IOT Smart Tile System - Discussions and Conclusion

Executive Summary

This document reflects on the challenges encountered during the IOT Smart Tile System project, discusses how we addressed them, and provides our conclusions and recommendations for future work. It serves as both a retrospective analysis and a guide for anyone continuing development on this system.

Table of Contents

---

1. Discussions and Challenges

1.1 Technical Challenges

Challenge 1: ESP-NOW and WiFi Coexistence

The Problem:

ESP-NOW and WiFi both use the 2.4GHz radio on the ESP32, but they have conflicting requirements. WiFi wants to connect to an access point on a specific channel, while ESP-NOW needs all devices to be on the same channel. When WiFi connects, it changes the radio channel, breaking ESP-NOW communication with nodes that are still on the old channel.

How We Tackled It:

```
// Solution: Initialize ESP-NOW BEFORE WiFi connects
bool Coordinator::begin() {
    // 1. Initialize ESP-NOW first (uses default channel 1)
    bool espNowOk = espNow->begin();

    // 2. Link ESP-NOW to WiFi manager for channel sync
    wifi->setEspNow(espNow);

    // 3. Now WiFi can connect - it will notify ESP-NOW of channel change
    bool wifiReady = wifi->begin();

    // 4. When WiFi connects, WifiManager calls espNow->setChannel()
    // 5. Nodes receive channel info in join_accept and switch channels
}
```

Lessons Learned:

- Order of initialization matters critically in embedded systems
- Abstractions (WifiManager, EspNow) need to communicate about shared resources
- The `setEspNow()` dependency injection pattern solved the coupling problem elegantly

Challenge 2: NVS (Non-Volatile Storage) Corruption

The Problem:

During development, we frequently changed the data structures stored in NVS (node registry, WiFi credentials, MQTT config). This caused corruption errors where the ESP32 would fail to boot or read garbage data.

Symptoms:

```
E (234) nvs: nvs_open failed (0x1117) - NVS_NOT_INITIALIZED
E (567) nvs: wrong checksum in namespace 'nodes'
```

How We Tackled It:

We implemented a defensive NVS initialization pattern:

```
bool ConfigManager::begin() {
    esp_err_t err = nvs_flash_init();

    if (err == ESP_ERR_NVS_NO_FREE_PAGES ||
        err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        // NVS partition is corrupted - erase and reinitialize
        Logger::warn("NVS corrupted, erasing...");
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }

    if (err != ESP_OK) {
        Logger::error("NVS init failed: %s", esp_err_to_name(err));
        return false;
    }

    return true;
}
```

Lessons Learned:

- Always handle NVS corruption gracefully—it WILL happen during development
- Use namespaces to isolate different data types
- Document the NVS schema so team members don't step on each other's keys

Challenge 3: MQTT Broker Discovery

The Problem:

Hardcoding the MQTT broker IP address doesn't work when:

- Developer laptops have different IPs
- The broker runs in Docker with dynamic port mapping
- Moving between home and university networks

How We Tackled It:

We implemented a multi-strategy approach:

1. **NVS Storage**: Save last known good broker address
2. **Serial Provisioning**: Interactive setup on first boot
3. **mDNS Discovery**: Automatic discovery (partially implemented)
4. **Fallback Defaults**: Reasonable defaults for common setups

```
bool Mqtt::ensureConfigLoaded() {
    // Try loading from NVS first
    brokerHost = config.getString("host", "");

    if (brokerHost.isEmpty()) {
        // Try mDNS discovery
        if (discoverBrokerMdns()) {
            return true;
        }

        // Fall back to serial provisioning
        if (Serial) {
            return promptForConfig();
        }

        // Last resort: use default
        brokerHost = "192.168.1.100";
        return false;
    }

    return true;
}
```

Lessons Learned:

- IoT devices need multiple configuration strategies
- Serial provisioning is invaluable during development
- mDNS is powerful but adds complexity (multicast, timeouts)

Challenge 4: Message Protocol Versioning

The Problem:

As we iterated on the ESP-NOW message format, old nodes couldn't communicate with new coordinators and vice versa. Adding new fields broke JSON parsing on devices with older firmware.

How We Tackled It:

We adopted a backwards-compatible evolution strategy:

```
// Always use | operator with defaults for optional fields
bool JoinAcceptMessage::fromJson(const String& json) {
    DynamicJsonDocument doc(768);
    deserializeJson(doc, json);

    // Required fields
    node_id = doc["node_id"].as<String>();

    // Optional fields with defaults (backwards compatible)
    wifi_channel = doc["wifi_channel"] | 1;  // Default to 1 if missing
    cfg.rx_window_ms = doc["cfg"]["rx_window_ms"] | 20;
    cfg.rx_period_ms = doc["cfg"]["rx_period_ms"] | 100;

    return true;
}
```

Lessons Learned:

- Never remove fields from messages—only add new ones
- Always provide sensible defaults for new fields
- Consider adding a protocol version field for major changes

1.2 Integration Challenges

Challenge 5: Frontend-Backend WebSocket Schema Mismatch

The Problem:

The backend was sending WebSocket messages in one format, but the frontend expected a different structure. This caused silent failures where data was received but not displayed.

Backend sent:

```
{
  "type": "coordinator_telemetry",
  "coord_id": "coord001",
  "light_lux": 450
}
```

Frontend expected:

```
{
  "type": "telemetry",
  "payload": {
    "device_type": "coordinator",
    "coord_id": "coord001",
    "light_lux": 450
  }
}
```

How We Tackled It:

1. Created a shared `mqtt_api.md` document defining all message schemas
2. Added TypeScript interfaces that match the backend JSON exactly
3. Implemented adapter layer in frontend to handle legacy formats

```
// Frontend adapter for different message formats
private handleMessage(raw: any): WSMessage {
    // Handle legacy format
    if (raw.type === 'coordinator_telemetry') {
        return {
            type: 'telemetry',
            payload: {
                device_type: 'coordinator',
                ...raw
            }
        };
    }

    // New format passes through
    return raw as WSMessage;
}
```

Lessons Learned:

- Define API contracts BEFORE implementing both sides
- Use code generation or shared schema files when possible
- TypeScript interfaces should be the source of truth for frontend

Challenge 6: Time Synchronization

The Problem:

Embedded devices don't have a real-time clock. Their `millis()` timestamp starts at zero on boot. This made it impossible to correlate events across devices or with backend timestamps.

How We Tackled It:

We used Unix timestamps from the backend when available, and relative timestamps otherwise:

```
// Coordinator gets Unix time from MQTT broker/backend
void Mqtt::handleTimeSync(const String& payload) {
    uint32_t unixTime = payload.toInt();
    timeOffset = unixTime - (millis() / 1000);
}

uint32_t Coordinator::getUnixTimestamp() {
    return (millis() / 1000) + timeOffset;
}
```

Lessons Learned:

- Time synchronization is crucial for distributed systems

- NTP is overkill for our use case—simple time sync messages work
- Always include timestamps in telemetry for debugging

1.3 Team and Process Challenges

Challenge 7: Parallel Development on Shared Code

The Problem:

Multiple team members working on the `shared/` library caused frequent merge conflicts. The `EspNowMessage.h` file was modified by everyone.

How We Tackled It:

1. **Clear ownership**: Assigned one person as "message protocol owner"
2. **Feature branches**: All changes go through pull requests
3. **Interface stability**: Freeze message formats after initial design
4. **Documentation**: Comments in code explaining field purposes

Lessons Learned:

- Shared code needs clear ownership
- Define interfaces early and resist changes
- Code review catches integration issues early

Challenge 8: Testing Without Hardware

The Problem:

Not everyone had access to ESP32 boards. Testing embedded code required physical hardware, slowing development.

How We Tackled It:

1. **Mock objects**: Created mock classes for hardware-dependent code
2. **Native builds**: Used PlatformIO's native environment for unit tests
3. **Integration tests**: Backend tests with mocked MQTT messages
4. **Shared test fixtures**: JSON files with sample messages

```
// Example: Mock EspNow for testing
class MockEspNow : public IEspNow {
public:
    void addPeer(const uint8_t* mac) override {
        addedPeers.push_back(macToString(mac));
    }

    std::vector<String> addedPeers;
};
```

Lessons Learned:

- Design for testability from the start
- Hardware abstraction enables testing without devices
- Integration tests are more valuable than unit tests for IoT

1.4 Most Difficult Parts

🏆 #1: ESP-NOW Pairing Flow

Why It Was Difficult:

The pairing flow involves multiple devices, multiple messages, timeouts, and state machines all working together. Any bug could leave devices in inconsistent states.

```
Node                    Coordinator
  |                          |
  |--- JOIN_REQUEST -------->|  (Node must be in pairing mode)
  |                          |  (Coordinator must have pairing window open)
  |<-- JOIN_ACCEPT ----------|  (Contains channel, encryption key, node ID)
  |                          |
  |  [Node switches channel] |
  |  [Node saves credentials]|
  |                          |
  |--- STATUS -------------->|  (Confirms pairing complete)
  |                          |
```

Complexity Factors:

- Timing: Pairing window timeout, message retry timeout, channel switch delay
- State: Both devices must track pairing state independently
- Security: Encryption key exchange, replay attack prevention
- Persistence: Both devices must save pairing info to NVS

How We Managed It:

- Extensive logging at every step
- State machine diagrams before coding
- Timeout values tuned through trial and error
- Visual feedback (LED colors) to show pairing state

🏆 #2: Real-time Data Pipeline

Why It Was Difficult:

Getting data from a sensor to the dashboard involves 6 hops:

```
Sensor → Node → ESP-NOW → Coordinator → MQTT → Backend → WebSocket → Frontend
```

Any issue at any hop causes data loss or delays. Debugging requires correlating logs across all components.

Complexity Factors:

- Asynchronous communication at every hop
- Different data formats (binary, JSON, TypeScript)
- Network unreliability (WiFi drops, broker disconnects)
- Backpressure (what if frontend can't keep up?)

How We Managed It:

- Added timestamps and sequence numbers to trace messages
- Centralized logging with structured format
- Built debug dashboard showing connection status of each hop
- Implemented reconnection logic at every layer

🏆 **#3: Power Management on Nodes**

Why It Was Difficult:

Battery-powered nodes need to sleep to conserve power, but they also need to wake up to receive commands. ESP32 deep sleep turns off the radio, so messages are lost.

The Tradeoff:

Sleep Mode	Power Draw	Can Receive?	Wake Latency
Active	100-200mA	Yes	0ms
Light Sleep	2-5mA	With radio on	<1ms
Deep Sleep	10-20μA	No	200-500ms

Why We Didn't Fully Solve It:

- Light sleep with radio on still draws too much for weeks of battery life
- Deep sleep requires scheduled wake or external interrupt
- We didn't have time to implement proper sleep scheduling

1.5 Key Realizations

Realization 1: IoT is Mostly About Reliability, Not Features

> *"A smart light that works 99% of the time is worse than a dumb light that works 100% of the time."*

We spent more time on error handling, reconnection logic, and fallback modes than on new features. This is the right priority for IoT.

Realization 2: The "Last Mile" is the Hardest

Getting data from the cloud to the embedded device is harder than getting it to the cloud. Downlink commands face:

- Device might be asleep
- Device might be offline
- Command might arrive out of order
- No way to confirm receipt without acknowledgment

Realization 3: JSON is Good Enough

We considered binary protocols (Protocol Buffers, MessagePack) for efficiency, but JSON worked fine:

- Our payloads are small (<250 bytes)
- Debugging is 10x easier with human-readable messages
- ArduinoJson library is fast and memory-efficient

Realization 4: Serial Logging is Non-Negotiable

Every embedded developer knows this, but it bears repeating: without Serial logging, debugging embedded code is nearly impossible. We added logging to every function entry, exit, and error condition.

Realization 5: Integration Testing > Unit Testing for IoT

Unit tests are useful, but the real bugs appear when components interact. We should have invested more in end-to-end integration tests:

```
# The test we wish we had:
# 1. Start all services in Docker
# 2. Flash coordinator with test firmware
# 3. Simulate node pairing
# 4. Verify data appears in database
# 5. Verify dashboard updates
```

1.6 What We Should Do Differently

1. Define APIs Before Implementation

What We Did: Started coding immediately, defined APIs as we went.

What We Should Have Done: Spent first week defining:

- MQTT topic structure
- JSON message schemas
- REST API endpoints
- WebSocket message formats

This would have prevented numerous integration bugs and rework.

2. Set Up CI/CD Earlier

What We Did: Manual builds and testing throughout the project.

What We Should Have Done:

```yaml
# GitHub Actions workflow we should have had
name: CI
on: [push, pull_request]
jobs:
  build-coordinator:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Build Coordinator
        run: pio run -e esp32-s3-devkitc-1

  build-backend:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Build Backend
        run: go build ./cmd/iot

  test-integration:
    runs-on: ubuntu-latest
    steps:
      - name: Start services
        run: docker-compose up -d
      - name: Run integration tests
        run: go test ./internal/...
```

3. Use Structured Configuration

What We Did: Hardcoded values, environment variables, scattered config files.

What We Should Have Done: Single configuration source with validation:

```yaml
# config.yaml - single source of truth
coordinator:
  site_id: "site001"
  coord_id: "coord001"

mqtt:
  broker: "tcp://localhost:1883"
  qos: 1

sensors:
  telemetry_interval_ms: 5000
  mmwave:
    enabled: true
    sensitivity: 0.8
```

4. Implement Feature Flags

What We Did: All features always enabled, hard to test individual components.

What We Should Have Done:

```c
// Feature flags for development
#define FEATURE_MMWAVE_ENABLED    1
#define FEATURE_DEEP_SLEEP        0  // Disabled until stable
#define FEATURE_OTA_UPDATES       0  // Not implemented yet
#define FEATURE_ENCRYPTION        1

#if FEATURE_MMWAVE_ENABLED
    mmWave = new MmWave();
    mmWave->begin();
#endif
```

5. Document Hardware Setup

What We Did: Tribal knowledge about wiring, pin assignments, power requirements.

What We Should Have Done: Created hardware setup guide with:

- Wiring diagrams
- Pin assignment tables
- Power budget calculations
- BOM (Bill of Materials)

---

2. Conclusion

2.1 Project Assessment

Overall: The Project Went Well ✅

Despite the challenges, we successfully delivered a working IoT system that demonstrates the key concepts of the course:

Objective	Status	Notes
Embedded sensor network	✅ Achieved	Coordinator + nodes communicating via ESP-NOW
Cloud connectivity	✅ Achieved	MQTT bridge to backend working reliably
Data persistence	✅ Achieved	MongoDB storing telemetry and device state
Real-time dashboard	✅ Achieved	WebSocket updates showing live data
Local automation	✅ Achieved	Presence detection triggers lighting
Mobile/remote access	✅ Achieved	Web dashboard accessible from anywhere

What Went Well

1. **Architecture held up**: The Manager pattern and modular design made it easy to add features and debug issues.
2. **Team collaboration**: Despite working on different components, we integrated successfully with minimal conflicts.
3. **Learning curve**: Everyone on the team gained practical experience with embedded systems, MQTT, and full-stack development.
4. **Working demo**: We have a functional system that can be demonstrated end-to-end.

What Could Have Been Better

1. **Power management**: Nodes don't achieve target battery life.
2. **Security**: No TLS, basic encryption only.
3. **Documentation**: Started documentation late in the project.
4. **Testing**: Insufficient automated testing, especially integration tests.

2.2 Course Feedback

Positive Aspects

1. **Hands-on learning**: Building a real IoT system taught us more than any textbook could. The practical experience with ESP32, MQTT, and embedded C++ is invaluable.
2. **Full-stack exposure**: The project covered embedded, backend, frontend, and DevOps. This breadth is rare in university courses.
3. **Freedom to choose technologies**: Being able to select our own frameworks (Go, Angular) let us learn industry-relevant skills.
4. **Real hardware**: Working with physical devices (not just simulators) made the challenges feel authentic.

Suggestions for Improvement

1. **Earlier hardware distribution**: Getting ESP32 boards earlier would give more time for embedded development.
2. **MQTT/Protocol lecture earlier**: Understanding MQTT patterns before starting implementation would have saved rework.
3. **Sample project structure**: A minimal working example showing the data path from sensor to dashboard would accelerate initial development.
4. **More on debugging techniques**: A session on debugging distributed systems (correlating logs, tracing messages) would be valuable.
5. **Power management focus**: Deep sleep and battery optimization deserves dedicated coverage—it's where IoT differs most from traditional software.

2.3 General Comments

On the Project

The IOT Smart Tile System was an ambitious project that pushed us to learn across multiple domains. The most valuable learning came from the integration challenges—making different technologies work together is the essence of IoT engineering.

We underestimated the complexity of seemingly simple features (like pairing) and overestimated how quickly we could add new capabilities. This is a valuable lesson for future projects.

On IoT Development in General

IoT development is fundamentally different from traditional software:

- **Reliability matters more than features**: Users tolerate bugs in apps, but not in their light switches.
- **Updates are hard**: You can't just deploy a fix; devices need OTA or physical access.
- **Power is a first-class concern**: Every line of code has a power cost.
- **Debugging is harder**: No breakpoints, limited visibility, physical access required.

These differences require a different mindset and different tooling than web or mobile development.

On Team Dynamics

Working on a full-stack IoT project with a team requires clear communication and interface definitions. Our biggest integration issues came from misunderstandings about data formats and API contracts. In future projects, we would invest more time upfront in defining these interfaces.

---

3. Future Work

If we were to continue developing the IOT Smart Tile System, here are the features and improvements we would prioritize:

3.1 Short-term Improvements (1-2 weeks)

1. Implement OTA Firmware Updates

Priority: High
Complexity: Medium

```
// Planned implementation:
void Coordinator::handleOtaCommand(const String& payload) {
    DynamicJsonDocument doc(256);
    deserializeJson(doc, payload);

    String firmwareUrl = doc["url"].as<String>();
    String expectedHash = doc["sha256"].as<String>();

    // Download firmware
    HTTPClient http;
    http.begin(firmwareUrl);

    // Verify hash before flashing
    // Flash to OTA partition
    // Reboot
}
```

2. Add TLS Encryption for MQTT

Priority: High
Complexity: Low

```
// Use WiFiClientSecure instead of WiFiClient
WiFiClientSecure secureClient;
secureClient.setCACert(ca_cert);
secureClient.setCertificate(client_cert);
secureClient.setPrivateKey(client_key);

PubSubClient mqtt(secureClient);
mqtt.setServer("broker.example.com", 8883);
```

3. Complete Deep Sleep Implementation

Priority: High
Complexity: High

```
// Proper deep sleep with scheduled wake
void Node::enterDeepSleep() {
    // Save state to RTC memory
    rtcData.lastWakeTime = millis();
    rtcData.wakeCount++;

    // Configure wake sources
    esp_sleep_enable_timer_wakeup(SLEEP_DURATION_US);
    esp_sleep_enable_ext0_wakeup(BUTTON_PIN, LOW);

    // Enter deep sleep
    esp_deep_sleep_start();
}
```

3.2 Medium-term Features (1-2 months)

4. Mobile App (React Native)

Motivation: Web dashboard isn't ideal for quick interactions.

Features:

- Push notifications for presence alerts
- Widget for quick light control
- Offline support with local caching

5. Voice Control Integration

Motivation: Hands-free control is expected for smart home devices.

Platforms:

- Google Home (partially implemented via `googlehome` module)
- Amazon Alexa
- Apple HomeKit (via HomeBridge)

6. Scene and Schedule System

Motivation: Users want automated behaviors, not just remote control.

```
interface Scene {
    id: string;
    name: string;
    actions: Action[];
    triggers: Trigger[];
}

interface Trigger {
    type: 'time' | 'presence' | 'button' | 'sunset';
    condition: any;
}

// Example: "Movie Mode" - dim lights when presence detected after 8pm
const movieScene: Scene = {
    id: 'movie-mode',
    name: 'Movie Mode',
    actions: [
        { type: 'set_light', target: 'all', brightness: 20, color: 'warm' }
    ],
    triggers: [
        { type: 'presence', condition: { detected: true } },
        { type: 'time', condition: { after: '20:00', before: '23:00' } }
    ]
};
```

7. Energy Monitoring Dashboard

Motivation: Users want to understand power consumption.

Features:

- Estimated power usage per node
- Historical consumption graphs
- Battery life predictions
- Alerts for low battery

3.3 Long-term Vision (6+ months)

8. Mesh Networking Between Nodes

Motivation: Extend range beyond coordinator's reach.

Implementation: Use ESP-MESH or PainlessMesh library to allow nodes to relay messages.

```
Current:     Node ↔ Coordinator ↔ Node
                        ↓
                      Cloud

Future:      Node ↔ Node ↔ Node
                     ↑
                Coordinator
                     ↓
                   Cloud
```

Challenges:

- Routing algorithm complexity
- Increased latency for relayed messages
- Power consumption for relay nodes

9. Machine Learning for Presence Prediction

Motivation: Predictive automation feels magical.

Approach:

- Collect historical presence data
- Train model to predict when users will arrive
- Pre-warm lights before predicted arrival

```python
# Example: Simple arrival prediction
from sklearn.ensemble import RandomForestClassifier

# Features: day_of_week, hour, minute, last_presence_time
# Target: will_arrive_in_next_10_minutes

model = RandomForestClassifier()
model.fit(historical_data, arrival_labels)

# In production:
prediction = model.predict([current_features])
if prediction == 1:
    mqtt.publish("site/001/coord/001/cmd", {"cmd": "prepare_lights"})
```

10. Multi-Site Management

Motivation: Scale to multiple homes/buildings.

Features:

- Central management console for all sites
- Cross-site analytics
- Bulk firmware updates
- Role-based access control

11. Hardware Product Development

Motivation: Move from prototype to product.

Tasks:

- Custom PCB design (reduce cost, improve reliability)
- Enclosure design (3D printed → injection molded)
- CE/FCC certification
- Manufacturing partnership

Final Thoughts

The IOT Smart Tile System project was a challenging but rewarding experience. We set out to build a complete IoT solution from embedded firmware to cloud dashboard, and we achieved that goal. Along the way, we learned not just technical skills, but also how to manage complexity, work as a team, and make pragmatic engineering decisions.

The system we built is a solid foundation. It demonstrates the core patterns of IoT development:

- Local-first architecture for reliability and responsiveness
- Cloud connectivity for remote access and analytics
- Modular design for maintainability and extensibility

While there's much more we could add (OTA, security, power optimization), the architecture supports these improvements. Anyone continuing this project will find a well-structured codebase with clear separation of concerns.

Our key takeaway: IoT development is about managing the inherent complexity of distributed, heterogeneous systems. The best architectures acknowledge this complexity and provide clear abstractions to contain it. We hope our work on the IOT Smart Tile System demonstrates these principles in practice.

Document Version: 1.0
Last Updated: December 2024
Authors: Group 15 - IOT Smart Tile System Team