Distributed Power Monitoring System with Multi-Node Sensors

Author: Noridel Herron

BS in Computer Engineering

Fall 2025

Introduction

Embedded power monitoring systems play a critical role in modern electrical distribution, industrial control, and energy management infrastructures. These systems continuously observe electrical conditions, detect abnormal behavior, and initiate protective responses to maintain safe and reliable operation. In real-world deployments, monitoring and protection functionality is often distributed across multiple embedded devices, where local sensing and decision-making are coordinated through a centralized supervisory controller.

Designing distributed embedded monitoring systems introduces several challenges. These include implementing accurate signal processing under constrained computational resources, coordinating fault detection at both local and system levels, and ensuring safe concurrency on centralized controllers. Additional design tradeoffs arise in determining where signal processing boundaries should be placed, specifically, whether nodes should transmit raw analog-to-digital converter (ADC) samples or preprocessed quantities such as RMS values, and how unit conversion and calibration should be managed across the system. Addressing these challenges is essential for developing scalable architectures that reflect the structure and behavior of deployed power monitoring systems.

This project explores the design of a distributed embedded power monitoring system using multiple ESP32 sensor nodes and a Raspberry Pi central controller, with emphasis on the accuracy and reliability of embedded signal processing and fault detection. The system adopts a hierarchical architecture in which sensor nodes perform local signal processing and immediate protection functions, while the central controller aggregates measurements and performs system-level analysis. To support development and validation without reliance on physical power hardware, the design incorporates a software-based waveform streaming mechanism that enables controlled fault injection while preserving identical signal-processing behavior to analog inputs.

A primary objective of this work is to demonstrate that embedded signal processing implemented on resource-constrained microcontrollers can achieve accuracy comparable to floating-point reference models, while maintaining reliable fault detection across a range of operating conditions. The remainder of this report describes the system architecture, node-level and controller-level implementation, experimental results and validation, and conclusion.

System Architecture

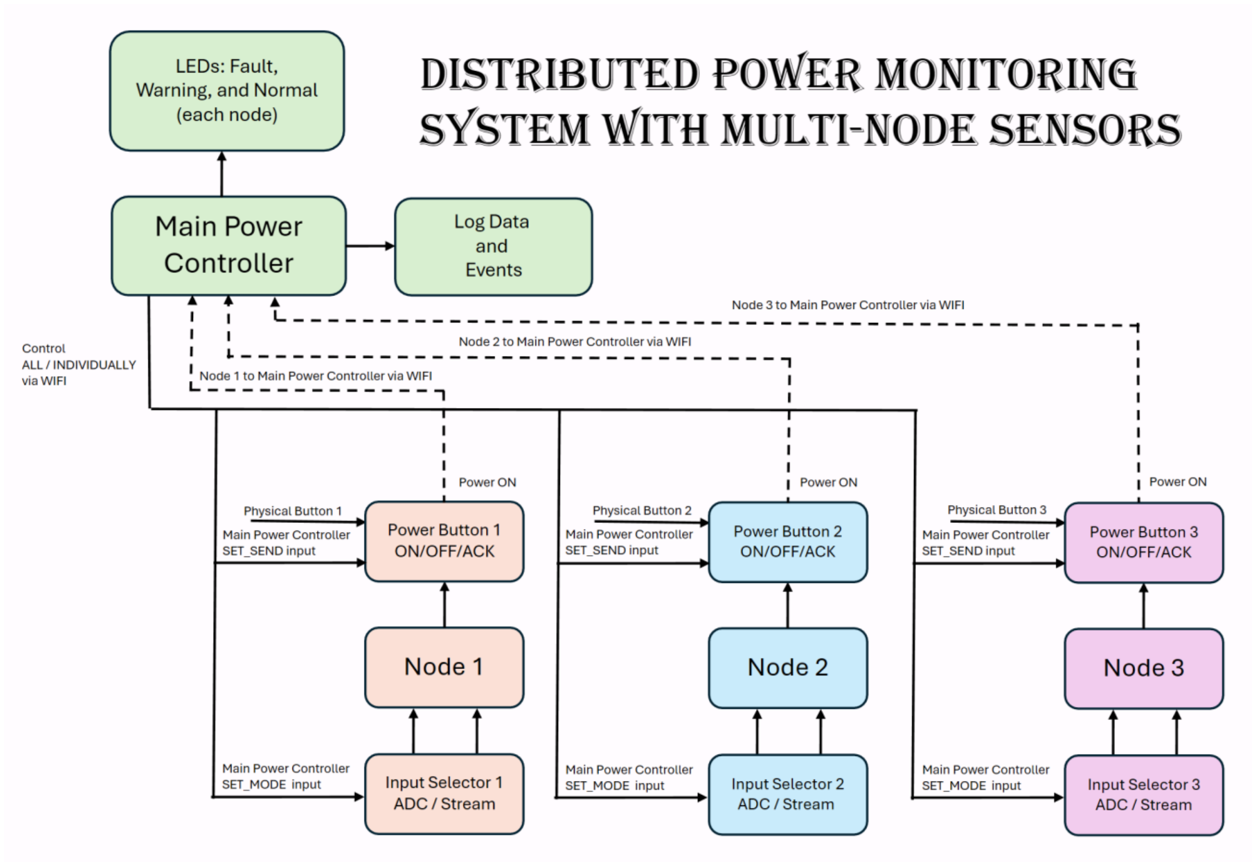

DISTRIBUTED POWER MONITORING SYSTEM WITH MULTI-NODE SENSORS
LEDs: Fault, Warning, and Normal (each node)
Main Power Controller
Log Data and Events
Control ALL / INDIVIDUALLY via WIFI
Node 1 to Main Power Controller via WIFI
Node 2 to Main Power Controller via WIFI
Node 3 to Main Power Controller via WIFI
Power ON
Physical Button 1
Main Power Controller SET_SEND input
Power Button 1 ON/OFF/ACK
Node 1
Main Power Controller SET_MODE input
Input Selector 1 ADC / Stream
Power ON
Physical Button 2
Main Power Controller SET_SEND input
Power Button 2 ON/OFF/ACK
Node 2
Main Power Controller SET_MODE input
Input Selector 2 ADC / Stream
Power ON
Physical Button 3
Main Power Controller SET_SEND input
Power Button 3 ON/OFF/ACK
Node 3
Main Power Controller SET_MODE input
Input Selector 3 ADC / Stream

Figure 1: Overall System Architecture

Figure 1 illustrates the overall system-level architecture of the distributed power monitoring system. The design consists of three independent ESP32-based sensor nodes and a Raspberry Pi–based main power controller. Each node performs local signal acquisition and processing, while the main controller coordinates system-wide control, fault indication, and data logging across all nodes.

The Main Power Controller acts as the centralized supervisory unit. It receives measurement data from each node via Wi-Fi, issues control commands either globally or on a per-node basis, and manages system feedback mechanisms. The controller drives visual fault indicators for each node and records time-stamped measurement data and fault events for offline analysis.

Each ESP32 node operates as an independent sensing subsystem. Nodes acquire voltage and current inputs, compute RMS values locally, and respond to control commands issued by the main controller. By performing signal processing at the node level, the system reduces communication bandwidth and enables low-latency device-level response. In addition, each node implements a local protection mechanism: when a persistent overcurrent condition is detected, the node autonomously suppresses data transmission to isolate the faulted node from system-level monitoring, independent of the central controller.

Each node includes an input selection mechanism that supports multiple operating modes, allowing voltage and current data to be obtained either from physical analog inputs or from software-driven waveform streams delivered over Wi-Fi. This approach enables repeatable testing and controlled fault injection without reliance on physical power hardware, while maintaining identical signal-processing behavior across all modes.

Nodes support both physical power control and remote enable/disable commands issued by the main controller during normal operation. When a node enters a locally detected overcurrent protection state, re-enabling operation requires explicit acknowledgment from the main controller. This dual-control mechanism reflects practical embedded power systems, where local protection actions and centralized supervisory control coexist.

Communication between nodes and the main controller is implemented over Wi-Fi using a lightweight, connectionless protocol (UDP). Each node transmits measurement data independently, allowing asynchronous operation and preventing a single node failure from blocking system-wide monitoring.

Implementation

ESP32 Node Firmware Implementation

Node Configuration and Communication Setup

Each ESP32 node is configured with a unique node identifier and establishes a Wi-Fi connection to the local network during initialization. Three UDP sockets are used to separate functionality: one for receiving control and acknowledgment commands from the Raspberry Pi, one for transmitting measurement data, and one for receiving streamed waveform samples when operating in software-driven input modes. This separation simplifies command handling and prevents interference between data streaming and control traffic.

Input Acquisition and Operating Modes

Each node supports multiple operating modes that determine the source of voltage and current samples. In analog input mode, samples are acquired directly from ADC pins connected to voltage and current sensors or external signal sources such as a function generator. In streaming mode, raw ADC-equivalent samples are received over UDP from a software-based waveform source. Regardless of the input mode, all samples are processed using a unified signal-processing pipeline to ensure consistent behavior across testing and deployment scenarios.

Mode selection is controlled remotely by the main controller and reflected locally through dedicated mode indicator LEDs.

Local RMS Computation

Voltage and current samples are accumulated into fixed-size buffers of 60 samples corresponding to a single measurement cycle. Once the buffer is filled, root mean square (RMS) values are computed for both voltage and current. Each RMS result is tagged with a cycle identifier to support offline validation and comparison against reference data.

Equations:

$$Vrms = \sqrt{\frac{1}{N}\Sigma_{j=1}^{N} v^2}$$

$$Irms = \sqrt{\frac{1}{N}\Sigma_{j=1}^{N} i^2}$$

Performing RMS computation locally reduces communication bandwidth by avoiding transmission of raw samples and allows nodes to operate independently of the central controller for measurement processing.

Local Overcurrent Detection and Fault Latching

Each ESP32 node implements autonomous overcurrent detection using locally computed RMS current values. An overcurrent condition is identified when the RMS current exceeds 15 A and is evaluated across three consecutive measurement cycles to distinguish persistent faults from transient spikes. When a persistent overcurrent condition is detected, the node enters a latched fault state.

In the faulted state, the node suppresses further data transmission and reports a fault event to the main controller. This behavior isolates the faulted node from system-level monitoring while allowing the remainder of the system to continue operating normally. Fault recovery is intentionally prevented until the RMS current falls below a clear threshold of 12 A and an explicit acknowledgment command is received from the main controller.

Data Transmission Control

Measurement data packets containing the node identifier, cycle counter, and RMS (voltage and current) values are transmitted to the main controller using UDP immediately after new RMS results are computed, subject to a maximum transmission rate of one packet every 100 ms. Transmission may be explicitly enabled or disabled either by commands from the main controller or via a local physical switch during normal operation.

Command Handling and Acknowledgment Logic

Each node continuously listens for incoming control commands from the main controller. Supported commands include mode selection, transmission enable/disable, cycle counter reset, and fault acknowledgment. When operating in UDP input mode, the node additionally listens for streamed waveform data on a dedicated UDP port used exclusively for software-based testing and validation.

Raspberry Pi

Process 1: Network Controller and Command Interface

Process 1 serves as the network-facing controller on the Raspberry Pi and is responsible for coordinating communication between the ESP32 sensor nodes and the data-processing in Process 2. Its primary functions include receiving measurement data from all nodes, handling asynchronous fault notifications, issuing control commands, and forwarding aggregated data to the data-processing process through inter-process communication (IPC).

Process 1 employs a multithreaded design consisting of three concurrent threads. A UDP receiver thread listens for incoming measurement packets from ESP32 nodes on UDP port **5005** and updates a shared aggregate data structure containing the latest RMS voltage and current values from each node. A separate fault receiver thread listens on UDP port **6000** for asynchronous fault event messages generated by ESP32 nodes when local protection mechanisms are triggered. An interactive command thread provides a terminalbased supervisory interface that allows the operator to issue control commands to individual nodes or to all nodes simultaneously.

Measurement data and control or fault messages are received on separate UDP ports to ensure that regular measurement traffic does not delay time-critical fault or command handling. Process 1 aggregates the latest measurements from all nodes and forwards them to Process 2 using shared memory and semaphores, allowing network communication to operate independently from data processing and logging tasks.

The command interface supports mode selection, transmission enable or disable, cycle counter reset, and fault acknowledgment commands. Commands may be broadcast to all nodes or targeted to a specific node. Fault acknowledgment commands are only acted upon when nodes have satisfied fault-clear conditions, ensuring that recovery from protection events is explicitly supervised and preventing premature resumption of operation.

For development and debugging purposes, Process 1 drives mode indicator LEDs on the Raspberry Pi to reflect the currently selected system input mode. When a mode change command is issued, the corresponding LED provides immediate visual confirmation that the command was received and processed, allowing verification that the ESP32 nodes are correctly responding to supervisory control commands without requiring intrusive debugging tools or serial output.

Process 2: Data Processing, Fault Classification, and Logging

Process 2 implements system-level signal processing, fault classification, visualization, and persistent data logging for the distributed power monitoring system. It receives aggregated RMS voltage and current measurements from Process 1 via POSIX shared memory and semaphores and performs all derived computations independently of network communication. This architectural separation ensures that analysis, logging, and visualization are decoupled from network latency and packet arrival variability.

Process 2 uses a multithreaded design consisting of four concurrent worker threads. A voltage processing thread computes peak voltage values from received RMS measurements and classifies voltage conditions such as **NORMAL**, **SAG**, or **SWELL**. A current processing thread computes peak current values and classifies current conditions as **NORMAL** or **OVERCURRENT**. A logging thread records time-stamped measurement data and fault events to persistent storage, while a dedicated LED control thread provides real-time visual fault indication for each monitored node.

Peak voltage and current values are computed from RMS measurements using the standard sinusoidal relationships

$Vpeak = Vrms \cdot \sqrt{2}$ $\qquad$ $Ipeak = Irms \cdot \sqrt{2}$

These peak values are logged and displayed for observation and analysis only and are not used to trigger protective actions. Voltage fault classification is performed using RMS voltage thresholds of **50 V** for voltage sag and **130 V** for voltage swell.

System-level overcurrent detection in Process 2 uses an RMS current threshold of **11 A**, which serves as a warning and diagnostic indicator only. This threshold is intentionally lower than the protection threshold enforced locally on each ESP32 node. When RMS current exceeds this level, the condition is logged and reported as an overcurrent warning, but does not trigger protective action at the controller.

Each ESP32 node independently enforces a higher overcurrent threshold and autonomously suppresses further data transmission when that threshold is exceeded persistently. As a result, the node becomes isolated from system-level monitoring without requiring intervention from the central controller. If a node subsequently stops transmitting data, historical log entries allow reviewers to infer that current levels were rising and likely reached the ESP32's protection threshold, resulting in intentional node-level isolation.

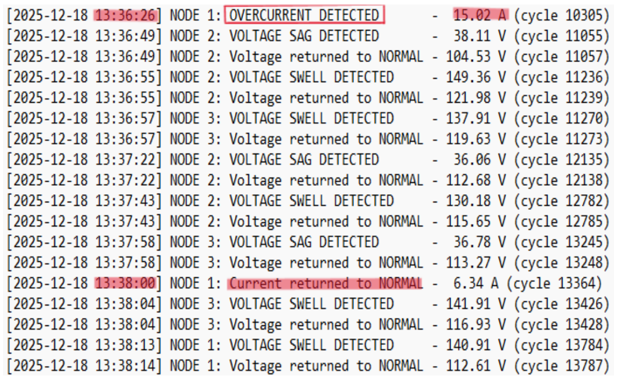
Voltage swell detection is intentionally handled only at the supervisory controller rather than at the ESP32 node level. Unlike overcurrent events, which represent immediate safety risks requiring fast device-level intervention, voltage swell is treated as a system-level operating condition. Voltage abnormalities typically persist over longer time scales and are therefore more appropriately monitored, logged, and analyzed centrally rather than triggering autonomous node shutdown.

To support offline analysis and traceability, Process 2 logs measurement data to a CSV file at a fixed interval of **10 seconds**, including RMS values, peak values, calculated power, and fault status for all nodes. Fault events are additionally recorded to a separate event log file, capturing transitions into and out of abnormal operating states with time stamps and node identifiers. This dual-logging approach enables both long-term trend analysis and precise reconstruction of fault behavior.

Visual fault indication is implemented using dedicated LEDs for each node, driven by the LED control thread. Voltage and current fault states are displayed using priority-based blinking patterns, allowing rapid identification of abnormal conditions without reliance on terminal output. LED updates are only performed when state changes occur, minimizing unnecessary GPIO activity. These indicators primarily serve as development and debugging aids during system integration and validation.

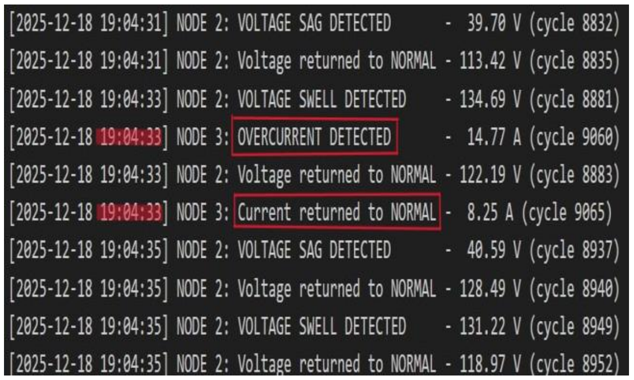
All shared data structures accessed by Process 2 threads are protected using mutexes to ensure thread-safe operation. Threads are executed at fixed update intervals of approximately **50 ms**, providing responsive monitoring while avoiding excessive CPU utilization. By separating data reception, processing, logging, and visualization into independent threads, Process 2 achieves deterministic behavior and scalable performance as system complexity increases.

Result and Validation



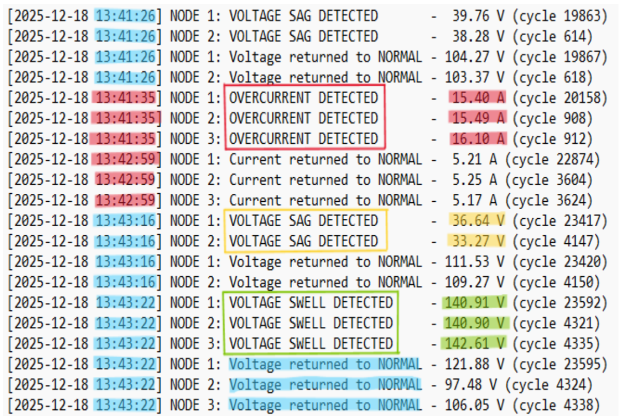
```
[2025-12-18 13:36:26] NODE 1: OVERCURRENT DETECTED       -  15.02 A (cycle 10305)
[2025-12-18 13:36:49] NODE 2: VOLTAGE SAG DETECTED       -  38.11 V (cycle 11055)
[2025-12-18 13:36:49] NODE 2: Voltage returned to NORMAL - 104.53 V (cycle 11057)
[2025-12-18 13:36:55] NODE 2: VOLTAGE SWELL DETECTED     - 149.36 V (cycle 11236)
[2025-12-18 13:36:55] NODE 2: Voltage returned to NORMAL - 121.98 V (cycle 11239)
[2025-12-18 13:36:57] NODE 3: VOLTAGE SWELL DETECTED     - 137.91 V (cycle 11270)
[2025-12-18 13:36:57] NODE 3: Voltage returned to NORMAL - 119.63 V (cycle 11273)
[2025-12-18 13:37:22] NODE 2: VOLTAGE SAG DETECTED       -  36.06 V (cycle 12135)
[2025-12-18 13:37:22] NODE 2: Voltage returned to NORMAL - 112.68 V (cycle 12138)
[2025-12-18 13:37:43] NODE 2: VOLTAGE SWELL DETECTED     - 130.18 V (cycle 12782)
[2025-12-18 13:37:43] NODE 2: Voltage returned to NORMAL - 115.65 V (cycle 12785)
[2025-12-18 13:37:58] NODE 3: VOLTAGE SAG DETECTED       -  36.78 V (cycle 13245)
[2025-12-18 13:37:58] NODE 3: Voltage returned to NORMAL - 113.27 V (cycle 13248)
[2025-12-18 13:38:00] NODE 1: Current returned to NORMAL -  6.34 A (cycle 13364)
[2025-12-18 13:38:04] NODE 3: VOLTAGE SWELL DETECTED     - 141.91 V (cycle 13426)
[2025-12-18 13:38:04] NODE 3: Voltage returned to NORMAL - 116.93 V (cycle 13428)
[2025-12-18 13:38:13] NODE 1: VOLTAGE SWELL DETECTED     - 140.91 V (cycle 13784)
[2025-12-18 13:38:14] NODE 1: Voltage returned to NORMAL - 112.61 V (cycle 13787)
```

Figure 2a: Fault Log Events



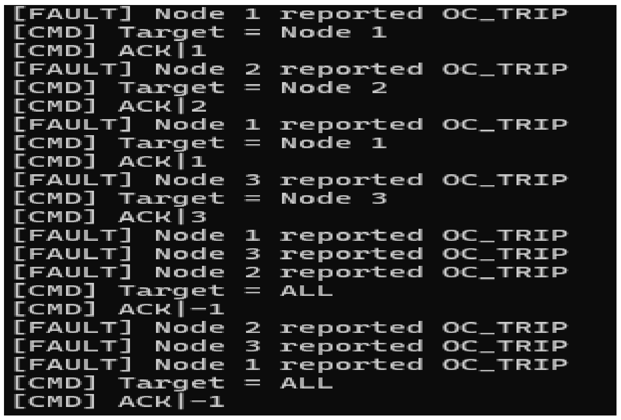
```
[2025-12-18 19:04:31] NODE 2: VOLTAGE SAG DETECTED       -  39.70 V (cycle 8832)
[2025-12-18 19:04:31] NODE 2: Voltage returned to NORMAL - 113.42 V (cycle 8835)
[2025-12-18 19:04:33] NODE 2: VOLTAGE SWELL DETECTED     - 134.69 V (cycle 8881)
[2025-12-18 19:04:33] NODE 3: OVERCURRENT DETECTED       -  14.77 A (cycle 9060)
[2025-12-18 19:04:33] NODE 2: Voltage returned to NORMAL - 122.19 V (cycle 8883)
[2025-12-18 19:04:33] NODE 3: Current returned to NORMAL -  8.25 A (cycle 9065)
[2025-12-18 19:04:35] NODE 2: VOLTAGE SAG DETECTED       -  40.59 V (cycle 8937)
[2025-12-18 19:04:35] NODE 2: Voltage returned to NORMAL - 128.49 V (cycle 8940)
[2025-12-18 19:04:35] NODE 2: VOLTAGE SWELL DETECTED     - 131.22 V (cycle 8949)
[2025-12-18 19:04:35] NODE 2: Voltage returned to NORMAL - 118.97 V (cycle 8952)
```

Figure 2b: Fault Log Events



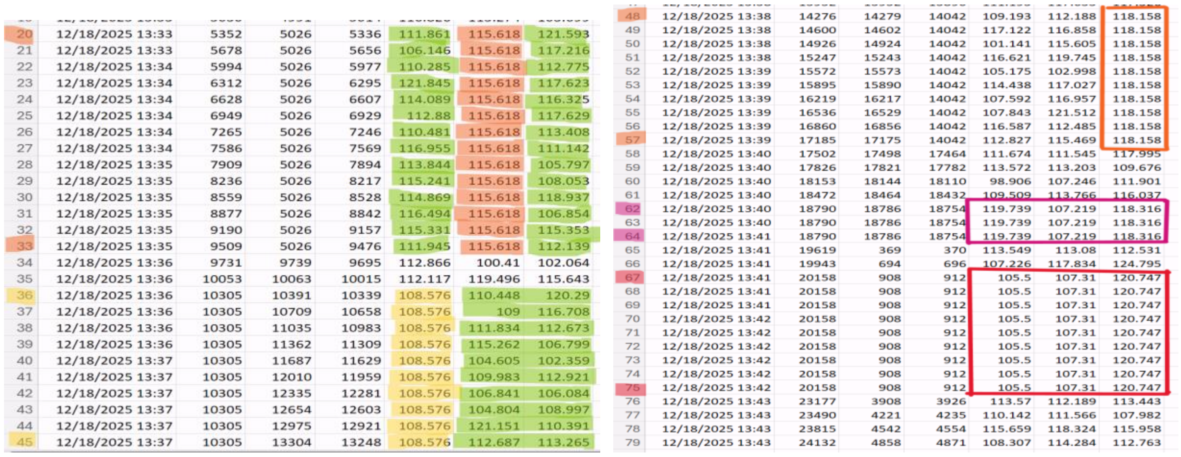
```
[2025-12-18 13:41:26] NODE 1: VOLTAGE SAG DETECTED       -  39.76 V (cycle 19863)
[2025-12-18 13:41:26] NODE 2: VOLTAGE SAG DETECTED       -  38.28 V (cycle 614)
[2025-12-18 13:41:26] NODE 1: Voltage returned to NORMAL - 104.27 V (cycle 19867)
[2025-12-18 13:41:26] NODE 2: Voltage returned to NORMAL - 103.37 V (cycle 618)
[2025-12-18 13:41:35] NODE 1: OVERCURRENT DETECTED       -  15.40 A (cycle 20158)
[2025-12-18 13:41:35] NODE 2: OVERCURRENT DETECTED       -  15.49 A (cycle 908)
[2025-12-18 13:41:35] NODE 3: OVERCURRENT DETECTED       -  16.10 A (cycle 912)
[2025-12-18 13:42:59] NODE 1: Current returned to NORMAL -  5.21 A (cycle 22874)
[2025-12-18 13:42:59] NODE 2: Current returned to NORMAL -  5.25 A (cycle 3604)
[2025-12-18 13:42:59] NODE 3: Current returned to NORMAL -  5.17 A (cycle 3624)
[2025-12-18 13:43:16] NODE 1: VOLTAGE SAG DETECTED       -  36.64 V (cycle 23417)
[2025-12-18 13:43:16] NODE 2: VOLTAGE SAG DETECTED       -  33.27 V (cycle 4147)
[2025-12-18 13:43:16] NODE 1: Voltage returned to NORMAL - 111.53 V (cycle 23420)
[2025-12-18 13:43:16] NODE 2: Voltage returned to NORMAL - 109.27 V (cycle 4150)
[2025-12-18 13:43:22] NODE 1: VOLTAGE SWELL DETECTED     - 140.91 V (cycle 23592)
[2025-12-18 13:43:22] NODE 2: VOLTAGE SWELL DETECTED     - 140.90 V (cycle 4321)
[2025-12-18 13:43:22] NODE 3: VOLTAGE SWELL DETECTED     - 142.61 V (cycle 4335)
[2025-12-18 13:43:22] NODE 1: Voltage returned to NORMAL - 121.88 V (cycle 23595)
[2025-12-18 13:43:22] NODE 2: Voltage returned to NORMAL - 97.48 V (cycle 4324)
[2025-12-18 13:43:22] NODE 3: Voltage returned to NORMAL - 106.05 V (cycle 4338)
```

Figure 2c: Fault Log Events



```
[FAULT] Node 1 reported OC_TRIP
[CMD] Target = Node 1
[CMD] ACK|1
[FAULT] Node 2 reported OC_TRIP
[CMD] Target = Node 2
[CMD] ACK|2
[FAULT] Node 1 reported OC_TRIP
[CMD] Target = Node 1
[CMD] ACK|1
[FAULT] Node 3 reported OC_TRIP
[CMD] Target = Node 3
[CMD] ACK|3
[FAULT] Node 1 reported OC_TRIP
[FAULT] Node 3 reported OC_TRIP
[FAULT] Node 2 reported OC_TRIP
[CMD] Target = ALL
[CMD] ACK|-1
[FAULT] Node 2 reported OC_TRIP
[FAULT] Node 3 reported OC_TRIP
[FAULT] Node 1 reported OC_TRIP
[CMD] Target = ALL
[CMD] ACK|-1
```

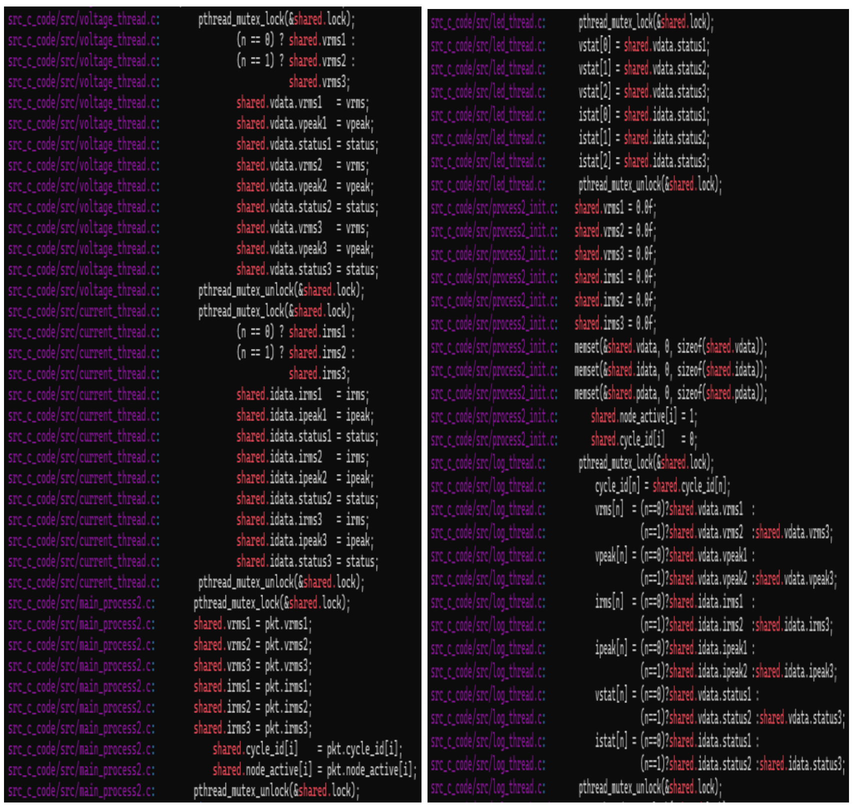
Figure 2d: ESP reported to PI



20	12/18/2025 13:33	5352	5026	5336	111.861	115.618	121.593
21	12/18/2025 13:33	5678	5026	5656	106.146	115.618	117.216
22	12/18/2025 13:34	5994	5026	5977	110.285	115.618	112.775
23	12/18/2025 13:34	6312	5026	6295	121.845	115.618	117.623
24	12/18/2025 13:34	6628	5026	6607	114.089	115.618	116.325
25	12/18/2025 13:34	6949	5026	6929	112.88	115.618	117.629
26	12/18/2025 13:34	7265	5026	7246	110.481	115.618	113.408
27	12/18/2025 13:34	7586	5026	7569	116.955	115.618	111.142
28	12/18/2025 13:35	7909	5026	7894	113.844	115.618	105.797
29	12/18/2025 13:35	8236	5026	8217	115.241	115.618	108.053
30	12/18/2025 13:35	8559	5026	8528	114.869	115.618	118.937
31	12/18/2025 13:35	8877	5026	8842	116.494	115.618	106.854
32	12/18/2025 13:35	9190	5026	9157	115.331	115.618	115.353
33	12/18/2025 13:35	9509	5026	9476	111.945	115.618	112.139
34	12/18/2025 13:36	9731	9739	9695	112.866	100.41	102.064
35	12/18/2025 13:36	10053	10063	10015	112.117	119.496	115.643
36	12/18/2025 13:36	10305	10391	10339	108.576	110.448	120.29
37	12/18/2025 13:36	10305	10709	10658	108.576	109	116.708
38	12/18/2025 13:36	10305	11035	10983	108.576	111.834	112.673
39	12/18/2025 13:36	10305	11362	11309	108.576	115.262	106.799
40	12/18/2025 13:37	10305	11687	11629	108.576	104.605	102.359
41	12/18/2025 13:37	10305	12010	11959	108.576	109.983	112.921
42	12/18/2025 13:37	10305	12335	12281	108.576	106.841	106.084
43	12/18/2025 13:37	10305	12654	12603	108.576	104.804	108.997
44	12/18/2025 13:37	10305	12975	12921	108.576	121.151	110.391
45	12/18/2025 13:37	10305	13304	13248	108.576	112.687	113.265

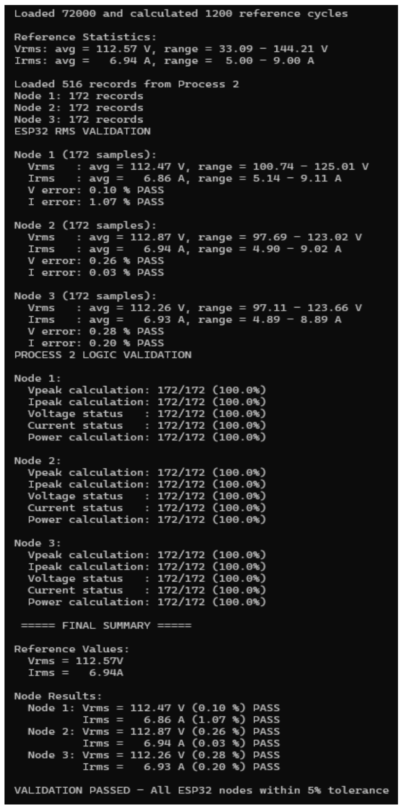
48	12/18/2025 13:38	14276	14279	14042	109.193	112.188	118.158
49	12/18/2025 13:38	14600	14602	14042	117.122	116.858	118.158
50	12/18/2025 13:38	14926	14924	14042	101.141	115.605	118.158
51	12/18/2025 13:38	15247	15243	14042	116.621	119.745	118.158
52	12/18/2025 13:39	15572	15573	14042	105.175	102.998	118.158
53	12/18/2025 13:39	15895	15890	14042	114.438	117.027	118.158
54	12/18/2025 13:39	16219	16217	14042	107.592	116.957	118.158
55	12/18/2025 13:39	16536	16529	14042	107.843	121.512	118.158
56	12/18/2025 13:39	16860	16856	14042	116.587	112.485	118.158
57	12/18/2025 13:39	17185	17175	14042	112.827	115.469	118.158
58	12/18/2025 13:40	17502	17498	17464	111.674	111.545	117.995
59	12/18/2025 13:40	17826	17821	17782	113.572	113.203	109.676
60	12/18/2025 13:40	18153	18144	18110	98.906	107.246	111.901
61	12/18/2025 13:40	18472	18464	18432	109.509	113.766	116.037
62	12/18/2025 13:40	18790	18786	18754	119.739	107.219	118.316
63	12/18/2025 13:40	18790	18786	18754	119.739	107.219	118.316
64	12/18/2025 13:41	18790	18786	18754	119.739	107.219	118.316
65	12/18/2025 13:41	19619	369	370	113.549	113.08	112.531
66	12/18/2025 13:41	19943	694	696	107.226	117.834	124.795
67	12/18/2025 13:41	20158	908	912	105.5	107.31	120.747
68	12/18/2025 13:41	20158	908	912	105.5	107.31	120.747
69	12/18/2025 13:41	20158	908	912	105.5	107.31	120.747
70	12/18/2025 13:42	20158	908	912	105.5	107.31	120.747
71	12/18/2025 13:42	20158	908	912	105.5	107.31	120.747
72	12/18/2025 13:42	20158	908	912	105.5	107.31	120.747
73	12/18/2025 13:42	20158	908	912	105.5	107.31	120.747
74	12/18/2025 13:42	20158	908	912	105.5	107.31	120.747
75	12/18/2025 13:42	20158	908	912	105.5	107.31	120.747
76	12/18/2025 13:43	23177	3908	3926	113.57	112.189	113.443
77	12/18/2025 13:43	23490	4221	4235	110.142	111.566	107.982
78	12/18/2025 13:43	23815	4542	4554	115.659	118.324	115.958
79	12/18/2025 13:43	24132	4858	4871	108.307	114.284	112.763

Figure 3



```
src_c_code/src/voltage_thread.c:      pthread_mutex_lock(&shared.lock);
src_c_code/src/voltage_thread.c:          (n == 0) ? shared.vrms1 :
src_c_code/src/voltage_thread.c:          (n == 1) ? shared.vrms2 :
src_c_code/src/voltage_thread.c:                     shared.vrms3;
src_c_code/src/voltage_thread.c:          shared.vdata.vrms1   = vrms;
src_c_code/src/voltage_thread.c:          shared.vdata.vpeak1  = vpeak;
src_c_code/src/voltage_thread.c:          shared.vdata.status1 = status;
src_c_code/src/voltage_thread.c:          shared.vdata.vrms2   = vrms;
src_c_code/src/voltage_thread.c:          shared.vdata.vpeak2  = vpeak;
src_c_code/src/voltage_thread.c:          shared.vdata.status2 = status;
src_c_code/src/voltage_thread.c:          shared.vdata.vrms3   = vrms;
src_c_code/src/voltage_thread.c:          shared.vdata.vpeak3  = vpeak;
src_c_code/src/voltage_thread.c:          shared.vdata.status3 = status;
src_c_code/src/voltage_thread.c:      pthread_mutex_unlock(&shared.lock);
src_c_code/src/current_thread.c:      pthread_mutex_lock(&shared.lock);
src_c_code/src/current_thread.c:          (n == 0) ? shared.irms1 :
src_c_code/src/current_thread.c:          (n == 1) ? shared.irms2 :
src_c_code/src/current_thread.c:                     shared.irms3;
src_c_code/src/current_thread.c:          shared.idata.irms1   = irms;
src_c_code/src/current_thread.c:          shared.idata.ipeak1  = ipeak;
src_c_code/src/current_thread.c:          shared.idata.status1 = status;
src_c_code/src/current_thread.c:          shared.idata.irms2   = irms;
src_c_code/src/current_thread.c:          shared.idata.ipeak2  = ipeak;
src_c_code/src/current_thread.c:          shared.idata.status2 = status;
src_c_code/src/current_thread.c:          shared.idata.irms3   = irms;
src_c_code/src/current_thread.c:          shared.idata.ipeak3  = ipeak;
src_c_code/src/current_thread.c:          shared.idata.status3 = status;
src_c_code/src/current_thread.c:      pthread_mutex_unlock(&shared.lock);
src_c_code/src/main_process2.c:     pthread_mutex_lock(&shared.lock);
src_c_code/src/main_process2.c:     shared.vrms1 = pkt.vrms1;
src_c_code/src/main_process2.c:     shared.vrms2 = pkt.vrms2;
src_c_code/src/main_process2.c:     shared.vrms3 = pkt.vrms3;
src_c_code/src/main_process2.c:     shared.irms1 = pkt.irms1;
src_c_code/src/main_process2.c:     shared.irms2 = pkt.irms2;
src_c_code/src/main_process2.c:     shared.irms3 = pkt.irms3;
src_c_code/src/main_process2.c:         shared.cycle_id[i]    = pkt.cycle_id[i];
src_c_code/src/main_process2.c:         shared.node_active[i] = pkt.node_active[i];
src_c_code/src/main_process2.c:     pthread_mutex_unlock(&shared.lock);
```

```
src_c_code/src/led_thread.c:      pthread_mutex_lock(&shared.lock);
src_c_code/src/led_thread.c:      vstat[0] = shared.vdata.status1;
src_c_code/src/led_thread.c:      vstat[1] = shared.vdata.status2;
src_c_code/src/led_thread.c:      vstat[2] = shared.vdata.status3;
src_c_code/src/led_thread.c:      istat[0] = shared.idata.status1;
src_c_code/src/led_thread.c:      istat[1] = shared.idata.status2;
src_c_code/src/led_thread.c:      istat[2] = shared.idata.status3;
src_c_code/src/led_thread.c:      pthread_mutex_unlock(&shared.lock);
src_c_code/src/process2_init.c:   shared.vrms1 = 0.0f;
src_c_code/src/process2_init.c:   shared.vrms2 = 0.0f;
src_c_code/src/process2_init.c:   shared.vrms3 = 0.0f;
src_c_code/src/process2_init.c:   shared.irms1 = 0.0f;
src_c_code/src/process2_init.c:   shared.irms2 = 0.0f;
src_c_code/src/process2_init.c:   shared.irms3 = 0.0f;
src_c_code/src/process2_init.c:   memset(&shared.vdata, 0, sizeof(shared.vdata));
src_c_code/src/process2_init.c:   memset(&shared.idata, 0, sizeof(shared.idata));
src_c_code/src/process2_init.c:   memset(&shared.pdata, 0, sizeof(shared.pdata));
src_c_code/src/process2_init.c:       shared.node_active[i] = 1;
src_c_code/src/process2_init.c:       shared.cycle_id[i]    = 0;
src_c_code/src/log_thread.c:      pthread_mutex_lock(&shared.lock);
src_c_code/src/log_thread.c:          cycle_id[n] = shared.cycle_id[n];
src_c_code/src/log_thread.c:          vrms[n]  = (n==0)?shared.vdata.vrms1  :
src_c_code/src/log_thread.c:                     (n==1)?shared.vdata.vrms2  :shared.vdata.vrms3;
src_c_code/src/log_thread.c:          vpeak[n] = (n==0)?shared.vdata.vpeak1 :
src_c_code/src/log_thread.c:                     (n==1)?shared.vdata.vpeak2 :shared.vdata.vpeak3;
src_c_code/src/log_thread.c:          irms[n]  = (n==0)?shared.idata.irms1  :
src_c_code/src/log_thread.c:                     (n==1)?shared.idata.irms2  :shared.idata.irms3;
src_c_code/src/log_thread.c:          ipeak[n] = (n==0)?shared.idata.ipeak1 :
src_c_code/src/log_thread.c:                     (n==1)?shared.idata.ipeak2 :shared.idata.ipeak3;
src_c_code/src/log_thread.c:          vstat[n] = (n==0)?shared.vdata.status1 :
src_c_code/src/log_thread.c:                     (n==1)?shared.vdata.status2 :shared.vdata.status3;
src_c_code/src/log_thread.c:          istat[n] = (n==0)?shared.idata.status1 :
src_c_code/src/log_thread.c:                     (n==1)?shared.idata.status2 :shared.idata.status3;
src_c_code/src/log_thread.c:      pthread_mutex_unlock(&shared.lock);
```



```
Loaded 72000 and calculated 1200 reference cycles

Reference Statistics:
Vrms: avg = 112.57 V, range = 33.09 - 144.21 V
Irms: avg =   6.94 A, range =  5.00 - 9.00 A

Loaded 516 records from Process 2
Node 1: 172 records
Node 2: 172 records
Node 3: 172 records
ESP32 RMS VALIDATION

Node 1 (172 samples):
  Vrms   : avg = 112.47 V, range = 100.74 - 125.01 V
  Irms   : avg =   6.86 A, range = 5.14 - 9.11 A
  V error: 0.10 % PASS
  I error: 1.07 % PASS

Node 2 (172 samples):
  Vrms   : avg = 112.87 V, range = 97.69 - 123.02 V
  Irms   : avg =   6.94 A, range = 4.90 - 9.02 A
  V error: 0.26 % PASS
  I error: 0.03 % PASS

Node 3 (172 samples):
  Vrms   : avg = 112.26 V, range = 97.11 - 123.66 V
  Irms   : avg =   6.93 A, range = 4.89 - 8.89 A
  V error: 0.28 % PASS
  I error: 0.20 % PASS
PROCESS 2 LOGIC VALIDATION

Node 1:
  Vpeak calculation: 172/172 (100.0%)
  Ipeak calculation: 172/172 (100.0%)
  Voltage status   : 172/172 (100.0%)
  Current status   : 172/172 (100.0%)
  Power calculation: 172/172 (100.0%)

Node 2:
  Vpeak calculation: 172/172 (100.0%)
  Ipeak calculation: 172/172 (100.0%)
  Voltage status   : 172/172 (100.0%)
  Current status   : 172/172 (100.0%)
  Power calculation: 172/172 (100.0%)

Node 3:
  Vpeak calculation: 172/172 (100.0%)
  Ipeak calculation: 172/172 (100.0%)
  Voltage status   : 172/172 (100.0%)
  Current status   : 172/172 (100.0%)
  Power calculation: 172/172 (100.0%)

 ===== FINAL SUMMARY =====

Reference Values:
  Vrms = 112.57V
  Irms =   6.94A

Node Results:
  Node 1: Vrms = 112.47 V (0.10 %) PASS
          Irms =   6.86 A (1.07 %) PASS
  Node 2: Vrms = 112.87 V (0.26 %) PASS
          Irms =   6.94 A (0.03 %) PASS
  Node 3: Vrms = 112.26 V (0.28 %) PASS
          Irms =   6.93 A (0.20 %) PASS

VALIDATION PASSED - All ESP32 nodes within 5% tolerance
```

Figure 4a: Mutex protected

Figure 4b: Validation

The system was evaluated through fault event logging, numerical validation, concurrency inspection, and visual observability. The results confirm correct detection and classification of voltage and current abnormalities, accurate distributed RMS computation, reliable multi-threaded execution, and intentional node isolation behavior under fault conditions.

As shown in Figures 2a–2c, voltage-related faults are detected and classified correctly based on RMS voltage thresholds. Voltage sag events occur when RMS voltage falls below 50 V, while voltage swell events occur when RMS voltage exceeds 130 V. In all observed cases, voltage sag and swell conditions are followed by a corresponding "Voltage returned to NORMAL" entry within the same or immediately subsequent timestamp. This behavior confirms that voltage faults are treated as transient operating conditions and recover automatically once voltage returns to an acceptable range.

Current-related behavior exhibits three distinct operating regions. When RMS current remains at or below 11 A, the system operates normally and no fault condition is recorded. When RMS current exceeds 11 A, but remains less than or equal to 15 A, the condition is detected and logged as a system-level overcurrent warning. In this range, the system behavior mirrors that of voltage sag and swell events: once the current returns below the threshold, a corresponding "Current returned to NORMAL" entry is recorded within the same or subsequent timestamp. No transmission suppression or node isolation occurs in this regime, confirming that these events are treated as recoverable warnings rather than protection triggers.

A different behavior is observed when RMS current exceeds 15 A. As shown in Figure 2c, currents above this threshold trigger a local fault condition on the ESP32 node. In this case, the node enters a latched fault state and autonomously suppresses further data transmission. Because the node intentionally stops sending measurement packets, the system does not immediately log a return to normal operation. Recovery from this condition requires an explicit acknowledgment command from the main controller before data transmission resumes. This distinction confirms that currents above 11 A, but below the local protection threshold are handled as transient warnings, while currents above 15 A trigger intentional node-level isolation.

This behavior is further validated in Figure 2d, which shows OC_TRIP fault messages reported by the ESP32 nodes to the main controller whenever the local protection threshold is exceeded. These messages indicate that the node has entered a faulted state and is awaiting acknowledgment before resuming operation. The presence of earlier system-level overcurrent warnings in the logs allows reviewers to infer that current levels were rising prior to node isolation.

Intentional suppression of data transmission following a local overcurrent trip is verified in Figure 3, where multiple consecutive log entries show unchanged values across specific line ranges (lines 20–33, 36–45, 48–57, 62–64, and 67–75). The absence of updated measurement values in these highlighted regions confirms that the ESP32 node stopped transmitting new packets after entering the faulted state. This demonstrates that missing data during overcurrent events is an intentional design feature rather than a communication failure.

Numerical accuracy of the distributed signal-processing pipeline is validated in Figure 4b, which compares RMS voltage and current values computed by the ESP32 nodes against offline reference calculations. A total of 72,000 samples were processed, yielding 1,200 reference RMS cycles, with 172 validated records per node received by Process 2. Across all nodes, RMS voltage and current errors remained below 0.3%, and all validation checks for peak calculation, voltage classification, current classification, and power computation passed with 100% success. These results confirm that embedded RMS computation on the ESP32 nodes is accurate and consistent with reference models.

Correct concurrent operation within Process 2 is demonstrated in Figure 4a, which shows mutex-protected access to shared voltage, current, status, and cycle count variables across multiple threads. Consistent use of mutex locking ensures thread-safe data exchange between voltage processing, current processing, logging, and LED control threads. No race conditions or inconsistent classifications were observed during continuous operation.

In addition to software-level validation, visual status indicators were used as a lightweight verification mechanism during development and testing. Onboard LEDs provided real-time feedback of node operating mode, transmission enable state, and fault conditions. These indicators were used to confirm correct mode transitions, verify intentional suppression of data transmission during fault events, and validate proper recovery behavior following fault acknowledgment without requiring intrusive debugging tools or serial output.

Overall, the results demonstrate that the system reliably detects voltage and current abnormalities, accurately performs distributed RMS computation, maintains data integrity under concurrent processing, and provides sufficient observability to interpret node isolation events. The layered warning-and-isolation approach enables early diagnostic insight at the supervisory controller while preserving autonomous node-level protection behavior.

Conclusion

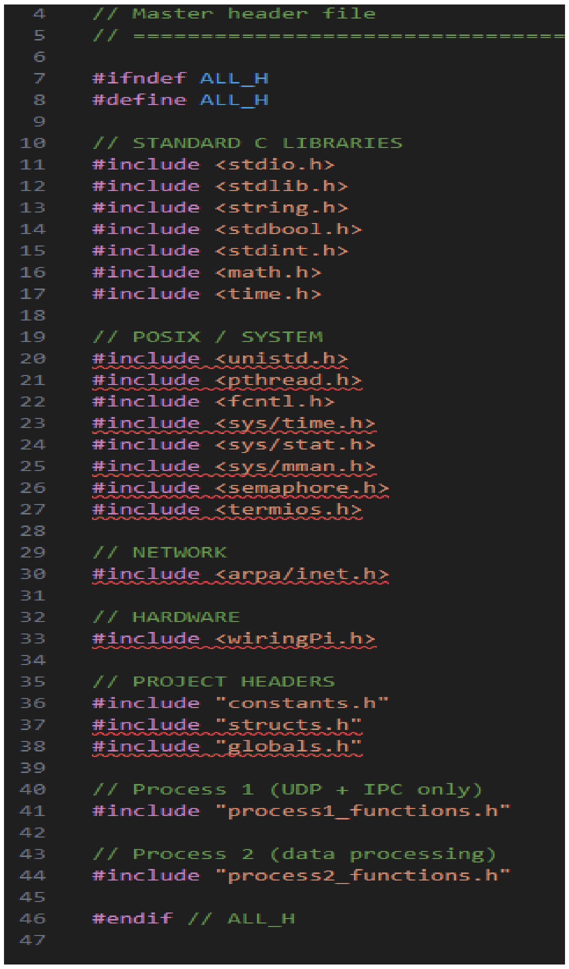
This project demonstrated the design and validation of a distributed embedded power monitoring system using multiple ESP32 sensor nodes and a Raspberry Pi central controller. By distributing signal acquisition and RMS computation to the node level, the system achieved accurate measurements while enabling early detection of abnormal operating conditions without relying on centralized processing.

Local RMS computation on the ESP32 nodes closely matched reference calculations, confirming that reliable signal processing can be performed on resource-constrained microcontrollers. Sustained overcurrent conditions triggered autonomous suppression of data transmission at the node level, intentionally isolating faulted nodes and requiring explicit acknowledgment before resuming operation. This behavior provided fast, localized protection while remaining observable at the system level through historical logs.

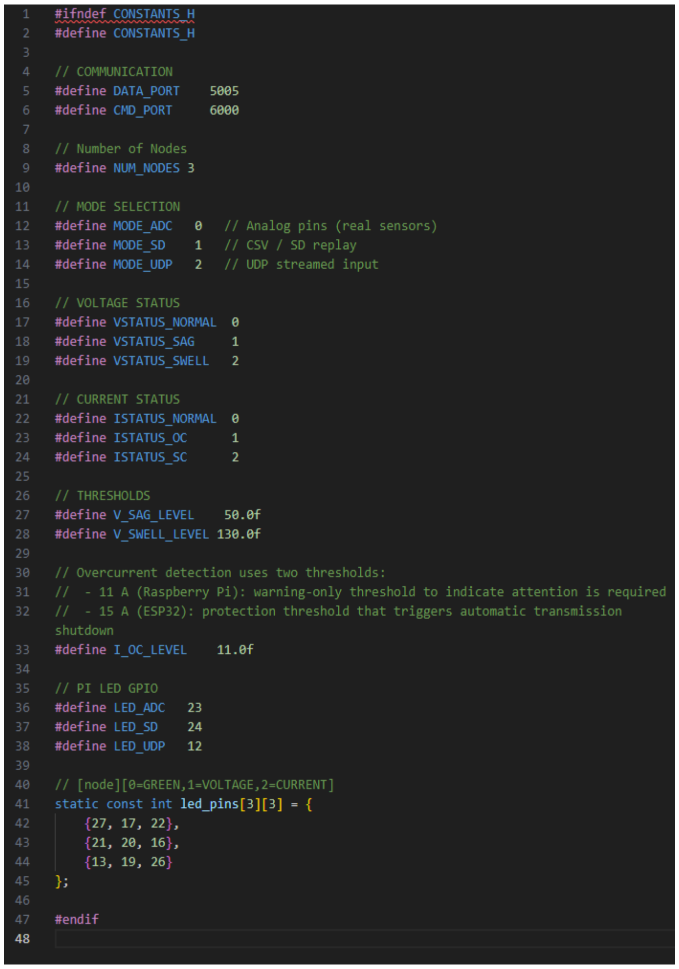
At the supervisory level, the Raspberry Pi aggregated measurements, classified voltage and current conditions, and logged system behavior without interfering with time-critical communication. Separating network handling from data processing ensured deterministic operation under concurrent execution. Layered current thresholds allowed the system to record early warnings while reserving transmission suppression for more severe conditions.

The use of software-based waveform streaming enabled repeatable testing and controlled fault injection without physical power hardware, supporting thorough validation of accuracy, fault detection, and recovery behavior. Overall, the project shows that combining local autonomy with centralized supervision results in a flexible and observable distributed embedded monitoring system suitable for studying real-world power system behavior.

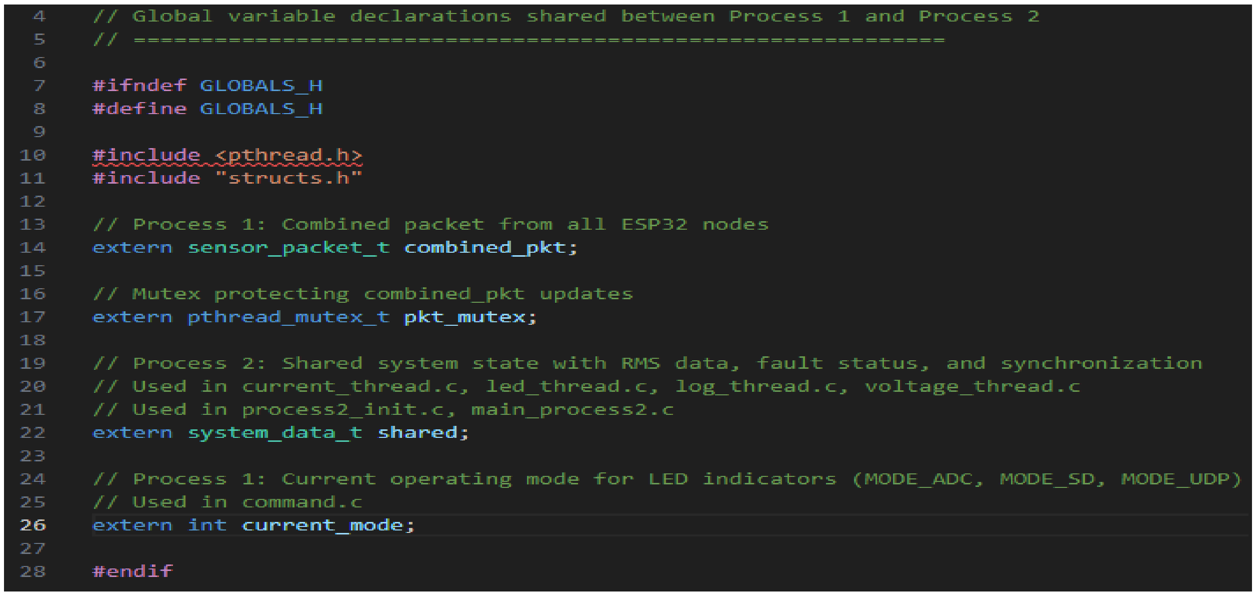
Codes



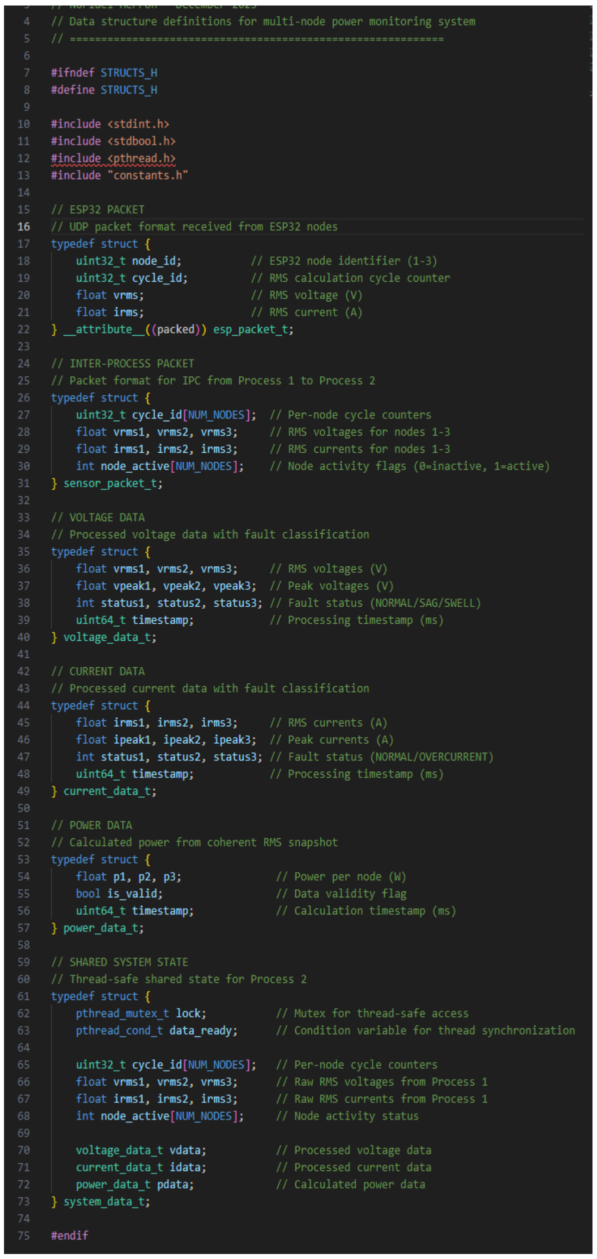
```c
// Master header file
// ================================

#ifndef ALL_H
#define ALL_H

// STANDARD C LIBRARIES
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <stdint.h>
#include <math.h>
#include <time.h>

// POSIX / SYSTEM
#include <unistd.h>
#include <pthread.h>
#include <fcntl.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <semaphore.h>
#include <termios.h>

// NETWORK
#include <arpa/inet.h>

// HARDWARE
#include <wiringPi.h>

// PROJECT HEADERS
#include "constants.h"
#include "structs.h"
#include "globals.h"

// Process 1 (UDP + IPC only)
#include "process1_functions.h"

// Process 2 (data processing)
#include "process2_functions.h"

#endif // ALL_H
```



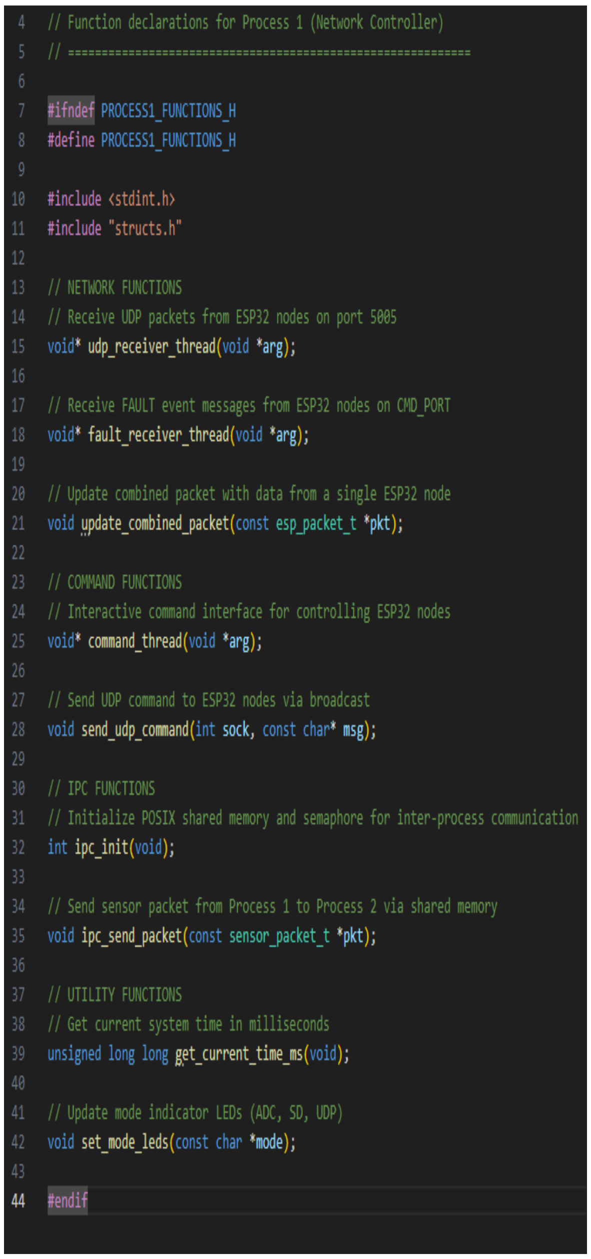
```c
#ifndef CONSTANTS_H
#define CONSTANTS_H

// COMMUNICATION
#define DATA_PORT    5005
#define CMD_PORT     6000

// Number of Nodes
#define NUM_NODES 3

// MODE SELECTION
#define MODE_ADC    0   // Analog pins (real sensors)
#define MODE_SD     1   // CSV / SD replay
#define MODE_UDP    2   // UDP streamed input

// VOLTAGE STATUS
#define VSTATUS_NORMAL  0
#define VSTATUS_SAG     1
#define VSTATUS_SWELL   2

// CURRENT STATUS
#define ISTATUS_NORMAL  0
#define ISTATUS_OC      1
#define ISTATUS_SC      2

// THRESHOLDS
#define V_SAG_LEVEL    50.0f
#define V_SWELL_LEVEL 130.0f

// Overcurrent detection uses two thresholds:
//  - 11 A (Raspberry Pi): warning-only threshold to indicate attention is required
//  - 15 A (ESP32): protection threshold that triggers automatic transmission
shutdown
#define I_OC_LEVEL    11.0f

// PI LED GPIO
#define LED_ADC   23
#define LED_SD    24
#define LED_UDP   12

// [node][0=GREEN,1=VOLTAGE,2=CURRENT]
static const int led_pins[3][3] = {
    {27, 17, 22},
    {21, 20, 16},
    {13, 19, 26}
};

#endif
```



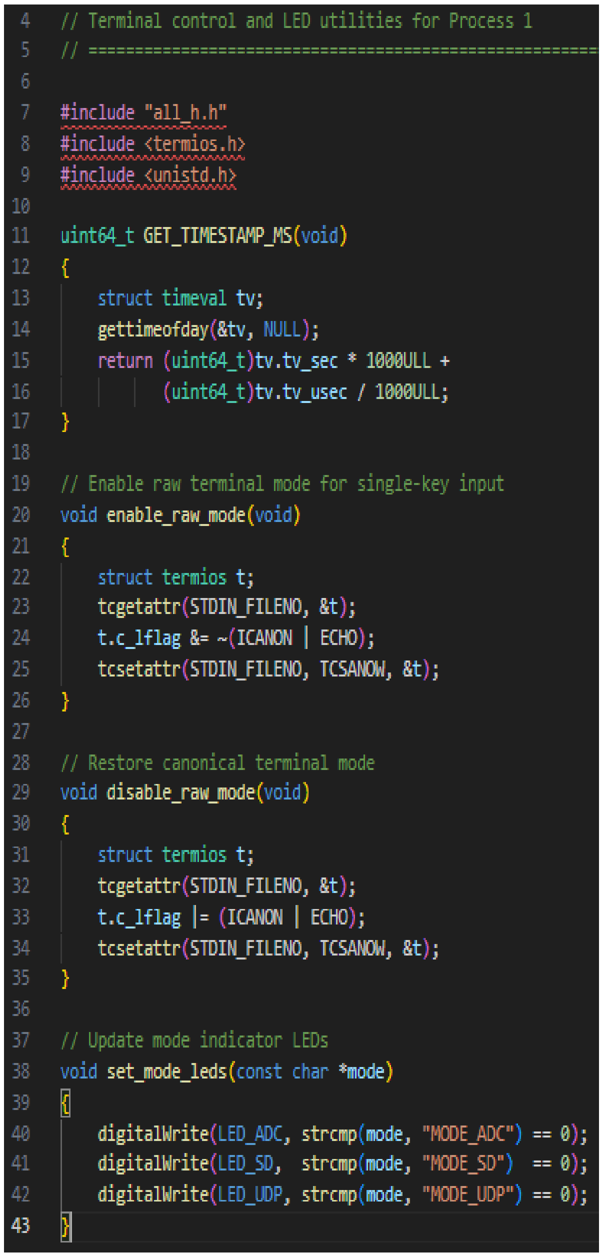
```c
// Global variable declarations shared between Process 1 and Process 2
// ============================================================

#ifndef GLOBALS_H
#define GLOBALS_H

#include <pthread.h>
#include "structs.h"

// Process 1: Combined packet from all ESP32 nodes
extern sensor_packet_t combined_pkt;

// Mutex protecting combined_pkt updates
extern pthread_mutex_t pkt_mutex;

// Process 2: Shared system state with RMS data, fault status, and synchronization
// Used in current_thread.c, led_thread.c, log_thread.c, voltage_thread.c
// Used in process2_init.c, main_process2.c
extern system_data_t shared;

// Process 1: Current operating mode for LED indicators (MODE_ADC, MODE_SD, MODE_UDP)
// Used in command.c
extern int current_mode;

#endif
```



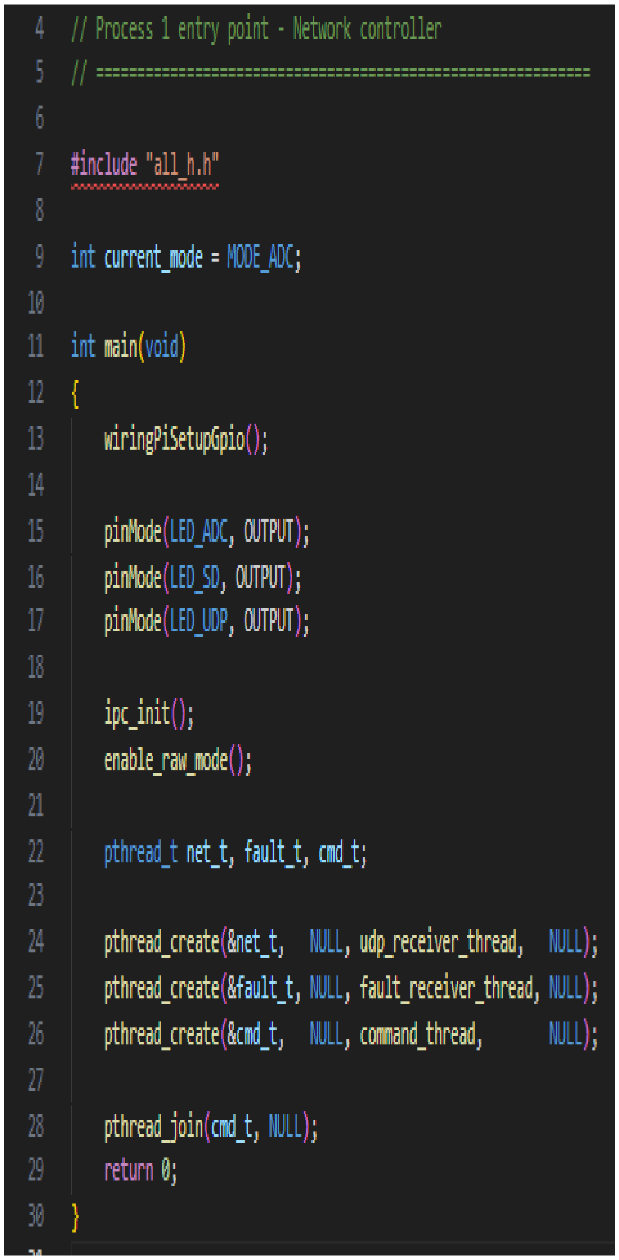
```c
// Data structure definitions for multi-node power monitoring system
// ============================================================

#ifndef STRUCTS_H
#define STRUCTS_H

#include <stdint.h>
#include <stdbool.h>
#include <pthread.h>
#include "constants.h"

// ESP32 PACKET
// UDP packet format received from ESP32 nodes
typedef struct {
    uint32_t node_id;           // ESP32 node identifier (1-3)
    uint32_t cycle_id;          // RMS calculation cycle counter
    float vrms;                 // RMS voltage (V)
    float irms;                 // RMS current (A)
} __attribute__((packed)) esp_packet_t;

// INTER-PROCESS PACKET
// Packet format for IPC from Process 1 to Process 2
typedef struct {
    uint32_t cycle_id[NUM_NODES];  // Per-node cycle counters
    float vrms1, vrms2, vrms3;     // RMS voltages for nodes 1-3
    float irms1, irms2, irms3;     // RMS currents for nodes 1-3
    int node_active[NUM_NODES];    // Node activity flags (0=inactive, 1=active)
} sensor_packet_t;

// VOLTAGE DATA
// Processed voltage data with fault classification
typedef struct {
    float vrms1, vrms2, vrms3;     // RMS voltages (V)
    float vpeak1, vpeak2, vpeak3;  // Peak voltages (V)
    int status1, status2, status3; // Fault status (NORMAL/SAG/SWELL)
    uint64_t timestamp;            // Processing timestamp (ms)
} voltage_data_t;

// CURRENT DATA
// Processed current data with fault classification
typedef struct {
    float irms1, irms2, irms3;     // RMS currents (A)
    float ipeak1, ipeak2, ipeak3;  // Peak currents (A)
    int status1, status2, status3; // Fault status (NORMAL/OVERCURRENT)
    uint64_t timestamp;            // Processing timestamp (ms)
} current_data_t;

// POWER DATA
// Calculated power from coherent RMS snapshot
typedef struct {
    float p1, p2, p3;               // Power per node (W)
    bool is_valid;                  // Data validity flag
    uint64_t timestamp;             // Calculation timestamp (ms)
} power_data_t;

// SHARED SYSTEM STATE
// Thread-safe shared state for Process 2
typedef struct {
    pthread_mutex_t lock;           // Mutex for thread-safe access
    pthread_cond_t data_ready;      // Condition variable for thread synchronization

    uint32_t cycle_id[NUM_NODES];   // Per-node cycle counters
    float vrms1, vrms2, vrms3;      // Raw RMS voltages from Process 1
    float irms1, irms2, irms3;      // Raw RMS currents from Process 1
    int node_active[NUM_NODES];     // Node activity status

    voltage_data_t vdata;           // Processed voltage data
    current_data_t idata;           // Processed current data
    power_data_t pdata;             // Calculated power data
} system_data_t;

#endif
```



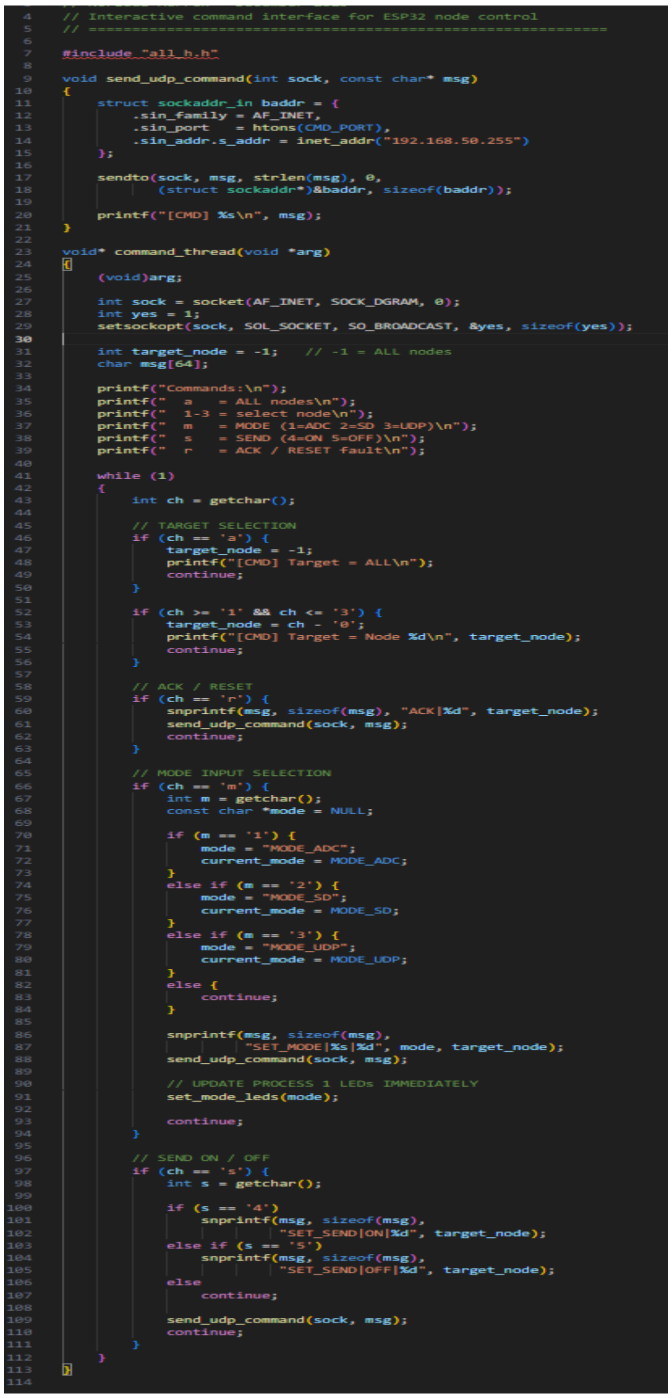
```c
// Function declarations for Process 1 (Network Controller)
// ============================================================

#ifndef PROCESS1_FUNCTIONS_H
#define PROCESS1_FUNCTIONS_H

#include <stdint.h>
#include "structs.h"

// NETWORK FUNCTIONS
// Receive UDP packets from ESP32 nodes on port 5005
void* udp_receiver_thread(void *arg);

// Receive FAULT event messages from ESP32 nodes on CMD_PORT
void* fault_receiver_thread(void *arg);

// Update combined packet with data from a single ESP32 node
void update_combined_packet(const esp_packet_t *pkt);

// COMMAND FUNCTIONS
// Interactive command interface for controlling ESP32 nodes
void* command_thread(void *arg);

// Send UDP command to ESP32 nodes via broadcast
void send_udp_command(int sock, const char* msg);

// IPC FUNCTIONS
// Initialize POSIX shared memory and semaphore for inter-process communication
int ipc_init(void);

// Send sensor packet from Process 1 to Process 2 via shared memory
void ipc_send_packet(const sensor_packet_t *pkt);

// UTILITY FUNCTIONS
// Get current system time in milliseconds
unsigned long long get_current_time_ms(void);

// Update mode indicator LEDs (ADC, SD, UDP)
void set_mode_leds(const char *mode);

#endif
```



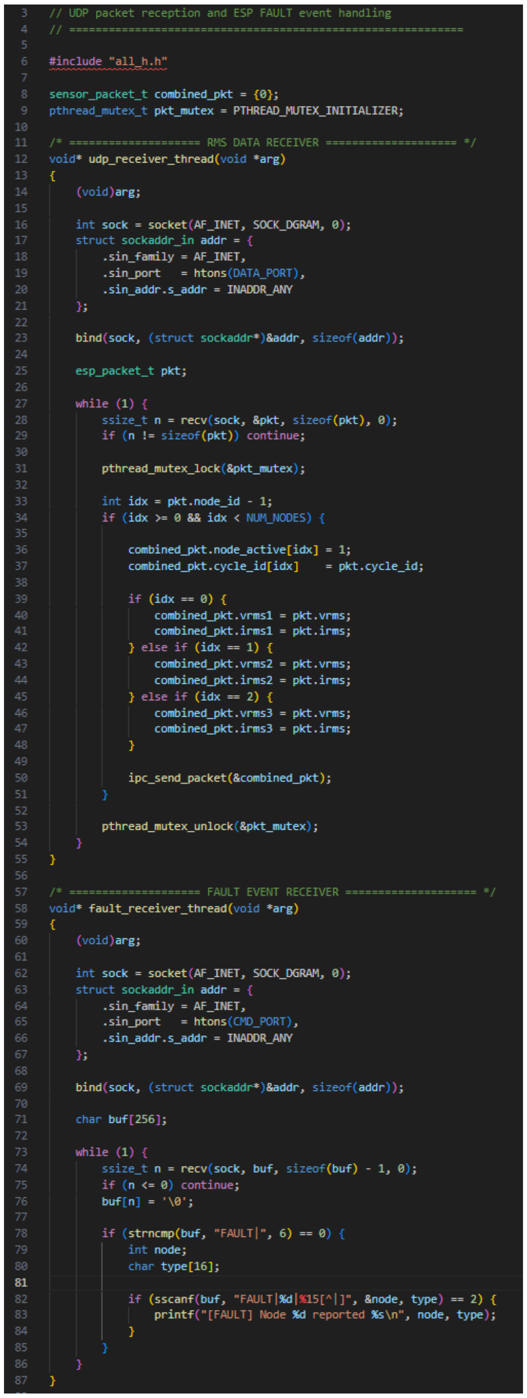
```c
// Terminal control and LED utilities for Process 1
// ==========================================================

#include "all_h.h"
#include <termios.h>
#include <unistd.h>

uint64_t GET_TIMESTAMP_MS(void)
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (uint64_t)tv.tv_sec * 1000ULL +
           (uint64_t)tv.tv_usec / 1000ULL;
}

// Enable raw terminal mode for single-key input
void enable_raw_mode(void)
{
    struct termios t;
    tcgetattr(STDIN_FILENO, &t);
    t.c_lflag &= ~(ICANON | ECHO);
    tcsetattr(STDIN_FILENO, TCSANOW, &t);
}

// Restore canonical terminal mode
void disable_raw_mode(void)
{
    struct termios t;
    tcgetattr(STDIN_FILENO, &t);
    t.c_lflag |= (ICANON | ECHO);
    tcsetattr(STDIN_FILENO, TCSANOW, &t);
}

// Update mode indicator LEDs
void set_mode_leds(const char *mode)
{
    digitalWrite(LED_ADC, strcmp(mode, "MODE_ADC") == 0);
    digitalWrite(LED_SD,  strcmp(mode, "MODE_SD")  == 0);
    digitalWrite(LED_UDP, strcmp(mode, "MODE_UDP") == 0);
}
```



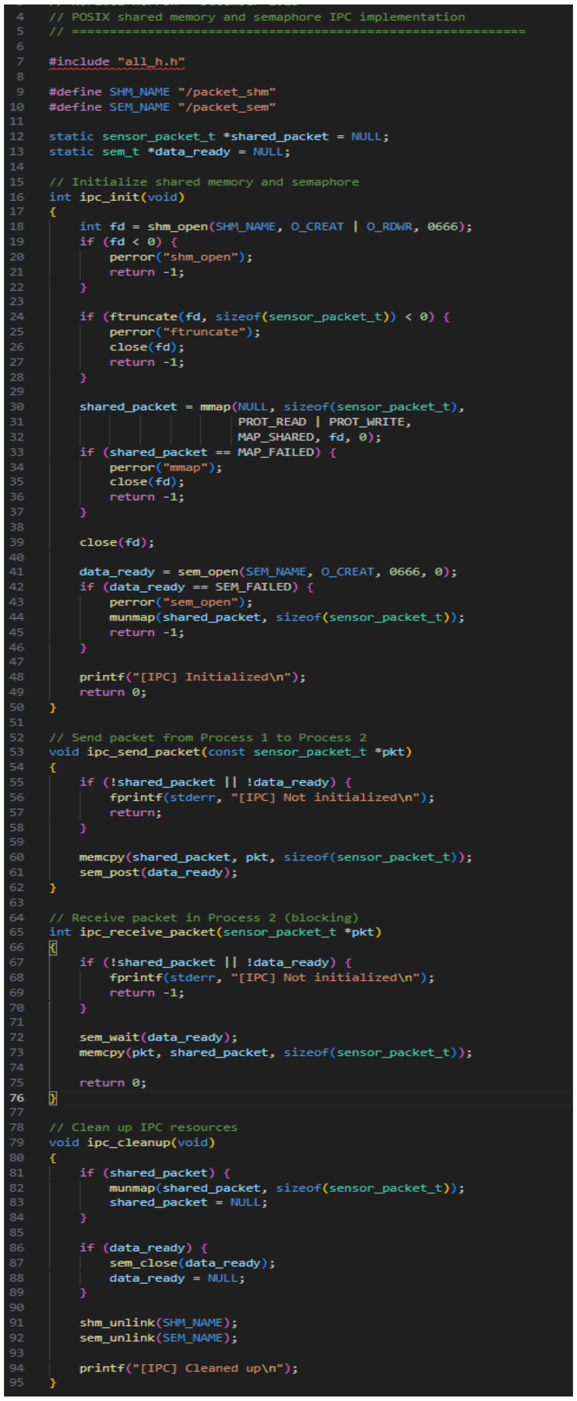
```c
// Process 1 entry point - Network controller
// ==========================================================

#include "all_h.h"

int current_mode = MODE_ADC;

int main(void)
{
    wiringPiSetupGpio();

    pinMode(LED_ADC, OUTPUT);
    pinMode(LED_SD, OUTPUT);
    pinMode(LED_UDP, OUTPUT);

    ipc_init();
    enable_raw_mode();

    pthread_t net_t, fault_t, cmd_t;

    pthread_create(&net_t,   NULL, udp_receiver_thread,   NULL);
    pthread_create(&fault_t, NULL, fault_receiver_thread, NULL);
    pthread_create(&cmd_t,   NULL, command_thread,        NULL);

    pthread_join(cmd_t, NULL);
    return 0;
}
```



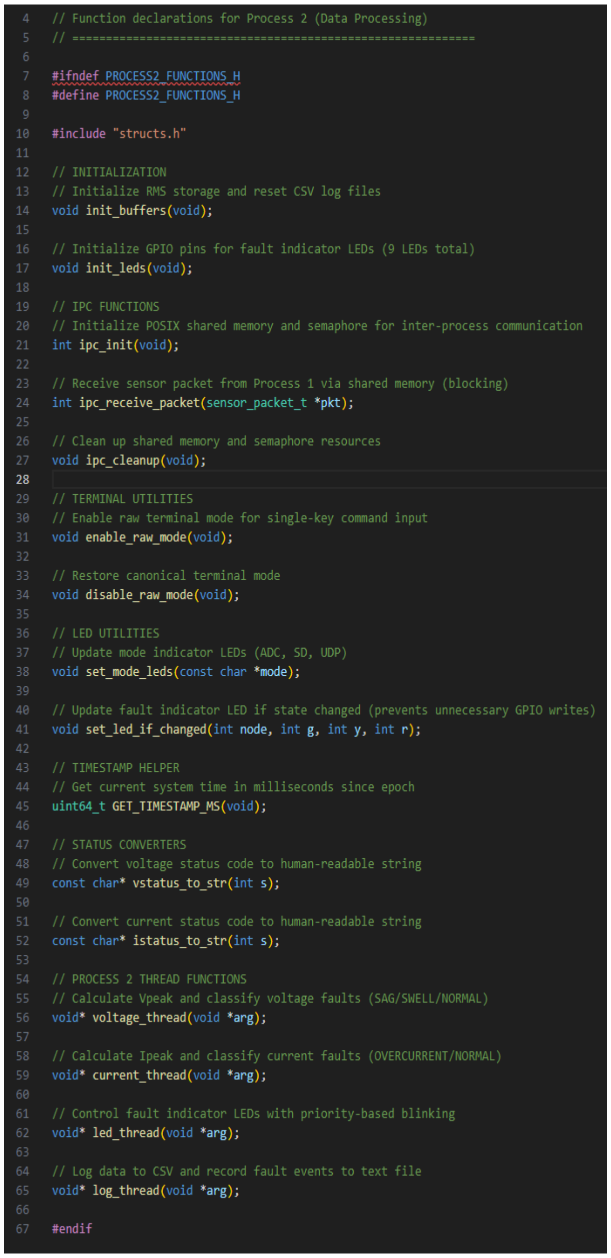
```c
// Interactive command interface for ESP32 node control
// ================================================================

#include "all_h.h"

void send_udp_command(int sock, const char* msg)
{
    struct sockaddr_in baddr = {
        .sin_family = AF_INET,
        .sin_port   = htons(CMD_PORT),
        .sin_addr.s_addr = inet_addr("192.168.50.255")
    };

    sendto(sock, msg, strlen(msg), 0,
           (struct sockaddr*)&baddr, sizeof(baddr));

    printf("[CMD] %s\n", msg);
}

void* command_thread(void *arg)
{
    (void)arg;

    int sock = socket(AF_INET, SOCK_DGRAM, 0);
    int yes = 1;
    setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &yes, sizeof(yes));

    int target_node = -1;   // -1 = ALL nodes
    char msg[64];

    printf("Commands:\n");
    printf("  a   = ALL nodes\n");
    printf("  1-3 = select node\n");
    printf("  m   = MODE (1=ADC 2=SD 3=UDP)\n");
    printf("  s   = SEND (4=ON 5=OFF)\n");
    printf("  r   = ACK / RESET fault\n");

    while (1)
    {
        int ch = getchar();

        // TARGET SELECTION
        if (ch == 'a') {
            target_node = -1;
            printf("[CMD] Target = ALL\n");
            continue;
        }

        if (ch >= '1' && ch <= '3') {
            target_node = ch - '0';
            printf("[CMD] Target = Node %d\n", target_node);
            continue;
        }

        // ACK / RESET
        if (ch == 'r') {
            snprintf(msg, sizeof(msg), "ACK|%d", target_node);
            send_udp_command(sock, msg);
            continue;
        }

        // MODE INPUT SELECTION
        if (ch == 'm') {
            int m = getchar();
            const char *mode = NULL;

            if (m == '1') {
                mode = "MODE_ADC";
                current_mode = MODE_ADC;
            }
            else if (m == '2') {
                mode = "MODE_SD";
                current_mode = MODE_SD;
            }
            else if (m == '3') {
                mode = "MODE_UDP";
                current_mode = MODE_UDP;
            }
            else {
                continue;
            }

            snprintf(msg, sizeof(msg),
                     "SET_MODE|%s|%d", mode, target_node);
            send_udp_command(sock, msg);

            // UPDATE PROCESS 1 LEDs IMMEDIATELY
            set_mode_leds(mode);

            continue;
        }

        // SEND ON / OFF
        if (ch == 's') {
            int s = getchar();

            if (s == '4')
                snprintf(msg, sizeof(msg),
                         "SET_SEND|ON|%d", target_node);
            else if (s == '5')
                snprintf(msg, sizeof(msg),
                         "SET_SEND|OFF|%d", target_node);
            else
                continue;

            send_udp_command(sock, msg);
            continue;
        }
    }
}
```



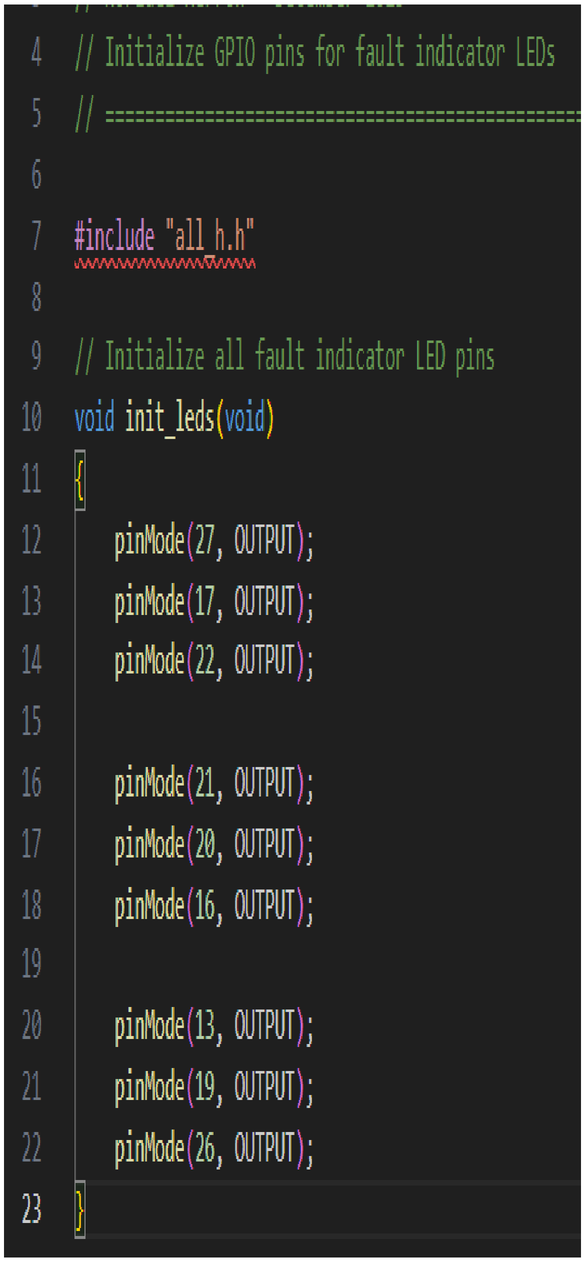
```c
// UDP packet reception and ESP FAULT event handling
// ============================================================

#include "all_h.h"

sensor_packet_t combined_pkt = {0};
pthread_mutex_t pkt_mutex = PTHREAD_MUTEX_INITIALIZER;

/* =================== RMS DATA RECEIVER =================== */
void* udp_receiver_thread(void *arg)
{
    (void)arg;

    int sock = socket(AF_INET, SOCK_DGRAM, 0);
    struct sockaddr_in addr = {
        .sin_family = AF_INET,
        .sin_port   = htons(DATA_PORT),
        .sin_addr.s_addr = INADDR_ANY
    };

    bind(sock, (struct sockaddr*)&addr, sizeof(addr));

    esp_packet_t pkt;

    while (1) {
        ssize_t n = recv(sock, &pkt, sizeof(pkt), 0);
        if (n != sizeof(pkt)) continue;

        pthread_mutex_lock(&pkt_mutex);

        int idx = pkt.node_id - 1;
        if (idx >= 0 && idx < NUM_NODES) {

            combined_pkt.node_active[idx] = 1;
            combined_pkt.cycle_id[idx]    = pkt.cycle_id;

            if (idx == 0) {
                combined_pkt.vrms1 = pkt.vrms;
                combined_pkt.irms1 = pkt.irms;
            } else if (idx == 1) {
                combined_pkt.vrms2 = pkt.vrms;
                combined_pkt.irms2 = pkt.irms;
            } else if (idx == 2) {
                combined_pkt.vrms3 = pkt.vrms;
                combined_pkt.irms3 = pkt.irms;
            }

            ipc_send_packet(&combined_pkt);
        }

        pthread_mutex_unlock(&pkt_mutex);
    }
}

/* =================== FAULT EVENT RECEIVER =================== */
void* fault_receiver_thread(void *arg)
{
    (void)arg;

    int sock = socket(AF_INET, SOCK_DGRAM, 0);
    struct sockaddr_in addr = {
        .sin_family = AF_INET,
        .sin_port   = htons(CMD_PORT),
        .sin_addr.s_addr = INADDR_ANY
    };

    bind(sock, (struct sockaddr*)&addr, sizeof(addr));

    char buf[256];

    while (1) {
        ssize_t n = recv(sock, buf, sizeof(buf) - 1, 0);
        if (n <= 0) continue;
        buf[n] = '\0';

        if (strncmp(buf, "FAULT|", 6) == 0) {
            int node;
            char type[16];

            if (sscanf(buf, "FAULT|%d|%15[^|]", &node, type) == 2) {
                printf("[FAULT] Node %d reported %s\n", node, type);
            }
        }
    }
}
```



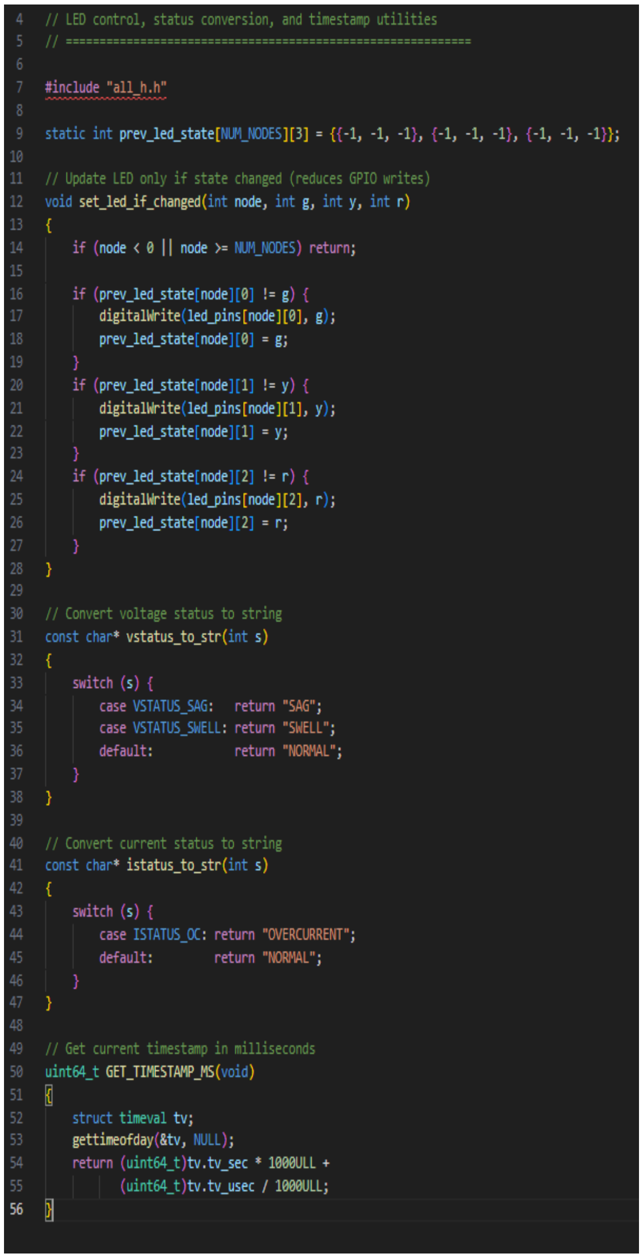
```c
// POSIX shared memory and semaphore IPC implementation
// ============================================================

#include "all_h.h"

#define SHM_NAME "/packet_shm"
#define SEM_NAME "/packet_sem"

static sensor_packet_t *shared_packet = NULL;
static sem_t *data_ready = NULL;

// Initialize shared memory and semaphore
int ipc_init(void)
{
    int fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
    if (fd < 0) {
        perror("shm_open");
        return -1;
    }

    if (ftruncate(fd, sizeof(sensor_packet_t)) < 0) {
        perror("ftruncate");
        close(fd);
        return -1;
    }

    shared_packet = mmap(NULL, sizeof(sensor_packet_t),
                         PROT_READ | PROT_WRITE,
                         MAP_SHARED, fd, 0);
    if (shared_packet == MAP_FAILED) {
        perror("mmap");
        close(fd);
        return -1;
    }

    close(fd);

    data_ready = sem_open(SEM_NAME, O_CREAT, 0666, 0);
    if (data_ready == SEM_FAILED) {
        perror("sem_open");
        munmap(shared_packet, sizeof(sensor_packet_t));
        return -1;
    }

    printf("[IPC] Initialized\n");
    return 0;
}

// Send packet from Process 1 to Process 2
void ipc_send_packet(const sensor_packet_t *pkt)
{
    if (!shared_packet || !data_ready) {
        fprintf(stderr, "[IPC] Not initialized\n");
        return;
    }

    memcpy(shared_packet, pkt, sizeof(sensor_packet_t));
    sem_post(data_ready);
}

// Receive packet in Process 2 (blocking)
int ipc_receive_packet(sensor_packet_t *pkt)
{
    if (!shared_packet || !data_ready) {
        fprintf(stderr, "[IPC] Not initialized\n");
        return -1;
    }

    sem_wait(data_ready);
    memcpy(pkt, shared_packet, sizeof(sensor_packet_t));

    return 0;
}

// Clean up IPC resources
void ipc_cleanup(void)
{
    if (shared_packet) {
        munmap(shared_packet, sizeof(sensor_packet_t));
        shared_packet = NULL;
    }

    if (data_ready) {
        sem_close(data_ready);
        data_ready = NULL;
    }

    shm_unlink(SHM_NAME);
    sem_unlink(SEM_NAME);

    printf("[IPC] Cleaned up\n");
}
```



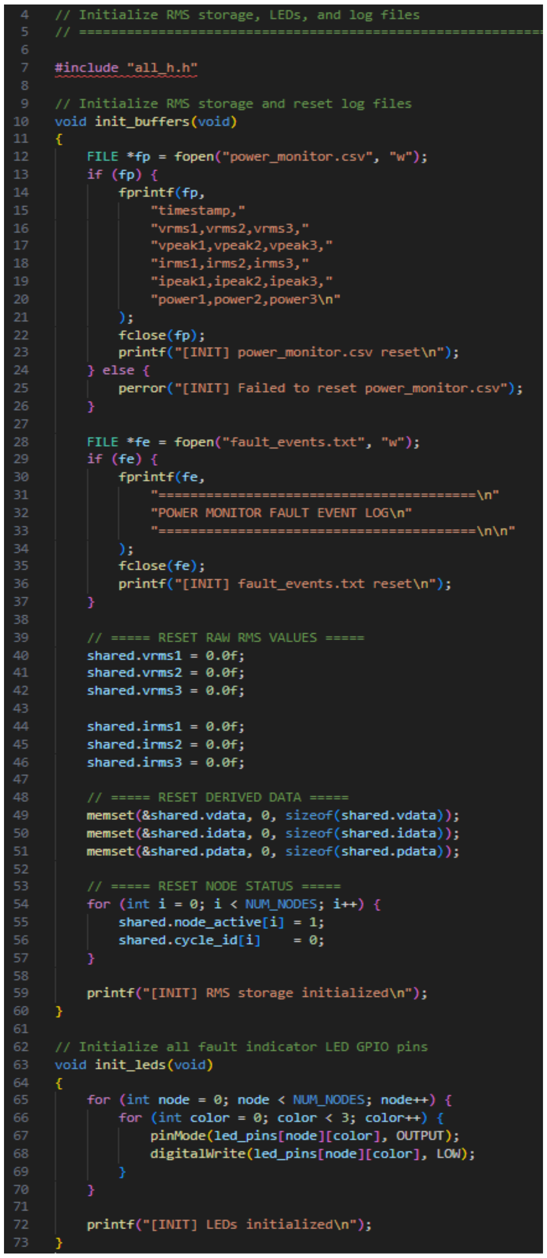
```c
// Function declarations for Process 2 (Data Processing)
// ============================================================

#ifndef PROCESS2_FUNCTIONS_H
#define PROCESS2_FUNCTIONS_H

#include "structs.h"

// INITIALIZATION
// Initialize RMS storage and reset CSV log files
void init_buffers(void);

// Initialize GPIO pins for fault indicator LEDs (9 LEDs total)
void init_leds(void);

// IPC FUNCTIONS
// Initialize POSIX shared memory and semaphore for inter-process communication
int ipc_init(void);

// Receive sensor packet from Process 1 via shared memory (blocking)
int ipc_receive_packet(sensor_packet_t *pkt);

// Clean up shared memory and semaphore resources
void ipc_cleanup(void);

// TERMINAL UTILITIES
// Enable raw terminal mode for single-key command input
void enable_raw_mode(void);

// Restore canonical terminal mode
void disable_raw_mode(void);

// LED UTILITIES
// Update mode indicator LEDs (ADC, SD, UDP)
void set_mode_leds(const char *mode);

// Update fault indicator LED if state changed (prevents unnecessary GPIO writes)
void set_led_if_changed(int node, int g, int y, int r);

// TIMESTAMP HELPER
// Get current system time in milliseconds since epoch
uint64_t GET_TIMESTAMP_MS(void);

// STATUS CONVERTERS
// Convert voltage status code to human-readable string
const char* vstatus_to_str(int s);

// Convert current status code to human-readable string
const char* istatus_to_str(int s);

// PROCESS 2 THREAD FUNCTIONS
// Calculate Vpeak and classify voltage faults (SAG/SWELL/NORMAL)
void* voltage_thread(void *arg);

// Calculate Ipeak and classify current faults (OVERCURRENT/NORMAL)
void* current_thread(void *arg);

// Control fault indicator LEDs with priority-based blinking
void* led_thread(void *arg);

// Log data to CSV and record fault events to text file
void* log_thread(void *arg);

#endif
```



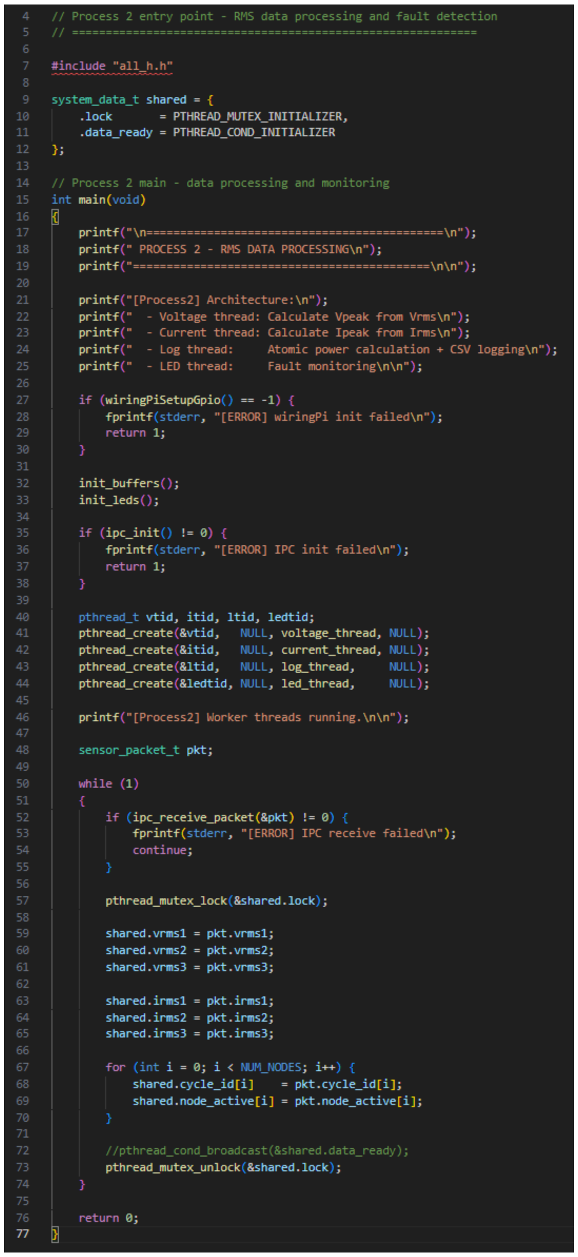
```c
// Initialize GPIO pins for fault indicator LEDs
// ================================================

#include "all_h.h"

// Initialize all fault indicator LED pins
void init_leds(void)
{
    pinMode(27, OUTPUT);
    pinMode(17, OUTPUT);
    pinMode(22, OUTPUT);

    pinMode(21, OUTPUT);
    pinMode(20, OUTPUT);
    pinMode(16, OUTPUT);

    pinMode(13, OUTPUT);
    pinMode(19, OUTPUT);
    pinMode(26, OUTPUT);
}
```



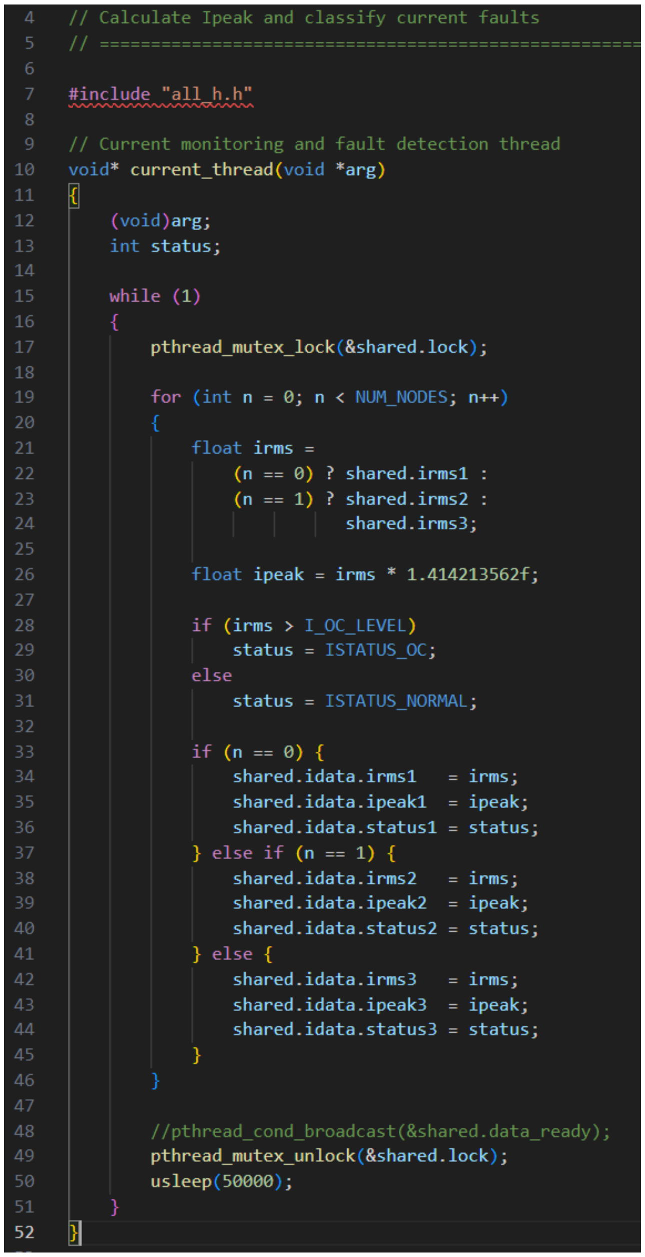
```c
// LED control, status conversion, and timestamp utilities
// ============================================================

#include "all_h.h"

static int prev_led_state[NUM_NODES][3] = {{-1, -1, -1}, {-1, -1, -1}, {-1, -1, -1}};

// Update LED only if state changed (reduces GPIO writes)
void set_led_if_changed(int node, int g, int y, int r)
{
    if (node < 0 || node >= NUM_NODES) return;

    if (prev_led_state[node][0] != g) {
        digitalWrite(led_pins[node][0], g);
        prev_led_state[node][0] = g;
    }
    if (prev_led_state[node][1] != y) {
        digitalWrite(led_pins[node][1], y);
        prev_led_state[node][1] = y;
    }
    if (prev_led_state[node][2] != r) {
        digitalWrite(led_pins[node][2], r);
        prev_led_state[node][2] = r;
    }
}

// Convert voltage status to string
const char* vstatus_to_str(int s)
{
    switch (s) {
        case VSTATUS_SAG:   return "SAG";
        case VSTATUS_SWELL: return "SWELL";
        default:            return "NORMAL";
    }
}

// Convert current status to string
const char* istatus_to_str(int s)
{
    switch (s) {
        case ISTATUS_OC: return "OVERCURRENT";
        default:         return "NORMAL";
    }
}

// Get current timestamp in milliseconds
uint64_t GET_TIMESTAMP_MS(void)
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (uint64_t)tv.tv_sec * 1000ULL +
           (uint64_t)tv.tv_usec / 1000ULL;
}
```



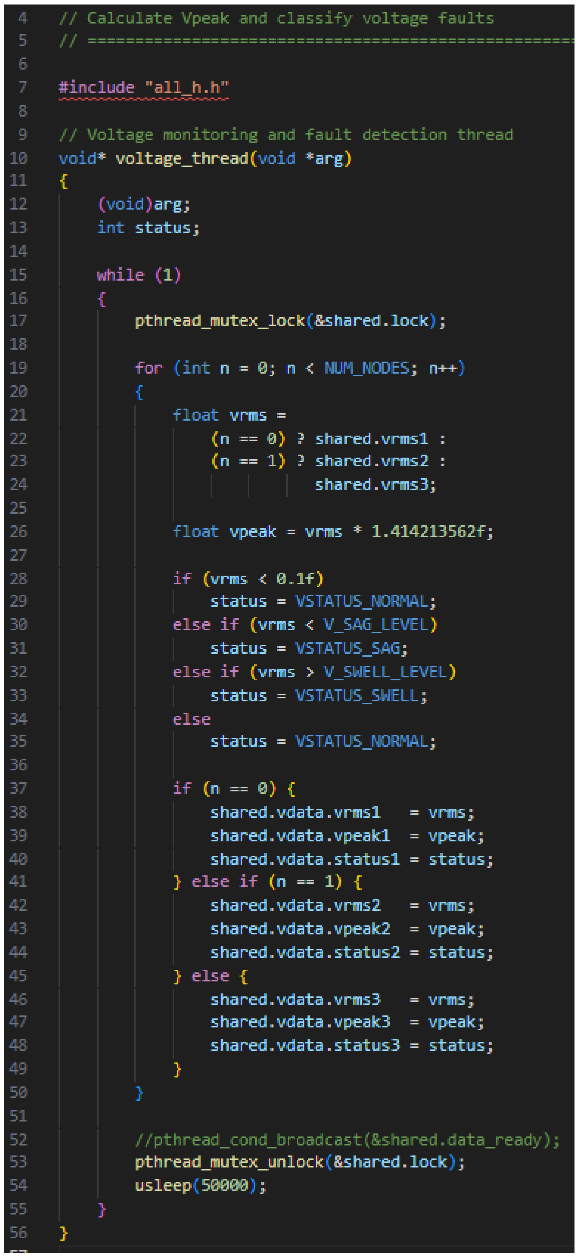
```c
// Initialize RMS storage, LEDs, and log files
// ============================================================

#include "all_h.h"

// Initialize RMS storage and reset log files
void init_buffers(void)
{
    FILE *fp = fopen("power_monitor.csv", "w");
    if (fp) {
        fprintf(fp,
            "timestamp,"
            "vrms1,vrms2,vrms3,"
            "vpeak1,vpeak2,vpeak3,"
            "irms1,irms2,irms3,"
            "ipeak1,ipeak2,ipeak3,"
            "power1,power2,power3\n"
        );
        fclose(fp);
        printf("[INIT] power_monitor.csv reset\n");
    } else {
        perror("[INIT] Failed to reset power_monitor.csv");
    }

    FILE *fe = fopen("fault_events.txt", "w");
    if (fe) {
        fprintf(fe,
            "========================================\n"
            "POWER MONITOR FAULT EVENT LOG\n"
            "========================================\n\n"
        );
        fclose(fe);
        printf("[INIT] fault_events.txt reset\n");
    }

    // ===== RESET RAW RMS VALUES =====
    shared.vrms1 = 0.0f;
    shared.vrms2 = 0.0f;
    shared.vrms3 = 0.0f;

    shared.irms1 = 0.0f;
    shared.irms2 = 0.0f;
    shared.irms3 = 0.0f;

    // ===== RESET DERIVED DATA =====
    memset(&shared.vdata, 0, sizeof(shared.vdata));
    memset(&shared.idata, 0, sizeof(shared.idata));
    memset(&shared.pdata, 0, sizeof(shared.pdata));

    // ===== RESET NODE STATUS =====
    for (int i = 0; i < NUM_NODES; i++) {
        shared.node_active[i] = 1;
        shared.cycle_id[i]    = 0;
    }

    printf("[INIT] RMS storage initialized\n");
}

// Initialize all fault indicator LED GPIO pins
void init_leds(void)
{
    for (int node = 0; node < NUM_NODES; node++) {
        for (int color = 0; color < 3; color++) {
            pinMode(led_pins[node][color], OUTPUT);
            digitalWrite(led_pins[node][color], LOW);
        }
    }

    printf("[INIT] LEDs initialized\n");
}
```



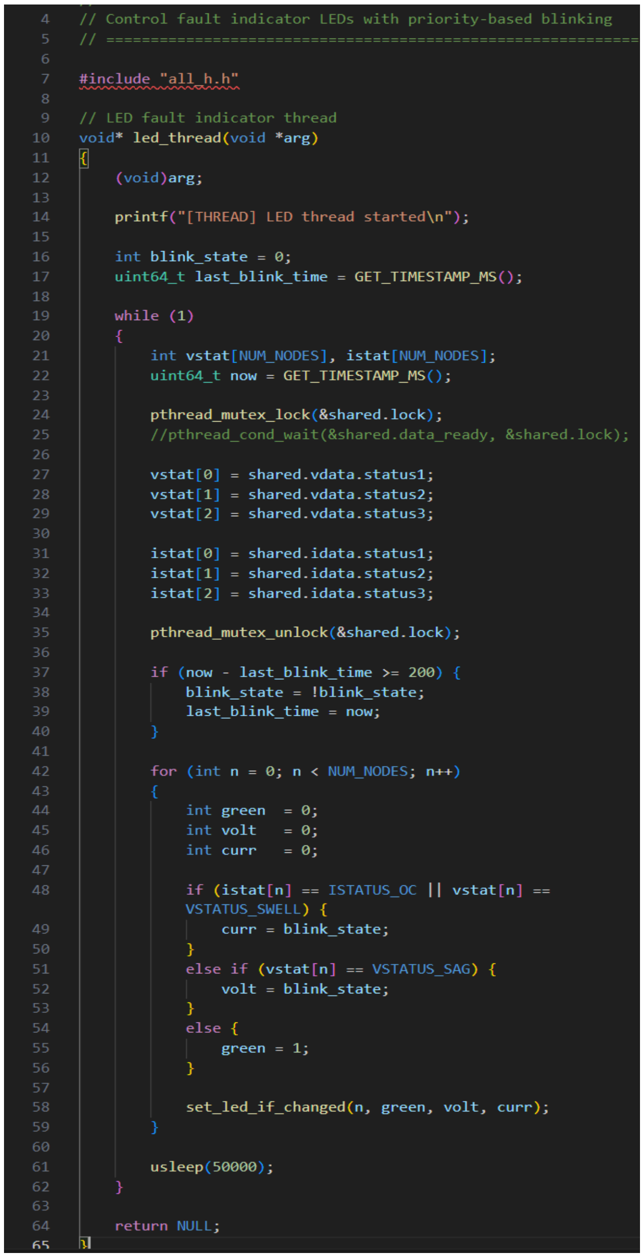
```c
// Process 2 entry point - RMS data processing and fault detection
// ============================================================

#include "all_h.h"

system_data_t shared = {
    .lock       = PTHREAD_MUTEX_INITIALIZER,
    .data_ready = PTHREAD_COND_INITIALIZER
};

// Process 2 main - data processing and monitoring
int main(void)
{
    printf("\n============================================\n");
    printf(" PROCESS 2 - RMS DATA PROCESSING\n");
    printf("============================================\n\n");

    printf("[Process2] Architecture:\n");
    printf("  - Voltage thread: Calculate Vpeak from Vrms\n");
    printf("  - Current thread: Calculate Ipeak from Irms\n");
    printf("  - Log thread:     Atomic power calculation + CSV logging\n");
    printf("  - LED thread:     Fault monitoring\n\n");

    if (wiringPiSetupGpio() == -1) {
        fprintf(stderr, "[ERROR] wiringPi init failed\n");
        return 1;
    }

    init_buffers();
    init_leds();

    if (ipc_init() != 0) {
        fprintf(stderr, "[ERROR] IPC init failed\n");
        return 1;
    }

    pthread_t vtid, itid, ltid, ledtid;
    pthread_create(&vtid,   NULL, voltage_thread, NULL);
    pthread_create(&itid,   NULL, current_thread, NULL);
    pthread_create(&ltid,   NULL, log_thread,     NULL);
    pthread_create(&ledtid, NULL, led_thread,     NULL);

    printf("[Process2] Worker threads running.\n\n");

    sensor_packet_t pkt;

    while (1)
    {
        if (ipc_receive_packet(&pkt) != 0) {
            fprintf(stderr, "[ERROR] IPC receive failed\n");
            continue;
        }

        pthread_mutex_lock(&shared.lock);

        shared.vrms1 = pkt.vrms1;
        shared.vrms2 = pkt.vrms2;
        shared.vrms3 = pkt.vrms3;

        shared.irms1 = pkt.irms1;
        shared.irms2 = pkt.irms2;
        shared.irms3 = pkt.irms3;

        for (int i = 0; i < NUM_NODES; i++) {
            shared.cycle_id[i]    = pkt.cycle_id[i];
            shared.node_active[i] = pkt.node_active[i];
        }

        //pthread_cond_broadcast(&shared.data_ready);
        pthread_mutex_unlock(&shared.lock);
    }

    return 0;
}
```



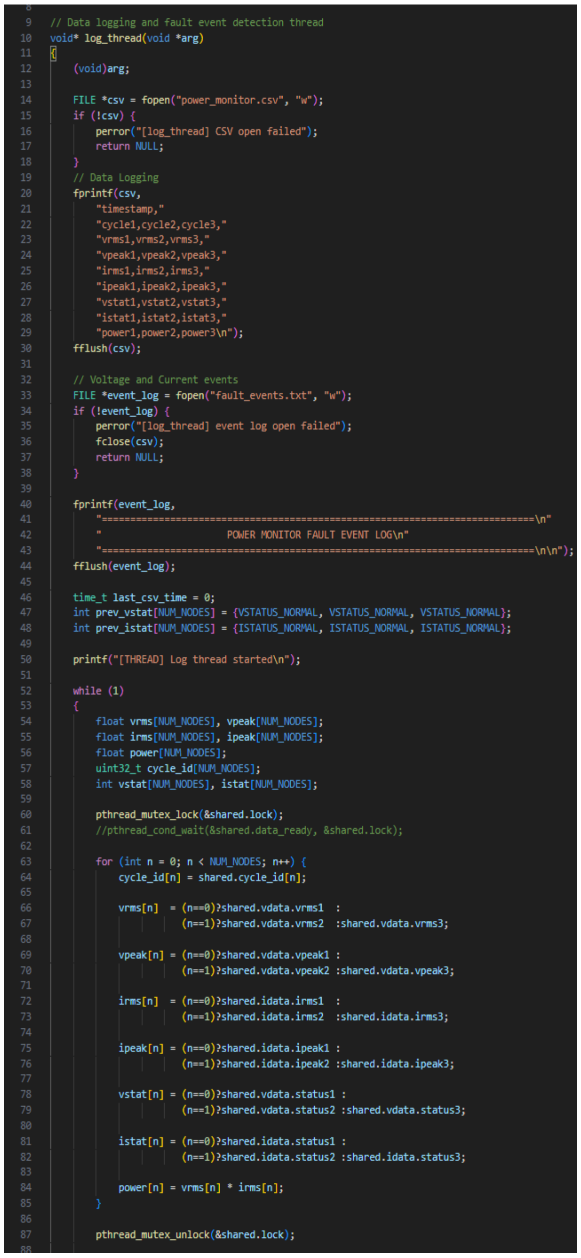
```c
// Calculate Ipeak and classify current faults
// ============================================================

#include "all_h.h"

// Current monitoring and fault detection thread
void* current_thread(void *arg)
{
    (void)arg;
    int status;

    while (1)
    {
        pthread_mutex_lock(&shared.lock);

        for (int n = 0; n < NUM_NODES; n++)
        {
            float irms =
                (n == 0) ? shared.irms1 :
                (n == 1) ? shared.irms2 :
                           shared.irms3;

            float ipeak = irms * 1.414213562f;

            if (irms > I_OC_LEVEL)
                status = ISTATUS_OC;
            else
                status = ISTATUS_NORMAL;

            if (n == 0) {
                shared.idata.irms1   = irms;
                shared.idata.ipeak1  = ipeak;
                shared.idata.status1 = status;
            } else if (n == 1) {
                shared.idata.irms2   = irms;
                shared.idata.ipeak2  = ipeak;
                shared.idata.status2 = status;
            } else {
                shared.idata.irms3   = irms;
                shared.idata.ipeak3  = ipeak;
                shared.idata.status3 = status;
            }
        }

        //pthread_cond_broadcast(&shared.data_ready);
        pthread_mutex_unlock(&shared.lock);
        usleep(50000);
    }
}
```



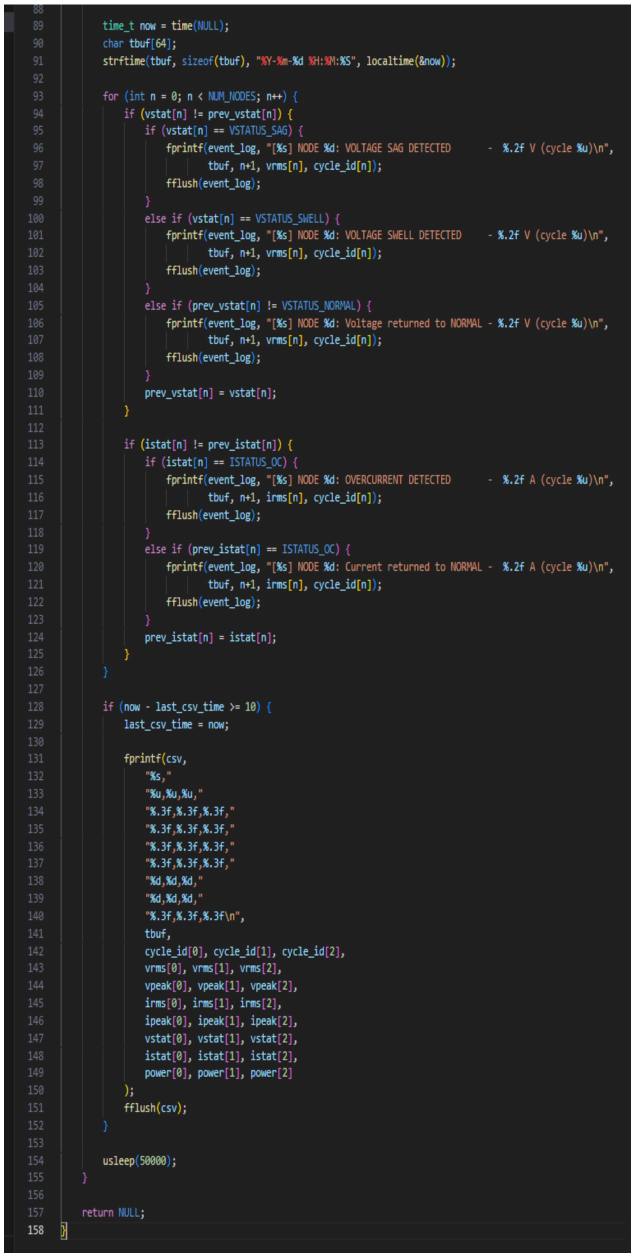
```c
// Calculate Vpeak and classify voltage faults
// ======================================================

#include "all_h.h"

// Voltage monitoring and fault detection thread
void* voltage_thread(void *arg)
{
    (void)arg;
    int status;

    while (1)
    {
        pthread_mutex_lock(&shared.lock);

        for (int n = 0; n < NUM_NODES; n++)
        {
            float vrms =
                (n == 0) ? shared.vrms1 :
                (n == 1) ? shared.vrms2 :
                           shared.vrms3;

            float vpeak = vrms * 1.414213562f;

            if (vrms < 0.1f)
                status = VSTATUS_NORMAL;
            else if (vrms < V_SAG_LEVEL)
                status = VSTATUS_SAG;
            else if (vrms > V_SWELL_LEVEL)
                status = VSTATUS_SWELL;
            else
                status = VSTATUS_NORMAL;

            if (n == 0) {
                shared.vdata.vrms1   = vrms;
                shared.vdata.vpeak1  = vpeak;
                shared.vdata.status1 = status;
            } else if (n == 1) {
                shared.vdata.vrms2   = vrms;
                shared.vdata.vpeak2  = vpeak;
                shared.vdata.status2 = status;
            } else {
                shared.vdata.vrms3   = vrms;
                shared.vdata.vpeak3  = vpeak;
                shared.vdata.status3 = status;
            }
        }

        //pthread_cond_broadcast(&shared.data_ready);
        pthread_mutex_unlock(&shared.lock);
        usleep(50000);
    }
}
```



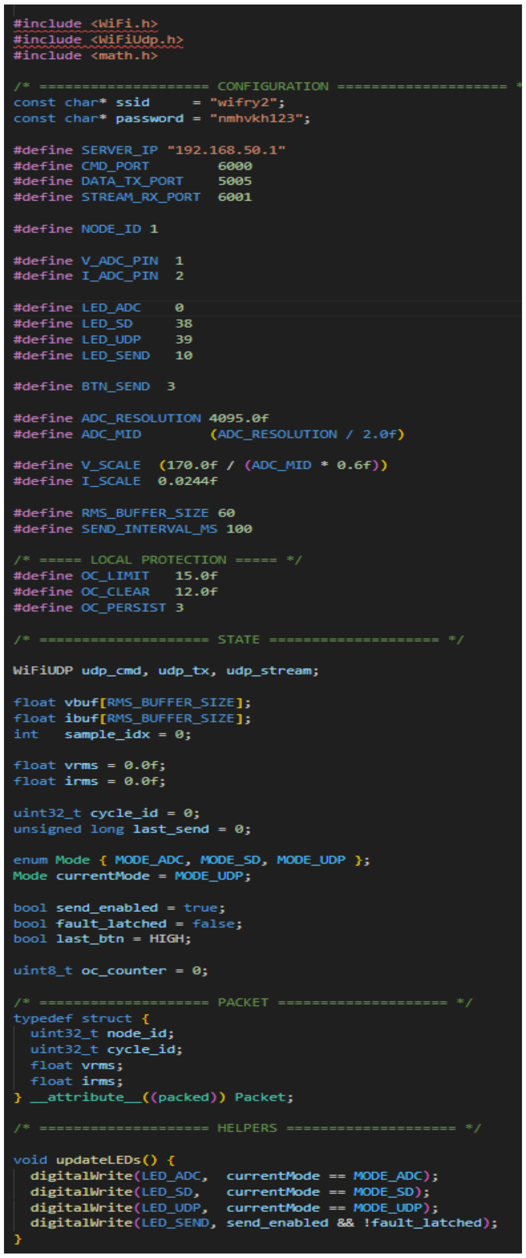
```c
// Control fault indicator LEDs with priority-based blinking
// ============================================================

#include "all_h.h"

// LED fault indicator thread
void* led_thread(void *arg)
{
    (void)arg;

    printf("[THREAD] LED thread started\n");

    int blink_state = 0;
    uint64_t last_blink_time = GET_TIMESTAMP_MS();

    while (1)
    {
        int vstat[NUM_NODES], istat[NUM_NODES];
        uint64_t now = GET_TIMESTAMP_MS();

        pthread_mutex_lock(&shared.lock);
        //pthread_cond_wait(&shared.data_ready, &shared.lock);

        vstat[0] = shared.vdata.status1;
        vstat[1] = shared.vdata.status2;
        vstat[2] = shared.vdata.status3;

        istat[0] = shared.idata.status1;
        istat[1] = shared.idata.status2;
        istat[2] = shared.idata.status3;

        pthread_mutex_unlock(&shared.lock);

        if (now - last_blink_time >= 200) {
            blink_state = !blink_state;
            last_blink_time = now;
        }

        for (int n = 0; n < NUM_NODES; n++)
        {
            int green  = 0;
            int volt   = 0;
            int curr   = 0;

            if (istat[n] == ISTATUS_OC || vstat[n] ==
            VSTATUS_SWELL) {
                curr = blink_state;
            }
            else if (vstat[n] == VSTATUS_SAG) {
                volt = blink_state;
            }
            else {
                green = 1;
            }

            set_led_if_changed(n, green, volt, curr);
        }

        usleep(50000);
    }

    return NULL;
}
```



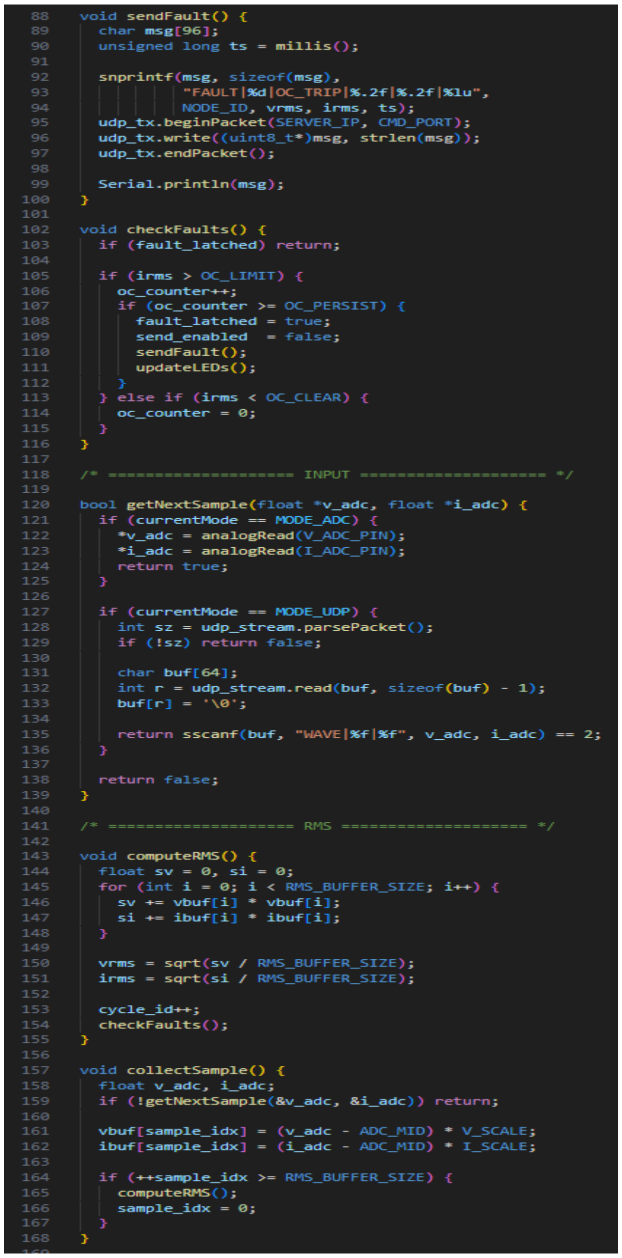
```c
// Data logging and fault event detection thread
void* log_thread(void *arg)
{
    (void)arg;

    FILE *csv = fopen("power_monitor.csv", "w");
    if (!csv) {
        perror("[log_thread] CSV open failed");
        return NULL;
    }
    // Data Logging
    fprintf(csv,
        "timestamp,"
        "cycle1,cycle2,cycle3,"
        "vrms1,vrms2,vrms3,"
        "vpeak1,vpeak2,vpeak3,"
        "irms1,irms2,irms3,"
        "ipeak1,ipeak2,ipeak3,"
        "vstat1,vstat2,vstat3,"
        "istat1,istat2,istat3,"
        "power1,power2,power3\n");
    fflush(csv);

    // Voltage and Current events
    FILE *event_log = fopen("fault_events.txt", "w");
    if (!event_log) {
        perror("[log_thread] event log open failed");
        fclose(csv);
        return NULL;
    }

    fprintf(event_log,
        "============================================================================\n"
        "                      POWER MONITOR FAULT EVENT LOG\n"
        "============================================================================\n\n");
    fflush(event_log);

    time_t last_csv_time = 0;
    int prev_vstat[NUM_NODES] = {VSTATUS_NORMAL, VSTATUS_NORMAL, VSTATUS_NORMAL};
    int prev_istat[NUM_NODES] = {ISTATUS_NORMAL, ISTATUS_NORMAL, ISTATUS_NORMAL};

    printf("[THREAD] Log thread started\n");

    while (1)
    {
        float vrms[NUM_NODES], vpeak[NUM_NODES];
        float irms[NUM_NODES], ipeak[NUM_NODES];
        float power[NUM_NODES];
        uint32_t cycle_id[NUM_NODES];
        int vstat[NUM_NODES], istat[NUM_NODES];

        pthread_mutex_lock(&shared.lock);
        //pthread_cond_wait(&shared.data_ready, &shared.lock);

        for (int n = 0; n < NUM_NODES; n++) {
            cycle_id[n] = shared.cycle_id[n];

            vrms[n]  = (n==0)?shared.vdata.vrms1  :
                       (n==1)?shared.vdata.vrms2  :shared.vdata.vrms3;

            vpeak[n] = (n==0)?shared.vdata.vpeak1 :
                       (n==1)?shared.vdata.vpeak2 :shared.vdata.vpeak3;

            irms[n]  = (n==0)?shared.idata.irms1  :
                       (n==1)?shared.idata.irms2  :shared.idata.irms3;

            ipeak[n] = (n==0)?shared.idata.ipeak1 :
                       (n==1)?shared.idata.ipeak2 :shared.idata.ipeak3;

            vstat[n] = (n==0)?shared.vdata.status1 :
                       (n==1)?shared.vdata.status2 :shared.vdata.status3;

            istat[n] = (n==0)?shared.idata.status1 :
                       (n==1)?shared.idata.status2 :shared.idata.status3;

            power[n] = vrms[n] * irms[n];
        }

        pthread_mutex_unlock(&shared.lock);
```



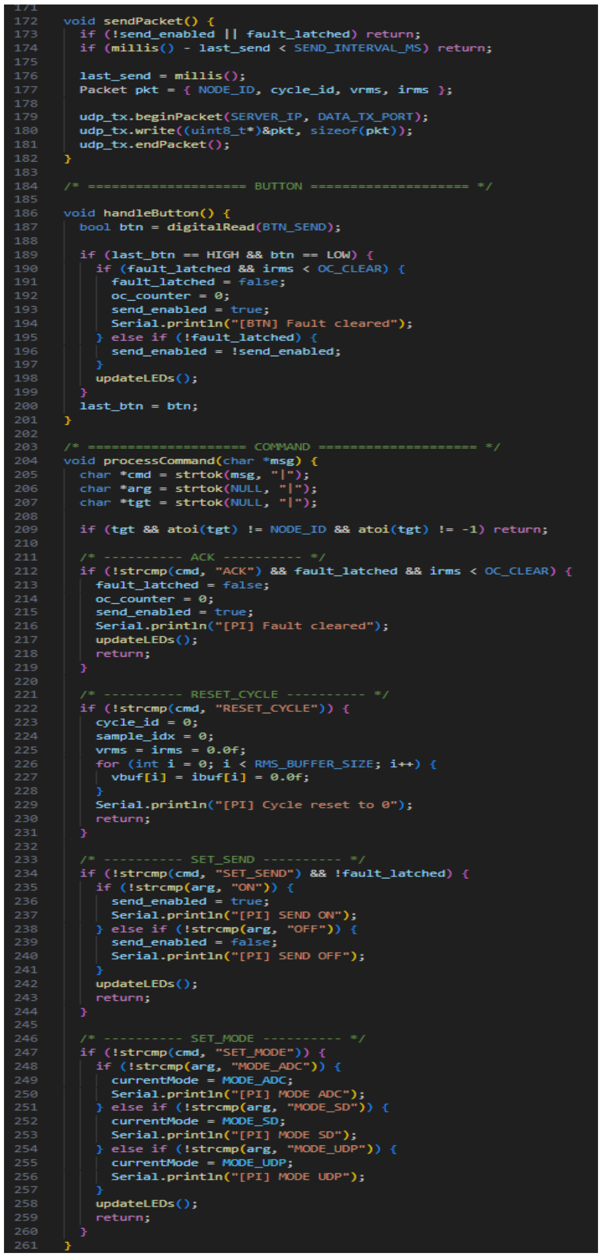
```c

        time_t now = time(NULL);
        char tbuf[64];
        strftime(tbuf, sizeof(tbuf), "%Y-%m-%d %H:%M:%S", localtime(&now));

        for (int n = 0; n < NUM_NODES; n++) {
            if (vstat[n] != prev_vstat[n]) {
                if (vstat[n] == VSTATUS_SAG) {
                    fprintf(event_log, "[%s] NODE %d: VOLTAGE SAG DETECTED       -  %.2f V (cycle %u)\n",
                            tbuf, n+1, vrms[n], cycle_id[n]);
                    fflush(event_log);
                }
                else if (vstat[n] == VSTATUS_SWELL) {
                    fprintf(event_log, "[%s] NODE %d: VOLTAGE SWELL DETECTED     - %.2f V (cycle %u)\n",
                            tbuf, n+1, vrms[n], cycle_id[n]);
                    fflush(event_log);
                }
                else if (prev_vstat[n] != VSTATUS_NORMAL) {
                    fprintf(event_log, "[%s] NODE %d: Voltage returned to NORMAL - %.2f V (cycle %u)\n",
                            tbuf, n+1, vrms[n], cycle_id[n]);
                    fflush(event_log);
                }
                prev_vstat[n] = vstat[n];
            }

            if (istat[n] != prev_istat[n]) {
                if (istat[n] == ISTATUS_OC) {
                    fprintf(event_log, "[%s] NODE %d: OVERCURRENT DETECTED       -  %.2f A (cycle %u)\n",
                            tbuf, n+1, irms[n], cycle_id[n]);
                    fflush(event_log);
                }
                else if (prev_istat[n] == ISTATUS_OC) {
                    fprintf(event_log, "[%s] NODE %d: Current returned to NORMAL -  %.2f A (cycle %u)\n",
                            tbuf, n+1, irms[n], cycle_id[n]);
                    fflush(event_log);
                }
                prev_istat[n] = istat[n];
            }
        }

        if (now - last_csv_time >= 10) {
            last_csv_time = now;

            fprintf(csv,
                "%s,"
                "%u,%u,%u,"
                "%.3f,%.3f,%.3f,"
                "%.3f,%.3f,%.3f,"
                "%.3f,%.3f,%.3f,"
                "%.3f,%.3f,%.3f,"
                "%d,%d,%d,"
                "%d,%d,%d,"
                "%.3f,%.3f,%.3f\n",
                tbuf,
                cycle_id[0], cycle_id[1], cycle_id[2],
                vrms[0], vrms[1], vrms[2],
                vpeak[0], vpeak[1], vpeak[2],
                irms[0], irms[1], irms[2],
                ipeak[0], ipeak[1], ipeak[2],
                vstat[0], vstat[1], vstat[2],
                istat[0], istat[1], istat[2],
                power[0], power[1], power[2]
            );
            fflush(csv);
        }

        usleep(50000);
    }

    return NULL;
}
```



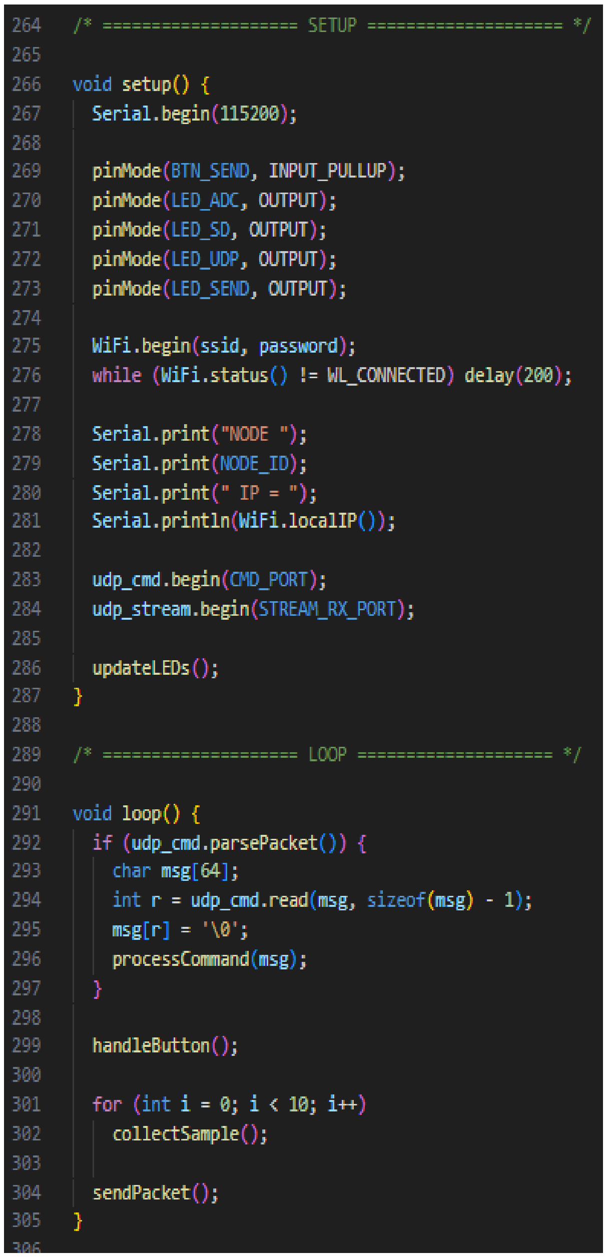
```cpp
#include <WiFi.h>
#include <WiFiUdp.h>
#include <math.h>

/* ==================== CONFIGURATION ==================== *
const char* ssid     = "wifry2";
const char* password = "nmhvkh123";

#define SERVER_IP "192.168.50.1"
#define CMD_PORT        6000
#define DATA_TX_PORT    5005
#define STREAM_RX_PORT  6001

#define NODE_ID 1

#define V_ADC_PIN  1
#define I_ADC_PIN  2

#define LED_ADC    0
#define LED_SD     38
#define LED_UDP    39
#define LED_SEND   10

#define BTN_SEND  3

#define ADC_RESOLUTION 4095.0f
#define ADC_MID        (ADC_RESOLUTION / 2.0f)

#define V_SCALE  (170.0f / (ADC_MID * 0.6f))
#define I_SCALE  0.0244f

#define RMS_BUFFER_SIZE 60
#define SEND_INTERVAL_MS 100

/* ===== LOCAL PROTECTION ===== */
#define OC_LIMIT   15.0f
#define OC_CLEAR   12.0f
#define OC_PERSIST 3

/* ==================== STATE ==================== */

WiFiUDP udp_cmd, udp_tx, udp_stream;

float vbuf[RMS_BUFFER_SIZE];
float ibuf[RMS_BUFFER_SIZE];
int   sample_idx = 0;

float vrms = 0.0f;
float irms = 0.0f;

uint32_t cycle_id = 0;
unsigned long last_send = 0;

enum Mode { MODE_ADC, MODE_SD, MODE_UDP };
Mode currentMode = MODE_UDP;

bool send_enabled = true;
bool fault_latched = false;
bool last_btn = HIGH;

uint8_t oc_counter = 0;

/* ==================== PACKET ==================== */
typedef struct {
  uint32_t node_id;
  uint32_t cycle_id;
  float vrms;
  float irms;
} __attribute__((packed)) Packet;

/* ==================== HELPERS ==================== */

void updateLEDs() {
  digitalWrite(LED_ADC,  currentMode == MODE_ADC);
  digitalWrite(LED_SD,   currentMode == MODE_SD);
  digitalWrite(LED_UDP,  currentMode == MODE_UDP);
  digitalWrite(LED_SEND, send_enabled && !fault_latched);
}
```



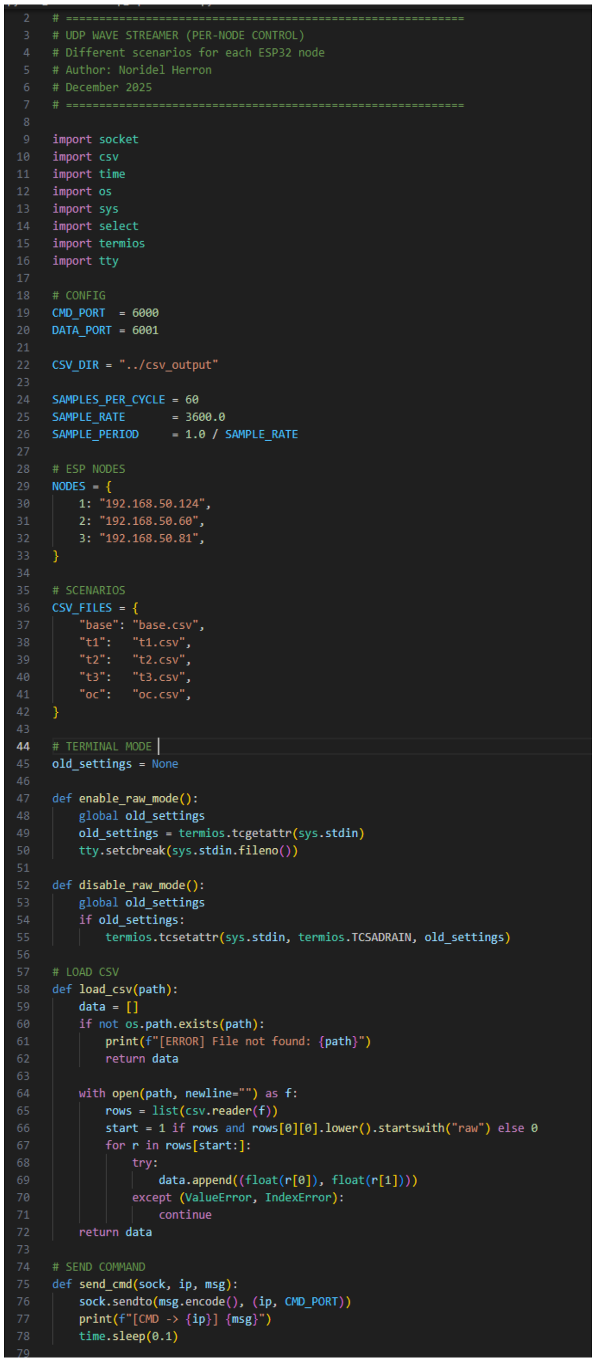
```cpp
void sendFault() {
  char msg[96];
  unsigned long ts = millis();

  snprintf(msg, sizeof(msg),
           "FAULT|%d|OC_TRIP|%.2f|%.2f|%lu",
           NODE_ID, vrms, irms, ts);
  udp_tx.beginPacket(SERVER_IP, CMD_PORT);
  udp_tx.write((uint8_t*)msg, strlen(msg));
  udp_tx.endPacket();

  Serial.println(msg);
}

void checkFaults() {
  if (fault_latched) return;

  if (irms > OC_LIMIT) {
    oc_counter++;
    if (oc_counter >= OC_PERSIST) {
      fault_latched = true;
      send_enabled  = false;
      sendFault();
      updateLEDs();
    }
  } else if (irms < OC_CLEAR) {
    oc_counter = 0;
  }
}

/* ==================== INPUT ==================== */

bool getNextSample(float *v_adc, float *i_adc) {
  if (currentMode == MODE_ADC) {
    *v_adc = analogRead(V_ADC_PIN);
    *i_adc = analogRead(I_ADC_PIN);
    return true;
  }

  if (currentMode == MODE_UDP) {
    int sz = udp_stream.parsePacket();
    if (!sz) return false;

    char buf[64];
    int r = udp_stream.read(buf, sizeof(buf) - 1);
    buf[r] = '\0';

    return sscanf(buf, "WAVE|%f|%f", v_adc, i_adc) == 2;
  }

  return false;
}

/* ==================== RMS ==================== */

void computeRMS() {
  float sv = 0, si = 0;
  for (int i = 0; i < RMS_BUFFER_SIZE; i++) {
    sv += vbuf[i] * vbuf[i];
    si += ibuf[i] * ibuf[i];
  }

  vrms = sqrt(sv / RMS_BUFFER_SIZE);
  irms = sqrt(si / RMS_BUFFER_SIZE);

  cycle_id++;
  checkFaults();
}

void collectSample() {
  float v_adc, i_adc;
  if (!getNextSample(&v_adc, &i_adc)) return;

  vbuf[sample_idx] = (v_adc - ADC_MID) * V_SCALE;
  ibuf[sample_idx] = (i_adc - ADC_MID) * I_SCALE;

  if (++sample_idx >= RMS_BUFFER_SIZE) {
    computeRMS();
    sample_idx = 0;
  }
}
```



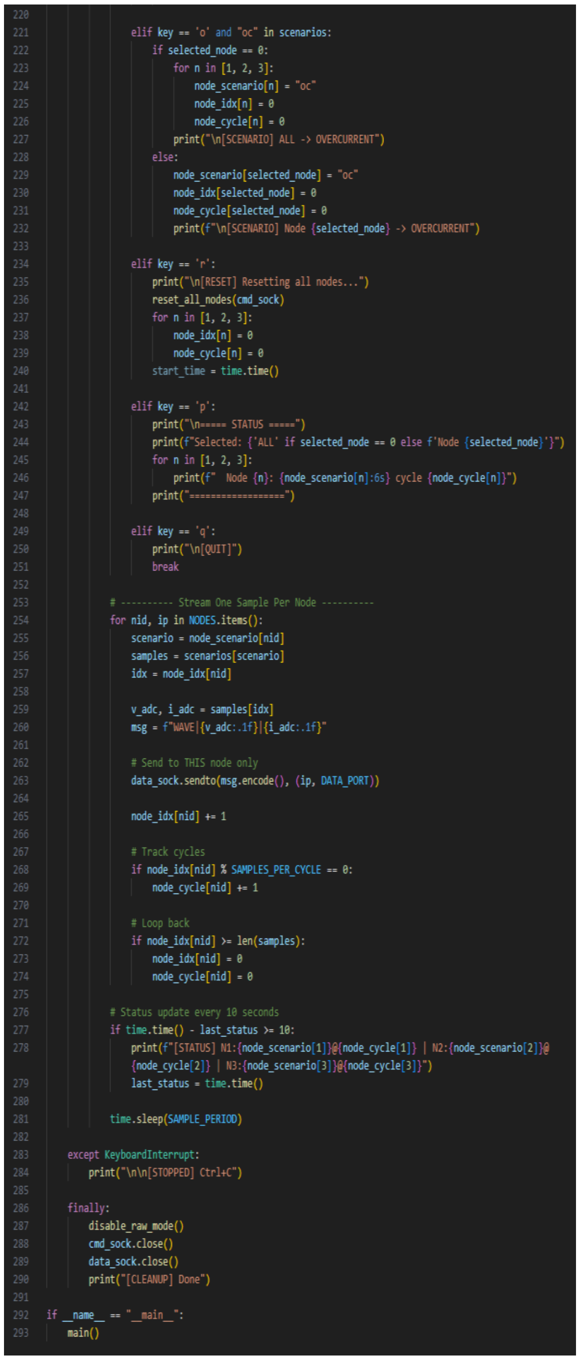
```cpp
void sendPacket() {
  if (!send_enabled || fault_latched) return;
  if (millis() - last_send < SEND_INTERVAL_MS) return;

  last_send = millis();
  Packet pkt = { NODE_ID, cycle_id, vrms, irms };

  udp_tx.beginPacket(SERVER_IP, DATA_TX_PORT);
  udp_tx.write((uint8_t*)&pkt, sizeof(pkt));
  udp_tx.endPacket();
}

/* ==================== BUTTON ==================== */

void handleButton() {
  bool btn = digitalRead(BTN_SEND);

  if (last_btn == HIGH && btn == LOW) {
    if (fault_latched && irms < OC_CLEAR) {
      fault_latched = false;
      oc_counter = 0;
      send_enabled = true;
      Serial.println("[BTN] Fault cleared");
    } else if (!fault_latched) {
      send_enabled = !send_enabled;
    }
    updateLEDs();
  }
  last_btn = btn;
}

/* ==================== COMMAND ==================== */
void processCommand(char *msg) {
  char *cmd = strtok(msg, "|");
  char *arg = strtok(NULL, "|");
  char *tgt = strtok(NULL, "|");

  if (tgt && atoi(tgt) != NODE_ID && atoi(tgt) != -1) return;

  /* ---------- ACK ---------- */
  if (!strcmp(cmd, "ACK") && fault_latched && irms < OC_CLEAR) {
    fault_latched = false;
    oc_counter = 0;
    send_enabled = true;
    Serial.println("[PI] Fault cleared");
    updateLEDs();
    return;
  }

  /* ---------- RESET_CYCLE ---------- */
  if (!strcmp(cmd, "RESET_CYCLE")) {
    cycle_id = 0;
    sample_idx = 0;
    vrms = irms = 0.0f;
    for (int i = 0; i < RMS_BUFFER_SIZE; i++) {
      vbuf[i] = ibuf[i] = 0.0f;
    }
    Serial.println("[PI] Cycle reset to 0");
    return;
  }

  /* ---------- SET_SEND ---------- */
  if (!strcmp(cmd, "SET_SEND") && !fault_latched) {
    if (!strcmp(arg, "ON")) {
      send_enabled = true;
      Serial.println("[PI] SEND ON");
    } else if (!strcmp(arg, "OFF")) {
      send_enabled = false;
      Serial.println("[PI] SEND OFF");
    }
    updateLEDs();
    return;
  }

  /* ---------- SET_MODE ---------- */
  if (!strcmp(cmd, "SET_MODE")) {
    if (!strcmp(arg, "MODE_ADC")) {
      currentMode = MODE_ADC;
      Serial.println("[PI] MODE ADC");
    } else if (!strcmp(arg, "MODE_SD")) {
      currentMode = MODE_SD;
      Serial.println("[PI] MODE SD");
    } else if (!strcmp(arg, "MODE_UDP")) {
      currentMode = MODE_UDP;
      Serial.println("[PI] MODE UDP");
    }
    updateLEDs();
    return;
  }
}
```



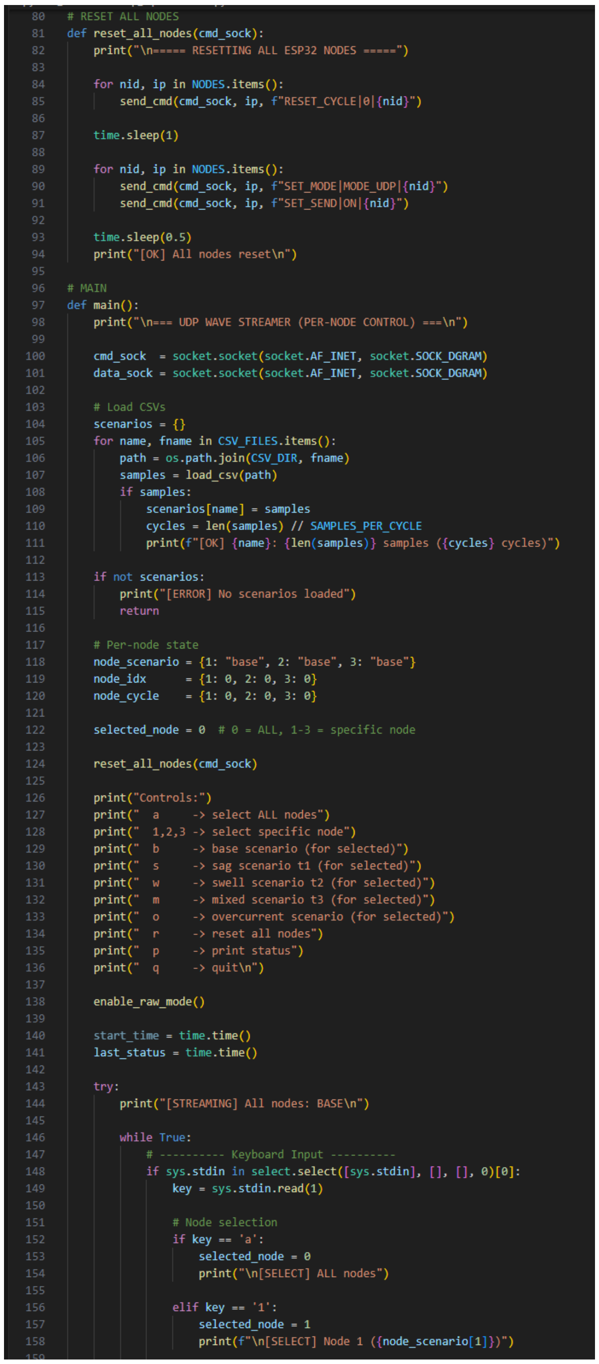
```cpp
/* ==================== SETUP ==================== */

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

  pinMode(BTN_SEND, INPUT_PULLUP);
  pinMode(LED_ADC, OUTPUT);
  pinMode(LED_SD, OUTPUT);
  pinMode(LED_UDP, OUTPUT);
  pinMode(LED_SEND, OUTPUT);

  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) delay(200);

  Serial.print("NODE ");
  Serial.print(NODE_ID);
  Serial.print(" IP = ");
  Serial.println(WiFi.localIP());

  udp_cmd.begin(CMD_PORT);
  udp_stream.begin(STREAM_RX_PORT);

  updateLEDs();
}

/* ==================== LOOP ==================== */

void loop() {
  if (udp_cmd.parsePacket()) {
    char msg[64];
    int r = udp_cmd.read(msg, sizeof(msg) - 1);
    msg[r] = '\0';
    processCommand(msg);
  }

  handleButton();

  for (int i = 0; i < 10; i++)
    collectSample();

  sendPacket();
}
```



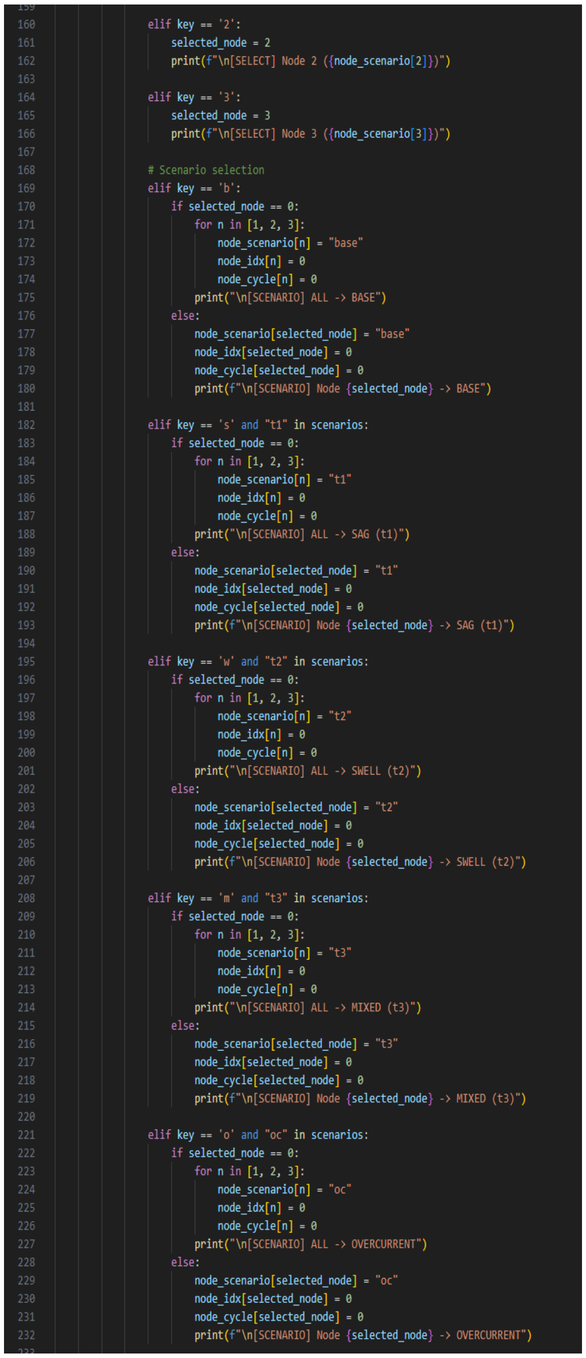
```python
# ============================================================
# UDP WAVE STREAMER (PER-NODE CONTROL)
# Different scenarios for each ESP32 node
# Author: Noridel Herron
# December 2025
# ============================================================

import socket
import csv
import time
import os
import sys
import select
import termios
import tty

# CONFIG
CMD_PORT  = 6000
DATA_PORT = 6001

CSV_DIR = "../csv_output"

SAMPLES_PER_CYCLE = 60
SAMPLE_RATE       = 3600.0
SAMPLE_PERIOD     = 1.0 / SAMPLE_RATE

# ESP NODES
NODES = {
    1: "192.168.50.124",
    2: "192.168.50.60",
    3: "192.168.50.81",
}

# SCENARIOS
CSV_FILES = {
    "base": "base.csv",
    "t1":   "t1.csv",
    "t2":   "t2.csv",
    "t3":   "t3.csv",
    "oc":   "oc.csv",
}

# TERMINAL MODE
old_settings = None

def enable_raw_mode():
    global old_settings
    old_settings = termios.tcgetattr(sys.stdin)
    tty.setcbreak(sys.stdin.fileno())

def disable_raw_mode():
    global old_settings
    if old_settings:
        termios.tcsetattr(sys.stdin, termios.TCSADRAIN, old_settings)

# LOAD CSV
def load_csv(path):
    data = []
    if not os.path.exists(path):
        print(f"[ERROR] File not found: {path}")
        return data

    with open(path, newline="") as f:
        rows = list(csv.reader(f))
        start = 1 if rows and rows[0][0].lower().startswith("raw") else 0
        for r in rows[start:]:
            try:
                data.append((float(r[0]), float(r[1])))
            except (ValueError, IndexError):
                continue
    return data

# SEND COMMAND
def send_cmd(sock, ip, msg):
    sock.sendto(msg.encode(), (ip, CMD_PORT))
    print(f"[CMD -> {ip}] {msg}")
    time.sleep(0.1)
```



```python

                elif key == 'o' and "oc" in scenarios:
                    if selected_node == 0:
                        for n in [1, 2, 3]:
                            node_scenario[n] = "oc"
                            node_idx[n] = 0
                            node_cycle[n] = 0
                        print("\n[SCENARIO] ALL -> OVERCURRENT")
                    else:
                        node_scenario[selected_node] = "oc"
                        node_idx[selected_node] = 0
                        node_cycle[selected_node] = 0
                        print(f"\n[SCENARIO] Node {selected_node} -> OVERCURRENT")

                elif key == 'r':
                    print("\n[RESET] Resetting all nodes...")
                    reset_all_nodes(cmd_sock)
                    for n in [1, 2, 3]:
                        node_idx[n] = 0
                        node_cycle[n] = 0
                    start_time = time.time()

                elif key == 'p':
                    print("\n===== STATUS =====")
                    print(f"Selected: {'ALL' if selected_node == 0 else f'Node {selected_node}'}")
                    for n in [1, 2, 3]:
                        print(f"  Node {n}: {node_scenario[n]:6s} cycle {node_cycle[n]}")
                    print("=================")

                elif key == 'q':
                    print("\n[QUIT]")
                    break

            # ---------- Stream One Sample Per Node ----------
            for nid, ip in NODES.items():
                scenario = node_scenario[nid]
                samples = scenarios[scenario]
                idx = node_idx[nid]

                v_adc, i_adc = samples[idx]
                msg = f"WAVE|{v_adc:.1f}|{i_adc:.1f}"

                # Send to THIS node only
                data_sock.sendto(msg.encode(), (ip, DATA_PORT))

                node_idx[nid] += 1

                # Track cycles
                if node_idx[nid] % SAMPLES_PER_CYCLE == 0:
                    node_cycle[nid] += 1

                # Loop back
                if node_idx[nid] >= len(samples):
                    node_idx[nid] = 0
                    node_cycle[nid] = 0

            # Status update every 10 seconds
            if time.time() - last_status >= 10:
                print(f"[STATUS] N1:{node_scenario[1]}@{node_cycle[1]} | N2:{node_scenario[2]}@
                {node_cycle[2]} | N3:{node_scenario[3]}@{node_cycle[3]}")
                last_status = time.time()

            time.sleep(SAMPLE_PERIOD)

    except KeyboardInterrupt:
        print("\n\n[STOPPED] Ctrl+C")

    finally:
        disable_raw_mode()
        cmd_sock.close()
        data_sock.close()
        print("[CLEANUP] Done")

if __name__ == "__main__":
    main()
```



```python
# RESET ALL NODES
def reset_all_nodes(cmd_sock):
    print("\n===== RESETTING ALL ESP32 NODES =====")

    for nid, ip in NODES.items():
        send_cmd(cmd_sock, ip, f"RESET_CYCLE|0|{nid}")

    time.sleep(1)

    for nid, ip in NODES.items():
        send_cmd(cmd_sock, ip, f"SET_MODE|MODE_UDP|{nid}")
        send_cmd(cmd_sock, ip, f"SET_SEND|ON|{nid}")

    time.sleep(0.5)
    print("[OK] All nodes reset\n")

# MAIN
def main():
    print("\n=== UDP WAVE STREAMER (PER-NODE CONTROL) ===\n")

    cmd_sock  = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    data_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Load CSVs
    scenarios = {}
    for name, fname in CSV_FILES.items():
        path = os.path.join(CSV_DIR, fname)
        samples = load_csv(path)
        if samples:
            scenarios[name] = samples
            cycles = len(samples) // SAMPLES_PER_CYCLE
            print(f"[OK] {name}: {len(samples)} samples ({cycles} cycles)")

    if not scenarios:
        print("[ERROR] No scenarios loaded")
        return

    # Per-node state
    node_scenario = {1: "base", 2: "base", 3: "base"}
    node_idx      = {1: 0, 2: 0, 3: 0}
    node_cycle    = {1: 0, 2: 0, 3: 0}

    selected_node = 0  # 0 = ALL, 1-3 = specific node

    reset_all_nodes(cmd_sock)

    print("Controls:")
    print("  a     -> select ALL nodes")
    print("  1,2,3 -> select specific node")
    print("  b     -> base scenario (for selected)")
    print("  s     -> sag scenario t1 (for selected)")
    print("  w     -> swell scenario t2 (for selected)")
    print("  m     -> mixed scenario t3 (for selected)")
    print("  o     -> overcurrent scenario (for selected)")
    print("  r     -> reset all nodes")
    print("  p     -> print status")
    print("  q     -> quit\n")

    enable_raw_mode()

    start_time = time.time()
    last_status = time.time()

    try:
        print("[STREAMING] All nodes: BASE\n")

        while True:
            # ---------- Keyboard Input ----------
            if sys.stdin in select.select([sys.stdin], [], [], 0)[0]:
                key = sys.stdin.read(1)

                # Node selection
                if key == 'a':
                    selected_node = 0
                    print("\n[SELECT] ALL nodes")

                elif key == '1':
                    selected_node = 1
                    print(f"\n[SELECT] Node 1 ({node_scenario[1]})")
```



```python
                elif key == '2':
                    selected_node = 2
                    print(f"\n[SELECT] Node 2 ({node_scenario[2]})")

                elif key == '3':
                    selected_node = 3
                    print(f"\n[SELECT] Node 3 ({node_scenario[3]})")

                # Scenario selection
                elif key == 'b':
                    if selected_node == 0:
                        for n in [1, 2, 3]:
                            node_scenario[n] = "base"
                            node_idx[n] = 0
                            node_cycle[n] = 0
                        print("\n[SCENARIO] ALL -> BASE")
                    else:
                        node_scenario[selected_node] = "base"
                        node_idx[selected_node] = 0
                        node_cycle[selected_node] = 0
                        print(f"\n[SCENARIO] Node {selected_node} -> BASE")

                elif key == 's' and "t1" in scenarios:
                    if selected_node == 0:
                        for n in [1, 2, 3]:
                            node_scenario[n] = "t1"
                            node_idx[n] = 0
                            node_cycle[n] = 0
                        print("\n[SCENARIO] ALL -> SAG (t1)")
                    else:
                        node_scenario[selected_node] = "t1"
                        node_idx[selected_node] = 0
                        node_cycle[selected_node] = 0
                        print(f"\n[SCENARIO] Node {selected_node} -> SAG (t1)")

                elif key == 'w' and "t2" in scenarios:
                    if selected_node == 0:
                        for n in [1, 2, 3]:
                            node_scenario[n] = "t2"
                            node_idx[n] = 0
                            node_cycle[n] = 0
                        print("\n[SCENARIO] ALL -> SWELL (t2)")
                    else:
                        node_scenario[selected_node] = "t2"
                        node_idx[selected_node] = 0
                        node_cycle[selected_node] = 0
                        print(f"\n[SCENARIO] Node {selected_node} -> SWELL (t2)")

                elif key == 'm' and "t3" in scenarios:
                    if selected_node == 0:
                        for n in [1, 2, 3]:
                            node_scenario[n] = "t3"
                            node_idx[n] = 0
                            node_cycle[n] = 0
                        print("\n[SCENARIO] ALL -> MIXED (t3)")
                    else:
                        node_scenario[selected_node] = "t3"
                        node_idx[selected_node] = 0
                        node_cycle[selected_node] = 0
                        print(f"\n[SCENARIO] Node {selected_node} -> MIXED (t3)")

                elif key == 'o' and "oc" in scenarios:
                    if selected_node == 0:
                        for n in [1, 2, 3]:
                            node_scenario[n] = "oc"
                            node_idx[n] = 0
                            node_cycle[n] = 0
                        print("\n[SCENARIO] ALL -> OVERCURRENT")
                    else:
                        node_scenario[selected_node] = "oc"
                        node_idx[selected_node] = 0
                        node_cycle[selected_node] = 0
                        print(f"\n[SCENARIO] Node {selected_node} -> OVERCURRENT")
```