AWS IoT & Somnus MQTT Interface

This guide documents how the firmware connects to AWS IoT Core, publishes Somnus telemetry, and receives cloud-originated actions.

Compatibility & Requirements

- **ESP-IDF**: v5.0 or later
- **AWS IoT Device SDK**: v3.0.0+ (embedded via esp_aws_iot component)
- **TLS**: mbedTLS (provided by ESP-IDF)
- **MQTT**: AWS IoT Core with mutual TLS authentication

Quick Start

1. **Configure AWS IoT endpoint** in idf.py menuconfig:
   - Component config → Naphome AWS IoT → Set CONFIG_NAPHOME_AWS_IOT_ENDPOINT
   - Set CONFIG_NAPHOME_AWS_IOT_CLIENT_ID to your Thing name
2. **Provision certificates**:

```
python scripts/provision_aws_thing.py \
  --thing-name SOMNUS_ABCDEF123456 \
  --policy-name SomnusDevicePolicy \
  --output-dir components/aws_iot/certs/generated/SOMNUS_ABCDEF123456
```

3. **Start MQTT service** in your application:

```
somnus_mqtt_config_t cfg = { .action_cb = handle_action, .action_ctx = NULL };
somnus_mqtt_start(&cfg);
```

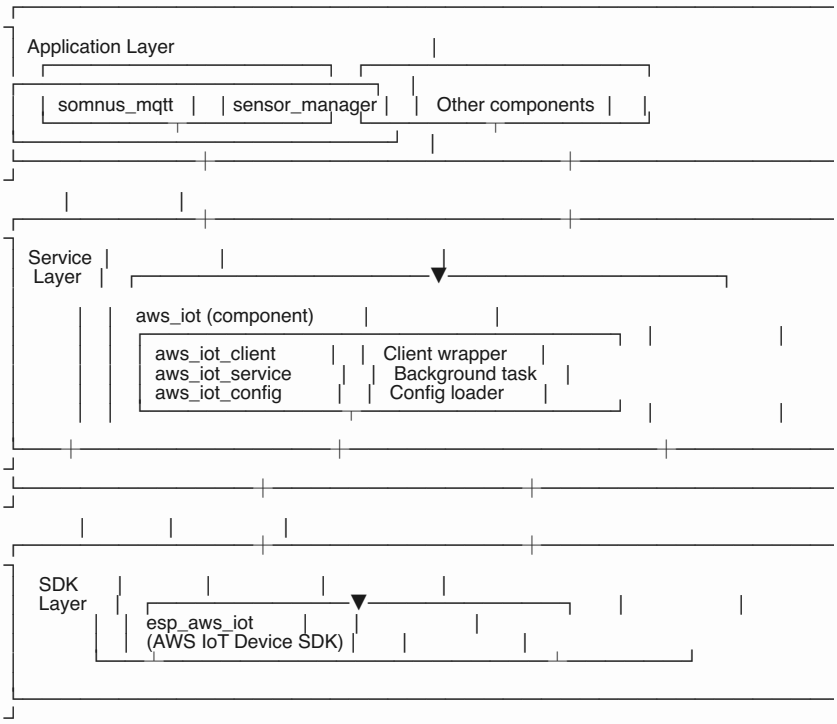
4. **Publish telemetry**:

```
somnus_mqtt_publish_telemetry(json_payload);
```

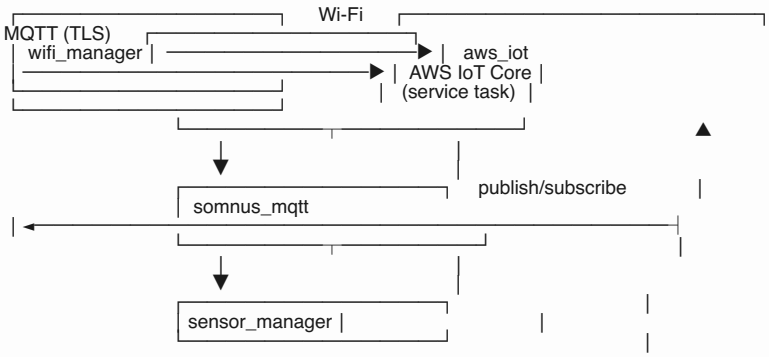
See Usage Examples and Testing & Troubleshooting for details.

Architecture Overview

The AWS IoT integration uses a three-layer architecture:



High-Level Flow



1. wifi_manager brings up the station interface and signals when an IP address is available.
2. aws_iot component's service layer (aws_iot_service) creates an aws_iot_client_t, connects using mutual TLS, and runs the MQTT yield loop inside a FreeRTOS task.
3. somnus_mqtt provides high-level helpers that publish telemetry/log payloads and dispatch action messages to the rest of the firmware.

Components

Layer 1: AWS IoT Device SDK (components/esp_aws_iot)

- Third-party AWS IoT Device SDK for Embedded C (v3.0.0+)
- Provides low-level MQTT client, TLS, and AWS IoT protocol handling
- Ported to ESP-IDF with FreeRTOS threading and mbedTLS

Layer 2: AWS IoT Wrapper (components/aws_iot)

The aws_iot component contains three modules: - **aws_iot_client** (aws_iot.h/c) - Thin wrapper over the AWS IoT Device SDK. Owns the MQTT client object and exposes

connect/publish/subscribe helpers. - **aws_iot_service** (aws_iot_service.h/c) - Background FreeRTOS task that waits for Wi-Fi, builds the MQTT configuration, maintains the connection, and optionally subscribes to topics. Handles reconnection and yield loop. - **aws_iot_config** (aws_iot_config.c) - Configuration loader that populates aws_iot_config_t from Kconfig settings and embedded certificates.

Layer 3: Application Layer

- **components/somnus_mqtt** — Somnus-specific glue that discovers certificates, composes topics via somnus_profile, and exposes log/telemetry APIs (somnus_mqtt_publish_telemetry, somnus_mqtt_publish_log, etc.).
- **components/somnus_profile** — shared constants and helpers for device IDs, topic names, and payload formatting so the MQTT surface mirrors the Somnus reference implementation.

Configuration

All defaults live in config/sdkconfig.defaults, but you can override them with idf.py menuconfig:

AWS IoT Core Settings

Navigate to **Component config → Naphome AWS IoT**:

Option	Default	Purpo
CONFIG_NAPHOME_AWS_IOT_ENDPOINT	""	AWS IoT C endpoint hostname xxxxxxxxxx- ats.iot.us-we 2.amazonav
CONFIG_NAPHOME_AWS_IOT_CLIENT_ID	""	Client ID / name. Typ matches th provisione name.
CONFIG_NAPHOME_AWS_IOT_PORT	8883	MQTT TLS for mutual authentica
CONFIG_NAPHOME_AWS_IOT_KEEPALIVE_SEC	60	Keep-alive interval for MQTT ses (10-1200 seconds). enforces n 1200s.
CONFIG_NAPHOME_AWS_IOT_CLEAN_SESSION	y	Request a MQTT ses connect (d previous s state).
CONFIG_NAPHOME_AWS_IOT_AUTO_RECONNECT	y	Enable au reconnecti the AWS Io after disconnec
CONFIG_NAPHOME_AWS_IOT_FAIL_ON_PLACEHOLDER_CERTS	y	Abort initia if placeholde certificates detected. Prevents accidental test creder
CONFIG_NAPHOME_AWS_IOT_SUBSCRIBE_TOPIC	""	Optional to auto-subsc connect. L empty to d
CONFIG_NAPHOME_AWS_IOT_SUBSCRIBE_QOS	0	QoS level used for th default subscriptio topic.
CONFIG_NAPHOME_AWS_IOT_YIELD_TIMEOUT_MS	200	Block time for the MQ yield loop. Controls h the service blocks wai incoming messages

Somnus MQTT Settings

Navigate to **Component config → Somnus MQTT Integration**:

Option	Default	Purpos
CONFIG_SOMNUS_MQTT_CERT_DISCOVERY	y	Enable runtime discovery the filesystem before falli embedded PEMs.
CONFIG_SOMNUS_MQTT_CERT_DIR	"/spiffs/Cert"	Directory path scanned fo credentials. Requires CONFIG_SOMNUS_MQTT_
CONFIG_SOMNUS_MQTT_SUBSCRIBE_QOS	1	QoS level (0 or 1) used w the Somnus command to

Provisioning Credentials

Use scripts/provision_aws_thing.py to create a Thing, generate certificates, and download the Amazon Root CA:

```
python scripts/provision_aws_thing.py \
  --thing-name SOMNUS_ABCDEF123456 \
  --policy-name SomnusDevicePolicy \
  --output-dir components/aws_iot/certs/generated/SOMNUS_ABCDEF123456
```

Copy the resulting .pem files onto the device filesystem (e.g. SPIFFS) under the directory configured via CONFIG_SOMNUS_MQTT_CERT_DIR. The runtime discovery logic looks for:

- <THING>-certificate.pem.crt
- <THING>-private.pem.key
- AmazonRootCA1.pem

If discovery fails, the component falls back to PEM blobs embedded through the component CMakeLists.

To streamline this for the Korvo Voice Assistant sample, run:

```
python scripts/provision_and_stage_somnus_cert.py --thing-name SOMNUS_ABCDEF123456
```

The wrapper invokes provision_aws_thing.py, stores the artifacts under components/aws_iot/certs/generated/, and copies them into samples/korvo_voice_assistant/spiffs/Cert/. The build then packs the updated SPIFFS image so /spiffs/Cert is ready immediately after flashing.

MQTT Topics

somnus_profile derives topics from the Somnus device ID (SOMNUS_ + station MAC):

Purpose	Pattern
Telemetry publish	device/telemetry/{DEVICE_ID}
Logs publish	device/receive/uat/{DEVICE_ID}
Command subscribe	device/somnus/{DEVICE_ID}

You can override the subscription topic or QoS via menuconfig or by supplying a custom handler in somnus_mqtt_start.

Payload Formats

Telemetry

The sensor manager creates JSON documents with a device ID, timestamp, and sensor blocks. Example:

```
{
  "deviceId": "SOMNUS_112233445566",
  "timestamp_ms": 1721165305123,
  "environment": {
    "temperature_c": 24.1,
    "humidity_pct": 48.3
  }
}
```

Call somnus_mqtt_publish_telemetry() with the serialized JSON.

Logs

Use somnus_mqtt_publish_log(level, message) to emit structured log events. The helper wraps somnus_profile_format_log_payload() and publishes to the log topic. Stages are inferred automatically (Onboarding vs AfterOnboarding) based on message content.

Actions

Incoming MQTT messages are parsed by somnus_mqtt. When a payload contains an

"Action" key or a routine list, the raw JSON string is passed to the optional action_cb configured in somnus_mqtt_start. Handlers should parse or dispatch the command to the appropriate subsystem.

Example action payload:

```
{
  "Action": "SetVolume",
  "Data": {
    "Volume": 50
  }
}
```

Usage Examples

Basic Initialization

```
#include "somnus_mqtt.h"
#include "sensor_manager.h"

// Action callback to handle incoming MQTT commands
static void handle_mqtt_action(const char *payload, void *ctx)
{
    ESP_LOGI("app", "Received MQTT action: %s", payload);

    // Parse JSON and dispatch to appropriate handler
    cJSON *json = cJSON_Parse(payload);
    if (json) {
        cJSON *action = cJSON_GetObjectItem(json, "Action");
        if (cJSON_IsString(action)) {
            const char *action_str = action->valuestring;
            if (strcmp(action_str, "SetVolume") == 0) {
                // Handle volume change
            }
        }
        cJSON_Delete(json);
    }
}

void app_main(void)
{
    // ... initialize Wi-Fi, NVS, etc ...

    // Start Somnus MQTT with action callback
    somnus_mqtt_config_t mqtt_cfg = {
        .action_cb = handle_mqtt_action,
        .action_ctx = NULL,
    };

    esp_err_t err = somnus_mqtt_start(&mqtt_cfg);
    if (err != ESP_OK) {
        ESP_LOGE("app", "Failed to start Somnus MQTT: %s", esp_err_to_name(err));
        return;
    }

    // Initialize sensor manager
    sensor_manager_init(NULL);
    sensor_manager_start();
}
```

Publishing Telemetry from Sensor Manager

```c
#include "somnus_mqtt.h"
#include "sensor_manager.h"
#include "cJSON.h"

// Observer callback called by sensor_manager when new samples are ready
static void sensor_observer(const char *sensor_name,
                            const cJSON *sensor_state,
                            void *user_ctx)
{
    // Build complete telemetry payload
    cJSON *root = cJSON_CreateObject();
    cJSON *device_id = cJSON_CreateString(somnus_mqtt_get_device_id());
    cJSON *timestamp = cJSON_CreateNumber(esp_timer_get_time() / 1000);

    cJSON_AddItemToObject(root, "deviceId", device_id);
    cJSON_AddItemToObject(root, "timestamp_ms", timestamp);

    // Add sensor data
    cJSON_AddItemToObject(root, sensor_name, cJSON_Duplicate(sensor_state, 1));

    char *json_str = cJSON_Print(root);
    if (json_str) {
        esp_err_t err = somnus_mqtt_publish_telemetry(json_str);
        if (err != ESP_OK) {
            ESP_LOGE("app", "Failed to publish telemetry: %s", esp_err_to_name(err));
        }
        free(json_str);
    }

    cJSON_Delete(root);
}

void app_main(void)
{
    // ... initialization ...

    // Set sensor observer to publish telemetry
    sensor_manager_set_observer(sensor_observer, NULL);
    sensor_manager_start();
}
```

Publishing Logs

```c
// Simple log message
somnus_mqtt_publish_log("INFO", "Device booted successfully");

// Log with automatic stage detection
somnus_mqtt_publish_log("WARN", "Wi-Fi connection failed during onboarding");
// Stage will be inferred as "Onboarding" based on message content

somnus_mqtt_publish_log("ERROR", "Sensor read timeout");
// Stage will be inferred as "AfterOnboarding"
```

Error Handling

```c
esp_err_t err = somnus_mqtt_start(NULL);
if (err == ESP_ERR_INVALID_STATE) {
    ESP_LOGW("app", "MQTT already started");
} else if (err == ESP_ERR_NOT_FOUND) {
    ESP_LOGE("app", "Certificate files not found");
    // Check CONFIG_SOMNUS_MQTT_CERT_DIR and certificate discovery
} else if (err != ESP_OK) {
    ESP_LOGE("app", "Failed to start MQTT: %s", esp_err_to_name(err));
    return;
}
```

Runtime Behaviour

Connection Lifecycle

1. **Initialization**: somnus_mqtt_start() is called, which internally calls

aws_iot_service_start()
2. **Wi-Fi Wait**: The aws_iot_service task waits for Wi-Fi to obtain an IP address (monitors IP_EVENT_STA_GOT_IP)
3. **Certificate Loading**:
   - If CONFIG_SOMNUS_MQTT_CERT_DISCOVERY=y, scans CONFIG_SOMNUS_MQTT_CERT_DIR for PEM files
   - Falls back to embedded certificates if discovery fails
   - If CONFIG_NAPHOME_AWS_IOT_FAIL_ON_PLACEHOLDER_CERTS=y and placeholder certs detected, initialization aborts
4. **MQTT Connect**: Establishes mutual TLS connection to AWS IoT Core
5. **Subscription**: Automatically subscribes to the Somnus command topic
6. **Yield Loop**: Service task continuously calls aws_iot_mqtt_yield() with CONFIG_NAPHOME_AWS_IOT_YIELD_TIMEOUT_MS timeout

Reconnection Handling

- When CONFIG_NAPHOME_AWS_IOT_AUTO_RECONNECT=y, the AWS IoT SDK automatically attempts reconnection on disconnect
- Subscriptions are automatically reinstated after successful reconnect
- Wi-Fi disconnection triggers the service to wait for IP before reconnecting
- The service task continues running and maintains connection state

Certificate Management

- Certificates discovered from filesystem are loaded into heap-allocated buffers
- Embedded certificates (from components/aws_iot/certs/) are referenced directly
- Certificate buffers are freed when somnus_mqtt_stop() is called
- The somnus_mqtt component owns certificate memory when discovery is enabled

Stopping the Service

Call somnus_mqtt_stop() to: - Stop the AWS IoT service task - Disconnect the MQTT client - Free certificate buffers (if owned by somnus_mqtt) - Clean up handlers and subscriptions

Integration with Sensor Manager

The sensor_manager can publish telemetry to both AWS IoT and Matter simultaneously:

```
// Set observer that publishes to AWS IoT
sensor_manager_set_observer(sensor_to_mqtt_observer, NULL);

// Matter bridge also registers as observer when enabled
// Both observers receive the same sensor snapshots
```

The observer pattern allows multiple consumers (AWS IoT, Matter, logging, etc.) to receive sensor updates without coupling.

Testing & Troubleshooting

Quick Start Checklist

1. **Wi-Fi Connectivity**: Verify Wi-Fi connects before AWS IoT attempts connection

   ```
   idf.py monitor | grep -i wifi
   # Look for: "Wi-Fi connected" or "IP_EVENT_STA_GOT_IP"
   ```

2. **Certificate Discovery**: Check if certificates are found

   ```
   idf.py monitor | grep -i "somnus_mqtt"
   # Success: "Certificate discovery succeeded"
   # Fallback: "Certificate discovery failed, falling back to embedded PEMs"
   # Error: "Failed to open cert dir" or "Incomplete certificate set"
   ```

3. **AWS IoT Connection**: Monitor connection attempts

```
idf.py monitor | grep -i "aws_iot"
# Look for: "Connected to AWS IoT as SOMNUS_XXXXXX"
# Errors: "aws_iot_client_connect failed" or TLS errors
```

Common Issues

Issue: “Certificate discovery failed”

Symptoms: Log shows “Certificate discovery failed, falling back to embedded PEMs”

Solutions: - Verify SPIFFS is mounted and contains /spiffs/Cert/ directory - Check CONFIG_SOMNUS_MQTT_CERT_DIR matches actual filesystem path - Ensure certificate files follow naming convention: <THING>-certificate.pem.crt, <THING>-private.pem.key, AmazonRootCA1.pem - Run provisioning script: python scripts/provision_and_stage_somnus_cert.py --thing-name SOMNUS_XXXXXX

Issue: “Failed to start AWS IoT service”

Symptoms: somnus_mqtt_start() returns error

Solutions: - Check CONFIG_NAPHOME_AWS_IOT_ENDPOINT is set to valid AWS IoT endpoint - Verify CONFIG_NAPHOME_AWS_IOT_CLIENT_ID matches provisioned Thing name - If CONFIG_NAPHOME_AWS_IOT_FAIL_ON_PLACEHOLDER_CERTS=y, ensure real certificates are provided - Check Wi-Fi is connected before calling somnus_mqtt_start()

Issue: MQTT publish fails or messages not received

Symptoms: No telemetry in AWS IoT console, or action callbacks not triggered

Solutions: - Verify device has publish/subscribe permissions in AWS IoT policy - Check topic names match: use somnus_mqtt_get_device_id() to confirm device ID - Test with AWS IoT MQTT test client: publish to device/somnus/{DEVICE_ID} and verify callback fires - Check CONFIG_NAPHOME_AWS_IOT_YIELD_TIMEOUT_MS is reasonable (default 200ms) - Monitor aws_iot_service logs for publish/subscribe errors

Issue: Connection drops frequently

Symptoms: Frequent reconnection logs

Solutions: - Increase CONFIG_NAPHOME_AWS_IOT_KEEPALIVE_SEC (max 1200s) - Check Wi-Fi signal strength and stability - Verify AWS IoT endpoint is reachable from network - Review CONFIG_NAPHOME_AWS_IOT_AUTO_RECONNECT is enabled

Testing with AWS IoT Console

1. **Get Device ID**: From boot logs or call somnus_mqtt_get_device_id() in code
2. **Subscribe to Telemetry**: In AWS IoT Console → Test, subscribe to device/telemetry/{DEVICE_ID}
3. **Publish Command**: Publish JSON to device/somnus/{DEVICE_ID}:

   ```
   {
     "Action": "Test",
     "Data": {}
   }
   ```

4. **Verify**: Check device logs show action callback was invoked

Debug Logging

Enable verbose logging:

```
// In menuconfig or sdkconfig
CONFIG_LOG_DEFAULT_LEVEL_DEBUG=y
CONFIG_LOG_MAXIMUM_LEVEL_DEBUG=y
```

Key log tags to monitor: - somnus_mqtt - Somnus MQTT layer - aws_iot_srv - AWS IoT service task - aws_iot - AWS IoT client wrapper