IoT Systems with ESP8266

Version 0.raw – (Early Development - Unrefined Draft)

Alimul H Khan

INDEPENDENTLY PUBLISHED BY THE AUTHOR

https://github.com/alimul-khan/CME466_students

First printing, March 2025

Contents

1. ESP with Arduino

1.1 Installing Arduino and Other Initial Setup

Step-by-Step Installation Guide

This guide walks you through the process of installing the Arduino IDE on a Windows machine.

Step 1: Agree to the License Agreement

Run the Arduino installer and accept the GNU Lesser General Public License. Click **I Agree** to proceed.

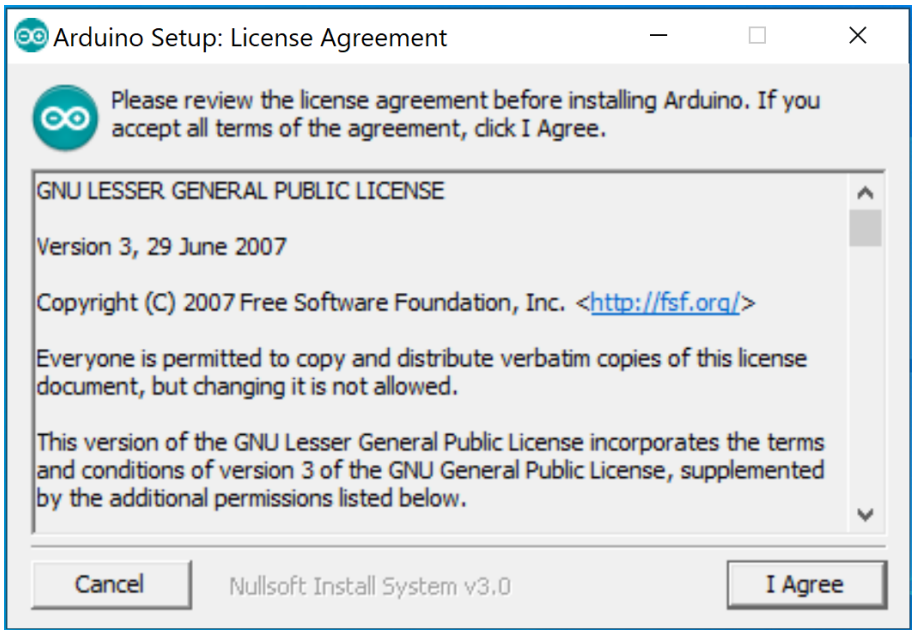

Figure 1.1: License Agreement Window

Step 2: Choose Installation Options

Select the components to install. Ensure the following options are checked:

- Install Arduino software
- Install USB driver
- Create Start Menu shortcut
- Create Desktop shortcut
- Associate .ino files

Click **Next** to continue.

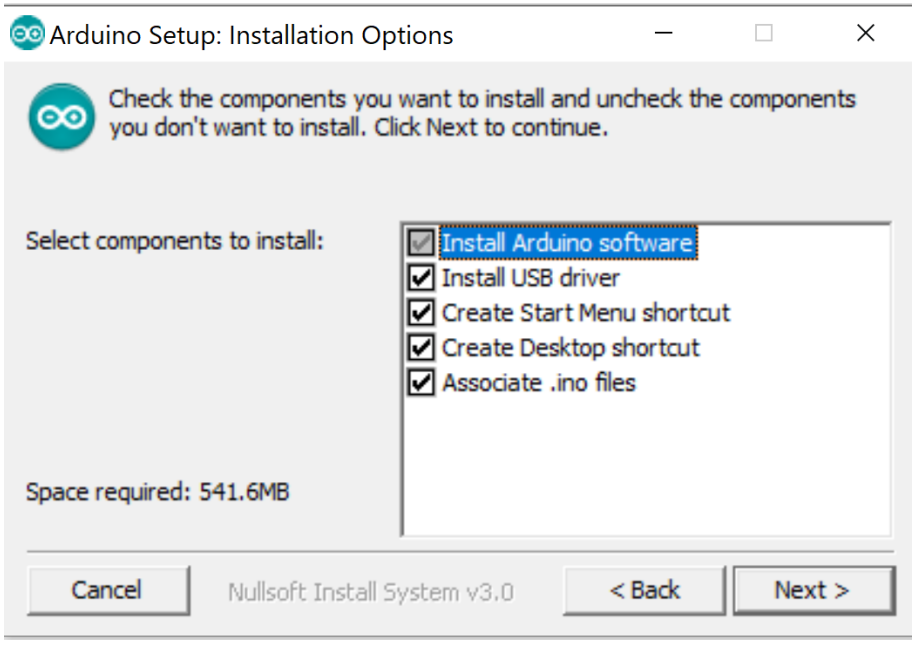

Figure 1.2: Installation Options Window

Step 3: Select Installation Folder

Choose the folder where Arduino will be installed. By default, it is installed in `C:\Program Files (x86)\Arduino`. Click **Install** to start the installation.

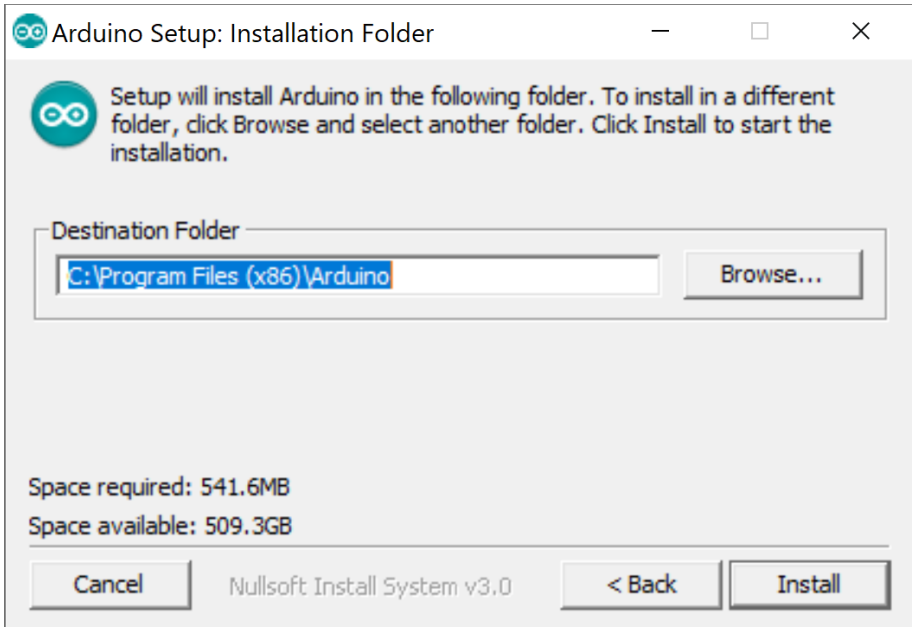

Figure 1.3: Installation Folder Selection

Step 4: Wait for Installation to Complete

The installer will copy all necessary files. This process may take a few minutes.

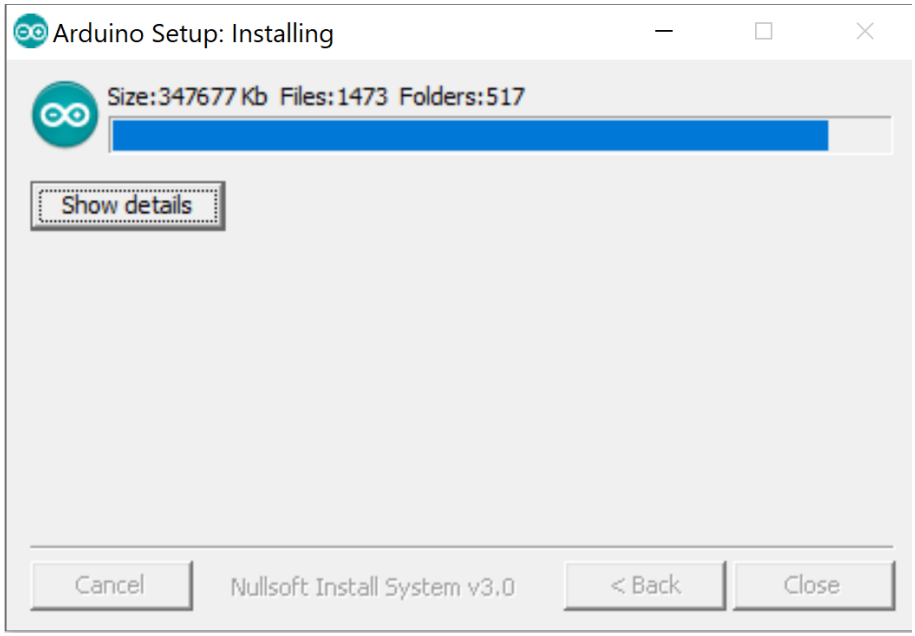

Figure 1.4: Installation in Progress

Step 5: Complete the Installation

Once the installation is complete, click **Close** to finish.

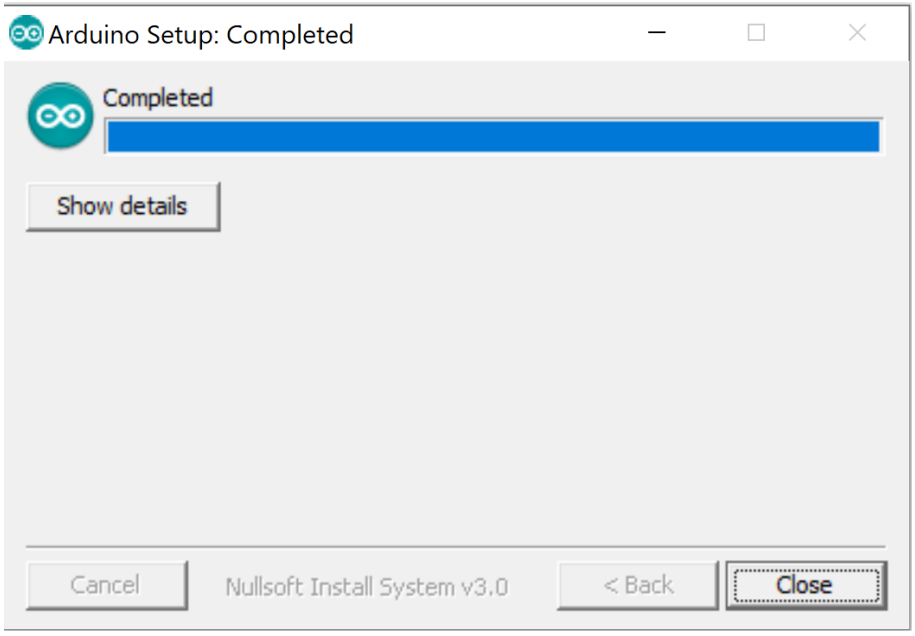

Figure 1.5: Installation Completed

You have successfully installed the Arduino IDE on your Windows machine. You can now start developing your Arduino projects!

1.2 Getting Started with ESP8266

Step 1: Open Arduino IDE

Launch the Arduino IDE to begin working with the ESP8266 module.

Figure 1.6: Opening the Arduino IDE

Step 2: Access Preferences

Navigate to **File > Preferences** in the Arduino IDE.

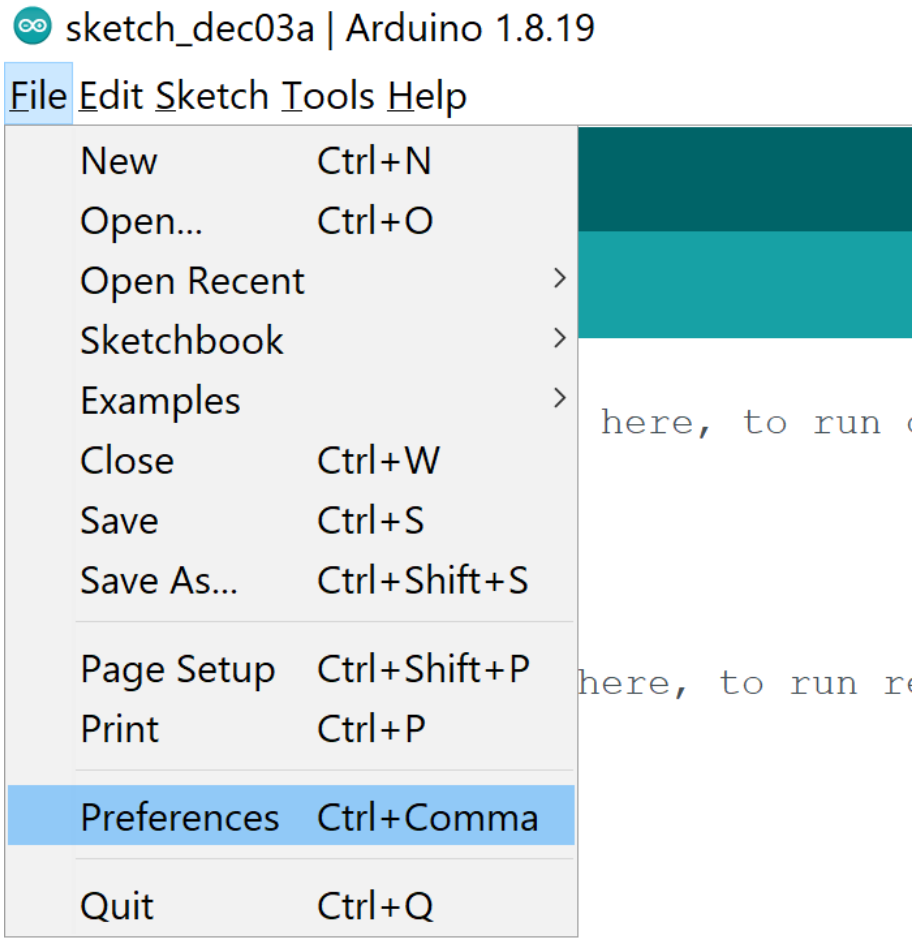

Figure 1.7: Accessing the Preferences menu in Arduino IDE

Step 3: Add Board Manager URLs

In the Preferences dialog, add the following URLs to the **Additional Board Manager URLs** field:

```
https://dl.espressif.com/dl/package_esp32_index.json,
http://arduino.esp8266.com/stable/package_esp8266com_index.json
```

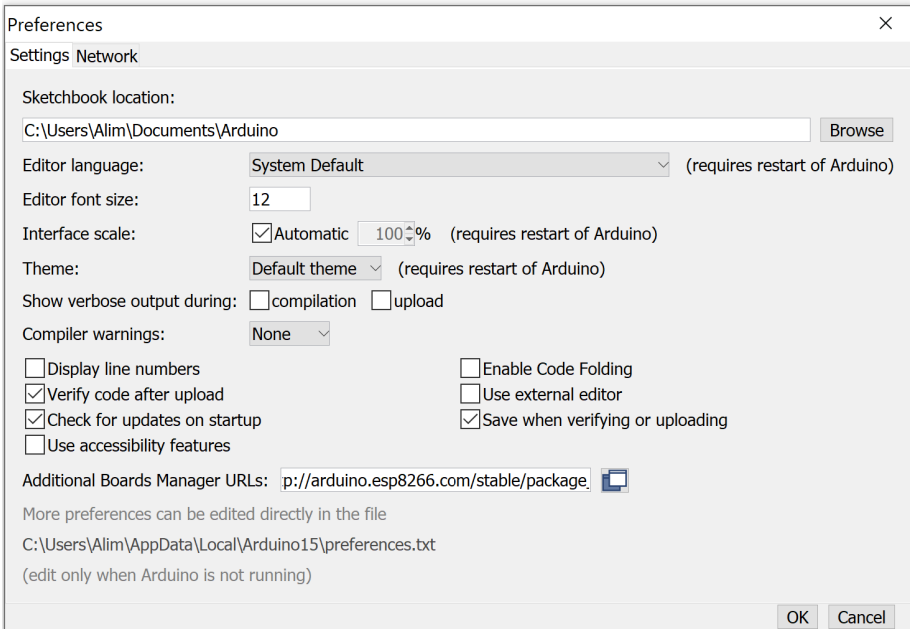

Figure 1.8: Adding Additional Board Manager URLs in Preferences

Step 4: Open Boards Manager

Navigate to **Tools > Board > Boards Manager...** in the Arduino IDE to access and install board packages, such as the ESP8266 package needed for the Wemos D1 Mini.

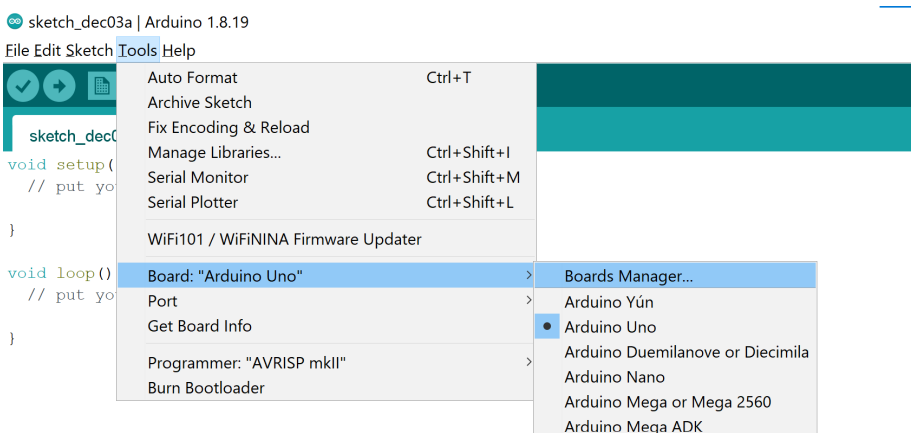

Figure 1.9: Opening Boards Manager in Arduino IDE

Step 5: Search for ESP8266

In the Boards Manager search bar, type `esp8266` to find the ESP8266 package.

Figure 1.10: Searching for ESP8266 in Boards Manager

Step 6: Install ESP8266 Package

Select the **ESP8266 by ESP8266 Community** package and click **Install**.

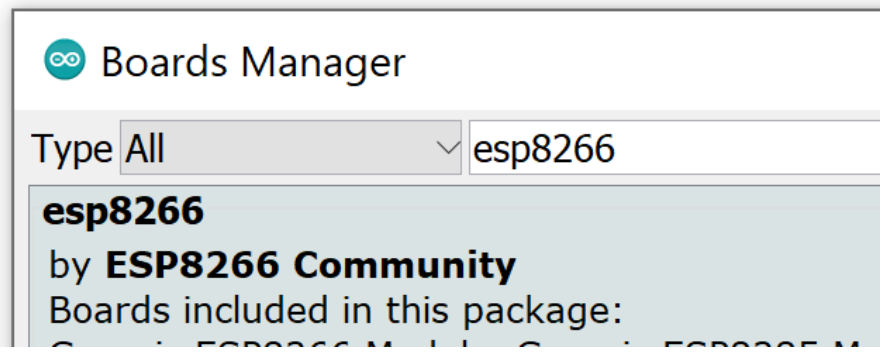

Figure 1.11: Selecting ESP8266 package by ESP8266 Community

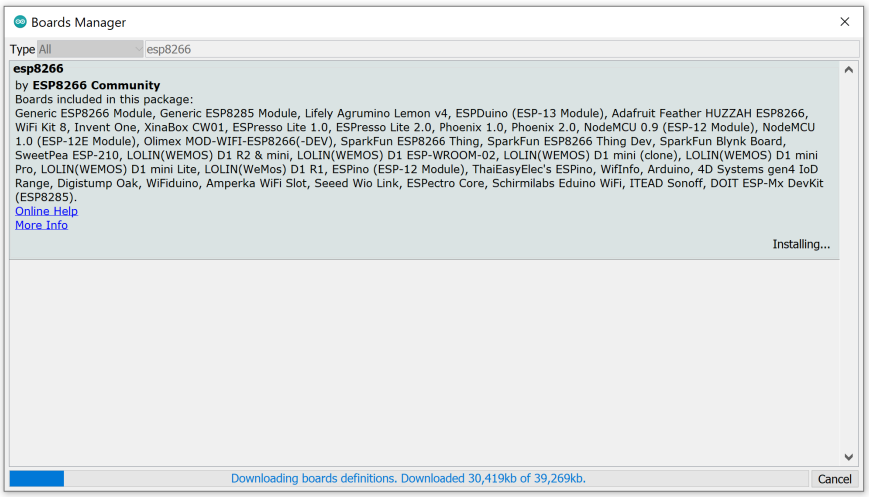

Figure 1.12: Installing the ESP8266 package

1.3 First code with Node MCU

Step 1: Select ESP8266 Board

From the **Tools > Board** menu, select your ESP8266 board (e.g., LOLIN(WEMOS) D1 mini).

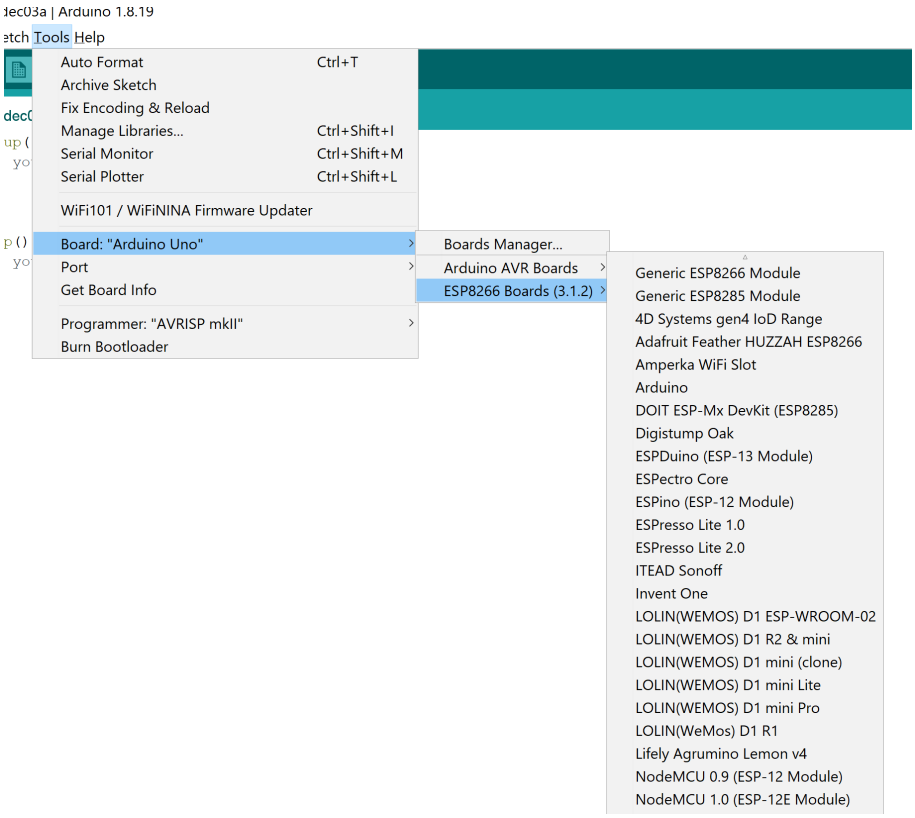

Figure 1.13: Selecting an ESP8266 board from the list

Step 2: Select Connected Port

From the **Tools > Port** menu, select the port connected with ESP8266 board (e.g., COM3 or COM4).

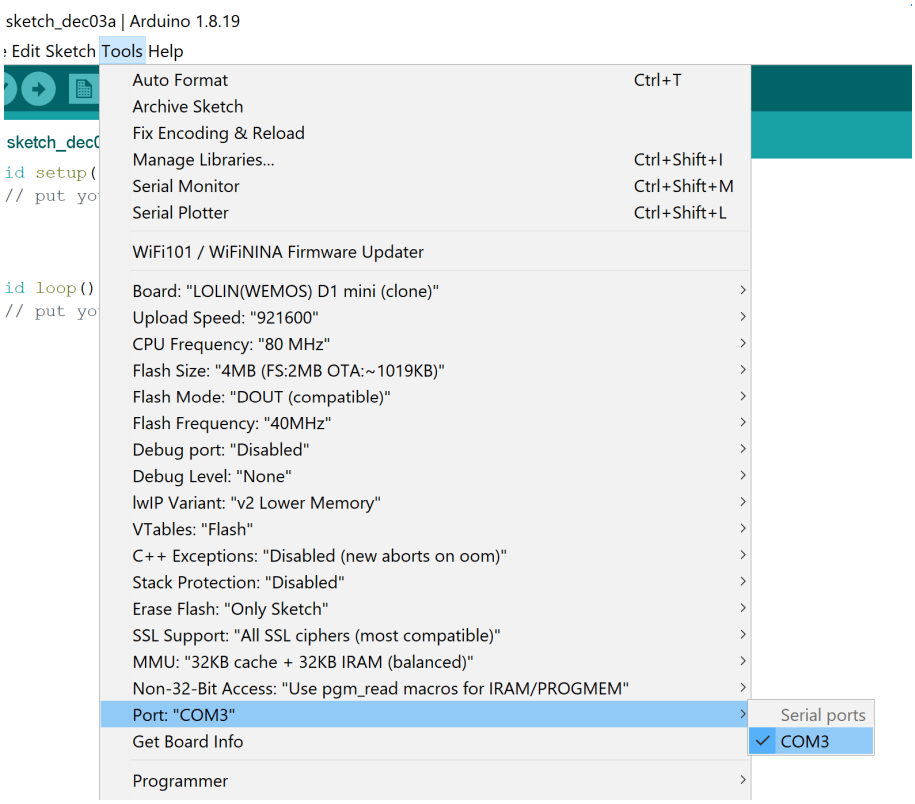

Figure 1.14: Configuring ESP8266 board details and COM port

Step 3: Load Blink Example

Open the **Blink** example from **File > Examples > 01.Basics > Blink**.

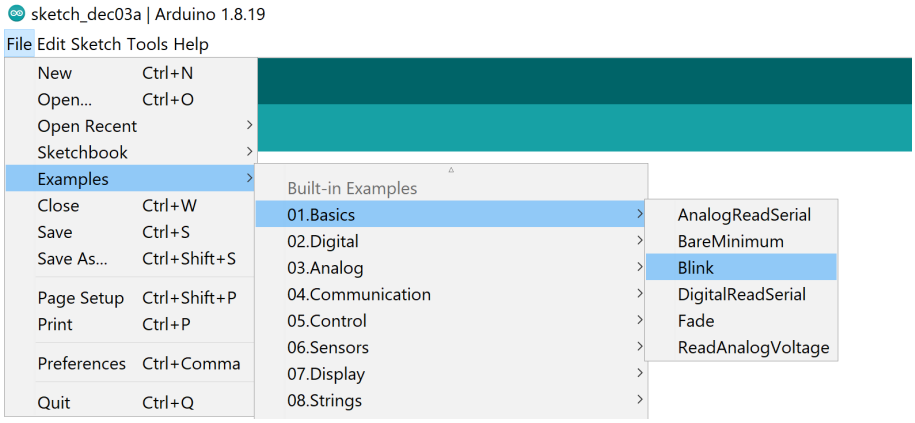

Figure 1.15: Selecting Blink example from the Examples menu

Step 4: Upload Blink Sketch

Compile and upload the Blink sketch to your ESP8266. If you encounter an error, follow the next step.

Figure 1.16: Blink sketch loaded in Arduino IDE

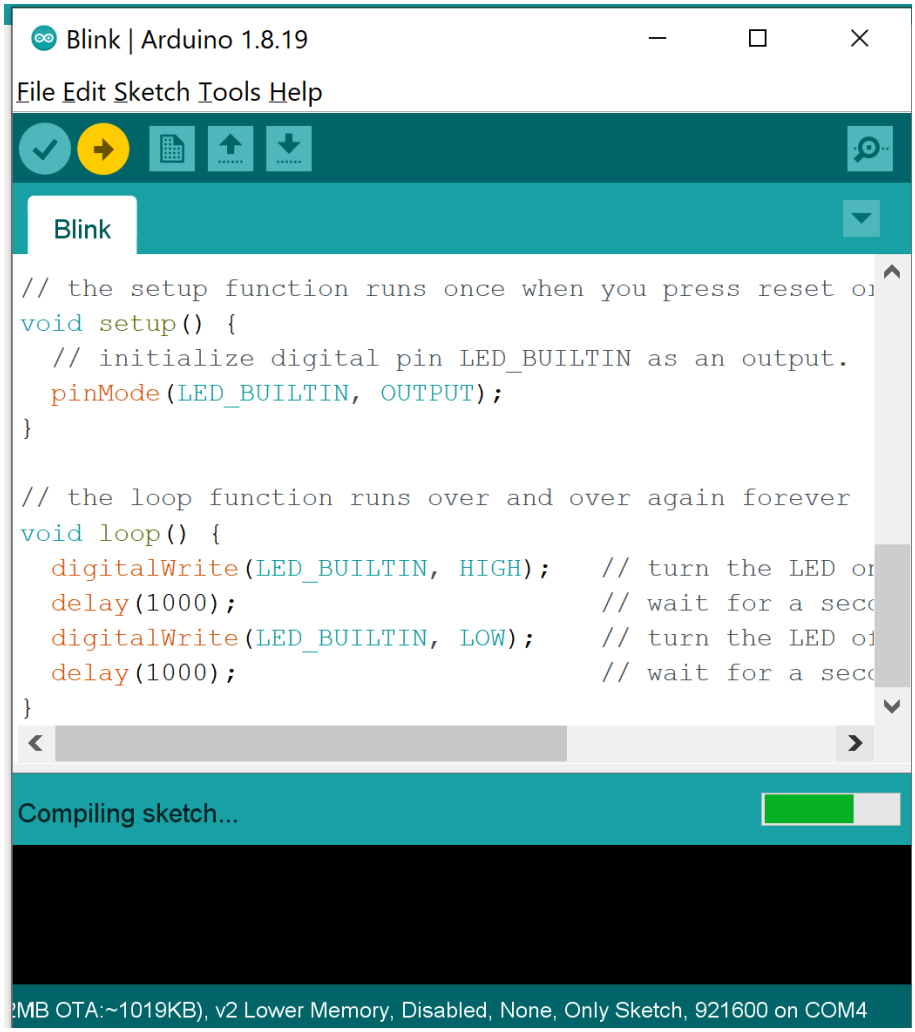

Figure 1.17: Compiling and uploading Blink sketch to ESP8266

Step 5: Verify Successful Upload

Once the program is successfully uploaded, the onboard LED should blink. Modify the delays in the code to observe different blink patterns.

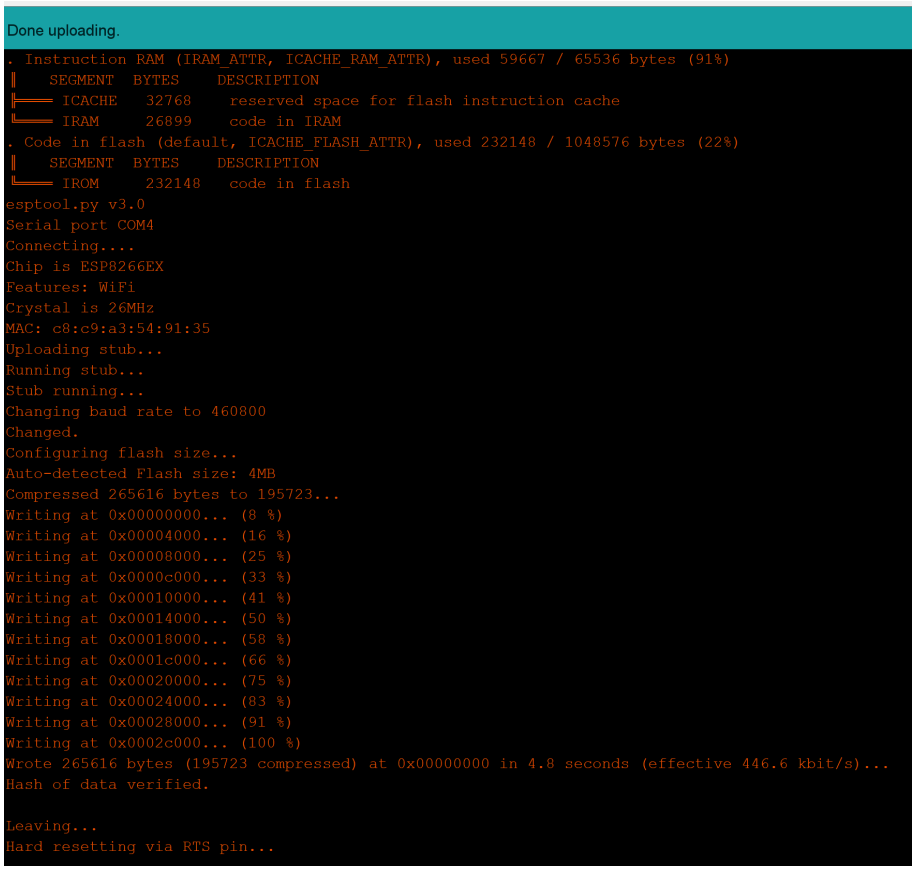

Figure 1.18: Successful Blink sketch upload with onboard LED blinking

Step 6: Install CH340 Driver

You may face errors releated with permission error and or esptool.py. In that case, download and install the CH340 driver from `https://sparks.gogo.co.nz/ch340.html`. Restart the Arduino IDE after installation.

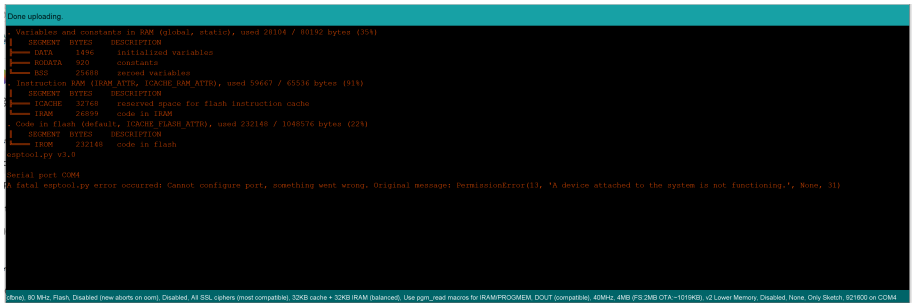

Figure 1.19: Error encountered during Blink sketch upload

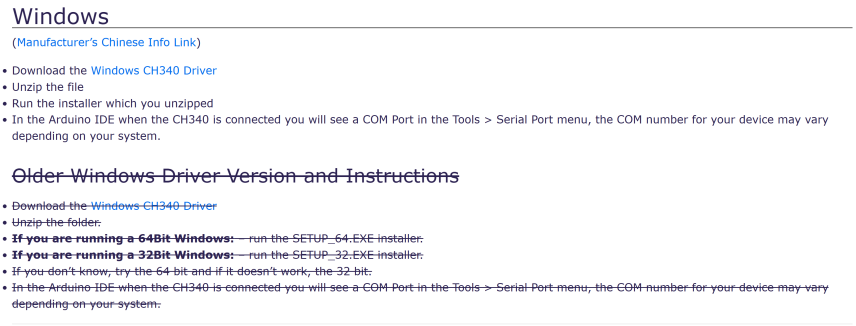

Figure 1.20: CH340 driver installation for USB-to-serial communication

1.4 Including Missing Libraries

To include the necessary libraries for your project, follow the steps below:

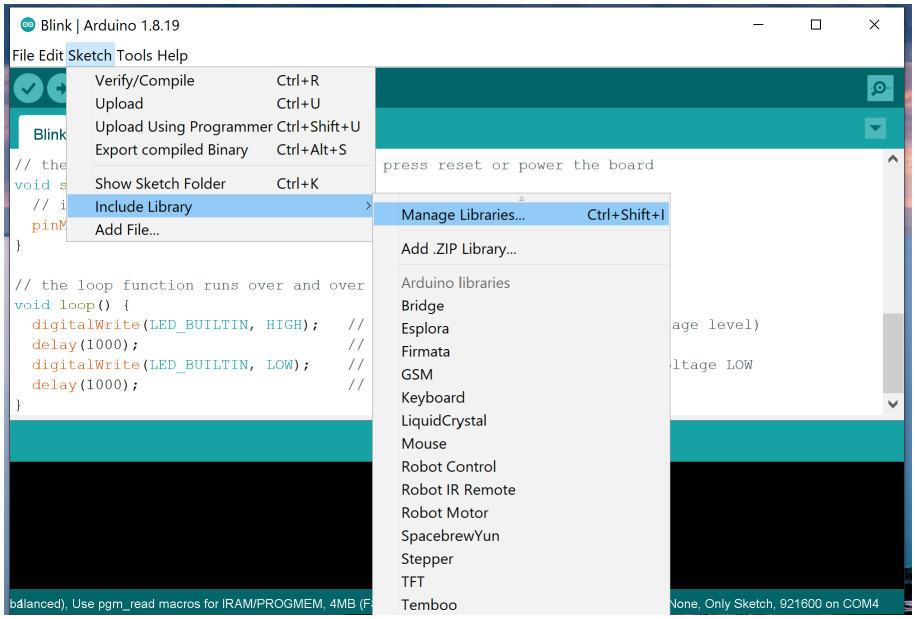

Figure 1.21: Navigating to Include Libraries in Arduino IDE

Open the Arduino IDE and navigate to 'Sketch > Include Library > Manage Libraries'. This opens the Library Manager where you can search for the required library.

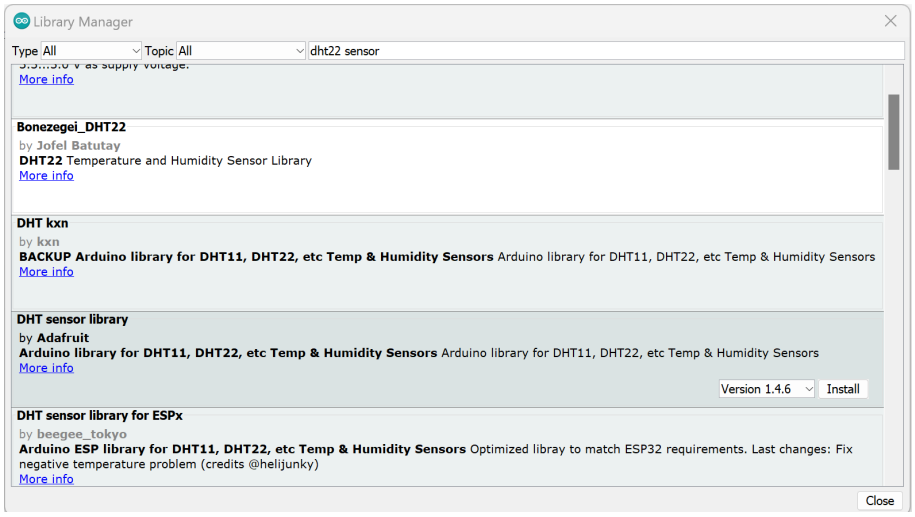

Figure 1.22: Searching for Desired Library in Library Manager

Once the library is installed, you can verify its status in the Library Manager.

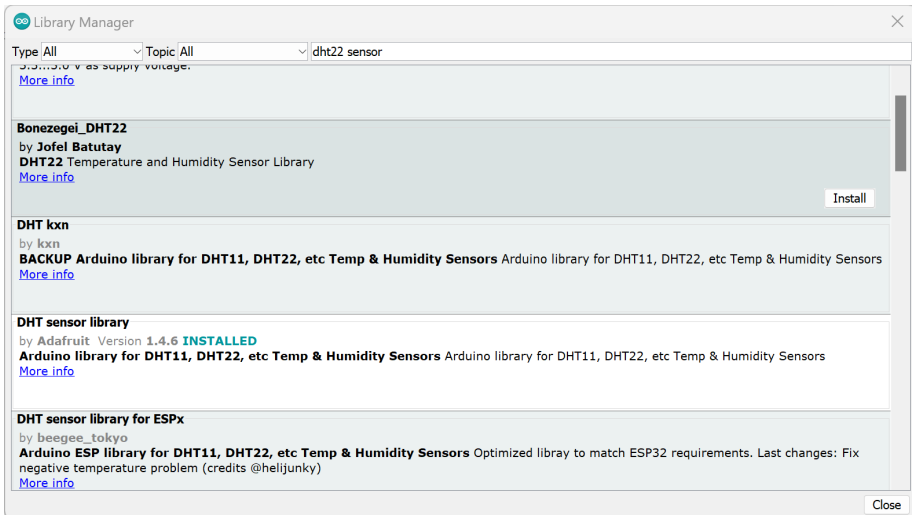

Figure 1.23: Library Installation Status in Library Manager

Alternatively, you can download the library from a valid source and place it in the Arduino **libraries** folder. Check the Arduino libraries folder to confirm that the library has been successfully installed.

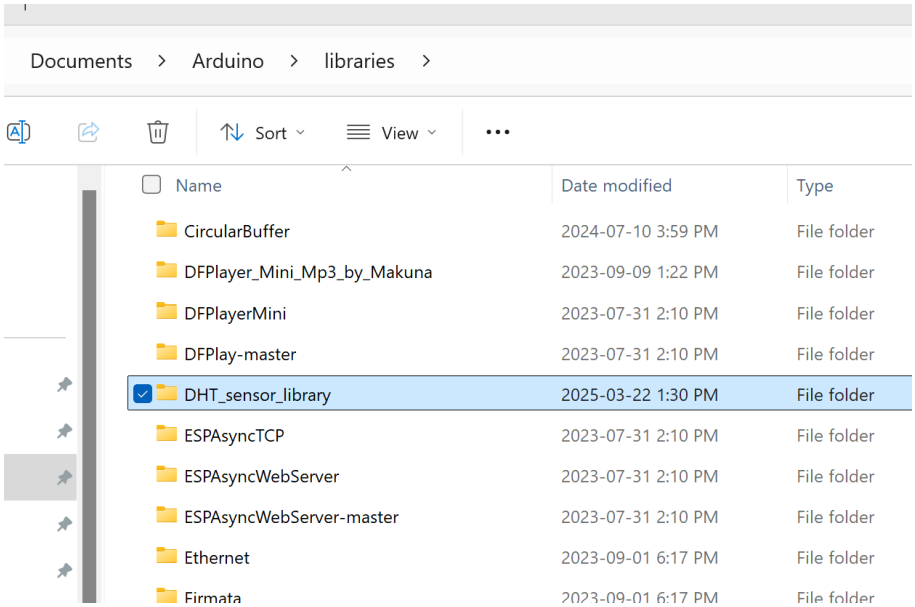

Figure 1.24: Arduino Library Folder View

2. Reading Data from DHT22 using ESP8266

The DHT22 sensor is widely used for measuring temperature and humidity in embedded projects. This chapter explains how to interface the DHT22 sensor with an ESP8266 module and read the data using Arduino code.

2.1 Hardware Requirements

- ESP8266 (e.g., NodeMCU)
- DHT22 Temperature and Humidity Sensor
- Jumper wires
- Breadboard (optional)

2.2 Pin Configuration: ESP8266 WeMos D1 Mini

The WeMos D1 Mini is a compact development board based on the ESP8266 chip, offering multiple GPIOs and integrated Wi-Fi. The board features labeled pins for easy prototyping, as shown below.

Pin Descriptions

- **GPIO0 – D3**: Often used for flashing or input; can also be used for sensors.
- **GPIO1 – TX**: UART transmit pin. Used for Serial communication.
- **GPIO2 – D4**: Can be used as digital I/O. Connected to onboard LED.
- **GPIO3 – RX**: UART receive pin.
- **GPIO4 – D2**: General-purpose I/O; often used for I2C (SDA).
- **GPIO5 – D1**: General-purpose I/O; often used for I2C (SCL).
- **GPIO12 – D6**, **GPIO13 – D7**, **GPIO14 – D5**, **GPIO15 – D8**: SPI-related pins (MISO, MOSI, SCLK, CS).
- **GPIO16 – D0**: Used for deep sleep wakeup.
- **ADC0**: Analog input (0–1V range).
- **3.3V and 5V**: Power supply pins. Provide 3.3V regulated output or take 5V input.
- **GND**: Ground connection.

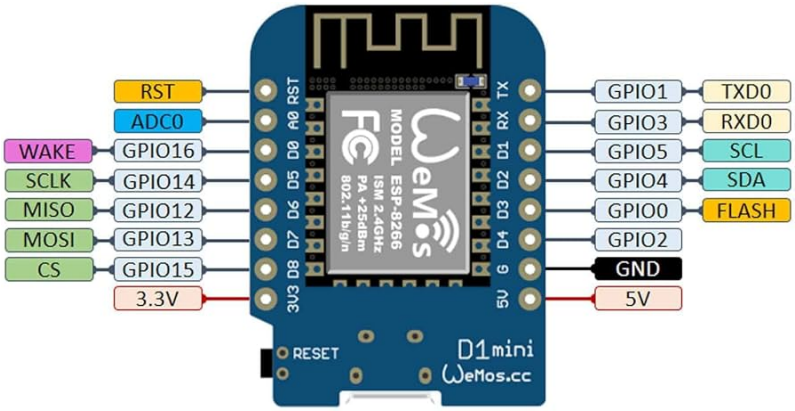

Figure 2.1: Pinout diagram of WeMos D1 Mini (ESP8266)

- **RST**: Used to reset the module.

Notes

- Many GPIOs are multifunctional and may have boot constraints (e.g., GPIO0, GPIO2, GPIO15).
- Always refer to boot mode requirements when using GPIO0, GPIO2, or GPIO15 in a project.
- Avoid using TX/RX pins for general I/O if Serial communication is active.

2.3 Wiring Diagram

DHT22 sensor provides digital temperature and humidity readings. It has three pins: GND (Ground), VCC (Power Supply), and DAT (Data Signal). It operates with 3.3V to 5V and outputs calibrated digital signals

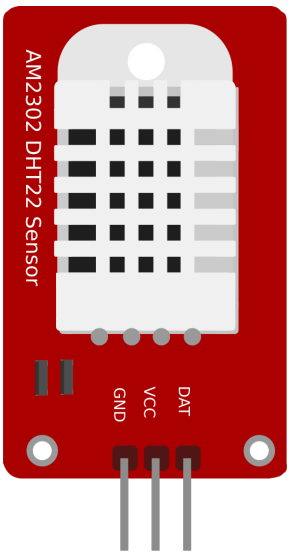

Figure 2.2: DHT22 Digital Sensor..

Connect the DHT22 sensor to the ESP8266 as follows:

- **VCC** → 3.3V on ESP8266
- **GND** → GND on ESP8266

- **DATA** → D3 (GPIO0) on ESP8266

2.4 Required Libraries

To communicate with the DHT22 sensor, install the following libraries from the Arduino Library Manager:

- `Adafruit Unified Sensor`
- `DHT sensor library by Adafruit`

2.5 Arduino Code

The code below reads temperature and humidity values from the DHT22 sensor:

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

#define DHTPIN D3
#define DHTTYPE DHT22

DHT dht(DHTPIN, DHTTYPE);

float humidity = 0.0;
float temperature = 0.0;
float temperatureF = 0.0;

void getTempHumidity();

void getTempHumidity() {
  float h = dht.readHumidity();
  float t = dht.readTemperature();
  float tF = dht.readTemperature(true);
  delay(1000);

  if (isnan(h) || isnan(t) || isnan(tF)) {
    Serial.println("Failed to read from DHT sensor! Using default values.")
        ;
    humidity = 5.0;
    temperature = 0.0;
    temperatureF = 32.0;
    return;
  }

  humidity = h;
  temperature = t;
  temperatureF = tF;
}

void setup() {
  delay(200);
  Serial.begin(115200);
  dht.begin();
  Serial.println("DHT22 sensor initialized");
}

void loop() {
  getTempHumidity();

  Serial.print("Humidity: ");
  Serial.print(humidity);
```

```
  Serial.print("%\t");

  Serial.print("Temperature: ");
  Serial.print(temperature);
  Serial.print("deg C / ");
  Serial.print(temperatureF);
  Serial.println("deg F");

  delay(2000);
}
```

Listing 2.1: Reading from DHT22 using ESP8266

2.6 Code Walkthrough and Explanation

This section provides a detailed explanation of each part of the code used to read data from the DHT22 sensor using ESP8266.

2.6.1 Importing Required Libraries

```
// Include required libraries
#include <Adafruit_Sensor.h>
#include <DHT.h>
```

Explanation: These two libraries provide the functionality to interact with the DHT22 sensor. `Adafruit_Sensor.h` is a unified sensor library, and `DHT.h` handles DHT-specific communication.

2.6.2 Defining Pin and Sensor Type

```
// Define DHT22 sensor pin
#define DHTPIN D3
#define DHTTYPE DHT22

DHT dht(DHTPIN, DHTTYPE);
```

Explanation: We define the digital pin (D3) to which the DHT22 is connected. 'DHTTYPE' specifies that the sensor model is DHT22. An object 'dht' is created from the DHT class for communication.

2.6.3 Declaring Global Variables

```
// Global variables to hold values
float humidity = 0.0;
float temperature = 0.0;
float temperatureF = 0.0;
```

Explanation: Three global float variables are declared to store humidity, temperature in Celsius, and temperature in Fahrenheit, respectively.

2.6.4 Function Prototype Declaration

```
// Function prototype
void getTempHumidity();
```

Explanation: A forward declaration of the function 'getTempHumidity()' ensures that it can be used before its definition later in the code.

2.6.5 Reading Temperature and Humidity

```
void getTempHumidity() {
  float h = dht.readHumidity();
  float t = dht.readTemperature();
  float tF = dht.readTemperature(true);
  delay(1000);

  if (isnan(h) || isnan(t) || isnan(tF)) {
    Serial.println("Failed to read from DHT sensor! Using default values.")
        ;
    humidity = 5.0;
    temperature = 0.0;
    temperatureF = 32.0;
    return;
  }

  humidity = h;
  temperature = t;
  temperatureF = tF;
}
```

Explanation: This function reads humidity and temperature (in both Celsius and Fahrenheit). If the sensor returns invalid data (‘NaN‘), fallback values are assigned. The readings are stored in the global variables for further use.

2.6.6 Setup Function

```
void setup() {
  delay(200);
  Serial.begin(115200);
  dht.begin();
  Serial.println("DHT22 sensor initialized");
}
```

Explanation: This runs once when the ESP8266 starts. It initializes the serial monitor for debugging and starts communication with the DHT sensor using ‘dht.begin()‘.

2.6.7 Main Loop

```
void loop() {
  getTempHumidity();

  Serial.print("Humidity: ");
  Serial.print(humidity);
  Serial.print("%\t");

  Serial.print("Temperaturre: ");
  Serial.print(temperature);
  Serial.print(" deg C / ");
  Serial.print(temperatureF);
  Serial.println("deg F");

  delay(2000);
}
```

Explanation: The ‘loop()‘ function repeatedly calls ‘getTempHumidity()‘ to update readings. It then prints the values to the serial monitor every 2 seconds. The delay helps to avoid flooding the serial output and reduces sensor polling frequency.

2.7 Expected Output on Serial Monitor

When the code is uploaded to the ESP8266 and the serial monitor is opened at a baud rate of 115200, you can expect an output similar to the following:

```
DHT22 sensor initialized
Humidity: 51.70% Temperature: 25.10 deg C / 77.18 deg F
Humidity: 51.70% Temperature: 25.20 deg C / 77.36 deg F
Humidity: 51.60% Temperature: 25.20 deg C / 77.36 deg F
Humidity: 51.50% Temperature: 25.30 deg C / 77.54 deg F
```

Note: If the DHT22 fails to respond or is not properly connected, you may see:

```
Failed to read from DHT sensor! Using default values.
Humidity: 5.00% Temperature: 0.00 deg C / 32.00 deg F
```

2.8 Conclusion

Using the DHT22 sensor with the ESP8266 is straightforward with the help of the Adafruit libraries. This setup allows for reliable monitoring of temperature and humidity in IoT applications.

3. ESP8266 as an Access Point

This chapter explains how to set up the ESP8266 as a Wi-Fi Access Point (AP) to serve real-time temperature and humidity data collected via a DHT22 sensor. Users can connect directly to the ESP8266 and view live environmental data through a browser interface.

3.1 Project Structure

The folder contains the following files:

```
espAccessPoint/
|- espAccessPoint.ino     % Main entry file
|- myDHT.h                % Header for DHT sensor functions
|- myDHT.cpp              % Implementation of DHT sensor reading
|- myServer.h             % Header for server setup functions
|- myServer.cpp           % Server routes and connection handling
|- serverHTML.h           % Embedded HTML and JavaScript for web UI
```

3.2 Main Entry: `espAccessPoint.ino`

This file contains only essential calls and is small enough to describe as a whole:

- Calls `initDHT()` to start DHT22.
- Calls `initServer()` to launch AP and HTTP server.
- Calls `getTempHumidity()` to update readings.
- Calls `updateServer()` to handle HTTP requests and new client logs.

3.3 Sensor Logic: `myDHT.h` and `myDHT.cpp`

We will explain this file in two parts.

3.3.1 myDHT.h

This header contains:

```
#ifndef MYDH_H
#define MYDHT_H

#include <Arduino.h>

extern float humidity;
extern float temperature;
extern float temperatureF;

void initDHT();
void getTempHumidity();

#endif
```

Listing 3.1: myDHT.h

Explanation:

- Declares global variables for humidity and temperature.
- Declares the function prototypes used to interface with the DHT22 sensor.

3.3.2 myDHT.cpp

This file contains the actual implementation.

Initialization Block:

```
#include "myDHT.h"
#include <Adafruit_Sensor.h>
#include <DHT.h>

#define DHTPIN D3
#define DHTTYPE DHT22

DHT dht(DHTPIN, DHTTYPE);

float humidity = 0.0;
float temperature = 0.0;
float temperatureF = 0.0;

void initDHT() {
  dht.begin();
}
```

Explanation:

- DHT is configured on pin D3.
- Global sensor variables are initialized.
- `initDHT()` starts the sensor.

Sensor Reading Block:

```
void getTempHumidity() {
  float h = dht.readHumidity();
  float t = dht.readTemperature();
  float tF = dht.readTemperature(true);
  delay(1000);

  if (isnan(h) || isnan(t) || isnan(tF)) {
    Serial.println("Failed to read from DHT sensor! Using default values.")
        ;
```

```
    humidity = 5.0;
    temperature = 0.0;
    temperatureF = 32.0;
    return;
  }

  humidity = h;
  temperature = t;
  temperatureF = tF;
}
```

Explanation:
- Reads humidity and temperature.
- Falls back to defaults on sensor failure.

3.4 Server Logic: myServer.h and myServer.cpp

This is a larger module and explained in blocks.

3.4.1 myServer.h

```
#ifndef MYSERVER_H
#define MYSERVER_H

#include <Arduino.h>

void initServer();
void updateServer();

#endif
```

Listing 3.2: myServer.h

Explanation: Declares functions for server initialization and loop-time updates.

3.4.2 myServer.cpp

This file defines Wi-Fi setup, HTTP routes, and client logging.

HTML Routes Block:

```
void handleRoot() {
  String html = FPSTR(HTML_HEAD);
  html += FPSTR(HTML_SCRIPT);
  server.send(200, "text/html", html);
}

void handleData() {
  String json = "{";
  json += "\"humidity\":" + String(humidity, 2) + ",";
  json += "\"temperatureC\":" + String(temperature, 2) + ",";
  json += "\"temperatureF\":" + String(temperatureF, 2);
  json += "}";
  server.send(200, "application/json", json);
}
```

Explanation:
- handleRoot() sends the HTML dashboard.
- handleData() returns current sensor values in JSON format.

Client Logging Block:

```
void checkNewConnections() {
  struct station_info *stat_info = wifi_softap_get_station_info();
  int i = 0;

  while (stat_info != NULL) {
    IPAddress ip = IPAddress(stat_info->ip.addr);
    bool isNew = true;

    for (int j = 0; j < 5; j++) {
      if (ip == lastConnectedIPs[j]) {
        isNew = false;
        break;
      }
    }

    if (isNew) {
      Serial.println("\n New device connected:");
      Serial.print("IP Address: ");
      Serial.println(ip);
      Serial.print("MAC Address: ");
      for (int k = 0; k < 6; k++) {
        Serial.print(stat_info->bssid[k], HEX);
        if (k < 5) Serial.print(":");
      }
      Serial.println();

      lastConnectedIPs[i % 5] = ip;
      i++;
    }

    stat_info = STAILQ_NEXT(stat_info, next);
  }

  wifi_softap_free_station_info();
}
```

Explanation: Prints new client IP and MAC when they connect to ESP AP. Keeps the last 5 IPs to avoid duplicates.

Server Initialization Block:

```
void initServer() {
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi ");
    while (WiFi.status() != WL_CONNECTED) {
        delay(1000);
        Serial.print(".");
    }
    Serial.print("\nConnected to WiFi");
    Serial.println(ssid);

    Serial.print("ESP IP address: ");
    Serial.println(WiFi.localIP());
    Serial.print("Visit http://");
    Serial.print(WiFi.localIP());
    Serial.println(" in your phone to view data.");

    server.on("/", handleRoot);
    server.on("/data", handleData);
    server.begin();
    Serial.println("HTTP server started");
```

```
}
```

Explanation: Connects to Wi-Fi, prints IP, and registers server routes.

Loop Handler Block:

```
void updateServer() {
  server.handleClient();
  checkNewConnections();
}
```

Explanation: Serves incoming HTTP requests and logs new clients continuously.

3.5 Web Dashboard: serverHTML.h

This file combines HTML, CSS, and JS.

Dashboard UI Code:

```
const char HTML_HEAD[] PROGMEM = R"rawliteral(
  ...
  <div class="card">
    <h1>ESP8266 Weather</h1>
    <div class="value" id="humidity">--</div>
    <div class="label">Humidity (%)</div>
    <div class="value" id="temperatureC">--</div>
    <div class="label">Temperature ( deg C)</div>
    <div class="value" id="temperatureF">--</div>
    <div class="label">Temperature (deg F)</div>
  </div>
...
```

JS Fetch Code:

```
<script>
  function fetchData() {
    fetch("/data")
      .then(response => response.json())
      .then(data => {
        document.getElementById("humidity").textContent = data.humidity.
            toFixed(2);
        document.getElementById("temperatureC").textContent = data.
            temperatureC.toFixed(2);
        document.getElementById("temperatureF").textContent = data.
            temperatureF.toFixed(2);
      });
  }
  setInterval(fetchData, 2000);
  fetchData();
</script>
```

Explanation: The UI updates sensor readings every 2 seconds by calling the ‘/data‘ endpoint.

3.6 Expected Serial Monitor Output

```
Connecting to WiFi ............
Connected to WiFiSHAW-A0A1
ESP IP address: 192.168.1.103
Visit http://192.168.1.103 in your phone to view data.
HTTP server started
```

```
New device connected:
IP Address: 192.168.1.109
MAC Address: 18:FE:34:A1:2B:3C
```

3.7 Outcome in Browser

Connecting to the printed IP shows a weather dashboard:

- Humidity in %
- Temperature in degC and degF
- Updates every 2 seconds

4. ESP8266 - Cloud Server

In this chapter, we demonstrate how to make the ESP8266 interact with a remote server using HTTP requests. A Flask server handles communication, stores data, and presents it on a user interface. This is particularly useful for scalable applications where data from multiple nodes is collected centrally.

4.1 Project Folder Structure

The project is split into two parts: the firmware on the ESP8266 and the Flask-based web server.

```
espCloud/                    |- Arduino firmware
|- espCloud.ino              |- Main program to interact with the remote server
|- SensorData.cpp            |- Generates random temperature/humidity values
|- SensorData.h              |- Function declarations

espCloudApp/                 |- Flask server application
|- espCloudApp.py            |- Main server logic
|- get_local_ip.py           |- Utility script to get the server's local IP
|- static/
|   |- style.css             |- Styling for the HTML dashboard
|- templates/
   |- node.html              |- HTML page served per ESP node
```

4.2 Arduino Code for Remote Server Communication

This section details how the ESP8266 collects temperature and humidity data, connects to a Wi-Fi network, and interacts with a remote server to send sensor values upon request.

4.2.1 `SensorData.h`

This header file contains the function prototypes for generating simulated sensor data.

```
#ifndef SENSORDATA_H
#define SENSORDATA_H

float generateTemperature();
float generateHumidity();

#endif
```

Listing 4.1: SensorData.h

Explanation: The header defines two functions: `generateTemperature()` and `generateHumidity()`, which are implemented in the corresponding `SensorData.cpp` file. These are used to simulate temperature and humidity readings.

4.2.2 SensorData.cpp

```
#include "SensorData.h"
#include <cstdlib>

float generateTemperature() {
    return 10.00 + (rand() % 4300) / 100.0; // Random temp (10.00 - 53.00)
}

float generateHumidity() {
    return 20.00 + (rand() % 6100) / 100.0; // Random humidity (20.00 -
        81.00)
}
```

Listing 4.2: SensorData.cpp

Explanation:

- `rand() % 4300` produces values from 0 to 4299. Dividing by 100.0 gives 0.00 to 42.99, then adding 10.00 makes the temperature range 10.00 to 52.99 degC.
- Similarly, humidity ranges from 20.00 to 80.99%.

These functions simulate realistic sensor data for testing without actual hardware.

4.2.3 espCloud.ino

This is the main Arduino sketch for ESP8266.

```
#include <ESP8266WiFi.h>
#include <ESP8266HTTPClient.h>
#include "SensorData.h"

const char* ssid = "YourWiFiSSID";
const char* password = "YourWiFiPassword";
const char* server = "http://192.168.1.100:5000";

String chip_id;

void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);
  Serial.print("Connecting to WiFi");

  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.print(".");
  }
```

```
20
21   Serial.println("\nConnected to WiFi");
22   chip_id = String(ESP.getChipId());
23   Serial.println("Chip ID: " + chip_id);
24 }
```

Listing 4.3: espCloud.ino – Includes and Setup

Explanation:

- Wi-Fi credentials are set.
- ESP connects to the local network.
- A unique ID is generated using `ESP.getChipId()`, used to identify the node on the Flask server.

```
1  void sendJSON(String endpoint, String key, float value) {
2    if (WiFi.status() == WL_CONNECTED) {
3      HTTPClient http;
4      String url = server + endpoint + "?chip_id=" + chip_id;
5      http.begin(url);
6      http.addHeader("Content-Type", "application/json");
7
8      String payload = "{\"" + key + "\":" + String(value, 2) + "}";
9      int httpResponseCode = http.POST(payload);
10
11     Serial.print("POST to ");
12     Serial.print(endpoint);
13     Serial.print(": ");
14     Serial.println(httpResponseCode);
15
16     http.end();
17   }
18 }
```

Listing 4.4: espCloud.ino – Helper Function to Send JSON

Explanation:

- This function sends a POST request with JSON payload to the given `endpoint`.
- `key` is either `"temperature"` or `"humidity"`.
- Example payload: `{"temperature": 26.37}`.

```
1  void loop() {
2    if (WiFi.status() == WL_CONNECTED) {
3      HTTPClient http;
4      String url = server + "/command?chip_id=" + chip_id;
5      http.begin(url);
6      int httpResponseCode = http.GET();
7
8      if (httpResponseCode == 200) {
9        String response = http.getString();
10       Serial.println("Server command: " + response);
11
12       if (response.indexOf("get_temp") > 0) {
13         float temp = generateTemperature();
14         Serial.println("Sending temperature: " + String(temp));
15         sendJSON("/send_temp", "temperature", temp);
16       } else if (response.indexOf("get_humidity") > 0) {
17         float hum = generateHumidity();
18         Serial.println("Sending humidity: " + String(hum));
19         sendJSON("/send_humidity", "humidity", hum);
20       }
21     }
```

```
22
23      http.end();
24    }
25
26    delay(2000); // Wait before checking again
27  }
```

Listing 4.5: espCloud.ino – Loop Function

Explanation:

- The ESP queries the server for a command.
- If the response includes `get_temp`, it generates and sends temperature data.
- If the response includes `get_humidity`, it generates and sends humidity data.
- After each request, it waits 2 seconds before polling again.

4.2.4 Expected Serial Monitor Output

```
Connecting to WiFi...
Connected to WiFi
Chip ID: ESP123456
Server command: {"command":"get_temp"}
Sending temperature: 26.47
POST to /send_temp: 200
Server command: {"command":"get_humidity"}
Sending humidity: 61.32
POST to /send_humidity: 200
```

4.3 Python Flask Server and Web Dashboard

This section explains the Python-based server which receives sensor data from the ESP8266, stores it, and provides a web interface to interact with each ESP node.

4.3.1 `espCloudApp.py`

This is the main Flask server script that:

- Manages command dispatch and data reception from ESPs
- Serves dynamic HTML pages for each node
- Offers endpoints for AJAX requests from the web UI

```
1  from flask import Flask, request, jsonify, render_template
2
3  app = Flask(__name__)
4  sensor_data = {}
5  commands = {}
6
7  print(f'use this link: http://127.0.0.1:5000/node/id')
```

Listing 4.6: espCloudApp.py – Imports and Setup

Explanation: Initial setup defines dictionaries to store:

- `sensor_data` — holds temperature/humidity values per chip ID.
- `commands` — holds the current command to be served to each chip.

```
1  @app.route('/')
2  def home():
3      return "Use /node/<chip_id> to access specific ESP8266 pages."
4
```

```
@app.route('/node/<chip_id>')
def node_page(chip_id):
    return render_template('node.html', chip_id=chip_id)
```

Listing 4.7: espCloudApp.py – Web Routing

Explanation:

- Root route (/) gives a hint to access nodes directly.
- /node/<chip_id> dynamically loads the HTML dashboard for a specific ESP.

```
@app.route('/trigger_temp')
def trigger_temp():
    chip_id = request.args.get("chip_id")
    commands[chip_id] = "get_temp"
    return jsonify({"status": f"Temperature request sent to {chip_id}"})

@app.route('/trigger_humidity')
def trigger_humidity():
    chip_id = request.args.get("chip_id")
    commands[chip_id] = "get_humidity"
    return jsonify({"status": f"Humidity request sent to {chip_id}"})
```

Listing 4.8: espCloudApp.py – Trigger Endpoints

Explanation: These endpoints allow the web UI to signal a data request (temperature or humidity) for a particular ESP node. The ESP will pull this command on its next '/command' request.

```
@app.route('/command')
def get_command():
    chip_id = request.args.get("chip_id")
    if chip_id in commands:
        return jsonify({"command": commands.pop(chip_id)})
    return jsonify({"command": "none"})

@app.route('/send_temp', methods=['POST'])
def send_temp():
    chip_id = request.args.get("chip_id")
    temp = request.json.get("temperature")
    sensor_data.setdefault(chip_id, {})["temperature"] = temp
    return jsonify({"status": "OK", "received": temp})

@app.route('/send_humidity', methods=['POST'])
def send_humidity():
    chip_id = request.args.get("chip_id")
    humidity = request.json.get("humidity")
    sensor_data.setdefault(chip_id, {})["humidity"] = humidity
    return jsonify({"status": "OK", "received": humidity})
```

Listing 4.9: espCloudApp.py – Command & Data Handling

Explanation:

- ESP sends data via POST, which is saved to sensor_data.
- '/command' is polled by ESPs to retrieve any pending commands.

```
@app.route('/get_temp')
def get_temp():
    chip_id = request.args.get("chip_id")
    return jsonify({"temperature": sensor_data.get(chip_id, {}).get("
        temperature", "No data")})

```

```
@app.route('/get_humidity')
def get_humidity():
    chip_id = request.args.get("chip_id")
    return jsonify({"humidity": sensor_data.get(chip_id, {}).get("humidity"
        , "No data")})
```

Listing 4.10: espCloudApp.py – Data Retrieval for Web UI

Explanation: These endpoints return the last known temperature/humidity for a given ESP, used by the frontend to update the UI.

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

Listing 4.11: espCloudApp.py – Server Start

Explanation: Starts the Flask server on port 5000, accessible to other devices on the network.

4.3.2 templates/node.html

This template generates the webpage for each ESP node.

```
<!DOCTYPE html>
<html>
<head>
  <title>ESP8266 Node {{ chip_id }}</title>
  <link rel="stylesheet" href="{{ url_for('static', filename='style.css')
     }}">
</head>
<body>
  <div class="container">
    <h2>ESP8266 Sensor Data - Node {{ chip_id }}</h2>
    <div class="button-group">
      <button onclick="fetchTemperature()">Load Temperature</button>
      <button onclick="fetchHumidity()">Load Humidity</button>
    </div>
    <p id="temp">Temperature: <span class="data highlight">Waiting for data
       ...</span></p>
    <p id="humidity">Humidity: <span class="data highlight">Waiting for
       data...</span></p>
  </div>
```

Listing 4.12: node.html – Dynamic Page

Explanation:

- HTML includes a reference to the shared CSS.
- Buttons allow the user to manually fetch temperature and humidity values.

```
<script>
  function fetchTemperature() {
    fetch(`/trigger_temp?chip_id={{ chip_id }}`)
    .then(() => fetch(`/get_temp?chip_id={{ chip_id }}`))
    .then(response => response.json())
    .then(data => {
      document.getElementById('temp').innerHTML =
        "Temperature: <span class='data highlight'>" + data.temperature + "
            deg C</span>";
    });
  }

  function fetchHumidity() {
    fetch(`/trigger_humidity?chip_id={{ chip_id }}`)
```

```
    .then(() => fetch(`/get_humidity?chip_id={{ chip_id }}`))
    .then(response => response.json())
    .then(data => {
      document.getElementById('humidity').innerHTML =
        "Humidity: <span class='data highlight'>" + data.humidity + "%</
            span>";
    });
  }
</script>
</body>
</html>
```

Listing 4.13: node.html – JavaScript Logic

Explanation:

- On button click, the page sends a trigger request and then pulls the updated value after the ESP responds.
- The values are dynamically inserted into the DOM.

4.3.3 `static/style.css`

This stylesheet enhances the dashboard's visual appearance.

```
.container {
  width: 50%;
  margin: 50px auto;
  background: white;
  padding: 20px;
  border-radius: 12px;
  box-shadow: 0 4px 10px rgba(0, 0, 0, 0.1);
}

button {
  background: linear-gradient(to right, #007bff, #0056b3);
  color: white;
  border: none;
  padding: 12px 25px;
  font-size: 16px;
  border-radius: 5px;
}
```

Listing 4.14: style.css – Highlighted Styles

Explanation: Gives the web dashboard a clean, modern look with rounded containers, vibrant button styles, and responsive layout.

4.3.4 Expected Browser Output

When visiting:

`http://<server_ip>:5000/node/ESP123456`

you will see a web page titled `ESP8266 Sensor Data - Node ESP123456` with two buttons:

- `Load Temperature`
- `Load Humidity`

Values update when you click the buttons.

4.3.5 Homework Tasks

- Add timestamps to temperature and humidity data.
- Add a chart (e.g., Chart.js) to plot the last 10 readings.
- Implement a persistent database using SQLite or CSV.
- Build a "Refresh All" button to load all data simultaneously.
- Add API key authentication to secure endpoints.