EE4216 Hardware for IoT

Chapter 3

Interface with Sensors and Actuators

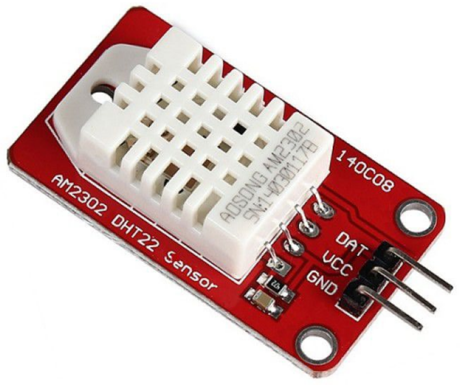
Dr. Zhang Jianwen
elezhan@nus.edu.sg
Office: E4-05-21

Use GPIOs to interface with the sensors

- GPIO can be configured as digital or analogy inputs.
  - DHT22: Temperature and humidity sensor
  - HC-SR04 Ultrasonic sensor
  - Relay
  - Light Sensor
  - Capacitive soil humidity sensor

Interface with DHT22

- DHT22 -- Temperature and humidity digital sensor
  - Temperature and humidity digital sensor
  - Temperature range: $-40^oC - 80^oC$ and accuracy $\pm 0.5^oC$.
  - Humidity range: 0% - 100% with accuracy $\pm(2-5)\%$.
  - Custom one-wire protocol require careful timing.
  - Read once every 2 seconds.
  - Working voltage 3.3V-6V



Interface with DHT22

- Connections
  - VCC: Connect to 3.3V from the MCU board
  - DATA: Connect to a digital GPIO pin (e.g., GPIO 4)
  - GND: Connect to GND on the MCU board
  - A pull-up $10\text{k}\Omega$ resistor between DATA and VCC.
- DHT Sensor Library -- DHT sensor library by Adafruit.
  - Abstracts the low-level communication with the sensor
  - Offers simple APIs to retrieve temperature and humidity readings

Data format for DHT22

Bits	Content
0-7	Humidity high byte
8-15	Humidity low byte
16-23	Temperature high byte
24-31	Temperature low byte
32-39	Checksum

$$Reading = integer\ part + \frac{Low\ byte}{256}$$

For example, if the sensor reads 55.3% humidity:

High Byte: 0x37 (which is 55 in decimal)

Low Byte: 0x4D (which is 77 in decimal, representing 0.3 when normalized)

Interface with DHT22

- Interface with DHT22 using library APIs

  - `void begin();`
  Initiate the DHT sensor.

  - `float readHumidity();`
  Read the relative humidity as a percentage.

  - `float readTemperature(bool isFahrenheit = false);`
  Read the temperature. It returns the temperature in Celsius by default.

Return NaN if reading is not successful.

Interface with DHT22

- Interface with DHT22 using library APIs

  - `float computeHeatIndex(float temperature, float humidity, bool isFahrenheit = true);`
    Calculates the heat index (also known as the "feels-like" temperature) based on the current temperature and humidity.

  - `float convertCtoF(float celsius);`
    Converts a temperature from Celsius to Fahrenheit.

  - `float convertFtoC(float fahrenheit);`
    Converts a temperature from Fahrenheit to Celsius.

```
#include "DHT.h“    //include the library.

#define DHTPIN 4    // define the constant of the GPIO pin to use.
#define DHTTYPE DHT22  // define the type of sensor.

DHT dht(DHTPIN, DHTTYPE); // create an instance of the DHT class.

void setup() {
    Serial.begin(115200);
    dht.begin();       //initiate the sensor. Call before read the data.
}

void loop() {
    delay(2000);       // wait 2 seconds.

    // Read temperature or humidity takes about 250 milliseconds.
    float humidity = dht.readHumidity();
    float temperature = dht.readTemperature();

    // Check if any reads failed and exit early (to try again).
    if (isnan(humidity) || isnan(temperature)) {
        Serial.println("Failed to read from DHT sensor!");  // handle the error
        return;
    }

    float heatIndex = dht.computeHeatIndex(temperature, humidity, false);

    Serial.print("Humidity: ");
    Serial.print(humidity);
    Serial.println(" %");

    Serial.print("Temperature: ");
    Serial.print(temperature);
    Serial.println(" *C");

    Serial.print("Heat Index: ");
    Serial.print(heatIndex);
    Serial.println(" *C");
}
```

Interface HC-SR04

- HC-SR04 -- ultrasonic distance sensor
  - Obstacle detection, distance measurement, and proximity sensing
  - Operating Voltage: 5V DC
  - Operating Current: 15mA
  - Measuring Range: 2 cm to 400 cm.
  - Resolution: Approximately 0.3 cm
  - Measuring Angle: 15 degrees

Interface HC-SR04

- Connections
  - VCC: Connect to 5V from the MCU board
  - Trig: output trigger pin (GPIO), output 10 µs TTL pulse
  - Echo: input echo pin (GPIO), pulse width proportional to the distance
  - GND: Connect to GND on the MCU board
- Programming
  - Custom code with the built-in function pulseIn()
  - NewPing Library

Interface HC-SR04

- Interface with HC-SR04 built-in function

```
long pulseIn(uint8_t pin, uint8_t value, unsigned long timeout);
```

❖ **pin**: GPIO pin number used.
❖ **value**: the type of pulse to measure. (HIGH or LOW)
❖ **timeout**: (optional) number of microseconds to wait for the pulse start. Default is no timeout.
❖ **return**: the number of microseconds or 0 for no pulse.

Example:

```
#define ECHO_PIN 18 // Pin connected to the ECHO pin of HC-SR04

long duration = pulseIn(ECHO_PIN, HIGH);
```

```
#define TRIG_PIN 5
#define ECHO_PIN 18

void setup() {
  // Initialize the serial communication:
  Serial.begin(115200);

  // Configure the pins:
  pinMode(TRIG_PIN, OUTPUT);
  pinMode(ECHO_PIN, INPUT);
}

void loop() {
  // Clear the trigger pin
  digitalWrite(TRIG_PIN, LOW);
  delayMicroseconds(2);

  // Send a 10us pulse to the trigger pin. This causes the sensor to send out an ultrasonic burst.
  digitalWrite(TRIG_PIN, HIGH);
  delayMicroseconds(10);
  digitalWrite(TRIG_PIN, LOW);

  // Measure the duration of the echo pin's pulse in microseconds.
  long duration = pulseIn(ECHO_PIN, HIGH);

  // Calculate the distance in centimeters:
  float distance = (duration * 0.0343) / 2;

  // Print the distance to the Serial Monitor
  Serial.print("Distance: ");
  Serial.print(distance);
  Serial.println(" cm");

  // Add a delay before the next measurement
  delay(500);
}
```

Sound speed in air 0.0343 cm/μs or 343 m/s

$$Distance = \left(\frac{duration \times 0.0343}{2}\right) \text{ cm}$$

Interface HC-SR04

- Other consideration
  - The sensor may return erroneous values if the object is out of range.
  - To improve accuracy, take multiple readings and average them.

```
float getDistance() {
    long sum = 0;

    //read 5 times and take average of them.
    for (int i = 0; i < 5; i++) {
        digitalWrite(TRIG_PIN, LOW);
        delayMicroseconds(2);
        digitalWrite(TRIG_PIN, HIGH);
        delayMicroseconds(10);
        digitalWrite(TRIG_PIN, LOW);
        long duration = pulseIn(ECHO_PIN, HIGH);
        float distance = duration * 0.0343 / 2;
        sum += distance;
        delay(50); // Small delay between readings
    }
    return sum / 5.0; // Return the average distance
}

void loop() {
    float distance = getDistance();

    // handling out-of-range error
    if (distance > 400 || distance < 2) {
        Serial.println("Out of range");
    } else {
        Serial.print("Distance: ");
        Serial.print(distance);
        Serial.println(" cm");
    }
    delay(1000);
}
```

Interface HC-SR04

- Interface using NewPing library APIs

Constructor: `NewPing(trigger_pin, echo_pin, max_distance);`
Create an instance of the Newping class.

❖ trigger_pin: The GPIO pin connected to the Trig pin of the ultrasonic sensor.
❖ echo_pin: The GPIO pin connected to the Echo pin of the ultrasonic sensor.
❖ max_distance: (Optional) The maximum distance you want to measure (in centimeters).

```
#include <NewPing.h>

#define TRIG_PIN 5
#define ECHO_PIN 18
#define MAX_DISTANCE 400

NewPing sonar(TRIG_PIN, ECHO_PIN, MAX_DISTANCE);
```

Interface HC-SR04

- `ping();`

Sends a ping and returns the echo time in microseconds.

```
unsigned int duration = sonar.ping();
```

- `ping_cm();`

Sends a ping and returns the distance in centimeters. If the object is out of range, it returns 0.

```
unsigned int distance = sonar.ping_cm();
Serial.print("Distance: ");
Serial.print(distance);
Serial.println(" cm");
```

- `ping_in();`

Sends a ping and returns the distance in inches. If the object is out of range, it returns 0.

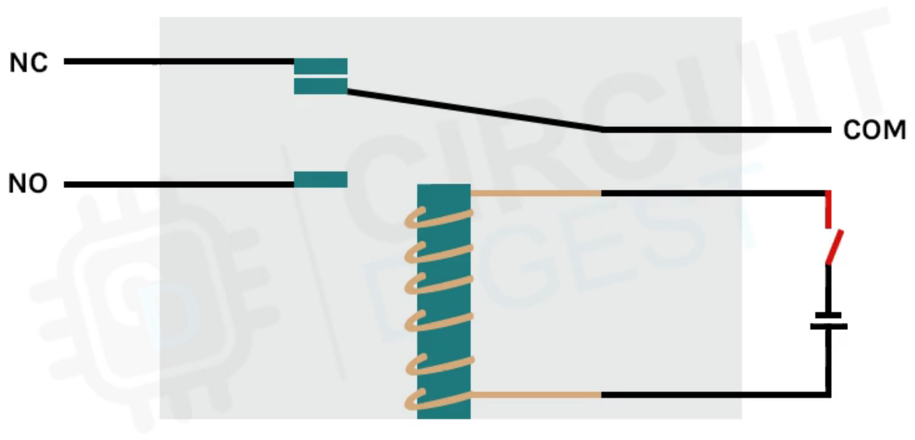
Interface HC-SR04

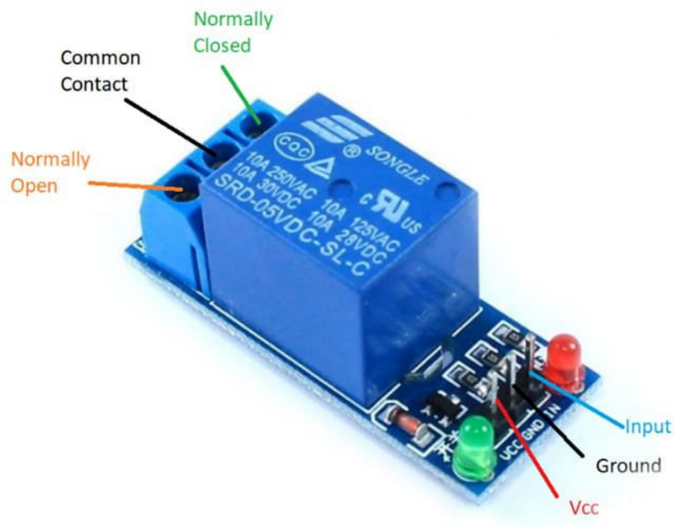
- ping_median(iterations);
  Sends multiple pings (as specified by iterations) and returns the median distance in centimeters. This helps in reducing noise and improving accuracy.

```
unsigned int distance = sonar.ping_median(5);
Serial.print("Median Distance: ");
Serial.print(distance);
Serial.println(" cm");
```

Interface with relay

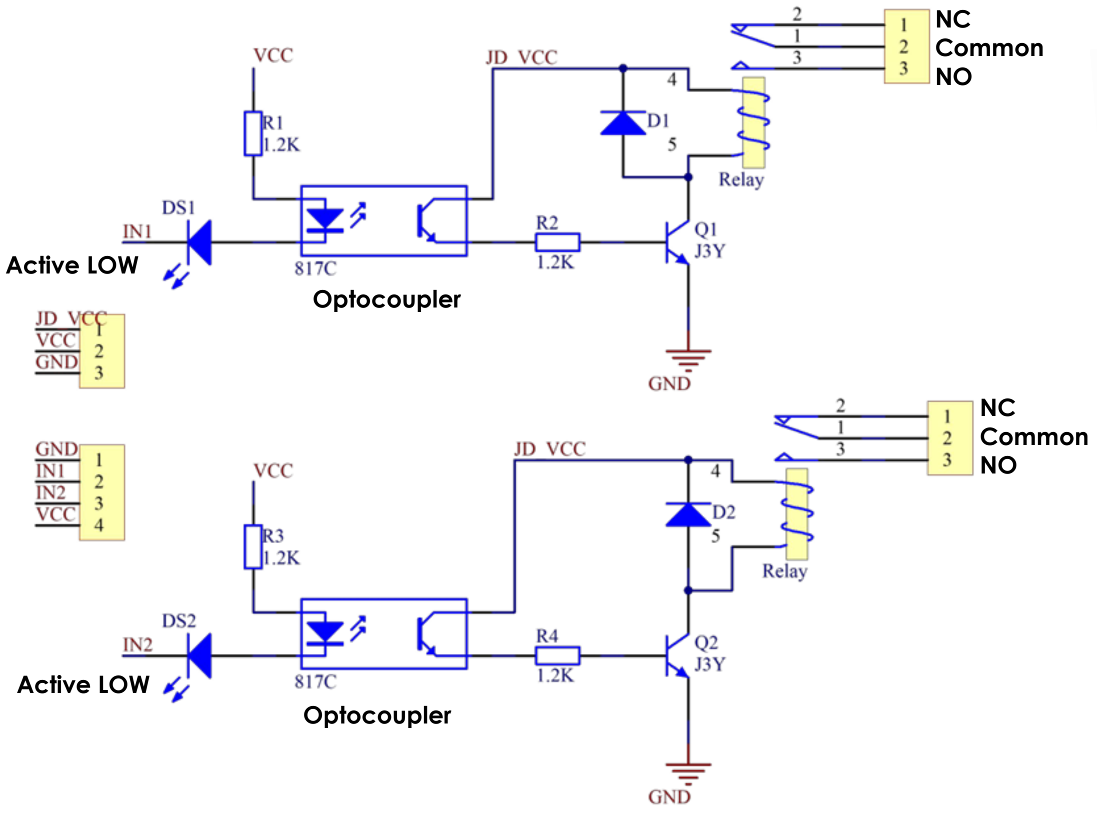
- Relay – Control high-voltage devices using MCU

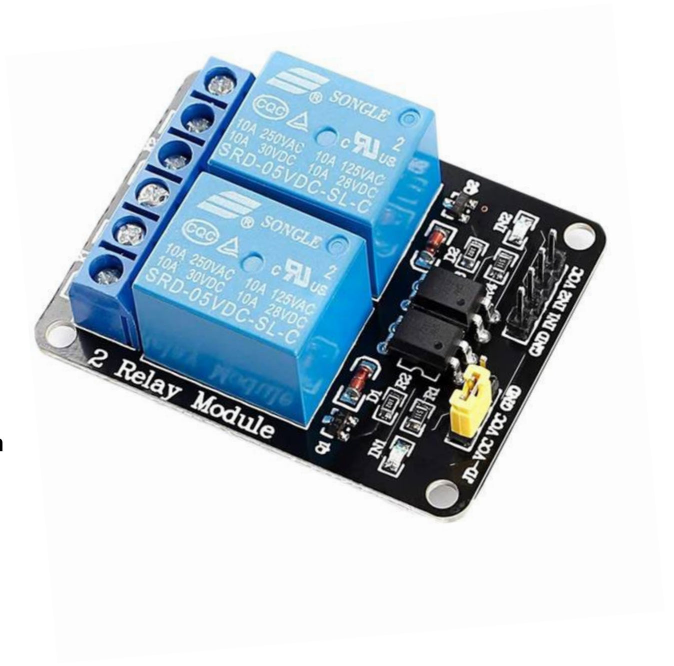




Interface with relay modules

- Relay -- 2 channel DC 5V Relay Module





Interface with relay modules

- Components of the relay module
  - Relay Coil: When energized by a small current, this coil creates a magnetic field that moves the internal switch contacts.
  - Common (COM) Terminal: The common terminal connected to either the Normally Open (NO) or Normally Closed (NC) terminal, depending on the relay state.
  - Normally Open (NO) Terminal: This terminal is not connected to the COM terminal when the relay is inactive. When the relay is activated, this terminal connects to COM, allowing current to flow.
  - Normally Closed (NC) Terminal: This terminal is connected to the COM terminal when the relay is inactive. When the relay is activated, this connection breaks.
  - Control Pins (IN1, IN2, etc.): Connected to the MCU GPIO pins to control the relay.

Interface with relay modules

- Wiring for 2 channel relay module to MCU
  - VCC: Connect to the 5V pin from MCU board. This powers the relay coil.
  - GND: Connect to the GND pin from MCU board. This completes the circuit.
  - IN1: Connect to a GPIO pin from MCU board(e.g., GPIO 4). This controls Relay 1.
  - IN2: Connect to another GPIO pin from MCU board(e.g., GPIO 5). This controls Relay 2.

Load Connections:

- COM (Common): Connect to one terminal of the load (e.g., the live wire of a light bulb).
- NO (Normally Open): Connect to the power source (e.g., 220V AC or 12V DC). The circuit will close when the relay is activated.
- NC (Normally Closed): This is connected if you need the circuit to be closed when the relay is inactive (usually left unconnected).

Interface with relay modules

– Safety Considerations

- Isolation: Ensure that the relay module's optocoupler provides adequate isolation between the high-voltage and low-voltage sides.
- High Voltage Handling: When dealing with AC mains voltage (e.g., 220V), ensure all connections are secure, insulated, and away from accidental contact.
- Current Rating: Verify that the relay's current rating matches the requirements of the load you intend to control. Most relays on such modules are rated for 10A at 250V AC or 30V DC.
- Use a Separate Power Supply (Optional): If your relay module draws more current than the ESP32 can supply, consider using an external 5V power supply to power the relay module, connecting only the GNDs together.

Interface with relay modules

- Applications
  - Home Automation: Control lights, fans, or appliances remotely via Wi-Fi or Bluetooth.
  - Industrial Automation: Control motors, solenoids, or other industrial equipment based on sensor inputs.
  - Security Systems: Activate alarms or lock mechanisms when certain conditions are met.
  - Remote Control: Combine the relay control with Wi-Fi or Bluetooth capabilities to create a remotely controllable power switch.
  - Sensor-Based Activation: Use sensors (like motion detectors, temperature sensors, etc.) to automatically control the relay.

Interface with relay modules

- Basic GPIO Control Functions.

`pinMode(pin, mode);`
Configures a specified pin to behave either as an input or an output.
❖ **pin**: GPIO pin number
❖ **mode**: OUTPUT, INPUT, or INPUT_PULLUP.

```
pinMode(RELAY1_PIN, OUTPUT); // Set the relay control pin as an output
```

`digitalWrite(pin, value);`
Sets the output level of a digital pin to HIGH (5V or 3.3V) or LOW (0V).
❖ **pin**: the GPIO number of the pin to control.
❖ **value**: HIGH or LOW.

```
digitalWrite(RELAY1_PIN, LOW); // Activate the relay (if low-level triggered)
```

Interface with relay modules

```
digitalRead(pin);
Reads the value from a specified digital pin, returning HIGH or LOW.
❖ pin: GPIO pin number
```

```
int state = digitalRead(RELAY1_PIN); // Read the current state of the relay control pin
```

Interface with relay modules

- Time Functions.

```
delay(ms);
Pauses the program for a specified number of milliseconds.
❖ ms: The number of milliseconds to pause.
```

```
delay(1000); // Wait for 1 second
```

```
millis();
Returns the number of milliseconds since the program started running.
Useful for non-blocking timing control.
```

```
unsigned long currentMillis = millis();
```

```
/*  Example that turns a relay on and off every second. */

#define RELAY1_PIN 4  // GPIO pin connected to IN1
#define RELAY2_PIN 5  // GPIO pin connected to IN2

void setup() {
    // Initialize the relay control pins as outputs
    pinMode(RELAY1_PIN, OUTPUT);
    pinMode(RELAY2_PIN, OUTPUT);

    // Set initial state for relays (usually HIGH to keep them off if low-level trigger)
    digitalWrite(RELAY1_PIN, HIGH);
    digitalWrite(RELAY2_PIN, HIGH);
}

void loop() {
    // Activate Relay 1 (turn on the device connected to Relay 1)
    digitalWrite(RELAY1_PIN, LOW); // LOW to activate if it's a low-level trigger
    delay(1000); // Keep it on for 1 second

    // Deactivate Relay 1 and activate Relay 2
    digitalWrite(RELAY1_PIN, HIGH); // HIGH to deactivate
    digitalWrite(RELAY2_PIN, LOW);  // LOW to activate Relay 2
    delay(1000); // Keep it on for 1 second

    // Deactivate Relay 2
    digitalWrite(RELAY2_PIN, HIGH); // HIGH to deactivate Relay 2
    delay(1000); // Wait 1 second before looping again
}
```

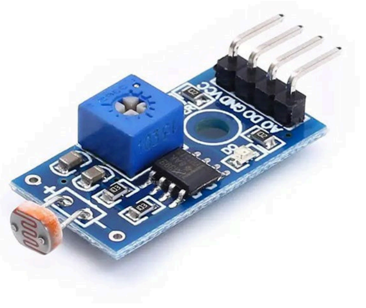
```
/* toggle the relay state by a push button with btn debouncing.*/

const int buttonPin = 2;  // The number of the pushbutton pin
const int relayPin = 4;   // The number of the relay control pin

int buttonState;             // The current reading from the input pin
int lastButtonState = LOW;   // The previous reading from the input pin

unsigned long lastDebounceTime = 0;  // The last time the output pin was toggled
unsigned long debounceDelay = 50;    // The debounce time; increase if the output flickers

void setup() {
  pinMode(buttonPin, INPUT); //
  pinMode(relayPin, OUTPUT);
}

void loop() {
  int reading = digitalRead(buttonPin);

  // button debounce
  if (reading != lastButtonState) {  // check if there is btn state change.
    lastDebounceTime = millis();     // record the time to start debounce if a btn change is detected.
  }

  if ((millis() - lastDebounceTime) > debounceDelay) {
    if (reading != buttonState) {
      buttonState = reading;         // update the btn state.

      if (buttonState == HIGH) {
        digitalWrite(relayPin, !digitalRead(relayPin)); // toggle the relay state.
      }
    }
  }

  lastButtonState = reading;
}
```

```
/* Basic Web Server to Control Relay */
#include <WiFi.h>
#include <ESPAsyncWebServer.h>

const char* ssid = "your-SSID";
const char* password = "your-PASSWORD";

AsyncWebServer server(80);

#define RELAY_PIN 4

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

  pinMode(RELAY_PIN, OUTPUT);
  digitalWrite(RELAY_PIN, HIGH);

  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }

  Serial.println("Connected to WiFi");

  server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send(200, "text/plain", "Hello, Relay!");
  });

  server.on("/relay_on", HTTP_GET, [](AsyncWebServerRequest *request){
    digitalWrite(RELAY_PIN, LOW);
    request->send(200, "text/plain", "Relay ON");
  });

  server.on("/relay_off", HTTP_GET, [](AsyncWebServerRequest *request){
    digitalWrite(RELAY_PIN, HIGH);
    request->send(200, "text/plain", "Relay OFF");
  });

  server.begin();
}

void loop(){}
```

Interface with light sensor module

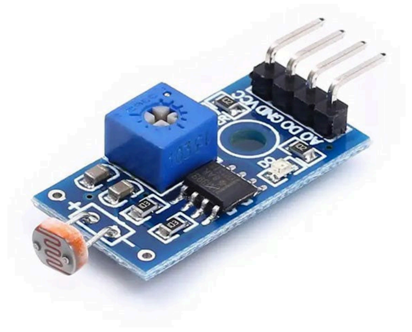
- Light sensor module – detect and measure the intensity of ambient light
  - Light-sensitive element

    Light dependent resistor/photodiode/phototransistor.
  - Supporting circuitry.
    - Op-Amp to amplify the small signal from light-sensitive element.
    - Comparator to compare the signal strength against a reference voltage.
    - Potentiometer to adjust the refence voltage level for different applications.
  - Working voltage 3.3V or 5V.



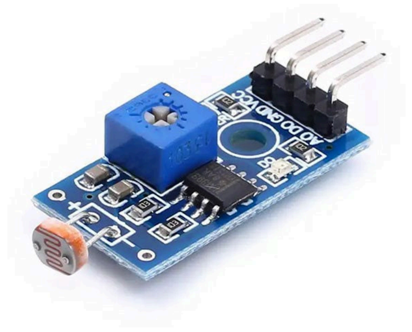
Interface with light sensor module

- Interface
  - VCC: Power supply pin (3.3V or 5V).
  - GND: Ground pin.
  - AO (Analog Output): Provides an analog voltage proportional to the light intensity.
  - DO (Digital Output): Provides a digital HIGH/LOW signal based on the light intensity compared to a set threshold.



Interface with light sensor module

- Interface
  - VCC: Power supply pin (3.3V or 5V).
  - GND: Ground pin.
  - AO (Analog Output): Provides an analog voltage proportional to the light intensity.
  - DO (Digital Output): Provides a digital HIGH/LOW signal based on the light intensity compared to a set threshold.



Interface with light sensor module

- Connections
  - VCC: Connect to 3.3V (or 5V, depending on the module specifications).
  - GND: Connect to the ground (GND) of the ESP32-S3.
  - AO (Analog Output): Connect to an analog input pin on the ESP32-S3 (e.g., GPIO 34).
  - DO (Digital Output): Connect to a digital input pin on the ESP32-S3 (optional, for using the digital output).

Interface with light sensor module

- Related functions
  - Analog Input API **analogRead()**

Syntax:

```
int value = analogRead(pin);
```

❖ **pin**: the analog pin number where the sensor's analog output is connected.

❖ **Return**: an integer value representing the voltage level on the pin, scaled between 0 and 4095. The ADC width is 12-bit by default. The width can be set and the attenuation level can be set.

Interface with light sensor module

– Digital Input API (`digitalRead()`)

Syntax:

```
int state = digitalRead(pin);
```

❖ **pin**: the digital pin number where the sensor's digital output is connected.

❖ **state**: the function returns either `HIGH` or `LOW`.

```
int digitalState = digitalRead(25); // Read from digital pin GPIO 25
if(digitalState == HIGH) {
    Serial.println("Light level is above the threshold");
} else {
    Serial.println("Light level is below the threshold");
}
```

Interface with light sensor module

– Pin Configuration API (`pinMode()`)

```
Syntax:

pinMode(pin, mode);
```

❖ **pin**: the pin number you want to configure.

❖ **mode**: `INPUT`, `OUTPUT`, or `INPUT_PULLUP`.

```
pinMode(25, INPUT); // Set GPIO 25 as an input
```

```
const int analogPin = 34;  // Analog output pin from the light sensor
const int digitalPin = 25; // Digital output pin from the light sensor

void setup() {
  Serial.begin(115200);    // Initialize serial communication
  pinMode(digitalPin, INPUT); // Set digital pin as input
}

void loop() {

  // Read the analog value from the sensor
  int lightLevel = analogRead(analogPin);

  // Read the digital state from the sensor
  int digitalState = digitalRead(digitalPin);

  Serial.print("Analog Light Level: ");
  Serial.println(lightLevel);      // Print the analog light level

  Serial.print("Digital Output State: ");
  Serial.println(digitalState == HIGH ? "HIGH" : "LOW"); // Print the digital state


  delay(500); // Wait for 500ms before the next loop
}
```

Interface with light sensor module

- Test and Calibrate
  - Use the serial monitor in the Arduino IDE to observe the values.
  - Calibration the threshold by adjusting the potentiometer on the light sensor module.

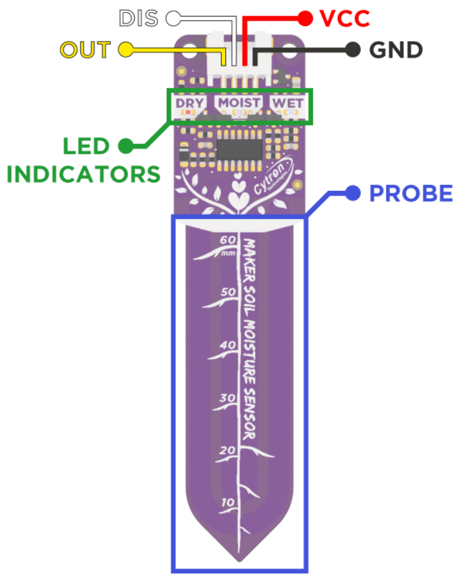
Interface with light sensor module

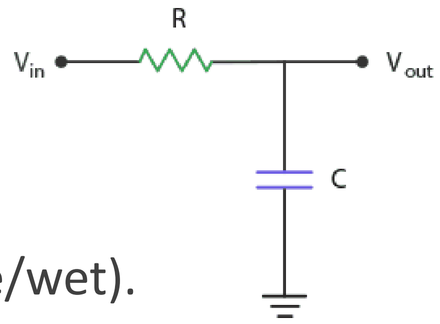
- Applications
  - Automatic Lighting: Turn lights on or off depending on the ambient light levels.
  - Light-Activated Alarms: Trigger alarms when light levels change unexpectedly.
  - Ambient Light Monitoring: Use in IoT projects to monitor environmental light levels and adjust settings (e.g., screen brightness) accordingly.
  - Robotics: Used in robots for light-following or avoiding behaviors.
  - IoT Projects: Monitor environmental conditions and send data to a cloud platform for analysis.

Interface with soil moisture sensor

- Capacitive soil moisture sensor
  - Measure the medium dielectric permittivity.

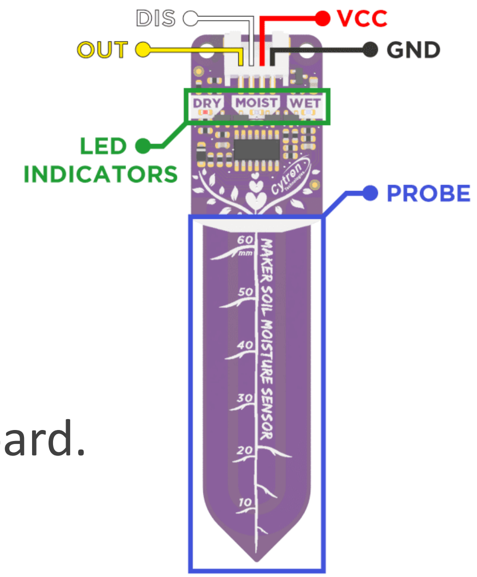
    It varies with the moisture.
  - No corrosion compared to resistive sensors.
  - Low current consumption (a few milliamps).
  - Save power by turning off the sensor.
  - Double-sided probe(more sensitive).
  - Three LEDs to indicate moisture levels (dry/moisture/wet).





Interface with soil moisture sensor

- Interface and connections
  - VCC: power supply. 3.3V to 5V on board
  - GND: Ground to the board.
  - Analog Output: To an analog input pin on the board.
    - Higher moisture → Lower voltage
    - Calibration is needed.
  - Disable pin: Active high. Connect this to a GPIO pin on the board and configure it as digital output pin.



```
/* capacitive moisture sensor example */
// Define pins
const int moisturePin = 36;  // Analog pin connected to sensor OUT
const int disablePin = 4;    // GPIO pin to control sensor power

void setup() {
  Serial.begin(115200);     // Start the serial communication
  pinMode(disablePin, OUTPUT);
  digitalWrite(disablePin, HIGH); // Disable the sensor by setting the disable pin HIGH
}

void loop() {
  // Enable the sensor
  digitalWrite(disablePin, LOW); // Enable the sensor by pulling the disable pin LOW

  delay(500);                    // Wait for sensor to stabilize

  // Read the moisture value
  int sensorValue = analogRead(moisturePin);

  // Print the moisture value
  Serial.print("Soil Moisture Value: ");
  Serial.println(sensorValue);

  // Disable the sensor to save power
  digitalWrite(disablePin, HIGH);  // Disable the sensor by pulling the disable pin HIGH

  // Wait before next reading
  delay(10000);  // Adjust delay as needed
}
```

What is this value?
Displays the raw sensor value, which corresponds to the soil moisture level. A lower value typically indicates higher moisture content.

Interface with soil moisture sensor

- Related functions

```
analogRead(sensorPin);

1. Reads the analog voltage from the soil moisture
   sensor.
2. The ESP32-S3’s ADC returns a value between 0 (0V)
   and 4095 (3.3V) based on the moisture content since
   it has a 12-bit ADC resolution.
3. Calibrate the sensor for a specific soil and
   environment by measuring the values in fully dry (0)
   and fully saturated soil (4095).
4. Map these raw ADC values to a percentage moisture
   for easier interpretation.
```

Interface with soil moisture sensor

```
map(value, fromLow, fromHigh, toLow, toHigh)
```

- ❖ **value:** the number you want to map.
- ❖ **fromLow:** the lower bound of the value’s current range.
- ❖ **fromHigh:** the upper bound of the value’s current range.
- ❖ **toLow:** the lower bound of the target range.
- ❖ **toHigh:** the upper bound of the target range.

```
//an analog sensor that gives values from 0 to 1023, but you
want to map these values to a range of 0 to 100.
int sensorValue = analogRead(A0);
int mappedValue = map(sensorValue, 0, 1023, 0, 100);
```

Interface with soil moisture sensor

```
constrain(value, min, max);
```

limits a value to a specified range. If the value is less than the lower limit, it returns the lower limit; if it’s greater than the upper limit, it returns the upper limit.

- **value:** the number you want to constrain.
- **min:** the lower bound to constrain the value to.
- **max:** the upper bound to constrain the value to.

```
// ensure value stays within 0 to 100
int value = 150;
// constrainedValue will be 100
int constrainedValue = constrain(value, 0, 100);
```

```
/* control an LED brightness based on a sensor reading, where the
sensor's output is mapped to the range 0-255 (for PWM), but you
want to ensure that the LED never goes below a brightness level of
50 or above 200. */

int sensorValue = analogRead(A0);
int brightness = map(sensorValue, 0, 1023, 0, 255);
brightness = constrain(brightness, 50, 200);
analogWrite(ledPin, brightness);
```

```
const int moisturePin = 36;  // Analog pin connected to sensor OUT
const int disablePin = 4;    // GPIO pin to control sensor power

// Define the range for dry and wet soil (calibration)
const int dryValue = 3500;  // Example value for dry soil
const int wetValue = 1500;  // Example value for wet soil

void setup() {
  Serial.begin(115200);
  pinMode(disablePin, OUTPUT);
  digitalWrite(disablePin, HIGH); // Disable the sensor by setting the disable pin HIGH

}

void loop() {

  digitalWrite(disablePin, LOW); // Enable the sensor by pulling the disable pin LOW
  delay(500);                    // Wait for sensor to stabilize

  int sensorValue = analogRead(sensorPin);

  // Map the sensor value to a percentage (0% - 100%)
  int moisturePercent = map(sensorValue, dryValue, wetValue, 0, 100);

  // Constrain the value to 0-100% range
  moisturePercent = constrain(moisturePercent, 0, 100);

  Serial.print("Soil Moisture Level: ");
  Serial.print(moisturePercent);
  Serial.println("%");

  // Disable the sensor to save power
  digitalWrite(disablePin, HIGH);

  delay(1000);
}
```

Interface with soil moisture sensor

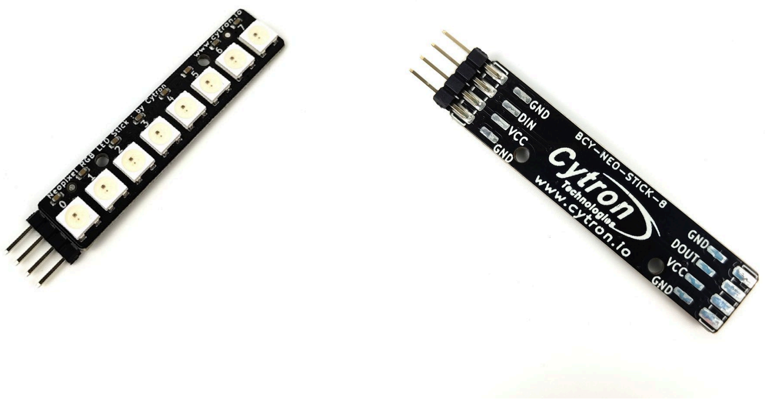
- Considerations
  - Soil Type: Different soil types (e.g., sandy, clay) might require calibration to ensure accurate readings.
  - Temperature Sensitivity: Although generally stable, extreme temperature changes can affect the accuracy.
  - Power Supply: Ensure the sensor is powered with the correct voltage as specified by the module.

Interface with soil moisture sensor

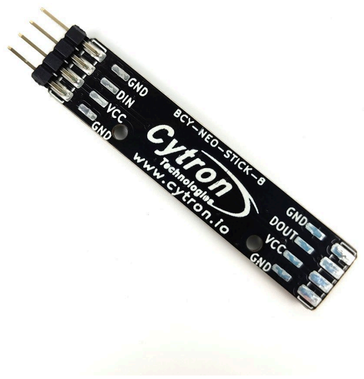
- Applications
  - Automatic Irrigation Systems: Control watering based on soil moisture levels.
  - Smart Gardening: Monitor plant health and optimize water usage.
  - Environmental Monitoring: Use in IoT projects to collect data on soil conditions over time.
- Advantages
  - Durability: Less prone to corrosion, making it ideal for long-term outdoor use.
  - Ease of Use: Simple to connect and use with microcontrollers.
  - Low Maintenance: Doesn’t require frequent calibration or cleaning.

Interface with RGBLED

- NEO-PIXEL RGBLED stick
  - WS2812 or similar LED
  - Individual addressable RGB LED.
  - RGB in each LED is adjustable.
  - All LEDs share the same power lines and data line.
  - More units can be put in tandem. (Power and response rate constraints)
  - Each LED on full brightness can draw 80 mA current.

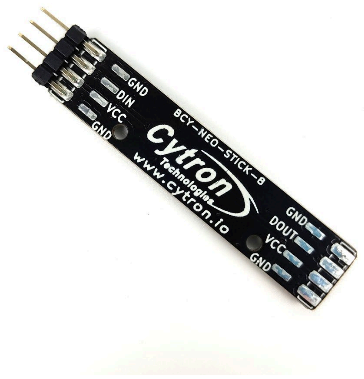


Interface with RGBLED



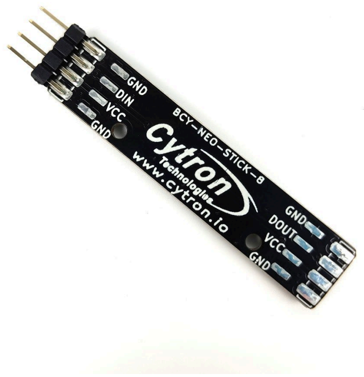
- Interface and connections
  - Input
    - DIN: Connect to one of the GPIO pins (e.g., GPIO 5).
    - VCC: Connect to the 5V pin on the developing board or an external 5V power supply.
    - GND: Connect to the GND pin on the developing board
  - Output – connect to next LED stick of the same type if needed.
    - DOUT: Connect to the DIN of next stick
    - VCC: Connect to the VCC of next stick
    - GND: Connect to the GND of next stick

Interface with RGBLED



- Interface and connections
  - Input
    - DIN: Connect to one of the GPIO pins (e.g., GPIO 5).
    - VCC: Connect to the 5V pin on the developing board or an external 5V power supply.
    - GND: Connect to the GND pin on the developing board
    - Optional but recommended: Put a 1000μF capacitor between VCC pin and GND pin to filter the power supply. and a resistor of 200-500Ω in series between the GPIO pin and the DIN pin to reduce data line noise .

Interface with RGBLED



- Output – connect to next LED stick of the same type if needed.
  - DOUT: Connect to the DIN of next stick
  - VCC: Connect to the VCC of next stick
  - GND: Connect to the GND of next stick
  - Make sure your power supply can provide sufficient current. Consider external 5V supply for all LEDs if needed.

Library and APIs

- Install the Adafruit NeoPixel library.

- Adafruit_NeoPixel(NUMPIXELS, PIN, NEO_GRB + NEO_KHZ800);
  Initializes the NeoPixel object.
  - ❖ **NUMPIXELS**: The number of LEDs in your NeoPixel strip or stick.
  - ❖ **PIN**: The GPIO pin number on the board that is connected to the data input of NeoPixel.
  - ❖ **NEO_GRB**: This indicates the color order used by the LEDs (Green, Red, Blue). Some LEDs may use NEO_RGB or another variant.
  - ❖ **NEO_KHZ800**: This indicates the communication speed. Most NeoPixels use 800kHz.

Library and APIs

- strip.begin();

Initializes the NeoPixel library and prepares the data line for communication with the LEDs.

- strip.show();

Updates the LED strip with any changes made to pixel colors. It should be called after setting colors with setPixelColor() to actually show the changes.

- strip.setPixelColor(n, color);

Sets the color of an individual LED.

  - **n**: The index of the LED you want to change (0 to NUMPIXELS-1).
  - **color**: The color you want to set, created using the strip.Color() function.

Library and APIs

- `strip.setPixelColor(n, r, g, b);`

Sets the color of an individual LED using separate red, green, and blue values. **r, g, b** are in all in the range 0-255.

- `strip.Color(r, g, b);`

Creates a 32-bit color value from separate red, green, and blue components. Each color is represented by 8 bits.

```
   uint32_t red = strip.Color(255, 0, 0);
```

- `strip.setBrightness(brightness);`

Adjusts the brightness of the entire strip. The range of **brightness** is between 0 (off) and 255 (full brightness).

Library and APIs

- `strip.clear();`

Sets all pixels to 'off' (sets them to black). Call for strip.show() after clear() is needed to update the LEDs.

- `strip.getPixelColor(n);`

Returns the current color of the LED at index **n** as a 32-bit RGB value.

```
  uint32_t currentColor = strip.getPixelColor(0);
```

- `strip.gamma32(uint32_t color);`

Applies gamma correction to a 32-bit **color** value. This is useful to correct the non-linear response of LED brightness to make colors look more natural.

```
 uint32_t correctedColor = strip.gamma32(strip.Color(255, 0, 0));
```

Library and APIs

- `strip.numPixels();`

Returns the number of pixels in the NeoPixel object (the value set by NUMPIXELS during initialization).

- `strip.updateLength(x);`

Dynamically changes the number of LEDs controlled by the NeoPixel object. This is rarely used but can be helpful in dynamic projects.

- `strip.updateType(x);`

Changes the LED type (GRB, RGB, etc.) dynamically. This is useful if you’re switching between different types of NeoPixel strips.

```
#include <Adafruit_NeoPixel.h>

#define PIN 5
#define NUMPIXELS 8

Adafruit_NeoPixel strip = Adafruit_NeoPixel(NUMPIXELS, PIN, NEO_GRB + NEO_KHZ800);

void setup() {
  strip.begin();
  strip.setBrightness(50); // Set brightness to 50 out of 255
  strip.show(); // Initialize all pixels to 'off'
}

void loop() {
  for (int i = 0; i < strip.numPixels(); i++) {
    strip.setPixelColor(i, strip.Color(0, 150, 0)); // Set pixel to green
    strip.show();
    delay(100);
  }

  strip.clear(); // Turn off all pixels
  strip.show();
  delay(500);

  for (int i = 0; i < strip.numPixels(); i++) {
    uint32_t color = strip.getPixelColor(i); // Get the current color of the pixel
    strip.setPixelColor(i, strip.gamma32(color)); // Apply gamma correction
    strip.show();
    delay(100);
  }
}
```

```
/* effect of filling the LED stick with a color, one pixel at a time */

#include <Adafruit_NeoPixel.h>

#define PIN 5
#define NUMPIXELS 8

Adafruit_NeoPixel strip = Adafruit_NeoPixel(NUMPIXELS, PIN, NEO_GRB + NEO_KHZ800);

void setup() {
  strip.begin();
  strip.show();
}

void loop() {
  colorWipe(strip.Color(255, 0, 0), 50); // Red
  colorWipe(strip.Color(0, 255, 0), 50); // Green
  colorWipe(strip.Color(0, 0, 255), 50); // Blue
}

void colorWipe(uint32_t color, int wait) {
  for (int i = 0; i < strip.numPixels(); i++) {
    strip.setPixelColor(i, color);
    strip.show();
    delay(wait);
  }
}
```

```
/* gradually increases and decreases the brightness of all LEDs, creating a breathing-like effect. */

#include <Adafruit_NeoPixel.h>

#define PIN 5
#define NUMPIXELS 8

Adafruit_NeoPixel strip = Adafruit_NeoPixel(NUMPIXELS, PIN, NEO_GRB + NEO_KHZ800);

void setup() {
  strip.begin();
  strip.show();
}

void loop() {
  breathingEffect(strip.Color(0, 0, 255), 10); // Blue breathing effect
}

void breathingEffect(uint32_t color, int wait) {
  for (int brightness = 0; brightness < 255; brightness++) {
    for (int i = 0; i < strip.numPixels(); i++) {
      strip.setPixelColor(i, color);
    }
    strip.setBrightness(brightness);
    strip.show();
    delay(wait);
  }
  for (int brightness = 255; brightness >= 0; brightness--) {
    for (int i = 0; i < strip.numPixels(); i++) {
      strip.setPixelColor(i, color);
    }
    strip.setBrightness(brightness);
    strip.show();
    delay(wait);
  }
}
```

Interface with RGBLED

- Applications
  - Ambient Lighting: Create dynamic ambient lighting displays with customizable colors and patterns.
  - Wearable Tech: Integrate with wearables for reactive and colorful displays associated with to different sensors like accelerometers or heart rate monitors.
  - Notification System: Use as a visual notification system with different colors or patterns.
  - Digital Art: Develop interactive light sculptures or installations that respond to environmental factors, such as sound, motion, or even weather data.
  - Prototyping: Rapidly prototype IoT devices or smart gadgets that require visual feedback or decorative elements.

Serial ports

- Three serial ports are provided
  - Serial0: used for programming, debugging and communications – avoid to use it for other purpose.
  - Serial1 and Serial2: used for connecting external peripherals and other MCU supporting serial communication protocols.
  - The APIs are similar for all three serial ports.

Serial port connections

- Serial port uses UART
  - TX.
  - RX.
  - GND.
  - Optional
    - CTS(Clear to Send)
    - RTS (Request to Send)

API to initiate the serial communication

- Serial.begin(): initialize the serial0

  Serial.begin(baudrate);

  Serial.begin(baudrate, config);

  Serial.begin(baudrate, config, rxPin, txPin); Serial.begin(baudrate, config, rxPin, txPin, bufferSize);

Start Bit	Data Bits (LSB -> MSB)	Parity Bit (Optional)	Stop Bit
0	1 0 1 0 0 1 0 1		1

SERIAL_8N1

API to initiate the serial communication

- parameters
  - **baudrate**: supporting 300 to 2M bauds. But usually used rates are 9600, 14400 and 115200 bauds.
  - **config**: default setting is ‘SERIAL_8N1’ meaning 8 data bits, no parity, 1 stop bit.
  - **rxPin**, **txPin**: receiving GPIO pin and transmitting GPIO pin.
  - **bufferSize**: 512 bytes by defaults for both transmitting and receiving.

Reading Data from Serial port

- `Serial.available();`

Returns the number of bytes available for reading from the serial buffer. This is useful to check if data is waiting to be read.

- `Serial.read();`

Reads the next byte of incoming serial data. If no data is available, it returns -1.

```
  char incomingByte = Serial.read();
```

- `Serial.readString();`

Reads characters from the serial buffer into a String object until a newline character is encountered or a timeout occurs.

```
  String incomingString = Serial.readString();
```

Reading Data from Serial port

- `Serial.readBytes(buffer, length);`

Reads multiple bytes from the serial buffer into a buffer array. It reads up to length bytes.

- `Serial.write(data);`

Writes binary data to the serial port. It can send single characters, strings, or arrays of bytes.

```
Serial.write('A’);
Serial.write(myBuffer, sizeof(myBuffer));
```

- `Serial.print(data)` and `Serial.println(data);`

Sends text to the serial port. print() sends data without a newline, while println() appends a newline at the end.

Reading Data from Serial port

- `Serial.readStringUntil();`

This function reads characters from the serial buffer into a String until the specified character is encountered.

```
   String incomingString = Serial.readStringUntil('\n’);
```

- `Serial.flush();`

Waits for the transmission of outgoing serial data to complete. Note that it does not clear the incoming data buffer.

```
void setup() {
  Serial.begin(115200); // Start the serial communication at 115200 baud rate
}

void loop() {
  if (Serial.available() > 0) {  // Check if data is available to read

    // Read the incoming data until newline character
    String inputString = Serial.readStringUntil('\n');

    // Print the received data back to the serial monitor
    Serial.print("You entered: ");
    Serial.println(inputString);
  }
}
```

SPI (Serial Peripheral Interface)

- SPI features
  - Synchronous serial communication protocol.
  - Full-Duplex Communication
  - Simple Hardware Connections
  - Supports multiple slave devices
  - Communicate with sensors, displays, SD cards, etc.

SPI (Serial Peripheral Interface)

- SPI pins and connections
  - MOSI (Master Out Slave In) – GPIO 23 (default).
  - MISO (Master In Slave Out) – GPIO 19 (default).
  - SCK (Serial Clock): Clock line driven by the master to synchronize data transmission. – GPIO 18 (default).
  - SS (Slave Select) or CS (Chip Select): Active low. – Any available GPIO

SPI APIs

- `SPI.begin(SCK, MISO, MOSI, SS);`

Initializes the SPI bus and sets the SCK (Serial Clock), MISO (Master In Slave Out), MOSI (Master Out Slave In), and SS (Slave Select) pins.

```
SPI.begin(18, 19, 23, 5);  // Example for ESP32 with custom pins
```

- `SPI.beginTransaction(SPISettings);`

Configures the SPI parameters for a transaction. It accepts an object of type SPISettings which specifies the clock speed, data order (LSBFIRST or MSBFIRST), and data mode (SPI_MODE0, SPI_MODE1, etc.).

```
SPI.beginTransaction(SPISettings(1000000, MSBFIRST, SPI_MODE0));
// 1 MHz, MSB first, Mode 0
```

SPI APIs

- `SPI.transfer(value);`

Transfers a byte of data to the SPI device and simultaneously receives a byte of data from the SPI device. This method is used for full-duplex communication.

```
// Send 0x42 and receive a response
uint8_t response = SPI.transfer(0x42);
```

- `SPI.transfer(buffer, size);`

Transfers multiple bytes of data to the SPI device and stores the response in a buffer. The buffer is a pointer to the data to be sent, and size specifies the number of bytes to transfer.

```
uint8_t data[2] = {0x01, 0x02};
SPI.transfer(data, sizeof(data));  // Send two bytes
```

SPI APIs

- `SPI.endTransaction();`

Ends the current SPI transaction. This function is typically called after all communication with the SPI device is complete.

- `SPI.end();`

Disables the SPI bus, freeing up the pins for other uses.

- `SPISettings(clock, bitOrder, dataMode);`

Construct a SPISettings object for configuring SPI transactions.

  - ❖ **clock**: The SPI clock speed (e.g., 1000000 for 1 MHz).
  - ❖ **bitOrder**: The bit order (either MSBFIRST or LSBFIRST).
  - ❖ **dataMode**: The SPI mode (one of SPI_MODE0, SPI_MODE1, SPI_MODE2, or SPI_MODE3).

```
/* example of how to communicate with an SPI device using the ESP32 */
#include <SPI.h>

const int CS_PIN = 5;  // Chip select pin

void setup() {
  Serial.begin(115200);
  SPI.begin(18, 19, 23, CS_PIN);  // Initialize SPI with custom pins

  pinMode(CS_PIN, OUTPUT);
  digitalWrite(CS_PIN, HIGH);  // Set CS high
}

void loop() {
  digitalWrite(CS_PIN, LOW);  // Select the SPI device
  SPI.beginTransaction(SPISettings(1000000, MSBFIRST, SPI_MODE0));  // 1 MHz, MSB first, Mode 0

  uint8_t response = SPI.transfer(0x42);  // Send data and receive a response

  SPI.endTransaction();
  digitalWrite(CS_PIN, HIGH);  // Deselect the SPI device

  Serial.print("Response: ");
  Serial.println(response, HEX);

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

```
/* example of using SPI to interface with an SD card to log data */
#include <SPI.h>
#include <SD.h>

const int chipSelect = 5;  // CS pin for SD card module

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

  // Initialize SPI communication with SD card
  if (!SD.begin(chipSelect)) { // Initializes the SD card for communication using the specified chip select pin.
    Serial.println("SD card initialization failed!");
    return;
  }

  Serial.println("SD card initialized successfully.");

  // Create or open a file on the SD card
  File dataFile = SD.open("datalog.txt", FILE_WRITE); // Opens a file on the SD card.

  // If the file opened successfully, write to it
  if (dataFile) {
    dataFile.println("Hello, this is a test log."); // Writes data to the file.
    dataFile.close();  // Closes the file to ensure data is properly saved.
    Serial.println("Data written to file.");
  } else {
    // If the file didn't open, print an error
    Serial.println("Error opening datalog.txt");
  }
}

void loop() {
  // Nothing needed in the loop for this basic example
}
```

SPIFFS

- SPI Flash File System(SPIFFS) is a lightweight file system created for embedded devices that use SPI NOR flash on board.
- Read, write, and delete files, as well as format the file system on the ESP32's flash memory (Configuration files, text, or images).
- SPIFFS is very useful when dealing with small amount non-volatile storage without needing external SD cards or similar hardware.
- SPIFFS is already integrated into the core libraries (Inlcude FS.h or SPIFFS.h header files). These libraries provide an abstraction layer for reading and writing files to the flash memory.

SPIFFS APIs

- SPIFFS.begin(bool formatOnFail): Initializes the SPIFFS file system. If formatOnFail is true, it will format the file system if the initialization fails.
- SPIFFS.open(const char* path, const char* mode) : Opens a file from the SPIFFS file system. The path specifies the location of the file, and mode specifies how the file should be opened (e.g., "r" for read, "w" for write, "a" for append).
- SPIFFS.exists(const char* path): Checks if a file exists at the specified path.
- SPIFFS.remove(const char* path): Removes (deletes) a file from the SPIFFS file system.
- SPIFFS.format(): Formats the SPIFFS file system, erasing all the files stored on it.

SPIFFS APIs

- Basic file I/O operations.These functions handle reading and writing data from/to files in SPIFFS.
  - file.read() reads one byte from the file.
  - file.readString() reads the entire file content as a string.
  - file.write() writes a single byte.
  - file.println() writes a string with a newline.

```
// Writing a string to a file
File file = SPIFFS.open("/example.txt", FILE_WRITE);
if (file) {
    file.println("Hello, SPIFFS!");
    file.close();
}

// Reading a file as a string
file = SPIFFS.open("/example.txt", FILE_READ);
if (file) {
    String content = file.readString();
    Serial.println(content);  // Prints: Hello, SPIFFS!
    file.close();
}
```

SPIFFS APIs

- file.available(): Checks if there are more bytes available for reading in the file

```
File file = SPIFFS.open("/example.txt", FILE_READ);
while (file.available()) {
    Serial.write(file.read()); // Read file byte by byte
}
file.close();
```

- file.seek(offset, seekMode): Moves the file pointer to the specified offset. The seekMode can be SeekSet, SeekCur, or SeekEnd to move relative to the start, current position, or end of the file, respectively.

```
File file = SPIFFS.open("/example.txt", FILE_READ);
file.seek(10, SeekSet); // Move to the 10th byte in the file
Serial.println(file.read()); // Read from the 10th byte
file.close();
```

Common Use Cases of SPIFFS

- Storing Configuration Data
- Logging Sensor Data
- Storing Web Server Content

```
#include "SPIFFS.h"

void setup() {
  Serial.begin(115200);
  if (!SPIFFS.begin(true)) {
    Serial.println("SPIFFS Mount Failed");
    return;
  }

  // Writing Wi-Fi credentials
  File file = SPIFFS.open("/wifi.txt", FILE_WRITE);
  if (file) {
    file.println("SSID: MyWiFi");
    file.println("Password: MyPassword");
    file.close();
    Serial.println("Wi-Fi credentials saved");
  }

  // Reading Wi-Fi credentials
  file = SPIFFS.open("/wifi.txt", FILE_READ);
  if (file) {
    Serial.println("Reading Wi-Fi credentials:");
    while (file.available()) {
      Serial.write(file.read());
    }
    file.close();
  }
}

void loop() {
  // Empty
}
```

```
#include "SPIFFS.h"

void setup() {
  Serial.begin(115200);
  if (!SPIFFS.begin(true)) {
    Serial.println("SPIFFS Mount Failed");
    return;
  }

  // Simulating sensor data
  int sensorValue = 42;

  // Logging data
  File logFile = SPIFFS.open("/log.txt", FILE_APPEND);
  if (logFile) {
    logFile.print("Sensor value: ");
    logFile.println(sensorValue);
    logFile.close();
    Serial.println("Data logged");
  }
}

void loop() {
  // Empty
}
```

```
#include <WiFi.h>
#include <WebServer.h>
#include "SPIFFS.h"

const char* ssid = "your_SSID";
const char* password = "your_PASSWORD";

WebServer server(80);

void handleRoot() {
  File file = SPIFFS.open("/index.html", FILE_READ);
  if (file) {
    server.streamFile(file, "text/html");
    file.close();
  } else {
    server.send(404, "text/plain", "File Not Found");
  }
}

void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  if (!SPIFFS.begin(true)) {
    Serial.println("SPIFFS Mount Failed");
    return;
  }

  server.on("/", handleRoot);
  server.begin();
}
```

```
void loop() {
  server.handleClient();
}
```

I2C (Inter-Integrated Circuit)

- Key Features of I2C:
  - Two-Wire synchronous communication: Only requires two lines, SDA (data line) and SCL (clock line).
  - Multi-Master and Multi-Slave: Multiple master and slave devices can share the same bus.
  - Addressing: Each device on the bus has a unique address, which allows the master to communicate with individual slaves.
  - Simple hardware implementation: Fewer pins are needed compared to other communication protocols like SPI.

I2C APIs and Library Functions

- Wire Library APIs.

  - `Wire.begin();`

    Initializes the I2C bus as a master or a slave. When used without parameters, it initializes the bus as a master.

  - `Wire.beginTransmission(address);`

    Begins a transmission to the I2C device with the specified address.

```
//Start communication with the device at address 0x76
Wire.beginTransmission(0x76);
```

I2C APIs and Library Functions

- `Wire.write(data);`

Sends data to the I2C device. Can send a byte or a string.

- `Wire.endTransmission()`

Sends the data and ends the transmission by releasing the bus.

- `Wire.requestFrom(address, quantity);`

Requests a specified number of bytes from the I2C device.

```
//Request 2 bytes from the device at address 0x76
Wire.requestFrom(0x76, 2);
```

- `Wire.read();`

Reads a byte of data from the I2C device.

```
/* I2C communication example */
#include <Wire.h>

void setup() {
  Wire.begin();
  Serial.begin(115200);
}

void loop() {
  Wire.beginTransmission(0x76); // Start communication with device at address 0x76
  Wire.write(0xF7);             // Write register address to read from
  Wire.endTransmission();       // End transmission

  Wire.requestFrom(0x76, 2);    // Request 2 bytes from the device
  while(Wire.available()) {
    char c = Wire.read();       // Read the bytes received
    Serial.print(c, HEX);       // Print the byte data in hexadecimal
  }

  delay(1000);
}
```

Example: Interfacing with an I2C Sensor BMP280

- Connections
  - SDA: Connect to GPIO 21 (default SDA pin on ESP32-S3).
  - SCL: Connect to GPIO 22 (default SCL pin on ESP32-S3).
  - VCC: Connect to 3.3V on the ESP32-S3.
  - GND: Connect to GND on the ESP32-S3.
- Library: Install Adafruit_BMP280

```
/* example interface with BMP280 by I2C */
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BMP280.h>

// Create an instance of the BMP280 sensor
Adafruit_BMP280 bmp;

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

  // Initialize I2C communication
  if (!bmp.begin(0x76)) {  // Address of the BMP280
    Serial.println("Could not find a valid BMP280 sensor, check wiring!");
    while (1);
  }

  Serial.println("BMP280 sensor initialized successfully.");
}

void loop() {
  // Read temperature and pressure values
  float temperature = bmp.readTemperature();
  float pressure = bmp.readPressure();

  // Display the values
  Serial.print("Temperature = ");
  Serial.print(temperature);
  Serial.println(" *C");

  Serial.print("Pressure = ");
  Serial.print(pressure / 100.0F);
  Serial.println(" hPa");

  delay(2000);  // Wait 2 seconds before the next reading
}
```

I2C applications

- Typical Use Cases of I2C:
  - Interfacing with Sensors: I2C is commonly used to interface with sensors like temperature sensors (e.g., BMP280), humidity sensors, accelerometers, and gyroscopes.
  - LCD Displays: I2C is often used to control character LCDs (like the 16x2 LCD) with an I2C adapter, reducing the number of pins needed.
  - EEPROM Memory: I2C EEPROM chips are used for storing non-volatile data.
  - Real-Time Clocks (RTC): Devices like the DS3231 RTC module use I2C for communicating time and date data.
  - Communication Between Microcontrollers: I2C can be used to establish communication between two microcontrollers in a master-slave configuration.