International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 11 | Nov - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

IOT-Based Real-Time Industrial Electrical Monitoring and Self-Healing System

Under the guidance of
Prof Deepalaxmi R Shenvi
Assistant Professor
Aditya College of Engineering and Technology
Bangalore, India

Rachana C N
Department of Electronics and Computer Engineering
Aditya College of Engineering and Technology
Bangalore, India
rachanagowda341@gmail.com

Sneha S
Department of Electronics and Computer Engineering
Aditya College of Engineering and Technology
Bangalore, India snehachinnu2426@gmail.com

Abstract- Industrial electrical loads are exposed to overload, abnormal voltage and overheating. Conventional protection devices such as fuses and circuit breakers only disconnect the supply after a fault has occurred. They show no real-time readings, keep no record of the fault, and need a manual reset. This leads to damaged equipment, longer downtime and safety risks.

The IoT-Based Real-Time Industrial Electrical Monitoring and Self-Healing System uses an ESP32 microcontroller as the main controller. A voltage sensormodule measures voltage, an ACS712 sensor measures current, and a DS18B20 sensor measures temperature. A 2-channel relay module switches two loads, which are selected by two toggle switches: Load 1 (motor) is the normal load and Load 2 (fan) is the fault load. A buzzer gives an audible alarm. The readings are shown on a 16x2 I2C LCD and sent over Wi-Fi to a web dashboard every 5 seconds.

The normal load runs on its own without any fault. When the fault load is switched on while the normal load is running, the system treats it as an overload and sounds the buzzer. It lets both loads run for a short 1-second window to capture the peak current, records it as the trip current, and then switches both relays OFF. When the user turns one switch OFF, the system clears the trip and runs the remaining load again automatically, without any manual reset. This is the self-healing action.

The system also cuts both loads and sounds the buzzer if the temperature rises above 50 °C, and restores them when it falls below 48 °C. The system gives a low-cost, wireless way to monitor, protect and recover electrical loads, and supports small-scale industrial automation and educational projects.

Keywords: IoT, ESP32, electrical monitoring, overload protection, self-healing, over-temperature protection, buzzer, ACS712 current sensor, DS18B20, relay, web dashboard.

I. INTRODUCTION

Reliable electrical supply is essential in industries, workshops and laboratories. Electrical loads run for long hours, and a fault can stop production and damage costly equipment. Overload current heats the windings and insulation, voltage sag reduces efficiency, and high temperature shortens equipment life. Detecting these conditions early is important for safety and productivity.

Traditional protection depends on fuses, thermal relays and circuit breakers. They cut the supply reliably, but they do not show live voltage, current or temperature, they do not record the current that caused the trip, and the user must be present to find the fault and reset the device. Wireless monitoring through IoT removes the need for constant physical presence.

This project combines measurement, automatic protection, automatic recovery and remote monitoring in one unit. The ESP32 is used as the controller because it has built-in Wi-Fi, a dual-core processor, a 12-bit ADC and enough pins for sensors, relays and a display. The networking runs as a separate task on the second core, so

a slow Wi-Fi connection never delays the protection logic. A buzzer and an over-temperature limit add an audible alarm and thermal protection.

II. LITERATURE SURVEY

2.1 Literature Survey

Electrical monitoring and protection systems have developed from simple mechanical devices to IoT-based smart systems. The survey below follows this development and looks at the protection devices, sensors, controllers and communication methods that the proposed system builds on.

Conventional devices protect the load but give no data. Sensor-based and microcontroller-based projects add measurement, but most work only locally. IoT systems add remote visibility, but many of them only report and do not act on a fault. This leaves a gap for a low-cost unit that measures, protects, recovers and reports together.

2.2.1 Conventional Protection

Fuses, miniature circuit breakers (MCBs) and thermal overload relays disconnect the supply on over-current. They are simple and reliable, but they give no data, do not record the fault current and need a manual reset.

2.2.2 Current Sensing with ACS712

Source: Allegro MicroSystems, ACS712 datasheet [2]

Hall-effect sensors such as the ACS712 give an output voltage that changes with the current flowing through them. In the 5 A version the output sits at about half the supply voltage with no current and changes by 185 mV per ampere. The sensor is low in cost and gives electrical isolation, but it is less accurate at very small currents.

2.2.3 Temperature Sensing with DS18B20

Source: Maxim Integrated, DS18B20 datasheet [3]

The DS18B20 is a digital temperature sensor that gives readings in degrees Celsius over a single data line. It needs only one pin and gives stable readings, which makes it suitable for monitoring equipment temperature.

2.2.4 Microcontroller-Based Monitoring

Arduino-based projects measure voltage and current and show them on an LCD. They are low in cost, but they work only locally and have no remote access.

2.2.5 ESP32 Microcontroller

Source: Espressif Systems, ESP32 datasheet and technical reference manual [1], Arduino-ESP32 core and FreeRTOS documentation [5]

The ESP32 has built-in Wi-Fi and is widely used for low-cost IoT projects. Its two cores allow communication and control to run at the same time, which the proposed system uses to keep protection independent of the network.

2.2.6 IoT-Based Monitoring

ESP8266 and ESP32 systems upload sensor values to cloud platforms or custom servers and give remote visibility. Many of them only monitor and do not act on faults.

2.2.7 SCADA Systems

SCADA offers complete industrial monitoring and control, but it is expensive and complex, so it is unsuitable for small industries and teaching.

2.2.8 Over-Temperature Protection and Alarm

Thermal protection normally uses a thermal relay that opens at a fixed temperature and gives no reading. With a digital sensor such as the DS18B20, a limit can be set in software and a hysteresis can be added, so the load does not switch on and off repeatedly near the limit. A buzzer is a simple output device that gives an immediate audible warning to the operator.

2.2.9 Proposed System

The proposed system combines an ESP32, voltage, current and temperature sensors, a relay module, an LCD and a web dashboard. It trips the loads on overload, records the trip current, and recovers automatically when the overload is removed. It also sounds a buzzer and cuts the loads on over-temperature.

III. METHODOLOGY

The system uses an ESP32 to read the sensors, apply the fault logic, control the relays and send data to the web dashboard.

1. **Start-up:** The ESP32 switches both relays OFF, starts the LCD and the temperature sensor, and connects to Wi-Fi. It waits up to 8 seconds for the connection and shows WiFi OK or WiFi FAIL.
2. **Measurement:** The voltage and the ACS712 output are each read many times and averaged. A smoothing filter then reduces noise, as shown in the formulas below.
3. **Voltage drop:** While the loads are OFF, the ESP32 learns the no-load voltage. When a load is ON, the drop is the no-load voltage minus the present voltage.
4. **Fault detection:** The two toggle switches are read continuously. Load 1 (motor) is the normal load and Load 2 (fan) is the fault load. Either switch alone runs its load, which is the NORMAL condition. Switching the fault load ON while the normal load is ON adds the two

currents and gives an OVERLOAD, and the buzzer sounds.

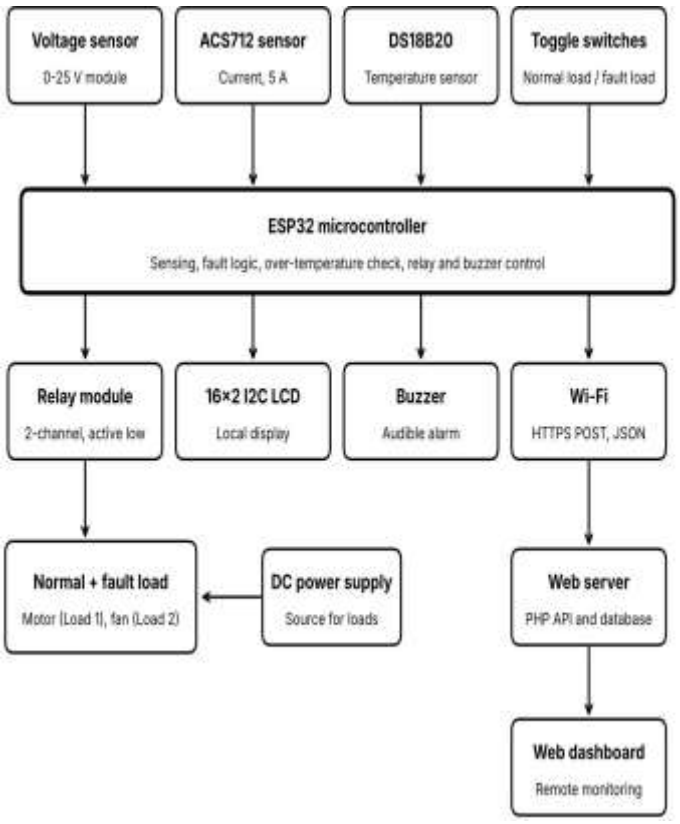
5. **Peak capture and trip:** On overload both loads run for a 1-second window while the current is sampled every 100 ms. The highest value is stored as the trip current, and both relays are switched OFF.
6. **Self-healing:** When one switch is turned OFF, the overload and the trip are cleared, the buzzer stops and the remaining load runs again automatically.
7. **Over-temperature protection:** The temperature is compared with a limit of 50 °C. Above it, both loads are switched OFF and the buzzer sounds, whatever the switch positions. A hysteresis of 2 °C is used, so the loads can run again only when the temperature falls below 48 °C.
8. **Display:** The LCD shows voltage and current on the first line, and temperature, active load and status on the second line. While tripped, the second line alternates between OVERLOAD and the trip current, and during an over-temperature fault it alternates between OVER TEMP and the limit.
9. **Remote reporting:** A separate task on the second core sends a JSON packet by HTTPS POST to the PHP web API every 5 seconds. Certificate checking is switched off with setInsecure(), which is acceptable for a project demonstration. If Wi-Fi is lost, the task reconnects without affecting protection.

Voltage (V) = V(pin) × module ratio (5.0)
Current (A) = Change in sensor output (V) / 0.185
Filtered value = (1 − α) × previous value + α × new value

Here 5.0 is the ratio of the voltage module, 0.185 V/A is the sensitivity of the 5 A ACS712, and α is 0.5 for voltage and 0.4 for current.

Overall working principle: The ESP32 measures voltage, current and temperature and watches the two switches. One load at a time runs normally. When the fault load is added to the normal load, the buzzer sounds and the system captures the peak current, trips both loads and records the value. When one switch is turned OFF, it restores the remaining load by itself. If the temperature crosses 50 °C, both loads are cut and the buzzer sounds until it falls below 48 °C. Every reading is shown on the LCD and sent to the web dashboard.

IV. Block Diagram with Explanation



The ESP32 sits at the centre of the system. The sensors and switches feed it, and it drives the relays, the LCD and the Wi-Fi link.

- **Voltage sensor (0-25 V):** A resistor-divider module that scales the input down for the ESP32 ADC.
- **ACS712 (5 A):** A Hall-effect sensor that gives an output voltage proportional to the load current.
- **DS18B20:** A digital sensor that gives the temperature over a single data line.
- **Toggle switches:** Switch 1 turns ON the normal load (motor, Load 1) and switch 2 turns ON the fault load (fan, Load 2).
- **ESP32 microcontroller:** Reads the sensors, applies the fault logic and the over-temperature check, controls the relays and the buzzer, updates the LCD and sends data to the server.
- **Buzzer:** Gives an audible alarm during an overload or an over-temperature fault.
- **Relay module (2-channel, active low):** Switches the normal load (motor) and the fault load (fan). A LOW signal turns a relay ON.

- **16x2 I2C LCD:** Shows the live readings and the fault status locally using two wires (SDA and SCL).
- **Wi-Fi, web server and dashboard:** The ESP32 sends JSON data by HTTPS POST to a PHP API and database, and the web dashboard shows it for remote monitoring.

DC power supply: Supplies the loads through the relays. The ESP32 is powered through USB or a regulated supply, and all grounds are joine

V. Flow Charts

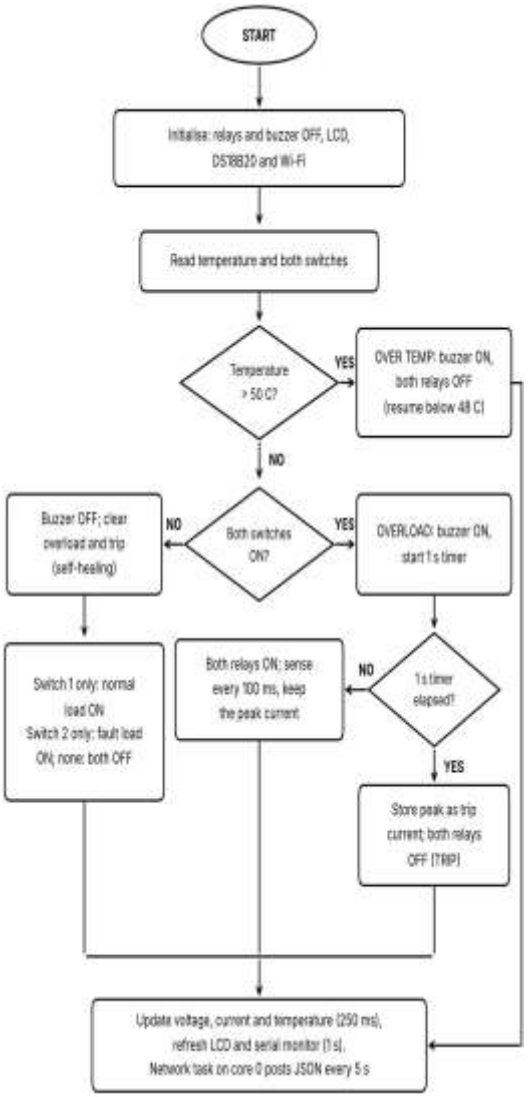


Figure 3.2: Program flow of the monitoring and self-healing logic

After start-up, the program checks the temperature first. If it is above 50 °C, the buzzer sounds and both loads are cut. Otherwise it reads the two switches. If both are ON, the buzzer sounds, a 1-second timer starts, the peak current is captured, and both loads are tripped. If not, the buzzer is switched OFF, any trip is cleared and only the selected load runs. The measurements, LCD and network task run alongside this loop.

VI. Algorithm for Measurement

1. Read the voltage pin many times and take the average.
2. Multiply the average by the module ratio (5.0) to get the voltage.
3. Read the ACS712 pin many times and take the average.
4. Calculate the current from the change in the sensor output, using 185 mV per ampere.
5. Apply the smoothing filter to the voltage and the current.
6. Learn the no-load voltage while the loads are OFF, and calculate the voltage drop while a load is ON.

Algorithm for Fault Detection, Alarm and Self-Healing

1. Read the temperature. If it is above 50 °C and the over-temperature flag is not set, set the flag. If the flag is set and the temperature is below 48 °C, clear it.
2. If the over-temperature flag is set, set the status to OVERTEMP, switch the buzzer ON, clear the overload and trip flags, switch both relays OFF and end this cycle.
3. Read switch 1 (normal load) and switch 2 (fault load). A LOW input means the switch is ON.
4. If both are ON, set the status to OVERLOAD and switch the buzzer ON. If this is the first time, record the start time and reset the peak current.
5. While the trip is not yet set and less than 1 second has passed, switch both relays ON and keep the highest current seen.

6. After 1 second, store the peak as the trip current, set the trip flag and switch both relays OFF. Keep both relays OFF while the trip flag is set.
7. If both are not ON, switch the buzzer OFF and clear the overload and trip flags.
8. If only switch 1 is ON, switch the normal load ON and set the status to NORMAL. If only switch 2 is ON, switch the fault load ON and set the status to NORMAL. If neither is ON, switch both relays OFF.

Algorithm for Data Upload

1. Create a task on core 0 at start-up.
2. Wait 5 seconds. If Wi-Fi is not connected, request a reconnect and wait again.
3. Build a JSON packet with voltage, current, temperature, fault status, relay and load states, trip status and trip current.
4. Send the packet to the web API by HTTPS POST through a secure client (certificate checking disabled) with a 4-second timeout, and print the response code.
5. Repeat from step 2.

Step-by-Step Process

1. The system is powered on. Both relays and the buzzer stay OFF, and the LCD shows Load Monitor and then the Wi-Fi result.
2. The user turns switch 1 ON. The normal load (motor) runs, and the LCD shows the readings with status NORMAL.
3. The user turns switch 1 OFF and switch 2 ON. The fault load (fan) runs alone, and the status is still NORMAL.
4. With the normal load running, the user turns the fault load ON. The ESP32 detects an OVERLOAD, the buzzer sounds, and both loads run for 1 second while the peak current is captured.
5. The ESP32 stores the peak as the trip current and switches both relays OFF. The LCD alternates between OVERLOAD and the trip current.
6. The user turns one switch OFF. The trip is cleared, the buzzer stops and the remaining load starts again by itself.
7. If the temperature rises above 50 °C at any time, both loads are cut, the buzzer sounds and the LCD alternates between OVER TEMP and LIMIT 50C.
8. When the temperature falls below 48 °C, the fault clears and the loads follow the switches again.
9. Every 5 seconds the readings are sent to the web dashboard, where they can be viewed remotely.

ESP32 Program

The complete program is given below. The Wi-Fi name, password and server address are replaced by placeholders and must be filled in before use.

```
// ESP32 Load Monitor - with overload buzzer + temperature threshold
// Required libraries (Library Manager):
//    - OneWire by Jim Studt / Paul Stoffregen (2.3.8 or newer)
//     - DallasTemperature by Miles Burton (latest)
//     - LiquidCrystal I2C by Frank de Brabander (or johnrickman fork)
// Board: ESP32 Arduino core 3.x

#include <WiFi.h>
#include <WiFiClientSecure.h>
#include <HTTPClient.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
#include <OneWire.h>
#include <DallasTemperature.h>

#define DEMO_MODE false
const unsigned long TRIP_DELAY_MS = 1000;
#define USE_ESTIMATE_FOR_SMALL_CURRENT true
const float MEAS_MIN_A  = 0.10;
const float MOTOR_EST_A = 0.06;
const float FAN_EST_A   = 0.06;

//    ----------    TEMPERATURE THRESHOLD (set here) ----------
```

```
const float TEMP_THRESHOLD_C = 50.0;     // over-temperature limit in deg C
const float TEMP_HYST_C    = 2.0;  // resets when temp < threshold - hysteresis

// ---------- WiFi & API ----------
const char*   WIFI_SSID    = "YOUR_WIFI_SSID";
const char*   WIFI_PASS    = "YOUR_WIFI_PASSWORD";
const char* API_URL        = "https://your-server/short-circuit/api.php";

// ---------- Pins ----------
#define VOLT_PIN      34
#define CURR_PIN      35
#define TEMP_PIN      5
#define SW_MOTOR      32
#define SW_FAN        33
#define RELAY_MOTOR   26
#define RELAY_FAN     27
#define BUZZER_PIN    25

#define RELAY_ON      LOW
#define RELAY_OFF     HIGH

#define BUZZER_ON    HIGH    // change to LOW if your buzzer module is active-low
#define BUZZER_OFF    LOW

const float VOLT_MODULE_RATIO = 5.0;
float VOLT_CAL = 1.0;
const float ACS_SENS = 0.185;
const float DIVIDER_FACTOR = 1.0;

float acsOffsetV = 2.5;
float acsOffsetStart = 2.5;

const float V_ALPHA = 0.5;
const float I_ALPHA = 0.4;

const float DEMO_BATT_V      = 12.6;
const float DEMO_SERIES_R    = 1.5;
const float DEMO_MOTOR_A     = 1.4;
const float DEMO_FAN_A       = 0.7;
const float DEMO_OVERLOAD_A  = 3.2;

LiquidCrystal_I2C lcd(0x27, 16, 2);

OneWire oneWire(TEMP_PIN);
DallasTemperature tempSensor(&oneWire);

volatile float voltage = 0, current = 0, currentMeasured = 0, temperature = 0;
volatile bool currentEst = false;
const char* faultStatus = " ";
volatile bool motorOn = false, fanOn = false;

volatile float vNoLoad = 0, vDrop = 0;
unsigned long loadsOffSince = 0;

volatile bool overloadActive = false;
volatile bool tripped = false;
volatile bool everTripped = false;
volatile bool tripEst = false;
volatile float tripCurrent = 0;
volatile float peakRaw = 0;
unsigned long overloadStart = 0;

volatile bool overTemp = false;
// NEW: over-temperature flag

bool tempPending = false;
bool firstSample = true;
unsigned long lastLcd = 0, lastTempReq = 0, lastSense = 0;

// ---------- Sensors ----------
float readPinVolts(int pin, int n, int us) {
  long sum = 0;
  for (int i = 0; i < n; i++) {
    sum += analogReadMilliVolts(pin);
    delayMicroseconds(us);
  }
  return (sum / (float)n) / 1000.0;
}
```

```
float readVoltageRaw() {
  float v = readPinVolts(VOLT_PIN, 300, 100) * VOLT_MODULE_RATIO * VOLT_CAL;
  if (v < 0.5) return 0;
  return v;
}

float readCurrentSensorV() {
  return readPinVolts(CURR_PIN, 400, 100) * DIVIDER_FACTOR;
}

void readTemperature() {
  unsigned long now = millis();
  if (!tempPending && now - lastTempReq >= 2000) {
    tempSensor.requestTemperatures();
    lastTempReq = now;
    tempPending = true;
  } else if (tempPending && now - lastTempReq >= 800) {
    float t = tempSensor.getTempCByIndex(0);
    if (t > -100) temperature = t;
    tempPending = false;
  }
}

// ---------- Buzzer ----------
void setBuzzer(bool on) {
  digitalWrite(BUZZER_PIN, on ? BUZZER_ON : BUZZER_OFF);
}

// ---------- Relays and fault logic ----------
void applyRelays(bool motor, bool fan) {
  digitalWrite(RELAY_MOTOR, motor ? RELAY_ON : RELAY_OFF);
  digitalWrite(RELAY_FAN, fan ? RELAY_ON : RELAY_OFF);
  motorOn = motor;
  fanOn = fan;
}

void recordTrip() {
  if (peakRaw >= MEAS_MIN_A) {
    tripCurrent = peakRaw;
    tripEst = false;
  } else {
    tripCurrent = MOTOR_EST_A + FAN_EST_A;
    tripEst = true;
  }
  everTripped = true;
  Serial.printf("TRIP! current at trip = %.2f A (%s)\n", (float)tripCurrent, tripEst ? "est" : "meas");
}

void evaluateFault() {
  bool sw1 = (digitalRead(SW_MOTOR) == LOW);
  bool sw2 = (digitalRead(SW_FAN) == LOW);

  // ---------------Over-temperature check (with hysteresis) -----
  if (!overTemp && temperature > TEMP_THRESHOLD_C) {
    overTemp = true;
    Serial.printf("OVER TEMP! %.1f C > %.1f C\n", (float)temperature, TEMP_THRESHOLD_C);
  } else if (overTemp && temperature < (TEMP_THRESHOLD_C - TEMP_HYST_C)) {
    overTemp = false;

Serial.println("Temperature back to normal");
  }

  if (overTemp) {
    faultStatus = "OVERTEMP";
    setBuzzer(true);
    overloadActive = false;
    tripped = false;
    applyRelays(false, false);
// cut both loads
    return;
  }

  // ----- Overload check ----
  if (sw1 && sw2) {
    faultStatus = "OVERLOAD";
    setBuzzer(true);
// buzzer ON during overload
    if (!overloadActive) {
      overloadActive = true;
      tripped = false;
      peakRaw = 0;
```

```
      overloadStart = millis();
    }
    if (!tripped) {
      if      (millis()      -
overloadStart >= TRIP_DELAY_MS)
{
        recordTrip();
        tripped = true;
        applyRelays(false,
false);
      } else {
        applyRelays(true,
true);
      }
    } else {
      applyRelays(false,
false);
    }
  } else {
    setBuzzer(false);
// buzzer OFF when everything
is normal
    overloadActive = false;
    tripped = false;
    if (sw1) {
      applyRelays(true, false);
      faultStatus = "NORMAL";
    } else if (sw2) {
      applyRelays(false, true);
      faultStatus = "NORMAL";
    } else {
      applyRelays(false,
false);
      faultStatus = " ";
    }
  }
}

//    ----------    Measurement
update ----------
void updateMeasurements() {
  float vNew, iShown, iMeas;
  bool est = false;

#if DEMO_MODE
  float load = 0;
  if           (overloadActive
&&     !tripped)     load     =
DEMO_OVERLOAD_A;
  else       if       (tripped)
load = 0;
  else       if       (motorOn)
load = DEMO_MOTOR_A;
  else        if        (fanOn)
load = DEMO_FAN_A;
  iMeas = load + (load > 0 ?
random(-5, 6) / 100.0 : 0);
  iShown = iMeas;
  vNew = DEMO_BATT_V - iMeas *
DEMO_SERIES_R + random(-3, 4)
/ 100.0;
#else
  vNew = readVoltageRaw();

  float          cv          =
readCurrentSensorV();

  if (!motorOn && !fanOn) {
    acsOffsetV += 0.01f * (cv
- acsOffsetV);
    acsOffsetV                =
constrain(acsOffsetV,
acsOffsetStart     -     0.05f,
acsOffsetStart + 0.05f);
  }
  iMeas = fabs(cv - acsOffsetV)
/ ACS_SENS;
  if (iMeas < 0.01) iMeas = 0;
  iShown = iMeas;

#if
USE_ESTIMATE_FOR_SMALL_CURRENT
  if (iMeas < MEAS_MIN_A) {
    float   e   =   (motorOn   ?
MOTOR_EST_A  :  0)  +  (fanOn  ?
FAN_EST_A : 0);
    if (e > 0) { iShown = e;
est = true; }
  }
#endif
#endif

  if           (overloadActive
&& !tripped && iMeas > peakRaw)
peakRaw = iMeas;

  if (firstSample) {
    voltage = vNew;
    current = iShown;
    currentMeasured = iMeas;
    firstSample = false;
  } else {
    voltage          = voltage
* (1.0f - V_ALPHA) + vNew    *
V_ALPHA;
    current          = current
* (1.0f - I_ALPHA) + iShown *
I_ALPHA;
    currentMeasured          =
currentMeasured   *   (1.0f   -
I_ALPHA) + iMeas * I_ALPHA;
    if (current < 0.02) current
= 0;
```

```
    if (currentMeasured < 0.01)
currentMeasured = 0;
  }
  currentEst = est;

  if (!motorOn && !fanOn) {
    if (loadsOffSince == 0)
loadsOffSince = millis();
    if       (millis()       -
loadsOffSince > 1500) {
      vNoLoad = (vNoLoad ==
0) ? voltage : vNoLoad * 0.9f
+ voltage * 0.1f;
    }
    vDrop = 0;
  } else {
    loadsOffSince = 0;
    vDrop = vNoLoad - voltage;
    if (vNoLoad == 0 || vDrop
< 0) vDrop = 0;
  }
}

// ---------------- LCD ----------------
void lcdLine(int row, const
char* text) {
  char buf[17];
  snprintf(buf,    sizeof(buf),
"%-16s", text);
  lcd.setCursor(0, row);
  lcd.print(buf);
}

void updateLCD() {
  char l1[24], l2[24];
  static bool toggle = false;
  toggle = !toggle;
  const char* load = motorOn ?
"MTR" : (fanOn ? "FAN" : "---
");

#if DEMO_MODE
  snprintf(l1,      sizeof(l1),
"V%.2f       I%.2f       DEMO",
(float)voltage,
(float)current);
#else
  snprintf(l1,      sizeof(l1),
"V%.2fV              I%s%.2fA",
(float)voltage,   currentEst   ?
"*" : "", (float)current);
#endif

  if (overTemp) {
    if (toggle) snprintf(l2,
sizeof(l2), "%.1fC OVER TEMP",
(float)temperature);
    else          snprintf(l2,
sizeof(l2),    "LIMIT    %.0fC",
TEMP_THRESHOLD_C);
  } else if (tripped) {
    if  (toggle)  snprintf(l2,
sizeof(l2), "%.1fC OVERLOAD",
(float)temperature);
    else          snprintf(l2,
sizeof(l2),  "TRIP  I%s%.2fA",
tripEst    ?    "*"    :    "",
(float)tripCurrent);
  }          else          if
(strcmp(faultStatus,
"OVERLOAD") == 0) {
    snprintf(l2,    sizeof(l2),
"%.1fC               OVERLOAD",
(float)temperature);
  } else {
    snprintf(l2,    sizeof(l2),
"%.1fC         %s         %s",
(float)temperature,       load,
faultStatus);
  }

  lcdLine(0, l1);
  lcdLine(1, l2);
}

// ---------- API task -------
---
void apiTask(void* pv) {
  for (;;) {
    vTaskDelay(5000          /
portTICK_PERIOD_MS);
    if    (WiFi.status()    !=
WL_CONNECTED) {
      WiFi.reconnect();
      continue;
    }

    char json[480];
    snprintf(json,
sizeof(json),

"{\"voltage\":%.2f,\"current\":
%.2f,\"current_measured\":%.3f,
"

"\"current_source\":\"%s\",\"t
emperature\":%.1f,"

"\"fault_status\":\"%s\",\"rel
ay\":\"%s\",\"motor\":\"%s\",\
"fan\":\"%s\","

"\"tripped\":\"%s\",\"trip_cur
rent\":%.2f,\"trip_source\":\"
%s\"%s}",
```

```
      (float)voltage,
(float)current,
(float)currentMeasured,
      currentEst   ?   "EST"   :
"MEAS",
      (float)temperature,
faultStatus,
      (motorOn   ||   fanOn)   ?
"ON" : "OFF",
      motorOn ? "ON" : "OFF",
      fanOn ? "ON" : "OFF",
      tripped ? "YES" : "NO",
      (float)tripCurrent,
      !everTripped ? "NONE" :
(tripEst ? "EST" : "MEAS"),
      DEMO_MODE                ?
",\"mode\":\"DEMO\"" : "");

    // HTTPS needs a secure
client on ESP32.
    //   setInsecure()   skips
certificate checking (fine for
a project/demo).
    WiFiClientSecure client;
    client.setInsecure();

    HTTPClient http;
    if     (http.begin(client,
API_URL)) {
      http.setTimeout(4000);
      http.addHeader("Content-
Type", "application/json");
      int        code        =
http.POST(json);
      Serial.print("API
response: ");
      Serial.println(code);
      http.end();
    } else {
      Serial.println("HTTP
begin failed");
    }
  }
}

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

  pinMode(RELAY_MOTOR, OUTPUT);
  pinMode(RELAY_FAN, OUTPUT);
  digitalWrite(RELAY_MOTOR,
RELAY_OFF);
  digitalWrite(RELAY_FAN,
RELAY_OFF);

  pinMode(BUZZER_PIN, OUTPUT);
  setBuzzer(false);
  pinMode(SW_MOTOR,
INPUT_PULLUP);
  pinMode(SW_FAN,
INPUT_PULLUP);

  analogReadResolution(12);

analogSetAttenuation(ADC_11db);

  lcd.init();
  lcd.backlight();
  lcdLine(0, "Load Monitor");
  lcdLine(1, DEMO_MODE ? "DEMO
MODE" : "Real readings");
  delay(1000);

  tempSensor.begin();

tempSensor.setWaitForConversio
n(false);
  Serial.printf("DS18B20
devices      found:      %d\n",
tempSensor.getDeviceCount());
  Serial.printf("Temp
threshold    =    %.1f    C\n",
TEMP_THRESHOLD_C);

  WiFi.mode(WIFI_STA);
  WiFi.setAutoReconnect(true);
  WiFi.begin(WIFI_SSID,
WIFI_PASS);
   unsigned long t0 = millis();
  while    (WiFi.status()    !=
WL_CONNECTED && millis() - t0
< 8000) delay(250);
  lcdLine(1, WiFi.status()  ==
WL_CONNECTED  ?  "WiFi  OK"  :
"WiFi FAIL");

  delay(500);
  acsOffsetV                    =
readCurrentSensorV();
  acsOffsetStart = acsOffsetV;
  Serial.printf("ACS712 offset
= %.3f V (should be near
2.5)\n", acsOffsetV);
  delay(1000);


xTaskCreatePinnedToCore(apiTas
k, "api", 12000, NULL, 1, NULL,
0);
}

void loop() {
  evaluateFault();
```

```
  readTemperature();

  unsigned long senseInterval
=              (overloadActive
&& !tripped) ? 100 : 250;
  if (millis() - lastSense >=
senseInterval) {
    lastSense = millis();
    updateMeasurements();
  }

  if (millis() - lastLcd > 1000)
{
    lastLcd = millis();
    updateLCD();
    Serial.printf("%sV=%.2f
Vdrop=%.2f I=%.2f%s (meas %.3f)
T=%.1f M=%d F=%d %s\n",
                  DEMO_MODE  ?
"[DEMO] " : "",

(float)voltage,  (float)vDrop,
(float)current,  currentEst  ?
"(est)" : "",

(float)currentMeasured,
(float)temperature,   motorOn,
fanOn, faultStatus);
  }
}
```

How the program works:

- **Settings:** DEMO_MODE selects simulated or real readings. TRIP_DELAY_MS sets the 1-second overload window. TEMP_THRESHOLD_C (50 °C) and TEMP_HYST_C (2 °C) set the over-temperature limit.
- **setup():** Switches both relays and the buzzer OFF, sets the switch pins with pull-ups, starts the LCD and the DS18B20, connects to Wi-Fi and starts the network task on core 0.
- **readPinVolts():** Averages many ADC readings in millivolts to reduce noise.
- **readTemperature():** Requests a temperature reading every 2 seconds and collects it 800 ms later, so the main loop is never blocked.
- **setBuzzer():** Switches the buzzer ON or OFF according to BUZZER_ON and BUZZER_OFF.
- **evaluateFault():** First checks the temperature against the limit with hysteresis and cuts both loads with the buzzer ON if it is exceeded. Otherwise it reads the switches, detects the overload, sounds the buzzer, runs the 1-second window, records the trip and applies the relays. It clears the trip and stops the buzzer when a switch is turned OFF.
- **recordTrip():** Stores the peak current as the trip current.
- **updateMeasurements():** Calculates voltage and current, applies the smoothing filter and calculates the voltage drop.
- **updateLCD():** Shows voltage and current on line 1, and temperature, load and status on line 2. During a trip, line 2 alternates between OVERLOAD and the trip current, and during an over-temperature fault it alternates between OVER TEMP and the limit.
- **apiTask():** Runs on core 0 and posts a JSON packet to the web API every 5 seconds through a secure HTTPS client (certificate checking disabled).
- **loop():** Runs the fault logic and temperature reading continuously, senses every 100 ms during the overload window and every 250 ms otherwise, and refreshes the LCD and serial monitor every second.

Discussion

The prototype shows that one ESP32 can measure, protect, recover and report at the same time. The following points are worth noting:

- **Trip current:** The recorded value is the highest current seen during the 1-second window, so it describes the overload that caused the trip.
- **Trip condition:** In this prototype the fault load is switched on together with the normal load to create an overload, and the program trips when both switches are ON. The fault load running alone is treated as NORMAL, because the trip is decided by the switch combination and not by a current limit. A real installation would also need current limits, and the unit should not replace certified protection devices such as MCBs.
- **ADC input range:** The ESP32 inputs are limited to 3.3 V. The ACS712 output can reach about 3.4 V at 5 A, and a 25 V module output reaches 5 V,

so a divider on the sensor outputs or lower operating ranges is advisable.

- **Timing:** The 100 ms sensing during the overload window gives about ten samples for peak capture, and the 250 ms interval is used at other times.
- **Network independence:** The upload task on core 0 has its own timeout of 4 seconds, so protection on the main core is never blocked by the network.
- **Over-temperature:** The 2 °C hysteresis stops the loads from switching on and off repeatedly near 50 °C. Both loads are cut whatever the switch positions, and they restart by themselves when the temperature falls below 48 °C. If the DS18B20 is disconnected, the last temperature value is kept, so the sensor wiring should be checked.
- **Buzzer:** The buzzer is ON for the whole time an overload or over-temperature fault is active. This includes the time after the trip, until a switch is turned OFF.
- **HTTPS upload:** The program uses an HTTPS client with certificate checking disabled (setInsecure). This is fine for a demonstration, but a real installation should verify the server certificate.
- **Security:** Wi-Fi credentials and the server address should be kept out of shared code.

1. and upcoming token numbers through LCD screens, helping visitors identify their service turn.
6. **Multi-Department Support:** Supports queue

Each LCD displays the department name, current token
a centralized application.

management for four different service departments through
7. **Efficient Staff Management:** Provides separate dashboards for authorized staff to handle registrations and queue activities.
8. **Reliable Data Storage:** Maintains visitor details, token records, department information, and service history using an SQLite database.
9. **IoT Integration:** Uses an ESP32 microcontroller and Wi-Fi connectivity to transfer updated token information to the LCD displays.
10. **Improved Communication:** Keeps visitors informed about queue progress through the web application and physical display units.

Conclusion

The IoT-Based Real-Time Industrial Electrical Monitoring and Self-Healing System is a low-cost system that combines measurement, automatic protection, automatic recovery and remote monitoring. The ESP32 reads the voltage, current and temperature sensors, controls the load relays and the buzzer, shows the readings on an LCD, and sends them to a web dashboard over Wi-Fi.

When the fault load causes an overload, the system sounds the buzzer, captures the peak current, records it as the trip current and switches the loads OFF. When the overload is removed, it restores operation automatically. If the temperature goes above 50 °C, it cuts both loads and restarts them below 48 °C. This reduces downtime and the need for manual attention. Running the network task on a separate core keeps the protection fast and independent of Wi-Fi.

The system shows how embedded systems, sensors, relay control and IoT communication can be integrated into a practical and affordable monitoring unit. In future work, it can be extended with AC voltage and power measurement, over-voltage and current limits, certificate-verified HTTPS, mobile alerts, data logging with graphs, and machine-learning-based predictive maintenance.

References

1. Espressif Systems, ESP32 Series Datasheet and ESP32 Technical Reference Manual, Espressif Systems, Shanghai, China.
2. Allegro MicroSystems, ACS712: Fully Integrated, Hall Effect-Based Linear Current Sensor IC (datasheet).
3. Maxim Integrated, DS18B20 Programmable Resolution 1-Wire Digital Thermometer (datasheet).
4. Arduino, Arduino IDE and Language Reference Documentation, arduino.cc.
5. Espressif Systems, Arduino-ESP32 Core and FreeRTOS Documentation.