# ESP32 SIM7600 IoT Device Guide with Python OTA Server

This guide provides an in-depth explanation of the implementation, usage, and testing of an ESP32-based IoT device with a SIM7600 modem, integrated with a Python-based server for MQTT-driven Over-The-Air (OTA) firmware updates. The system supports secure communication, remote updates, and hardware feedback, leveraging the EMQX MQTT broker.

## Overview

* ESP32 Firmware

: Runs on an ESP32 microcontroller with a SIM7600 modem for cellular connectivity. It uses MQTT for communication, implements AES-256 encryption for security, supports OTA firmware updates, and provides visual feedback through a NeoPixel LED and I2C LCD.

* Python OTA Server

: A server-side script that manages the OTA update process by publishing firmware chunks to the ESP32 over MQTT, ensuring reliable delivery with progress tracking and retry mechanisms.

## Implementation Details

### ESP32 Firmware

#### Hardware Components

Microcontroller

: ESP32, serving as the core processing unit.

SIM7600 Modem

: Provides cellular connectivity via GPRS.

Pins: TX (16), RX (17), Power (21)

NeoPixel LED

: Single RGB LED for status indication.

Pin: 48

I2C LCD

: 16x2 character display for local feedback.

Pins: SDA (35), SCL (36), I2C Address: 0x27

Status LED

: General-purpose LED for toggling on message receipt.

Pin: 13

#### Software Libraries

* TinyGsm

: Handles modem communication and AT commands.

* Adafruit\_NeoPixel

: Controls the RGB LED.

* LCD\_I2C

: Manages the LCD display.

* mbedtls

: Implements AES-256 encryption/decryption.

* Update

: Facilitates OTA firmware updates.

* HardwareSerial

: Manages serial communication with the modem.

* Wire

: Supports I2C communication for the LCD.

#### State Machine

The firmware uses a robust state machine with 14 states to manage the connection lifecycle:  
-

Initialization

:

STATE\_INIT\_MODEM

,

STATE\_WAIT\_NETWORK

,

STATE\_CONNECT\_GPRS

-

Setup

:

STATE\_UPLOAD\_CERTIFICATE

,

STATE\_SETUP\_SSL

,

STATE\_SETUP\_MQTT

,

STATE\_CONNECT\_MQTT

,

STATE\_SUBSCRIBE\_MQTT

-

Operation

:

STATE\_RUNNING

-

Error Handling

:

STATE\_ERROR

,

STATE\_STOPPED

-

Recovery

:

STATE\_RECOVER\_NETWORK

,

STATE\_RECOVER\_GPRS

,

STATE\_RECOVER\_MQTT

- Features retry logic with configurable parameters:  
 -

MAX\_RETRIES

: 3 attempts  
 -

RETRY\_DELAY

: 2000ms between retries

#### MQTT Communication

* Broker

:

u008dd8e.ala.dedicated.aws.emqxcloud.com:8883

* Topics

:

* esp32\_status

: Device publishes status updates and OTA progress.

* server\_cmd

: Receives encrypted commands from the server.

* firmware/update

: Receives OTA commands and firmware chunks.

* Security

: Uses SSL/TLS with a certificate (

iot\_inverter2.pem

) uploaded to the modem.

* Client ID

: Dynamically generated as

ESP32\_SIM7600\_<millis>

.

#### Security Implementation

* Encryption

: AES-256 in CBC mode with PKCS7 padding.

* Key: 32-byte static value (see Testing Notes).
* IV: 16-byte static value (see Testing Notes).
* Encoding

: Base64 applied to encrypted payloads for safe MQTT transmission.

* Message Handling

: Decrypts incoming messages, echoes them back (plain and prefixed), and updates hardware indicators.

#### OTA Updates

* Mechanism

: Chunk-based updates with configurable sizes:

* OTA\_CHUNK\_SIZE

: 512 bytes (total chunk size including header)

* OTA\_MAX\_DATA\_SIZE

: 508 bytes (data portion, excluding 4-byte chunk number)

* Process

:

* Starts with

OTA:BEGIN:<size>

command.

* Receives Base64-encoded chunks with 4-byte chunk numbers.
* Tracks progress, requests missing chunks via

OTA:REQUEST:<num>

.

* Completes with

OTA:END

and restarts on success.

* Error Handling

: Aborts on network loss, invalid chunks, or write failures, reporting via

OTA:ERROR:<reason>

.

### Python OTA Server

#### Purpose

The Python server script automates the OTA update process by interacting with the ESP32 over MQTT. It reads a firmware binary, splits it into chunks, and ensures reliable delivery with confirmation checks.

#### Architecture

* MQTT Client

: Uses the

paho-mqtt

library to connect to the EMQX broker with SSL/TLS.

* Configuration

:

* Broker:

u008dd8e.ala.dedicated.aws.emqxcloud.com:8883

* Credentials: Username

ESP32

, Password

12345

* Topics: Publishes to

firmware/update

, subscribes to

esp32\_status

* Chunk Size: 508 bytes (matches

OTA\_MAX\_DATA\_SIZE

)

* Timeout: 20 seconds per response
* Workflow

:

* Connects to the broker and subscribes to device status updates.
* Validates the firmware file and calculates total size.
* Initiates OTA with

OTA:BEGIN:<size>

.

* Sends chunks with retries (up to 10 attempts per chunk).
* Finalizes with

OTA:END

and awaits completion confirmation.

#### OTA Process Details

* Initialization

: Sends the total firmware size to prepare the ESP32.

* Chunk Transmission

:

* Reads the firmware file in 508-byte segments.
* Prepends a 4-byte chunk number (big-endian).
* Base64-encodes each chunk for MQTT compatibility.
* Publishes with QoS 1 for guaranteed delivery.
* Progress Tracking

: Listens for

OTA:PROGRESS

messages to confirm chunk receipt.

* Retry Logic

: Resends chunks if no response within 20 seconds, up to 10 retries.

* Completion

: Sends

OTA:END

and waits for

OTA:SUCCESS

or an error message.

#### Error Handling

* Checks file existence and size consistency.
* Times out and reports failures if responses are delayed beyond 20 seconds.
* Logs detailed status messages for debugging.

## Usage

### Prerequisites

* ESP32 Hardware

:

* SIM7600, NeoPixel, and LCD connected as per pin definitions.
* SIM card with data plan (APN: "internet").
* Python Environment

:

* Python 3.x installed.
* paho-mqtt

library (

pip install paho-mqtt

).

* EMQX Broker

: Use the provided broker or configure your own.

* Certificates

: Define

root\_ca

in

certificates.h

for ESP32 SSL.

### Configuration

#### ESP32 Firmware

* Network

:

apn

,

gprsUser

,

gprsPass

* MQTT

:

mqtt\_server

,

mqtt\_user

,

mqtt\_pass

,

mqtt\_port

* Topics

:

mqtt\_topic\_send

(

esp32\_status

),

mqtt\_topic\_recv

(

server\_cmd

),

mqtt\_topic\_firmware

(

firmware/update

)

* OTA

:

OTA\_CHUNK\_SIZE

(512),

OTA\_MAX\_DATA\_SIZE

(508)

#### Python OTA Server

* Firmware File

: Specify path (e.g.,

OTA\_test.bin

).

* MQTT Settings

: Matches ESP32 (broker, port, credentials, topics).

* Chunk Size

: Set to 508 to align with

OTA\_MAX\_DATA\_SIZE

.

* Timeout

: Default 20 seconds, adjustable for network conditions.

### Running

#### ESP32 Device

Upload Firmware

: Flash the code to the ESP32 using an IDE (e.g., Arduino IDE).

Power On

: The LCD displays "Connecting..." as the device initializes.

Operation

:

Initializes the modem and retrieves IMEI.

Connects to GPRS and uploads the SSL certificate.

Establishes an MQTT connection and subscribes to

server\_cmd

and

firmware/update

.

Enters

STATE\_RUNNING

, ready for commands and updates.

Indicators

:

LCD: Shows connection status or received messages.

NeoPixel: Configurable RGB feedback (currently unconfigured in code).

LED: Toggles on message receipt.

#### Python OTA Server

Prepare Firmware

: Place the binary file (e.g.,

OTA\_test.bin

) in the script directory.

Run Script

: Execute

python ota\_server.py

in a terminal.

Process

:

Connects to the MQTT broker and subscribes to

esp32\_status

.

Sends the firmware update, monitoring progress via console output.

Disconnects cleanly upon completion or failure.

## Testing with MQTTX and EMQX

### Setup

#### EMQX Platform

* Broker

:

u008dd8e.ala.dedicated.aws.emqxcloud.com:8883

* Credentials

: Username

ESP32

, Password

12345

* SSL/TLS

: Enabled with appropriate CA certificate.

#### MQTTX

* Installation

: Download from

mqttx.app

and install.

* Configuration

:

* Name: "ESP32 Test"
* Host:

u008dd8e.ala.dedicated.aws.emqxcloud.com

* Port: 8883
* Username:

ESP32

* Password:

12345

* SSL/TLS: Enabled
* Client ID: Unique (e.g., "MQTTX\_Test")

### Test Scenarios

#### 1. Basic Messaging (MQTTX)

* Objective

: Verify MQTT communication and encryption functionality.

* Steps

:

* Connect MQTTX to the broker.
* Subscribe to

esp32\_status

to monitor device responses.

* Publish a message to

server\_cmd

:

* Payload: Base64-encoded AES-256 encrypted message (see Notes for key/IV).
* Example: Encrypt "Test Message" using an online tool or script, then Base64-encode.

Observe device behavior and MQTTX output.

Expected Results

:

LCD displays "MQTT Msg: Test Message" (first 16 characters).

Status LED toggles state.

MQTTX receives two messages on

esp32\_status

:

* Plaintext: "Test Message"
* Encrypted: Base64-encoded "ESP32\_Test Message"

Notes

: Ensure encryption matches the device’s AES key and IV.

#### 2. OTA Update (Python Server)

* Objective

: Validate the OTA update process using the Python server.

* Steps

:

* Create a small test firmware file (e.g., 2048 bytes, named

OTA\_test.bin

).

* Example: Use

dd if=/dev/zero of=OTA\_test.bin bs=2048 count=1

on Unix-like systems.

Ensure ESP32 is running and connected to the broker.

Run the Python script in a terminal.

Monitor:

* ESP32 Serial Monitor (115200 baud) for state transitions and OTA logs.
* Python console for send/receive progress.

Expected Results

:

Python Output

:

* Connects to broker, subscribes to

esp32\_status

.

* Sends

OTA:BEGIN:2048

, followed by 5 chunks (4 full 508-byte, 1 partial).

* Confirms each chunk with

OTA:PROGRESS

responses.

* Sends

OTA:END

and receives

OTA:SUCCESS

.

ESP32 Behavior

:

* Transitions to OTA mode, processes chunks.
* LCD shows "OTA Complete" before restarting.
* Serial logs detail chunk receipt and completion.

Notes

: If chunks fail, the server retries up to 10 times; ESP32 may request missing chunks.

#### 3. Connection Recovery (MQTTX)

* Objective

: Test the device’s ability to recover from network disruptions.

* Steps

:

* Connect MQTTX and subscribe to

esp32\_status

.

* While device is in

STATE\_RUNNING

, briefly disconnect SIM7600 power (e.g., pull pin 21).

* Monitor Serial Monitor and MQTTX for recovery.
* Expected Results

:

* Serial logs show "Network lost" and transition to

STATE\_RECOVER\_NETWORK

.

* Device resets modem, reconnects GPRS, and restores MQTT subscription.
* MQTTX sees resumed status messages on

esp32\_status

.

* Notes

: Recovery may take several seconds due to modem reset delays.

### Debugging

* ESP32

:

* Enable

DUMP\_AT\_COMMANDS

in the code to log modem AT commands.

* Use Serial Monitor (115200 baud) for detailed state and error messages.
* Check LCD for real-time status updates.
* Python Server

:

* Console logs include connection status, publish events, and received messages.
* Look for timeout or retry messages indicating delivery issues.

### Notes

* AES Key and IV for Testing

:

* Key (32 bytes)

:  
 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,  
 0x38, 0x39, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46,  
 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,  
 0x38, 0x39, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46

* ASCII: "0123456789ABCDEF0123456789ABCDEF"
* IV (16 bytes)

:  
 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,  
 0x38, 0x39, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46

* ASCII: "0123456789ABCDEF"
* Usage

:

* Encrypt test messages using AES-256 CBC mode with this key and IV.
* Tools like

CyberChef

can perform encryption:

* Input: Message (e.g., "Test Message")
* Steps: AES Encrypt (CBC, key, IV), then Base64 Encode
* Output: Publish to

server\_cmd

* Example: "Hello" might yield

pXgM7i9m8gQ=

(exact output varies with padding).

* OTA Compatibility

:

* Python chunk size (508) must match

OTA\_MAX\_DATA\_SIZE

.

* Adjust both values together if modifying chunk size (e.g., to 256 or 1024).
* Timeouts

:

* Python uses 20s per response; increase if cellular latency exceeds this.
* MQTTX may need timeout adjustments for slow networks.
* EMQX

:

* Ensure the SSL certificate in

root\_ca

matches the broker’s certificate.

## Additional Tips

* Security

: In production, replace static AES key/IV with device-specific, securely generated values stored in ESP32 NVS or an external secure element.

* Enhancements

:

* Add Python logic to handle

OTA:REQUEST

messages for retransmitting specific chunks.

* Implement version checking in both firmware and server to prevent redundant updates.
* Testing

:

* Start with a small firmware file (e.g., 2KB) to verify OTA workflow.
* Use Wireshark or MQTTX logs to inspect message flow if issues arise.