IRR_ESP866

1.1 SYSTEM ARCHITECTURE

1. **Initialization**: ESP8266 initializes serial communication, SPIFFS, WiFi, MQTT connection, and NTP time synchronization.
2. **Event Handling**: ESP8266 processes event data from Mega 2560 to determine actions based on temperature, VPD, and irrigation events.
3. **Manual Relay Status**: Updates and manages manual relay statuses based on incoming JSON data from Mega 2560.
4. **Alert Management**: Processes and handles alert modes and levels, triggering actions based on alert data.
5. **Data Transmission to Cloud**: ESP8266 publishes sensor data and status updates to a cloud server via MQTT.
6. **MQTT Communication**: Manages MQTT connections, handles publishing and subscribing to topics, and processes incoming messages.
7. **Timing and Scheduling**: Executes periodic tasks, such as sending data, updating statuses, and managing time-based functions.

1.2 LIBRARIES:

- **ESP8266WiFi.h**: Manages WiFi connectivity for the ESP8266 microcontroller.
- **PubSubClient.h**: Provides MQTT client functionality to publish and subscribe to topics.
- **ArduinoJson.h**: Enables parsing and generation of JSON formatted data.
- **RunningAverage.h**: Calculates the running average of a series of numbers (not directly used in the provided code).
- **NTPClient.h**: Synchronizes time with an NTP server over the internet.
- **WiFiUdp.h**: Provides UDP (User Datagram Protocol) support for networking.
- **millisDelay.h**: Implements non-blocking delays using the millis() function.
- **FS.h**: Handles file system operations on the ESP8266.
- **DNSServer.h**: Sets up a DNS server, commonly used for captive portal functionality in WiFiManager.
- **ESP8266WebServer.h**: Creates a simple web server to serve web pages or handle HTTP requests.
- **WiFiManager.h**: Simplifies WiFi configuration management, allowing users to configure WiFi credentials via a web portal.

1.3 GLOBAL VARIABLES:

- **chipid**: Stores the unique identifier for the ESP8266 chip.
- **eventogo, eventogo1, eventogo2**: Store event-related data received from the Mega controller.
- **alertmode1, alertlevel, valuedata**: Store alert data received from the Mega controller.
- **tempstring1, tempstring2, tempstring11, RELAYSTRING**: Buffers for storing and processing string data.
- **Manualstatus_irz1a, Manualstatus_irz1b, Manualstatus_irz2a, Manualstatus_irz2b, Manualstatus_doa, Manualstatus_dob**: Hold manual control statuses for different irrigation zones and devices.
- **Alerts_Mode, Alerts_level, MANUALRELAY**: Store alert mode, alert level, and manual relay status received from the Mega controller.
- **Events_TEMP, Events_VPD, Events_IRR**: Store event statuses for temperature, VPD, and irrigation received from the Mega controller.

- **hostname**: Stores the MQTT broker's IP address or domain name.
- **port_str**: String representation of the MQTT broker's port number.
- **port**: Integer value of the MQTT broker's port number.
- **clientId, user, password**: Store the MQTT client ID, username, and password for authentication.
- **MQTT_SUBTOPIC, MQTT_PUBTOPIC, MQTT_PUBTOPIC1**: MQTT topics used for subscribing to control commands and publishing sensor data and system status.
- **sendtopic_ESMesh, sendtopic_WSMesh**: MQTT topics for sending climate data to specific destinations.

1.4 FUNCTION DESCRIPTION

1.4.1 void eventstatus()

Updates event status and initializes actions based on event data received from the Mega 2560.

1.4.2 void manualstatus()

Parses and updates manual relay statuses based on JSON data received from the Mega 2560.

1.4.3 void alert()

Processes and sets alert modes and levels based on JSON data received from the Mega 2560.

1.4.4 void intializer(int toperform)

Processes and sets alert modes and levels based on JSON data received from the Mega 2560.

1.4.5 void tocloud()

Sends sensor and alert data to a cloud server using MQTT, including formatted time and event metrics.

1.4.6 void sendsetpointmega()

Sends updated relay control data to the Mega 2560 if the valuedata has changed.

1.4.7 void sendnodedata()

Placeholder function for preparing and sending node data, currently not implemented.

1.4.8 void callback(char* topic, byte * payload, unsigned int length)

Handles incoming MQTT messages, deserializes payloads, and updates global variables.

1.4.9 void reconnect()

Reconnects to the MQTT broker and re-subscribes to topics if the connection is lost.

1.4.10 void HandleMqttState()

Maintains MQTT connection and processes incoming messages, ensuring the client stays connected.

1.5 FUNCTION BREAKDOWN

1.5.1 eventstatus()

Purpose:
This function processes event data received from the Mega controller via Serial communication. It parses the incoming JSON data, extracts event statuses for temperature, VPD, and irrigation, and triggers specific actions based on the event values.

```
void eventstatus() {
  DynamicJsonDocument event(500);

  // Deserialize JSON data received over Serial
  DeserializationError error = deserializeJson(event, Serial);

  // If deserialization fails, exit the function
  if (error) {
    return;
  }

  // Extract event statuses for TEMP, VPD, and IRR
  Events_TEMP = event["Events"]["TEMP"];
  Events_VPD = event["Events"]["VPD"];
  Events_IRR = event["Events"]["IRR"];

  // Trigger actions based on event statuses
  if (Events_TEMP == 3) {
    intializer(3);
    intializer(1);
  }

  if (Events_VPD == 4) {
    intializer(4);
```

```
  intializer(1);
 }
 if (Events_IRR == 5) {
  intializer(5);
 }
 if (Events_IRR == 6) {
  intializer(6);
 }
 if (Events_IRR == 21) {
  intializer(21);
 }
}
```

1.5.2 manualstatus()

Purpose:
This function retrieves the manual status of irrigation zones and devices from the Mega controller via Serial communication. It parses the incoming JSON data, extracts the manual statuses, and stores them in global variables for further use.

```
void manualstatus() {
 DynamicJsonDocument relaystatus(500);

 // Deserialize JSON data received over Serial
 DeserializationError error = deserializeJson(relaystatus, Serial);

 // If deserialization fails, exit the function
 if (error) {
  return;
 }

 // Extract manual statuses for irrigation zones and devices
 JsonObject Manualstatus = relaystatus["Manualstatus"];
```

```
  Manualstatus_irz1a = Manualstatus["irz1a"];
  Manualstatus_irz1b = Manualstatus["irz1b"];
  Manualstatus_irz2a = Manualstatus["irz2a"];
  Manualstatus_irz2b = Manualstatus["irz2b"];
  Manualstatus_doa = Manualstatus["doa"];
  Manualstatus_dob = Manualstatus["dob"];
}
```

1.5.3 alert()

Purpose:
This function processes alert data received from the Mega controller via Serial communication. It parses the incoming JSON data, extracts alert mode and level information, and triggers specific actions based on the alert values.

```
void alert() {
  DynamicJsonDocument alertfrom(500);

  // Deserialize JSON data received over Serial
  DeserializationError error = deserializeJson(alertfrom, Serial);

  // If deserialization fails, exit the function
  if (error) {
    return;
  }

  // Extract alert mode and level
  Alerts_Mode = alertfrom["Alerts"]["Mode"];
  Alerts_level = alertfrom["Alerts"]["level"];

  // Trigger actions based on alert mode and level
  if (Alerts_Mode == 7) {
    intializer(7);
```

```
  }
  if (Alerts_Mode == 12) {
    intializer(12);
  }
  if (Alerts_level == 8) {
    intializer(8);
  }
  if (Alerts_level == 9) {
    intializer(9);
  }
  if (Alerts_level == 10) {
    intializer(10);
  }
  if (Alerts_level == 11) {
    intializer(11);
  }
}
```

1.5.4 intializer(int toperform)

Purpose:
This function handles the initialization and configuration of different events, alerts, and manual operations based on the toperform parameter. It sets appropriate string values based on the provided event or alert identifier.

```
void intializer(int toperform) {
  switch (toperform) {
    case 1:
      // Nullify event strings if events are not triggered
      if (Events_TEMP == 0) {
        eventogo = "null";
      }
      if (Events_VPD == 0) {
```

```
    eventogo1 = "null";
  }
  if (Events_IRR == 0) {
    eventogo2 = "null";
  }
  break;

case 3:
  // Set event string for automatic temperature control
  eventogo = "Autotemp";
  break;

case 4:
  // Set event string for automatic VPD control
  eventogo1 = "Autovpd";
  break;

case 5:
  // Set event string for automatic irrigation control
  eventogo2 = "AutoIrr";
  break;

case 6:
  // Set event string for scheduled irrigation control
  eventogo2 = "ScheIrr";
  break;

case 7:
  // Set alert mode string for VPD
  alertmode1 = "VPD";
```

```
  break;

case 8:
  // Set alert level string for critical
  alertlevel = "Critical";
  break;

case 9:
  // Set alert level string for moderate
  alertlevel = "moderate";
  break;

case 10:
  // Set alert level string for low
  alertlevel = "low";
  break;

case 11:
  // Set alert level string for normal
  alertlevel = "normal";
  break;

case 12:
  // Set alert mode string for temperature
  alertmode1 = "Temperature";
  break;

case 21:
  // Set alert mode string for auto and scheduled control
  alertmode1 = "Auto and Sche";
```

```
    break;
  }
}
```

1.5.5 tocloud()

Purpose:

The tocloud() function collects the current system statuses, including alerts, manual irrigation statuses, and events, formats them into a JSON structure, and publishes this data to a cloud topic named "axalyn" via MQTT.

```
void tocloud() {
  // Get the ESP8266 chip ID
  chipid = ESP.getChipId();

  // Get the current epoch time
  time_t epochTime = timeClient.getEpochTime();

  // Format the current time (HH:MM:SS)
  String formattedTime = timeClient.getFormattedTime();

  // Create a JSON document with a capacity of 1024 bytes
  DynamicJsonDocument mydoc(1024);

  // Populate JSON with device information and timestamp
  mydoc["dn"] = "IRR_ESP";  // Device name
  mydoc["zone_id"] = "5805";  // Zone ID
  mydoc["did"] = ESP.getChipId();  // Device ID (Chip ID)
  mydoc["ts"] = epochTime;  // Timestamp

  // Create a nested object for metrics
  JsonObject PFCDATA = mydoc.createNestedObject("metrics");
```

IRR_ESP866

```
// Create a nested object within metrics for alerts
JsonObject Alerts = PFCDATA.createNestedObject("alerts");
Alerts["Mode"] = alertmode1;  // Alert mode (e.g., VPD, Temperature)
Alerts["level"] = alertlevel;  // Alert level (e.g., Critical, moderate)

// Create a nested object within alerts for manual irrigation statuses
JsonObject MANUALstatus = Alerts.createNestedObject("Manualstatus");
MANUALstatus["irz1a"] =  Manualstatus_irz1a;  // Status of irrigation zone 1a
MANUALstatus["irz1b"] =  Manualstatus_irz1b;  // Status of irrigation zone 1b
MANUALstatus["irz2a"] =  Manualstatus_irz2a;  // Status of irrigation zone 2a
MANUALstatus["irz2b"] =  Manualstatus_irz2b;  // Status of irrigation zone 2b
MANUALstatus["doa"] =  Manualstatus_doa;  // Status of device output A
MANUALstatus["dob"] =  Manualstatus_dob;  // Status of device output B

// Create a nested object within metrics for events
JsonObject EVENTS = PFCDATA.createNestedObject("Events");
EVENTS["IRR"] = eventogo2;  // Irrigation event status

// Serialize JSON document to a string and store it in tempstring11
serializeJson(mydoc, tempstring11);

// Print the JSON document to the Serial monitor (for debugging)
Serial.println("");
serializeJson(mydoc, Serial);

// Publish the JSON document to the "axalyn" MQTT topic
client.publish("axalyn", tempstring11);

// Clear the tempstring11 buffer for future use
memset(tempstring11, 0, 500);
```

```
}
```

1.5.6 sendsetpointmega()

Purpose: This function constructs a JSON object to send relay control data (FAN1 status) to the Mega 2560 if there's a change in the valuedata. The data is then serialized and sent over Serial communication.

```
void sendsetpointmega() {
  DynamicJsonDocument relaydoc(1024);  // Create a JSON document with 1024 bytes capacity
  JsonObject CHANGERELAY = relaydoc.createNestedObject("Allrelays");  // Create a nested object


  if (valuedata != NULL) {
    String compare;
    String readd = valuedata;


    if (compare != readd) {
      compare = readd;  // Update the comparison string
      CHANGERELAY["FAN1"] =  readd;  // Set FAN1 status in the JSON document
    }


    serializeJson(relaydoc, RELAYSTRING);  // Serialize the JSON document to RELAYSTRING
    Serial.println("");
    delay(50);
    serializeJson(relaydoc, Serial);  // Send the JSON over Serial to Mega 2560
    Serial.print("\n");
    memset(RELAYSTRING, 0, 1024);  // Clear the RELAYSTRING buffer
  }
}
```

1.5.7 sendnodedata()

Purpose: This function appears to be intended for preparing and sending node-specific data metrics, but the implementation is incomplete. The function currently creates two JSON documents, likely intended for different metrics or nodes.

```
void sendnodedata() {
```

```
  DynamicJsonDocument metricdoc(1024);  // Create a JSON document for metrics
  DynamicJsonDocument Nodedoc(1024);  // Create a separate JSON document for node data

  // Function implementation is incomplete and doesn't perform any operations yet
}
```

1.5.8 callback(char* topic, byte * payload, unsigned int length)

Purpose: This is an MQTT callback function that is triggered when a message is received on a subscribed topic. The function deserializes the JSON payload and updates the global valuedata variable.

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

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

  DynamicJsonDocument doc(512);  // Create a JSON document with 512 bytes capacity
  DeserializationError error = deserializeJson(doc, payload);  // Deserialize the JSON payload

  if (error) {
    Serial.print(F("deserializeJson() failed: "));
    Serial.println(error.f_str());
    return;  // Exit if deserialization fails
  }

  valuedata = doc["data"];  // Update the global valuedata variable
}
```

1.5.9 reconnect()

Purpose: This function handles reconnection to the MQTT broker if the client is disconnected. It attempts to reconnect and resubscribe to the necessary topics.

```
void reconnect() {
  while (!client.connected()) {  // Loop until reconnected
    if (client.connect("ESP8266Client", user, password)) {  // Attempt to connect
      client.subscribe(sendtopic_WSMesh);  // Subscribe to the necessary topics
      client.subscribe("axalyn");
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      delay(5000);  // Wait 5 seconds before retrying
      ESP.restart();  // Restart the ESP8266 if the connection fails
    }
  }
}
```

1.5.10 HandleMqttState()

Purpose: This function checks the MQTT client state and reconnects if the client is disconnected. It also processes incoming MQTT messages by calling client.loop().

```
void HandleMqttState() {
  if (!client.connected()) {
    reconnect();  // Reconnect if the client is disconnected
  }
  client.loop();  // Process incoming MQTT messages
}
```

1.5.11 setup()

Purpose: The setup() function initializes the system, including the Serial communication, SPIFFS file system, WiFi, NTP time synchronization, and MQTT client setup. It ensures that the device is ready for operation by loading configurations and setting up network connectivity.

IRR_ESP866

```
void setup() {
  Serial.begin(115200);  // Initialize Serial communication at 115200 baud rate

  // Initialize SPIFFS (File System)
  if (!SPIFFS.begin()) {
    Serial.println("SPIFFS Mount Failed");
    while (true) {
      delay(1000);  // Halt the system if SPIFFS fails to mount
    }
  }

  // Save updated configuration if needed
  if (shouldSaveConfig) {
    Serial.println("Saving updated configuration...");
    saveConfig();  // Save configuration settings
  }

  loadConfig();  // Load configuration settings from SPIFFS or EEPROM
  initializeWifiManager();  // Set up WiFiManager for WiFi and MQTT configuration
  wifiManager.setTimeout(180);  // Set WiFiManager timeout to 180 seconds

  wifiInfo();  // Display WiFi information
  timeClient.begin();  // Start NTP time synchronization
  timeClient.setTimeOffset(19800);  // Set time offset for your timezone (e.g., IST)

  // MQTT setup
  client.setServer(hostname, port);  // Set the MQTT server and port
  client.setCallback(callback);  // Set the MQTT message callback function
}
```

1.5.12 loop()

Purpose: The loop() function handles the main logic of the device, including MQTT state management, periodic tasks like sending data to the Mega 2560, handling alerts, and sending data to the cloud. It uses non-blocking timing based on the millis() function to execute tasks at specified intervals.

```
unsigned long SCpreviousMillis = 0;
unsigned long SendtomegapreviousMillis = 0;
unsigned long tocloudstatus = 0;
unsigned long torecievestatus = 0;
unsigned long torecievestatusm = 0;


void loop() {
  HandleMqttState();  // Check and maintain MQTT connection
  timeClient.update();  // Update the NTP time
  client.loop();  // Process incoming MQTT messages

  unsigned long currentMillis = millis();

  // Send setpoint data to Mega 2560 every 1000ms
  if (currentMillis - SCpreviousMillis >= 1000) {
    SCpreviousMillis = currentMillis;
    sendsetpointmega();
  }

  // Start getting values from Mega 2560 every 3000ms
  unsigned long megacurrentMillis = millis();
  if (megacurrentMillis - SendtomegapreviousMillis >= 3000) {
    SendtomegapreviousMillis = megacurrentMillis;
    getvalue.start(100);  // Start a timed process to get values
  }
```

```
// Send node data while the getvalue process is running
if (getvalue.isRunning()) {
  sendnodedata();
}

// Check if getvalue process just finished
if (getvalue.justFinished()) {
  // Add any required operations when the process finishes
}

// Handle alert and event status every 5000ms
unsigned long torecievecurrentMillis = millis();
if (torecievecurrentMillis - torecievestatus >= 5000) {
  torecievestatus = torecievecurrentMillis;
  alert();  // Handle alerts
  delay(15);  // Short delay
  eventstatus();  // Handle event status
}

// Handle manual status every 4000ms
unsigned long torecievecurrentMillism = millis();
if (torecievecurrentMillism - torecievestatusm >= 4000) {
  torecievestatusm = torecievecurrentMillism;
  manualstatus();  // Handle manual status
}

// Send data to the cloud every 8000ms
unsigned long cloudcurrentMillis = millis();
if (cloudcurrentMillis - tocloudstatus >= 8000) {
  tocloudstatus = cloudcurrentMillis;
```

```
    tocloud();  // Send data to the cloud
  }
}
```