SMX - Soil Moisture Sensor v0.1
Description
SMX is a sophisticated soil moisture sensor system that combines impedance measurement with LoRaWAN communication capabilities. The system uses capacitive sensing technology with temperature compensation for accurate soil moisture measurements.

Hardware Components
- RAK4631 (nRF52840-based) - Main processor and LoRaWAN communication
- AD5933 - Impedance measurement IC
- TMP102 - Temperature sensor
- PCA9536D - I/O expander for sensor selection
- EEPROM - Configuration storage
- Dual-range moisture sensing circuit

Features
- Dual-range moisture measurement (High/Low gain)
- Temperature compensation
- LoRaWAN communication
- Configurable sleep intervals
- Low power operation
- Battery monitoring
- Remote configuration capability
- EEPROM-based calibration storage

Project Structure
```
SMX_v0.1/
├── src/
│   ├── config/
│   │   └── config.h          // System configuration and constants
│   ├── sensors/
│   │   ├── impedance_meter.h  // Soil moisture measurement
│   │   ├── impedance_meter.cpp
│   │   ├── temperature.h      // Temperature sensing
│   │   └── temperature.cpp
│   ├── communication/
│   │   ├── lora_handler.h     // LoRaWAN communication
│   │   ├── lora_handler.cpp
│   │   └── lora_config.h      // LoRaWAN settings
│   ├── power/
│   │   ├── power_manager.h    // Power management
│   │   └── power_manager.cpp
│   └── storage/
│       ├── eeprom_manager.h   // Configuration storage
│       └── eeprom_manager.cpp
├── include/
│   └── main.h                // Main header file
├── SMX_v0.1.ino              // Main Arduino file
└── README.md
```

Dependencies
Libraries
- LoRaWan-RAK4630
- Wire (I2C communication)
- SPI
- AD5933
- SparkFunTMP102
- PCA9536
- Adafruit_EEPROM_I2C

Hardware Requirements
- RAK4631 WisDuo module
- Custom sensor board with:
  - AD5933 impedance analyzer
  - TMP102 temperature sensor
  - PCA9536 I/O expander
  - EEPROM

Installation
1. Install Arduino IDE
2. Add RAK BSP to Arduino IDE
3. Install required libraries
4. Clone this repository
5. Open SMX_v0.1.ino in Arduino IDE

Configuration
LoRaWAN Settings
Update the following in `lora_config.h`:
```cpp
const uint8_t DEVICE_EUI[8] = {...};
const uint8_t APP_EUI[8] = {...};
const uint8_t APP_KEY[16] = {...};
```

Hardware Pins
Configured in `config.h`:
```cpp
namespace Pins {
   constexpr uint8_t EN = WB_SW1;
   constexpr uint8_t LOW_DIV = WB_IO2;
   constexpr uint8_t BATT = WB_A0;
   constexpr uint8_t C_SEL = WB_IO2;
   constexpr uint8_t EN_SEL = 1;
}
```

Usage
Initial Setup
1. Configure LoRaWAN credentials

2. Perform sensor calibration
3. Upload code to device
4. Verify LoRaWAN connection

Operation Modes
The device operates in four states:
1. INIT - System initialization
2. MEASUREMENT - Sensor reading
3. TRANSMIT - LoRaWAN communication
4. SLEEP - Power saving mode

LoRaWAN Commands
- 0x01: Update measurement interval
- 0x02: Request immediate measurement
- 0x03: System reset

Data Format
```
Payload (9 bytes):
[0-1]: Low gain moisture (%)
[2-3]: High gain moisture (%)
[4]: Temperature (°C)
[5]: Battery level (%)
[6-7]: Serial Number
[8]: Sleep interval (minutes)
```

Power Management
- Sleep current: <50μA
- Active measurement: ~15mA
- Transmission: ~40mA
- Configurable sleep intervals

Calibration
The system supports two-point calibration for each gain range:
1. Dry soil reference (0%)
2. Saturated soil reference (100%)
Values are stored in EEPROM for persistence.

Error Handling
- Communication errors
- Sensor failures
- Battery monitoring
- LoRaWAN connection issues

Contributing
1. Fork the repository
2. Create a feature branch
3. Commit changes
4. Push to the branch

5. Create Pull Request

Version History
- v0.1 (Current)
 - Initial release
 - Basic functionality
 - Dual-range measurements
 - LoRaWAN communication

Future Improvements
- [ ] Enhanced power optimization
- [ ] Advanced calibration routines
- [ ] Over-the-air firmware updates
- [ ] Enhanced error logging
- [ ] Web interface for configuration
- [ ] Data logging capabilities

License
[Your License Here]

Contact
[Your Contact Information]

Acknowledgments
- RAKwireless for WisDuo module
- Analog Devices for AD5933
- SparkFun for TMP102
- Community contributors

SMX - Soil Moisture Sensor v0.1
[Previous general description remains the same...]

Software Architecture

Core Components

1. Impedance Measurement System
```cpp
class ImpedanceMeter {
   static constexpr uint32_t START_FREQ = 99930;  // ~100kHz operating frequency
   static constexpr uint16_t FREQ_INCR = 10;      // 10Hz increments
   static constexpr uint8_t NUM_INCR = 12;        // 12 measurement points
   static constexpr uint8_t NUM_SAMPLES = 5;      // Samples for averaging
```
- Utilizes AD5933 for impedance spectroscopy
- Multiple frequency sweep for noise reduction
- Temperature compensation using empirical coefficient
- Averaging algorithm for measurement stability
- Capacitance calculation formula:
 ```
 ```

 Cin = 1E+12 / (2 * π * f * |Z|)
 where:
 - f = START_FREQ + (FREQ_INCR * NUM_INCR / 2)
 - |Z| = measured impedance magnitude
 ```

#### 2. Power Management System
```cpp
class PowerManager {
    static constexpr uint32_t STARTUP_DELAY_MS = 100;
    static constexpr float LOW_BATTERY_THRESHOLD = 20.0;
    static constexpr int BATTERY_SAMPLES = 5;
```
- State-based power control
- Configurable sleep modes
- Battery monitoring with voltage divider
- ADC averaging for stable readings
- Power consumption profiles:
 - Sleep: <50µA
 - Measurement: ~15mA
 - Transmission: ~40mA
 - Wake-up time: 100ms

3. LoRaWAN Communication
```cpp
namespace LoRaConfig {
    constexpr uint8_t JOIN_TRIALS = 8;
    constexpr uint8_t APP_PORT = LORAWAN_APP_PORT;
    constexpr uint8_t DATA_BUFF_SIZE = 64;
```
- OTAA authentication
- EU868 frequency band
- ADR disabled for stability
- Data Rate: DR_3
- Retry mechanism with backoff
- Payload structure:
 ```
 struct PayloadFormat {
     int16_t moistureL;    // 2 bytes
     int16_t moistureH;    // 2 bytes
     int8_t temperature;   // 1 byte
     uint8_t battery;      // 1 byte
     uint16_t serialNum;   // 2 bytes
     uint8_t interval;     // 1 byte
 } __attribute__((packed)); // Total: 9 bytes
 ```

4. Storage Management
```cpp
class EEPROMManager {

```
    static constexpr uint32_t WRITE_DELAY_MS = 5;
    static constexpr uint32_t INIT_DELAY_MS = 300;
```

- EEPROM layout:

```
Address | Size | Description
0x00    | 4    | Low Gain Calibration
0x0A    | 4    | High Gain Calibration
0x14    | 2    | Low Range Max Cap
0x1E    | 2    | High Range Max Cap
0x28    | 2    | Low Range Min Cap
0x32    | 2    | High Range Min Cap
0x3C    | 2    | Serial Number
0x46    | 1    | Sleep Time
```

- Write verification
- Wear leveling considerations
- Data integrity checks

State Machine Implementation

1. System States

```cpp
enum class SystemState {
    INIT,
    MEASUREMENT,
    TRANSMIT,
    SLEEP
};
```

State transitions and conditions:

```
INIT → MEASUREMENT:
- Condition: Successful sensor initialization
- Actions: Configure peripherals, load calibration

MEASUREMENT → TRANSMIT:
- Condition: Valid measurements obtained
- Actions: Process readings, prepare payload

TRANSMIT → SLEEP:
- Condition: Successful transmission or max retries
- Actions: Update timing, enter low power

SLEEP → MEASUREMENT:
- Condition: Timer expiry or external trigger
- Actions: Wake-up sequence, sensor power-up
```

2. Task Management
```cpp
SemaphoreHandle_t taskEvent;
SoftwareTimer taskWakeupTimer;
```
- FreeRTOS task synchronization
- Timer-based wake-up mechanism
- Event-driven state transitions
- Critical section handling

Software Optimization

1. Memory Management
- Static allocation for critical components
- Stack optimization in interrupt handlers
- Heap fragmentation prevention
- Buffer size optimization

2. Processing Efficiency
- Interrupt-based timing
- Efficient numeric calculations
- Optimized I2C communications
- Power-aware state transitions

Error Handling

1. Error Categories
```cpp
enum class ErrorCode {
  NONE = 0,
  SENSOR_INIT_FAILED = 1,
  CALIBRATION_INVALID = 2,
  MEASUREMENT_ERROR = 3,
  LORA_JOIN_FAILED = 4,
  TRANSMISSION_FAILED = 5,
  LOW_BATTERY = 6
};
```

2. Recovery Mechanisms
- Automatic retry logic
- Graceful degradation
- Watchdog implementation
- Error logging in EEPROM

Calibration System

1. Two-Point Calibration
```cpp
struct CalibrationPoint {
```

```
  float capacitance;
  float moisture;
  float temperature;
};
```

2. Temperature Compensation
```cpp
float tempCompensation(float capacitance, float temp) {
  return capacitance * (1 + TEMP_COEFF * (temp - REF_TEMP));
}
```

Remote Configuration

1. LoRaWAN Downlink Commands
```cpp
enum class LoRaCommand : uint8_t {
  UPDATE_INTERVAL = 0x01,
  REQUEST_MEASUREMENT = 0x02,
  SYSTEM_RESET = 0x03
};
```

2. Configuration Parameters
- Measurement interval
- Transmission power
- Gain settings
- Calibration values

Debugging Features

1. Serial Output Format
```cpp
Debug output format:
[timestamp][state][function] message
Example:
[1234567][MEAS][ImpedanceMeter] Reading: 234.5 pF
```

2. Debug Levels
```cpp
enum class DebugLevel {
  NONE,
  ERROR,
  WARN,
  INFO,
  DEBUG,
  VERBOSE
};
```

```

### Performance Metrics

#### 1. Timing
- Measurement cycle: ~500ms
- LoRaWAN join: <10s
- Wake-up time: <100ms
- Sleep transition: <10ms

#### 2. Accuracy
- Moisture: ±2% absolute
- Temperature: ±0.5°C
- Battery: ±2%

[Previous sections about Contributing, Version History, etc. remain the same...]

Would you like me to:
1. Add more implementation details for specific components?
2. Include code examples for common operations?
3. Add debugging procedures?
4. Expand on any specific section?

# SMX Technical Implementation Details

## 1. Impedance Measurement System

### A. AD5933 Configuration and Control
```cpp
class ImpedanceMeter {
private:
    // Frequency sweep parameters
    static constexpr uint32_t START_FREQ = 99930;  // Optimized for soil capacitance
    static constexpr uint16_t FREQ_INCR = 10;      // Small increment for noise reduction
    static constexpr uint8_t NUM_INCR = 12;        // Multiple points for averaging
    static constexpr uint8_t NUM_SAMPLES = 5;      // Sample averaging for stability

    // Measurement control
    static constexpr uint16_t PGA_GAIN = PGA_GAIN_X1;
    static constexpr uint8_t SETTLING_CYCLES = 15;
    static constexpr uint8_t SETTLING_MULTIPLIER = 4;
};
```

B. Measurement Sequence
```cpp
bool ImpedanceMeter::initialize() {
    // Configuration sequence
    if (!AD5933::reset()) return false;

```
    if (!AD5933::setInternalClock(true)) return false;
    if (!AD5933::setStartFrequency(START_FREQ)) return false;
    if (!AD5933::setIncrementFrequency(FREQ_INCR)) return false;
    if (!AD5933::setNumberIncrements(NUM_INCR)) return false;
    if (!AD5933::setPGAGain(PGA_GAIN)) return false;
    if (!AD5933::setSettlingCycles(SETTLING_CYCLES, SETTLING_MULTIPLIER)) return false;

    return true;
}

float ImpedanceMeter::measureImpedance(double gain) {
    float sumMagnitude = 0;
    uint8_t validSamples = 0;

    for (uint8_t sample = 0; sample < NUM_SAMPLES; sample++) {
        // Initialize sweep
        AD5933::setPowerMode(POWER_STANDBY);
        AD5933::setControlMode(CTRL_INIT_START_FREQ);
        AD5933::setControlMode(CTRL_START_FREQ_SWEEP);

        // Perform sweep
        double magnitude = performFrequencySweep();

        if (magnitude > 0) {
            sumMagnitude += magnitude;
            validSamples++;
        }

        // Short delay between samples
        delay(10);
    }

    AD5933::setPowerMode(POWER_DOWN);
    return (validSamples > 0) ? (sumMagnitude / validSamples) : -1;
}
```

C. Complex Impedance Calculation

```cpp
double ImpedanceMeter::performFrequencySweep() {
    int real, imag;
    double magnitude = 0;
    uint8_t measurements = 0;

    while ((AD5933::readStatusRegister() & STATUS_SWEEP_DONE) != STATUS_SWEEP_DONE) {
        if (AD5933::getComplexData(&real, &imag)) {
            // Calculate magnitude
            double currentMagnitude = sqrt(pow(real, 2) + pow(imag, 2));
```

```
            // Running average
            if (measurements == 0) {
                magnitude = currentMagnitude;
            } else {
                magnitude = (magnitude * measurements + currentMagnitude) / (measurements + 1);
            }
            measurements++;

            AD5933::setControlMode(CTRL_INCREMENT_FREQ);
        }
    }

    return magnitude;
}
```

D. Capacitance Calculation and Temperature Compensation

```cpp
int ImpedanceMeter::getMoisture(double gain, int Cmin, int Cmax, float temp) {
    // Get impedance measurement
    double impedance = measureImpedance(gain);
    if (impedance < 0) return -1;

    // Calculate capacitance
    double frequency = START_FREQ + (FREQ_INCR * NUM_INCR / 2);
    double Cin = 1E+12 / (2 * M_PI * frequency * impedance);

    // Apply temperature compensation
    Cin = tempCompensation(Cin, temp);

    // Convert to percentage
    return constrain(
        static_cast<int>(((Cin - Cmin) * 100.0) / (Cmax - Cmin)),
        0,
        100
    );
}

float ImpedanceMeter::tempCompensation(float capacitance, float temp) {
    static constexpr float TEMP_COEFF = 0.02;   // 2% per degree C
    static constexpr float REF_TEMP = 25.0;     // Reference temperature

    // Linear temperature compensation
    float tempDiff = temp - REF_TEMP;
    float compensationFactor = 1.0 + (TEMP_COEFF * tempDiff);

    return capacitance * compensationFactor;
}
```

E. Error Detection and Handling

```cpp
struct MeasurementResult {
  float impedance;
  float capacitance;
  float temperature;
  uint8_t errorCode;
  bool isValid;
};

MeasurementResult ImpedanceMeter::performMeasurement(double gain) {
  MeasurementResult result = {0};

  // Check system status
  if (!checkSystemStatus()) {
     result.errorCode = ERROR_SYSTEM_STATUS;
     return result;
  }

  // Perform measurement with bounds checking
  float impedance = measureImpedance(gain);
  if (impedance < MIN_VALID_IMPEDANCE || impedance > MAX_VALID_IMPEDANCE) {
     result.errorCode = ERROR_INVALID_IMPEDANCE;
     return result;
  }

  // Calculate capacitance
  result.impedance = impedance;
  result.capacitance = calculateCapacitance(impedance);
  result.isValid = true;

  return result;
}

bool ImpedanceMeter::checkSystemStatus() {
  uint8_t status = AD5933::readStatusRegister();

  // Check for system errors
  if (status & STATUS_SYSTEM_ERROR) {
     return false;
  }

  // Verify temperature is within bounds
  float temp = AD5933::getTemperature();
  if (temp < MIN_OPERATING_TEMP || temp > MAX_OPERATING_TEMP) {
     return false;
  }
```

```
    return true;
}
```

F. Calibration Support

```cpp
struct CalibrationPoint {
    float rawCapacitance;
    float actualMoisture;
    float temperature;
    float gainFactor;
};

class CalibrationManager {
public:
    bool calibrate(float knownCapacitance) {
        // Measure reference capacitor
        MeasurementResult result = measureReferenceCapacitor();
        if (!result.isValid) return false;

        // Calculate gain factor
        float gainFactor = knownCapacitance / result.capacitance;

        // Store calibration
        CalibrationPoint cal = {
            result.capacitance,
            knownCapacitance,
            result.temperature,
            gainFactor
        };

        return storeCalibration(cal);
    }

private:
    static constexpr float REFERENCE_TOLERANCE = 0.02; // 2%
    CalibrationPoint currentCalibration;
};
```

[Continue with next part? The remaining sections are:
1. Power Management System
2. LoRaWAN Communication
3. Storage System
4. State Machine
5. Task Management]

Let me know which section you'd like to see next!

2. Power Management System

A. Power States Implementation
```cpp
class PowerManager {
public:
   enum class PowerState {
       FULL_POWER,
       MEASUREMENT_MODE,
       LOW_POWER,
       SLEEP_MODE
   };

private:
   struct PowerProfile {
       bool i2cEnabled;
       bool spiEnabled;
       bool sensorsEnabled;
       bool loraEnabled;
       uint32_t targetCurrent;  // µA
   };

   const PowerProfile POWER_PROFILES[4] = {
       {true,  true,  true,  true,  40000}, // FULL_POWER
       {true,  false, true,  false, 15000}, // MEASUREMENT_MODE
       {false, false, false, true,  5000},  // LOW_POWER
       {false, false, false, false, 50}     // SLEEP_MODE
   };
};

void PowerManager::setPowerState(PowerState state) {
   const PowerProfile& profile = POWER_PROFILES[static_cast<int>(state)];

   // Configure I2C
   if (profile.i2cEnabled) {
       Wire.begin();
   } else {
       Wire.end();
   }

   // Configure SPI
   if (profile.spiEnabled) {
       SPI.begin();
   } else {
       SPI.end();
   }

   // Configure sensors
   configurePins(profile.sensorsEnabled);
```

```
    // Configure LoRa
    if (profile.loraEnabled) {
        Radio.Standby();
    } else {
        Radio.Sleep();
    }

    // Set system power mode
    if (state == PowerState::SLEEP_MODE) {
        sd_power_mode_set(NRF_POWER_MODE_LOWPWR);
    } else {
        sd_power_mode_set(NRF_POWER_MODE_NORMAL);
    }

    currentPowerState = state;
}
```

B. Battery Management
```cpp
class BatteryManager {
private:
    static constexpr uint16_t VMAX_MV = 4150;
    static constexpr uint16_t VMIN_MV = 3300;
    static constexpr uint8_t NUM_SAMPLES = 5;
    static constexpr uint8_t CRITICAL_LEVEL = 10;
    static constexpr uint8_t LOW_LEVEL = 20;

public:
    struct BatteryStatus {
        float voltageMillivolts;
        uint8_t percentage;
        bool isLow;
        bool isCritical;
        float temperature;
    };

    BatteryStatus getBatteryStatus() {
        BatteryStatus status = {0};

        // Configure ADC
        analogReference(AR_INTERNAL_3_0);
        analogReadResolution(12);

        // Take multiple samples
        float voltage = 0;
        for (int i = 0; i < NUM_SAMPLES; i++) {
            voltage += readBatteryVoltage();
            delay(10);
```

```
        }
        voltage /= NUM_SAMPLES;

        // Calculate status
        status.voltageMillivolts = voltage;
        status.percentage = calculatePercentage(voltage);
        status.isLow = (status.percentage <= LOW_LEVEL);
        status.isCritical = (status.percentage <= CRITICAL_LEVEL);
        status.temperature = readBatteryTemperature();

        return status;
    }

private:
    float readBatteryVoltage() {
        digitalWrite(Pins::LOW_DIV, LOW);
        delay(5);
        uint16_t raw = analogRead(Pins::BATT);
        digitalWrite(Pins::LOW_DIV, HIGH);

        return raw * BatteryConfig::REAL_MV_PER_LSB;
    }

    uint8_t calculatePercentage(float voltage) {
        float percentage = ((voltage - VMIN_MV) / (VMAX_MV - VMIN_MV)) * 100.0f;
        return constrain(static_cast<uint8_t>(percentage), 0, 100);
    }

    float readBatteryTemperature() {
        // Implementation for battery temperature monitoring
        return 25.0f; // Default room temperature if no sensor
    }
};
```

3. LoRaWAN Communication System

A. LoRaWAN Manager Implementation
```cpp
class LoRaWANManager {
public:
    struct LoRaConfig {
        uint8_t deviceEUI[8];
        uint8_t appEUI[8];
        uint8_t appKey[16];
        uint8_t dataRate;
        bool adrEnabled;
        uint8_t txPower;
        uint8_t retryCount;
        uint32_t joinTimeout;
```

```
    };

private:
    static constexpr uint8_t MAX_PAYLOAD_SIZE = 51;
    static constexpr uint8_t DEFAULT_PORT = 1;
    static constexpr uint32_t JOIN_RETRY_INTERVAL = 60000; // 1 minute

    LoRaConfig config;
    uint8_t txBuffer[MAX_PAYLOAD_SIZE];
    bool isJoined = false;
    uint8_t currentRetryCount = 0;

public:
    bool initialize(const LoRaConfig& cfg) {
        config = cfg;

        // Initialize radio
        if (lora_rak4630_init() != 0) {
            return false;
        }

        // Configure LoRaWAN parameters
        lmh_setDevEui(config.deviceEUI);
        lmh_setAppEui(config.appEUI);
        lmh_setAppKey(config.appKey);

        // Initialize LoRaWAN stack
        lmh_param_t lora_param_init = {
            config.adrEnabled ? LORAWAN_ADR_ON : LORAWAN_ADR_OFF,
            static_cast<dr_id_t>(config.dataRate),
            LORAWAN_PUBLIC_NETWORK,
            JOINREQ_NBTRIALS,
            config.txPower,
            LORAWAN_DUTYCYCLE_OFF
        };

        if (lmh_init(&loraCallbacks, lora_param_init, true,
                     CLASS_A, LORAMAC_REGION_EU868) != 0) {
            return false;
        }

        return true;
    }

    bool sendData(const uint8_t* data, uint8_t length, bool confirmed = false) {
        if (!isJoined || length > MAX_PAYLOAD_SIZE) {
            return false;
        }

        memcpy(txBuffer, data, length);
```

```
        lmh_error_status result = lmh_send(
            &lmh_app_data,
            confirmed ? LMH_CONFIRMED_MSG : LMH_UNCONFIRMED_MSG
        );

        return (result == LMH_SUCCESS);
    }

private:
    static void handleJoinedNetwork(void) {
        isJoined = true;
        currentRetryCount = 0;
    }

    static void handleJoinFailed(void) {
        isJoined = false;
        if (++currentRetryCount < config.retryCount) {
            // Schedule retry
            delay(JOIN_RETRY_INTERVAL);
            lmh_join();
        }
    }

    static void handleRxData(lmh_app_data_t* app_data) {
        processDownlinkCommand(app_data->buffer, app_data->buffsize);
    }
};
```

B. Payload Formatting
```cpp
class PayloadFormatter {
public:
    struct SensorData {
        int16_t moistureL;
        int16_t moistureH;
        int8_t temperature;
        uint8_t battery;
        uint16_t serialNum;
        uint8_t interval;
    };

    static uint8_t formatPayload(const SensorData& data, uint8_t* buffer) {
        uint8_t index = 0;

        // Add moisture readings
        buffer[index++] = data.moistureL & 0xFF;
        buffer[index++] = (data.moistureL >> 8) & 0xFF;
        buffer[index++] = data.moistureH & 0xFF;
```

```
        buffer[index++] = (data.moistureH >> 8) & 0xFF;

        // Add temperature and battery
        buffer[index++] = data.temperature;
        buffer[index++] = data.battery;

        // Add serial number
        buffer[index++] = data.serialNum & 0xFF;
        buffer[index++] = (data.serialNum >> 8) & 0xFF;

        // Add interval
        buffer[index++] = data.interval;

        return index;
    }

    static void parseDownlink(const uint8_t* buffer, uint8_t length) {
        if (length < 1) return;

        switch (buffer[0]) {
            case 0x01: // Update interval
                if (length >= 2) {
                    handleIntervalUpdate(buffer[1]);
                }
                break;

            case 0x02: // Request measurement
                handleMeasurementRequest();
                break;

            case 0x03: // System reset
                handleSystemReset();
                break;
        }
    }
};
```

[Continue with Storage System and State Machine implementation details? Let me know if you'd like to see those next!]

4. Storage System

A. EEPROM Manager Implementation
```cpp
class EEPROMManager {
public:
    struct StorageLayout {
        static constexpr uint16_t GAIN_L_ADDR = 0x00;
        static constexpr uint16_t GAIN_H_ADDR = 0x0A;
```

```
        static constexpr uint16_t CMAX_L_ADDR = 0x14;
        static constexpr uint16_t CMAX_H_ADDR = 0x1E;
        static constexpr uint16_t CMIN_L_ADDR = 0x28;
        static constexpr uint16_t CMIN_H_ADDR = 0x32;
        static constexpr uint16_t SNR_ADDR = 0x3C;
        static constexpr uint16_t SLEEP_TIME_ADDR = 0x46;
        static constexpr uint16_t ERROR_LOG_START = 0x50;
        static constexpr uint16_t CALIBRATION_DATA = 0x100;
    };

private:
    static constexpr uint8_t WRITE_RETRY_COUNT = 3;
    static constexpr uint8_t VERIFY_RETRY_COUNT = 2;
    static constexpr uint32_t WRITE_TIMEOUT_MS = 5;

    struct StorageMetadata {
        uint32_t writeCount;
        uint32_t lastWriteTime;
        uint16_t errorCount;
    };

public:
    template<typename T>
    bool writeData(uint16_t addr, const T& data) {
        uint8_t buffer[sizeof(T)];
        memcpy(buffer, &data, sizeof(T));

        for (uint8_t retry = 0; retry < WRITE_RETRY_COUNT; retry++) {
            bool success = true;

            // Write data
            for (uint8_t i = 0; i < sizeof(T); i++) {
                if (!writeByte(addr + i, buffer[i])) {
                    success = false;
                    break;
                }
                delay(WRITE_TIMEOUT_MS);
            }

            // Verify write
            if (success && verifyWrite(addr, buffer, sizeof(T))) {
                updateWriteMetadata(addr, sizeof(T));
                return true;
            }
        }

        logStorageError(addr, ERROR_WRITE_FAILED);
        return false;
    }
```

```
  template<typename T>
  bool readData(uint16_t addr, T& data) {
    uint8_t buffer[sizeof(T)];

    for (uint8_t retry = 0; retry < VERIFY_RETRY_COUNT; retry++) {
      if (readBytes(addr, buffer, sizeof(T))) {
        memcpy(&data, buffer, sizeof(T));
        return true;
      }
    }

    logStorageError(addr, ERROR_READ_FAILED);
    return false;
  }

private:
  bool writeByte(uint16_t addr, uint8_t data) {
    return LoraMem.write(addr, data);
  }

  bool readBytes(uint16_t addr, uint8_t* buffer, size_t size) {
    for (size_t i = 0; i < size; i++) {
      buffer[i] = LoraMem.read(addr + i);
    }
    return true;
  }

  bool verifyWrite(uint16_t addr, const uint8_t* data, size_t size) {
    uint8_t readBuffer[32];  // Max verification chunk size

    for (size_t i = 0; i < size; i += sizeof(readBuffer)) {
      size_t chunk = min(size - i, sizeof(readBuffer));
      if (!readBytes(addr + i, readBuffer, chunk)) {
        return false;
      }
      if (memcmp(data + i, readBuffer, chunk) != 0) {
        return false;
      }
    }
    return true;
  }

  void updateWriteMetadata(uint16_t addr, size_t size) {
    StorageMetadata metadata;
    readData(addr + size, metadata);
    metadata.writeCount++;
    metadata.lastWriteTime = millis();
    writeData(addr + size, metadata);
  }
```

```cpp
    void logStorageError(uint16_t addr, uint8_t errorCode) {
        struct ErrorLog {
            uint32_t timestamp;
            uint16_t address;
            uint8_t errorCode;
        } log = {millis(), addr, errorCode};

        writeData(StorageLayout::ERROR_LOG_START +
                 (errorCount % MAX_ERROR_LOGS) * sizeof(ErrorLog), log);
        errorCount++;
    }
};
```

B. Configuration Management
```cpp
class ConfigurationManager {
public:
    struct SystemConfig {
        SensorConfig sensorConfig;
        LoRaWANConfig loraConfig;
        PowerConfig powerConfig;
        CalibrationData calibration;
    };

private:
    EEPROMManager& eeprom;
    SystemConfig currentConfig;
    bool configLoaded = false;

public:
    bool loadConfiguration() {
        if (!eeprom.readData(EEPROMManager::StorageLayout::GAIN_L_ADDR,
                            currentConfig.sensorConfig.gainL)) {
            return false;
        }
        // Load other configuration parameters...
        configLoaded = true;
        return true;
    }

    bool saveConfiguration() {
        return eeprom.writeData(EEPROMManager::StorageLayout::GAIN_L_ADDR,
                               currentConfig.sensorConfig.gainL);
        // Save other configuration parameters...
    }

    bool updateParameter(ConfigParameter param, const void* value, size_t size) {
        uint16_t addr = getParameterAddress(param);
        if (addr == 0xFFFF) return false;
```

```
        return eeprom.writeData(addr, value, size);
    }

private:
    uint16_t getParameterAddress(ConfigParameter param) {
        switch (param) {
            case ConfigParameter::GAIN_L:
                return EEPROMManager::StorageLayout::GAIN_L_ADDR;
            // Other parameter addresses...
            default:
                return 0xFFFF;
        }
    }
};
```

5. State Machine Implementation

A. State Management System
```cpp
class StateMachine {
public:
    enum class State {
        INIT,
        MEASUREMENT,
        TRANSMIT,
        SLEEP
    };

    enum class Event {
        SYSTEM_INIT,
        MEASUREMENT_READY,
        TRANSMISSION_COMPLETE,
        TRANSMISSION_FAILED,
        SLEEP_TIMEOUT,
        ERROR_OCCURRED
    };

private:
    struct StateTransition {
        State currentState;
        Event event;
        State nextState;
        std::function<void()> action;
    };

    std::vector<StateTransition> transitions;
    State currentState = State::INIT;
```

```
public:
  StateMachine() {
    initializeTransitions();
  }

  void handleEvent(Event event) {
    for (const auto& transition : transitions) {
      if (transition.currentState == currentState &&
        transition.event == event) {
        // Execute transition action
        if (transition.action) {
          transition.action();
        }

        // Update state
        currentState = transition.nextState;

        // Notify observers
        notifyStateChange(currentState);
        return;
      }
    }
  }

private:
  void initializeTransitions() {
    // Initialize -> Measurement
    transitions.push_back({
      State::INIT,
      Event::SYSTEM_INIT,
      State::MEASUREMENT,
      [this]() {

powerManager.setPowerState(PowerManager::PowerState::MEASUREMENT_MODE);
        initializeSensors();
      }
    });

    // Measurement -> Transmit
    transitions.push_back({
      State::MEASUREMENT,
      Event::MEASUREMENT_READY,
      State::TRANSMIT,
      [this]() {
        powerManager.setPowerState(PowerManager::PowerState::FULL_POWER);
        prepareMeasurementData();
      }
    });

    // Transmit -> Sleep
```

        transitions.push_back({
            State::TRANSMIT,
            Event::TRANSMISSION_COMPLETE,
            State::SLEEP,
            [this]() {
                powerManager.setPowerState(PowerManager::PowerState::SLEEP_MODE);
                setupSleepTimer();
            }
        });

        // Sleep -> Measurement
        transitions.push_back({
            State::SLEEP,
            Event::SLEEP_TIMEOUT,
            State::MEASUREMENT,
            [this]() {
                powerManager.setPowerState(PowerManager::PowerState::MEASUREMENT_MODE);
                startMeasurement();
            }
        });

        // Error handling transitions
        transitions.push_back({
            State::MEASUREMENT,
            Event::ERROR_OCCURRED,
            State::SLEEP,
            [this]() {
                handleMeasurementError();
            }
        });
    }
};
```

### B. Event Handler Integration
```cpp
class EventHandler {
private:
    StateMachine& stateMachine;
    std::queue<StateMachine::Event> eventQueue;
    SemaphoreHandle_t eventSemaphore;

public:
    void postEvent(StateMachine::Event event) {
        // Add event to queue from ISR context
        BaseType_t xHigherPriorityTaskWoken = pdFALSE;

        if (xSemaphoreTakeFromISR(eventSemaphore,
                                  &xHigherPriorityTaskWoken) == pdTRUE) {

```
            eventQueue.push(event);
            xSemaphoreGiveFromISR(eventSemaphore, &xHigherPriorityTaskWoken);
        }

        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    }

    void processEvents() {
        if (xSemaphoreTake(eventSemaphore, portMAX_DELAY) == pdTRUE) {
            while (!eventQueue.empty()) {
                StateMachine::Event event = eventQueue.front();
                eventQueue.pop();

                stateMachine.handleEvent(event);
            }
            xSemaphoreGive(eventSemaphore);
        }
    }
};
```

C. Timer Management

```cpp
class TimerManager {
private:
    SoftwareTimer measurementTimer;
    SoftwareTimer transmissionTimer;
    SoftwareTimer watchdogTimer;
    EventHandler& eventHandler;

public:
    void initializeTimers() {
        // Setup measurement timer
        measurementTimer.begin(config.measurementInterval, [this]() {
            eventHandler.postEvent(StateMachine::Event::MEASUREMENT_READY);
        });

        // Setup transmission retry timer
        transmissionTimer.begin(TRANSMISSION_RETRY_INTERVAL, [this]() {
            eventHandler.postEvent(StateMachine::Event::TRANSMISSION_RETRY);
        });

        // Setup watchdog timer
        watchdogTimer.begin(WATCHDOG_INTERVAL, [this]() {
            checkSystemHealth();
        });
    }

private:
    void checkSystemHealth() {
```

```
        // Implement system health checks
        if (batteryTooLow() || systemError()) {
            eventHandler.postEvent(StateMachine::Event::ERROR_OCCURRED);
        }
    }
};
```

Would you like me to:
1. Add more implementation details for specific components?
2. Include error handling mechanisms?
3. Add debugging and logging systems?
4. Include calibration procedures?
5. Add power optimization techniques?

Let me know what additional details would be most helpful!