LCD. |
| **DB0-DB7** | Data Pins | Used to send **commands and data**. |
| **LED+ / LED-** | Backlight Control | Controls brightness of the LCD screen. |

## Using LCD with Arduino

Arduino provides a built-in **LiquidCrystal** library for interfacing with LCD modules.

### 1. Importing the Library

**#include <LiquidCrystal.h>**

This library allows communication with **character LCDs.**

### 2. Creating an LCD Object

The LCD can be connected in **4-bit mode** or **8-bit mode** using different configurations. Examples:

```
LiquidCrystal lcd(rs, enable, d4, d5, d6, d7);  // 4-bit mode
LiquidCrystal lcd(rs, rw, enable, d4, d5, d6, d7);  // 4-bit mode with Read/Write
LiquidCrystal lcd(rs, enable, d0, d1, d2, d3, d4, d5, d6, d7);  // 8-bit mode
LiquidCrystal lcd(rs, rw, enable, d0, d1, d2, d3, d4, d5, d6, d7);  // 8-bit mode with
Read/Write
```

For **most applications**, the **4-bit mode** is preferred since it requires fewer connections and saves digital pins on the microcontroller.

### 3. Initializing the LCD

Before using the LCD, it must be initialized in the **setup()** function:

**lcd.begin(16, 2);** // Initializes a 16x2 LCD

## Basic LCD Commands

| Command | Description |
|---|---|
| **lcd.print("Text")** | Displays text on the LCD screen. |
| **lcd.write(data)** | Displays a single character. |
| **lcd.clear()** | Clears the display. |
| **lcd.setCursor(col, row)** | Moves the cursor to a specific position. |
| **lcd.leftToRight()** | Sets text direction from left to right. |
| **lcd.scrollDisplayLeft()** | Scrolls the display to the left. |

**Example Code: Display "Hello World" on LCD**

```
#include <LiquidCrystal.h>

LiquidCrystal lcd(7, 8, 9, 10, 11, 12);  // RS, E, D4, D5, D6, D7

void setup() {
  lcd.begin(16, 2);   // Initialize 16x2 LCD
  lcd.print("Hello, World!");  // Print message
}

void loop() {
  // Do nothing
}
```

This code will display **"Hello, World!"** on the LCD screen.

## Advantages of LCD

✅ **No Image Burn-in** – Unlike CRT displays, LCDs do not suffer from image retention issues.
✅ **Thin and Lightweight** – LCDs are much thinner and more compact than traditional display systems.
✅ **Sharp Image Resolution** – Produces **crisp and clear** text with **no color bleeding.**
✅ **Not Affected by Magnetic Fields** – Unlike CRTs, LCDs **do not distort** due to magnetic fields.

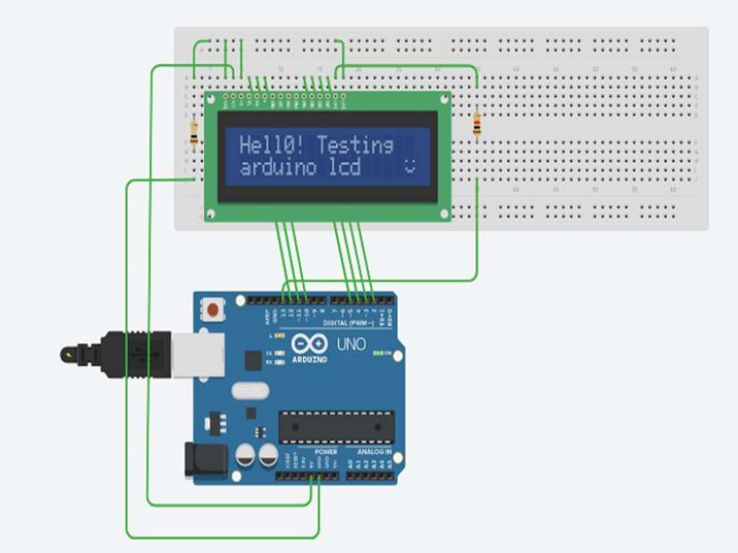
## Disadvantages of LCD

❌ **Less Energy Efficient** – LCDs **consume more power** compared to modern **LED** displays.
❌ **Lower Contrast Ratio** – LCDs rely on **backlight**, which limits contrast levels.
❌ **No True Blacks** – Since LCDs use a backlight, they **cannot achieve true black colors.**

**EXAMPLE Code an Arduino board to display "Hello! Testing" in the first line and "arduino lcd" in the second line. Also display a smiley in the second line**

```
#include <LiquidCrystal.h>

// Connections:
// rs (LCD pin 4) to Arduino pin 12
// rw (LCD pin 5) to Arduino pin 11
// enable (LCD pin 6) to Arduino pin 10
// LCD pin 15 to Arduino pin 13
// LCD pins d4, d5, d6, d7 to Arduino pins 5, 4, 3, 2

LiquidCrystal lcd(12, 11, 10, 5, 4, 3, 2);

byte smiley[8] = {
  B00000,
  B10001,
  B00000,
  B00000,
  B10001,
  B01110,
  B00000,
};

int backLight = 13; // pin 13 will control the backlight

void setup() {
  pinMode(backLight, OUTPUT);
  digitalWrite(backLight, HIGH); // turn backlight on

  lcd.begin(16, 2); // Initialize LCD with 16 columns and 2 rows
  lcd.clear();      // Clear the LCD screen
  lcd.setCursor(0, 0); // Set cursor to column 0, row 0
  lcd.print("Hell0! Testing "); // Display text

  lcd.setCursor(0, 1); // Set cursor to column 0, row 1
  lcd.print("Arduino lcd"); // Display message
```

```
  lcd.createChar(0, smiley); // Create custom smiley character
  lcd.setCursor(15, 1); // Set cursor to last column of second row
  lcd.write(byte(0)); // Display smiley face

}

void loop() {
  delay(1000);
  // Turn off the display:
  lcd.noDisplay();
  delay(500);
  // Turn on the display:
  lcd.display();
}
```




## KEYPAD IN EMBEDDED SYSTEMS

A **keypad** is an input device commonly used in embedded systems to allow user interaction. It consists of an **array of mechanical switches** arranged in a grid pattern, usually in a **matrix configuration** of rows and columns. Keypads are widely used in **ATMs, mobile phones, security systems, and industrial control panels.**

### Working Principle of a Keypad

A keypad works by **detecting a pressed key** based on the **contact made between a row and a column** in the matrix. It simplifies microcontroller interfacing by reducing the number of input pins required.

- **Matrix Arrangement:**
    - Each key in the keypad connects a specific **row** and **column.**
    - The **rows are connected to an output port,** while the **columns are connected to an input port** of the microcontroller.
    - The microcontroller **scans** the matrix by **setting rows HIGH one at a time** and **reading the columns** to check for a pressed key.

### Debouncing in Keypad

When a mechanical switch is pressed or released, it may **bounce** (rapidly turn ON and OFF) before settling into a stable state. This can lead to **multiple detections** of a single keypress, causing errors in input.

**Debouncing Techniques:**

1. **Hardware Debouncing**
    - Uses an **RC (resistor-capacitor) circuit** or **one-shot timer** to filter out unwanted voltage fluctuations.
2. **Software Debouncing**
    - **Wait-and-see technique:** The microcontroller waits for a short duration (typically **10ms**) and checks the key state again. If the key state remains the same, it is considered a valid keypress.
    - **Reading multiple times:** The switch state is read multiple times, and only if it remains stable, it is registered as a keypress.

**Scanning a 4×4 Matrix Keypad**

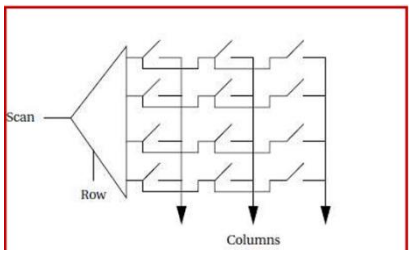
A **4×4 matrix keypad** consists of **4 rows and 4 columns**, requiring only **8 GPIO pins** instead of **16 individual pins**.

**Steps to Detect a Key Press**

1. Set **all rows HIGH** and **all columns as inputs**.
2. Set each row LOW, one at a time, while keeping others HIGH.
3. Read the **column inputs**.
4. If a column is detected LOW, the intersection of the active row and column identifies the pressed key.
5. Implement **debouncing** to ensure stable input detection.

**Encoded Keyboards**

An **encoded keyboard** uses a method where **each key is assigned a unique code** to identify the keypress.




- Instead of scanning the entire keypad matrix, an encoded keyboard **directly sends a key code** to the microcontroller.
- This approach reduces **software processing complexity**.
- Some **keypad encoders** use **multiplexing or shift registers** to reduce the number of connections.

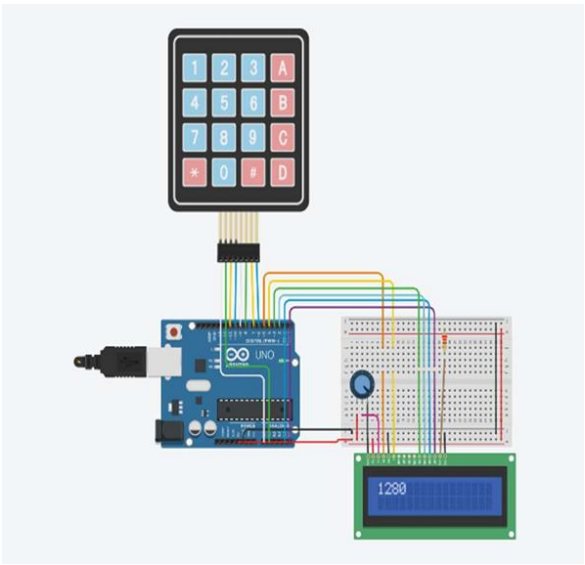
**Applications of Keypad**

- **ATMs and POS machines**
- **Security Systems (Door Locks, Access Control)**
- **Industrial Control Panels**
- **Smart Home Systems**
- **Digital Keyboards for Embedded Devices**

Keypads are an essential input component in **embedded systems** and **microcontroller-based projects**, providing a simple and effective way to collect user input.

**EXAMPLE Code an Arduino board to display a key press on a 4x4 keypad on a 16x2 LCD screen.**

```
#include <Keypad.h>
#include <LiquidCrystal.h>

LiquidCrystal lcd(5, 4, 3, 2, A4, A5);

const byte ROWS = 4; // Four rows
const byte COLS = 4; // Four columns

char keys[ROWS][COLS] = {
  {'1', '2', '3', 'A'},
  {'4', '5', '6', 'B'},
  {'7', '8', '9', 'C'},
  {'*', '0', '#', 'D'}
};

byte rowPins[ROWS] = {A0, A1, 11, 10}; // Connect to the row pinouts of the keypad
byte colPins[COLS] = {9, 8, 7, 6}; // Connect to the column pinouts of the keypad

int LCDRow = 0;

Keypad keypad = Keypad(makeKeymap(keys), rowPins, colPins, ROWS, COLS);

void setup() {
  Serial.begin(9600);
  lcd.begin(16, 2);
  lcd.setCursor(LCDRow, 0);
}

void loop() {
  char key = keypad.getKey();
  if (key) {
    Serial.println(key);
    lcd.print(key);
    lcd.setCursor(++LCDRow, 0);
  }
}
```




**QUESTION PAPER ANSWERS**

**Q1) Explain the different phases involved in the embedded system design process for designing a digital pulse oximeter to measure oxygen saturation level for the covid patients. apply top down approach**

**1. A 32 bit arm cortex M4 core with supports Digital Signal Processing for filtering signals**

**2. Measure oxygen saturation spo2 by observing pulse rate heartbeats per minute and display the values on digital interface.**

**3. Provide real time feedback and alert system 4. Photodiodes One red and infrared LED's are used as sensors in oximeter are powered by Li- ion or LiPo battery with charging circuitry**

**5. Continuously update the display with calculated spo2 and Pulse rate in oled. if needed implement warning messages for abnormal readings ,for example spo2 is lesser than 90%**

**Solution:**

**Phase 1: System Specification & Requirements Analysis**

**Objective:**

Design a **digital pulse oximeter** to measure **$SpO_2$ and pulse rate**, display the values on an **OLED screen**, and provide real-time alerts for abnormal readings.

**Key Functional Requirements:**

- Use a **32-bit ARM Cortex-M4 core** for **signal processing**.
- Measure **oxygen saturation ($SpO_2$) and heart rate** using **red and infrared LEDs** with **photodiodes**.
- Power the system using a **Li-ion/LiPo battery with charging circuitry**.
- Display **real-time values** on an **OLED screen** with alerts when **$SpO_2$ < 90%**.
- Provide **real-time feedback and alerts** for abnormal readings.

---

**Phase 2: High-Level System Architecture Design**

**1. Hardware Components Selection:**

- **Microcontroller:** ARM Cortex-M4 (with DSP support for filtering signals).
- **Sensors:** Red and Infrared LEDs + Photodiodes to detect oxygen levels.
- **Power Management:** Li-ion/LiPo battery with charging circuitry.
- **Display Interface:** OLED screen.
- **Alert System:** Buzzer or LED for real-time warnings.
- **Wireless Communication (Optional):** BLE/Wi-Fi for mobile app connectivity.

**2. Software Functional Blocks:**

- **Signal Acquisition:** Read sensor data (red/IR absorption).
- **Pre-processing:** Apply noise filtering using DSP.
- **Computation:** Calculate $SpO_2$ and pulse rate.
- **Display Management:** Update OLED with real-time values.
- **Alert Mechanism:** Trigger warnings if $SpO_2$ < 90%.

---

**Phase 3: Detailed Design & Implementation**

**1. Hardware Design**

- **Microcontroller Interface:**

- Connect **LEDs & photodiodes** to ADC of ARM Cortex-M4.
- Interface **OLED via I2C/SPI.**
- Connect a **buzzer/LED for alerts.**
- Implement a **battery management system (BMS).**

**2. Software Development**

- **Signal Processing (DSP):**
  - Remove noise from raw signals.
  - Extract **PPG waveform** (Photoplethysmography).
  - Use **Fast Fourier Transform (FFT)** for frequency analysis.
- **$SpO_2$ & Pulse Rate Calculation:**
  - Compute **Red/IR ratio** using Beer-Lambert law.
  - Apply **peak detection for pulse rate.**
- **Display & Alerts:**
  - Update OLED with **live $SpO_2$ and pulse rate.**
  - Show **warnings** if $SpO_2$ < 90%.
- **Power Management:**
  - Optimize **low-power modes.**
  - Implement **charging indicator.**

---

**Phase 4: Integration & Testing**

- **Unit Testing:** Test individual modules (sensor, display, alert).
- **System Testing:** Validate $SpO_2$ accuracy using reference oximeter.
- **Power Testing:** Measure battery efficiency.
- **Real-time Response Testing:** Ensure alerts trigger instantly.

---

**Phase 5: Deployment & Optimization**

- **Optimize DSP algorithms** for faster signal processing.
- **Improve battery efficiency** using power-saving techniques.
- **Add BLE/Wi-Fi for remote monitoring (if needed).**
- **Calibrate sensors periodically** for accuracy.

**Q) 2Write an Arduino program to develop an Automatic Toll Collection System for vehicles passing through a toll booth, detected using an IR sensor. The system uses a timer or counter for the following operations:**

**1. Vehicle Detection and Toll Charging:**

- When a vehicle passes the IR sensor, a timer starts. The vehicle's crossing duration is tracked.
- If the vehicle takes less than 10 seconds to cross, it is classified as a fast vehicle, and a standard toll fee is applied.
- If the vehicle takes more than 10 seconds, it is classified as a slow vehicle, and an additional surcharge is applied to the toll fee.

2. Vehicle Flow Rate Monitoring:
   - The counter keeps track of how many vehicles have passed through the toll booth in the last minute.

3. Overstay Detection:
   - If a vehicle remains in the IR sensor's range for longer than 30 seconds, an alert is raised by a buzzer. The buzzer stays on for 5 seconds and then turns off.

**Write the Arduino code to implement this system. Include a connection diagram for the setup and the program for implementation.**

```
#define IR_SENSOR_ENTRY 2  // IR sensor at entry point
#define IR_SENSOR_EXIT 3   // IR sensor at exit point
#define BUZZER_PIN 6       // Buzzer for overstay alert
#define LED_PIN 5          // LED to indicate vehicle classification

unsigned long entryTime = 0; // Stores entry time of vehicle
unsigned long exitTime = 0;  // Stores exit time of vehicle
unsigned long vehicleCount = 0; // Number of vehicles counted
unsigned long lastMinuteCount = 0; // Vehicles counted in the past minute
unsigned long overstayStart = 0;  // Overstay detection time
bool vehiclePresent = false;      // Track if vehicle is in the booth
bool alertTriggered = false;      // Track if buzzer alert was triggered

void setup() {
    Serial.begin(9600);
    pinMode(IR_SENSOR_ENTRY, INPUT);
    pinMode(IR_SENSOR_EXIT, INPUT);
    pinMode(BUZZER_PIN, OUTPUT);
    pinMode(LED_PIN, OUTPUT);
}

void loop() {
    static unsigned long lastMinuteCheck = millis(); // Track time for 1-minute monitoring

    // Check vehicle entry
    if (digitalRead(IR_SENSOR_ENTRY) == LOW && entryTime == 0) {
        entryTime = millis();
        vehiclePresent = true;
        overstayStart = millis(); // Start overstay timer
        Serial.println("Vehicle Entered.");
    }

    // Check vehicle exit
```

```
    if (digitalRead(IR_SENSOR_EXIT) == LOW && entryTime != 0) {
        exitTime = millis();
        unsigned long duration = (exitTime - entryTime) / 1000; // Convert to seconds

        // Determine vehicle classification
        if (duration <= 10) {
            Serial.println("Fast Vehicle: Standard Toll Fee Applied.");
            digitalWrite(LED_PIN, HIGH); // Indicate fast vehicle
        } else {
            Serial.println("Slow Vehicle: Additional Surcharge Applied.");
            digitalWrite(LED_PIN, LOW); // Indicate slow vehicle
        }

        vehicleCount++;  // Increment total vehicle count
        entryTime = 0;   // Reset timer
        vehiclePresent = false;
    }

    // Overstay detection (More than 30 seconds)
    if (vehiclePresent && (millis() - overstayStart) / 1000 > 30 && !alertTriggered) {
        Serial.println("ALERT: Vehicle Overstayed! Triggering Buzzer.");
        digitalWrite(BUZZER_PIN, HIGH);
        alertTriggered = true;
        delay(5000); // Buzzer stays ON for 5 seconds
        digitalWrite(BUZZER_PIN, LOW);
    }

    // Monitor vehicle flow rate per minute
    if (millis() - lastMinuteCheck >= 60000) { // Every 60 seconds
        lastMinuteCount = vehicleCount;
        Serial.print("Vehicles passed in the last minute: ");
        Serial.println(lastMinuteCount);
        lastMinuteCheck = millis(); // Reset timer
    }

    delay(100); // Small delay to prevent rapid signal fluctuation
}
```

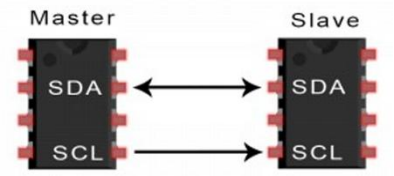
# MODULE 6

## INTER-INTEGRATED CIRCUIT (I2C)

**Introduction**

- I²C (Inter-Integrated Circuit) was invented in 1989 by Philips Semiconductor.
- It is a **serial communication protocol** with only **two-wire interface** (like UART).

- Commonly used for **short-distance, intra-board communication.**
- Combines the **best features of both SPI and UART** protocols.
- Supports **multi-master** and **multi-slave** configurations.

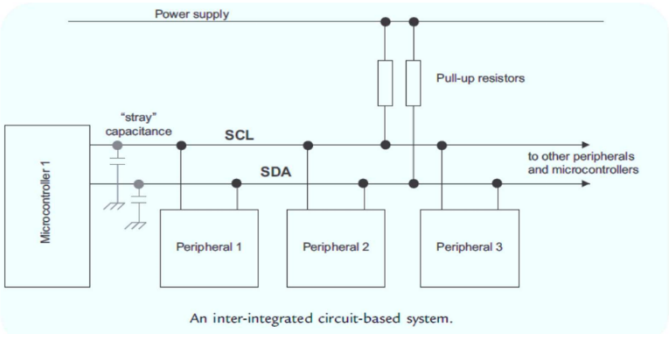


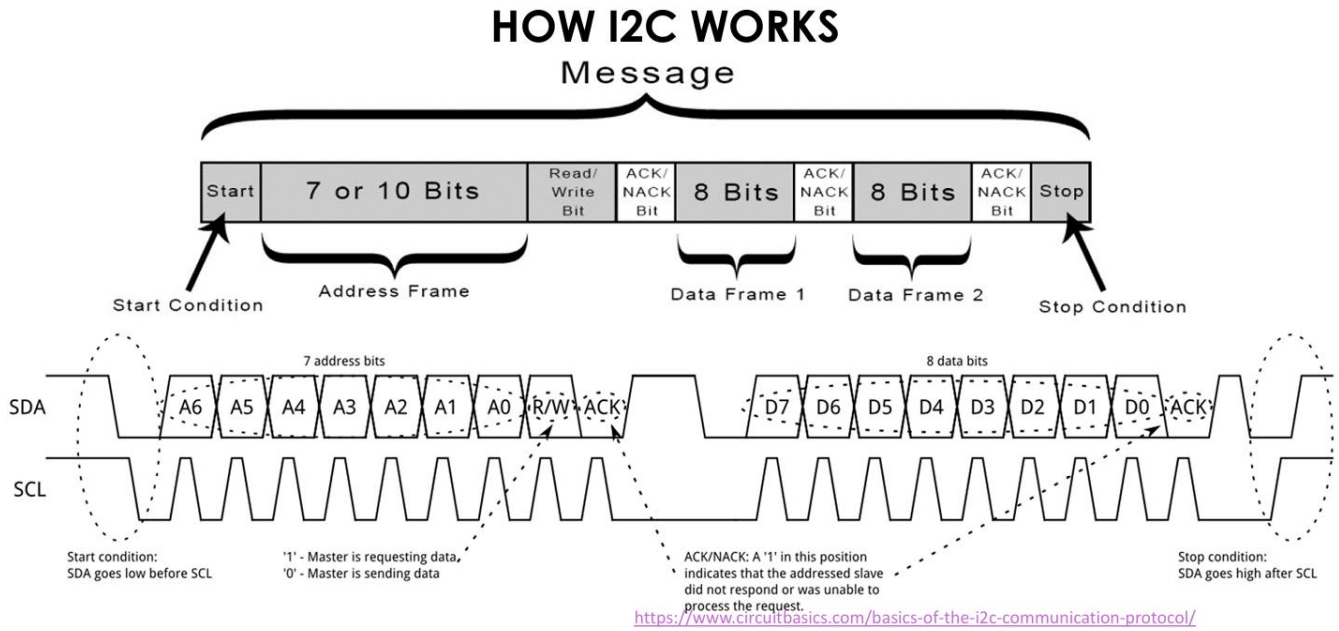

## Basic Features

- Uses only **two pins**: (transfers data bit by bit)
  - ***SDA (Serial Data):*** Bidirectional data line between master and slave.
  - ***SCL (Serial Clock):*** Clock signal generated by the master.
- It is a **synchronous protocol**, meaning data is synchronized with a clock.
- The **master controls the clock** and addresses the slave devices.

## How I²C Works

- Data is transferred in the form of **messages.**
- Each message contains:
  - A **start condition**
  - A **slave address frame**
  - **Read/Write bit**
  - One or more **data frames**
  - An **ACK/NACK bit** after each data frame
  - A **stop condition**




https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/

## Message Frame Structure

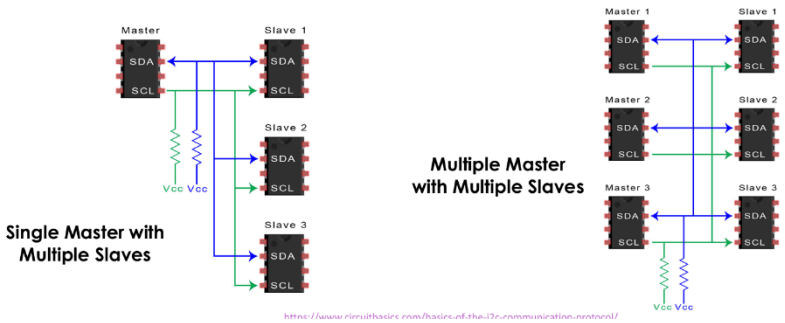
1. **Start Condition**:
   SDA goes from HIGH to LOW while SCL is HIGH.
2. **Address Frame**:
   7 or 10-bit unique address assigned to each slave.
3. **Read/Write Bit**:
   - 0 – Write (master sends data)
   - 1 – Read (master receives data)
4. **ACK/NACK Bit**:
   - After each frame, receiver sends ACK (0) or NACK (1).
5. **Stop Condition**:
   SDA goes from LOW to HIGH while SCL is HIGH.

## I²C Data Transmission Steps

1. **Start**:
   Master sets SDA LOW, then SCL LOW (start condition).
2. **Address + R/W Bit**:
   Master sends 7/10-bit slave address + Read/Write bit.
3. **Acknowledgment**:
   - Slave compares address.
   - If matched, it pulls SDA LOW (ACK), else leaves it HIGH (NACK).
4. **Data Transfer**:
   Master sends or receives data frame(s).
5. **ACK Response**:
   After each frame, receiver sends ACK if received successfully.
6. **Stop**:
   Master sets SCL HIGH, then SDA HIGH to end communication.

## I²C Configurations

- **Single Master - Multiple Slaves**
- **Multiple Masters - Multiple Slaves**




## Advantages

- Only two wires needed.
- Supports multiple masters and slaves.
- Uses chip addressing.
- ACK/NACK mechanism ensures reliability.
- Compatible with both fast and slow ICs.

**Disadvantages**

- Slower data rate compared to SPI.
- Clock speed limited due to pull-up resistors.
- Increased hardware complexity in multi-master/slave systems.
- Overhead due to address and ACK bits.

**I²C on Arduino**

| Signal | Arduino Pin |
|---|---|
| **SDA** | A4 |
| **SCL** | A5 |

**Arduino Wire Library Functions**

- **Wire.begin()** – Initialize I2C bus.
- **Wire.requestFrom(address, quantity)** – Master requests bytes from slave.
- **Wire.read() / Wire.receive()** – Read byte from slave.
- **Wire.write(data) / Wire.send(data)** – Send byte to master/slave.
- **Wire.onReceive(handler)** – Set function for receiving data (slave).
- **Wire.onRequest(handler)** – Set function for sending data (slave).
- **Wire.setClock(frequency)** – Change I2C clock speed.

***Arduino Example – Master Reads from Slave***

**Master Code (Read from Slave):**

```
#include <Wire.h>
void setup() {
  Wire.begin();            // Join I2C bus as master
  Serial.begin(9600);      // Start serial communication
}
void loop() {
  Wire.requestFrom(8, 6);  // Request 6 bytes from slave at address 8
  while (Wire.available()) {
    char c = Wire.read();  // Read a byte
    Serial.print(c);       // Print the character
  }
  delay(500);
}
```

**Slave Code (Respond to Master):**

```
#include <Wire.h>
```

```
void setup() {
  Wire.begin(8);                 // Join I2C bus with address 8
  Wire.onRequest(requestEvent); // Register function to run on request
}

void loop() {
  delay(100);
}

void requestEvent() {
  Wire.write("hello "); // Send 6 bytes to master
}
```

## CONTROL AREA NETWORK (CAN)

**Introduction**

- **CANbus** allows all ECUs (Electronic Control Units) to communicate with each other via a single bus.
- **CANbus** has two wires: **CAN Low** and **CAN High**.

***Need for CAN: ( Advantages)***

- **Simple and Low Cost**: Communicates via one single bus.
- **Fully Centralized**: One point of entry.
- **Extremely Robust**: Resistant to electromagnetic interference and electrical disturbances.
- **Efficient**: CAN frame has an ID for prioritization.

**CAN Protocol Overview**

- The **CAN protocol** is a set of rules for transmitting and receiving messages in a network of electronic devices.
- It is used in automobiles to communicate between electronic devices like:
    - Active suspension
    - ABS (Anti-lock Braking System)
    - Gear control
    - Lighting control
    - Air conditioning
    - Airbags
    - Central locking

**Key Components:**

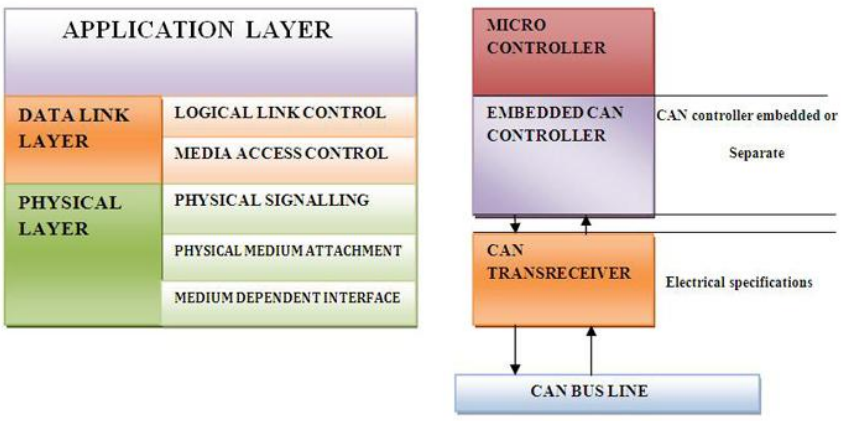
- Each electronic device is connected using a ***common serial bus*** to receive and transmit data using the CAN protocol.
- Nodes must have hardware and software embedded for data exchange.
- The **Host Controller** (ECU/MCU) is responsible for the functioning of the respective node.
- The **CAN controller** converts messages according to the CAN protocols for transmission via the CAN transceiver. It can either be connected separately or embedded inside the host controller.

## Advantages of CAN

- **Low Cost**: Economical in bulk production.
- **Reliable**: Highly immune to EMI (Electromagnetic Interference) with excellent error detection and handling.
- **Flexibility**: Easy to connect or disconnect nodes.
- **Good Speed**: Up to 1 MBit/s @ 40m bus length.
- **Multi-master Communication**: Allows multiple nodes to communicate.
- **Fault Confinement**: Isolates faults effectively.
- **Broadcast Capability**: Can send messages to multiple nodes simultaneously.

## CAN Architecture

- **CAN** uses the existing ***OSI reference model*** for data transmission.
- The protocol utilizes the lower two layers of the OSI model: the **Physical Layer** and the **Data Link Layer**.
- The ***system designer*** defines the functionality of the remaining five layers according to needs.
- The **Host Controller** uses the ***application layer*** of the OSI model.
- The **CAN Controller** incorporates Logical Link Control (LLC) and MAC (Medium Access Control) of the ***Data Link Layer***, allowing ***message filtering*** by unique ID.




***Message Processing:***

- Once framing is done, it is followed by ***arbitration, error detection,*** and ***acknowledgment*** handled by the ***MAC*** layer.
- The **CAN transceiver** takes care of ***encoding*** and ***decoding*** and ***synchronizes*** with the CAN bus for message transmission.

## How CAN Protocol Works

- The **master-slave configuration** is not supported in the CAN protocol architecture.
- Each node cannot access every other node for reading and writing data on the CAN bus.
- Based on bus availability, the CAN bus writes the CAN frame.
- A ***frame*** is a defined structure carrying a sequence of bits or bytes of data.
- In the CAN protocol, a predefined unique ID is used instead of a destination address.

**Frame Acceptance:**

- Every byte of information is defined in the CAN protocol.
- All nodes receive the CAN frame; based on the ID, nodes decide whether to accept it or not.
- The highest priority message gets bus access, while lower priority nodes wait until the bus is available.
- Nodes can be added without reprogramming.

## Message Framing

- Messages in CAN are sent in a format called **frames**. Framing is done by the MAC sublayer of the Data Link Layer.
- There are two types of frames: **Standard** and **Extended**. These can be differentiated based on ***identifier fields.***
    - **Standard CAN**: 11-bit identifier field.
    - **Extended CAN**: 29-bit identifier field.

**Binary Values:**

- Binary values in the CAN protocol are termed as **dominant** (logic 0) and **recessive** (logic 1) bits.

## Standard CAN - Message Framing

| SOF | 11 bit identifier | RTR | IDE | r0 | DLC | 0…8 byte data | CRC | ACK | EOF | IFS |
|---|---|---|---|---|---|---|---|---|---|---|

---

**Frame Structure:**

- **SOF**:It indicates  start of message and used to synchronize the nodes on a bus. A dominant bit (logic 0) in the field marks the start of frame.
- **IDENTIFIER**: Determines which node has access to the bus and identifies the type of message.
- **RTR**: Remote Transmission Request, indicates whether it's a data frame or a remote frame. 0-data frame, 1-remote frame
- **IDE**: Identifier Extension, specifies the frame format. 0-standard, 1-extended frame.
- **R0**: Reserved bit, not currently used.
- **DLC**: 4 bit Data Length Code, contains the number of bytes being transmitted.
- **DATA**: Stores up to 64 data bits of application data.
- **CRC**: Cyclic Redundancy Check, 16-bit (15 bits plus delimiter), contains checksum of the preceding application data for error detection.
- **ACK**: Acknowledge field for confirming data reception, ACK slot and the ACK delimiter. When the data is received correctly the recessive bit (1) in ACK slot is overwritten as dominant bit (0) by the receiver.
- **EOF**: 7 bit End of Frame, marks the end of a CAN frame and disables.
- **IFS**: Inter Frame Space, specifies the minimum number of bits separating consecutive messages. Provides the intermission between two frames and consists of three recessive bits (1 1 1) known as intermission bits. This time allows nodes for internal processing before the start of next frame.

**Extended CAN - Message Framing**

- Similar to the 11-bit identifier with added fields:
    - **SRR**: Substitute Remote Request, transmitted as a recessive bit (1) to make sure standard data frame always have the same priority.
    - **R1**: Another reserved bit, not currently used.

**Message Frame Types**

1. **Data Frames**: Most commonly used for data transmission. Contains Arbitration Fields, Control Fields, Data Fields, CRC Fields, a 2-bit Acknowledge Field and an End of Frame.
2. **Remote Frames**: Seeks permission for data transmission from another node. It is similar to data frame without data field and RTR bit is recessive (1) .
3. **Error Frames**: If transmitting or receiving node detects an error, it will immediately abort transmission and send error frame consisting of an error flag made up of dominant bits (0) and error flag delimiter made up of eight recessive bits (1).
4. **Overload Frames**: Provide extra delay between messages when a node is busy. An Overload frame is generated by a node when it becomes too busy and is not ready to receive.

**Future of CANbus**

- Need to accommodate increasingly advanced vehicles.
- Role of **Internet of Things** in vehicles.
- Development of **autonomous vehicles.**
- Advancement of **cloud computing.**
- Addressing **security issues.**
- Vehicle telematics.

**Bit Stuffing Method**

- After 5 bits of identical value, a supplementary bit with an opposite value is added to break the rhythm.
- This technique safeguards content during transport but slightly prolongs transmission time.

**Arbitration Field**

- A set of bits in the message frame assigned to each message for controlling arbitration.
- The identifier consists of the identifier bits and the RTR bit (0).
- The identifier length is 11 bits transmitted in the order from ID_10 toID_0, and the 7 most significant bits (ID_10 to ID_4) must not all be recessive.

**Remote Frame** – RTR bit (1)

**Types of Errors in CAN**

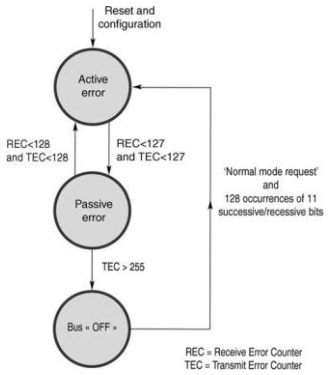
- ***Physical Layer Errors:***
    - **Bit Error**: Affected bit itself.
    - **Bit Stuffing Error**: Involuntary (parasites, transmissions, forgotten elements) or deliberate errors.
- ***Frame Layer Errors:***
    - CRC delimiter error.
    - Acknowledgment delimiter error.
    - End of frame error.
    - Error delimiter error.
    - Overload delimiter error.

❖ Errors are signaled by an error frame generated on the bus.

- ***Confinement Errors***

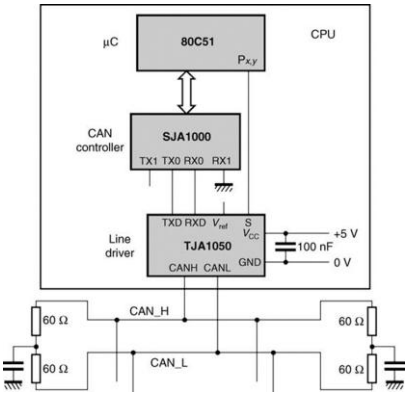
  Determines whether a node is:
    - Not disturbed.

- Slightly disturbed.
- Seriously disturbed.
- Too disturbed, requiring a switch to bus off.




**Bus Access**

- Details on how nodes access the bus for communication.




## ZIGBEE

**Introduction**

- **ZigBee protocol** is primarily used for ***power-constrained, short-distance*** communication.
- **Why another short-range communication protocol?**
  - **Bluetooth**
  - **Wi-Fi**
- ZigBee is specially built for ***control and sensor*** applications.
- It is a standard characterized by:
  - Very low cost
  - Low power consumption
  - Low data rate
  - Short-range communications

## Key Features of ZigBee

- Most commonly used standard in **Wireless Sensor Networks (WSN)** and **Internet of Things (IoT).**
- Open-source standard developed by the **ZigBee Alliance.**
- Operates on the **IEEE 802.15.4** standard for wireless personal area networks.
- Frequencies:
    - 868 MHz (Europe)
    - 902-928 MHz (US & Australia)
    - 2.4 GHz (worldwide)
- Data rate of **250 kbps** for low data rate applications.

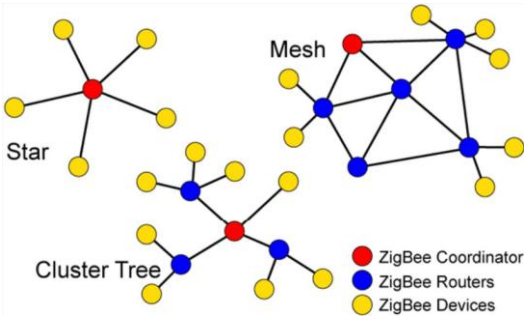
## ZigBee Device Characteristics

- ZigBee devices can cover a distance range of **10-100 meters.**
- Supports both **master-slave** and **master-master** configurations.
- ZigBee architecture consists of:
    - **Coordinator**
    - **Router**
    - **End Device**
- Supports **star**, **tree**, and **mesh** topologies.
- Modes of operation: **Beacon** and **Beaconless.**

## Network Components

### *Coordinator*

- The root of the network.
- Only one coordinator per network.
- Performs:
    - ✓ Channel selection
    - ✓ Assigning node IDs
    - ✓ Allocation of unique addresses




### *Routers*

- Intermediate nodes between the coordinator and end nodes.
- Route traffic between different nodes.
- Allow other end devices to join the network.

### *End Node*

- Operates in sleep mode to increase battery efficiency.

- All traffic to the end device is first routed to the parent node.

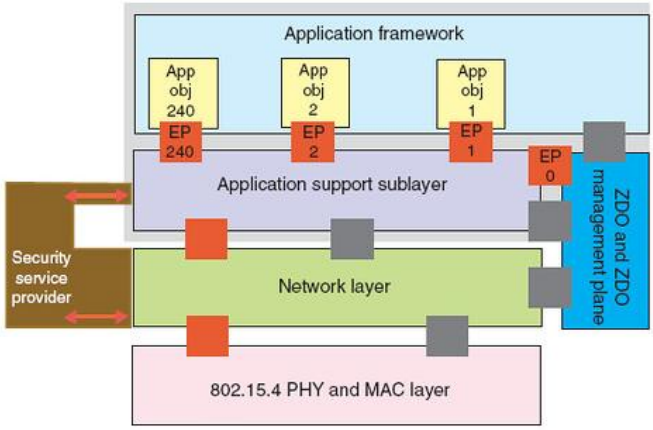
## Channel Access

- The coordinator assigns only one channel to the network.
- Methods of channel access:
    - ***Contention-based:*** Not synchronized, uses **CSMA** (Carrier Sense Multiple Access).
        - CSMA: The channel goes to receive mode and checks for any signal. If no signal is detected, it starts data transmission; otherwise, it waits for a random period.
    - ***Contention-free:*** Allocates specific time slots to each device.
        - **GTS** (Guaranteed Time Slot).

## ZigBee Characteristics

- **Low power consumption**: Extended battery life.
- **Low data rate**: 200-250 kbps (compared to Wi-Fi: 11 Mbps, Bluetooth: 1 Mbps).
- **Short range**:
    - 75-100 meters indoors
    - Up to 300 meters outdoors
- Network join time: **30 seconds**.
- Supports large and small networks (up to **65,000** devices theoretically).
- **Low cost**.
- Uses **AES encryption** for security.

## ZigBee Protocol Stack




## IEEE 802.15.4 PHY Packet Structure

| Preamble (32 ) (Synchronization) | Start of packet delimiter (8) | PHY header (8) (PSDU Length) | PHY Service Data Unit (PSDU) (0-1016 bits) (Data Field) |
|---|---|---|---|

- **Preamble** (32 bits): Synchronization.
- **Start of Packet Delimiter** (8 bits).
- **PHY Header** (8 bits): PSDU Length.
- **PSDU**: Data field (0-1016 bits).

## IEEE 802.15.4 MAC Frame Format

### Frame Types

- **Data Frame**
- **Beacon Frame**
- **Acknowledgment Frame**
- **Command Frame**

### General MAC Frame Format

| Octets : 2 | 1 | 0/2 | 0/2/8 | 0/2 | 0/2/8 | Variable | 2 |
|---|---|---|---|---|---|---|---|
| Frame Control | Sequence number | Destination PAN identifier | Destination Address | Source PAN Identifier | Source Address | Frame payload | Frame Check Sequence |
| MAC header | | | | | | MAC Payload | MAC footer |

### IEEE 802.15.4 MAC Frame Format - Beacon Frame

| Octets : 2 | 1 | 4 or10 | 2 | Variable | variable | Variable | 2 |
|---|---|---|---|---|---|---|---|
| Frame control | Beacon sequence number | Source address information | Super frame specification | GTS field | Pending address field | Beacon payload | Frame check sequence |
| MAC header | | | MAC payload | | | | MAC Footer |

- The beacon frame is transmitted periodically by the PAN coordinator, providing network management information through the super frame and GTS fields. It synchronizes network devices and indicates the communication period.

---

### IEEE 802.15.4 MAC Frame Format - Command Frame

| Octets: 2 | 1 | 4 to 20 | 1 | Variable | 2 |
|---|---|---|---|---|---|
| Frame control | Data Sequence number | Address information | Command type | Command payload | Frame Check sequence |
| | MAC header | | MAC payload | | MAC Footer |

- The command identifier specifies actions like association, disassociation, and data, GTS, or beacon requests.
- Useful for communication between the network devices.

### IEEE 802.15.4 MAC Frame Format - Data Frame

| Octets : 2 | 1 | 4 to 20 | Variable | 2 |
|---|---|---|---|---|
| Frame Control | Data Sequence number | Address information | Data payload | Frame check sequence |
| | MAC header | | MAC payload | MAC footer |

**Acknowledgment Frame Format**

| Octets : 2 | 1 | 2 |
|---|---|---|
| Frame control | Data Sequence Number | Frame Check sequence |
| | MAC header | MAC footer |

## ZigBee Parameters

| Parameter | Description |
|---|---|
| **Radio Technology** | IEEE 802.15.4 |
| **Frequency Band/Channels** | 2.4 GHz (ISM band), 16 channels (2 MHz wide) |
| **Data Rate** | 250 Kbits/sec |
| **Encryption Support** | AES-128 at Network Layer |
| **Communication Range** | 75-100 meters indoor, 300+ meters line of sight |
| **Network Size (Theoretical)** | Up to 65,000 |

## Advantages of ZigBee Protocol

- Simple setup and maintenance.
- Low power consumption, ideal for long-lifetime devices.
- Suitable for applications in lighting, security, appliances, and home access.
- Low latency and low data rate.
- Offers network flexibility and scalability.

## ZigBee Connection with Arduino

**Arduino Coding for ZigBee Interface**

```
#include <SoftwareSerial.h>
SoftwareSerial XBee(2, 3);

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

void loop() {
    if (Serial.available()) {
        XBee.write(Serial.read());
    }
    if (XBee.available()) {
        Serial.write(XBee.read());
```

```
    }
}
```

This code establishes a serial communication between the Arduino and the ZigBee module using the SoftwareSerial library.

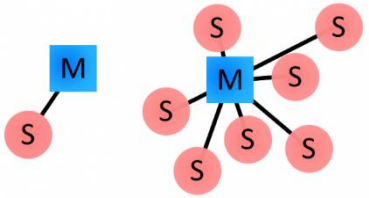
## BLUETOOTH

### Introduction

- **Bluetooth** is a transceiver protocol that operates on a **2.4 GHz** wireless link.
- It is a secure protocol, ideal for short-range, low-power, low-cost wireless transmissions.
- Both **ZigBee** and **Wi-Fi** use the same bandwidth.
- Bluetooth is a **dynamic** standard where devices can **automatically** find each other, establish connections, and discover capabilities on an ad hoc basis.
- Bluetooth networks can be referred to as **piconets** and **scatternets**.
- It uses a **master/slave model**.

### Bluetooth Master/Slave Model

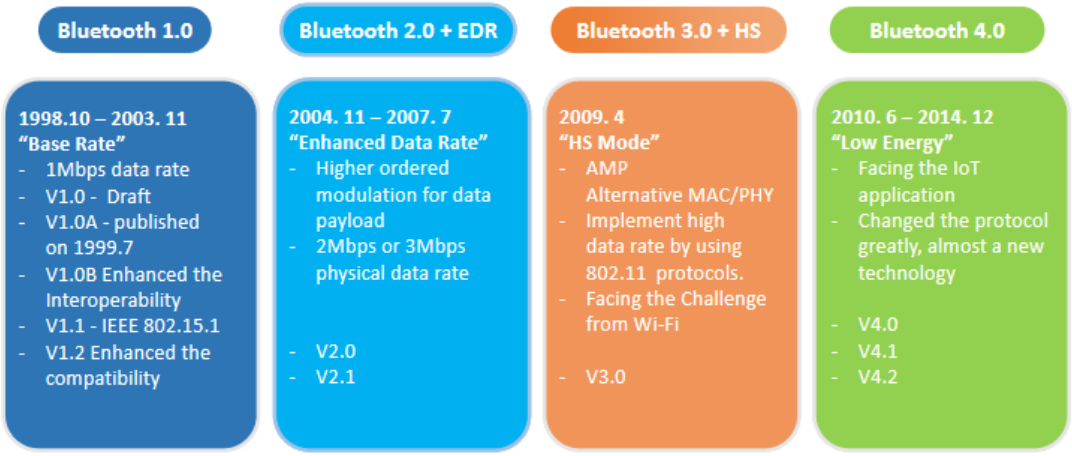
- A single master device can connect to up to **seven** different slave devices.
- The master coordinates communication throughout the piconet, sending data to any of its slaves and requesting data from them.




- Slaves can only transmit to and receive from their master; they cannot communicate with other slaves in the piconet.
- Every Bluetooth device has a unique **48-bit address**, commonly abbreviated as **BD_ADDR**, usually presented in the form of a **12-digit hexadecimal value**.

---

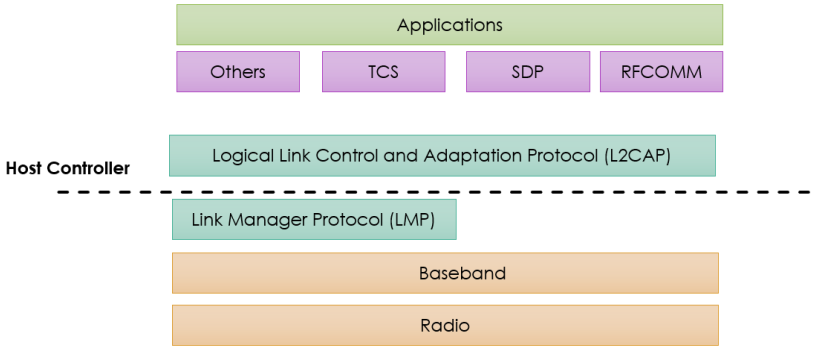
**Bluetooth Versions**



| Version | Key Feature | Max Speed | Target Use Case |
|---|---|---|---|
| **1.0–1.2** | Basic connectivity | 1 Mbps | Early Bluetooth devices |
| **2.0 + EDR** | Enhanced data rate | 2–3 Mbps | Audio, file transfer |
| **3.0 + HS** | High-speed via Wi-Fi | ~24 Mbps | Large file transfer |
| **4.0** | Low Energy for IoT | ~1 Mbps BLE | Fitness, health, sensors |

## BLUETOOTH ARCHITECTURE

### *Components*




**Radio Layer**

- Defines the **requirements** for a transceiver to operate in the **2.4 GHz ISM band**, including **frequency bands, frequency hopping specifications, and modulation techniques.**

**Baseband Layer**

- Describes the **specification** of the Bluetooth **Link Controller (LC)** and carries the **baseband protocol.**
- Defines the **addressing scheme, packet frame format, timing, and power control algorithms.**
- Two types of links can be created in baseband:
  - **ACL (Asynchronous Connection Less)**
  - **SCO (Synchronous Connection Oriented)**

### *Link Types*

**Asynchronous Connection Less (ACL)**

- Packet-switched data.
- Slave can have only one ACL link to the master.
- Used for correct delivery over fast delivery.
- Maximum data rate of **721 kbps.**

**Synchronous Connection Oriented (SCO)**

- Real-time data transmission.
- Damaged packets cannot be retransmitted.
- Data rate of **64 kbps**.

**Link Manager Protocol (LMP)**

- Used by link managers for link setup and control.
- Main functions include:
    - Device authentication
    - Message encryption
    - Negotiation of packet sizes

🔷 **Host Controller Interface (HCI)**

- Think of HCI like a **remote control**.
- It **sends commands** to the internal parts of Bluetooth (like the baseband and link manager).
- It also **checks the status** of the hardware.

👉 Example: If your computer wants to send a file to your phone, it sends a command through HCI asking the Bluetooth hardware to start a connection.

---

🔷 **L2CAP (Logical Link Control and Adaptation Protocol)**

- L2CAP is like a **traffic controller**.
- It breaks **big chunks of data** into small Bluetooth packets and puts them back together later.
- It lets **multiple apps** use Bluetooth at once.
- It can manage **QoS** (Quality of Service) – for example, ensuring your audio doesn't lag during a call.

---

🔷 **RFCOMM (Radio Frequency Communication)**

- RFCOMM pretends to be a **serial port** (like old COM ports).
- Used for Bluetooth devices that send data like they were using a **wired cable**.

👉 Example: When your Bluetooth printer or a GPS device talks to your laptop.

---

🔷 **SDP (Service Discovery Protocol)**

- SDP is how devices ask each other:
  *"What can you do?"*
- It helps devices **find services** (like a speaker, keyboard, or health monitor).

👉 Example: When your phone connects to a smartwatch, it uses SDP to find out if the watch supports heart rate monitoring.

---

🔷 **TCS (Telephony Control Protocol)**

- TCS handles **Bluetooth calling** features.
- It controls call setup and audio routing (usually in car systems or Bluetooth headsets).

---

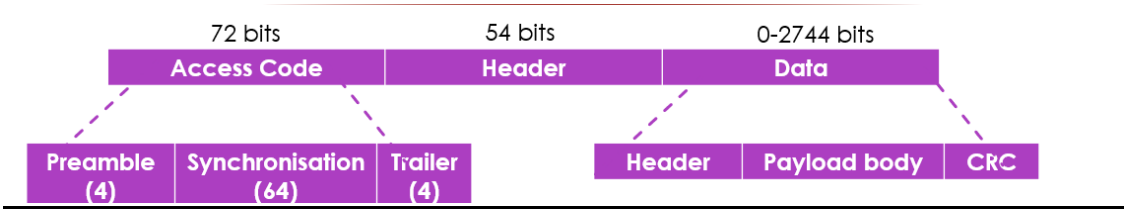
🔷 **Applications Layer**

- This is the **user-facing** part – the apps you use.
- Each app follows a **profile** (like a blueprint).

👉 Example: The "A2DP profile" is used for streaming audio from your phone to a speaker.

**BLUETOOTH PACKET STRUCTURE**

**Packet Components**




**Access Code**

- Used for packet identification. The receiver compares the incoming signal with the Access Code (AC); if a mismatch is found, the packet is ignored.
- The **72-bit AC** is derived from the master identity and is used for synchronization.
- Three types of Access Codes:
  - **Channel AC**: Targets piconet ID.
  - **Device AC**: Individual devices.
  - **Inquiry AC**: Used during pairing.

**Header Structure**


- **Address**: Can address up to **7 slave devices**; if it is 0, messages will be broadcasted.
- **Type**: Defines the type of incoming data
  - ACL – Data medium (DM) or  Data High (DH)

    - SCO - Data Voice (DV) and High Quality Voice (HV)
- 12 types of packet for each SCO and HV
- 4 common control packets
- **Flow Control**: 1-bit flow control.
- **ARQN**: 1-bit acknowledgment.
- **SEQN**: 1-bit sequential numbering scheme for packet ordering.
- **HEC**: Header error check.
- Header = 3 X 18 bits = 54 bits

**BONDING AND PAIRING**

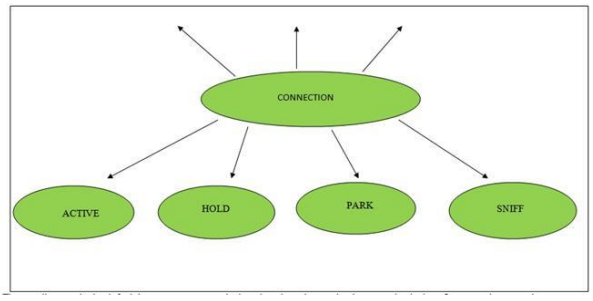
- Pairing requires an authentication process to establish a connection between devices.
- Pairing can be a simple "Just Works" operation (common for devices with no UI, like headsets).
- Older pairing processes (v2.0 and earlier) involve entering a common PIN code on each device, which can range from four numbers (e.g., "0000" or "1234") to a 16-character alphanumeric string.

**Connection Process**

Creating a Bluetooth connection involves three progressive states:

1. **Inquiry**: If two Bluetooth devices know absolutely nothing about each other, one must run an inquiry to try to discover the other. On receiving the inquiry the device which listens to such a request will send back address, name and other information
2. **Paging (Connecting)**: The process of forming a connection between two Bluetooth devices.
3. **Connection**: After paging, the device enters the connection state, where the mode of operation is decided.

**Connection Modes**




- **Active Mode**: Regular connected mode where devices send and receive data (up to 7 devices).
- **Sniff Mode**: Power-saving mode that frees the slave for a predetermined period.
- **Hold Mode**: Temporary mode where the device sleeps for a specific period.
- **Park Mode**: Sleep mode with infinite time; the slave wakes only if the master tells it to.

**POWER CLASSES**

The transmit power determines the range of a Bluetooth module, defined by its power class:

| Class Number | Max Output Power | Max Range |
|---|---|---|
| **Class 1** | 20 dBm (100 mW) | 100 m |
| **Class 2** | 4 dBm (2.5 mW) | 10 m |
| **Class 3** | 0 dBm (1 mW) | 10 cm |

## HC-05 Bluetooth Module

- The **HC-05 module** is an easy-to-use Bluetooth SPP (Serial Port Protocol) module designed for transparent wireless serial connection setup.
- It can be configured as either a Master or Slave, making it suitable for wireless communication.
- Fully qualified Bluetooth V2.0+EDR (Enhanced Data Rate) with **3 Mbps** modulation.
- The Bluetooth module HC-05 is a MASTER/SLAVE module. By default the factory setting is SLAVE.
- The Role of the module (Master or Slave) can be configured only by AT COMMANDS.
- Master module can initiate a connection to other devices. The user can use it simply for a serial port replacement to establish connection between MCU and GPS, PC to your embedded project, etc.

### Features

- **Hardware Features**:
  - Up to +4 dBm RF transmit power.
  - 3.3 to 5 V I/O.
  - Programmable Input/Output (PIO) control.
  - UART interface with programmable baud rate.
  - Integrated antenna.
- **Software Features**:
  - Slave default Baud rate: **9600**, Data bits: **8**, Stop bit: **1**, No parity.
  - Auto-connect to the last device on power by default.
  - Permit pairing device to connect by default.
  - Auto-pairing PIN code: **"1234"** by default.

---

### Arduino Control Program for HC-05

This program controls an LED connected to Pin 7 of the Arduino through commands sent by a Bluetooth-supported mobile device.

```
#define ledPin 7
int state = 0;
```

```
void setup() {
    pinMode(ledPin, OUTPUT);
    digitalWrite(ledPin, LOW);
    Serial.begin(38400);
}

void loop() {
    if (Serial.available() > 0) { // Checks whether data is coming from the serial port
        state = Serial.read(); // Reads the data from the serial port
    }
    if (state == '0') {
        digitalWrite(ledPin, LOW); // Turn LED OFF
        Serial.println("LED: OFF"); // Send back to the phone the String "LED: OFF"
        state = 0;
    }
    else if (state == '1') {
        digitalWrite(ledPin, HIGH); // Turn LED ON
        Serial.println("LED: ON"); // Send back to the phone the String "LED: ON"
        state = 0;
    }
}
```

This code allows you to control an LED via Bluetooth commands sent from a mobile device.

## WI-FI

**Introduction**

- **Wi-Fi** is an alternative network to wired connections, commonly used for connecting devices in wireless mode.
- **Wi-Fi** stands for **Wireless Fidelity** and refers to the **IEEE 802.11** standard for Wireless Local Area Networks (WLANs).
- Wi-Fi connects computers to each other, to the internet, and to wired networks.

### WI-FI TECHNOLOGY

**Elements of a Wi-Fi Network**

- Wi-Fi uses **radio technology** to transmit and receive data at high speeds.
- **Access Point (AP)**: A wireless LAN transceiver or "base station" that can connect multiple wireless devices simultaneously to the Internet.
- **Wi-Fi Cards**: Devices that accept the wireless signal and relay information. They can be internal or external.
- **Safeguards**: Firewalls and antivirus software protect networks from unauthorized access and keep information secure.
- **SSID (Service Set Identifier)**: A 32-character name that identifies the Wi-Fi network, differentiating it from others. All devices attempting to connect use this SSID.

- **WPA-PSK (Wi-Fi Protected Access - Pre-Shared Key)**: A program developed by the Wi-Fi Alliance to secure wireless networks using PSK authentication. It includes WPA, WPA2, and WPA3, which encrypt Wi-Fi signals to protect against unauthorized users.
- Wi-Fi uses **Ad-Hoc networks** to transmit, which are point-to-point networks without a central interface.

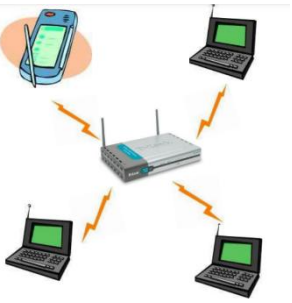
## WI-FI TOPOLOGIES

- **Peer-to-Peer Topology (Ad-hoc Mode)**
- **AP-Based Topology (Infrastructure Mode)**

### Peer-to-Peer Topology

- An Access Point (AP) is not required.
- Client devices within a cell can communicate with each other directly.
- Useful for quickly and easily setting up a wireless network.

### Infrastructure Network



- Clients communicate through an Access Point.
- All communication must go through the AP.
- If a Mobile Station (MS), such as a computer, PDA, or phone, wants to communicate with another MS, it sends the information to the AP first, which then forwards it to the destination MS.

## HOTSPOTS

- A **Hotspot** is a geographical area with a readily accessible wireless network.
- Hotspots are equipped with broadband Internet connections and one or more Access Points that allow users to access the Internet wirelessly.
- They can be set up in any public location that supports an Internet connection.

### How a Wi-Fi Network Works

- A Wi-Fi hotspot is created by installing an Access Point connected to the Internet.
- The Access Point acts as a base station.
- When a Wi-Fi-enabled device encounters a hotspot, it can connect to that network wirelessly.
- A single Access Point can support up to **30 users** and function within a range of **100-150 feet indoors** and **up to 300 feet outdoors.**
- Multiple Access Points can be connected via Ethernet cables to create a single large network.

**ADVANTAGES OF WI-FI**

- **Mobility**
- **Ease of Installation**
- **Flexibility**
- **Cost-Effectiveness**
- **Reliability**
- **Security**
- Uses unlicensed parts of the radio spectrum.
- **Roaming** capabilities.
- **Speed** of data transmission.

**LIMITATIONS OF WI-FI**

- **Interference** from other devices.
- **Degradation in performance** due to distance or obstacles.
- **High power consumption** in some scenarios.
- **Limited range** compared to wired networks.

**INTERFACE WITH ARDUINO**

**ESP8266 Module**

The **ESP8266** is a popular Wi-Fi module used for interfacing with Arduino.

**Sample Code for ESP8266 with Arduino**

```
#include <SoftwareSerial.h>
SoftwareSerial ESPserial(2, 3); // RX | TX

void setup() {
    Serial.begin(115200); // Communication with the host computer
    // while (!Serial) { ; }
    // Start the software serial for communication with the ESP8266
    ESPserial.begin(115200);
    Serial.println("");
    Serial.println("Remember to set Both NL & CR in the serial monitor.");
    Serial.println("Ready");
    Serial.println("");
}

void loop() {
    // Listen for communication from the ESP8266 and write it to the serial monitor
    if (ESPserial.available()) {
        Serial.write(ESPserial.read());
    }
    // Listen for user input and send it to the ESP8266
```

```
    if (Serial.available()) {
        ESPserial.write(Serial.read());
    }
}
```

This code allows communication between the ESP8266 Wi-Fi module and the Arduino, enabling wireless data transmission.

## SPI

### Introduction

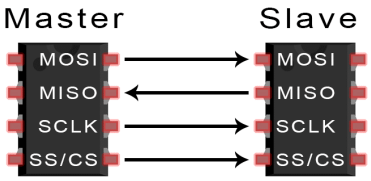
**Definition**: Serial Peripheral Interface (SPI) is a ***full-duplex serial*** communication protocol.

- **Operation**:
  - Works in ***synchronous*** mode at very ***high speeds***.
  - More suitable for ***short-distance*** communication with specific devices or onboard devices.
- **Availability**: Commonly found in most microcontrollers like PIC, AVR, ARM, and others.

### COMMUNICATION MODEL

- **Master-Slave Model**:
  - SPI supports a master-slave communication model.
  - Facilitates a ***low-cost*** and ***flexible interface*** between the processor and peripherals.
  - Example: A microcontroller acts as the master while devices like GSM, GPS, and sensors act as slaves.
- **Limitations**:
  - Does not support multi-master communication.

### SPI BUS SIGNALS




The SPI bus consists of four signals or pins:

1. **Master-Out / Slave-In (MOSI)**:
   - Master generates data, and the slave receives it.
2. **Master-In / Slave-Out (MISO)**:
   - Slave generates data and transmits it to the master.
3. **Serial Clock (SCLK)**:
   - The master generates the clock signal, supplying it to the clock input of the slave.
4. **Chip Select (CS) or Slave Select (SS)**:

- The master selects a particular slave using the chip select line.

**SPI Parameters**

| Parameter | Specification |
|---|---|
| **Wires used** | 4 |
| **Maximum speed** | 10 Mbps |
| **Synchronous** | Yes |
| **Serial** | Yes |
| **Max number of masters** | 1 |
| **Max number of slaves** | Theoretically unlimited |

**HOW SPI WORKS**

**Clock**

- The clock ***synchronizes the data bits*** from the master to the slave.
- Only ***one bit*** is transmitted with each clock signal.
- The ***frequency*** of the clock signal determines the ***data transfer rate***.
- The master initiates communication in SPI and generates the clock signal.

**Slave Select (SS)**

- The slave select pin is ideally kept at a ***high voltage level***.
- The master decides which slave to communicate with by setting the specific ***slave's SS to a low voltage level***.
- The master can have multiple SS/CS pins to enable ***multiple slaves in parallel***.
- If only a single SS pin is available, multiple slaves can be connected to the master in a ***daisy chain configuration***.

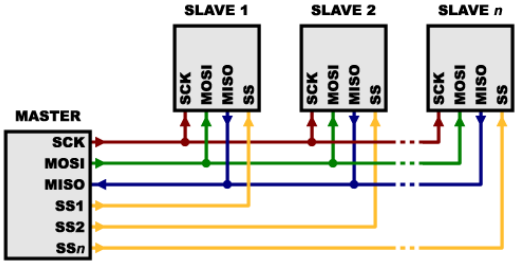
**MOSI and MISO**

- The master sends data serially through the MOSI line, and the slave receives it at the MOSI pin.
- The slave communicates back to the master using the MISO pin.
- In master-to-slave communication, the Most Significant Bit (MSB) is sent first.
- In slave-to-master communication, the Least Significant Bit (LSB) is sent first.

**SPI CONFIGURATIONS**

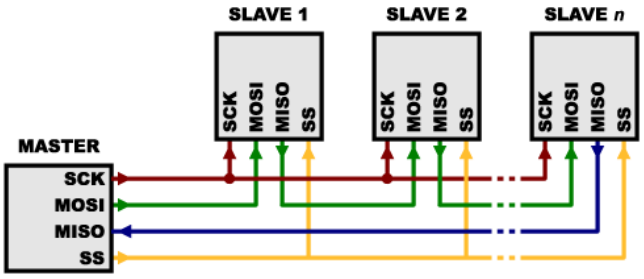
**Independent Slave Configuration**

- In this configuration, all slaves have dedicated select lines and are connected individually to the master.



- The master SCK is connected to all slave clock lines.
- MOSI and MISO pins of both master and slaves are interconnected.

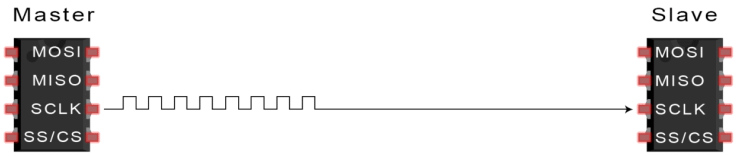
**Daisy Chain Configuration**



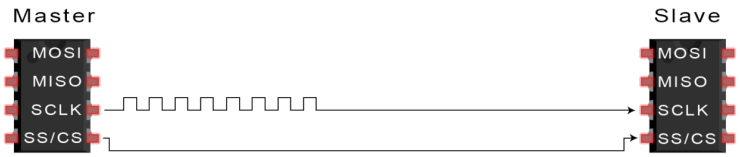

- In this configuration, only a single Slave Select line is connected to all slaves.
- The MOSI of the master is connected to the MOSI of slave 1, and the MISO of slave 1 is connected to the MOSI of slave 2.
- The MISO of the final slave is connected to the MISO of the master.

**STEPS OF SPI COMMUNICATION**

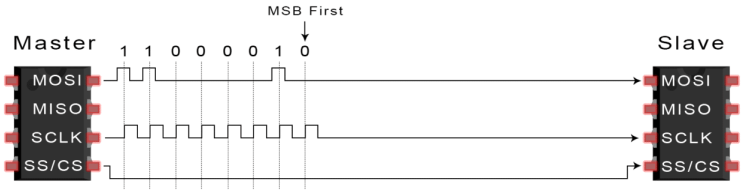
1. **Step 1**: The master outputs the clock signal.



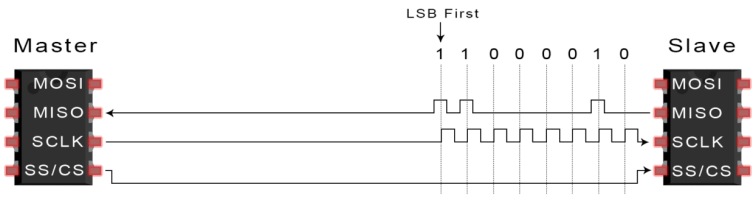

2. **Step 2**: The master configures the SS/CS pin to a low voltage state to activate the slave.




3. **Step 3**: The master sends data serially one bit at a time to the slave via the MOSI pin, and the slave receives the data in sequence.




4. **Step 4**: The slave responds to the master by sending one bit at a time, and the master reads the received message.




**MODES OF OPERATION**

- The clock logic regulates the sending of data and can be configured in ***four*** possible modes.

| Mode | Clock Polarity | Clock Phase |
|---|---|---|
| SPI_Mode0 | 0 | 0 |
| SPI_Mode1 | 0 | 1 |
| SPI_Mode2 | 1 | 0 |
| SPI_Mode3 | 1 | 1 |

- **Clock Polarity**: Refers to the ***logic level*** of the clock when no data is being transferred.
- **Clock Phase**: Indicates whether the data will be read by the devices on the rising or falling edges of the clock pulses.

**APPLICATIONS**

Applications of the SPI protocol include:

- **Memory**: SD Cards, MMC, EEPROM, Flash
- **Sensors**: Various types of sensors
- **Control Devices**: ADC, Trim POTS, Audio Codec
- **Others**: Camera Lens Mount, Touchscreen, LCD, etc.

**ADVANTAGES**

- Full duplex communication.
- High-speed data transmission.
- Not limited to 8 bits during transfer.
- Slave uses a master clock instead of oscillators.
- Master device handles all slaves, minimizing conflict.
- Continuous data streaming without start and stop bits.
- Simple slave addressing system.

**DISADVANTAGES**

- Uses four pins.
- Only a single master is allowed in SPI.
- A dedicated pin is required for the slave on the master (CS/SS).
- No acknowledgment mechanism is provided.
- Device-based speed of data transmission.

**ARDUINO FUNCTIONS FOR SPI PROTOCOL**

- **SPI.begin():** Initializes the SPI bus.
- **SPI.end():** Disables the SPI bus.
- **SPI.beginTransaction():** Initializes the SPI bus using SPISettings.
    - Example: SPI.beginTransaction(SPISettings(14000000, MSBFIRST, SPI_MODE0))
- **SPI.endTransaction():** Stops using the SPI bus.
- **SPI.setClockDivider(divider)**: Sets the SPI clock divider (options: 2, 4, 8, 16, 32, 64, or 128).
- **SPI.setDataMode(mode)**: Sets the data mode (options: Mode_0, Mode_1, Mode_2, Mode_3).
- **SPI.transfer(val):** Transfers a byte of data (send and receive) over the bus.
- **SPI.transfer(buffer, size):** Transfers an array of data (send and receive).
- **SPI.usingInterrupt(interruptNumber):** Registers an interrupt number.

**Reference URL**: [Arduino SPI Reference]

**Example: Arduino - Serial Peripheral Interface (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); // Send the string "Hello, world!\r" and use \r as a flag to
signal the slave that transfer is done.
        Serial.print(c);
    }
    digitalWrite(SS, HIGH); // Disable Slave Select
    delay(2000);
}
```

**Example: Arduino - Serial Peripheral Interface (Slave)**

```
#include <SPI.h>

char buff[50];  // Stores the incoming values via SPI.
volatile byte indx; // Stores the index of the 8-bit data.
volatile boolean process;  // Saves the current status of the transmission.

void setup(void) {
    Serial.begin(115200);
```

```
    pinMode(MISO, OUTPUT); // Set MISO as output to send data to master
    SPCR |= _BV(SPE); // Enable SPI in slave mode (SPCR - SPI control register)
    indx = 0; // Initialize buffer index
    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 buffer
        if (c == '\r') // Check for the end of the word
            process = true;
    }
}

void loop(void) {
    if (process) {
        process = false; // Reset the process flag
        Serial.println(buff); // Print the array on the serial monitor
        indx = 0; // Reset index to zero
    }
}
```

# MODULE 7

**Applications of Embedded Systems**

**A Deep Dive into Real-World Use Cases**

**Introduction to Embedded Systems**

- **Definition of Embedded Systems**: A combination of hardware and software that forms a component of a larger machine. An example is a microprocessor that controls an automobile engine.
- **Key Features**:
    - **Real-time**
    - **Reliability**
    - **Efficiency**
    - **Custom Design**
- **Examples**: Microcontrollers, SoCs (System on Chips), FPGAs (Field Programmable Gate Arrays).

**Categories of Embedded Systems**

- **Based on Performance**:
    - Real-time
    - Standalone
    - Networked

        - Mobile
    - **Based on Complexity**:
        - Small-scale
        - Medium-scale
        - Sophisticated

---

**Embedded Systems in Consumer Electronics**

- **Devices**: Smartphones, Smart TVs, Digital Cameras
- **Home Automation**: Smart bulbs, Thermostats, Voice Assistants

**Embedded Systems in the Automotive Industry**

- **Components**:
    - Engine Control Units (ECUs)
    - Anti-lock Braking Systems (ABS)
    - Airbag Control
    - Infotainment Systems
    - Advanced Driver Assistance Systems (ADAS)

**Embedded Systems in Healthcare**

- **Devices**:
    - Wearable Health Monitors (Heart Rate, SpO2 Sensors)
    - Portable Diagnostic Devices
    - Imaging Systems (MRI, CT Scanners)
    - Smart Prosthetics

**Embedded Systems in Industrial Automation**

- **Applications**:
    - PLCs (Programmable Logic Controllers)
    - Robotics and CNC Machines
    - SCADA Systems
    - Predictive Maintenance using IoT

**Embedded Systems in Telecommunications**

- **Devices**:
    - Routers, Switches, and Base Stations
    - Signal Encoding/Decoding
    - Satellite Communication Systems

**Embedded Systems in Aerospace and Defense**

- **Applications**:

- Avionics Systems
- Drones and UAVs
- Radar and Sonar Systems
- Navigation and Missile Guidance

**Embedded Systems in Agriculture**

- **Applications**:
  - Precision Farming using Sensors
  - Automated Irrigation Systems
  - Soil Health Monitoring
  - Livestock Tracking with RFID/LoRa

**Embedded Systems in Smart Cities**

- **Applications**:
  - Smart Traffic Control
  - Waste Management
  - Public Surveillance
  - Smart Grids & Energy Meters

**Embedded Systems in Education & Research**

- **Applications**:
  - Lab Instruments (Oscilloscopes, Spectrometers)
  - Educational Robotics
  - Real-Time Simulation Kits

---

**Future Trends in Embedded Systems**

- **Emerging Technologies**:
  - Edge AI & Machine Learning on Microcontrollers
  - 5G-Enabled Embedded Devices
  - Cybersecurity in Embedded Systems
  - Quantum Embedded Devices

**Summary**

- **Recap**: Key applications of embedded systems.
- **Ubiquity**: Significance in modern systems and interdisciplinary integration.

---

**A General Overview of Embedded System Applications**

**Application Areas**

- **Automobiles**

- **Telecommunication**
- **Medical Systems**
- **Consumer Electronics**
- **Banking**
- **Home Appliances**
- **Offices**
- **Security**
- **Academia**
- **Agriculture**

**Specific Applications**

- **Automobiles**:
    - Anti-lock Brakes
    - Automatic Transmission
- **Telecommunication**:
    - Routers and Switches
- **Medical Systems**:
    - Pacemakers, Patient Monitoring Systems
- **Consumer Electronics**:
    - MP3 Players, Mobile Phones, Video Game Consoles, GPS Receivers, Printers

---

**Role of Embedded Systems in Our Daily Routine**

- **Functionality**: Embedded systems perform specific tasks with a microcontroller as the main component controlling operations.
- **Presence**: Almost every device we use today is an example of embedded systems, found in homes, offices, industries, and automation systems.
- **Impact**: These systems are smart and efficient, increasing their usage day by day.

**Examples of Embedded Systems**

- **Common Devices**:
    - Automobiles
    - Home Security Systems
    - Automated Teller Machines (ATMs)
    - Industrial Robots
    - Digital Cameras
    - Personal Digital Assistants (PDAs)
    - Automatic Washing Machines

- Microwave Ovens

**HOME SECURITY SYSTEM**

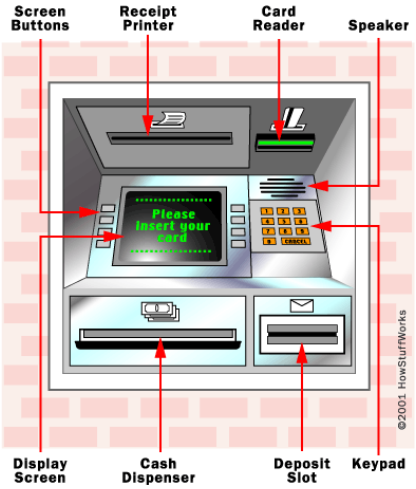
- **Overview**: Home security systems are widely used today.
- **Features**:
  - Checking for fire or gas leakages
  - Detecting suspicious entries
- **Components**:
  - A microcontroller controls all operations.
  - Sensors provide data; if an anomaly is detected, safety alarms are activated.
- **Types of Sensors Used**:
  - Gas sensors
  - Smoke sensors
  - Temperature sensors
  - IR sensors
- **User Interface**: A keypad is included for entering a password at the gate.

**Operation**

- If the correct password is entered, the embedded system opens the gate. If an incorrect password is entered, the alarm is triggered, and the gate remains closed.
- Outputs include alarms and displays; information can be sent to remote locations.
- Family members can monitor activities at home, even when they are not present.
- Home security systems are not limited to residential use; they can also be employed in shops, stores, and industries.
- Many offices and industries utilize security systems that recognize workers via facial recognition or identity cards.

**AUTOMATED TELLER MACHINE (ATM)**

- **Overview**: An ATM is also an embedded system.
- **Functionality**: Customers can access and perform transactions without visiting the bank.
- **Components**:
    - Card reader for detecting cards and accessing user information
    - Keypad for entering commands and passwords
    - Screen for displaying information
    - Printer for receipts
    - Cash dispenser for dispensing cash




- **Network Connectivity**: There is a connection between the bank computer and the ATM through a host computer.
- **Data Management**: All transactions are verified with the bank computer and stored in it.
- **Control**: All input and output operations are managed by a microcontroller, making it a prime example of an embedded system.

## INDUSTRIAL ROBOTS

- **Overview**: Industrial robots are embedded systems that come in various forms, each performing different tasks.
- **Applications**:
    - Moving parts, tools, and materials
    - Assembly operations
    - Manufacturing processes
- **Benefits**: Increased productivity and precision in operations, especially in hard-to-reach areas.
- **Examples**:
    - **Painting Robots**: Widely used in painting applications.
    - **Assembly Robots**: Create assemblies from multiple parts by collecting and assembling them in the correct sequence.

## DIGITAL CAMERA

- **Overview**: A digital camera is an excellent example of an embedded system.
- **Functions**:
    - Capture images (data)
    - Store image data
    - Represent the data

- **Storage**: Images are stored and processed as digital data in bits, eliminating the need for film and increasing storage capacity.

**Image Processing**

- The camera captures an image and converts it to digital form, which is stored in internal memory.
- When connected to a computer, the camera transfers the stored data.
- Smart cameras can analyze images to detect humans, motion, faces, etc.
- Object detection involves both low-level and high-level image processing using various algorithms.

---

**Personal Digital Assistant (PDA)**

- **Overview**: A PDA functions like a handheld personal computer, used before smartphones became prevalent.
- **Functionality**: Acts as an information manager with internet connectivity.
- **Components**:
    - Touchscreen display for user interaction
    - Memory card for data storage
    - Bluetooth or Wi-Fi for connectivity
- **Input Methods**: Some PDAs use keypads instead of touchscreens for data entry.
- **Portability**: Lightweight and multifunctional, making it convenient for managing personal information.

---

**Automatic Washing Machine**

- **Overview**: Embedded systems have simplified the task of washing clothes.
- **Components**:
    - Microcontroller for controlling tasks
    - Sensors and actuators (level sensors, valves, motor)
    - Display and keypad for user input
- **Operation**: The process consists of three cycles: washing, rinsing, and spinning, initiated by the machine itself after the user inputs water temperature and presses the start button.

**Process Control**

- During washing and rinsing, water is added to the drum through pipes, controlled by valves checked by level sensors.
- The drum rotates for a preset time, after which water is drained.
- During the spinning cycle, no water is added, and the drum rotates for a set time.
- All processes are managed by the microcontroller program, with timing adjustable via the keypad.

**Microwave Oven**

- **Overview**: The operating systems within a microwave oven connect directly through electrical pathways.
- **Power Supply**: Electricity enters through a power cord and moves through fuses to the computer system.
- **Control Mechanism**: The computer system relays information through a switch called a “Triac,” which protects against electrical issues.
- **Cooking Process**: If all systems work correctly, the Triac activates the high-voltage transformer, generating microwaves that cook food.

**Command Execution**

- The embedded system in the microwave translates commands from the keypad into operational instructions.
- For example, programming the microwave to operate on high for two minutes triggers the transformer to run at full power for that duration.
- Once the time expires, the embedded system commands the transformer to turn off.