Real-Time Energy Monitor — Modbus RTU + MQTT

STM32F401RE + FreeRTOS + PZEM-004T + ESP8266

Portfolio Project — Embedded Systems / IoT Architecture

Palmer Hutt | huttlabs.com | github.com/phutt7

1. Goal

Build a real-time energy monitoring node that reads AC power measurements from a PZEM-004T energy meter over Modbus RTU, processes the data under FreeRTOS with fault detection, and publishes live readings to an MQTT broker via ESP8266 WiFi. The project targets embedded systems engineering roles at solar, storage, and energy management companies by demonstrating the core protocols and architectural patterns used in real industrial monitoring systems.

2. Project Objective

Design and implement a firmware framework that:

- Reads six AC power measurements from a PZEM-004T over Modbus RTU
- Implements Modbus RTU framing and CRC16 validation from scratch — no library
- Runs three prioritized FreeRTOS tasks with a mutex-protected shared data struct
- Detects over-power, under-voltage, low power factor, and communication timeout faults
- Publishes live data to an MQTT broker via ESP8266 WiFi module
- Logs CSV telemetry to UART for graphing and documentation
- Operates continuously under IWDG watchdog supervision

3. Why This Project

Modbus RTU is the dominant communication protocol for solar inverters, battery management systems, and energy meters globally. MQTT is the standard IoT telemetry protocol used by Stem Inc, Omnidian, Span.io, and virtually every cloud-connected energy platform. Implementing both from scratch on a real sensor producing real AC measurements demonstrates the exact skills required for implementation, applications, and solutions engineering roles at cleantech companies.

The PZEM-004T is used in commercial solar monitoring installations. The six measurements it provides — voltage, current, active power, energy (kWh), frequency, and power factor — are the same parameters logged by Enphase Envoy units, SolarEdge monitoring systems, and Span smart panels. When a hiring manager at one of those companies sees this project they recognise the stack immediately.

4. Target Hardware

Component	Part	Connection	Notes
MCU	STM32F401 RE Nucleo	—	ARM Cortex-M4 @ 84 MHz
Energy meter	PZEM-004T V3.0 100A	USART1 @ 9600 baud	Via logic level converter

Component	Part	Connection	Notes
WiFi module	Wemos D1 Mini ESP8266	USART3 @ 115200 baud	MQTT bridge
Logic level converter	4-channel bidirectional	Between PZEM and STM32	5V↔3.3V
Test load	Extension cord + lamp/charger	CT clamp around live wire	500W max to start
Fault indicator	LED + 330Ω resistor	GPIO output	Over-power / fault state

Table 1: Bill of Materials

Safety Note — PZEM-004T Mains Connection

The PZEM-004T has built-in galvanic isolation between the 120V AC measurement side and the TTL communication side. The CT clamp goes around the live wire without cutting it. Treat the AC screw terminals with standard mains safety practice. Use a lamp or phone charger as the test load — nothing above 500W during initial bring-up.

5. Software Architecture

5.1 Layered Structure

- HAL Layer: USART1 (Modbus), USART2 (telemetry), USART3 (ESP8266), GPIO, IWDG
- Driver Layer: Modbus RTU framing + CRC16, PZEM-004T register parser, ESP8266 AT command driver
- Application Layer: Task A (poller), Task B (fault monitor), Task C (telemetry)

5.2 Data Flow

1. TIM2 1s interrupt wakes Task A
2. Task A sends Modbus request frame over USART1
3. PZEM-004T responds with 25-byte register block
4. Task A validates CRC16, parses all six measurements
5. Task A acquires mutex, writes to shared PzemData_t struct, releases mutex
6. Task B acquires mutex, reads struct, checks thresholds, drives fault LED, kicks IWDG
7. Task C acquires mutex, reads struct, formats CSV to USART2 and JSON to USART3
8. ESP8266 receives JSON, publishes to MQTT broker over WiFi

6. System Architecture

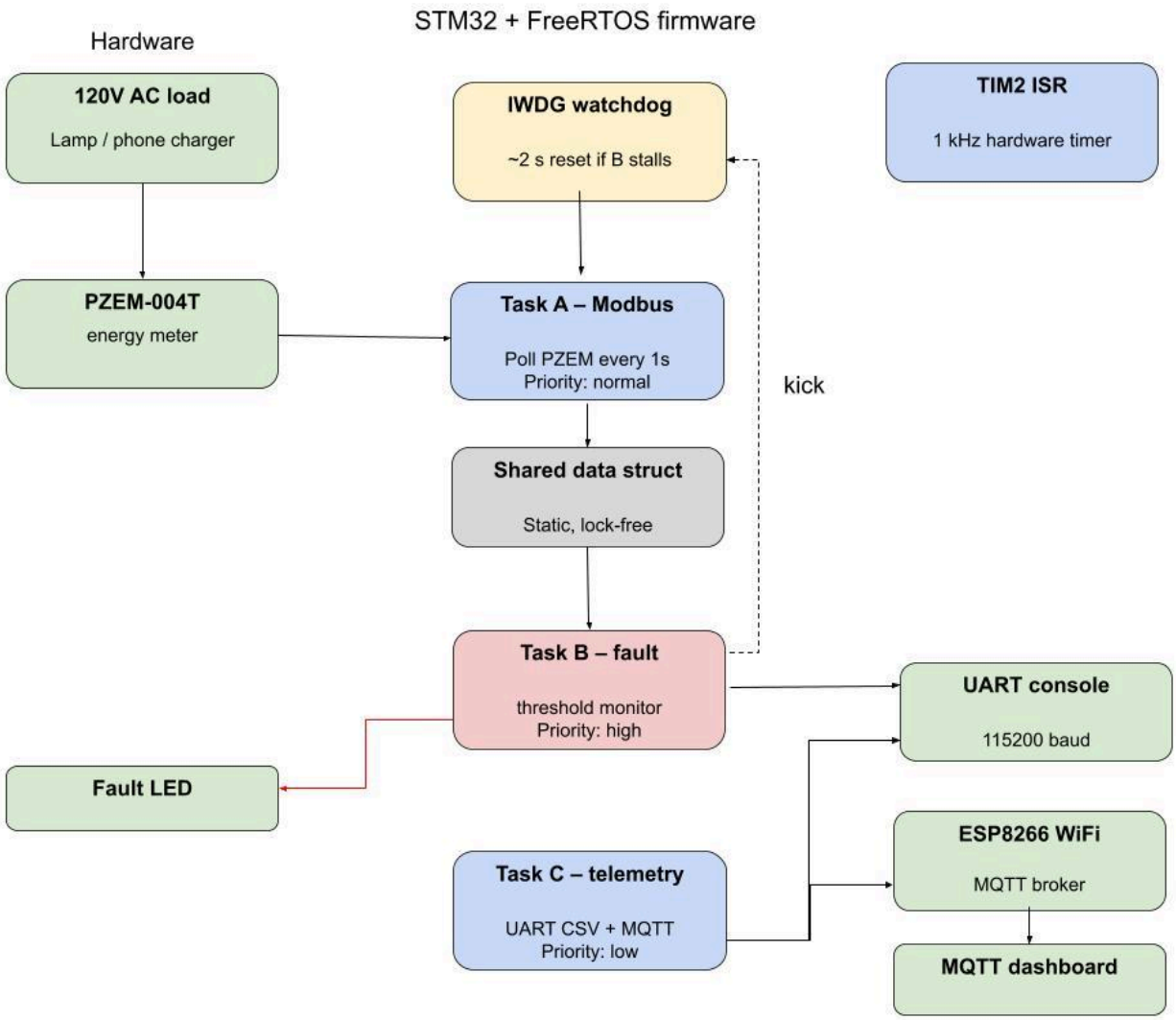


Fig. 1: System Block Diagram — Hardware, STM32 firmware, MQTT cloud

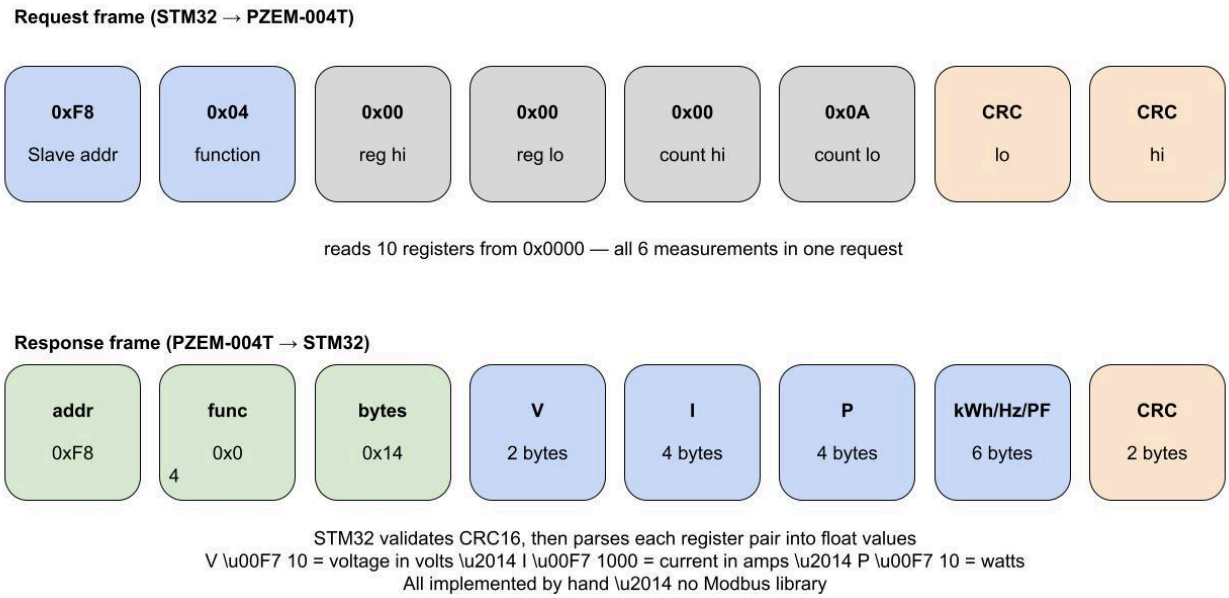


Fig. 2: Modbus RTU Request and Response Frame Structure

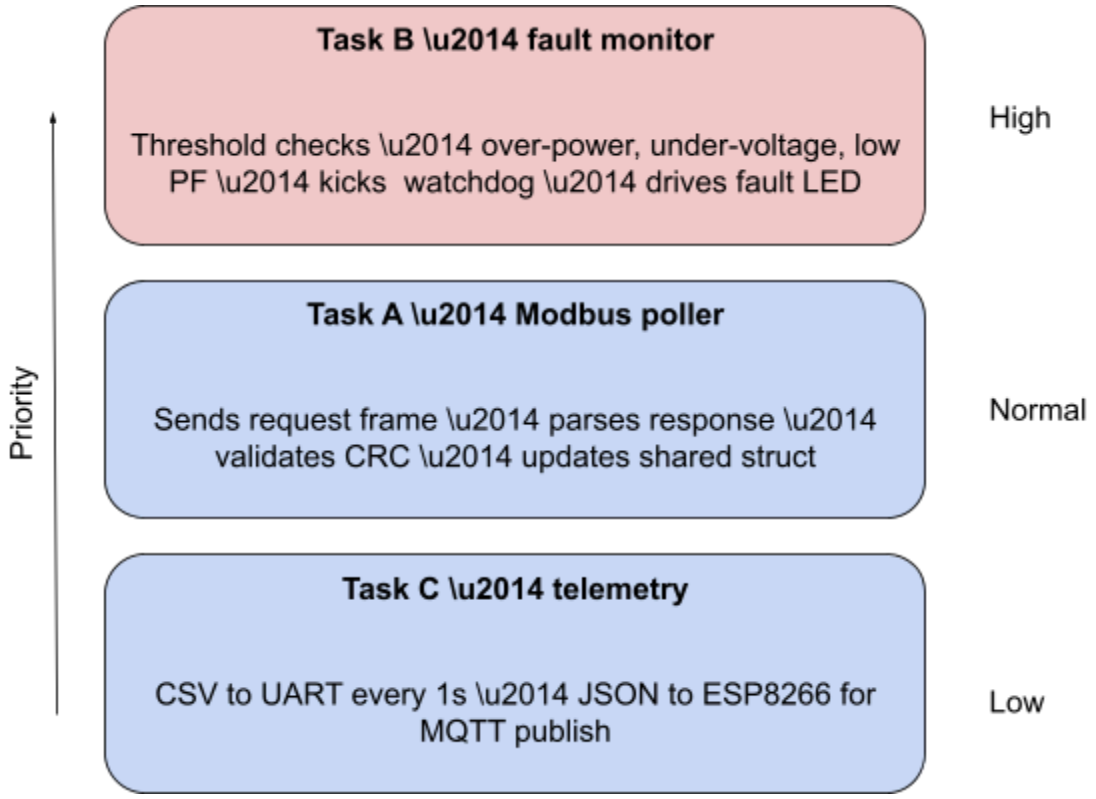


Fig. 3: FreeRTOS Task Priority Levels

7. Key Features

7.1 Modbus RTU — Implemented from Scratch

The STM32 constructs the full 8-byte Modbus RTU request frame, including CRC16 computed with the standard Modbus polynomial (0xA001). It sends a single read request (function code 0x04) that returns all 10 registers in one transaction, covering all six measurements. The response parser validates the CRC, then divides each register by its documented scale factor to produce float values.

Why no library:

Writing your own Modbus framing means you can explain every byte in a technical interview. In production work integrating with a SolarEdge inverter or a BMS with a custom register map, you will need to read a datasheet and implement the exact frame format specified. This project demonstrates that capability.

7.2 FreeRTOS Three-Task Pipeline

Task A — Modbus Poller (Normal Priority)

- Polls PZEM-004T every 1 second via USART1
- Validates CRC16 on every response
- Tracks consecutive communication errors — 3 failures sets FAULT_COMMS
- Writes parsed data to shared struct under mutex protection

Task B — Fault Monitor (High Priority)

- Reads shared struct every 100ms
- Checks: over-power > 1500W, under-voltage < 108V, low PF < 0.85, comms timeout
- Drives fault LED on any active fault
- Kicks IWDG watchdog every cycle — system resets if Task B stalls

Task C — Telemetry (Low Priority)

- Reads shared struct every 1 second
- Formats CSV string to USART2 for PC terminal and Excel graphing
- Formats JSON string to USART3 for ESP8266 MQTT publish

7.3 Mutex-Protected Shared Data

Unlike Project 1 which used a FreeRTOS queue for inter-task data, this project uses a mutex-protected shared struct. The PZEM data is not a stream — it is a current snapshot that all three tasks need to read. The mutex ensures Tasks B and C never read a partially-written struct mid-update from Task A. This is the standard pattern for shared sensor data in multi-task embedded firmware.

7.4 MQTT Telemetry via ESP8266

Task C sends JSON-formatted strings to the ESP8266 over USART3. The ESP8266 runs AT firmware and connects to a home WiFi network. On receiving a complete JSON line it publishes to the configured MQTT broker topic. A free HiveMQ Cloud instance or a local Mosquitto broker handles the subscription side. From any MQTT client you can subscribe to energy/node1/# and watch all six measurements update in real time.

8. Measurement Parameters

Parameter	Range	Resolution	PZEM Register Scale
Voltage	80–260V	0.1V	reg[0] ÷ 10
Current	0–10A	0.001A	(reg[2]<<16 | reg[1]) ÷ 1000
Active Power	0–2300W	0.1W	(reg[4]<<16 | reg[3]) ÷ 10
Energy	0–9999 kWh	1 Wh	(reg[6]<<16 | reg[5]) ÷ 1000
Frequency	45–65Hz	0.1Hz	reg[7] ÷ 10
Power Factor	0.00–1.00	0.01	reg[8] ÷ 100

Table 2: PZEM-004T Register Map and Scale Factors

9. Fault Detection Thresholds

Fault	Threshold	Meaning
FAULT_OVER_POWER	Power > 1500W	Load exceeding circuit capacity
FAULT_UNDER_VOLT	Voltage < 108V	Brownout condition (US nominal 120V)
FAULT_LOW_PF	Power factor < 0.85	Highly inductive load, poor efficiency
FAULT_COMMS	3 consecutive failures	Meter disconnected or wiring fault

Table 3: Fault Detection Thresholds (all configurable via #define)

10. CubeMX Configuration

Peripheral	Setting	Value	Purpose
USART1	Baud	9600	Modbus RTU to PZEM-004T
USART2	Baud	115200	CSV telemetry to PC

Peripheral	Setting	Value	Purpose
USART3	Baud	115200	JSON to ESP8266
TIM2	Period	1000ms	Task A poll trigger
IWDG	Prescaler	16	LSI 32kHz / 16 = 2kHz
IWDG	Reload	4095	~2s watchdog timeout
SYS	Timebase	TIM1	FreeRTOS owns SysTick
FreeRTOS	API	CMSIS-RTOS V2	
FreeRTOS	Heap	heap_4	Best-fit with coalescence

Table 4: CubeMX Peripheral Configuration

11. Telemetry Output Format

11.1 UART CSV (115200 baud, USART2)

Header row:

```
timestamp_ms,voltage_V,current_A,power_W,energy_kWh,frequency_Hz,pf,fault
```

Sample output:

```
1000,120.1,0.542,65.1,0.003,60.0,0.99,0
2000,120.0,0.541,65.0,0.003,60.0,0.99,0
3000,118.2,0.544,64.3,0.004,60.0,0.98,1
```

Third line: voltage dropped below 108V → fault bit set

11.2 MQTT JSON (USART3 to ESP8266)

```
{"v":120.1,"i":0.542,"p":65.1,"e":0.003,"hz":60.0,"pf":0.99,"fault":0}
```

Published to topic: energy/node1/data

12. Project Rules

- Modbus framing implemented manually — no external library
- No dynamic memory allocation
- IWDG tied to Task B (fault monitor) only
- Mutex protecting shared data struct between all three tasks
- All thresholds configurable via #define at top of file
- All modules separated: HAL → Driver → Application

13. Success Criteria

- PZEM-004T responds to Modbus requests with valid CRC on every poll
- Voltage reading within 1% of known value (verified with multimeter)
- Power reading within 2% of known load (phone charger ~5W, lamp ~60W)
- FAULT_OVER_POWER triggers at correct threshold
- FAULT_UNDER_VOLT triggers at correct threshold
- MQTT broker receives live updates within 2 seconds of measurement
- CSV exported to Excel — voltage and power graphed over 10 minutes
- System runs 1 hour without watchdog reset

14. Design Rationale

14.1 Why PZEM-004T?

The PZEM-004T is a genuine industrial-grade energy meter used in commercial solar installations, not a hobby sensor. It measures six real AC parameters with documented accuracy specifications. It communicates via Modbus RTU — the same protocol used by SolarEdge inverters, Fronius inverters, and most commercial BMS units. It costs under $20 and is available on Amazon Prime. It is the most realistic and most accessible hardware choice for demonstrating industrial energy monitoring firmware.

14.2 Why Modbus from Scratch?

Using a Modbus library would reduce this to a configuration exercise. Implementing the framing, CRC calculation, and register parsing from scratch demonstrates that you understand the protocol, not just how to call a function. In a real application engineering role you will regularly encounter devices with custom register maps and non-standard Modbus implementations. The skill being demonstrated is the ability to read a datasheet and implement what it specifies.

14.3 Why MQTT?

MQTT is the dominant IoT telemetry protocol for cloud-connected energy systems. Stem Inc, Omnidian, Span.io, and virtually every other cloud energy platform uses MQTT at the edge node layer. Including it demonstrates awareness of the full stack — not just the microcontroller, but how it connects to the monitoring infrastructure above it.

14.4 Why Mutex Instead of Queue?

The PZEM data is a current snapshot, not a stream of events. All three tasks need access to the latest reading at any time, not a historical sequence. A mutex-protected shared struct is the correct pattern here — it avoids queue depth management complexity and accurately models how real sensor data is shared in multi-task firmware.

15. Risks & Mitigations

- PZEM no response → check baud rate, verify logic level converter wiring, check USART1 init
- CRC mismatch → verify bit order in CRC16 (Modbus uses little-endian CRC byte order)
- ESP8266 AT commands failing → check USART3 baud, verify AT firmware version with AT+GMR
- Readings inaccurate → verify load is > 10W (PZEM has minimum load threshold), check CT clamp orientation
- Task B fault false positives → add 3-sample debounce before setting fault flag

16. Business-Level Explanation

This system does what every solar and energy storage monitoring platform does at the edge: read a sensor, check if something is wrong, send the data to the cloud. The difference is that it is built from scratch using the same industrial protocols — Modbus and MQTT — that real inverters, meters, and battery systems use in the field.

From an applications engineering perspective: when a customer's Enphase system shows an anomaly, or a SolarEdge installation reports a power quality issue, someone has to understand what the monitoring node is actually measuring and why the alert triggered. This project demonstrates that understanding at the firmware level.

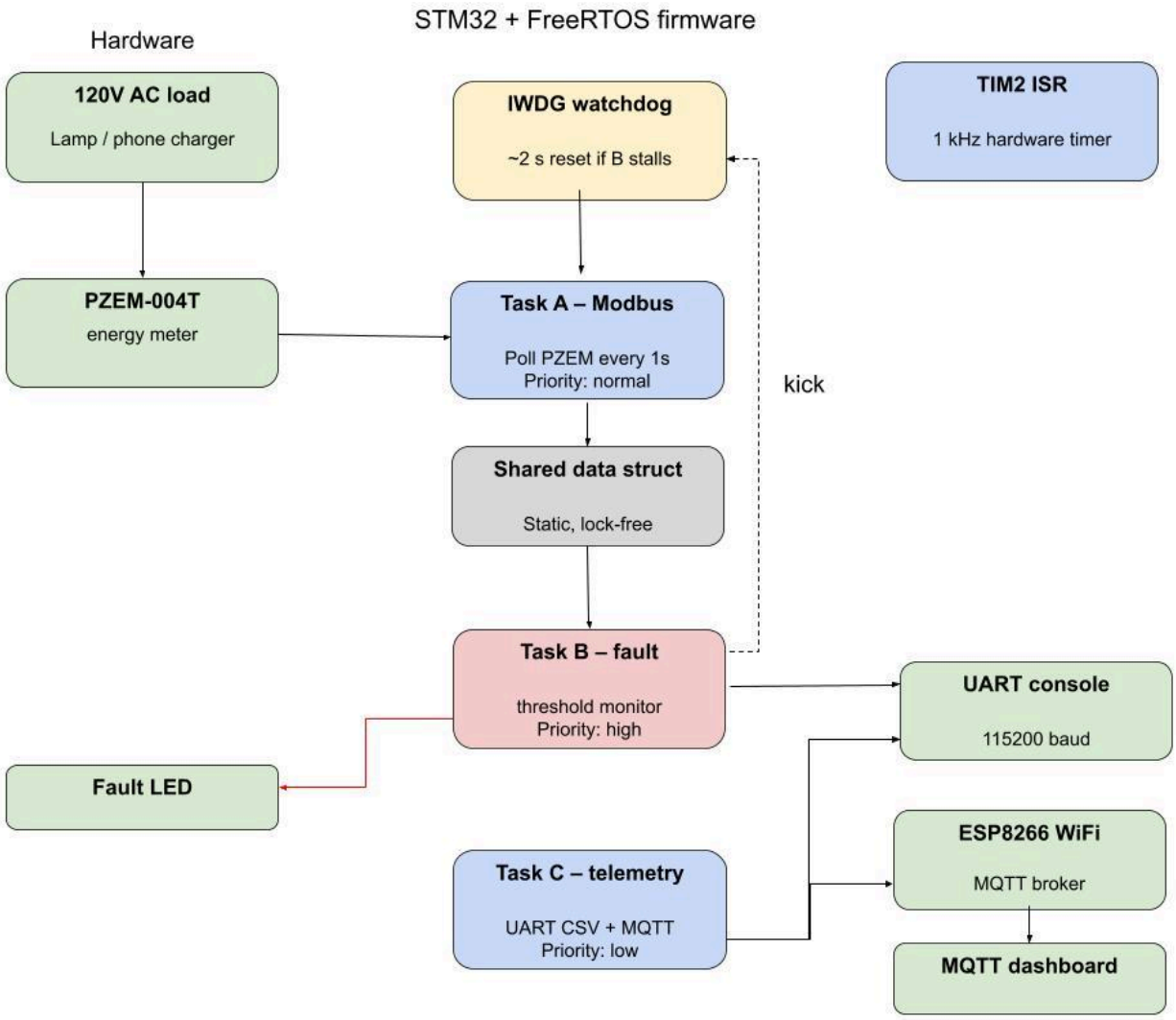
17. Upgrade Path

- Add SD card logging — FAT filesystem, daily CSV files, data persistence across reboots
- Add multiple PZEM nodes on a Modbus RS-485 bus — multi-circuit monitoring
- Replace ESP8266 with ESP32 — BLE + WiFi, local dashboard, over-the-air firmware update
- Add energy tariff calculation — kWh × rate = cost per day, week, month
- Connect to Home Assistant via MQTT autodiscovery — full home energy dashboard

Appendix A — System Diagrams

Screenshot diagrams from Claude and any additional hand-drawn block diagrams go here.

A1. Full System Block Diagram



A2. Modbus Frame Structure

Request frame (STM32 → PZEM-004T)

0xF8	0x04	0x00	0x00	0x00	0x0A	CRC	CRC
Slave addr	function	reg hi	reg lo	count hi	count lo	lo	hi

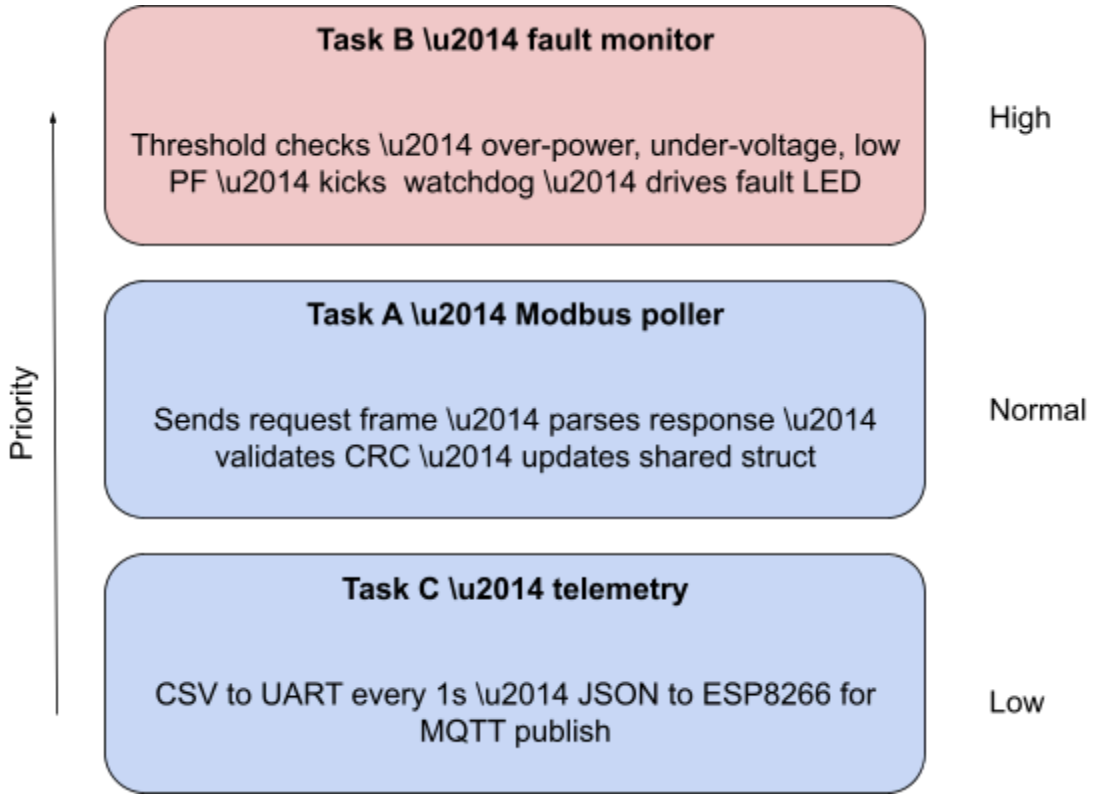
reads 10 registers from 0x0000 — all 6 measurements in one request

Response frame (PZEM-004T → STM32)

addr	func	bytes	V	I	P	kWh/Hz/PF	CRC
0xF8	0x04	0x14	2 bytes	4 bytes	4 bytes	6 bytes	2 bytes

STM32 validates CRC16, then parses each register pair into float values
V \u00F7 10 = voltage in volts \u2014 I \u00F7 1000 = current in amps \u2014 P \u00F7 10 = watts
All implemented by hand \u2014 no Modbus library

A3. Task Priority Diagram



Appendix B — Source Code

Full source available at github.com/phutt7. Core implementation files reproduced here for reference.

B1. Modbus RTU Request Frame Construction

```
/* Modbus RTU request frame — reads all 10 registers in one shot */
#define PZEM_ADDR     0xF8
#define PZEM_CMD      0x04
#define PZEM_REG_START 0x0000
#define PZEM_REG_COUNT 0x000A

uint8_t request[8];
request[0] = PZEM_ADDR;
request[1] = PZEM_CMD;
request[2] = (PZEM_REG_START >> 8) & 0xFF;
request[3] = PZEM_REG_START & 0xFF;
request[4] = (PZEM_REG_COUNT >> 8) & 0xFF;
request[5] = PZEM_REG_COUNT & 0xFF;
uint16_t crc = modbus_crc16(request, 6);
request[6] = crc & 0xFF;       // CRC low byte
request[7] = (crc >> 8) & 0xFF; // CRC high byte
HAL_UART_Transmit(&huart1, request, 8, 100);
```

B2. PZEM-004T Response Parser

```
/* Parse PZEM response — all values scaled by fixed factors */
typedef struct {
    float voltage;    // V   = reg[0] / 10.0f
    float current;    // A   = (reg[1] | reg[2]<<16) / 1000.0f
    float power;      // W   = (reg[3] | reg[4]<<16) / 10.0f
    float energy;     // kWh = (reg[5] | reg[6]<<16) / 1000.0f
    float frequency;  // Hz  = reg[7] / 10.0f
    float pf;         // PF  = reg[8] / 100.0f
    uint8_t fault;
} PzemData_t;

void pzem_parse(uint8_t *buf, PzemData_t *d) {
    if (!modbus_crc_valid(buf, 25)) { d->fault = 1; return; }
    uint16_t *r = (uint16_t*)(buf + 3);
    d->voltage   = __builtin_bswap16(r[0]) / 10.0f;
    d->current   = (__builtin_bswap16(r[2])<<16 | __builtin_bswap16(r[1])) / 1000.0f;
    d->power     = (__builtin_bswap16(r[4])<<16 | __builtin_bswap16(r[3])) / 10.0f;
    d->energy    = (__builtin_bswap16(r[6])<<16 | __builtin_bswap16(r[5])) / 1000.0f;
    d->frequency = __builtin_bswap16(r[7]) / 10.0f;
    d->pf        = __builtin_bswap16(r[8]) / 100.0f;
    d->fault = 0;
}
```

B3. Task A — Modbus Poller

```
/* Task A — Modbus Poller (Normal Priority) */
void StartModbusTask(void *argument)
{
    uint8_t rxBuf[25];
    PzemData_t reading;
    for(;;) {
        pzem_send_request(&huart1);
        if (HAL_UART_Receive(&huart1, rxBuf, 25, 500) == HAL_OK) {
            pzem_parse(rxBuf, &reading);
            osMutexAcquire(dataMutex, osWaitForever);
            memcpy(&sharedData, &reading, sizeof(PzemData_t));
            osMutexRelease(dataMutex);
            commsErrorCount = 0;
        } else {
            commsErrorCount++;
            if (commsErrorCount >= 3) sharedData.fault = FAULT_COMMS;
        }
        osDelay(1000);
    }
}
```

B4. Task B — Fault Monitor

```
/* Task B — Fault Monitor (High Priority) */
#define THRESH_OVER_POWER   1500.0f  // W
#define THRESH_UNDER_VOLT   108.0f   // V  (brownout)
#define THRESH_LOW_PF       0.85f

void StartFaultTask(void *argument)
{
    PzemData_t local;
    for(;;) {
        osMutexAcquire(dataMutex, osWaitForever);
        memcpy(&local, &sharedData, sizeof(PzemData_t));
        osMutexRelease(dataMutex);

        uint8_t fault = 0;
        if (local.power     > THRESH_OVER_POWER)  fault |= FAULT_OVER_POWER;
        if (local.voltage   < THRESH_UNDER_VOLT)  fault |= FAULT_UNDER_VOLT;
        if (local.pf        < THRESH_LOW_PF)      fault |= FAULT_LOW_PF;
        if (local.fault     == FAULT_COMMS)       fault |= FAULT_COMMS;

        HAL_GPIO_WritePin(FAULT_LED_PORT, FAULT_LED_PIN,
            fault ? GPIO_PIN_SET : GPIO_PIN_RESET);
        faultStatus = fault;

        HAL_IWDG_Refresh(&hiwdg);
        osDelay(100);
    }
}
```

B5. Task C — Telemetry

```
/* Task C — Telemetry (Low Priority) */
void StartTelemetryTask(void *argument)
{
    char csvBuf[128];
    char jsonBuf[128];
    uint32_t ts = 0;
    PzemData_t local;
    for(;;) {
        osMutexAcquire(dataMutex, osWaitForever);
        memcpy(&local, &sharedData, sizeof(PzemData_t));
        osMutexRelease(dataMutex);

        // CSV to PC terminal
        snprintf(csvBuf, sizeof(csvBuf),
            "%lu,%.1f,%.3f,%.1f,%.3f,%.1f,%.2f,%d\r\n",
            ts, local.voltage, local.current, local.power,
            local.energy, local.frequency, local.pf, faultStatus);
        HAL_UART_Transmit(&huart2, (uint8_t*)csvBuf, strlen(csvBuf), 200);

        // JSON to ESP8266 for MQTT
        snprintf(jsonBuf, sizeof(jsonBuf),

"{\"v\":%.1f,\"i\":%.3f,\"p\":%.1f,\"e\":%.3f,\"hz\":%.1f,\"pf\":%.2f,\"fault\":%d}\r\
n",
            local.voltage, local.current, local.power,
            local.energy, local.frequency, local.pf, faultStatus);
        HAL_UART_Transmit(&huart3, (uint8_t*)jsonBuf, strlen(jsonBuf), 200);

        ts += 1000;
        osDelay(1000);
    }
}
```

Complete main.c (peripheral init), CRC16 implementation, and ESP8266 AT driver available in the GitHub repository.