Project 3: Embedded Sensor Management System

Problem Statement

Design a C program for an embedded system that manages multiple sensor types in a smart device (e.g., IoT environmental monitor). Use unions, structs, and enums to efficiently handle different sensor types (Temperature, Humidity, Pressure) with type-specific configurations and data processing. The program should run on a constrained embedded system with limited memory, emphasizing efficient use of unions for sensor data storage, and include functions to configure sensors, read data, and process readings based on sensor-specific logic.

Requirements

1. Enum for Sensor Types:
   - Define sensor types: TEMPERATURE, HUMIDITY, PRESSURE.
2. Sensor Struct:
   - id: unsigned char (for memory efficiency)
   - name: char array (max 20 characters)
   - type: Sensor type (enum)
   - data: Union for type-specific configuration and data:
     - Temperature: min_range (short int), max_range (short int), reading (float)
     - Humidity: calibration (float), reading (float)
     - Pressure: altitude (short int), reading (float)
   - status: Enum (ACTIVE, INACTIVE, ERROR)
3. Functions:
   - Initialize a sensor: Configure with type-specific settings (static array, max 10 sensors).
   - Read sensor data: Store latest reading.
   - Process sensor data:
     - Temperature: Flag ERROR if reading is outside min/max range.
     - Humidity: Apply calibration factor to reading.
     - Pressure: Adjust reading based on altitude compensation.
   - Display sensors: Show details and status.
4. Embedded System Optimization:
   - Use fixed-size arrays (no dynamic memory allocation).
   - Minimize memory usage with unions and appropriate data types.
   - Include basic error handling for invalid readings.
5. Simulation:
   - Simulate sensor readings with random values (no hardware I/O).

Example Usage

- Initialize a Temperature sensor: ID 1, “Temp1”, range -10 to 50°C.
- Initialize a Humidity sensor: ID 2, “Hum1”, calibration factor 1.05.
- Initialize a Pressure sensor: ID 3, “Pres1”, altitude compensation 100m.
- Read and process data for all sensors.
- Display sensor details, latest readings, and status.

Challenges

- Optimize memory usage with unions for sensor-specific data.
- Handle type-specific data processing in a constrained environment.
- Ensure robust error checking for sensor status.
- Simulate realistic sensor readings without hardware.

Sample Code

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

// Enum for sensor types
typedef enum {
    TEMPERATURE,
    HUMIDITY,
    PRESSURE
} SensorType;

// Enum for sensor status
typedef enum {
    ACTIVE,
    INACTIVE,
    ERROR
} SensorStatus;

// Union for sensor-specific configuration and data
typedef union {
    struct {
        short int min_range;  // Temperature range min (Celsius)
        short int max_range;  // Temperature range max (Celsius)
        float reading;      // Latest reading
    } temperature;
    struct {
        float calibration;  // Calibration factor
        float reading;      // Latest reading
    } humidity;
    struct {
        short int altitude;   // Altitude compensation (meters)
        float reading;      // Latest reading
    } pressure;
} SensorData;

// Struct for Sensor
typedef struct {
    unsigned char id;
    char name[20];
    SensorType type;
    SensorData data;
    SensorStatus status;
} Sensor;

// Function prototypes
void init_sensor(Sensor *sensors, unsigned char *count, unsigned char max_sensors);
void read_sensor_data(Sensor *sensor);
void process_sensor_data(Sensor *sensor);
void display_sensors(Sensor *sensors, unsigned char count);

// Main function with sample usage
int main() {
    Sensor sensors[10] = {0};  // Static array for max 10 sensors
    unsigned char count = 0;
    char choice;

    srand(time(NULL));  // Seed for simulated readings

    do {
        printf("\n1. Initialize Sensor\n2. Read Sensor Data\n3. Display Sensors\n4. Exit\n");
        printf("Enter choice: ");
        scanf(" %c", &choice);

        switch (choice) {
            case '1':
                init_sensor(sensors, &count, 10);
                break;
            case '2':
                for (unsigned char i = 0; i < count; i++) {
                    read_sensor_data(&sensors[i]);
                    process_sensor_data(&sensors[i]);
                }
                break;
            case '3':
                display_sensors(sensors, count);
                break;
            case '4':
                return 0;
            default:
                printf("Invalid choice!\n");
        }
    } while (1);

    return 0;
}

// Function implementations (to be completed)
void init_sensor(Sensor *sensors, unsigned char *count, unsigned char max_sensors) {
    // TODO: Initialize sensor with type-specific configuration
    // TODO: Validate inputs and ensure count < max_sensors
}

void read_sensor_data(Sensor *sensor) {
    // TODO: Simulate sensor reading with random values
}

void process_sensor_data(Sensor *sensor) {
    // TODO: Apply type-specific processing logic
    // TODO: Update sensor status based on processing
}

void display_sensors(Sensor *sensors, unsigned char count) {
    // TODO: Display sensor details, readings, and status
}
```