IMP CODE ( MAIN ONES)

#include <LiquidCrystal.h>

#include <ESP8266WiFi.h> // NodeMCU or ESP8266 Wi-Fi communication

#include <ACS712.h> // Current sensor library

// Pin definitions

const int currentSensorPin = A0;

const int relayPin = 7;

const int buzzerPin = 8;

const int buttonPin = 2;

const int potentiometerPin = A1;

// Wi-Fi credentials

const char\* ssid = "your\_wifi\_ssid";

const char\* password = "your\_wifi\_password";

// Node-RED server details

const char\* server = "your\_node\_red\_server\_ip";

const int port = 1880; // Typically for Node-RED, or use 80 for HTTP

WiFiClient client;

ACS712 currentSensor(ACS712\_30A, currentSensorPin);

// LCD

LiquidCrystal lcd(12, 11, 5, 4, 3, 2); // Modify based on your connections

void setup() {

// Initialize hardware

pinMode(relayPin, OUTPUT);

pinMode(buzzerPin, OUTPUT);

pinMode(buttonPin, INPUT);

pinMode(potentiometerPin, INPUT);

// Initialize LCD

lcd.begin(16, 2);

lcd.print("System Booting...");

// Wi-Fi connection

connectWiFi();

// Initialize current sensor

currentSensor.calibrate();

lcd.clear();

lcd.print("System Ready");

}

void loop() {

float current = currentSensor.getCurrentDC(); // Get current reading from sensor

lcd.setCursor(0, 1);

lcd.print("Current: ");

lcd.print(current);

lcd.print(" A");

// Fault detection logic

if (current < 10) { // Example open circuit threshold

lcd.clear();

lcd.print("Fault Detected");

// Activate relay and buzzer

digitalWrite(relayPin, HIGH);

digitalWrite(buzzerPin, HIGH);

// Send fault data to Node-RED

sendDataToNodeRed(current);

} else {

// System normal

lcd.clear();

lcd.print("Line OK");

// Deactivate relay and buzzer

digitalWrite(relayPin, LOW);

digitalWrite(buzzerPin, LOW);

}

delay(1000); // Adjust delay as needed

}

// Wi-Fi connection function

void connectWiFi() {

WiFi.begin(ssid, password);

while (WiFi.status() != WL\_CONNECTED) {

delay(1000);

lcd.clear();

lcd.print("Connecting...");

}

lcd.clear();

lcd.print("Wi-Fi Connected");

}

// Send data to Node-RED

void sendDataToNodeRed(float current) {

if (client.connect(server, port)) {

String data = "current=" + String(current);

client.println("POST /update HTTP/1.1");

client.println("Host: your\_node\_red\_server\_ip");

client.println("Content-Type: application/x-www-form-urlencoded");

client.print("Content-Length: ");

client.println(data.length());

client.println();

client.println(data);

}

client.stop();

}

**2. Node-RED Setup**

**Node-RED** is a flow-based development tool that helps visualize data from IoT devices, control relays, and integrate cloud-based services for monitoring.

**Steps:**

1. **Install Node-RED**: You can install it locally or run it on a server (like Raspberry Pi or cloud VM).
   * Use the command: npm install -g node-red to install.
   * Start with: node-red
2. **Create a Flow**:
   * Drag and drop an **HTTP input node** to listen for data from the Arduino code (POST method).
   * Use a **function node** to process incoming current data.
   * Add **Dashboard nodes** (for web-based monitoring) and **chart nodes** to display current status.
   * **Optional:** Use an **email or Telegram node** to send notifications if a fault is detected.

**Example Node-RED Flow:**

* **Input**: HTTP (NodeMCU sends current data).
* **Process**: Use a function node to check the value of current and trigger alarms if needed.
* **Output**: Show the current status in a Dashboard UI and send an alert when fault detected.

JSON CODE

[

{

"id": "f91d0",

"type": "http in",

"name": "Fault Data",

"url": "/update",

"method": "post",

"swaggerDoc": "",

"x": 100,

"y": 100,

"wires": [["ea2b0"]]

},

{

"id": "ea2b0",

"type": "function",

"name": "Process Data",

"func": "var current = parseFloat(msg.payload.current);\nif (current < 10) {\n node.send([msg, null]); // Fault condition\n} else {\n node.send([null, msg]); // Normal condition\n}\nreturn msg;",

"outputs": 2,

"x": 300,

"y": 100,

"wires": [["dashboard\_fault"], ["dashboard\_normal"]]

},

{

"id": "dashboard\_fault",

"type": "ui\_text",

"group": "UI",

"label": "Fault Detected",

"format": "{{msg.payload}}",

"x": 500,

"y": 100,

"wires": []

},

{

"id": "dashboard\_normal",

"type": "ui\_text",

"group": "UI",

"label": "Line OK",

"format": "{{msg.payload}}",

"x": 500,

"y": 150,

"wires": []

}

]

**3. Cloud Integration**

You can integrate with cloud services for real-time monitoring and data storage:

**Options:**

* **ThingSpeak**: Easily integrates with NodeMCU or ESP8266 for IoT data visualization.
* **Firebase**: Real-time database for tracking current sensor data and fault status.
* **AWS IoT or Azure IoT Hub**: For more advanced cloud-based deployments.

**Example:**

In the code above, modify the sendDataToNodeRed() function to send data to **ThingSpeak** or **Firebase** if you prefer those platforms.

**4. KiCAD for PCB Design**

Once your system works reliably on a breadboard, you can move to a more permanent solution using a **custom PCB**.

* **KiCAD**: An open-source tool for PCB design.
  + You can create a schematic diagram with your components (Arduino, ESP8266, ACS712, relay module, etc.).
  + Design the PCB layout.
  + Use the **Gerber files** generated by KiCAD to manufacture the PCB.

**5. Testing & Debugging**

* Use **TinkerCAD** or **Proteus** for simulation before hardware deployment.
* Validate **ESP8266 connectivity** and **Node-RED data flow** using serial prints or dashboard visuals.
* Conduct real-world testing with actual electrical loads to ensure the system detects faults accurately.

**Final Tech Stack:**

* **Hardware**:
  + Arduino, ACS712 Current Sensor, Relay Module, ESP8266/NodeMCU, LCD Display, Potentiometer, Buzzer.
* **Software**:
  + Arduino IDE (for coding), Node-RED (for flow-based logic and dashboards), KiCAD (PCB design), ThingSpeak/Firebase (for cloud integration).

**1. Cloud API Integration (ThingSpeak or Firebase)**

If you want to store your sensor data in the cloud, you’ll need to send the data to services like **ThingSpeak** or **Firebase**.

**Example Code for ThingSpeak Integration:**

#include <ESP8266WiFi.h>

#include <ThingSpeak.h>

const char\* ssid = "your\_wifi\_ssid";

const char\* password = "your\_wifi\_password";

const unsigned long channelID = YOUR\_CHANNEL\_ID;

const char\* apiKey = "your\_thingspeak\_api\_key";

WiFiClient client;

void setup() {

WiFi.begin(ssid, password);

while (WiFi.status() != WL\_CONNECTED) {

delay(1000);

Serial.println("Connecting to Wi-Fi...");

}

ThingSpeak.begin(client); // Initialize ThingSpeak connection

}

void loop() {

float current = currentSensor.getCurrentDC();

// Update ThingSpeak channel

ThingSpeak.setField(1, current); // Assuming Field 1 for current data

int x = ThingSpeak.writeFields(channelID, apiKey);

if (x == 200) {

Serial.println("Data updated successfully");

} else {

Serial.println("Error updating data");

}

delay(10000); // Adjust the delay as per your requirements

}

This will upload the current data from your sensor to **ThingSpeak**, which can then be visualized in real-time using their dashboard.

**Firebase Realtime Database Code Example:**

#include <ESP8266WiFi.h>

#include <FirebaseESP8266.h>

// Firebase credentials

#define FIREBASE\_HOST "your-firebase-database.firebaseio.com"

#define FIREBASE\_AUTH "your-firebase-auth-key"

WiFiClient client;

void setup() {

WiFi.begin(ssid, password);

while (WiFi.status() != WL\_CONNECTED) {

delay(1000);

Serial.println("Connecting to Wi-Fi...");

}

Firebase.begin(FIREBASE\_HOST, FIREBASE\_AUTH); // Initialize Firebase connection

}

void loop() {

float current = currentSensor.getCurrentDC();

// Send data to Firebase

if (Firebase.setFloat("current", current)) {

Serial.println("Data uploaded to Firebase successfully");

} else {

Serial.println("Error uploading to Firebase");

}

delay(10000); // Adjust as needed

}

**2. LCD Display Interaction Improvement**

You can add some interactivity with the **LCD** to display more comprehensive information, such as fault count or time since the last fault detection.

lcd.setCursor(0, 0);

lcd.print("Line Status: ");

if (current < 10) {

lcd.setCursor(0, 1);

lcd.print("FAULT DETECTED");

} else {

lcd.setCursor(0, 1);

lcd.print("Line OK");

}

**3. More Advanced Node-RED Flows**

You might want to add a flow in **Node-RED** that:

* Logs historical data to a database like **MongoDB** or **InfluxDB**.
* Generates real-time charts and fault alerts using the **Node-RED Dashboard**.

Example flow:

* Add nodes like **HTTP Input**, **Function**, **UI Chart**, and **Email Node** to trigger notifications when faults are detected and visualize the current levels over time.

**4. Calibration Code for ACS712 Sensor**

If you want more accurate current measurements, you can include a calibration function for the **ACS712** sensor:

float sensorValue = 0;

int offset = 512; // Default midpoint for analogRead on ACS712

void calibrateSensor() {

long totalValue = 0;

for (int i = 0; i < 100; i++) {

totalValue += analogRead(A0);

delay(10);

}

offset = totalValue / 100;

}

**5. Relay Module (for Cutoff or Fault Response):**

* If you’re including a relay to switch off faulty lines, integrate the relay with the code. You’ll need to:
  + Define a pin to control the relay.
  + Add logic to trigger the relay when a fault is detected.

Example:

int relayPin = D2; // Define relay pin

void setup() {

pinMode(relayPin, OUTPUT);

}

void loop() {

if (current < 10) {

digitalWrite(relayPin, HIGH); // Turn off the relay

} else {

digitalWrite(relayPin, LOW); // Turn relay back on

}

}

**6. Additional Sensors (Temperature, Humidity, etc.):**

* If you're using additional sensors like **DHT11/DHT22** for temperature and humidity, integrate those into the code as well.

#include "DHT.h"

DHT dht(D4, DHT11); // Adjust for DHT22 if needed

void setup() {

dht.begin();

}

void loop() {

float temp = dht.readTemperature();

float hum = dht.readHumidity();

if (!isnan(temp) && !isnan(hum)) {

lcd.setCursor(0, 0);

lcd.print("Temp: ");

lcd.print(temp);

lcd.setCursor(0, 1);

lcd.print("Humidity: ");

lcd.print(hum);

}

}

**7. Push Buttons and Potentiometer:**

* If you're including a **push button** or **potentiometer** for user input (e.g., resetting the system or adjusting thresholds), ensure they’re properly wired and handled in the code.

Example for Push Button:

int buttonPin = D3;

int buttonState = 0;

void setup() {

pinMode(buttonPin, INPUT);

}

void loop() {

buttonState = digitalRead(buttonPin);

if (buttonState == HIGH) {

// Perform an action, e.g., reset the system

}

}

COMPLETE CODE REVISED

#include <LiquidCrystal.h> // Include the library for LCD

#include <ESP8266WiFi.h> // Include library for WiFi

#include <PubSubClient.h> // Include library for MQTT

// WiFi and MQTT Setup

const char\* ssid = "Your\_SSID"; // Replace with your SSID

const char\* password = "Your\_PASSWORD"; // Replace with your WiFi password

const char\* mqttServer = "your.mqtt.broker"; // Replace with your MQTT broker address

const int mqttPort = 1883; // MQTT broker port

const char\* mqttTopic = "openCircuitStatus"; // MQTT topic to publish

// Current Sensor Pin

const int currentSensorPin = A0; // Analog pin connected to the current sensor

const float sensitivity = 0.185; // Sensitivity for ACS712 5A model (adjust based on your model)

const float voltageSupply = 5.0; // Supply voltage for the current sensor

// LCD Pins

LiquidCrystal lcd(12, 11, 5, 4, 3, 2); // Adjust according to your wiring

// Relay Pin

const int relayPin = 7; // Digital pin connected to the relay

// MQTT Client

WiFiClient espClient;

PubSubClient client(espClient);

// Threshold for fault detection

const float currentThreshold = 0.5; // 0.5A (this can be adjusted based on requirements)

void setup() {

// Initialize Serial, WiFi, LCD and MQTT

Serial.begin(115200);

lcd.begin(16, 2); // Set up the LCD size

pinMode(relayPin, OUTPUT);

digitalWrite(relayPin, LOW); // Initialize relay to off

// Connect to WiFi

WiFi.begin(ssid, password);

while (WiFi.status() != WL\_CONNECTED) {

delay(1000);

Serial.println("Connecting to WiFi...");

}

Serial.println("Connected to WiFi");

// Connect to MQTT

client.setServer(mqttServer, mqttPort);

client.connect("OpenCircuitDetector");

}

void loop() {

if (!client.connected()) {

client.connect("OpenCircuitDetector");

}

client.loop();

float currentReading = readCurrent(); // Read current from sensor

Serial.print("Current: ");

Serial.println(currentReading);

// Check for open circuit fault

if (currentReading < currentThreshold) {

lcd.setCursor(0, 0);

lcd.print("Open Circuit Fault");

digitalWrite(relayPin, HIGH); // Activate relay for fault

client.publish(mqttTopic, "Fault Detected"); // Publish to MQTT

} else {

lcd.setCursor(0, 0);

lcd.print("System Normal ");

digitalWrite(relayPin, LOW); // Deactivate relay

client.publish(mqttTopic, "System Normal"); // Publish to MQTT

}

delay(2000); // Delay before the next reading

}

// Function to read current from ACS712 sensor

float readCurrent() {

int sensorValue = analogRead(currentSensorPin);

float voltage = (sensorValue / 1023.0) \* voltageSupply; // Convert ADC value to voltage

float current = (voltage - (voltageSupply / 2)) / sensitivity; // Convert voltage to current

return current;

}

**Explanation of the Code:**

1. **Libraries**: The code includes libraries for the LCD, WiFi connectivity, and MQTT functionality.
2. **WiFi Setup**: Replace Your\_SSID and Your\_PASSWORD with your actual WiFi credentials. Set up the MQTT broker details to publish status updates.
3. **Current Sensor Configuration**: The code reads current from the ACS712 sensor. Adjust sensitivity based on the model you're using (5A, 20A, etc.).
4. **LCD Display**: It shows whether the system is in a normal state or if an open circuit fault is detected.
5. **Relay Activation**: If a fault is detected (current below the threshold), it activates a relay, which could be used to trigger an alarm or other actions.
6. **MQTT Publishing**: It sends updates on the current system status to an MQTT broker.

**Important Notes:**

* Make sure to wire your components according to the pin definitions in the code.
* Test the code with a physical setup to validate the performance and accuracy of fault detection.
* Adjust the currentThreshold as needed based on the characteristics of the loads in your application.

**Additional Components:**

If you have additional sensors or features (like a potentiometer for calibration), you'll need to integrate those into this code structure as needed. Each additional feature may require additional logic and sensor readings.

**CODE WITHOUT BUZZER AND CT’S**

#include <LiquidCrystal.h> // Include the library for LCD

#include <ESP8266WiFi.h> // Include library for WiFi

#include <PubSubClient.h> // Include library for MQTT

// WiFi and MQTT Setup

const char\* ssid = "Your\_SSID"; // Replace with your SSID

const char\* password = "Your\_PASSWORD"; // Replace with your WiFi password

const char\* mqttServer = "your.mqtt.broker"; // Replace with your MQTT broker address

const int mqttPort = 1883; // MQTT broker port

const char\* mqttTopic = "openCircuitStatus"; // MQTT topic to publish

// Current Sensor Pin

const int currentSensorPin = A0; // Analog pin connected to the current sensor

const float sensitivity = 0.185; // Sensitivity for ACS712 5A model (adjust based on your model)

const float voltageSupply = 5.0; // Supply voltage for the current sensor

// LCD Pins

LiquidCrystal lcd(12, 11, 5, 4, 3, 2); // Adjust according to your wiring

// Threshold for fault detection

const float currentThreshold = 0.5; // 0.5A (this can be adjusted based on requirements)

// MQTT Client

WiFiClient espClient;

PubSubClient client(espClient);

void setup() {

// Initialize Serial, WiFi, LCD and MQTT

Serial.begin(115200);

lcd.begin(16, 2); // Set up the LCD size

// Connect to WiFi

WiFi.begin(ssid, password);

while (WiFi.status() != WL\_CONNECTED) {

delay(1000);

Serial.println("Connecting to WiFi...");

}

Serial.println("Connected to WiFi");

// Connect to MQTT

client.setServer(mqttServer, mqttPort);

client.connect("OpenCircuitDetector");

}

void loop() {

if (!client.connected()) {

client.connect("OpenCircuitDetector");

}

client.loop();

float currentReading = readCurrent(); // Read current from sensor

Serial.print("Current: ");

Serial.println(currentReading);

// Check for open circuit fault

if (currentReading < currentThreshold) {

lcd.setCursor(0, 0);

lcd.print("Open Circuit Fault");

client.publish(mqttTopic, "Fault Detected"); // Publish to MQTT

} else {

lcd.setCursor(0, 0);

lcd.print("System Normal ");

client.publish(mqttTopic, "System Normal"); // Publish to MQTT

}

delay(2000); // Delay before the next reading

}

// Function to read current from ACS712 sensor

float readCurrent() {

int sensorValue = analogRead(currentSensorPin);

float voltage = (sensorValue / 1023.0) \* voltageSupply; // Convert ADC value to voltage

float current = (voltage - (voltageSupply / 2)) / sensitivity; // Convert voltage to current

return current;

}

**Key Changes and Considerations:**

1. **Removed Buzzer and Transformer**: The code no longer includes any components related to audible alerts (buzzer) or power transformation, focusing solely on current detection and status reporting.
2. **LCD for Local Feedback**: The LCD will still provide local feedback regarding the system's status, indicating whether an open circuit fault has been detected.
3. **Low Current Detection**: The current Threshold is set to 0.5A, making it suitable for your requirement to detect low fault currents that might occur when conductors fall on high-resistance surfaces.
4. **MQTT Communication**: The device continues to publish its status to an MQTT broker, which can be monitored remotely.
5. **Installation on Electric Poles**: Ensure that the device is properly sealed and rated for outdoor use, especially since it will be mounted on electric poles exposed to various weather conditions.

**Additional Hardware Requirements:**

* **Current Sensor**: Use an appropriate current sensor (like ACS712) to accurately read low currents.
* **Microcontroller**: An ESP8266 or similar IoT module is recommended for WiFi capabilities.
* **LCD Display**: A simple 16x2 LCD for displaying status messages.
* **Power Supply**: Make sure to provide a reliable power source suitable for outdoor use.

**Additional Concepts for Success:**

To ensure the success of your project at the hackathon and beyond, consider integrating the following concepts:

1. **Data Logging and Analysis**: Store historical data for further analysis to improve the fault detection algorithm.
2. **Alert Mechanisms**: Although you removed the buzzer, consider incorporating notification systems through MQTT or another protocol to alert maintenance teams in real-time.
3. **Robust Communication**: Ensure that the MQTT connection is reliable, with fallback mechanisms in case of network interruptions.
4. **Calibration and Testing**: Properly calibrate the current sensor to ensure accurate readings, and conduct extensive field tests to validate performance under various conditions.
5. **Enclosure Design**: Design a protective enclosure for the electronics to safeguard against weather and environmental factors.
6. **User Interface**: If feasible, consider a mobile or web application to visualize the data and receive alerts for better monitoring and management.