Doser ESP8266

1.1 System Architecture

1. **Data Transmission**: ESP32 sends data to ESP8266 via MQTT.
2. **Data Forwarding**: ESP8266 receives data from Arduino Mega 2560 over Serial.
3. **Fan Control**: Arduino Mega 2560 controls mixer, RO, and sampler pump based on pH and EC.

1.2 Overview

In your current setup, the **ESP8266** acts as the main controller responsible for managing Wi-Fi, MQTT communication, NTP synchronization, and receiving data from the **Arduino Mega** over serial communication. The Mega handles data collection for nutrient values (such as pH, EC, dissolved oxygen (DO), temperature, and pump controls for the mixer, RO, and sampler pump) and sends this data to the ESP8266.

1.3 Libraries Used

- **ESP8266WiFi.h**: Provides the Wi-Fi functions for the ESP8266.
- **PubSubClient.h**: A lightweight MQTT client that allows the ESP8266 to publish and subscribe to messages.
- **ArduinoJson.h**: A library used to parse and create JSON documents for efficient data handling.
- **NTPClient.h**: Used to get the current time from an NTP server.
- **WiFiUdp.h**: Required for the NTP client to communicate using UDP.
- **WiFiManager.h**: Simplifies the Wi-Fi setup process, allowing the ESP8266 to connect to different Wi-Fi networks without hardcoding credentials.
- **FS.h**: Provides access to the filesystem on the ESP8266.
- **DNSServer.h**: Part of WiFiManager for handling DNS redirection.

- **EEPROM.h**: It provides mechanisms to store settings (such as WiFi credentials) that persist even after the device is reset or powered off.

- **ArduinoOTA.h**: This feature is enabled if the USEOTA flag is defined. It makes it possible to upload new firmware to the ESP8266 or ESP32 without connecting it to a computer.

1.4 Global Variables and Constants

- **WiFi and MQTT Setup:**
  - **espClient:** This object represents the client for connecting to WiFi.
  - **client:** Created from PubSubClient, it manages MQTT communication.
  - **WiFiManager Objects:**
    - wm and wifiManager: These objects are used to manage WiFi configurations, handling AP/STA modes and credentials storage.
- **NTP Time Synchronization:**
  - **timeClient:** Fetches current time from the NTP server "pool.ntp.org".
  - **weekDays[] and months[]:** Arrays storing names of weekdays and months, which can be used for displaying human-readable dates and times.
- **MQTT Settings:**

- **hostname and port_str:**
  The MQTT broker address (hostname) and port (port_str), which are used for connecting to the MQTT broker (164.52.223.248 or "cea.axalyn.com").
- **clientId, user, and password:**
  MQTT client ID, username, and password are used for authenticating the connection with the MQTT broker.
- **MQTT Topics:**
  - topic: The topic to which data is published.
  - MQTT_PUBTOPIC and MQTT_SUBTOPIC: Topics for publishing and subscribing data in the system.

- **JSON Document Handling:**
  - **DynamicJsonDocument Objects (megadoc, megadoc1, and doc):**
    These are used to handle JSON data for communication, ensuring data is efficiently serialized and deserialized when sent to or received from the MQTT broker.
- **Miscellaneous:**
  - **LED Pin (LED):** Pin 2 is used as an indicator LED.
  - **TEST OPTION FLAGS:**
    - TEST_CP: Used to decide if the configuration portal should always be started.
    - TESP_CP_TIMEOUT: Timeout period for the configuration portal.
    - TEST_NET: Flag to determine whether network tests (such as getting NTP time) should be conducted after connecting to WiFi.
    - ALLOWONDEMAND: Enables or disables the ability to trigger on-demand actions.
    - ONDDEMANDPIN: GPIO pin assigned for triggering on-demand actions (set to 0).
    - WMISBLOCKING: Controls whether WiFiManager operates in blocking or non-blocking mode.
    - shouldSaveConfig: Flag that indicates if the configuration needs to be saved.
- **Timers and Delays:**
  - **previousMillis and interval:**
    Used for non-blocking delays, allowing scheduled tasks (such as MQTT updates or NTP time retrieval) to happen at set intervals.
- **MQTT Control:**
  - **MQTT_Flag:**
    This flag controls whether or not MQTT-related tasks should be processed.
- **Scheduler Object:**
  - **r:** A Scheduler object used to manage timed tasks.

1.5 Functions

1. **sendDataMQTT()**
   - Purpose: Handles sending data to an MQTT broker.
2. **senddatatoKURA()**
   - Purpose: Sends data to a platform called KURA.
3. **SendDataToMega()**
   - Purpose: Manages sending data to another device, possibly an Arduino Mega, over serial or MQTT.

1.6 Function Breakdown

1.6.1 bindServerCallback ()

The bindServerCallback function is responsible for setting up custom routes on the built-in web server in WiFiManager. When called, it binds a new custom route (/custom) to the server, and any client accessing this route will trigger the handleRoute function.

```
void bindServerCallback() {
  wm.server->on("/custom", handleRoute);
  // wm.server->on("/info",handleRoute); // you can override wm!
}
```

1.6.2 handleRoute()

The handleRoute function is the callback that is triggered when the /custom route is accessed. It sends a response of "hello from user code" in plain text to the client.

```
void handleRoute() {
  Serial.println("[HTTP] handle route");
  wm.server->send(200, "text/plain", "hello from user code");
}
```

1.6.3 SendDataToMega()

The SendDataToMega function is responsible for sending a set of data to the Mega via a JSON-formatted string over the Serial interface.

```
void SendDataToMega() {
  DynamicJsonDocument espdata(250);


  espdata["id"] = "WSnode";
  espdata["WIF_v"] = WIF_v;
  espdata["WOF_v"] = WOF_v;
  espdata["WTL_v"] = WTL_v;
  espdata["WTSP_v"] = WTSP_v;
  espdata["WifiSt"] = WifiSt;
  espdata["MqttSt"] = MqttSt;

  Serial.println("Publish mega:");
  serializeJson(espdata, Serial);
  Serial.print("\n");
}
```

1.6.4 **callback()**

The callback function is triggered when a new MQTT message is received on a subscribed topic. It processes the incoming payload and extracts specific data fields.

```
void callback(char*topic, byte* payload, unsigned int length) {
  Serial.print("Message arrived in topic: ");
  Serial.println(topic);

  for (int i = 0; i < length; i++) {
    Serial.print((char)payload[i]);
  }

  DynamicJsonDocument doc(512);
  DeserializationError error = deserializeJson(doc, payload);

  JsonObject As_ZA = doc["As"]["ZA"];
  As_ZA_WIF_v = As_ZA["WIF_v"];
  As_ZA_WOF_v = As_ZA["WOF_v"];
  As_ZA_WTL_v = As_ZA["WTL_v"];
  As_ZA_WTSP_v = As_ZA["WTSP_v"];

  WIF_v = As_ZA_WIF_v;
  WOF_v = As_ZA_WOF_v;
  WTL_v = As_ZA_WTL_v;
  WTSP_v = As_ZA_WTSP_v;
}
```

1.6.5 **sendDataMQTT()**

The sendDataMQTT function is used to send sensor data over MQTT. It collects the current sensor readings and packs them into a JSON document, which is then published to the MQTT topic.

```
void sendDataMQTT() {
  time_t epochTime = timeClient.getEpochTime();
  String phval;
  String ecval;

  if (Serial.available()) {
    StaticJsonDocument<500> doc;
    DeserializationError err = deserializeJson(doc, Serial);
    if (err == DeserializationError::Ok) {
      ph_value = doc["pH"].as<float>();
      ec_value = doc["EC"].as<float>();
      nt_value = doc["NT"].as<float>();
      ndo_value = doc["DO"].as<float>();
      ntl_value = doc["NL"].as<int>();

      ph_aspvalue = doc["pHSetPA"].as<long>();
      ec_aspvalue = doc["ECSetPA"].as<long>();
      SubP = doc["Sampler"].as<long>();
      Heater = doc["Heater"].as<long>();
      Mixer = doc["Mixer"].as<long>();
      RO = doc["RO"].as<long>();
    }
  }

  DynamicJsonDocument doc(1024);
  doc["id"] = "TxDc8266";
  doc["ph"] = ph_value;
  doc["ec"] = ec_value;
  doc["ntemp"] = nt_value;
  doc["do"] = ndo_value;
  doc["ntl"] = ntl_value;
  doc["pHSp"] = ph_aspvalue;
  doc["EcSp"] = ec_aspvalue;
  doc["SubP"] = SubP;
  doc["Heater"] = Heater;
  doc["Mixer"] = Mixer;
  doc["RO"] = RO;

  serializeJson(doc, tempString);
  client.publish("axalyn", tempString);
}
```

1.6.6 **reconnect()**

The reconnect function ensures the MQTT client is connected to the broker. If disconnected, it tries to reconnect and subscribes to the required MQTT topics.

```
void reconnect() {
  while (!client.connected()) {
    if (client.connect("ESP32Client", user, password)) {
      client.subscribe("axalyn");
    } else {
      delay(5000);
    }
  }
}
```

1.6.7 **void loop()**

The loop() function runs continuously after the setup phase. It handles several key tasks such as MQTT reconnection, data transmission to the MQTT broker and to the Mega, and WiFi management.

```
void loop() {
  r.execute();
  timeClient.update(); // Updates the NTP client for time

  // Check and reconnect MQTT client every 3 seconds
  unsigned long now1 = millis();
  if (now1 - lastMsg > 3000) {
    lastMsg = now1;
    if (!client.connected()) {
      reconnect();
    }
  }

  // Process WiFiManager if not blocking
  if (!WMISBLOCKING) {
    wm.process();
```

```
}


// Send data to MQTT broker every second
unsigned long currentMillis = millis();
if (currentMillis - previousMillis >= 1000) {
  previousMillis = currentMillis;
  Serial.println("Sending Data");
  if (client.connected()) {
    sendDataMQTT();
  }
}


// MQTT client loop to handle messages
client.loop();
delay(100);


// Send data to Mega microcontroller every 3 seconds
unsigned long megacurentmills = millis();
unsigned long megaprevious = 0;
if (megacurentmills - megaprevious >= 3000) {
  megaprevious = megacurentmills;
  Serial.println("Sending Data");
  if (client.connected()) {
    SendDataToMega();
   }
  }
}
```

1.6.8 **void saveConfig()**

This function saves the current MQTT configuration to the ESP's SPIFFS (SPI Flash File System) to allow it to persist even after rebooting.

```
void saveConfig() {
  File configFile = SPIFFS.open("/config.json", "FILE_WRITE");
  if (!configFile) {
    Serial.println("Failed to open config file for writing");
    return;
  }

  // Create a JSON object and write to the file
  DynamicJsonDocument json(1024);
  json["clientID"] = clientId;
  json["mqttServer"] = hostname;
  json["mqttPort"] = port_str;
  json["mqttUserName"] = user;
  json["mqttPwd"] = password;

  // Serialize JSON and save to file
  serializeJson(json, configFile);

  configFile.close();

  Serial.println("Configuration saved to SPIFFS");

}
```

1.6.9 **void loadConfig()**

This function loads the MQTT configuration from the SPIFFS. It is used to retrieve previously saved settings during boot-up.

```
void loadConfig() {

  File configFile = SPIFFS.open("/config.json", "FILE_READ");

  if (!configFile) {

    Serial.println("Config file not found.");

    return;

  }



  DynamicJsonDocument json(1024);

  DeserializationError error = deserializeJson(json, configFile);

  if (error) {

    Serial.println("Failed to parse config file.");

    return;

  }



  String testValue = json["test"] | "default";

  Serial.println("Test value loaded: " + testValue);
  configFile.close();

}
```

1.6.10 **void wifiInfo()**

This function displays diagnostic information about the current WiFi connection and saved WiFi credentials.

```
void wifiInfo() {

  WiFi.printDiag(Serial); // Prints WiFi diagnostic info to the serial monitor

  Serial.println("SAVED: " + (String)wm.getWiFiIsSaved() ? "YES" : "NO");

  Serial.println("SSID: " + (String)wm.getWiFiSSID());

  Serial.println("PASS: " + (String)wm.getWiFiPass());

}
```

1.6.11 **void initializeWifiManager()**

This function sets up the WiFiManager for handling the device's WiFi configuration and connection to the MQTT broker.

```
void initializeWifiManager() {

  WiFiManagerParameter custom_mqtt_client_id("client_id", "mqtt client id", clientId, 40);

  WiFiManagerParameter custom_mqtt_server("server", "mqtt server", hostname, 40);

  WiFiManagerParameter custom_mqtt_port("port", "mqtt port", port_str, 6);

  WiFiManagerParameter custom_mqtt_user("user", "mqtt user", user, 20);

  WiFiManagerParameter custom_mqtt_pass("pass", "mqtt pass", password, 20);



  wifiManager.setSaveConfigCallback(saveConfigCallback);
  wifiManager.addParameter(&custom_mqtt_client_id);
  wifiManager.addParameter(&custom_mqtt_server);
  wifiManager.addParameter(&custom_mqtt_port);
  wifiManager.addParameter(&custom_mqtt_user);
  wifiManager.addParameter(&custom_mqtt_pass);

  // Auto connect to WiFi or enter configuration mode
  if (!wifiManager.autoConnect("DOSER CONTROLLER")) {
    Serial.println("Failed to connect and hit timeout");
    delay(3000);
    ESP.restart();  // Restart ESP if connection fails
    delay(5000);
  }

  // Read and save the updated WiFi credentials
  strcpy(clientId, custom_mqtt_client_id.getValue());
  strcpy(hostname, custom_mqtt_server.getValue());
  strcpy(port_str, custom_mqtt_port.getValue());
  strcpy(user, custom_mqtt_user.getValue());
  strcpy(password, custom_mqtt_pass.getValue());
}
```

1.6.12 **void saveConfigCallback()**

This callback function is triggered when the WiFiManager configuration needs to be saved. It sets a flag (shouldSaveConfig) to true, indicating that the configuration needs to be written to SPIFFS.

```
void saveConfigCallback () {

  Serial.println("Should save config");

  shouldSaveConfig = true;  // Set flag to true to save the config later

}
```