ESP32-P4 Networking Workshop

Hands-On Workshop: Basic Networking to Secure MQTT

ESP32-P4 Function EV Board + ESP32-C6

3-Part Workshop • Basic Networking • Custom Firmware • Secure MQTT

Workshop Agenda

Part 1: Basic Networking

Foundation (45 min)

- ESP32-P4 + ESP-NETIF Overview
- Ethernet RMII Implementation
- WiFi Remote Configuration
- Network Event Handling
- **Hands-on:** Basic Connectivity

Part 2: ESP32-C6 Firmware

Customization (60 min)

- ESP-Hosted Architecture
- Custom Firmware Development
- UART/JTAG Programming
- SDIO Communication
- Advanced WiFi Features
- **Hands-on:** Custom Firmware

Part 3: Secure MQTT

Production Security (45 min)

- MQTT + TLS Fundamentals
- X.509 Certificate Management
- Mutual Authentication
- Secure Key Storage
- **Hands-on:** Secure Communication

Total Duration: ~2.5 hours with hands-on practice

ESP32-P4 + ESP32-C6 Architecture

ESP32-P4 (Host)

- **400MHz Dual RISC-V** cores
- **768KB SRAM** + external PSRAM
- **Integrated Ethernet MAC** (RMII)
- **Hardware crypto** acceleration
- **55 GPIO** pins

ESP32-C6 (Co-processor)

- **WiFi 6 + Bluetooth 5** support
- **SDIO interface** to ESP32-P4
- **Programmable firmware** (ESP-Hosted)
- **Custom protocol** support
- **Advanced wireless** features

Workshop Focus

Modular design enables flexible networking solutions with high-performance wired and wireless connectivity

Part 1: WiFi Networking - Hands-On First!

Our Approach: Action → Understanding

1. **Get it working** - WiFi connection in 5 minutes
2. **Understand why** - ESP-NETIF architecture
3. **See the magic** - WiFi Remote component
4. **Build on success** - Extend your working code

What We'll Build

- Minimal WiFi connection project
- Using ESP-IDF's proven patterns
- **Same approach** as all protocol examples

Learning Objectives

- **Experience** immediate success with WiFi
- Understand protocol_examples_common pattern
- Learn ESP-NETIF through working code
- See WiFi Remote transparency in action
- **Build confidence** for complex applications

Tools We'll Use

- `protocol_examples_common` component
- `esp_wifi_remote` for ESP32-P4
- Standard ESP-IDF initialization pattern

Philosophy: Working code first, theory second. Let's connect!

Quick Start: Create Project & Dependencies

1. Create Project

```
idf.py create-project networking
cd networking
```

2. Add Dependencies

Create `main/idf_component.yml`:

```
dependencies:
  protocol_examples_common:
    path: ${IDF_PATH}/examples/common_components/protocol_examples_common
  espressif/esp_wifi_remote: "^0.14.4"
```

That's it! These two components give you everything you need for WiFi on ESP32-P4.

Minimal Working Code

3. Replace `main/wifi_quickstart.c`

```c
#include "esp_system.h"
#include "esp_log.h"
#include "nvs_flash.h"
#include "esp_netif.h"
#include "esp_event.h"
#include "protocol_examples_common.h"
static const char *TAG = "wifi_quickstart";
void app_main(void)
{
    ESP_LOGI(TAG, "[APP] Startup..");
    // Initialize required systems
    ESP_ERROR_CHECK(nvs_flash_init());
    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_event_loop_create_default());
    // Connect to WiFi
    ESP_ERROR_CHECK(example_connect());
    ESP_LOGI(TAG, "Connected! Ready for networking");
}
```

Configure WiFi Settings

3. Set Target & Configure

```
idf.py set-target esp32p4
idf.py menuconfig
```

Navigate to: **Example Connection Configuration →**

Set these values:

- ✅ **connect using WiFi interface**
- **WiFi SSID**: `[Workshop WiFi Name]`
- **WiFi Password**: `[Workshop WiFi Password]`
- ❌ connect using Ethernet interface

4. Build & Flash

```
idf.py build
idf.py flash monitor
```

Expected Output:

```
I (xxx) wifi_quickstart: [APP] Startup..
I (xxx) example_connect: Start example_connect.
I (xxx) example_connect: Connecting to [WiFi Name]...
I (xxx) example_connect: Connected to AP SSID:[WiFi Name]
I (xxx) example_connect: Got IPv4 address: 192.168.1.100
I (xxx) wifi_quickstart: Connected! Ready for networking
```

Success! WiFi connected in under 5 minutes! 🎉

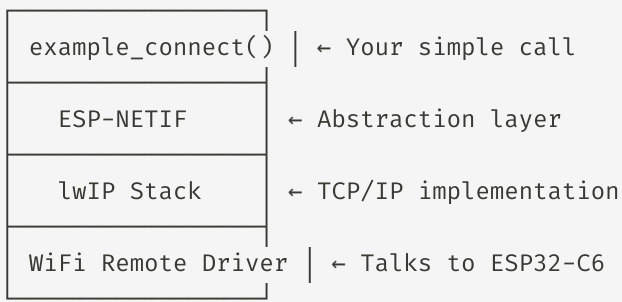
Workshop WiFi: SSID and password will be shared during the session

What Just Happened? ESP-NETIF Explained

Your Code Called This:

```
ESP_ERROR_CHECK(esp_netif_init());
ESP_ERROR_CHECK(example_connect());
```

ESP-NETIF Did This:



Why This Matters

- **One API** works for WiFi, Ethernet, any transport
- **Your app doesn't care** how networking happens
- **Same pattern** in all ESP-IDF protocol examples

The Power

- Change network type? **Code stays same**
- Add multiple interfaces? **ESP-NETIF handles it**
- Switch hardware? **Application unchanged**

ESP-NETIF = Write once, network anywhere

ESP-NETIF: Manual Initialization Patterns

Manual Interface Creation

Beyond `esp_netif_create_default_wifi_sta()`:

```
// Custom WiFi station configuration
esp_netif_config_t netif_config = ESP_NETIF_DEFAULT_WIFI_
esp_netif_t* netif = esp_netif_new(&netif_config);
// Attach to WiFi driver
esp_netif_attach_wifi_station(netif);
```

Custom Configuration

```
// Set hostname before DHCP
esp_netif_set_hostname(netif, "my-esp32-p4");
// Configure custom DHCP behavior
esp_netif_dhcpc_option(netif,
    ESP_NETIF_OP_SET, ESP_NETIF_DOMAIN_NAME_SERVER,
    &dns_server, sizeof(dns_server));
```

Status Monitoring Functions

```
// Get IP as string (for logging)
esp_netif_ip_info_t ip_info;
esp_netif_get_ip_info(netif, &ip_info);
char ip_str[16];
snprintf(ip_str, sizeof(ip_str), IPSTR, IP2STR(&ip_info.i
// Check connection health
bool has_ip = (ip_info.ip.addr != 0);
ESP_LOGI(TAG, "IP: %s, Connected: %s",
         ip_str, has_ip ? "YES" : "NO");
```

ESP-NETIF - Recommended setup patterns

Use default interface creation

```
esp_netif_create_default_wifi_sta();
esp_netif_create_default_wifi_ap();
```

Setting Interface Priority

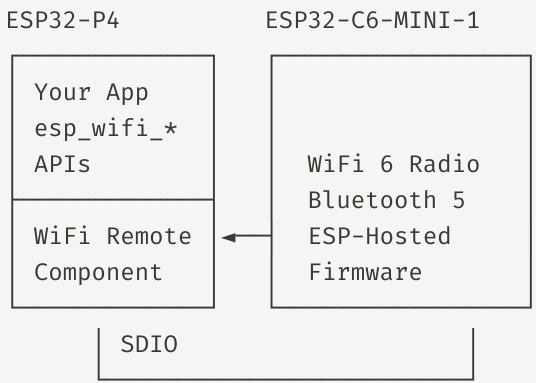
```
esp_netif_t* wifi_netif = esp_netif_create_default_wifi_sta();
esp_netif_t* eth_netif = esp_netif_new(ESP_NETIF_DEFAULT_ETH());

// Set priority (lower = higher priority)
esp_netif_set_route_prio(wifi_netif, 10);
esp_netif_set_route_prio(eth_netif, 20);
```

The Magic: WiFi Remote Component

Your `idf_component.yml` added:

```
espressif/esp_wifi_remote: "^0.14.4"
```

The Magic Architecture



Why This Works

- **ESP32-C6** handles all wireless operations
- **WiFi Remote** bridges ESP32-P4 ↔ ESP32-C6
- **Standard WiFi APIs** work unchanged

The Result

```
esp_wifi_connect(); // Just works! ✨
```

This architecture means your system will:

- ✅ Use standard `esp_wifi_*` APIs
- ✅ Handle events as usual
- ✅ Work with any ESP32-P4 board setup

Protocol Examples Common - Quickstart

Why `protocol_examples_common`?

Provides a quickstart point to fast start

What `example_connect()` Does

```
ESP_ERROR_CHECK(example_connect());
```

This single line handles:

- WiFi initialization, Credential configuration, Connection establishment, Event handling, Retry logic, IP address assignment

Configuration via Menuconfig

```
Example Connection Configuration  ⟶
    [*] connect using WiFi interface
    (YourSSID) WiFi SSID
    (YourPassword) WiFi Password
    [ ] connect using Ethernet interface
```

All The Complex Stuff Hidden

Instead of 100+ lines of WiFi code, you get:

```
ESP_ERROR_CHECK(example_connect());
```

But What If You Need More?

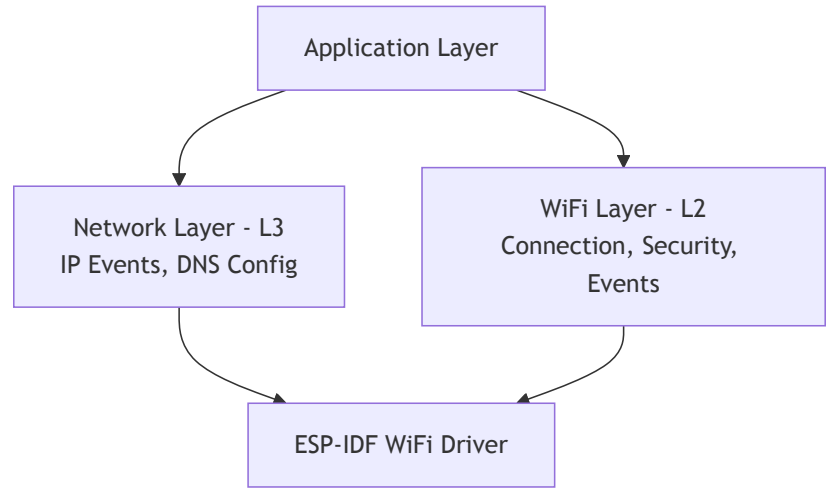
- Custom retry strategies
- Security configuration
- Event handling for diagnostics
- Production error recovery

Time to Build Your Own Connection Manager

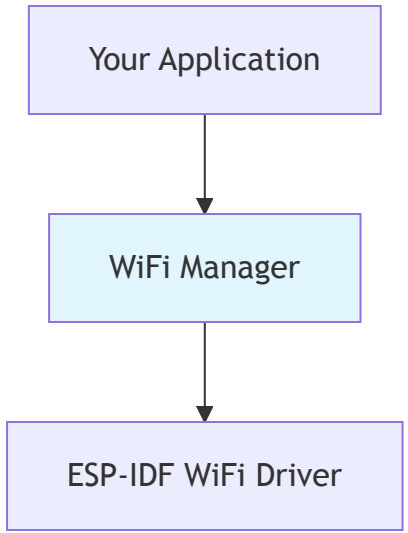
You've Seen the Magic ✨

- `example_connect()` got you connected instantly
- But what's happening inside that black box?
- What if you need custom behavior?

Production-Ready Architecture



WiFi Manager: Core Concepts



Key Responsibilities:

- 🔧 Configuration management
- 🔄 Connection lifecycle control
- 📡 Event handling & monitoring
- 🛡️ Security policy enforcement

Lifecycle Pattern

```
Init → Configure → Start → Monitor → Destroy
```

Why This Pattern?

- **Predictable**: Clear state progression
- **Testable**: Each phase can be validated
- **Robust**: Proper resource management
- **Flexible**: Can pause/resume as needed

WiFi Manager: Interface Design

Configuration Structure

```c
// my_wifi.h
typedef struct {
    char ssid[32];          // WiFi network name
    char password[64];      // WiFi password
    uint8_t max_retry;      // Maximum connection attempt
    uint32_t timeout_ms;    // Connection timeout
} my_wifi_config_t;
```

Core Functions

```c
// Initialize WiFi subsystem
esp_err_t my_wifi_init(const my_wifi_config_t *config);
// Start connection (blocking until success/failure)
esp_err_t my_wifi_start(void);
// Check current connection status
bool my_wifi_is_connected(void);
// Clean shutdown and resource cleanup
esp_err_t my_wifi_destroy(void);
```

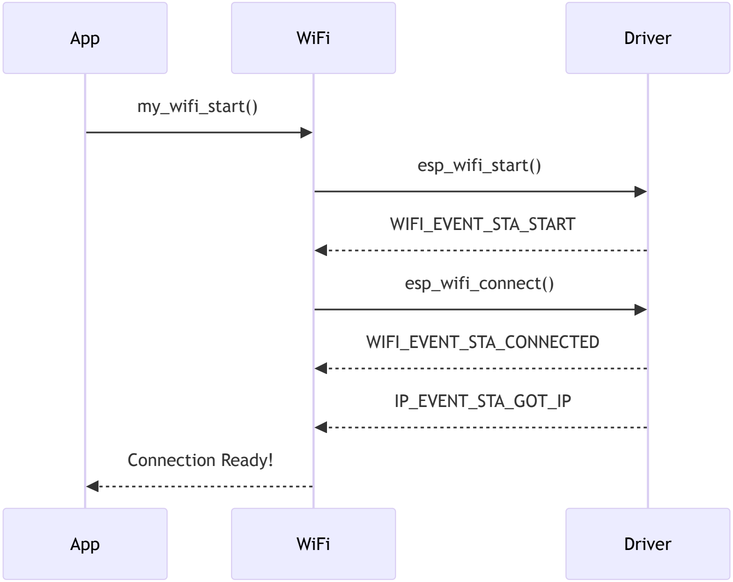
Usage Pattern

```c
void app_main(void) {
    // 1. Configure
    my_wifi_config_t config = {
        .ssid = "WorkshopWiFi",
        .password = "workshop123",
        .max_retry = 5,
        .timeout_ms = 10000
    };
    // 2. Initialize
    ESP_ERROR_CHECK(my_wifi_init(&config));
    // 3. Start (blocks until connected)
    ESP_ERROR_CHECK(my_wifi_start());
    // 4. Use connection
    if (my_wifi_is_connected()) {
        // Your application logic here
    }
    // 5. Cleanup when done
    my_wifi_destroy();
}
```

WiFi Events: Understanding the Flow

Event-Driven Programming

WiFi operates asynchronously - events tell you what happened



Why Events Matter

- **Asynchronous**: WiFi connection takes time
- **Stateful**: Track connection progress
- **Reactive**: Respond to network changes

Connection State Management

```
IDLE] → [CONNECTING] → [CONNECTED] → [MONITORING

    ↑                                                ↓
    ←────────────[DISCONNECTED]←──────────────────┘
```

Event-driven state transitions enable:

- Non-blocking operations
- Real-time status updates
- Intelligent retry logic
- Security monitoring

WiFi Event Types & Categories

Connection Events

- `WIFI_EVENT_STA_START`
- `WIFI_EVENT_STA_CONNECTED`
- `WIFI_EVENT_STA_DISCONNECTED`

Security Events

- `WIFI_EVENT_STA_AUTHMODE_CHANGE`
- `WIFI_EVENT_STA_BEACON_TIMEOUT`

Quality Events

- `WIFI_EVENT_STA_BSS_RSSI_LOW`
- `WIFI_EVENT_SCAN_DONE`

IP Layer Events

- `IP_EVENT_STA_GOT_IP`
- `IP_EVENT_STA_LOST_IP`

Why Comprehensive Event Handling?

Production Benefits:

- **Reliability**: Handle all failure scenarios
- **Security**: Detect potential attacks
- **User Experience**: Meaningful status updates
- **Diagnostics**: Debug connection issues effectively

Event-Driven Architecture Benefits

- **Non-blocking**: WiFi operations don't freeze your app
- **Responsive**: React immediately to network changes
- **Scalable**: Handle multiple network conditions
- **Maintainable**: Clear separation of concerns

WiFi Events: Implementation

Basic Event Handler Structure

```
static void wifi_event_handler(void *arg,
                               esp_event_base_t event_base
                               int32_t event_id,
                               void *event_data) {
    if (event_base == WIFI_EVENT) {
        switch (event_id) {
            case WIFI_EVENT_STA_START:
                ESP_LOGI(TAG, "WiFi started, connecting..
                esp_wifi_connect();
                break;
            case WIFI_EVENT_STA_CONNECTED:
                ESP_LOGI(TAG, "WiFi connected to AP");
                s_is_connected = true;
                break;
            case WIFI_EVENT_STA_DISCONNECTED:
                handle_disconnect(event_data);
                break;
            default:
                ESP_LOGI(TAG, "Unhandled WiFi event: %ld"
        }
    }
}
```

Disconnect Handling with Reason Codes

```
static void handle_disconnect(void *event_data) {
    wifi_event_sta_disconnected_t *event =
        (wifi_event_sta_disconnected_t *)event_data;
    const char *reason = wifi_reason_to_string(event->rea
    ESP_LOGW(TAG, "Disconnected from %s, reason: %s",
             event->ssid, reason);
    s_is_connected = false;
    // Smart retry logic
    if (should_retry(event->reason) &&
        s_retry_count < s_max_retry) {

        vTaskDelay(pdMS_TO_TICKS(1000)); // Brief delay
        esp_wifi_connect();
        s_retry_count++;
    } else {
        ESP_LOGE(TAG, "Connection failed permanently");
        xEventGroupSetBits(s_event_group, WIFI_FAIL_BIT);
    }
}
```

WiFi Disconnect Reason Codes

Reason Code Mapping (Essential 15)

```
static const char* wifi_reason_to_string(uint8_t reason)
    switch (reason) {
        case WIFI_REASON_AUTH_FAIL: return "Auth failed";
        case WIFI_REASON_ASSOC_FAIL: return "Assoc failed
        case WIFI_REASON_BEACON_TIMEOUT: return "Beacon t
        case WIFI_REASON_NO_AP_FOUND: return "No AP found
        case WIFI_REASON_CONNECTION_FAIL: return "Connect
        case WIFI_REASON_4WAY_HANDSHAKE_TIMEOUT: return "
        case WIFI_REASON_MIC_FAILURE: return "MIC failure
        case WIFI_REASON_ASSOC_TOOMANY: return "Too many
        case WIFI_REASON_ASSOC_EXPIRE: return "Assoc expi
        case WIFI_REASON_HANDSHAKE_TIMEOUT: return "Hands
        case WIFI_REASON_CIPHER_SUITE_REJECTED: return "C
        case WIFI_REASON_802_1X_AUTH_FAILED: return "802.
        case WIFI_REASON_AUTH_EXPIRE: return "Auth expire
        case WIFI_REASON_ROAMING: return "Roaming";
        default: return "Unknown";
    }
}
```

Smart Retry Logic

```
static bool should_retry(uint8_t reason) {
    switch (reason) {
        // Retry these network issues
        case WIFI_REASON_BEACON_TIMEOUT:
        case WIFI_REASON_NO_AP_FOUND:
        case WIFI_REASON_CONNECTION_FAIL:
        case WIFI_REASON_ASSOC_TOOMANY:
            return true;

        // Don't retry auth/security failures
        case WIFI_REASON_AUTH_FAIL:
        case WIFI_REASON_4WAY_HANDSHAKE_TIMEOUT:
        case WIFI_REASON_MIC_FAILURE:
        case WIFI_REASON_CIPHER_SUITE_REJECTED:
            return false;

        default:
            return true; // Conservative
    }
}
```

Network Layer: Separation of Concerns

Network Layer Interface Design

Network Manager API

```c
// my_network.h
typedef enum {
    NETWORK_EVENT_GOT_IP,
    NETWORK_EVENT_LOST_IP,
} network_event_t;

typedef void (*network_event_cb_t)(network_event_t event,

// Core functions
esp_err_t my_network_init(network_event_cb_t callback);
bool my_network_has_ip(void);
esp_err_t my_network_get_ip_string(char *ip_str);
esp_err_t my_network_get_gateway_string(char *gw_str);
esp_err_t my_network_destroy(void);
```

Application Callback Pattern

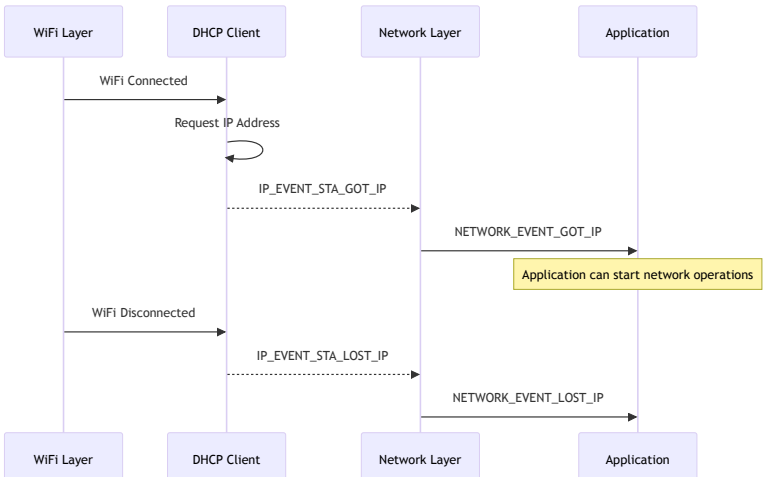
```c
void app_network_callback(network_event_t event, void *da
    switch (event) {
        case NETWORK_EVENT_GOT_IP:
            ESP_LOGI(TAG, "Network ready");
            // TODO: Start your network services
            break;
        case NETWORK_EVENT_LOST_IP:
            ESP_LOGW(TAG, "Network lost");
            // TODO: Pause network operations
            break;
    }
}

// TODO: Add to your app_main()
ESP_ERROR_CHECK(my_network_init(app_network_callback));
```

Key Benefits

- **Layer 3 focus**: IP address management separate from WiFi
- **Callback-driven**: React to network state changes

IP Events: Understanding Network Layer

IP Event Lifecycle



Key Timing:

- IP events happen **after** WiFi events
- DHCP negotiation takes additional time
- IP can be lost without WiFi disconnect

IP Event Types & When They Occur

IP_EVENT_STA_GOT_IP

IP_EVENT_STA_LOST_IP

IP_EVENT_GOT_IP6

Network Monitoring Strategy

Monitor both layers independently:

```
WiFi Status: Connected | Disconnected
IP Status:   Has IP    | No IP
```

Production Considerations

- Handle IP loss during operations
- Automatic IP renewal attempts
- Pause/resume network services
- Accurate connectivity status reporting

IP Events: Event Handler Implementation

IP Event Handler

```
static void ip_event_handler(void *arg,
                             esp_event_base_t event_base,
                             int32_t event_id,
                             void *event_data) {
    if (event_base == IP_EVENT) {
        switch (event_id) {
            case IP_EVENT_STA_GOT_IP: {
                ip_event_got_ip_t *event =
                    (ip_event_got_ip_t *)event_data;
                ESP_LOGI(TAG, "Got IP: " IPSTR,
                         IP2STR(&event->ip_info.ip));
                ESP_LOGI(TAG, "Gateway: " IPSTR,
                         IP2STR(&event->ip_info.gw));
                s_has_ip = true;
                break;
            }
            case IP_EVENT_STA_LOST_IP:
                ESP_LOGW(TAG, "Lost IP address");
                s_has_ip = false;

                break;
        }
```

Event Registration

```
ESP_ERROR_CHECK(esp_event_handler_instance_register(
    IP_EVENT, ESP_EVENT_ANY_ID,
    &ip_event_handler, NULL, &instance_ip));
```

Handler Pattern Benefits

- **State tracking**: Maintains `s_has_ip` flag
- **Application callbacks**: Notifies app layer
- **Structured logging**: Clear network information
- **Proper cleanup**: Sets state on IP loss

Key Implementation Details

- **Event data casting**: `ip_event_got_ip_t`
- **Conservative approach**: Clear state immediately on IP loss

IP Events: Network Status API

Network Status Functions

```c
esp_err_t my_network_get_ip_string(char *ip_str) {
    if (!ip_str) return ESP_ERR_INVALID_ARG;

    if (!s_has_ip) {
        strcpy(ip_str, "0.0.0.0");
        return ESP_ERR_INVALID_STATE;
    }

    // Get current IP from netif
    esp_netif_t *netif = esp_netif_get_handle_from_ifkey(
    if (!netif) return ESP_FAIL;

    esp_netif_ip_info_t ip_info;
    ESP_ERROR_CHECK(esp_netif_get_ip_info(netif, &ip_info

    snprintf(ip_str, 16, IPSTR, IP2STR(&ip_info.ip));
    return ESP_OK;
}

bool my_network_has_ip(void) {
    return s_has_ip;
}
```

Production Usage

```c
// Quick connectivity check
if (my_network_has_ip()) {
    // Safe to start network operations
    start_http_client();
}

// Get IP for logging/debugging
char ip_str[16];
if (my_network_get_ip_string(ip_str) == ESP_OK) {
    ESP_LOGI(TAG, "Device IP: %s", ip_str);
}
```

Smart Retry: Network Instability Patterns

Temporary Issues (Should Retry)

- **Signal interference**: Temporary radio congestion
- **AP overload**: Too many clients connecting
- **DHCP delays**: Slow IP address assignment
- **Roaming**: Moving between access points

Permanent Issues (Should NOT Retry)

- **Wrong password**: Authentication will keep failing
- **Security mismatch**: Incompatible encryption
- **MAC filtering**: Device blocked by AP
- **Hidden network**: SSID not found

Decision Strategy

Rule of thumb: If the problem is likely to resolve itself (signal, congestion, timing), retry. If it's a configuration issue (credentials, security), don't spam the network.

Smart Retry: Strategy Comparison

Naive Retry (Poor)

```
Attempt 1: Immediate
Attempt 2: Immediate
Attempt 3: Immediate
Result: Network spam, battery drain
```

Fixed Delay (Better)

```
Attempt 1: Immediate
Attempt 2: Wait 1 second
Attempt 3: Wait 1 second
Result: Predictable, but not adaptive
```

Exponential Backoff (Best)

```
Attempt 1: Immediate
Attempt 2: Wait 1 second
Attempt 3: Wait 2 seconds
Attempt 4: Wait 4 seconds
Result: Adaptive to network conditions
```

Smart Retry

- **Reason-based**: Different strategies per disconnect cause
- **Progressive backoff**: Increase delays over time
- **Maximum attempts**: Prevent infinite retry loops
- **User notification**: Inform application of failure

Smart Retry: Logic Implementation

Retry Configuration

```
static struct {
    uint8_t current_attempt;
    uint8_t max_attempts;
    uint32_t base_delay_ms;
} s_retry = {0, 5, 1000};
```

Decision Logic

```
static bool should_retry(uint8_t reason) {
    switch (reason) {
        // Retry these network issues
        case WIFI_REASON_BEACON_TIMEOUT:
        case WIFI_REASON_NO_AP_FOUND:
        case WIFI_REASON_CONNECTION_FAIL:
        case WIFI_REASON_ASSOC_TOOMANY:
            return true;
        // Don't retry auth/security failures
        case WIFI_REASON_AUTH_FAIL:
        case WIFI_REASON_4WAY_HANDSHAKE_TIMEOUT:
        case WIFI_REASON_MIC_FAILURE:
            return false;
        default:
```

Disconnect Handler

```
static void handle_disconnection(
    wifi_event_sta_disconnected_t *event) {
    ESP_LOGW(TAG, "Disconnected: reason %d", event->reaso
    s_is_connected = false;
    if (!should_retry(event->reason)) {
        ESP_LOGE(TAG, "Permanent failure");
        xEventGroupSetBits(s_event_group, WIFI_FAIL_BIT);
        return;
    }
    if (s_retry.current_attempt >= s_retry.max_attempts)
        ESP_LOGE(TAG, "Max retries reached");
        xEventGroupSetBits(s_event_group, WIFI_FAIL_BIT);
        return;
    }
    // Exponential backoff: 1s, 2s, 4s, 8s, 16s
    uint32_t delay = s_retry.base_delay_ms << s_retry.cur
    ESP_LOGI(TAG, "Retry %d in %lu ms",
             s_retry.current_attempt + 1, delay);
    vTaskDelay(pdMS_TO_TICKS(delay));
    esp_wifi_connect();
    s_retry.current_attempt++;
}
```

Smart Retry: Success Handling

Reset Counter on Success

```
// Reset counter on successful connection
case WIFI_EVENT_STA_CONNECTED:
    ESP_LOGI(TAG, "WiFi connected successfully");
    s_retry.current_attempt = 0; // Reset for next time
    s_is_connected = true;
    xEventGroupSetBits(s_event_group, WIFI_CONNECTED_BIT)
    break;
```

Integration in Event Handler

```
static void wifi_event_handler(void *arg,
                               esp_event_base_t event_base
                               int32_t event_id,
                               void *event_data) {
    switch (event_id) {
        case WIFI_EVENT_STA_DISCONNECTED:
            handle_disconnection(event_data);
            break;

        case WIFI_EVENT_STA_CONNECTED:
            s_retry.current_attempt = 0;
            s_is_connected = true;
```

Key Implementation Features

- **Exponential backoff**: 1→2→4→8→16 seconds
- **Reason-aware**: Uses should_retry() logic
- **Bounded retries**: Prevents infinite loops
- **Reset on success**: Ready for next disconnect event

State Management

```
// Global state tracking
static bool s_is_connected = false;

// Event group bits for coordination
#define WIFI_CONNECTED_BIT BIT0
#define WIFI_FAIL_BIT      BIT1

// FreeRTOS event group for task synchronization
static EventGroupHandle_t s_event_group;
```

Disconnect Reason Codes: Essential Mapping

Essential Reason Codes

```
const char *wifi_reason_to_string(uint16_t reason) {
    switch (reason) {
        // Auth/Security issues (don't retry)
        case WIFI_REASON_AUTH_FAIL:
            return "Wrong password";
        case WIFI_REASON_4WAY_HANDSHAKE_TIMEOUT:
            return "Handshake failed";
        case WIFI_REASON_MIC_FAILURE:
            return "Encryption error";
        case WIFI_REASON_CIPHER_SUITE_REJECTED:
            return "Cipher rejected";
        // Network issues (retry with backoff)
        case WIFI_REASON_BEACON_TIMEOUT:
            return "Signal lost";
        case WIFI_REASON_NO_AP_FOUND:
            return "AP not found";
        case WIFI_REASON_ASSOC_TOOMANY:
            return "AP overloaded";
        case WIFI_REASON_CONNECTION_FAIL:
            return "Connection failed";
        default: return "Unknown";
```

Usage in Disconnect Handler

```
static void handle_disconnect(
    wifi_event_sta_disconnected_t *event) {

    const char *reason = wifi_reason_to_string(event→rea
    ESP_LOGW(TAG, "Disconnect: %s (code %d)",
             reason, event→reason);

    // Use should_retry() logic
    if (should_retry(event→reason)) {
        // Implement retry with backoff
    } else {
        // Report permanent failure
    }
}
```

Disconnect Reason Codes: Diagnostic Categories

🔐 **Security/Auth Issues** → Don't retry

📡 **Network Issues** → Retry with backoff

⚠️ **Security Alerts** → Log and investigate

Production Implementation

```
typedef enum {
    WIFI_FAILURE_AUTH,      // Don't retry
    WIFI_FAILURE_NETWORK,   // Retry with backoff
    WIFI_FAILURE_SECURITY,  // Log and alert
    WIFI_FAILURE_UNKNOWN    // Conservative retry
} wifi_failure_category_t;

wifi_failure_category_t categorize_failure(uint8_t reason
    switch (reason) {
        case WIFI_REASON_AUTH_FAIL:
        case WIFI_REASON_4WAY_HANDSHAKE_TIMEOUT:
        case WIFI_REASON_MIC_FAILURE:
            return WIFI_FAILURE_AUTH;
        case WIFI_REASON_BEACON_TIMEOUT:
        case WIFI_REASON_NO_AP_FOUND:
        case WIFI_REASON_ASSOC_TOOMANY:
            return WIFI_FAILURE_NETWORK;
        default:
            return WIFI_FAILURE_UNKNOWN;
    }
}
```

Event Handler Registration

```c
static esp_event_handler_instance_t instance_wifi_any;
static esp_event_handler_instance_t instance_ip_any;

esp_err_t register_event_handlers(void) {
    // Register WiFi event handler
    ESP_ERROR_CHECK(esp_event_handler_instance_register(
        WIFI_EVENT,              // Event base
        ESP_EVENT_ANY_ID,        // All WiFi events
        &wifi_event_handler,     // Handler function
        NULL,                    // Handler argument
        &instance_wifi_any       // Instance handle
    ));
    // Register IP event handler
    ESP_ERROR_CHECK(esp_event_handler_instance_register(
        IP_EVENT,                // Event base
        ESP_EVENT_ANY_ID,        // All IP events
        &ip_event_handler,       // Handler function
        NULL,                    // Handler argument
        &instance_ip_any         // Instance handle
    ));
    ESP_LOGI(TAG, "Event handlers registered");
    return ESP_OK;
}
```

Registration Best Practices

- **Store instance handles**
- **Register early**
- **Check return values**

Integration Example

```c
esp_err_t my_wifi_init(const my_wifi_config_t *config) {
    // Initialize event loop first
    ESP_ERROR_CHECK(esp_event_loop_create_default());
    // Register handlers before WiFi init
    ESP_ERROR_CHECK(register_event_handlers());
    // Initialize WiFi stack
    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));
    return ESP_OK;
}
```

Handler Arguments

Event Handler Cleanup & Integration

Handler Cleanup

```
esp_err_t unregister_event_handlers(void) {
    esp_err_t ret = ESP_OK;

    // Unregister WiFi handler
    if (instance_wifi_any) {
        ret = esp_event_handler_instance_unregister(
            WIFI_EVENT, ESP_EVENT_ANY_ID, instance_wifi_a
        if (ret ≠ ESP_OK) {
            ESP_LOGE(TAG, "Failed to unregister WiFi hand
        }
        instance_wifi_any = NULL;
    }

    // Unregister IP handler
    if (instance_ip_any) {
        ret = esp_event_handler_instance_unregister(
            IP_EVENT, ESP_EVENT_ANY_ID, instance_ip_any);
        if (ret ≠ ESP_OK) {
            ESP_LOGE(TAG, "Failed to unregister IP handle
        }
        instance_ip_any = NULL;
    }
```

Complete Manager Integration

```
esp_err_t my_wifi_init(const my_wifi_config_t *config) {
    // ... initialization steps ...

    // Register event handlers
    ESP_ERROR_CHECK(register_event_handlers());

    return ESP_OK;
}

esp_err_t my_wifi_destroy(void) {
    // Unregister handlers first
    unregister_event_handlers();

    // Then cleanup WiFi
    esp_wifi_deinit();
    esp_netif_deinit();

    return ESP_OK;
}
```

Cleanup Best Practices

Integration: System Architecture Overview

Complete System Architecture

Integration Principles

Sequential Initialization

Order matters for proper system startup:

```
1. Base Systems (NVS, NETIF, Event Loop)
2. WiFi Manager (Configuration, Handlers)
3. Network Manager (IP Event Monitoring)
4. Start Connection (Blocking until ready)
5. Application Logic (Network operations)
```

Resource Lifecycle Management

```
Initialize → Configure → Start → Monitor → Cleanup
```

Error Propagation

- **Initialization errors**: Fail fast, don't continue
- **Runtime errors**: Retry logic and graceful degradation

Integration: Complete app_main() Implementation

Final Workshop Implementation

```
void app_main(void) {
    ESP_LOGI(TAG, "Starting ESP32-P4 networking workshop"

    // 1. Base ESP-IDF initialization
    ESP_ERROR_CHECK(nvs_flash_init());
    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_event_loop_create_default());

    // 2. Your custom managers (workshop tasks)
    my_wifi_config_t wifi_config = {
        .ssid = "WorkshopWiFi",
        .password = "workshop123",
        .max_retry = 5
    };

    ESP_ERROR_CHECK(my_wifi_init(&wifi_config));
    ESP_ERROR_CHECK(my_network_init(app_network_callback)

    // 3. Start connection and monitoring
    ESP_ERROR_CHECK(my_wifi_start());
    app_main_loop(); // Your monitoring implementation
}
```

Workshop Checklist

✅ **Tasks to complete:**

1. **my_wifi_init()** - Initialize WiFi manager
2. **my_network_init()** - Initialize network layer
3. **my_wifi_start()** - Connect with retry logic
4. **app_network_callback()** - Handle IP events
5. **app_main_loop()** - Monitor connection status

📝 **TODO Integration points:**

- Replace `example_connect()` with your managers
- Add comprehensive event handling
- Implement smart retry logic

Part 2: Wireless Connectivity for ESP32-P4

Espressif provides four main solutions for adding Wi-Fi and Bluetooth:

- **ESP-AT**: Simple, AT command-based interface for basic connectivity. Easy to use, but limited in performance and flexibility.
- **ESP-Extconn**: Integrates external wireless connectivity using familiar ESP-IDF APIs. Flexible, but requires more development effort. low SoC supported
- **Wi-Fi Remote over ESP-Hosted**: Provides a standard network interface to the host. Supports advanced features, higher performance, and is open-source for customization.
- **Wi-Fi Remote over EPPP**: General-purpose PPP connectivity for WiFi gateway applications. Uses standard PPP protocol with support for multiple transports and logical channels.

Feature Comparison

Feature	ESP-AT	ESP-Extconn	ESP-Hosted	EPPP Link
Interfaces	UART	SDIO	SPI/SDIO/UART	UART/SPI/SDIO/Ethernet
Customization	Limited	Zero	Full source	High
Ease	Simple	Complex	Moderate	Moderate
Performance	Basic	High	High	Good (2-11 Mbps)
ESP SoC Supported	ESP32, ESP32-C2/C3/C6, ESP32-S2	ESP8689	ESP32, ESP32-C2/C3/C6, ESP32-S2/S3	ESP32, ESP32-C2/C3/C6, ESP32-S2/S3

Hands-On: ESP-Hosted Build & Flash

Task 1: Build ESP-Hosted Firmware

1. Clone ESP-Hosted repo
2. Configure for ESP32-C6
3. Build firmware binary

Task 2: Flash ESP32-C6

1. Enter programming mode
2. Flash firmware
3. Monitor boot

Success Criteria

- ✅ Firmware built successfully
- ✅ ESP32-C6 programmed with new firmware
- ✅ Device boots and is ready

Time: 30 minutes

ESP-Hosted: Build Steps

1. Clone ESP-Hosted

```
idf.py create-project-from-example "espressif/esp_hosted:slave"
cd esp_hosted_slave
```

2. Configure for ESP32-C6

```
idf.py set-target esp32c6
idf.py menuconfig
```

In menuconfig, confirm that the connection type is set to SDIO (default).

3. Build Firmware

```
idf.py build
```

Flashing ESP-Hosted Firmware

Connections: ESP-Prog to P4 Board

- Use the ESP-Prog debugger to connect to the ESP32-C6 on the P4 board.
- Refer to the diagram below for wiring:

Flashing ESP-Hosted Firmware

Enter Programming Mode:

- Power the esp32p4-function-board trough the **POWER-IN** usb-c.
- Connect the ESP-Prog to your computer.
- Halt P4:
  1. Press and hold the **Boot** button.
  2. Press & Release the **Reset** button (while still holding Boot).
  3. Release the **Boot** button.
- Halt C6 (Same as above, but using the ESP-Prog Buttons.)

Flash with:

```
idf.py -p /dev/ttyUSB0 flash
```

- Replace /dev/ttyUSB0 with the correct port to your ESP-Prog.

Part 3: Secured MQTT with Mutual Authentication

What We'll Cover

- **MQTT + TLS** fundamentals
- **X.509 certificates** and PKI
- **Mutual authentication** implementation
- **Secure key storage** in NVS
- **Production security** best practices

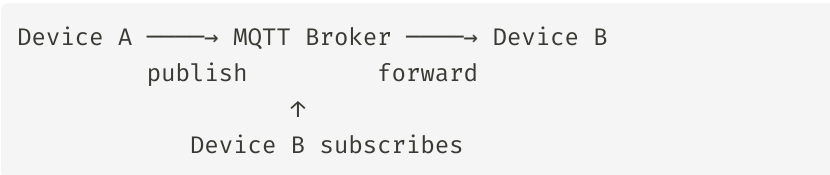
Learning Objectives

- Understand MQTT security architecture
- Implement certificate-based authentication
- Configure mutual TLS (mTLS)
- Secure credential management
- **Deploy production-ready** security

Goal: Build enterprise-grade secure IoT communication system

MQTT: Publish/Subscribe Messaging

Message Flow



Pattern: Publisher → Broker → Subscribers

EOF < /dev/null

Key Features

- **Topics** - Message routing (`sensors/temp`)
- **QoS Levels** - Delivery guarantees (0,1,2)
- **Retain** - Persistent messages
- **Last Will** - Offline detection

MQTT Security Fundamentals

Security Challenges

- **Unencrypted** MQTT traffic
- **Weak authentication** (username/password)
- **No message integrity** verification
- **Identity spoofing** possibilities
- **Man-in-the-middle** attacks

Security Solutions

- **TLS encryption** for transport security
- **X.509 certificates** for identity
- **Mutual authentication** (client + server)
- **Message integrity** via digital signatures
- **Certificate validation** and pinning

Security Layers

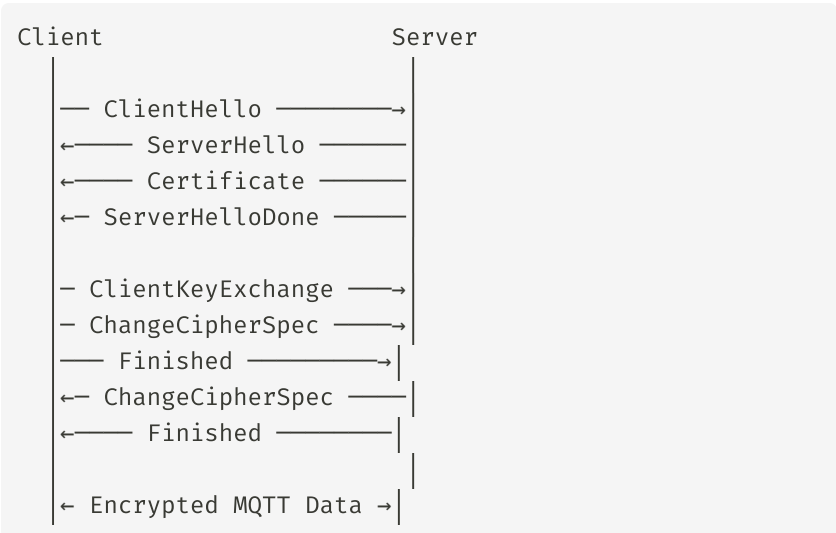
```
Application Data
├── MQTT Protocol Layer
├── TLS/SSL Encryption    ← Transport Security
├── Certificate Auth      ← Identity Verification
└── TCP/IP Network        ← Network Layer
```

Goal: Defense in depth with multiple security layers

TLS/SSL Overview

Transport Layer Security Architecture

TLS Handshake Process



Key Security Features

- **Encryption** - AES-256-GCM
- **Authentication** - RSA/ECDSA certificates
- **Integrity** - HMAC verification
- **Perfect Forward Secrecy** - ECDHE
- **Certificate validation** - Chain of trust

ESP32-P4 Hardware Support

- **Hardware crypto** acceleration
- **Secure boot** capabilities
- **Flash encryption** support
- **Random number** generation

Next: Certificate management and PKI infrastructure

X.509 Certificate Management

Certificate Components

Certificate Authority (CA)

- **Root CA** - Trust anchor
- **Intermediate CA** - Signing authority
- **Certificate chain** validation
- **CRL/OCSP** revocation checking

Certificate Types

- **Server certificate** - MQTT broker identity
- **Client certificate** - Device identity
- **CA certificate** - Trust verification

Certificate Generation

```
# Generate CA private key
openssl genrsa -out ca.key 4096

# Create CA certificate
openssl req -new -x509 -days 365 \
  -key ca.key -out ca.crt

# Generate client private key
openssl genrsa -out client.key 2048

# Create client certificate request
openssl req -new -key client.key \
  -out client.csr

# Sign client certificate
openssl x509 -req -days 365 \
  -in client.csr -CA ca.crt \
  -CAkey ca.key -out client.crt
```

Certificate Storage in ESP32-P4

Mutual Authentication (mTLS)

Bidirectional Certificate Verification

Traditional TLS (Server-only)

```
Client ──────────→ Server
       Trust
  "Is this the    ✓ Server cert
   real server?"    verified

❌ Server cannot verify client identity
❌ Vulnerable to unauthorized clients
```

Mutual TLS (Both directions)

```
Client ←──────────→ Server
    Mutual Trust

  ✓ Server cert     ✓ Client cert
    verified          verified

✅ Both parties authenticated
✅ Strong device identity
```

Implementation Benefits

- **Device identity** - Each device has unique certificate
- **Access control** - Only authorized devices connect
- **Non-repudiation** - Cryptographic proof of identity
- **Scalable security** - PKI infrastructure
- **Compliance ready** - Meets enterprise security standards

Secure MQTT Configuration

ESP32-P4 mTLS Implementation

Certificate Storage

```
// Embed certificates at compile time
extern const char ca_cert_pem_start[] asm("_binary_ca_cer
extern const char client_cert_pem_start[] asm("_binary_cl
extern const char client_key_pem_start[] asm("_binary_cli

// Or load from NVS encrypted storage
esp_err_t load_certificates_from_nvs(void) {
    size_t cert_len;
    esp_err_t err = nvs_get_blob(nvs_handle,
        "client_cert", client_cert, &cert_len);
    return err;
}
```

MQTT Client Configuration

```
esp_mqtt_client_config_t mqtt_cfg = {
    .broker = {
        .address.uri = "mqtts://broker.com:8883",
        .verification = {
            .certificate = ca_cert_pem_start,
            .skip_cert_common_name_check = false,
        }
    },
    .credentials = {
        .authentication = {
            .certificate = client_cert_pem_start,
            .key = client_key_pem_start,
        }
    }
};
```

Security Validation

- **Certificate chain** verification against CA

Certificate Validation & Pinning

Advanced Validation Techniques

Certificate Pinning

```
// Pin specific certificate or public key
const char* pinned_cert_sha256 =
    "A1:B2:C3:D4:E5:F6:07:08:09:0A:1B:2C:3D:4E:5F:60";

esp_err_t validate_server_cert(mbedtls_x509_crt* cert) {
    // Calculate certificate fingerprint
    unsigned char fingerprint[32];
    mbedtls_sha256_ret(cert→raw.p, cert→raw.len,
                       fingerprint, 0);

    // Compare with pinned value
    return memcmp(fingerprint, expected_sha256, 32);
}
```

Custom Validation Callback

```
int custom_cert_verify(void *data, mbedtls_x509_crt *crt,
                       int depth, uint32_t *flags) {
    // Custom validation logic
    if (depth == 0) {  // Server certificate
        // Check subject CN
        if (strstr(crt→subject.val, "mqtt.company.com")
            *flags |= MBEDTLS_X509_BADCERT_CN_MISMATCH;
            return -1;
        }
    }

    // Additional security checks
    return check_certificate_security_policy(crt);
}
```

Security Policies

- **Certificate expiration** monitoring
- **Weak cryptography** detection (RSA < 2048, etc.)

Secure Key Storage

NVS Encryption for Credentials

NVS Encryption Setup

```
// Enable NVS encryption in menuconfig
CONFIG_NVS_ENCRYPTION=y
CONFIG_NVS_SEC_KEY_PROTECTION_SCHEME=1

// Initialize encrypted NVS partition
esp_err_t init_secure_storage(void) {
    nvs_sec_cfg_t cfg = {};
    esp_err_t err = nvs_flash_read_security_cfg(
        NVS_DEFAULT_PART_NAME, &cfg);

    if (err == ESP_ERR_NVS_SEC_KEY_NOT_FOUND) {
        // Generate new encryption key
        err = nvs_flash_generate_keys(
            NVS_DEFAULT_PART_NAME, &cfg);
    }

    return nvs_flash_secure_init(&cfg);
}
```

Secure Certificate Storage

```
// Store encrypted certificate
esp_err_t store_client_cert(const char* cert_pem) {
    nvs_handle_t nvs_handle;
    esp_err_t err = nvs_open("certificates",
                             NVS_READWRITE, &nvs_handle);

    if (err == ESP_OK) {
        err = nvs_set_blob(nvs_handle, "client_cert",
                           cert_pem, strlen(cert_pem));
        nvs_commit(nvs_handle);
        nvs_close(nvs_handle);
    }

    return err;
}
```

Hardware Security Features

Complete Secure MQTT Implementation

End-to-End Security Example

Initialization

```c
void app_main(void) {
    // Initialize secure storage
    init_secure_storage();

    // Load certificates
    load_certificates_from_nvs();

    // Configure secure MQTT
    esp_mqtt_client_handle_t client =
        esp_mqtt_client_init(&secure_mqtt_cfg);

    // Register security event handler
    esp_mqtt_client_register_event(client,
        ESP_EVENT_ANY_ID, mqtt_security_handler, NULL);

    // Start secure connection
    esp_mqtt_client_start(client);
}
```

Security Event Handling

```c
static void mqtt_security_handler(void *args,
    esp_event_base_t base, int32_t event_id,
    void *event_data) {

    switch (event_id) {
    case MQTT_EVENT_CONNECTED:
        ESP_LOGI(TAG, "Secure MQTT connected");
        // Verify connection security
        verify_connection_security();
        break;

    case MQTT_EVENT_ERROR:
        ESP_LOGE(TAG, "MQTT security error");
        handle_security_error(event_data);
        break;
    }
}
```

Production Deployment Checklist

Part 3: Hands-On Assignment

Assignment: Secure MQTT with Mutual Authentication

Task 1: Certificate Setup

1. Generate CA and client certificates
2. Configure certificate pinning
3. Enable NVS encryption
4. Store certificates securely
5. Implement certificate validation

Task 2: Secure MQTT Connection

1. Configure mTLS MQTT client
2. Implement custom cert validation
3. Test mutual authentication
4. Verify encrypted communication
5. Handle security errors gracefully

Success Criteria

- ✅ Certificates generated and validated
- ✅ mTLS connection established successfully
- ✅ Mutual authentication working
- ✅ Encrypted MQTT communication verified
- ✅ Security monitoring and error handling
- ✅ **Production-ready** secure IoT system

Workshop Summary

What We Accomplished

Part 1: Basic Networking

- ✅ **ESP-NETIF** mastery
- ✅ **Ethernet RMII** implementation
- ✅ **WiFi Remote** configuration
- ✅ **Network events** handling
- ✅ **Multi-interface** management

Part 2: Custom Firmware

- ✅ **ESP-Hosted** deep dive
- ✅ **Custom firmware** development
- ✅ **UART/JTAG** programming
- ✅ **SDIO** communication
- ✅ **Advanced WiFi** features

Part 3: Secure MQTT

- ✅ **mTLS** implementation
- ✅ **Certificate** management
- ✅ **Mutual authentication**
- ✅ **Secure storage** (NVS)
- ✅ **Production security**

Key Achievements

- **Complete networking stack** from basic to secure
- **Hardware customization** capabilities unlocked
- **Enterprise-grade security** implementation
- **Production-ready** IoT communication system

Next Steps & Advanced Topics

Further Learning

- **Performance optimization** with DMA and PSRAM
- **OTA updates** with secure boot
- **WiFi mesh networking** with ESP32-C6
- **Edge AI** integration with networking
- **Fleet management** and device provisioning

Resources & Documentation

- ESP32-P4 Programming Guide
- ESP-Hosted Documentation
- mbedTLS Documentation
- MQTT Security Best Practices
- Espressif Security Guide

Advanced Security

- **Hardware security modules** (HSM)
- **Secure element** integration
- **Certificate lifecycle** management
- **Security monitoring** and logging
- **Compliance** frameworks (IEC 62443, etc.)

Community & Support

- Espressif Developer Forum
- GitHub Issues & Examples
- ESP32 Discord Community

Questions & Discussion

Thank you!

ESP32-P4 Networking Workshop

Basic Networking • Custom Firmware • Secure MQTT

Continue your learning journey:

Workshop materials: memory/assignments/

Reference examples: github.com/espressif