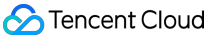
Internet of Things Hub Developer Manual

Copyright Notice

Trademark Notice



Service Notice

This document provides an overview of the as-is details of Tencent Cloud's products and services in their entirety or part. The descriptions of certain products and services may be subject to adjustments from time to time.

The commercial contract concluded by you and Tencent Cloud will provide the specific types of Tencent Cloud products and services you purchase and the service standards. Unless otherwise agreed upon by both parties, Tencent Cloud does not make any explicit or implied commitments or warranties regarding the content of this document.

Contact Us

We are committed to providing personalized pre-sales consultation and technical after-sale support. Don't hesitate to contact us at 4009100100 or 95716 for any inquiries or concerns.

Contents

Developer Manual
Feature Components

Last updated: 2026-03-20 15:16:36

1. SDK

For more details, please refer to the SDK documentation . Currently, the device SDK supports Linux and Android platforms, and can be ported to different hardware platforms.

2. Device Access

Devices can connect to the Tencent IoT Hub platform through the SDK:

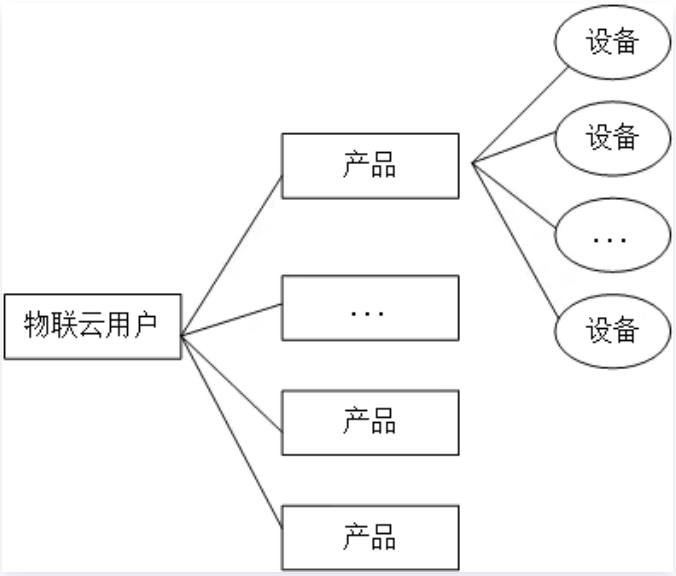
- The application layer is based on MQTT and CoAP protocols.
- The transport layer is based on TCP and UDP protocols, and it introduces secure network transfer protocols (TLS and DTLS) to achieve two-way authentication and data encryption transmission between the client and server.
- The SDK supports RTOS portability and cross-platform porting. The framework is detached from the hardware abstraction layer, enabling quick and easy access to IoT Hub from different platforms.

The device SDK supports two authentication methods for TLS (corresponding to MQTT) and DTLS (corresponding to CoAP) asymmetric and symmetric encryption to secure device communication:

- Asymmetric Encryption
  High security level, based on certificates and asymmetric encryption algorithms, suitable for devices with high hardware specifications and low sensitivity to power consumption. It depends on device certificates, private keys, and root certificates. When creating devices in IoT Hub, relevant information will be returned.
- Symmetric Encryption
  Normal security level, based on keys and symmetric encryption algorithms, suitable for resource-constrained devices sensitive to power consumption. It depends on the device psKey. When creating devices in IoT Hub, relevant information will be returned.

In addition to device SDK access, Tencent IoT Hub also provides HTTP access with a low protocol threshold, suitable for low power consumption and short connection data reporting scenarios.

3. Device Management



- Under one Tencent Cloud account, you can create up to 2000 products, and up to 1 million devices for each product. A device can only belong to one product. Product names and device names must be unique within the same cloud account.
- Provides the enable/disable feature for devices. Once a device is disabled, it will not be able to access the IoT Hub platform and will be unable to perform operations related to the device. However, information associated with the device is retained and device-related information can still be queried.

4. Topic Management

In Tencent Internet of Things Hub, the topics that devices can publish and subscribe to are strictly managed. All devices under a product have the same topic class permissions, which by default include:

Topic	Description
${productId}/${deviceName}/event	Publish permissions, used for devices to report data.
${productId}/${deviceName}/control	Subscribe permissions, used for devices to receive data sent from the backend.

The above "$" symbol contains the productId and deviceName, which will be mapped to the specific product Id and device name for the created device.

> For example, for a product named "pro" (assuming the product Id is "pro_id") with 2 devices (assuming the device names are "dev_1" and "dev_2"), the topics that "dev_1" can publish include pro_id/dev_1/event, and the topics it can subscribe to include pro_id/dev_1/control, but it cannot publish to pro_id/dev_2/event or subscribe to pro_id/dev_2/control.

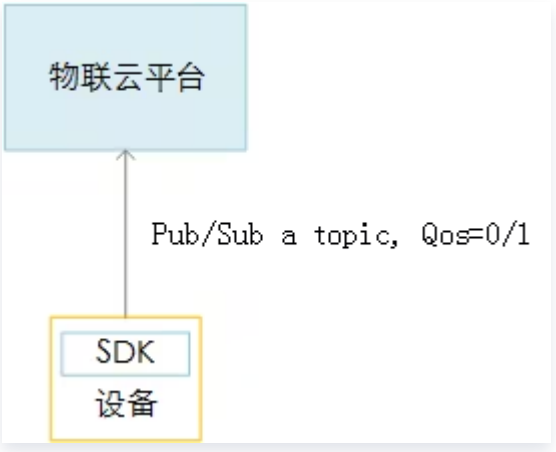
Users can edit and modify topic permissions and add or delete product topic class permissions through the console.
To facilitate subscription to multiple topics by the device SDK, devices can use wildcards to represent multiple matching topics when subscribing and unsubscribing:

Wildcard	Description
#	This wildcard can only appear at the end of the topic and represents the current and all sub-level topics. For example, the wildcard topic pro_id/dev_1/# can represent both pro_id/dev_1/event and pro_id/dev_1/event/subeventA.
+	Represents all topics at the same level, can only appear after deviceName. For example, wildcard topic pro_id/dev_1/event/+, can represent pro_id/dev_1/event/subeventA or pro_id/dev_1/event/subeventB, but not pro_id/dev_1/event/subeventA/close. It can appear multiple times, such as pro_id/dev_1/event/+/subeventA/+.

Wildcards must be a complete level, ${productId}/${deviceName}/e# and ${productId}/${deviceName}/e+ are invalid formats.
Tencent Internet of Things Hub-defined system topics ($shadow, $ota, $sys) do not support wildcards.

- When adding subscriptions, wildcards work as: subscribe to the topics that match the wildcard topic and have subscription rights under the product. Even if the matched topic list is empty, it still returns success.
- When unsubscribing, wildcards work as: unsubscribe from the subscribed topics that match the wildcard topic. Even if the matched topic list is empty, it still returns success. ${productId}/${deviceName}/# clears all user topic subscriptions.

5. Message Management



For MQTT data transfer, Tencent Internet of Things Hub supports QoS=0 or 1, but does not support QoS=2. Based on the MQTT protocol, device messages support offline storage.

- QoS=0, at most send to the device once
  For scenarios with general data transfer reliability requirements, choose this QoS when publishing and subscribing.
- QoS=1, ensure the device receives at least once
  For scenarios with high data transfer reliability requirements, choose this QoS when publishing and subscribing.

Other parameters are as follows:

Parameters	Description
Topic name length	No longer than 64 bytes.
MQTT Protocol Packet Size	No more than 16K bytes.
Message Storage Duration for QoS=1 (receiver offline or unable to receive online)	24 hours.
Number of unacknowledged QoS=1 messages	No more than 150 messages.

6. Device Shadow

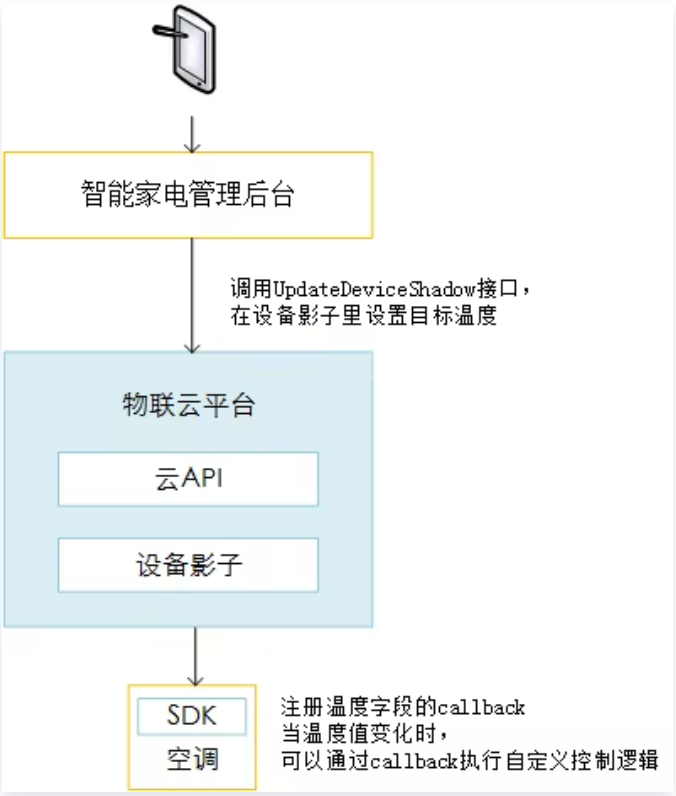
Device shadow is essentially a copy of device data in JSON format cached on the server and is mainly used to save:

- Current device configuration
- Current device status

As a medium, device shadow can effectively implement two-way data sync between device and user application:

- For device configuration, the user application does not need to directly modify the device; instead, it can modify the device shadow on the server, which will sync modifications to the device. In this way, if the device is offline at the time of modification, it will receive the latest configuration from the shadow once going back online.
- For device status, the device reports the status to the device shadow, and when users initiate queries, they can simply query the shadow. This can effectively reduce the network interactions between the device and the server, especially for low-power devices.

The following is an application example of Device Shadow in the "Quick Start":



> **Note:**
> The use cases for Device Shadow and Device Messages are different. From an implementation mechanism perspective, the server-side Device Shadow always saves the last data, while multiple messages that arrive sequentially do not overwrite each other.

- For scenarios where devices report data, the device shadow is more suitable for reporting some device information (such as device energy consumption), while device messages are more suitable for reporting data collected by the device (such as measured temperature)
- In scenarios where devices receive data, the device shadow is more suitable for notifying the device to update configurations (such as changing the target operating temperature), while device messages are more suitable for real-time control of the device (such as instructing the device to turn left 45 degrees)

For details, please refer to Device Shadow Details.

7. Rule Engine

Based on the rule engine, users can configure rules to achieve the following operations:

- **Syntax Rules**
  Supports SQL-like syntax and basic semantic operations. Through simple syntax, the content of Device Messages can be parsed, filtered, extracted, re-integrated, and then

forwarded to backend services, seamlessly connecting to various Tencent Cloud's backend storage components, function compute, big data analysis suite, etc.

- **Device Interconnection**
  To achieve data isolation, devices can only publish and subscribe to their own Topic Messages (please refer to Topic Management). To achieve interoperability, the repub feature of the rule engine is required.
- **Device–User Server Interconnection**
  The rule provides a simple forward feature, which can copy messages to the user server via HTTP Request, enabling quick interoperability between Device Messages and user services.
- **Device–Cloud Interconnection**
  For scenarios where users require further processing of device data (such as persistent storage and big data analysis), Tencent Cloud offers corresponding products (such as TencentDB and Big Data Analysis Suite).
  For details, please refer to Rule Engine Details.

8. Console

The console provides a visual management interface, supporting features such as product management, device management, Topic management, and rule engine configuration. You can experience it at the Internet of Things Hub Console.

9. TencentCloud API

For IoT scenarios, backend interfaces for quick and batch device operations are provided. Currently, Python, PHP, Java, Go, NodeJS, and .Net toolkits are supported. Tencent Internet of Things Hub currently offers APIs related to products, devices, tasks, messages, rule engine, and device shadow. For details, please refer to TencentCloud API.

10. Updating Firmware

When firmware has security risks or functional problems, IoT Hub servers can perform OTA updates to eliminate dangers and reduce security risks.

11. Collaboration Management

Internet of Things Hub supports secure access, use, and management of cloud account resources through CAM. Isolation and collaboration of Internet of Things Hub resources are implemented through identity and policy management of sub-accounts and collaborators.

Signature Method

Last updated: 2026-05-09 15:48:14

Overview

When the device initiates an `HTTP/HTTPS` request to the platform, the request message must include signature information (X-TC-Signature) to verify the identity of the requester.

Signing Steps

Sample device request message:

```
curl -X POST https://ap-
guangzhou.gateway.tencentdevices.com/device/register \
-H "Content-Type: application/json; charset=utf-8" \
-H "X-TC-Algorithm: hmacsha256" \
-H "X-TC-Timestamp: 155****065" \
-H "X-TC-Nonce: 5456" \
-H "X-TC-Signature:
2230eefd229f582d8b1b891af****b91597240707d778ab3738f756258d7652c" \
-d '{"ProductId":"ASJ****GX","DeviceName":"xyz"}'
```

1. Concatenate the string to sign

```
StringToSign =
    HTTPRequestMethod + \n +
    CanonicalHost + \n +
    CanonicalURI + \n +
    CanonicalQueryString + \n +
    Algorithm + \n +
    RequestTimestamp + \n +
    Nonce + \n +
    HashedCanonicalRequest
```

Parameter Name	Description
HTTPRequestMethod	HTTP request method. POST is supported.

CanonicalHost	Host address of the HTTP request.
CanonicalURI	URI of the HTTP request; for example, the URI of `https://ap-guangzhou.gateway.tencentdevices.com/device/register` is `/device/register`.
CanonicalQueryString	Query string in the URL of the initiated HTTP request, which is always an empty string `""` for POST requests.
Algorithm	Signature algorithm. Currently, HMACSHA256 and HMACSHA1 are supported.
RequestTimestamp	Timestamp in seconds.
Nonce	A random number.
HashedCanonicalRequest	Hash value of the request body, which is calculated by SHA256 hashing the HTTP request body, performing hexadecimal encoding, and then converting the encoded string to lowercase letters.

According to the above rules, the canonical signature string obtained in the sample is as follows:

```
POST
ap-guangzhou.gateway.tencentdevices.com
/device/register

hmacsha256
155****065
5456
35e9c5b0e3ae67532d3c9f17ead6c902226****b1ff7f6e89887f1398934f064
```

2. Calculate the signature

- The pseudo code for using key signatures, including product-level keys and device-level keys, is as follows:

```
Signature = Base64_Encode(HMAC_SHA256(SignSecret, StringToSign))
```

Parameter Name	Description
SignSecret	Signature key. `ProductSecret` is used for dynamic registration, and `psk` is used for devices to publish messages or report logs.
StringToSign	String to sign.

- The pseudo code for using certificate signatures is as follows:

```
Signature = Base64_Encode(RSA_SHA256(PrivateKey, StringToSign))
```

Parameter Name	Description
PrivateKey	Certificate private key. Device X.509 private key certificate is used for devices to publish messages or report logs.
StringToSign	String to sign.

3. Assemble the request message

Based on the signature string obtained above, the final complete request is as follows:

```
POST https://ap-guangzhou.gateway.tencentdevices.com/devregister
Content-Type: application/json
Host: ap-guangzhou.gateway.tencentdevices.com
X-TC-Algorithm: hmacsha256
X-TC-Timestamp: 155****065
X-TC-Nonce: 5456
X-TC-Signature:
2230eefd229f582d8b1b891af71****1597240707d778ab3738f756258d7652c


{"ProductId":"ASJ****GX","DeviceName":"xyz"}
```

Sample Code

C language sample code

After downloading the project, compile it in a Linux environment with the following command:

```
./cmake_build.sh
```

After successful compilation, execute the signature example with the following command:

```
./out/bin/qcloud-dynreg-sign product_id product_secretkey device_name
```

Python3 sample code

```
import hashlib
import random
import time
import hmac
import base64

if __name__ == '__main__':
    sign_format = '%s\n%s\n%s\n%s\n%s\n%d\n%d\n%s'
    url_format = '%s://ap-
guangzhou.gateway.tencentdevices.com/device/register'
    request_format = "{\"ProductId\":\"%s\",\"DeviceName\":\"%s\"}"
device_name = 'dev***'
product_id = 'JCZ****KXS'
product_secret = 'X42fPqw********94cY5sQ1Y'

request_text = request_format % (product_id, device_name)
request_hash = hashlib.sha256(request_text.encode("utf-8")).hexdigest()

nonce = random.randrange(2147483647)
timestamp = int(time.time())
sign_content = sign_format % (
    "POST", "ap-guangzhou.gateway.tencentdevices.com",
    "/device/register", "", "hmacsha256", timestamp,
    nonce, request_hash)
print("\nsign_content: \n" + sign_content)

sign_base64 = base64.b64encode(hmac.new(product_secret.encode("utf-8"),
                sign_content.encode("utf-8"), hashlib.sha256).digest())

print("sign_base64: " + str(sign_base64))
```

Java Sample Code

```
package com.tencent.iot.hub.device.java.core.sign;

import org.json.JSONException;
import org.json.JSONObject;
import org.junit.Test;

import java.nio.charset.Charset;
import java.security.InvalidKeyException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Base64;

import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;

import static junit.framework.TestCase.assertTrue;
import static junit.framework.TestCase.fail;

public class SignForHttpTest {

private static final String HMAC_ALGO = "hmacsha256";

@Test
public void testSign() {
    try {
        JSONObject yourPayload = new JSONObject();
        String signature = getSignature("YourProductId",
"YourDeviceName",
                "YourDevicePsk", "YourTopicName", yourPayload, 0);
        System.out.println(signature);
        assertTrue(true);
    } catch (Exception e) {
        e.printStackTrace();
        fail();
    }
```

```
}

public static String getSignature(String productID, String deviceName,
String devicePsk,
                                  String topicName, JSONObject payload,
Integer qos) {
    int randNum = (int) (Math.random() * ((1 << 31) - 1));
    int timestamp = (int) (System.currentTimeMillis() / 1000);
    final JSONObject obj = new JSONObject();
    try {
        obj.put("ProductId", productID);
        obj.put("DeviceName", deviceName);
        obj.put("TopicName", topicName);
        obj.put("Payload", payload.toString());
        obj.put("Qos", qos);
    } catch (JSONException e) {
        e.printStackTrace();
    }
    String originRequest = obj.toString();
    String hashedRequest = "";
    try {
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        byte[] encodedhash =
digest.digest(originRequest.getBytes(Charset.forName("UTF-8")));
        hashedRequest = HMACSHA256.bytesToHexString(encodedhash);
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }

    @SuppressWarnings("DefaultLocale")
    String signSourceStr =
String.format("%s\n%s\n%s\n%s\n%s\n%d\n%d\n%s",
            "POST",
            "ap-guangzhou.gateway.tencentdevices.com",
            "/device/publish",
            "",
            HMAC_ALGO,
            timestamp,
            randNum,
            hashedRequest
```

```
        );

        return HMACSHA256.getSignature(signSourceStr.getBytes(),
devicePsk.getBytes());
    }

    public static class HMACSHA256 {

        /**
         * Generate the signature data
         *
         * @param data The data to be encrypted
         * @param key The key used for encryption
         * @return The generated hexadecimal string
         */
        public static String getSignature(byte[] data, byte[] key)  {
            Mac mac;
            SecretKeySpec signKey = new SecretKeySpec(key, HMAC_ALGO);
            try {
                mac = Mac.getInstance(HMAC_ALGO);
                mac.init(signKey);
                byte[] rawHmac = mac.doFinal(data);
                return Base64.getEncoder().encodeToString(rawHmac);
            } catch (InvalidKeyException | NoSuchAlgorithmException e) {
                e.printStackTrace();
            }
            return null;
        }

        /**
         * Convert the `byte[]` array into a hexadecimal string
         *
         * @param bytes The byte array to be converted
         * @return Converted result
         */
        private static String bytesToHexString(byte[] bytes) {
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < bytes.length; i++) {
                String hex = Integer.toHexString(0xFF & bytes[i]);
                if (hex.length() == 1) {
```

```
                sb.append('0');
            }
            sb.append(hex);
        }
        return sb.toString();
    }
}

}
```

Device Authentication Overview

Last updated: 2025-03-19 14:52:28

The IoT Hub platform assigns a unique ProductID to each created product. Users can customize the Devicename to label the equipment. The product label, equipment label, and equipment certificate/key are used to verify the legality of the equipment. When creating a product, users need to select the equipment authentication method. During equipment access, users need to report product, equipment information, and key information according to the specified method. Only after authentication can the equipment connect to the IoT Hub platform. Due to different user equipment resources and security level requirements, the platform provides various authentication schemes to meet different usage scenarios.
IoT Hub provides the following three authentication schemes:

- Certificate authentication (device-level): it assigns a certificate + private key to each device and uses asymmetric encryption to authenticate the access. You need to burn different configuration information for each device.
- Key authentication (device-level): it assigns a device key to each device and uses symmetric encryption to authenticate the access. You need to burn different configuration information for each device.
- Dynamic registration authentication (product-level): it assigns a unified key to all devices under the same product, and a device gets a device certificate/key through a registration request for authentication. You can burn the same configuration information for the same batch of devices.

The three schemes have their own pros and cons in terms of ease of use, security, and device resource requirement. You can comprehensively evaluate them and choose the most appropriate one according to your own business scenarios. They are as compared below:

Feature	Certificate	Key Authentication	Dynamic Registration Authentication
Burned device information	ProductId, Devicename, device certificate, device private key	ProductId, Devicename, device key	ProductId、Devicename、ProductSecret
Whether device creation is required	Mandatory	Mandatory	Support automatically creating based on the

			Devicename carried in the registration request
Security	High	General	General
Use Limits	Up to 1 million devices can be created under a single product	Up to 1 million devices can be created under a single product	Up to 1 million devices can be created under a single product, users can customize the upper limit of devices automatically created through registration requests
Device resource requirement	High, with TLS support required	Moderately Low	Low, with only AES support required

Device-Level Key Authentication

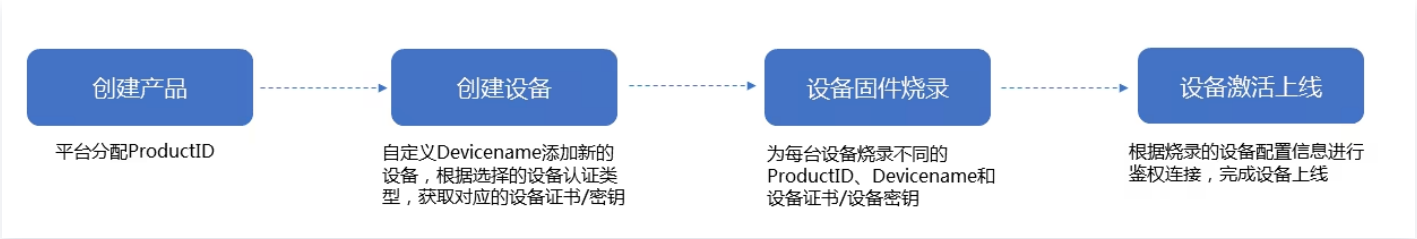
Last updated: 2025-03-19 14:52:42

Overview

The IoT Hub platform supports device-level key authentication. In this mode, you need to burn different configuration firmware for each device. The platform will perform certificate authentication or key authentication according to your selection. After successful authentication, devices can establish a connection with the platform for data communication.

Flowchart

Device-level key authentication requires you to burn different firmware for each device. It incurs certain implementation costs in production applications, but it has higher security and is thus recommended.



Operations

1. Log in to the IoT Hub console and create a product and a device. For specific creation steps, please refer to Device Connection Preparations.
2. Click on the target product name to get the product information on the product details page; select **Device List,** click the device name to get the device name and device certificate/key on the device details page.
   - Product information

- Device information

3. Burn the device firmware in the following steps:
   3.1 Download Device SDK.
   3.2 Implement the HAL layer functions in the SDK for reading and writing product and device information, including ProductID, DeviceName, and device certificate or key. For details, refer to Device Access.
   3.3 Develop the device firmware based on the SDK according to your actual business needs so as to implement various features such as unique device ID reading, dynamic device registration, connection authentication, communication, and OTA.
   3.4 In the production process, batch burn the developed and tested device firmware into the device.
   3.5 The device uses the burned device-level certificate/key to establish a connection with the platform. After successful authentication, it will be activated and connected. At this time, it can exchange data with the cloud to implement business requirements.

Product-Level Key Authentication

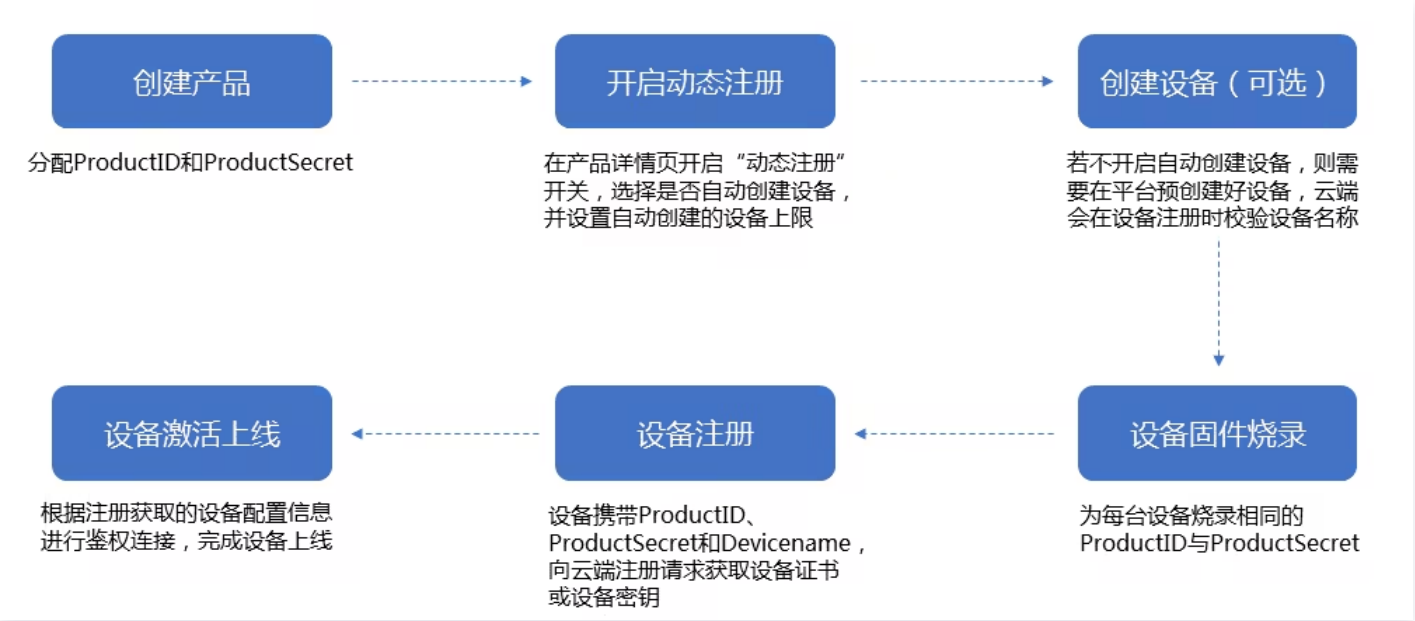
Last updated: 2025-03-19 14:52:55

Overview

The IoT Hub platform supports product-level key authentication. In this mode, you only need to enable dynamic device registration and then burn the same configuration firmware (ProductID + ProductSecret) for all devices under the same product. In this way, the devices can get device certificates or keys through registration requests and then communicate with the platform.

Flowchart

> **Note:**
> If you want to use the dynamic registration feature, you need to manually enable dynamic registration for the product on the product details page in the console.



Operation Steps

1. Log in to the IoT Hub console and create a product as instructed in Device Connection Preparations.
2. Click the target product name, enable the dynamic registration switch on the product details page, choose whether to automatically create devices, and set the upper limit for automatically created devices.

> **Notes:**
> To prevent too many devices from being created in unpredictable situations (such as device firmware bugs and product key theft), if you select automatic device creation, we recommend you set an appropriate device quantity upper limit.

3. Create devices under the product (optional).
   - You can click "Add New Device" in the device list on the console or create devices through TencentCloud API.
   - If automatic device creation is not enabled, the cloud will verify whether the device name of each request has been created on the cloud during device registration. It is recommended to use a unique identifier readable by the device as the DeviceName, such as the device's IMEI number, SN number, MAC address, etc., to facilitate the smooth completion of the entire process.

4. Burn the device firmware in the following steps:
   4.1 Download Device SDK.
   4.2 Implement the HAL layer functions in the SDK for reading and writing product and device information, including ProductID, ProductSecret, and DeviceName, and enable dynamic registration in the SDK, as described in Device Connection.
   4.3 Develop the device firmware based on the SDK according to your actual business needs so as to implement various features such as unique device ID reading, dynamic device registration, connection authentication, communication, and OTA.
   4.4 In the production process, batch burn the developed and tested device firmware into the device.
   4.5 After successful device registration, power-on, and connection, initiate a registration request to get the device certificate or key.
   4.6 The device uses the obtained device-level certificate/key to establish a connection with the platform. After successful authentication, it will be activated and connected. At this time, it can exchange data with the cloud to implement business requirements.

Dynamic Registration API Description

Last updated: 2025-03-19 14:53:07

Parameter Description

When performing dynamic device registration, carry ProductId and DeviceName to initiate an `http/https` request to the platform. The request API and parameters are as follows:

- The requested URL is: `https://ap-guangzhou.gateway.tencentdevices.com/device/register` `http://ap-guangzhou.gateway.tencentdevices.com/device/register`
- Request method: Post

Request Parameter

Parameter Name	Required	Type	Description
ProductId	Yes	string	Product Id.
DeviceName	Yes	string	Device name.
EncryptionType	No	Int64	When you choose to use custom CA certificate authentication, upload this field with a value of 1.
ClientCert	No	string	When you choose to use custom CA certificate authentication, upload this field with the device certificate file string.

Note:
The API only supports application/json format.

Signature Generation

Use the HMAC-sha256 algorithm to sign the request message. For details, see Signature Method.

Platform Response Parameters

Parameter Name	Type	Description
RequestId	String	Request Id.
Len	Int64	The length of the returned Payload.
Payload	String	Returned device registration information. This data is returned after encryption and needs to be decrypted by the device.
State	Int64	Status: 0: newly-added; 1: already exists.

Note:
The encryption process is to convert the original json format Payload into a string, then perform AES encryption, and then perform base64 encryption. The AES encryption algorithm is in CBC mode, with a key length of 128 bits, taking the first 16 bits of productSecret, and an offset of 16 characters "0".

Content description of the original Payload

key	value	Description
encryptionType	1	Encryption type. • 1 indicates certificate authentication. • 2 indicates key authentication.
psk	1239466501	Device key. This parameter is present when the product certification type is signature authentication.
clientCert	-	Device certificate file string format. This parameter is present when the product certification type is certificate authentication.
clientKey	-	Device private key file string format. This parameter is present when the product authentication type is certificate authentication.

Example Code

Request Packet

```
POST https://ap-guangzhou.gateway.tencentdevices.com/device/register
Content-Type: application/json
```

```
Host: ap-guangzhou.gateway.tencentdevices.com
X-TC-Algorithm: HmacSha256
X-TC-Timestamp: 1551****65
X-TC-Nonce: 5456
X-TC-Signature:
2230eefd229f582d8b1b891af7107b91597****07d778ab3738f756258d7652c
{"ProductId":"ASJ****GX","DeviceName":"xyz"}
```

Response Package

```
{
    "Response": {
        "Len": 53,
        "Payload":
"s6FB3a1BA/YYbcmSE12XpeDVmQNDcf1QgVD141RRbmmAnFwQfp1ECAu5O016mCOvYlJJ6V5
9yM4OqQSiWphfTg==",
        "RequestId": "636a3cdc-*****-a226-858c",
        "State": 1
    }
}
```

Payload Data Parsing Example

> **Note:**
> The following data is provided only for your testing purposes. Ensure that your information is not leaked when you use it officially.

1. The original content of the Payload is:

```
s6FB3a1BA/YYbcmSE12XpeDVmQNDcf1QgVD141RRbmmAnFwQfp1ECAu5O016mCOvYlJJ6V
59yM4OqQSiWphfTg==
```

2. Base64 Decoded:

```
b3a141ddad4103f6186dc992135d97a5e0d599034371fd508150f5e354516e69809c5c
107e9d44080bb93b4d7a9823af625249e95e7dc8ce0ea904a25a985f4e
```

3. AES Decryption

Use AES-128 decryption, with the initialization vector vi being 16 zero bytes, use the product key as the key, and adopt the CBC mode.

- Product Key: `hzvf5LF9S0isvBhDSauWMaIk`
- decrypted data: `{"encryptionType":2,"psk":"lDZ6Uqt+I9E0wW7rvDUs7Q=="}`

go decryption example:

```
import (
    "bytes"
    "crypto/aes"
    "crypto/cipher"
    "encoding/base64"
    "fmt"
)

var vi = bytes.Repeat([]byte{'0'}, 16)

func aesDecrypt(cipherData, key []byte) ([]byte, error) {
    block, err := aes.NewCipher(key)
    if err != nil {
       return nil, err
    }
    blockSize := block.BlockSize()

    plainData := make([]byte, len(cipherData))
    cipherBlockMode := cipher.NewCBCDecrypter(block, vi[:blockSize])
    cipherBlockMode.CryptBlocks(plainData, cipherData)

    padding := len(plainData)
    for i := 0; i < padding; i++ {
       if plainData[i] == 0 {
          padding = i
          break
       }
    }

    return plainData[:padding], nil
}

func main() {
```

```
    productSecret := "hzvf5LF9S0isvBhDSauWMaIk"
    key := []byte(productSecret[:16]) // Take the first 16 bytes as
the AES-128 key
    aesBase64Result :=
"s6FB3a1BA/YYbcmSE12XpeDVmQNDcf1QgVD141RRbmmAnFwQfp1ECAu5O016mCOvYlJJ6
V59yM4OqQSiWphfTg=="

    ciphertext, _ := base64.StdEncoding.DecodeString(aesBase64Result)
    decryptedData, err := aesDecrypt(ciphertext, key)
    if err != nil {
       fmt.Println("decryption failed:", err)
       return
    }
    fmt.Println("decrypted data:", string(decryptedData)) //
{"encryptionType":2,"psk":"lDZ6Uqt+I9E0wW7rvDUs7Q=="}
}
```

Device Connection Protocol MQTT-Based Device Connection Equipment'S MQTT Access Based on TCP

Last updated: 2026-05-09 16:01:16

MQTT Protocol Description

Currently, Internet of Things (IoT) Hub supports MQTT standard protocol access (compatible with Version Agreement 3.1.1). For the specific protocol, please refer to MQTT 3.1.1 protocol document.

Difference From Standard MQTT

1. Support MQTT messages such as PUB, SUB, PING, PONG, CONNECT, DISCONNECT, and UNSUB.
2. Support cleanSession.
3. Not support will, retain msg.
4. Not support QOS2.

MQTT Channel, Security Level

Support TLSV1, TLSV1.1, and TLSV1.2 protocol editions to establish a secure connection with a high security level.

TOPIC Specification

By default, after creating a product, all devices under the product have the following topic category permissions:

- Subscribe to ${productId}/${deviceName}/control.
- `${productId}/${deviceName}/event` Published.
- `${productId}/${deviceName}/data` Subscription and publication.
- `$shadow/operation/${productId}/${deviceName}` Publish. Distinguish by the type in the package body: update/get, which correspond to the update and retrieval operations of the Device Shadow Document, respectively.
- `$shadow/operation/result/${productId}/${deviceName}` Subscribe. Distinguish by the type in the package body: update/get/delta. Type update/get correspond to the results of the update and retrieval operations of the Device Shadow Document, respectively; when

users modify the Device Shadow Document through the restAPI, the server will publish a message through this topic, where type is delta.

- `$ota/report/${productID}/${deviceName}` Publish. Devices report version numbers and download and upgrade progress to the cloud.
- `$ota/update/${productID}/${deviceName}` Subscribe. Devices receive upgrade messages from the cloud.

MQTT Access

The MQTT protocol supports two methods of access to the IoT communication platform through device certificates and key signatures. Based on your own scenario, you can choose a method to access. The access parameters are as follows:

Access Authentication Method	Connection Domain Name and Port	Connect Message Parameter
Certificate authentication	MQTT server connection address, devices in Guangzhou fill in: ${productId}.iotcloud.tencentdevices.com, here ${productId} is a variable parameter, and users must enter the product ID automatically generated when creating the product. For example, 1A17RZR3XX.iotcloud.tencentdevices.com; Port: 8883.	• KeepAlive: The duration of keep-alive, with a value range of 0 – 900 s. If the Internet of Things Platform still has not received the data of the client after exceeding 1.5 times the KeepAlive duration, the platform will disconnect from the client. • ClientId: ${productId}${deviceName}, a combined string of product ID and device name. • UserName: `${productId}${deviceName};${sdkappid};${connid};${expiry}` . For details, see the username part in the MQTT-based signature authentication access guide below. • PassWord: PassWord (can be assigned any value).
SSH key authentication	The MQTT server connection address matches the certificate authentication; Port: 1883.	• KeepAlive: The duration of keep-alive, with a value range of 0 – 900 s; • ClientId: ${productId}${deviceName}; • UserName: `${productId}${deviceName};${sdkappid};${connid};${expiry}` . For details, see the username part

		in the MQTT-based signature authentication access guide below. • PassWord: password. For details, see the password part in the MQTT-based signature authentication access guide below.

Note:
When a device with certificate authentication enabled is connecting, the PassWord field you fill in will not be verified. For certificate authentication, any value can be filled in the PassWord field.

Certificate-Authenticated Device Access Guide

The IoT platform uses the TLS encryption method to ensure the security of device-transmitted data. When a certificate device is accessing, after obtaining the certificate, key, and CA certificate file of the certificate device, set KeepAlive, ClientId, UserName, PassWord, etc. (Devices that access by using the Tencent Cloud Device SDK do not need to set these parameters. The SDK can automatically generate them based on device information). The device uploads the authentication file to the URL (connection domain name and port) corresponding to certificate authentication. After passing the verification, send an MqttConnect message to complete the certificate device's TCP-based MQTT access.

Key Authentication Device Access Guide

The IoT platform supports generating digest signatures based on the device key by using methods such as HMAC-SHA256 and HMAC-SHA1. The process of accessing the IoT cloud platform by using the signature method is as follows:

1. Log in to the IoT Hub console. You can create a product, add a device, and obtain the device key in the console.
2. Generate the username field according to the constraints of IoT Hub. The username field format is as follows:

```
The format of the username field is:
${productId}${deviceName};${sdkappid};${connid};${expiry}
Note: ${} indicates a variable, not a specific concatenation symbol.
```

The meanings of each field are as follows:

- productId: product ID.
- deviceName: device name.

- sdkappid: fixed fill 12010126.
- connid: a random string.
- expiry: indicates the signature validity period, format is unix timestamp such as 1704363215. expiry should be set to a timestamp far exceeding the actual lifecycle of the device or obtained by adding a large integer to the current system time each time the device side performs MQTT identity verification. If the value of expiry is less than the current system time of the server, MQTT identity verification will fail.

3. Decode the device key using base64 to obtain the original key raw_key.
4. Use the raw_key generated in step 3 to generate a hash value for the username through the HMAC-SHA1 or HMAC-SHA256 algorithm, abbreviated as Token.
5. Generate the password field according to the constraints of IoT Hub. The password field format is:

```
The password field format is:
${token}; hmac signature method
Among them, fill in the hash algorithm used in step 3 for the hmac
signature method field. Optional values are hmacsha256 and hmacsha1.
```

As a reference, sample code for users to generate signatures in Python, Java, Node.js, JavaScript, and C is as follows:
Python code:

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
import base64
import hashlib
import hmac
import random
import string
import time
import sys
# Generate a random string of the specified length
def RandomConnid(length):
    return  ''.join(random.choice(string.ascii_uppercase +
string.digits) for _ in range(length))
# Generate parameters required for IoT Hub integration
def IotHmac(productID, devicename, devicePsk):
```

```
    # 1. Generate connid as a random string to facilitate background
issue location
    connid   = RandomConnid(5)
    # 2. Generate the expiration time, which indicates the
signature's expiry date, as a UTF8 string representing the number of
seconds from the epoch January 1, 1970, 00:00:00 UTC to the present.
    expiry   = int(time.time()) + 60 * 60
    # 3. Generate the clientid part of MQTT in the format of
${productid}${devicename}
    clientid = "{}{}".format(productID, devicename)
    # 4. Generate the username part of MQTT in the format of
${clientid};${sdkappid};${connid};${expiry}
    username = "{};12010126;{};{}".format(clientid, connid, expiry)
    # 5. Sign the username and generate a token
    secret_key = devicePsk.encode('utf-8')  # convert to bytes
    data_to_sign = username.encode('utf-8')  # convert to bytes
    secret_key = base64.b64decode(secret_key)  # this is still bytes
    token = hmac.new(secret_key, data_to_sign,
digestmod=hashlib.sha256).hexdigest()
    # 6. Generate the password field according to the rules of the
IoT communication platform
    password = "{};{}".format(token, "hmacsha256")
    return {
        "clientid" : clientid,
        "username" : username,
        "password" : password
    }
if __name__ == '__main__':
    print(IotHmac(sys.argv[1], sys.argv[2], sys.argv[3]))
```

Save the above code to IotHmac.py and execute the following command. Here, "YOUR_PRODUCTID", "YOUR_DEVICENAME" and "YOUR_PSK" are to fill in your actual product ID, device name and device key of the created device.

```
python3 IotHmac.py "YOUR_PRODUCTID" "YOUR_DEVICENAME" "YOUR_PSK"
```

Java code is:

```
package com.tencent.iot.hub.device.java.core.sign;

import org.junit.Test;

import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import java.util.*;

import static junit.framework.TestCase.fail;
import static org.junit.Assert.assertTrue;

public class SignForMqttTest {

    @Test
    public void testMqttSign() {
        try {

System.out.println(SignForMqttTest("YourProductId","YourDeviceName","Y
ourPsk"));
            assertTrue(true);
        } catch (Exception e) {
            e.printStackTrace();
            fail();
        }
    }

    public static Map<String, String> SignForMqttTest(String
productID, String devicename, String
            devicePsk) throws Exception {
        final Base64.Decoder decoder = Base64.getDecoder();
        //1. Generate connid as a random string to facilitate
background issue location
        String connid = HMACSHA256.getConnectId(5);
        //2. Generate the expiration time, which indicates the
signature's expiration time, as a UTF8 string representing the number
of seconds from the epoch January 1, 1970, 00:00:00 UTC to now.
        Long expiry = Calendar.getInstance().getTimeInMillis()/1000 +
600;
```

```
        //3. Generate the clientid part of MQTT in the format of
${productid}${devicename}
        String clientid = productID+devicename;
        //4. Generate the username part of MQTT in the format of
${clientid};${sdkappid};${connid};${expiry}
        String username = clientid+";"+"12010126;"+connid+";"+expiry;
        //5. Sign the username, generate a token, and generate the
password field according to the rules of the IoT Hub
        String password = HMACSHA256.getSignature(username.getBytes(),
decoder.decode(devicePsk)) + ";hmacsha256";
        Map<String,String> map = new HashMap<>();
        map.put("clientid",clientid);
        map.put("username",username);
        map.put("password",password);
        return map;
    }
    public static class HMACSHA256 {
        private static final String HMAC_SHA256 = "HmacSHA256";
        /**
         * Generate signature data
         *
         * @param data Data to be encrypted
         * @param key Key used for encryption
         * @return Generate a hexadecimal encoded string
         */
        public static String getSignature(byte[] data, byte[] key)  {
            try {
                SecretKeySpec signingKey = new SecretKeySpec(key,
HMAC_SHA256);
                Mac mac = Mac.getInstance(HMAC_SHA256);
                mac.init(signingKey);
                byte[] rawHmac = mac.doFinal(data);
                return bytesToHexString(rawHmac);
            }catch (Exception e) {
                e.printStackTrace();
            }
            return null;
        }
        /**
         * Convert byte[] array to a hexadecimal string
```

```
     *
     * @param bytes Byte array to be switched
     * @return The converted result
     */
    private static String bytesToHexString(byte[] bytes) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < bytes.length; i++) {
            String hex = Integer.toHexString(0xFF & bytes[i]);
            if (hex.length() == 1) {
                sb.append('0');
            }
            sb.append(hex);
        }
        return sb.toString();
    }


    /**
     * Get connection ID (a random alphanumeric string with a
length of 5)
     */
    public static String getConnectId(int length) {
        StringBuffer connectId = new StringBuffer();
        for (int i = 0; i < length; i++) {
            int flag = (int) (Math.random() * Integer.MAX_VALUE) %
3;
            int randNum = (int) (Math.random() *
Integer.MAX_VALUE);
            switch (flag) {
                case 0:
                    connectId.append((char) (randNum % 26 + 'a'));
                    break;
                case 1:
                    connectId.append((char) (randNum % 26 + 'A'));
                    break;
                case 2:
                    connectId.append((char) (randNum % 10 + '0'));
                    break;
            }
        }
```

```
            return connectId.toString();
        }
    }

}
```

Node.js and JavaScript code are:

```
// Following is the introduction method for node. For browsers, use
the corresponding method to introduce the crypto-js library.
const crypto = require('crypto-js')

// Function to generate a random number
const randomString = (len) => {
    len = len || 32;
    var chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789';
    var maxPos = chars.length;
    var pwd = '';
    for (let i = 0; i < len; i++) {
        pwd += chars.charAt(Math.floor(Math.random() * maxPos));
    }
    return pwd;
}
// product id, device name and device key are required.
const productId = 'YOUR_PRODUCTID';
const deviceName = 'YOUR_DEVICENAME';
const devicePsk = 'YOUR_PSK';

// 1. Generate connid as a random string to facilitate backend issue
location.
const connid =  randomString(5);
// 2. Generate expiration time, which indicates the signature's
expiration time, as a UTF8 string of seconds from the epoch January 1,
1970, 00:00:00 UTC to now
const expiry = Math.round(new Date().getTime() / 1000) + 3600 * 24;
// 3. Generate the clientid part of MQTT in the format of
${productid}${devicename}
```

```
const clientId = productId + deviceName;
// 4. Generate the username part of MQTT in the format of
${clientid};${sdkappid};${connid};${expiry}
const userName = `${clientId};12010126;${connid};${expiry}`;
//5. Sign the username, generate a token, and generate the password
field according to the rules of the IoT Hub.
const rawKey = crypto.enc.Base64.parse(devicePsk);      // Decode the
device key using base64
const token =  crypto.HmacSHA256(userName, rawKey);
const password = token.toString(crypto.enc.Hex) + ";hmacsha256";
console.log(`userName:${userName}\npassword:${password}`);
```

The following is C code:

> **Note:**
> If you want to learn more about C code content, for details, see project download.

```
#include "limits.h"
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#include "HAL_Platform.h"
#include "utils_base64.h"
#include "utils_hmac.h"

/* Max size of base64 encoded PSK = 64, after decode: 64/4*3 = 48*/
#define DECODE_PSK_LENGTH 48

/* MAX valid time when connect to MQTT server. 0: always valid */
/* Use this only if the device has accurate UTC time. Otherwise, set
to 0 */
#define MAX_ACCESS_EXPIRE_TIMEOUT (0)

/* Max size of conn Id  */
#define MAX_CONN_ID_LEN (6)

/* IoT C-SDK APPID */
```

```
#define QCLOUD_IOT_DEVICE_SDK_APPID      "21****06"
#define QCLOUD_IOT_DEVICE_SDK_APPID_LEN
(sizeof(QCLOUD_IOT_DEVICE_SDK_APPID) - 1)

static void HexDump(char *pData, uint16_t len)
{
    int i;

    for (i = 0; i < len; i++) {
        HAL_Printf("0x%02.2x ", (unsigned char)pData[i]);
    }
    HAL_Printf("\n");
}

static void get_next_conn_id(char *conn_id)
{
    int i;
    srand((unsigned)HAL_GetTimeMs());
    for (i = 0; i < MAX_CONN_ID_LEN - 1; i++) {
        int flag = rand() % 3;
        switch (flag) {
            case 0:
                conn_id[i] = (rand() % 26) + 'a';
                break;
            case 1:
                conn_id[i] = (rand() % 26) + 'A';
                break;
            case 2:
                conn_id[i] = (rand() % 10) + '0';
                break;
        }
    }

    conn_id[MAX_CONN_ID_LEN - 1] = '\0';
}

int main(int argc, char **argv)
{
    char *product_id    = NULL;
    char *device_name   = NULL;
```

```
    char *device_secret = NULL;

    char *username     = NULL;
    int   username_len = 0;
    char  conn_id[MAX_CONN_ID_LEN];

    char password[51]      = {0};
    char username_sign[41] = {0};

    char   psk_base64decode[DECODE_PSK_LENGTH];
    size_t psk_base64decode_len = 0;

    long cur_timestamp = 0;

    if (argc != 4) {
        HAL_Printf("please ./qcloud-mqtt-sign product_id device_name
device_secret\r\n");
        return -1;
    }

    product_id    = argv[1];
    device_name   = argv[2];
    device_secret = argv[3];

    /* first device_secret base64 decode */
    qcloud_iot_utils_base64decode((unsigned char *)psk_base64decode,
DECODE_PSK_LENGTH, &psk_base64decode_len,
                                  (unsigned char *)device_secret,
strlen(device_secret));
    HAL_Printf("device_secret base64 decode:");
    HexDump(psk_base64decode, psk_base64decode_len);

    /* second create mqtt username
     * [productdevicename;appid;randomconnid;timestamp] */
    cur_timestamp = HAL_Timer_current_sec() +
MAX_ACCESS_EXPIRE_TIMEOUT / 1000;
    if (cur_timestamp <= 0 || MAX_ACCESS_EXPIRE_TIMEOUT <= 0) {
        cur_timestamp = LONG_MAX;
    }
```

```
    // 20 for timestampe length & delimiter
    username_len = strlen(product_id) + strlen(device_name) +
QCLOUD_IOT_DEVICE_SDK_APPID_LEN + MAX_CONN_ID_LEN + 20;
    username     = (char *)HAL_Malloc(username_len);
    if (username == NULL) {
        HAL_Printf("malloc username failed!\r\n");
        return -1;
    }

    get_next_conn_id(conn_id);
    HAL_Snprintf(username, username_len, "%s%s;%s;%s;%ld", product_id,
device_name, QCLOUD_IOT_DEVICE_SDK_APPID,
                 conn_id, cur_timestamp);

    /* third use psk_base64decode hamc_sha1 calc mqtt username sign
crate mqtt
     * password */
    utils_hmac_sha1(username, strlen(username), username_sign,
psk_base64decode, psk_base64decode_len);
    HAL_Printf("username sign: %s\r\n", username_sign);
    HAL_Snprintf(password, 51, "%s;hmacsha1", username_sign);

    HAL_Printf("Client ID: %s%s\r\n", product_id, device_name);
    HAL_Printf("username : %s\r\n", username);
    HAL_Printf("password : %s\r\n", password);

    HAL_Free(username);

    return 0;
}
```

6. Finally, fill in the above generated parameters into the corresponding MQTT connect message.
7. Enter the clientid into the clientid field of the MQTT protocol.
8. Enter the username into the username field of MQTT.
9. Enter the password into the password field of MQTT and send the MqttConnect message to the domain name and port of key authentication to integrate into the IoT Cloud Communication Platform.

MQTT-Based Device Connection over WebSocket

Last updated: 2025-03-19 14:54:09

MQTT-WebSocket Overview

The IoT Hub platform supports MQTT communication based on WebSocket, so that devices can use the MQTT protocol for message transfer on the basis of the WebSocket protocol. In this way, browser-based applications can implement data communication with the platform and devices connected to the platform. In addition, WebSocket uses ports 443/80, which means that messages can pass through most firewalls during transfer.

MQTT-WebSocket Connection

As both the MQTT-WebSocket and MQTT-TCP protocols ultimately transfer messages based on MQTT, they have the same parameters for MQTT connection. The main difference lies in the protocol and port of the MQTT connection to the platform. Key-authenticated devices use WS for connection, while certificate-authenticated devices use WSS, i.e., WS+TLS.

Guide to connecting certificate-authenticated device

1. Download files such as the certificate and device private key.
2. Connection Domain: Devices in the Guangzhou domain need to connect to `${ProductId}.ap-guangzhou.iothub.tencentdevices.com:443`, where ${ProductId} is a variable parameter for the product ID.
3. Set the MQTT connection parameters:
   The connection parameter settings are the same as those for MQTT-TCP connection. For more information, please see the MQTT connection section in MQTT-Based Device Connection over TCP documentation.

```
UserName:${productid}${devicename};${sdkappid};${connid};${expiry}
PassWord: password (you can set any value)
ClientId:${ProductId}${DeviceName}
KeepAlive: time to keep the connection alive. Value range: 0-900s
```

Guide to connecting key-authenticated device

1. Get the device key.

2. Connect Domain: Guangzhou Domain devices need to connect, `${ProductId}.ap-guangzhou.iothub.tencentdevices.com:80`, where ${ProductId} is the variable parameter for the Product ID.
3. Set the MQTT connection parameters:
   The connection parameter settings are the same as those for MQTT-TCP connection. For more information, please see the Key Device Access Guide section in the MQTT-Based Device Connection over TCP documentation.

```
UserName:${productid}${devicename};${sdkappid};${connid};${expiry}
PassWord:${token};hmac signature algorithm
ClientId:${ProductId}${DeviceName}
KeepAlive: time to keep the connection alive. Value range: 0-900s
```

MQTT Persistent Session

Last updated: 2025-03-19 14:54:21

IoT Hub supports the MQTT v3.1.1 protocol and supports the quality of service levels of QoS 0 and QoS 1 (but not QoS 2). Using an MQTT persistent session can save the subscription status of devices and subscribed messages that devices have not received. When a device goes offline and goes online again, it can restore the previous session to receive subscribed messages sent when it was offline.

Creating MQTT Persistent Session on Device

When the device connects to IoT Hub, it can set the CleanSession flag in the variable header of the Connect message to 0. IoT Hub will determine the session state of the device based on the ClientId of the device when connected. If there is no current session, a new persistent session will be created. If there is an existing session, communication will be based on the existing session process.

IoT Hub response description

After the device sends the Connect message, IoT Hub will return a Connack message. The SessionPresent flag in the message indicates whether IoT Hub already contains the session state corresponding to the ClientId of the device. SessionPresent being 0 indicates that a persistent session has not been created, and the device needs to re-establish a session state. SessionPresent being 1 indicates that a persistent session has been created.

- After the device is successfully connected, if it enters an existing persistent session, IoT Hub will send the stored QoS 1 messages and unacknowledged QoS 1 messages to the device.
- After the device is successfully connected, if a new persistent session is created, IoT Hub will save the subscription status of the device and store the QoS 1 (excluding QoS 0) messages that the device has subscribed to when it is offline. When it goes online again, IoT Hub will send the stored QoS 1 messages and unacknowledged QoS 1 messages to it.

Note:

- IoT Hub sends the stored QoS 1 messages sequentially at 500 ms intervals.
- Only QoS 1 messages can be stored in a persistent session. Up to 150 messages can be stored for a maximum of 24 hours for each device.

Closing MQTT Persistent Session

You can close the MQTT Persistent Session in the following two ways.

- When the device connects to IoT Hub, set the CleanSession flag bit in the Connect message variable header to 1.
- When the device is disconnected for **more than 24 hours**, the persistent session will be closed automatically.

> **Note:**
> The disconnection of the device includes the disconnection caused by the device sending a disconnect message and communication timeout.

CoAP-Based Device Connection

Last updated: 2025-03-19 14:54:36

Currently, IoT Hub supports CoAP standard protocol access. For more information, please see RFC7252.

Differences from Standard CoAP

1. Currently, only message reporting is supported, i.e., reporting SDK messages to IoT Hub.
2. The POST method is supported, while GET/PUT/DELETE methods are not.

Security Level of CoAP Channel

1. DTLS protocol is supported to establish a secure connection.
2. Asymmetric encryption is supported.

Access Parameters

1. The server address for Guangzhou Domain Devices is: ${ProductId}.iotcloud.tencentdevices.com. Here, ${ProductId} is a variable parameter. Users need to input the Product ID automatically generated when creating the product.
2. The connection port is: 5684.

URI Specification

CoAP messages are sent to the URI. The URI format is `/${productId}/${deviceName}/xxx`. The productId is the product ID registered in the console, and the deviceName is the device name under the productId.
By default, after a product is created, all devices under it will have the permissions of the following topic classes:

- `${productId}/${deviceName}/event` for publishing.
- `${productId}/${deviceName}/control` for subscribing.
- `${productId}/${deviceName}/data` for publishing and subscribing.

In other words, the URI corresponds to the MQTT topic.

HTTP-Based Device Connection

Last updated: 2025-03-19 14:54:49

Parameter Description

When reporting messages, the device needs to carry ProductId, DeviceName, and TopicName to initiate an `http/https` request to the platform. The request interface and parameter description are as follows:

- Requested URL:
  `https://ap-guangzhou.gateway.tencentdevices.com/device/publish`
  `http://ap-guangzhou.gateway.tencentdevices.com/device/publish`
- Request method: POST.

Request Parameters

Parameter Name	Required	Type	Description
ProductId	Yes	String	Product ID.
DeviceName	Yes	String	Device name.
TopicName	Yes	String	Name of the topic for publishing the message.
Payload	Yes	String	Content of the published message.
PayloadEncoding	No	String	Encoding of the published message. Currently, only base64 encoding is supported. If not provided, the original message content will be sent by default.
Qos	Yes	Integer	Message QoS level.

Note:
The interface only supports application/json format.

Signature generation

There are two types of signatures for request messages. Key authentication uses the HMAC-sha256 algorithm, and certificate authentication uses the RSA_SHA256 algorithm. For more information, please see Signature method.

Platform response parameters

Parameter Name	Type	Description
RequestId	String	Request ID.

Sample Code

Request package

```
POST https://ap-guangzhou.gateway.tencentdevices.com/device/publish
Content-Type: application/json
Host: ap-guangzhou.gateway.tencentdevices.com
X-TC-Algorithm: HmacSha256
X-TC-Timestamp: 155****065
X-TC-Nonce: 5456
X-TC-Signature:
2230eefd229f582d8b1b891af7107b915972407****78ab3738f756258d7652c
{"DeviceName":"AAAAAA","Payload":"123","ProductId":"G8N**AHB","Qos":1,"T
opicName":"G8N**AHB/AAAAAA/data"}
```

Return package

```
{
  "Response": {
    "RequestId": "f4da4f1f-d72e-40f1-****-349fc0072ba0"
  }
}
```

Device Connection Regions

Last updated: 2025-03-19 14:55:17

Currently supported IoT Hub access regions:

Domestic site	International site
Chinese mainland	China Mainland
East US (Virginia)	U.S East (Virginia)
Central Europe (Frankfurt)	Central Europe (Frankfurt)
Southeast Asia (Bangkok)	Southeast Asia Pacific (Bangkok)

MQTT connected domain

When accessing devices, users can choose the following server addresses based on their needs:

Region	Domain name
Chinese mainland	${productid}.iotcloud.tencentdevices.com
East US (Virginia)	${productid}.us-east.iotcloud.tencentdevices.com
Central Europe (Frankfurt)	${productid}.europe.iothub.tencentdevices.com
Southeast Asia (Bangkok)	${productid}.ap-bangkok.iothub.tencentdevices.com

CoAP connected domain

When accessing devices, users can choose the following server addresses based on their needs:

Region	Domain name
Chinese mainland	${productid}.iotcloud.tencentdevices.com
East US (Virginia)	${productid}.us-east.iotcloud.tencentdevices.com

Central Europe (Frankfurt)	${productid}.europe.iothub.tencentdevices.com
Southeast Asia (Bangkok)	${productid}.ap-bangkok.iothub.tencentdevices.com

HTTP access domain name

When accessing devices, users can choose the following server addresses based on their needs:

Region	Domain name
Chinese mainland	ap-guangzhou.gateway.tencentdevices.com
East US (Virginia)	us-east.gateway.tencentdevices.com
Central Europe (Frankfurt)	europe.gateway.tencentdevices.com
Southeast Asia (Bangkok)	ap-bangkok.gateway.tencentdevices.com

Gateway Subdevice Feature Overview

Last updated: 2025-03-19 14:56:01

Equipment Classification

The IoT Hub platform divides devices into the following two categories (i.e., node categories) according to their functionality:

- **Ordinary Devices**: This device category is further divided into two types: devices that have the ability to connect to the platform, and devices that can connect to the platform through gateway devices.
- **Gateway device**: this category of device can directly connect to the Internet of Things Platform, and can accept sub-devices for them to join the Local Area Network.

Overview

Devices that don't have direct access to the Ethernet can be connected to the network of the local gateway device first and then connected to the IoT Hub platform through the communication feature of the gateway device. For the subdevices that join or leave the LAN, the gateway device can bind or unbind them on the platform and report the topological relationships with them, so as to implement the real-time monitoring of all devices in the LAN by the platform.

Connection Method

- The method for gateway devices to connect to the IoT Hub platform is the same as that for regular devices; for details, refer to Device Access. Once connected, gateway devices can manage the online/offline status of sub-devices within the same local network and manage the topology relationship with sub-devices.
- Sub-devices need to be connected through the gateway device. Once the sub-devices are authenticated by the gateway device, they can successfully connect to the cloud. There are two authentication methods:
  - **Device-Level Key Authentication**
    The gateway obtains the device certificate or key of the sub-device and generates a binding signature string for it. The gateway then reports the binding signature string information of the sub-device to the platform and verifies the sub-device's identity on its behalf.
  - **Product-Level Key Authentication**
    The gateway obtains the ProductKey of the sub-device and generates a signature. The

gateway sends a dynamic registration request to the platform. Once the verification is successful, the platform returns the DeviceCert or DeviceSecret of the sub-device to the gateway, which then generates a binding signature string for the sub-device and reports it to the platform. After successful verification, the sub-device is connected.

Topological Relationship Management

Last updated: 2026-03-20 15:15:12

Feature Overview

A gateway device can bind and unbind subdevices under it through data communication with the cloud. To implement this feature, the following two topics will be used:

- Upstream data Topic (for publishing): `$gateway/operation/${productid}/${devicename}`
- Downstream data Topic (for subscribing): `$gateway/operation/result/${productid}/${devicename}`

Binding a Device

The gateway device can request to add its topological relationship with the subdevice through the data upstream topic so as to bind the subdevice. After the request succeeds, the platform will return the binding information of the subdevice through the data downstream topic.
Data format of the subdevice binding request:

```
{
  "type": "bind",
  "payload": {
    "devices": [
      {
        "product_id": "CFCS****G7",
        "device_name": "****ev",
        "signature": "signature",
        "random": 121213,
        "timestamp": 1589786839,
        "signmethod": "hmacsha256",
        "authtype": "psk"
      }

  }
}
```

Request parameter description:

Parameters	Type	Description
type	String	Gateway message type. The value for binding sub-devices is: `bind`.
payload.devices	Array	List of subdevices to be bound.
product_id	String	Subdevice product ID.
device_name	String	Subdevice name.
signature	String	Signature string for subdevice binding. Signature algorithm: 1. Signature string, concatenate the product ID, device name, random number, and timestamp: `text=${product_id}${device_name};${random}${expiration_time}`. 2. Sign using the device's Psk secret or the certificate's Sha1 digest: `sign = hmac_sha1(device_secret, text)`.
random	Int	A random number.
timestamp	Int	Timestamp in seconds.
signmethod	String	Signature algorithm. HMACSHA1 and HMACSHA256 are supported.
authtype	String	Signature type. • PSK: Use the device PSK to sign. • certificate: Use the device's public key certificate for signature.

Data format of the subdevice binding response:

```
{
  "type": "bind",
  "payload": {
    "devices": [
      {
        "product_id": "CFCS****G7",
        "device_name": "****ev",
        "result": -1
```

```
        }

    }
}
```

Response parameter description:

Parameters	Type	Description
type	String	Gateway message type. For subdevice binding, the value is `bind`.
payload.devices	Array	List of subdevices to be bound.
product_id	String	Subdevice product ID.
device_name	String	Subdevice name.
result	Int	Subdevice binding result, see the following table for specific error code.

Unbind Device

The gateway device can request to unbind its topological relationship with the subdevice through the data upstream topic. After the request succeeds, the platform will return the unbinding information of the subdevice through the data downstream topic.
Data format of the subdevice unbinding request:

```
{
  "type": "unbind",
  "payload": {
    "devices": [
      {
        "product_id": "CFCS****G7",
        "device_name": "****ev"
      }

    }
}
```

Request parameter description:

Parameters	Type	Description
type	String	Gateway message type. For subdevice unbinding, the value is `unbind`.
payload.devices	Array	List of subdevices to be unbound.
product_id	String	Subdevice Product ID.
device_name	String	Subdevice Name.

Data format of the subdevice unbinding response:

```
{
  "type": "unbind",
  "payload": {
    "devices": [
      {
        "product_id": "CFCS****G7",
        "device_name": "****ev",
        "result": -1
      }

  }
}
```

Response parameter description:

Parameters	Type	Description
type	String	Gateway message type. For subdevice unbinding, the value is `unbind`.
payload.devices	Array	List of subdevices to be unbound.
product_id	String	Subdevice Product ID.
device_name	String	Subdevice Name.
result	Int	Subdevice binding result, see Error Code.

Query Topology Relationship

Gateway devices can query the subdevice topology relationship via the upstream request of this topic.
Data Upstream Topic: `$gateway/operation/${productid}/${devicename}`
Data Downstream Topic: `$gateway/operation/result/${productid}/${devicename}`
Gateway Query Subdevice Topology Relationship Request Data Format:

```
{
    "type": "describe_sub_devices"
}
```

Request parameter description:

Parameters	Type	Description
type	String	Gateway message type. For subdevice query, the value is `describe_sub_devices`.

Gateway Query Subdevice Topology Relationship Response Data Format:

```
{
  "type": "describe_sub_devices",
  "payload": {
    "devices": [
      {
        "product_id": "XKFA****LX",
        "device_name": "2OGDy7Ws8mG****YUe"
      }
      {
        "product_id": "XKFA****LX",
        "device_name": "5gcEHg3Yuvm****2p8"
      }
      {
        "product_id": "XKFA****LX",
        "device_name": "hmIjq0gEFcf****F5X"
      }
      {
        "product_id": "XKFA****LX",
```

```
        "device_name": "x9pVpmdRmET****mkM"
      }
      {
        "product_id": "XKFA****LX",
        "device_name": "zmHv6o6n4G3****Bgh"
      }

  }
}
```

Response parameter description:

Parameters	Type	Description
type	String	Gateway message type. The value for querying subdevices is: `describe_sub_devices`.
payload.devices	Array	List of subdevices bound to the gateway.
product_id	String	Subdevice product ID.
device_name	String	Subdevice name.

Topological relationship change

The gateway device can subscribe to this topic for receiving topological relationship changes of subdevices.
Data downstream topic: `$gateway/operation/result/${productid}/${devicename}`
When subdevices are bound or unbound, the gateway will receive topological relationship changes of subdevices with the following data format:

```
{
  "type": "change",
  "payload": {
  "status": 0,  //0-Unbound 1-Bound
  "devices": [
      {
        "product_id": "CFCS****G7",
        "device_name": "****ev",
      }
```

```
    }
}
```

Request parameter description:

Parameters	Type	Description
type	String	Gateway Message Type. Topology change value: change.
status	Int	Topology Change Status. • 0: Unbind. • 1: Bind.
payload.devices	Array	List of Subdevices bound to the gateway.
product_id	String	Subdevice Product ID.
device_name	String	Subdevice Name.

Gateway Response, data format as follows:

```
{
  "type": "change",
  "result": 0
}
```

Response parameter description:

Parameters	Type	Description
type	String	Gateway Message Type. Topology change value: change.
result	Int	Gateway Response Process Results.

Error Code

Error Code	Description

0	Execution succeeded.
-1	The gateway device is not bound to the subdevice.
-2	System error. Subdevice connection or disconnection failed.
801	Request parameter error.
802	The device name is invalid, or the device does not exist.
803	Signature verification failed.
804	The signature algorithm is not supported.
805	The signature request has expired.
806	This device has already been bound.
807	Non-general devices cannot be bound.
808	Forbidden operation.
809	Duplicate binding.
810	Unsupported subdevice.

Proxied Subdevice Connection and Disconnection

Last updated: 2025-03-19 14:56:32

Feature Overview

A gateway device can proxy the online and decommission operations of its sub-devices through data communication with the cloud. The topics used for this feature are the same as those for gateway subdevice topology management:

- Upstream data Topic (for publishing): `$gateway/operation/${productid}/${devicename}`
- Downstream data Topic (for subscribing): `$gateway/operation/result/${productid}/${devicename}`

Proxies subdevice connection

The gateway device can proxy subdevice connection through the data upstream topic. After the request succeeds, the platform will return the connection information of the subdevice through the data downstream topic.
Data format of the proxy subdevice connection request:

```
{
  "type": "online",
  "payload": {
    "devices": [
      {
        "product_id": "CFCSQ5EAG7",
        "device_name": "onlinedev00"
      }

  }
}
```

Data format of the proxy subdevice connection response:

```
{
  "type": "online",
  "payload": {
    "devices": [
```

```
        {
          "product_id": "CFCSQ5EAG7",
          "device_name": "onlinedev00",
          "result":0
        }

    }
}
```

Request parameter description:

Field	Type	Description
type	String	Gateway message type. For agent sub-device online, the value is `online`.
payload.devices	Array	List of sub-devices to be online.
product_id	String	Subdevice product ID.
device_name	String	Subdevice name.

Response parameter description:

Field	Type	Description
type	String	Gateway message type. For agent sub-device online, the value is `online`.
payload.devices	Array	List of sub-devices to be online.
product_id	String	Subdevice product ID.
device_name	String	Subdevice name.
result	Int	Subdevice connection results. For specific error codes, see the table below.

Proxies subdevice disconnection.

The gateway device can proxy subdevice disconnection through the data upstream topic. After the request succeeds, the platform will return the successful disconnection information of the subdevice through the data downstream topic.
Data format of the subdevice disconnection request:

```
{
  "type": "offline",
  "payload": {
    "devices": [
      {
        "product_id": "CFCSQ5EAG7",
        "device_name": "offlinedev00"
      }

  }
}
```

Data format of the subdevice disconnection response:

```
{
  "type": "offline",
  "payload": {
    "devices": [
      {
        "product_id": "CFCSQ5EAG7",
        "device_name": "offlinedev00",
        "result":-1
      }

  }
}
```

Request parameter description:

Field	Type	Description
type	String	Gateway message type. The value for agent sub-device offline is: `offline`.
payload.devices	Array	List of sub-devices to be proxied offline.
product_id	String	Subdevice product ID.
device_name	String	Subdevice name.

Response parameter description:

Field	Type	Description
type	String	Gateway message type. The value for agent sub-device offline is: `offline`.
payload.devices	Array	List of sub-devices to be proxied offline.
product_id	String	Subdevice product ID.
device_name	String	Subdevice name.
result	Int	Subdevice offline result. For specific error codes, see the table below.

Error Codes

Error Codes	Description
0	Execution succeeded.
-1	The gateway device is not bound to the subdevice.
-2	System error. Subdevice connection or disconnection failed.
801	Request parameter error.
802	The device name is invalid, or the device does not exist.
810	Unsupported subdevice.

Proxied Subdevice Publishing and Subscribing

Last updated: 2025-03-19 14:57:03

Feature Overview

A gateway device can publish and subscribe to messages on behalf of subdevices under it through data communication with the cloud.

Prerequisites

Before publishing and subscribing to messages, please connect gateway devices and subdevices as instructed in Gateway Product Connection and Proxied Subdevice Connection and Disconnection.

Publishing and Subscribing to Messages

Gateway devices can use the topic permissions of subdevices and send/receive messages on behalf of subdevices.

Subdevice Firmware Update

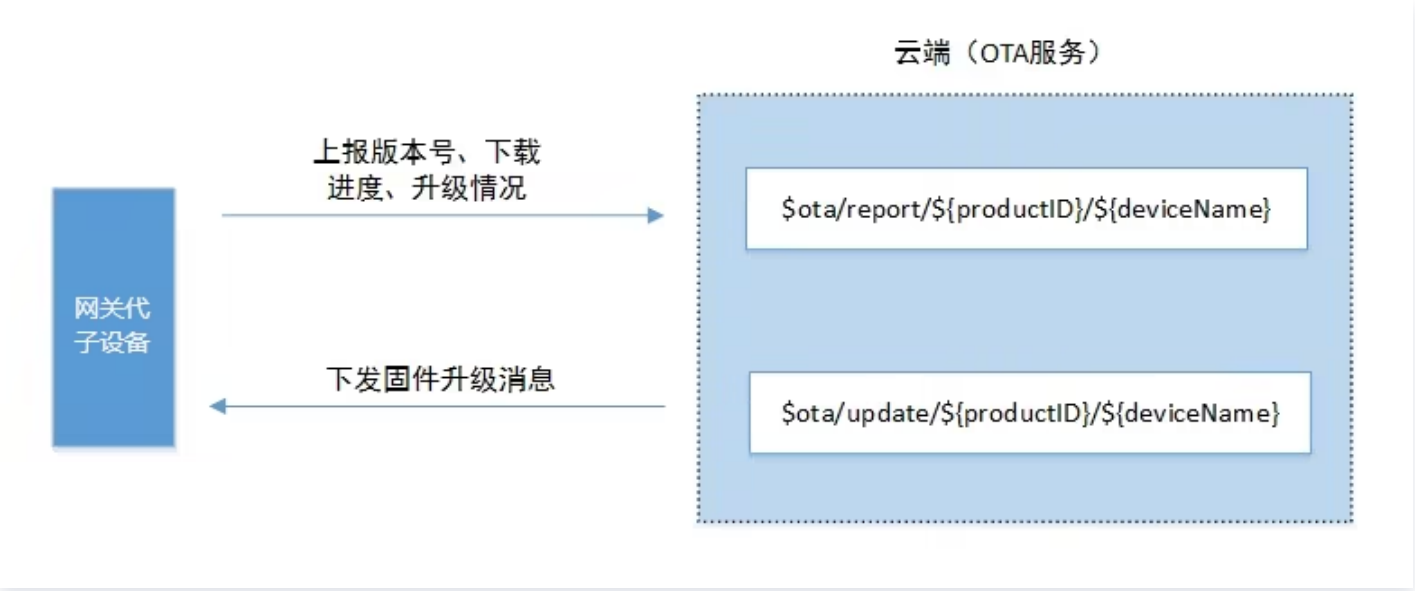
Last updated: 2025-03-19 14:57:15

Overview

When a subdevice under a gateway has new features available or vulnerabilities that need to be fixed, firmware update can be quickly performed for it through the device firmware update service.

How It Works

During the firmware upgrade process, the gateway needs to use the following two topics to communicate with the cloud on behalf of the sub-devices, as shown below:



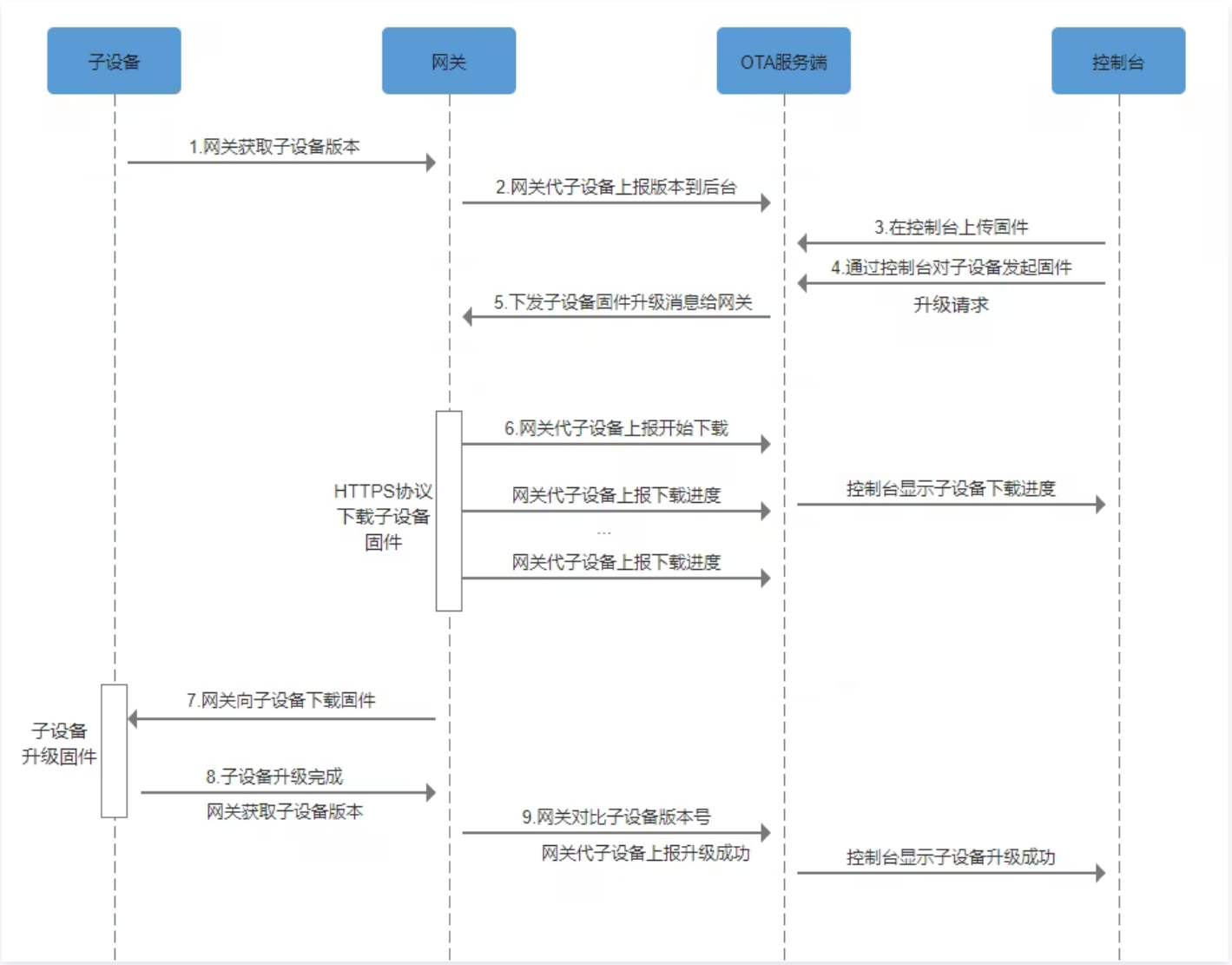
Below is the sample code:

```
$ota/report/${productID}/${deviceName}
For publishing (uplink) messages, through which the device reports the
version number and the download/upgrade progress to the cloud.
$ota/update/${productID}/${deviceName}
For subscribing (downlink) messages, through which the device receives
the upgrade message from the cloud.
```

Directions

Taking MQTT as an example, the update process of the subdevice is as follows:

Note:
For the specific directions of firmware update, please see Device Firmware Update.



Message Communication
MQTT Communication

Last updated：2025-03-19 14:57:39

Feature Overview

Devices connect to the IoT Hub platform via MQTT, using Topics for message communication to achieve device status monitoring, device data reporting, and device control features. This helps customer business platforms solve device monitoring and data concurrent processing issues, enhancing the reliability, high security, and high stability of the access platform.

Message Types

Device Status Monitoring

The device connects to the IoT Hub platform using the MQTT protocol. The IoT Hub platform monitors the MQTT connection status of the device to achieve online and offline status monitoring. Through the rule engine, the device status changes are notified to the customer business platform.
To enable customers' business platforms to receive real-time device online status, the IoT Hub platform provides device online status change notification capabilities. Simply select the product and device to monitor in the rule engine, and configure a device status change notification rule as follows. For detailed configuration, please refer to the Rule Engine guide

documentation.

Device Data Reporting

The device has successfully connected to the IoT Hub platform via MQTT. Select a Topic with **publish permissions** from the device's custom Topic list. The device successfully publishes messages to the IoT platform via MQTT. After receiving the message, the IoT platform will determine its validity and then forward the device-reported data to the customer's business platform or Tencent Cloud components as per the configured rule engine.
To add a custom Topic, you can operate in Product Details. For specifics, please refer to

Topic Management .

Device message delivery

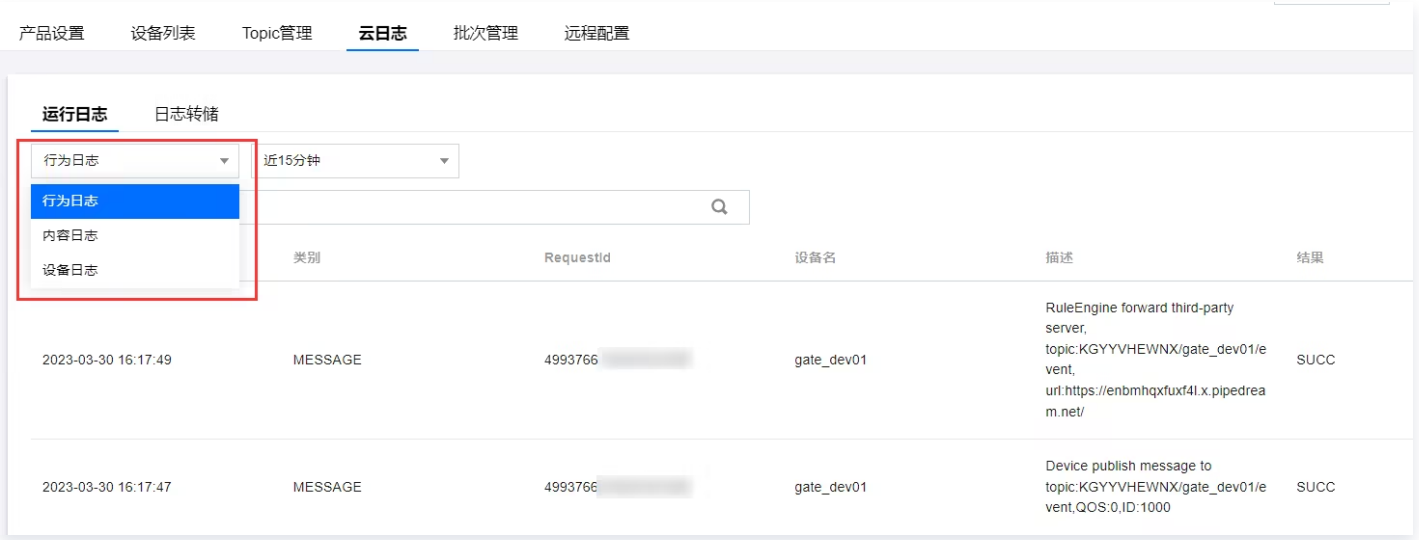
The device has successfully connected to the IoT Hub platform via MQTT. From the device's custom topic list, select a topic with **subscription permissions**. The device successfully subscribes to this topic via MQTT to receive messages from the IoT Hub platform. The customer's business platform uses TencentCloud API to publish messages to send control commands or configuration information to the successfully subscribed topic. After the IoT platform receives the message and verifies its legality, it will deliver the message to the device.

To add a custom topic, operate in the product details. Please refer to Topic Management for details.

After the device successfully subscribes to the topic, it will be displayed in the **Subscribed Topic List** as follows:

Troubleshooting

When the device fails to connect, the data reporting and message delivery are unsuccessful, and other anomalies occur, you can view and analyze them in the Cloud Logs under product details. Please refer to the Cloud Log usage feature for details.



RRPC Communication

Last updated: 2026-03-20 15:18:25

Feature Overview

Since the MQTT protocol is based on the publish/subscribe asynchronous communication mode, the server cannot synchronously perceive the returned result of the equipment after controlling it. To solve this problem, the IoT Hub platform uses RRPC (Revert RPC) to achieve a synchronous communication mechanism.

Communication principle

Communication Topic

- Subscription message topic: `$rrpc/rxd/${productID}/${deviceName}/+` is used to subscribe to RRPC request messages sent by the cloud (downstream).
- Request message topic: `$rrpc/rxd/${productID}/${deviceName}/${processID}` is used for the cloud to publish (downstream) RRPC request messages.
- Response message topic: `$rrpc/txd/${productID}/${deviceName}/${processID}` is used to publish (upstream) RRPC response messages.

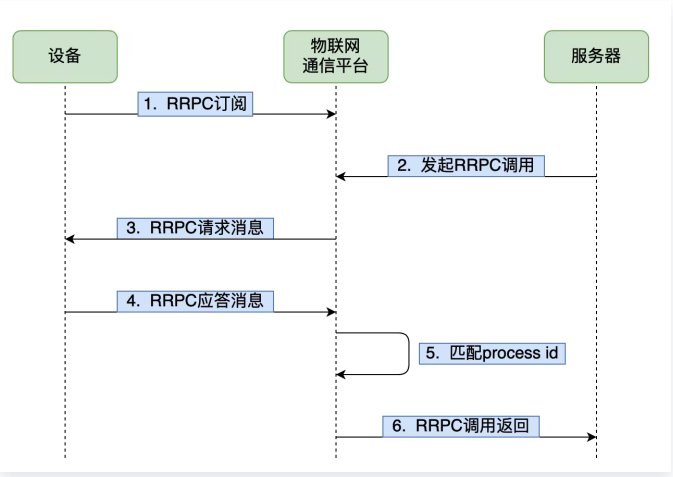
> **Note:**
> - `${productID}` : Product ID.
> - `${deviceName}` : Device name.
> - `${processID}` : A unique message ID generated by the server to identify different RRPC messages. The corresponding RRPC request message can be found through the processID carried in the RRPC response message.

Communication process

1. The device subscribes to the RRPC subscription message topic.
2. The server publishes an RRPC request message by calling the PublishRRPCMessage API.
3. After the device side receives the message, it extracts the processID from the request message Topic sent by the cloud. The device sets the processID of the response message Topic to the extracted processID and publishes the device's return message to the response message Topic.
4. After the IoT Hub receives the return message from the device side, it matches the message based on the processID and sends the device's return message to the server.

> ⚠ **Note:**
> **RRPC request timed out in 10s**, meaning the request is considered timed out if the device side does not respond within 10 seconds.

The flowchart is as follows:



RRPC communication sample

The example is based on the Linux platform using the device side C-SDK for connection, and the TencentCloud API Explorer tool for interface calls. The specific steps are as follows.

Creating device in console

Creating product and device

Please refer to Device Connection to create an air conditioning product and create an air conditioning device named airConditioner1.

Compiling and running demo (with key-authenticated device as example)

1. Compile the SDK

Modify `CMakeLists.txt` to ensure the following options are present:

```
set(BUILD_TYPE                    "release")
set(COMPILE_TOOLS                 "gcc")
set(PLATFORM                      "linux")
set(FEATURE_MQTT_COMM_ENABLED ON)
set(FEATURE_RRPC_ENABLED ON)
set(FEATURE_AUTH_MODE "KEY")
set(FEATURE_AUTH_WITH_NOTLS OFF)
```

```
set(FEATURE_DEBUG_DEV_INFO_USED  OFF)
```

Run the following script for compilation:

```
./cmake_build.sh
```

The demo output `rrpc_sample` is in the `output/release/bin` folder.

2. Enter the device information

Fill in the device information of the airConditioner1 device created above into the JSON file `aircond_device_info1.json`.

```
{
    "auth_mode":"KEY",
    "productId":"KL4J2****8",
    "deviceName":"airConditioner1",
    "key_deviceinfo":{
        "deviceSecret":"zOZXUaycuwleP****78dBA=="
    }
}
```

3. Run the `rrpc_sample` demo

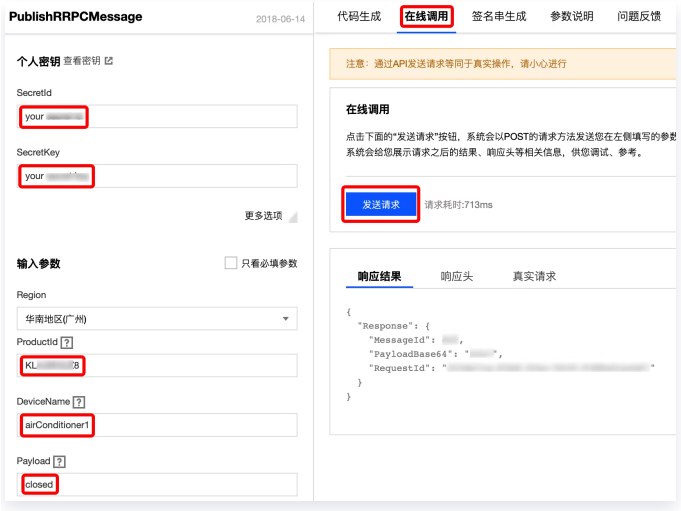
You can see that the device airConditioner1 has subscribed to the RRPC message and is in a waiting state.

```
./rrpc_sample -c ./aircond_device_info1.json -l 1000
INF|2020-08-03 23:57:55|qcloud_iot_device.c|iot_device_info_set(50):
SDK_Ver: 3.2.0, Product_ID: KL4J2****8, Device_Name: airConditioner1
DBG|2020-08-03 23:57:55|HAL_TLS_mbedtls.c|HAL_TLS_Connect(200): Setting
up the SSL/TLS structure...
DBG|2020-08-03 23:57:55|HAL_TLS_mbedtls.c|HAL_TLS_Connect(242):
Performing the SSL/TLS handshake...
DBG|2020-08-03 23:57:55|HAL_TLS_mbedtls.c|HAL_TLS_Connect(243):
Connecting to /KL4J2****8.iotcloud.tencentdevices.com/8883...
INF|2020-08-03 23:57:55|HAL_TLS_mbedtls.c|HAL_TLS_Connect(265):
connected with /KL4J2****8.iotcloud.tencentdevices.com/8883...
INF|2020-08-03 23:57:56|mqtt_client.c|IOT_MQTT_Construct(113): mqtt
connect with id: 2**** success
```

```
INF|2020-08-03 23:57:56|rrpc_sample.c|main(206): Cloud Device Construct
Success
DBG|2020-08-03
23:57:56|mqtt_client_subscribe.c|qcloud_iot_mqtt_subscribe(142):
topicName=$rrpc/rxd/KL4J2****8/airConditioner1/+|packet_id=****
INF|2020-08-03 23:57:56|rrpc_sample.c|_mqtt_event_handler(49): subscribe
success, packet-id=*****
DBG|2020-08-03 23:57:56|rrpc_client.c|_rrpc_event_callback(104): rrpc
topic subscribe success
```

4. Call TencentCloud API `PublishRRPCMessage` to send an RRPC request message

Open the Tencent Cloud API Console, fill in personal key and device parameter information, select online invocation, and send the request.



5. Observe the RRPC request message

Observe the print output of the device airConditioner1, and you can see that the RRPC request message has been received, `process id` is ***.

```
DBG|2020-08-04 00:07:36|rrpc_client.c|_rrpc_message_cb(85):
topic=$rrpc/rxd/KL4J2****8/airConditioner1/***
INF|2020-08-04 00:07:36|rrpc_client.c|_rrpc_message_cb(86): len=6,
topic_msg=closed
INF|2020-08-04 00:07:36|rrpc_client.c|_rrpc_get_process_id(76): len=3,
process id=***
INF|2020-08-04 00:07:36|rrpc_sample.c|_rrpc_message_handler(137): rrpc
message=closed
```

6. Observe the RRPC response message

Observe the print output of the device airConditioner1, and you can see that the RRPC request message has been processed and an RRPC response message has been replied, `process id` is ***.

```
DBG|2020-08-04
00:07:36|mqtt_client_publish.c|qcloud_iot_mqtt_publish(340): publish
packetID=0|topicName=$rrpc/txd/KL4J2****8/airConditioner1/***|payload=ok
```

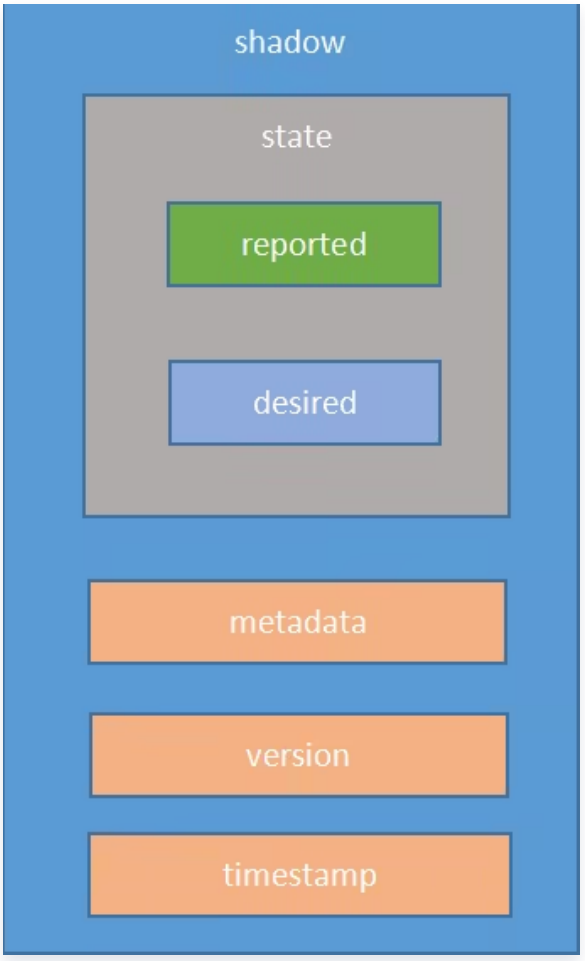
7. Observe the server response result

Observe the response result of the server, and you can see that the RRPC response message has been received. `MessageId` is ***, `Payload` is **** after being `Base64` -encoded, which is the same as the actual response message of the client after being `Base64` -encoded. At this point, it can be confirmed that the response message has been received.

Device Shadow
Device Shadow Introduction

Last updated: 2025-03-19 14:58:17

A device shadow document is a file of device status and configuration data cached by the server. It is stored as JSON text and consists of the following parts:



state

reported
Status reported by the storage device. Data is written to this document section via the Device Shadow Update protocol to report its new status. Applications can subscribe to receive device update reported shadow data through the Rule Engine, or proactively read this document section via the Get Device Shadow TencentCloud API to obtain the device status.
desired
Stores the data for the application to configure the device. Applications write data to this document to update configuration data via the Update Device Shadow TencentCloud API, and

devices can proactively fetch shadow data to the device through the Device Fetch Shadow protocol.

metadata

Device Shadow's metadata information, including the Last Updated Time of each Attribute Item in the state section, etc.

version

This is the version number of the device shadow document, which is increased each time the document is updated. The version number is maintained by Tencent Cloud on the backend, ensuring that the data of the device is consistent with that of the device shadow.

timestamp

The last update time of the device shadow document.

> **Note:**
> The device shadow document supports arrays. Only an entire array but not an element in the array can be updated, and none of the elements can be null.

Device Shadow Data Flow

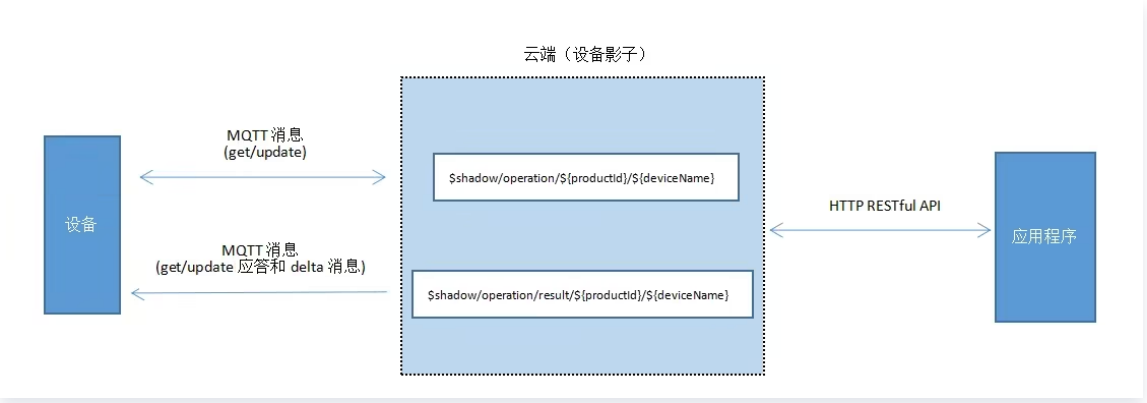
Last updated: 2025-03-19 14:58:31

Device Shadow Topic

A device shadow acts as an intermediary, allowing the device and your application to view the device status and update the device configuration. Communication between the device, application, and device shadow is implemented through two special topics:

- `$shadow/operation/${productId}/${deviceName}` : Used to publish (uplink) messages, enabling get/update operations on device shadow data.
- `$shadow/operation/result/${productId}/${deviceName}` : Used to subscribe to (downlink) messages. The shadow server sends response and push messages through this topic.

> **Note:**
> The above topics are created by the system by default when the device is created and will be automatically subscribed to within the device SDK.



Getting Shadow by Device

If the device wants to get the latest data of the device shadow, it needs to publish a GET message to the `$shadow/operation/${productId}/${deviceName}` topic. The message format is as follows:

```
{
    "type": "get",
    "clientToken": 8480
}
```

> **Note:**
> clientToken is a TOKEN used to uniquely identify the session business, generated by the device side and returned as is by the IoT Hub.

After receiving the request, the IoT Hub responds by publishing a message to the `$shadow/operation/result/${productId}/${deviceName}` topic. The format is as follows:

```
{
    "clientToken": "8480",
    "payload": {
        "state": {
            "delta": {
                "alarmvalue": "50"
            }
            "desired": {
                "alarmvalue": "50"
            }
            "reported": {
                "temperature": 32
            }
        }
        "timestamp": 1678783950012,
        "version": 14
    }
    "result": 0,
    "timestamp": 1678783991,
    "type": "get"
}
```

If there is a desired part in the device shadow document, the IoT Hub will automatically generate the corresponding delta part. If there is no desired part, there will be no content for desired and delta parts.

Updating Shadow by Device

> **Note:**
> The version in the message must match the one on the platform. If unsure, the version can be set to 0 to skip version verification.

The device updates the device shadow data by sending an update message to the `$shadow/operation/${productId}/${deviceName}` topic. The format is as follows:

```
{
    "type": "update",
    "state": {
        "reported": {
            "temperature": 35
        }
    }
    "version": 0,
    "clientToken": 5551
}
```

When the IoT Hub receives this message, it first determines whether the version in the message matches the version stored on the device shadow server. If so, the device shadow server will perform the device shadow update process.
After receiving the request, the IoT Hub responds by publishing a message to the `$shadow/operation/result/${productId}/${deviceName}` topic. At this time, the content field in the payload only includes the relevant content of this update. The format is as follows:

```
{
    "clientToken": "5551",
    "payload": {
        "state": {
            "reported": {
                "temperature": 35
            }
        }
        "timestamp": 1678785359634,
        "version": 15
    }
    "result": 0,
    "timestamp": 1678785359634,
    "type": "update"
}
```

If the version in the message does not match the version on the device shadow server, the IoT Hub platform will respond by sending the following message to `$shadow/operation/result/ABC1234567/AirConditioner`. At this time, the content in the payload will return the complete device shadow document.

```
{
    "clientToken": "8734",
    "payload": {
        "state": {
            "desired": {
                "alarmvalue": "50"
            }
            "reported": {
                "temperature": 35
            }
        }
        "timestamp": 1678785359634,
        "version": 15
    }
    "result": 5005,
    "timestamp": 1678785670967,
    "type": "update"
}
```

If the device wants to delete some fields in the reported section of the device shadow server, it can set these fields to null during the update, as shown below:

```
{
    "type": "update",
    "state": {
        "reported": {
            "temperature": null
        }
    }
    "version": 0,
    "clientToken": 6584
}
```

Updating Shadow by Application

> **Note:**
> The version in the message must match the one on the platform. If unsure, the version can be set to 0 to skip version verification.

User applications modify the desired field of the device shadow through the "Updating Shadow by Device" TencentCloud API to change shadow data and preconfigure devices.
When the IoT Hub receives this TencentCloud API request, it first checks whether the version in the request matches the version in the device shadow server. If they match, it proceeds with the device shadow update process and responds to the TencentCloud API request.
After the application successfully modifies the desired field of the device shadow, if the device is online at that moment, the IoT communication platform will send a delta message to the device through "$shadow/operation/result/ABC1234567/AirConditioner", informing it of the updated content of the device shadow; if the device is offline at that time, upon coming online, it can obtain the content of the desired field through "Getting Shadow Status by Device".

```
{
    "payload": {
        "state": {
            "alarmvalue": "50"
        }
        "timestamp": 1678786529510,
        "version": 17
    }
    "timestamp": 1678786529510,
    "type": "delta"
}
```

When the device receives a delta message, it clears the desired field content on the device shadow server by reporting a message with the desired field set to null. This is done by sending a message to the `$shadow/operation/${productId}/${deviceName}` topic in the following format:

```
{
    "type": "update",
    "state": {
        "desired": null
```

```
    }
    "version": 0,
    "clientToken": 5686
}
```

Error Codes

Error Codes	Description
5000	Incomplete parameters
5001	Shadow does not exist
5004	Illegal request parameter
5005	Version mismatch
5007	DB error
5008	Version conflict
5009	Illegal topic
5010	Shadow update failed
5011	Document size exceeds limit
5100	Internal Error

Device firmware update

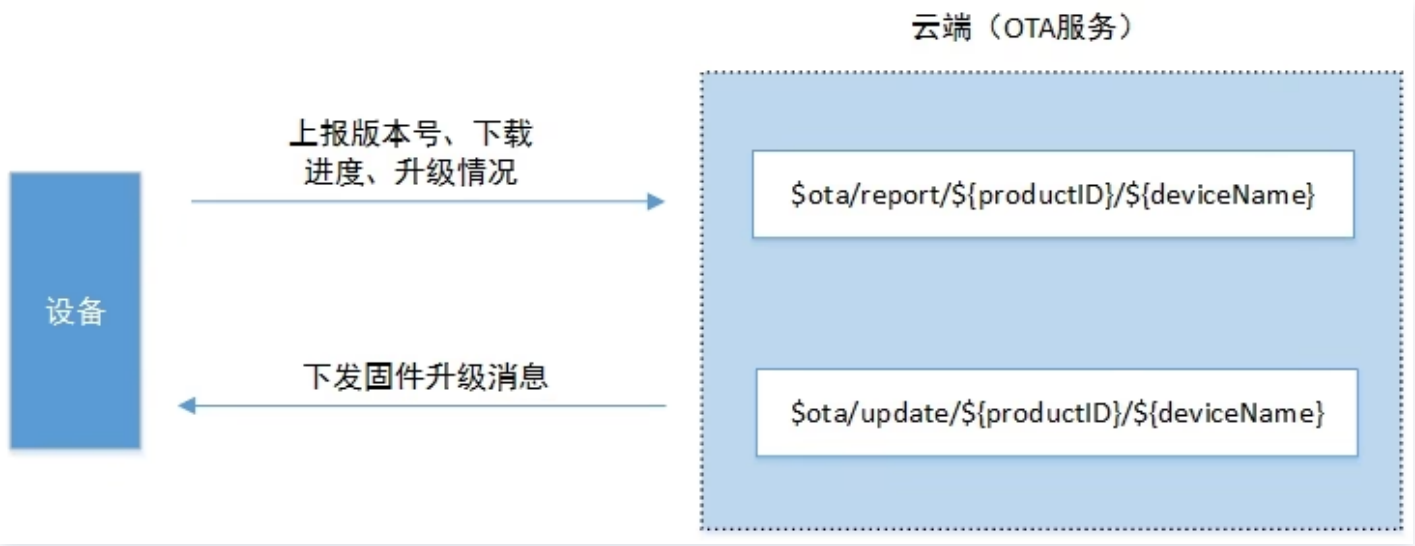
Last updated: 2025-03-19 14:58:42

Overview

Device firmware update is an important part of the IoT Hub service. When a device has new features available or vulnerabilities that need to be fixed, firmware update can be quickly performed for it through the device firmware update feature.

How It Works

During the firmware upgrade process, the device needs to subscribe to the following two topics to communicate with the cloud as shown below:

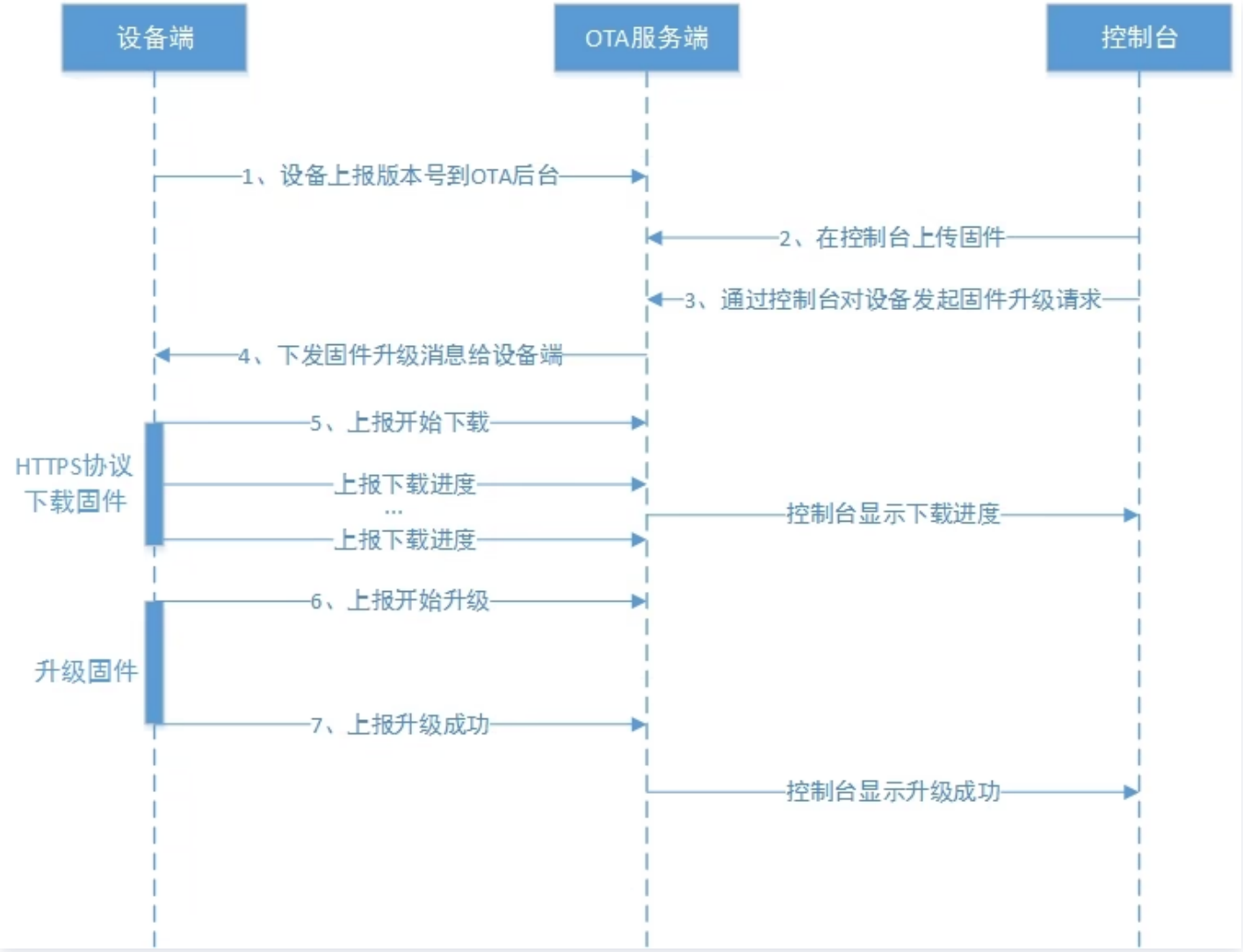


For example:

```
$ota/report/${productID}/${deviceName}
For publishing (uplink) messages, through which the device reports the
version number and the download/upgrade progress to the cloud.
$ota/update/${productID}/${deviceName}
For subscribing (downlink) messages, through which the device receives
the upgrade message from the cloud.
```

Directions

Taking MQTT as an example, the upgrade process of the device is as follows:



1. The device reports its current version number. It publishes a message in JSON format with the following content to the `$ota/report/${productID}/${deviceName}` topic over the MQTT protocol to report its version number:

```
{
        "type": "report_version",
        "report":{
                "version": "0.1"
        }
}
// type: message type
// version: the reported version number
```

2. Log in to the IoT Hub console, upload the firmware, and update the specified device to the specified version.

3. After the firmware update operation is triggered, the device will receive a firmware update message through the subscribed `$ota/update/${productID}/${deviceName}` topic. The content is as follows:

```
{
        "file_size": 708482,
        "md5sum": "36eb5951179db14a63****a37a9322a2",
        "type": "update_firmware",
        "url": "https://ota-1255858890.cos.ap-
guangzhou.myqcloud.com",
        "version": "0.2"
}
// type: the message type is update_firmware
// version: updated version
// url: URL of the downloaded firmware
// md5asum: MD5 value of the firmware
// file_size: firmware size in bytes
```

4. After receiving the firmware update message, the device will download the firmware from the URL. During the download process, the device SDK will keep reporting the download progress through the `$ota/report/${productID}/${deviceName}` topic. The reported content is as follows:

```
{
            "type": "report_progress",
            "report":{
                    "progress":{
                                "state":"downloading",
                                "percent":"10",
                                "result_code":"0",
                                "result_msg":""
                    }
                    "version": "0.2"
            }
}
// type: message type
// state: downloading
// percent: the current download progress in percentages
```

5. After downloading the firmware, the device needs to report an update start message through the topic `$ota/report/${productID}/${deviceName}` with the following content:

```
{
        "type": "report_progress",
        "report":{
                "progress":{
                            "state":"burning",
                            "result_code":"0",
                            "result_msg":""
                }
                "version": "0.2"
        }
}
// type: message type
// state: Status is burning in progress
```

6. After the device firmware upgrade is complete, report the upgrade success message to Topic `$ota/report/${productID}/${deviceName}`. The content is as follows:

```
{
        "type": "report_progress",
        "report":{
            "progress":{
                        "state":"done",
                        "result_code":"0",
                        "result_msg":""
            }
            "version": "0.2"
        }
}
// type: message type
// state: Status is completed
```

If the firmware download or upgrade fails, report the upgrade failure message through Topic `$ota/report/${productID}/${deviceName}`. The content is as follows:

```
{
     "type": "report_progress",
```

```
    "report":{
        "progress":{
                    "state":"fail",
                    "result_code":"-1",
                    "result_msg":"time_out"
        }
        "version": "0.2"
    }
}
// state: Failed
// result_code: error code. -1: download timed out; -2: the file does
not exist; -3: the signature expired; -4: MD5 mismatch; -5: firmware
update failed
// result_msg: error message
```

OTA Breakpoint Resume

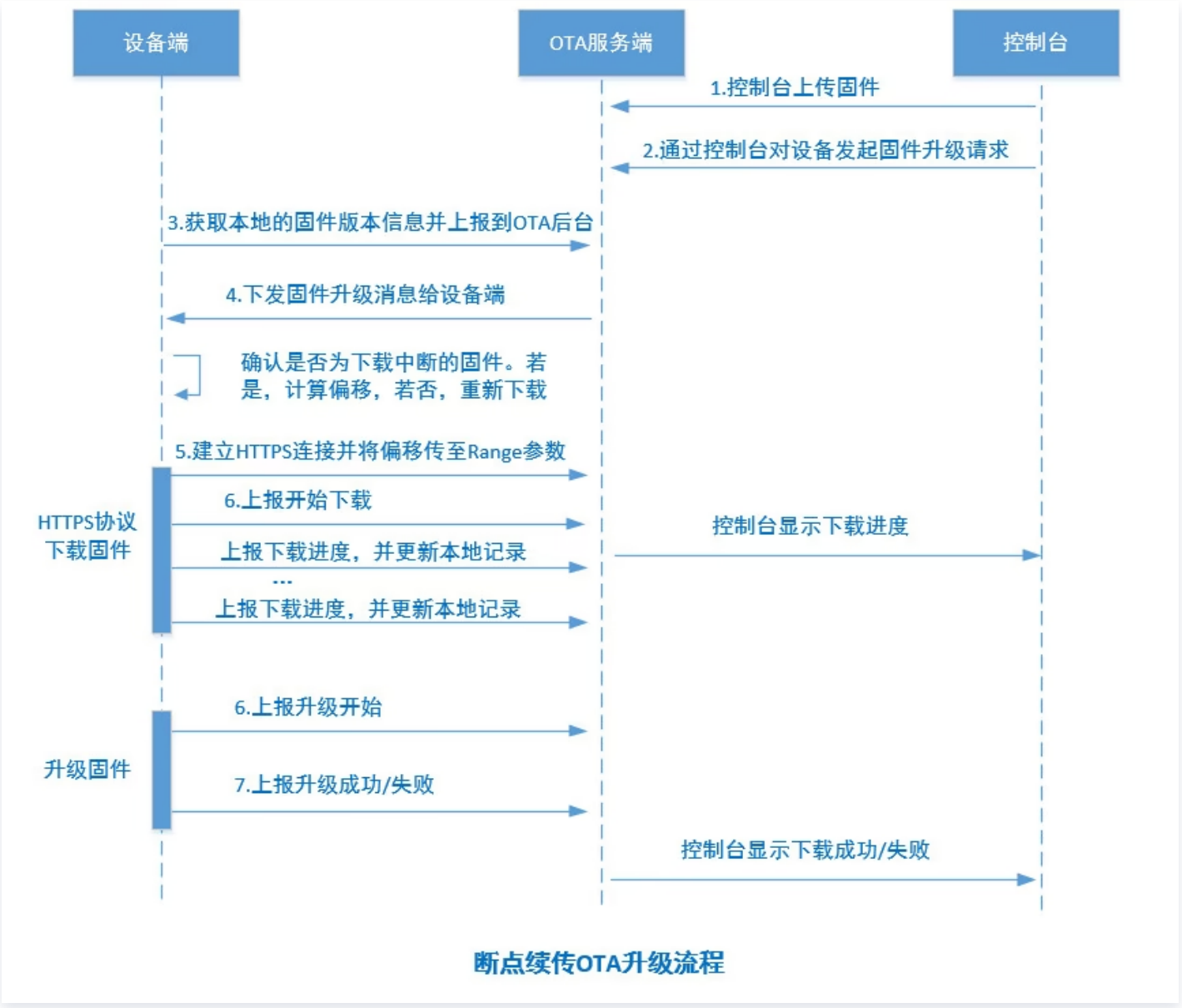
IoT devices may be in poor network environments in some scenarios, causing unstable connections. This may result in firmware download interruptions. If the firmware download always restarts from the offset 0, it may never complete in a poor network environment. Therefore, the firmware's breakpoint resume feature is particularly necessary. The specific explanation of the breakpoint resume is as follows.

- Checkpoint restart refers to resuming file download or upload from where interrupted. To implement this feature, the device needs to record the checkpoint where firmware download is interrupted as well as the MD5, file size, and version information of the firmware.
- In case of OTA interruption, the device will report its version number to the IoT Hub platform, and if the reported version number is different from the target version number to be updated to, the platform will distribute a firmware update message again, and the device will get the target firmware information and compare it with the locally recorded interrupted firmware information. After determining that they are the same firmware, the device will restart download from the checkpoint.

The OTA upgrade process with breakpoint resume is as follows:

Notes:

- Steps 3 – 6 in a poor network environment may be performed multiple times, and step 7 will be performed only after they succeed.
- After step 3 is performed, the device will receive the message that step 4 needs to be performed.



断点续传OTA升级流程

NTP Service

Last updated: 2025-03-19 14:58:55

Feature Overview

The NTP feature is mainly used to solve the problem with resource-constrained devices where they don't have the NTP service and thus have no accurate timestamps. The following two topics are required for this feature:

- Request Topic (for publishing): `$sys/operation/${ProductId}/${DeviceName}`.
- Response Topic (for subscribing): `$sys/operation/result/${ProductId}/${DeviceName}`.

How It Works

The IoT Hub platform draws on the principles of the NTP protocol and uses the platform itself as an NTP server. After a device requests the platform, the platform will return the NTP time. After the device receives the response, it will calculate the current accurate time based on the request time and receipt time.

Operations

1. The device sends a message in JSON format with the following content to `$sys/operation/${ProductId}/${DeviceName}` using the MQTT protocol. It requests the platform to issue NTP time and simultaneously records the request time as deviceSendtime:

```
{
      "type": "get",
      "resource": [
       "time"

}
```

2. The platform returns NTP time through `$sys/operation/result/${ProductId}/${DeviceName}`. Meanwhile, the device side records the Reception time deviceRecvtime. The returned message is in JSON format, as follows:

```
{
      "type": "get",
      "time": 1621562342,
      "ntptime1": 1621562342773,
      "ntptime2": 1621562342773
```

```
}
```

3. The precise time is calculated using the NTP time received from the device end (${ntptime1} + ${ntptime2}), the receiving time (${deviceRecvtime}), and the request time (${deviceSendtime}) together. The method is as follows:
Precise time = (${ntptime1} + ${ntptime2} + ${deviceRecvtime} – ${deviceSendtime}) / 2

Device Log Reporting

Last updated: 2025-03-19 14:59:07

Feature Overview

The device log feature is mainly used for the platform to remotely view the device operation logs. The platform can ask a device to report logs by sending a message. Log levels include ERROR, WARN, INFO, and DEBUG. The following two topics are needed for this feature:

- Data upstream topic (for publishing): `$log/operation/${productid}/${devicename}`.
- Data downstream topic (for subscribing): `$log/operation/result/${productid}/${devicename}`.

Querying Log Level

1. The device sends a message in JSON format with the following content to the `$log/operation/${productid}/${devicename}` topic to query whether it should upload logs and the required log level:

```
{
    "type": "get_log_level",
    "clientToken": "PPXLSKBUPZ-**"
}
```

2. The device actively queries whether to report logs, or the platform remotely notifies the device to start log reporting. The background sends a message in JSON format with the following content through `$log/operation/result/${productid}/${devicename}` to inform the device whether to start log reporting and the log level:

```
{
    "type": "get_log_level",
    "clientToken": "PPXLSKBUPZ-**",
    "log_level": 4,
    "result": 0,
    "timestamp": 1619599073
}
//log_level: 0: do not report logs; 1: ERROR; 2: WARN; 3: INFO; 4:
DEBUG
```

Log Upload

Parameter Description

When uploading device logs, it is required to carry ProductId and DeviceName to initiate a `http/https` request to the platform. The request interface and parameters are as follows:

- Requested URL: `http://ap-guangzhou.gateway.tencentdevices.com/device/reportlog` `https://ap-guangzhou.gateway.tencentdevices.com/device/reportlog`
- Request method: Post.

Request Parameters

Parameter Name	Required	Type	Description
ProductId	Yes	String	Product ID.
DeviceName	Yes	String	Device name.
Message	Yes	Array	Reported log content. It is a string array, and the log level needs to be added before each log entry. Currently, DBG, INF, ERR, and WRN are supported.

Note:
The interface only supports application/json format.

Signature generation

There are two types of signatures for request messages. Key authentication uses the HMAC-sha256 algorithm, and certificate authentication uses the RSA_SHA256 algorithm. For more information, please see Signature Method.

Platform response parameters

Parameter Name	Type	Description
RequestId	String	Request ID.

Sample Code

Request package

```
POST https://ap-guangzhou.gateway.tencentdevices.com/device/reportlog
Content-Type: application/json
Host: ap-guangzhou.gateway.tencentdevices.com
X-TC-Algorithm: HmacSha256
X-TC-Timestamp: 1551****65
X-TC-Nonce: 5456
X-TC-Signature:
2230eefd229f582d8b1b891af7107b91597240707d7****3738f756258d7652c
{"DeviceName":"AAAAAA","Message":["INFmqtt connect
success."],"ProductId":"G8N9****HB"}
```

Return package

```
{
    "Response":{
        "RequestId":"f4da4f1f-d72e-40f1-****-349fc0072ba0"
    }
}
```