IoT Hub

Device Connection Manual

Product Documentation

Contents

Device Connection Manual
Device Connection Overview

Last updated：2021-08-31 11:09:28

Feature Overview

To facilitate the connection of your devices and ensure the security of connection, IoT Hub provides a complete device connection service. To connect a device to IoT Hub, you need to complete device registration/creation first. The process of connection to IoT Hub be completed only after the device registration/creation succeeds.

Device connection service

- The device connection service provides the feature of dynamic device registration, so device registration can be completed by devices themselves.
- The device connection service supports connection over diverse protocols, including MQTT, WebSocket, HTTP/HTTPS, and CoAP.
- The device connection service is capable of connection authentication, so devices need to be authenticated based on the connection protocol to ensure the connection security.
- The device connection service offers device SDKs, based on which devices can be connected easily.

Device connection based on SDK

IoT Hub provides SDKs for C, Android, and Java for device connection. They are integrated with the features included in the device connection service, so you only need to set the device information (for key-authenticated devices: `ProductID` , `DeviceName` , and device key; for certificate-authenticated devices: `ProductID` , `DeviceName` , certificate file, key file, and CA Certificate) in them and integrate their corresponding features into your devices to complete device connection. In addition to the connection service features, the SDKs also include functional APIs for device shadow, OTA, and RRPC. For more information on the APIs, please see:

- SDK for C Use Instructions
- SDK for Android Use Instructions
- SDK for Java Use Instructions

> Note：
> IoT Hub supports custom connection. You can connect devices to it in a custom way simply by following the protocols and authentication processes it provides.

Connection Based on SDK for C

SDK for C Download

Last updated：2023-07-27 10:41:13

Code Hosting

- The code of the device SDK has been hosted on GitHub since v1.0.0
  https://github.com/tencentyun/qcloud-iot-sdk-embedded-c

- Download the latest version
  https://github.com/tencentyun/qcloud-iot-sdk-embedded-c/releases

  ### v3.2.1

- Release date: August 4, 2020

- Programming language: C

- Development environments: Linux/Windows

- Content:

  i. Added the RRPC sync communication feature and samples.
  ii. Added the broadcasting feature and samples.
  iii. Added the subdevice binding/unbinding APIs for gateway devices.
  iv. Updated the documentation.

v3.2.0

- Release date: April 30, 2020
- Programming language: C
- Development environments: Linux/Windows
- Content:
  i. Merged the MTMC branch code, supported multi-device connection, and optimized multithreaded APIs.
  ii. Fixed some potential memory leak and out-of-bounds issues as well as cross-platform compilation and running issues.
  iii. Used clang-format to format the code and introduced the code checkers clang-tidy and cpplint.

v3.1.3

- Release date: March 6, 2020
- Programming language: C
- Development environments: Linux/Windows
- Content:
  1. Optimized `ota_mqtt_sample` to decouple and separate the OTA process and the places where file operations were required, and added the checkpoint restart capability for the sample in case of MQTT reconnection.
  2. Optimized `gateway_sample` and added the sample code for proxying more than one subdevice.
  3. Added the API for querying whether the MQTT topic was subscribed to successfully.
  4. Optimized and updated the documentation.
  5. Fixed some compilation warnings and bugs.
  6. Unified the code indentation style.

v3.1.2

- Release date: November 11, 2019
- Programming language: C
- Development environments: Linux/Windows
- Content:
  1. Removed the relevant code and documentation for IoT Explorer to support IoT Hub only, and optimized the document descriptions.
  2. Fixed memory leaks in the OTA module, `device_info.json` file parsing issues, and Windows time format issues.
  3. Renamed `ca.c/h` to `qcloud_iot_ca.c/h` and `device.c/h` to `qcloud_iot_device.c/h` to avoid filename conflicts.

v3.1.0

- Release date: September 19, 2019
- Programming language: C
- Development environments: Linux/Windows
- Content:
  Refactored C-SDK:
  1. Optimized the code structure and directory hierarchy, used English comments, improved the documentation, and improved the usability and portability.
  2. Added the CMake compilation method and code extraction method on the basis of original Makefile compilation to adapt to multiple compilation environments.
  3. Added support for Windows to support development in Microsoft Visual Studio.

iv. Added the `AT_socket` network layer to support the development and porting of MCU+TCP AT module devices.
v. Added the porting adaptation for FreeRTOS + lwIP platforms.

v3.0.3

- Release date: August 26, 2019
- Programming language: C
- Development environments: Linux/GNU Make
- Content:
  i. Supported OTA checkpoint restart: added local firmware version information management (version, checkpoint, and MD5) in `ota_mqtt_sample.c`, and supported the `range` parameter when an HTTPS connection is established during firmware download.
  ii. Updated the SDK version number to v3.0.3.

v3.0.2

- Release date: July 18, 2019
- Programming language: C
- Development environments: Linux/GNU Make
- Content:
  i. Supported escape character processing for the string type in data templates.
  ii. Removed device version management from device shadow.
  iii. Optimized relevant examples of data templates.

v3.0.1

- Release date: June 11, 2019
- Programming language: C
- Development environments: Linux/GNU Make
- Content:
  i. Optimized the log reporting feature, introduced dynamic buffer memory allocation, and supported multipart log reporting for large logs in various scenarios.
  ii. Added the event handler callback of `subscribe` for MQTT to notify the status change of the subscribed topic timely.
  iii. Fixed some code issues, such as improper judgment on the return values of MQTT APIs.

v3.0.0

- Release date: May 17, 2019
- Programming language: C

- Development environments: Linux/GNU Make
- Content:
  1. Added the data template feature based on shadow.
  2. Added the event reporting feature.
  3. Added the data template code generation script tool.
  4. Fixed several bugs in JSON processing.
  5. Added data template samples, event samples, and smart light scenario samples in the data template.
  6. Adjusted the documentation structure and added the document directory `docs` and platform SDK use instructions.
  7. Supported both IoT Hub and IoT Explorer starting from v3.0.0.

v2.3.5

- Release date: May 15, 2019
- Programming language: C
- Development environments: Linux/GNU Make
- Content:
  1. Added the dynamic device registration feature.
  2. Added dynamic device registration samples.
  3. Added device information read/write HAL APIs.
  4. Added AES encryption and decryption APIs.
  5. Changed the device information acquisition method of all samples to implementation by APIs at the HAL layer.

v2.3.3

- Release date: May 6, 2019
- Programming language: C
- Development environments: Linux/GNU Make
- Content:
  1. Optimized the MQTT keepalive connection mechanism and ping request packet sending policy.
  2. Stored the topic names of MQTT subscription/unsubscription in the dynamic memory to make them easier to be called.
  3. Changed the maximum length of topic name to 128 for consistency with the cloud backend.
  4. Fixed the bugs with the acquisition of `sys` and `log` messages by HTTPC and MQTT.
  5. Optimized the error code types.

v2.3.2

- Release date: April 12, 2019
- Programming language: C

- Development environments: Linux/GNU Make
- Content:
  i. Fixed user experience issues: added the gateway compilation option (disabled by default) in `make.settings` and modified the firmware update print level.
  ii. Fixed the problem where the MQTT receiving buffer was prone to loss during shadow message downstreaming: added an error message when the receiving buffer was insufficient, and changed the default size of the MQTT sending/receiving buffer to 2,048 bytes.
  iii. Changed the maximum number of successfully subscribed topics to 10.

v2.3.1

- Release date: March 12, 2019
- Programming language: C
- Development environments: Linux/GNU Make
- Content:

1. Added the device log reporting feature in the SDK, making it easier for users to remotely monitor and diagnose the network status of devices in the console (only supported for the MQTT mode).
2. Streamlined the printout content of SDK logs, fixed several bugs, and optimized the code design.
3. Changed the maximum length of device name to 48 characters for consistency with the IoT Hub console.

v2.3.0

- Release date: February 25, 2019
- Programming language: C
- Development environments: Linux/GNU Make
- Content:

1. Added the gateway feature to allow gateway devices to connect/disconnect and send/receive messages on behalf of subdevices based on the MQTT protocol.
2. Optimized the thread safety design for multithreaded applications and added multithreaded routines and precautions in the samples.
3. Optimized the MQTT reconnection mechanism and heartbeat packet timer refresh policy.
4. Fixed several bugs and added validity checks for some memory operations.
5. Removed the bit field operation mode from some structures to reduce cross-platform errors.

v2.2.0

- Release date: July 20, 2018
- Programming language: C
- Development environments: Linux/GNU Make
- Content:

  i. Added the NB-IoT device connection capability.

  ii. Adapted to the topic wildcards `#` and `+`.

  iii. Organized the directory structure of third-party libraries.

  iv. Fixed several bugs.

v2.1.0

- Release date: May 2, 2018
- Programming language: C
- Development environments: Linux/GNU Make
- Content:

  i. Added the new firmware update capability (over the OTA-CoAP channel).
  ii. Added the HMAC-SHA1 connection authentication capability for low-end resource-constrained devices.
  iii. Added the capability to get backend time.

v2.0.0

- Release date: March 12, 2018
- Programming language: C
- Development environments: Linux/GNU Make
- Content:

i. Added the new firmware update capability (over the OTA-MQTT channel).
ii. Fixed the issue where the device shadow heartbeat interval was invalid.
iii. Fixed the issue where the data received by MQTT caused buffer overflow when the data length was at the threshold.

v1.2.2

- Release date: February 7, 2018
- Programming language: C
- Development environments: Linux/GNU Make
- Content:

i. Added support for MQTT/CoAP symmetric encryption connection.
ii. Optimized the Linux C compilation.

v1.2.1

- Release date: February 2, 2018
- Programming language: C
- Development environments: Linux/GNU Make
- Content: fixed the incorrect logic of message publishing timeout callback.

v1.2.0

- Release date: January 17, 2018
- Programming language: C
- Development environments: Linux/GNU Make
- Content:
 1. Modified the message publishing/subscribing ACKs for receipt through the callback without blocking the sending thread.
 2. Added the capabilities of devices and the backend for connection and logging.
 3. Added the new UDP-based CoAP channel which used DTLS asymmetric encryption and consumed less power in pure data reporting scenarios.

v1.0.0

- Release date: November 15, 2017
- Programming language: C

- Development environments: Linux/GNU Make
- Content:

1. Added support for the MQTT protocol: devices could quickly and easily connect to the cloud server of IoT Hub.

For more information, please see MQTT Protocol Details.

2. Added support for device shadow: for more information, please see Device Shadow Details.
3. Added support for symmetric and asymmetric encryption.

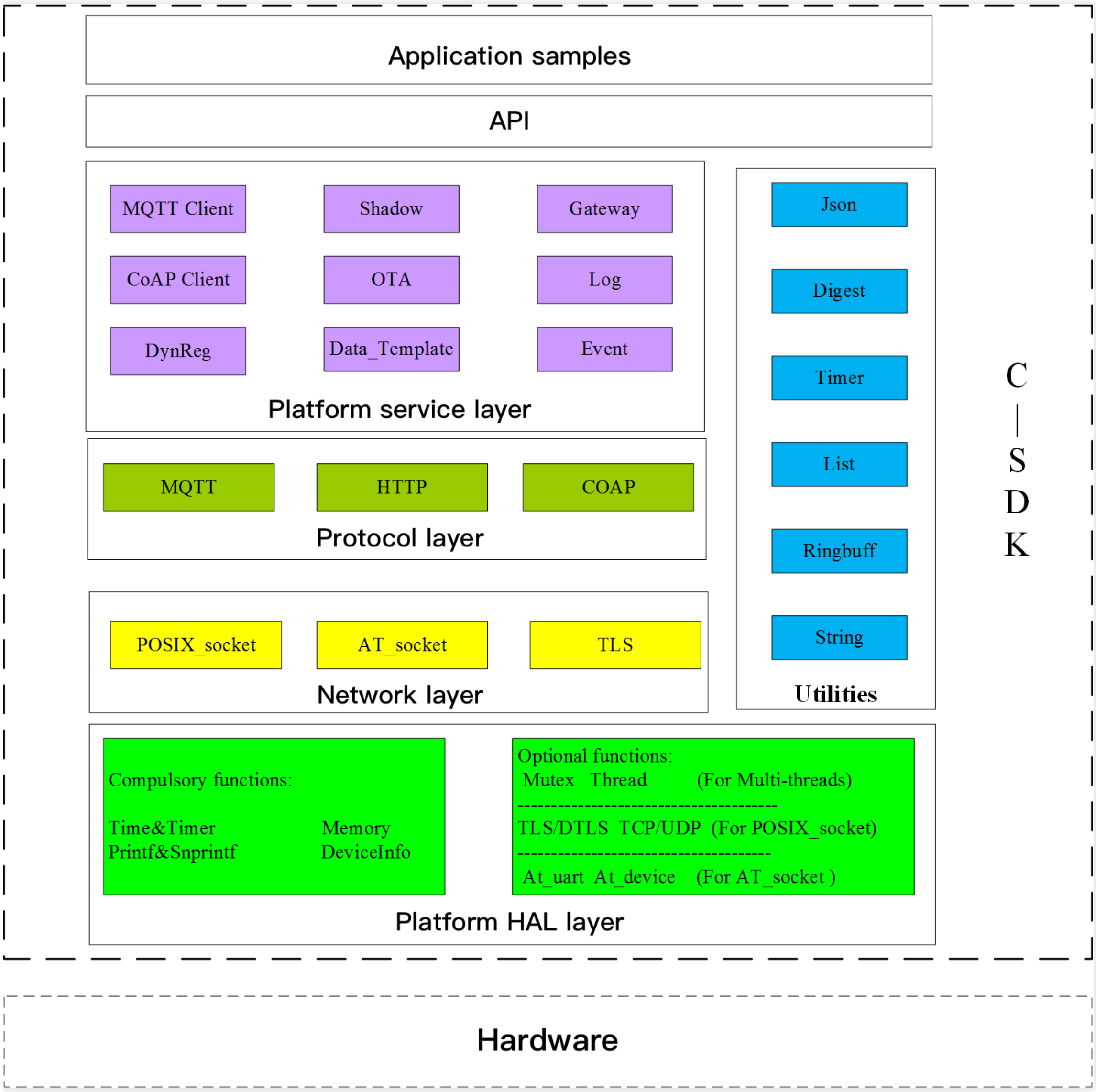
SDK for C Cross-Platform Porting Overview

Last updated：2023-07-27 10:41:13

This document describes how to port the device C-SDK to the target hardware platform. C-SDK adopts modular design to separate the core protocol service and hardware abstraction layer (HAL). When porting across platforms, you generally only need to modify and adapt the HAL.

C-SDK Architecture

Architecture diagram



Architecture description

The SDK is designed into four layers from top to bottom: platform service layer, core protocol layer, network layer, and hardware abstraction layer.

- **Service layer**
  This layer is above the network protocol layer and implements features such as device connection authentication, device shadow, gateway, dynamic registration, log reporting, and OTA.
- **Protocol layer**
  The network protocols over which devices can interact with the IoT Hub platform include MQTT, CoAP, and HTTP.
- **Network layer**
  This layer implements network protocol stacks based on TLS/SSL (TLS/DTLS), POSIX_socket (TCP/UDP), and AT_socket. Different services can use different protocol stack API functions as needed.
- **Hardware abstraction layer**
  To implement the abstract encapsulation of underlying operations of different hardware platforms, it is necessary to conduct porting for the specific software and hardware platforms, which is divided into two parts of required and optional HAL APIs.

HAL Porting

HAL mainly has several major parts for porting, including those related to the OS, network and TLS, time and print, and device information.
In the **platform/os** directory, the SDK demonstrates the implementation of HAL in four scenarios: Linux, Windows, FreeRTOS, and nonOS. You can refer to the corresponding directory to port for the target platform.

OS APIs

No.	Function	Description
1	HAL_Malloc	Dynamically applies for memory block
2	HAL_Free	Releases memory block
3	HAL_ThreadCreate	Creates thread
4	HAL_ThreadDestroy	Terminates thread
5	HAL_MutexCreate	Creates mutex lock
6	HAL_MutexDestroy	Terminates mutex lock
7	HAL_MutexLock	Locks mutex

No.	Function	Description
8	HAL_MutexUnlock	Unlocks mutex
9	HAL_SemaphoreCreate	Creates semaphore
10	HAL_SemaphoreDestroy	Terminates semaphore
11	HAL_SemaphoreWait	Waits for semaphore
12	HAL_SemaphorePost	Releases semaphore
13	HAL_SleepMs	Sleeps

Network and TLS HAL APIs

Network APIs provide either-or adaptation and porting. For devices that have network communication capabilities and integrate TCP/IP network protocol stacks, you need to implement the `POSIX_socket` network HAL APIs. For devices using TLS/SSL for encrypted communication, you also need to implement the TLS HAL APIs. For devices with **MCU + universal TCP_AT module**, you can choose the `AT_Socket` framework provided by the SDK and implement relevant AT module APIs.

HAL APIs based on POSIX_socket

Among them, TCP/UDP APIs are implemented based on POSIX socket functions. TLS APIs are dependent on the **mbedtls** library. Before porting, you must ensure that the **mbedtls** library is available on the system. If you use other TLS/SSL libraries, please refer to the relevant implementation of **`platform/tls/mbedtls`** for porting and adapting.
UDP/DTLS functions need to be ported only when **CoAP** communication is enabled.

No.	Function	Description
1	HAL_TCP_Connect	Establishes TCP connection
2	HAL_TCP_Disconnect	Closes TCP connection
3	HAL_TCP_Write	Writes data to TCP connection
4	HAL_TCP_Read	Reads data from TCP connection
5	HAL_TLS_Connect	Establishes TLS connection
6	HAL_TLS_Disconnect	Closes TLS connection
7	HAL_TLS_Write	Writes data to TLS connection

No.	Function	Description
8	HAL_TLS_Read	Reads data from TLS connection
9	HAL_UDP_Connect	Establishes UDP connection
10	HAL_UDP_Disconnect	Closes UDP connection
11	HAL_UDP_Write	Writes data to UDP connection
12	HAL_UDP_Read	Reads data from UDP connection
13	HAL_DTLS_Connect	Establishes DTLS connection
14	HAL_DTLS_Disconnect	Closes DTLS connection
15	HAL_DTLS_Write	Writes data to DTLS connection
16	HAL_DTLS_Read	Reads data from DTLS connection

HAL APIs based on AT_socket

After `AT_socket` is selected by enabling the compilation macro **`AT_TCP_ENABLED`** , the SDK will call the `at_socket` API of `network_at_tcp.c` . You don't need to port the `at_socket` layer, but you need to implement the AT serial port driver and AT module driver. For the AT module driver, you only need to implement the driver API of the driver structure `at_device_op_t` in `at_device` of the AT framework. You can refer to the supported modules in the `at_device` directory. For the AT serial port driver, you need to implement serial port receipt interruption and then call the callback function `at_client_uart_rx_isr_cb` in the interruption service program. You can refer to `HAL_AT_UART_freertos.c` to port for the target platform.

No.	Function	Description
1	HAL_AT_Uart_Init	Initializes AT serial port
2	HAL_AT_Uart_Deinit	Deinitializes AT serial port
3	HAL_AT_Uart_Send	Sends data over AT serial port
4	HAL_AT_UART_IRQHandler	Handles AT serial port receipt interruption

Time and print HAL APIs

No.	Function	Description
1	HAL_Printf	Writes formatted data to standard output stream

No.	Function	Description
2	HAL_Snprintf	Writes formatted data to string
3	HAL_UptimeMs	Retrieves the number of milliseconds that elapsed since the system has started
4	HAL_DelayMs	Blocking delay in milliseconds

Device information HAL APIs

To connect a device to the IoT Hub platform, you need to create product and device information on the platform and save such information in a non-volatile storage medium on the device. You can refer to `platform/os/linux/HAL_Device_linux.c` for implementation.

No.	Function	Description
1	HAL_GetDevInfo	Reads device information
2	HAL_SetDevInfo	Saves device Information

FreeRTOS + lwIP Platform Porting Description

Last updated：2023-07-27 10:41:13

This document describes how to port IoT Hub C-SDK to the **FreeRTOS + lwIP** platform.

FreeRTOS Porting Overview

As a micro-kernel system, FreeRTOS mainly provides core OS mechanisms such as task creation and scheduling and inter-task communication. Different device platforms also should be equipped with different software components before they can form a complete embedded operating platform, including C runtime libraries (such as Newlib or ARM CMSIS library) and TCP/IP network protocol stacks (such as lwIP). In addition, the compilation and development environments vary by device platform, so when porting C-SDK, you need to adapt it according to the specific conditions of different devices.

> Note：
> The SDK provides a reference implementation based on **FreeRTOS + lwIP + Newlib** in `platform/os/freertos` , which has been verified and tested on Espressif's ESP8266 platform.

Code Extraction

Because different RTOS-based platforms have different compilation methods, it is generally impossible to directly use the SDK's CMake or Make to compile. Therefore, the SDK provides the code extraction feature. It allows you to extract the relevant code into a separate folder based on your needs. The code hierarchy in the folder is concise, making it easy for you to copy and integrate it into your own development environment.

1. Change the platform in `CMakeLists.txt` to FreeRTOS and enable the code extraction feature:

```
set(BUILD_TYPE "release")
set(PLATFORM "freertos")
set(EXTRACT_SRC ON)
set(FEATURE_AT_TCP_ENABLED OFF)
```

2. Run the following command on Linux:

```
mkdir build
cd build
cmake ..
```

3. You can find the relevant code files in `output/qcloud_iot_c_sdk` with the following directory hierarchy:

```
qcloud_iot_c_sdk
├── include
│   ├── config.h
│   ├── exports
├── platform
└── sdk_src
└── internal_inc
```

Note：

- `include` directory: contains the SDK APIs and variable parameters, where `config.h` is the compilation macros generated according to the compilation options. For more information, please see API and Variable Parameter Description.
- `platform` directory: contains platform-related code, which can be modified and adapted according to the specific conditions of the device. For more information on functions, please see Overview.
- `sdk_src` directory: contains the SDK core logic and protocol-related code, which generally don't need to be modified, where `internal_inc` is the header file used internally by the SDK.

4. You can copy `qcloud_iot_c_sdk` to the compilation and development environment of your target platform and then modify the compilation options as needed.

Porting Sample

Build a demo project based on Espressif's ESP8266 RTOS platform in the Linux development environment.

1. Please refer to ESP8266_RTOS_SDK to obtain the RTOS_SDK and cross compiler and create a project.
2. Copy the `qcloud_iot_c_sdk` directory extracted above to `components/qcloud_iot`.
3. In `components/qcloud_iot`, create a compilation configuration file `component.mk` with the following content:

```
#
# Component Makefile
#
COMPONENT_ADD_INCLUDEDIRS := \
qcloud_iot_c_sdk/include \
qcloud_iot_c_sdk/include/exports \
qcloud_iot_c_sdk/sdk_src/internal_inc
COMPONENT_SRCDIRS := \
qcloud_iot_c_sdk/sdk_src \
qcloud_iot_c_sdk/platform
```

At this point, you can compile `qcloud_iot_c_sdk` as a component and then call the IoT Hub C-SDK APIs in your code to connect devices and send/receive messages.

MCU + Universal TCP_AT Module Porting (FreeRTOS)

Last updated：2023-07-27 10:41:13

For MCUs that have no network communication capabilities, the "MCU + communication module" combination is often used. Communication modules (including Wi-Fi/2G/4G/NB-IoT) generally provide serial port-based AT instruction protocols for MCUs to communicate over the network. For this scenario, the C-SDK encapsulates the AT-socket network layer, where the core protocol and service layer don't need to be ported. This document describes how to port C-SDK for connection to IoT Hub in the target environment of MCU (FreeRTOS) + universal TCP AT module.

SDK Download

Download the latest version of the device C-SDK.

SDK Feature Configuration

Use the general TCP module to compile and configure the options as follows:

Name	Configuration	Description
BUILD_TYPE	debug/release	Set as needed
EXTRACT_SRC	ON	Enable code extraction
COMPILE_TOOLS	gcc/MSVC	Set as needed and ignore in case of IDE
PLATFORM	Linux/Windows	Set as needed and ignore in case of IDE
FEATURE_OTA_COMM_ENABLED	ON/OFF	Set as needed
FEATURE_AUTH_MODE	KEY	Key authentication is recommended for resource-constrained devices
FEATURE_AUTH_WITH_NOTLS	ON/OFF	Enable TLS as needed
FEATURE_EVENT_POST_ENABLED	ON/OFF	Enable event reporting as needed
FEATURE_AT_TCP_ENABLED	ON	Whether to enable TCP feature in AT module

Name	Configuration	Description
FEATURE_AT_UART_RECV_IRQ	ON	Whether to enable receipt interruption feature in AT module
FEATURE_AT_OS_USED	ON	Whether to enable multithreaded feature in AT module
FEATURE_AT_DEBUG	OFF	The AT module debugging feature is disabled by default, and it needs to be enabled during debugging

Code Extraction

1. Run the following command on Linux:

```
mkdir build
cd build
cmake ..
```

2. You can find the relevant code files in `output/qcloud_iot_c_sdk` with the following directory hierarchy:

```
qcloud_iot_c_sdk
├── include
│   ├── config.h
│   ├── exports
├── platform
└── sdk_src
└── internal_inc
```

Note：

- `include` directory: contains the SDK APIs and variable parameters, where `config.h` is the compilation macros generated according to the compilation options.
- `platform` directory: contains platform-related code, which can be modified and adapted according to the specific conditions of the device.
- `sdk_src` directory: contains the SDK core logic and protocol-related code, which generally don't need to be modified, where `internal_inc` is the header file used internally by the SDK.

3. You can copy `qcloud_iot_c_sdk` to the compilation and development environment of your target platform and then modify the compilation options as needed.

HAL Porting

Please refer to Overview to port first.

For network HAL APIs, the `AT_Socket` framework provided by the SDK has been selected through the above compilation options. The SDK will call the `at_socket` API of `network_at_tcp.c`. You don't need to port the `at_socket` layer, but you need to implement the AT serial port driver and AT module driver. For the AT module driver, you only need to implement the driver API of the driver structure `at_device_op_t` in `at_device` of the AT framework. You can refer to the supported modules in the `at_device` directory.

Currently, the SDK provides underlying API implementation for the Wi-Fi module ESP8266, which is widely used in the IoT field, for reference when you port to other communication modules.

Business Logic Development

You can refer to the routines in the SDK's `samples` directory for development.

MCU + Universal TCP_AT Module Porting (nonOS)

Last updated：2023-07-27 10:41:13

For MCUs that have no network communication capabilities, the "MCU + communication module" combination is often used. Communication modules (including Wi-Fi/2G/4G/NB-IoT) generally provide serial port-based AT instruction protocols for MCUs to communicate over the network. For this scenario, the C-SDK encapsulates the AT-socket network layer, where the core protocol and service layer don't need to be ported. This document describes how to port C-SDK for connection to IoT Hub in the target environment of MCU (nonOS) + universal TCP AT module.

Compared with the RTOS scenario, the network data received by `at_socket` is processed differently. The application layer needs to periodically call **`IOT_MQTT_Yield`** to receive the server's downstream data. If the receipt window is missed, there will be data loss. Therefore, in scenarios with complex business logic, we recommended you use RTOS and select the nonOS mode by configuring `FEATURE_AT_OS_USED = OFF` .

SDK Download

Download the latest version of the device C-SDK.

SDK Feature Configuration

Use the general TCP module to compile and configure the options for nonOS as follows:

Name	Configuration	Description
BUILD_TYPE	debug/release	Set as needed
EXTRACT_SRC	ON	Enable code extraction
COMPILE_TOOLS	gcc/MSVC	Set as needed and ignore in case of IDE
PLATFORM	Linux/Windows	Set as needed and ignore in case of IDE
FEATURE_OTA_COMM_ENABLED	ON/OFF	Set as needed
FEATURE_AUTH_MODE	KEY	Key authentication is recommended for resource-constrained devices
FEATURE_AUTH_WITH_NOTLS	ON/OFF	Enable TLS as needed

Name	Configuration	Description
FEATURE_EVENT_POST_ENABLED	ON/OFF	Enable event reporting as needed
FEATURE_AT_TCP_ENABLED	ON	Enable `at_socket` component
FEATURE_AT_UART_RECV_IRQ	ON	Enable AT serial port receipt interruption
FEATURE_AT_OS_USED	OFF	Use `at_socket` component in environment without RTOS
FEATURE_AT_DEBUG	OFF	The AT module debugging feature is disabled by default, and it needs to be enabled during debugging

Code Extraction

1. Run the following command on Linux:

```
mkdir build
cd build
cmake ..
```

2. You can find the relevant code files in `output/qcloud_iot_c_sdk` with the following directory hierarchy:

```
qcloud_iot_c_sdk
├── include
|    ├── config.h
|    ├── exports
├── platform
└── sdk_src
└── internal_inc
```

Note：

- `include` directory: contains the SDK APIs and variable parameters, where `config.h` is the compilation macros generated according to the compilation options.
- `platform` directory: contains platform-related code, which can be modified and adapted according to the specific conditions of the device.

> - `sdk_src` directory: contains the SDK core logic and protocol-related code, which generally don't need to be modified, where `internal_inc` is the header file used internally by the SDK.

3. You can copy `qcloud_iot_c_sdk` to the compilation and development environment of your target platform and then modify the compilation options as needed.

HAL Porting

Please refer to Overview first.

For network HAL APIs, the `AT_Socket` framework provided by the SDK has been selected through the above compilation options. The SDK will call the `at_socket` API of `network_at_tcp.c` . You don't need to port the `at_socket` layer, but you need to implement the AT serial port driver and AT module driver. For the AT module driver, you only need to implement the driver API of the driver structure `at_device_op_t` in `at_device` of the AT framework. You can refer to the supported modules in the `at_device` directory. For the AT serial port driver, you need to implement serial port receipt interruption and then call the callback function `at_client_uart_rx_isr_cb` in the interruption service program. You can refer to `HAL_OS_nonos.c` to port for the target platform.

Business Logic Development

You can refer to the routines in the SDK's `samples` directory for development.

SDK for C Connection Description

Last updated：2023-07-27 10:41:13

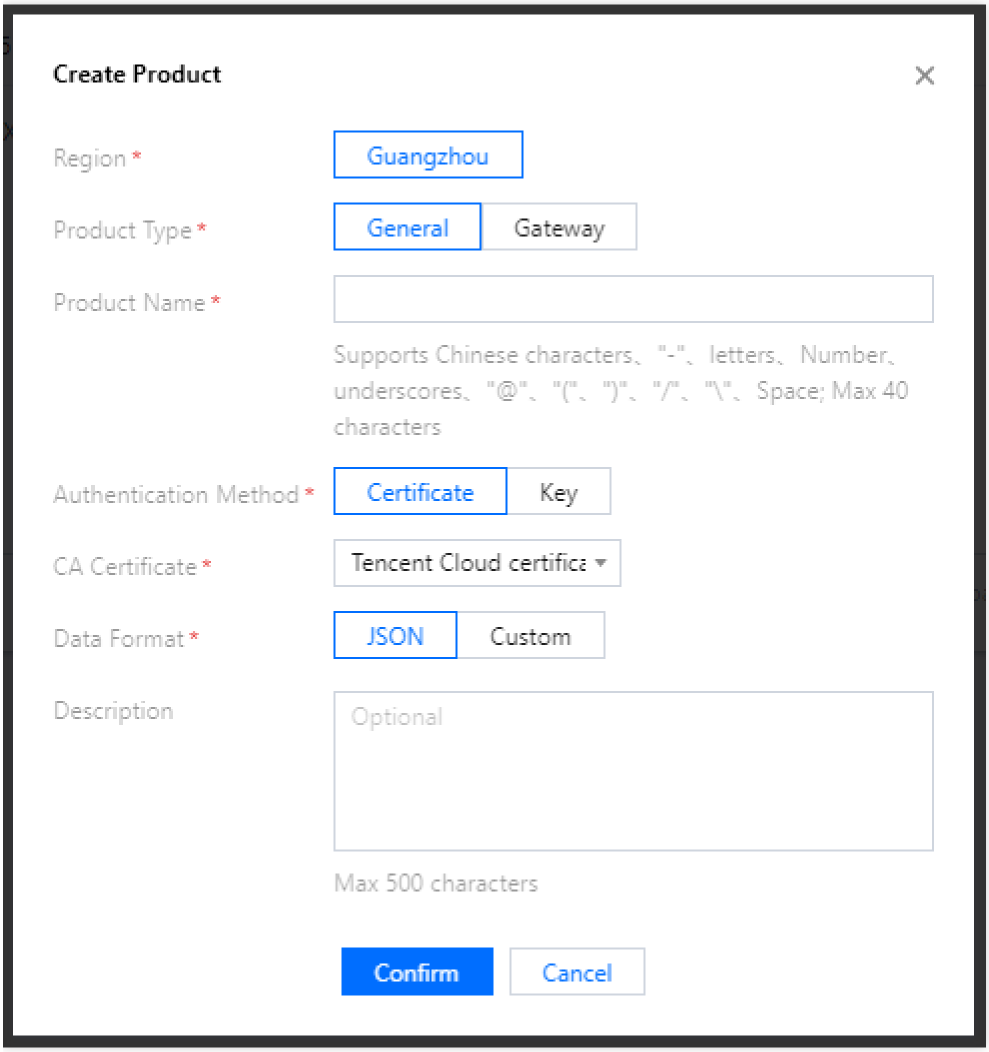
To ensure security, IoT Hub verifies the validity of each connected device. For this reason, it provides multiple authentication methods to meet the needs for connection of devices with different resources in different use cases.

Device Identity Information

Depending on the form of device key, devices are divided into certificate-authenticated devices and key-authenticated devices. Certificate authentication is more secure, but it consumes more software and hardware resources.

- Certificate-authenticated devices must carry the following four pieces of information before it can pass the authentication by the platform: product ID (ProductId), device name (DeviceName), device certificate (DeviceCert), and device private key (DevicePrivateKey), among which, the certificate and private key files are generated by the platform and correspond to each other.
- Key-authenticated devices must carry the following three pieces of information before it can pass the authentication by the platform: product ID (ProductId), device name (DeviceName), and device key (DeviceSecret), among which, the device key is generated by the platform.

The device key is determined by setting the authentication method during product creation as shown below:



Device Identity Information Burning

Device information burning is divided into preset burning and dynamic burning, which differ in terms of convenience and security.

Preset burning

After a product is created, you can create devices one by one in the IoT Hub console or through TencentCloud API, get their corresponding device information, and burn the above three or four pieces of information into a non-volatile medium in a specific step of device production, so that the device SDK can read the stored device information during running for device authentication.

Dynamic burning

- Preset burning: this involves performing personalized production actions in the mass production process and thus affects the production efficiency. To improve the ease of use, the platform supports dynamic burning. This feature is implemented as follows: after a product is created, its dynamic registration feature can be used to generate a product key (ProductSecret). Unified product information can be burned for all devices under it in the production process, i.e., product ID (ProductId) and product key (ProductSecret). After the devices are shipped, the device identity information can be obtained through dynamic registration and then saved, and then obtained three or four pieces of information can be used for device authentication.
- Device name (DeviceName) generation for dynamic burning: if automatic device creation is used during dynamic registration, device names can be generated by devices themselves, which are generally device IMEIs or MAC addresses but must be unique under the same product ID (ProductId). If automatic device creation is not used during dynamic registration, device names should be entered on the platform in advance, and the platform will verify whether the requested device names are validly entered during dynamic device registration. This can reduce the security risks in case of product key leakage.

> Note：
> For dynamic registration, you should ensure the security of the product key (ProductSecret); otherwise, major security risks may arise.

Programming for Authenticating Preset Burnt Devices

Writing device information

For certificate-authenticated devices, implement the following HAL APIs:

HAL_API	Description
HAL_SetProductID	Sets the product ID, which must be stored on a non-volatile storage medium
HAL_SetDevName	Sets the device name, which must be stored on a non-volatile storage medium
HAL_SetDevCertName	Sets the device certificate file name. The certificate file should be placed in the `certs` directory
HAL_SetDevPrivateKeyName	Sets the device private key file name. The private key file should be placed in the `certs` directory

For key-authenticated devices, implement the following HAL APIs:

HAL_API	Description

HAL_API	Description
HAL_SetProductID	Sets the product ID, which must be stored on a non-volatile storage medium
HAL_SetDevName	Sets the device name, which must be stored on a non-volatile storage medium
HAL_SetDevSec	Sets the device key, which must be stored on a non-volatile storage medium. We recommend you encrypt and scramble it

Getting device information

For certificate-authenticated devices, implement the following HAL APIs:

HAL_API	Description
HAL_GetProductID	Gets product ID
HAL_GetDevName	Gets device name
HAL_GetDevCertName	Gets device certificate file name
HAL_GetDevPrivateKeyName	Gets device certificate private key file name

For key-authenticated devices, implement the following HAL APIs:

HAL_API	Description
HAL_GetProductID	Gets product ID
HAL_GetDevName	Gets device name
HAL_GetDevSec	Gets device key. If it is encrypted and scrambled during write, it should be decrypted and descrambled during read

Application demos

- Initialize the connection parameters

```
static DeviceInfo sg_devInfo;

static int _setup_connect_init_params(MQTTInitParams* initParams)
{
int ret;

ret = HAL_GetDevInfo((void *)&sg_devInfo);
```

```
if(QCLOUD_ERR_SUCCESS != ret){
return ret;
}

initParams->device_name = sg_devInfo.device_name;
initParams->product_id = sg_devInfo.product_id;
......
}
```

- Get the device information

```
int HAL_GetDevInfo(void *pdevInfo)
{
int ret;
DeviceInfo *devInfo = (DeviceInfo *)pdevInfo;

memset((char *)devInfo, 0, sizeof(DeviceInfo));
ret = HAL_GetProductID(devInfo->product_id, MAX_SIZE_OF_PRODUCT_ID);
ret |= HAL_GetDevName(devInfo->device_name, MAX_SIZE_OF_DEVICE_NAME);

#ifdef AUTH_MODE_CERT
ret |= HAL_GetDevCertName(devInfo->devCertFileName, MAX_SIZE_OF_DEVICE_CERT_FILE_
NAME);
ret |= HAL_GetDevPrivateKeyName(devInfo->devPrivateKeyFileName, MAX_SIZE_OF_DEVIC
E_KEY_FILE_NAME);
#else
ret |= HAL_GetDevSec(devInfo->devSerc, MAX_SIZE_OF_DEVICE_SERC);
#endif

if(QCLOUD_ERR_SUCCESS != ret){
Log_e("Get device info err");
ret = QCLOUD_ERR_DEV_INFO;
}

return ret;
}
```

- Generate the authentication parameters

```
static int _serialize_connect_packet(unsigned char *buf, size_t buf_len, MQTTConn
ectParams *options, uint32_t *serialized_len) {
......
......
int username_len = strlen(options->client_id) + strlen(QCLOUD_IOT_DEVICE_SDK_APPI
```

```
D) + MAX_CONN_ID_LEN + cur_timesec_len + 4;
options->username = (char*)HAL_Malloc(username_len);
get_next_conn_id(options->conn_id);
HAL_Snprintf(options->username, username_len, "%s;%s;%s;%ld", options->client_id,
QCLOUD_IOT_DEVICE_SDK_APPID, options->conn_id, cur_timesec);

#if defined(AUTH_WITH_NOTLS) && defined(AUTH_MODE_KEY)
if (options->device_secret != NULL && options->username != NULL) {
char sign[41] = {0};
utils_hmac_sha1(options->username, strlen(options->username), sign, options->devi
ce_secret, options->device_secret_len);
options->password = (char*) HAL_Malloc (51);
if (options->password == NULL) IOT_FUNC_EXIT_RC(QCLOUD_ERR_INVAL);
HAL_Snprintf(options->password, 51, "%s;hmacsha1", sign);
}
#endif
......
}
```

Programming for Authenticating Dynamically Burnt Devices

- Determine whether to initiate a dynamic request

```
int main(int argc, char **argv) {
......
memset((char *)&sDevInfo, 0, sizeof(DeviceInfo));
ret = HAL_GetProductID(sDevInfo.product_id, MAX_SIZE_OF_PRODUCT_ID);
ret |= HAL_GetProductKey(sDevInfo.product_key, MAX_SIZE_OF_PRODUCT_KEY);
ret |= HAL_GetDevName(sDevInfo.device_name, MAX_SIZE_OF_DEVICE_NAME); // Dynamic
registration. We recommend you use a unique identifier of the device as the devic
e name, such as chip ID or IMEI

#ifdef AUTH_MODE_CERT
ret |= HAL_GetDevCertName(sDevInfo.devCertFileName, MAX_SIZE_OF_DEVICE_CERT_FILE_
NAME);
ret |= HAL_GetDevPrivateKeyName(sDevInfo.devPrivateKeyFileName, MAX_SIZE_OF_DEVIC
E_KEY_FILE_NAME);
if(QCLOUD_ERR_SUCCESS != ret){
Log_e("Get device info err");
return QCLOUD_ERR_FAILURE;
}
/*You need to modify the logic for empty device information based on your own pro
duct conditions. Here is only a sample*/
```

```
if(!strcmp(sDevInfo.devCertFileName, QCLOUD_IOT_NULL_CERT_FILENAME)
||!strcmp(sDevInfo.devPrivateKeyFileName, QCLOUD_IOT_NULL_KEY_FILENAME)){
Log_d("dev Cert not exist!");
infoNullFlag = true;
}else{
Log_d("dev Cert exist");
}
#else
ret |= HAL_GetDevSec(sDevInfo.devSerc, MAX_SIZE_OF_PRODUCT_KEY);
if(QCLOUD_ERR_SUCCESS != ret){
Log_e("Get device info err");
return QCLOUD_ERR_FAILURE;
}
/*You need to modify the logic for empty device information based on your own pro
duct conditions. Here is only a sample*/
if(!strcmp(sDevInfo.devSerc, QCLOUD_IOT_NULL_DEVICE_SECRET)){
Log_d("dev psk not exist!");
infoNullFlag = true;
}else{
Log_d("dev psk exist");
}
#endif
......

}
```

- Initiate a dynamic request and save the requested device information

```
/*The device information is empty. Initiate device registration. Note: after succ
essful device registration and connection, registration cannot be initiated agai
n, so please save the device information properly*/
if(infoNullFlag){
if(QCLOUD_ERR_SUCCESS == qcloud_iot_dyn_reg_dev(&sDevInfo)){

ret = HAL_SetDevName(sDevInfo.device_name);
#ifdef AUTH_MODE_CERT
ret |= HAL_SetDevCertName(sDevInfo.devCertFileName);
ret |= HAL_SetDevPrivateKeyName(sDevInfo.devPrivateKeyFileName);
#else
ret |= HAL_SetDevSec(sDevInfo.devSerc);
#endif
if(QCLOUD_ERR_SUCCESS != ret){
Log_e("devices info save fail");
}else{
#ifdef AUTH_MODE_CERT
```

```
Log_d("dynamic register success, productID: %s, devName: %s, CertFile: %s, KeyFil
e: %s", \
sDevInfo.product_id, sDevInfo.device_name, sDevInfo.devCertFileName, sDevInfo.dev
PrivateKeyFileName);
#else
Log_d("dynamic register success,productID: %s, devName: %s, devSerc: %s", \
sDevInfo.product_id, sDevInfo.device_name, sDevInfo.devSerc);
#endif
}
}else{
Log_e("%s dynamic register fail", sDevInfo.device_name);
}
}
```

After the device information is dynamically requested successfully, the preset burning feature will be completed. The subsequent authentication process is the same as that with preset burning.

SDK for C Use Instructions
Usage Overview

Last updated：2023-07-27 10:41:13

IoT Hub device SDK for C relies on a secure and powerful data channel to enable IoT developers to quickly connect devices to the cloud for two-way communication.

> Note：
> After v3.1.0, the SDK refactored and optimized the compilation environment, code, and directory structure, increasing the availability and portability.

Scope of Application of SDK for C

Featuring a modular design, the SDK for C separates the core protocol service from the hardware abstraction layer and provides flexible configuration options and multiple compilation methods, making it suitable for development platforms and use environments of different devices.

Network communication-capable devices on Linux/Windows

- For devices that have network communication capabilities and run on standard Linux/Windows, such as PCs, servers, and gateway devices, as well as advanced embedded devices such as Raspberry Pi, you can directly compile and run the SDK on them.
- For embedded Linux devices that require cross compilation, if the toolchain of the development environment has `glibc` or similar libraries which can provide system calls, including socket communication, SELECT sync IO, dynamic memory allocation, functions for getting time/sleeping/generating random number/printing, as well as critical data protection such as the mutex mechanism (only when multiple threads are required), only simple adaptation (e.g., changing the cross compiler settings in `CMakeLists.txt` or `make.settings`) is required before the SDK can be compiled and run.

Network communication-capable devices on RTOS

- For IoT devices that have network communication capabilities and run on RTOS, the SDK for C needs to be adapted to different RTOS systems for porting. Currently, it has been adapted to multiple IoT-oriented RTOS platforms, including FreeRTOS, RT-Thread, and TencentOS tiny.
- When porting the SDK to an RTOS device, if the platform provides C runtime libraries like `Newlib` and embedded TCP/IP protocol stacks like lwIP, adaptation for porting can be done easily.

Devices with MCU + communication module

- For MCUs that have no network communication capabilities, the "MCU + communication module" combination is often used. Communication modules (including Wi-Fi/2G/4G/NB-IoT) generally provide serial port-based AT instruction protocols for MCUs to communicate over the network. For this scenario, the SDK for C encapsulates the AT-socket network layer, where the core protocol and service layer don't need to be ported. In addition, it provides FreeRTOS-based and nonOS HAL implementation methods.
- In addition, IoT Hub provides a dedicated AT instruction set. If the communication module implements this instruction set, it will be easier for devices to connect and communicate, and less code will be required. For this scenario, please refer to the SDK for MCU AT dedicated to the Tencent Cloud customized AT module.

SDK Directory Structure Overview

The directory structure and top-level documents are described as follows:

Name	Description
CMakeLists.txt	CMake compilation and description file
CMakeSettings.json	CMake configuration file on Visual Studio
cmake_build.sh	Compilation script with CMake on Linux
make.settings	Configuration file compiled directly by Makefile on Linux
Makefile	Direct compilation with Makefile on Linux
device_info.json	Device information file. If `DEBUG_DEV_INFO_USED` = `OFF` , the device information will be parsed from this file
docs	Documentation directory, i.e., the use instructions of the SDKs for different platforms
external_libs	Third-party package components, such as Mbed TLS
samples	Application demos
include	External header files provided to users
platform	Platform source code files. Currently, implementations are provided for different OS (Linux/Windows/FreeRTOS/nonOS), TLS (Mbed TLS), and AT module
sdk_src	Core communication protocols and service code of the SDK
tools	Compilation and code generation script tools supporting the SDK

SDK Compilation Method Description

The SDK for C supports three compilation methods:

- CMake
- Makefile
- Code extraction

For more information on the compilation methods and compilation configuration options, please see Compilation Configuration Description and Compilation Environment (Linux and Windows).

SDK Demos

The `samples` directory of the SDK for C contains demos showing how to use the features. For more information on how to run the demos, please see the corresponding documents in the SDK documentation directory.
For more information on device connection to and message sending/receiving in IoT Hub over MQTT, please see Getting Started with MQTT.

Notes

API changes for OTA update

Starting from SDK v3.0.3, OTA update supports checkpoint restart. When the firmware download process is interrupted due to network exceptions or other issues, the downloaded part of the firmware can be saved, so that the download can start from where interrupted instead of from the beginning when it is resumed.
After this new feature was supported, the methods of using relevant OTA APIs changed. If you have upgraded from v3.0.2 or below, you should modify your logic code; otherwise, firmware download will fail. For more information on how to modify it, please see `samples/ota/ota_mqtt_sample.c` .

Code name changes

To improve the code readability and comply with the naming conventions, SDK v3.1.0 incorporated changes to certain variables, functions, and macro names. If you have upgraded from v3.0.3 or below, you can run the `tools/update_from_old_SDK.sh` script on Linux to replace the names in your own code, and then you can use the new version of the SDK directly.

Old Name	New Name
QCLOUD_ERR_SUCCESS	QCLOUD_RET_SUCCESS

Old Name	New Name
QCLOUD_ERR_MQTT_RECONNECTED	QCLOUD_RET_MQTT_RECONNECTED
QCLOUD_ERR_MQTT_MANUALLY_DISCONNECTED	QCLOUD_RET_MQTT_MANUALLY_DISC
QCLOUD_ERR_MQTT_CONNACK_CONNECTION_ACCEPTED	QCLOUD_RET_MQTT_CONNACK_CONN
QCLOUD_ERR_MQTT_ALREADY_CONNECTED	QCLOUD_RET_MQTT_ALREADY_CONN
MAX_SIZE_OF_DEVICE_SERC	MAX_SIZE_OF_DEVICE_SECRET
devCertFileName	dev_cert_file_name
devPrivateKeyFileName	dev_key_file_name
devSerc	device_secret
MAX_SIZE_OF_PRODUCT_KEY	MAX_SIZE_OF_PRODUCT_SECRET
product_key	product_secret
DEBUG	eLOG_DEBUG
INFO	eLOG_INFO
WARN	eLOG_WARN
ERROR	eLOG_ERROR
DISABLE	eLOG_DISABLE
Log_writter	IOT_Log_Gen
qcloud_iot_dyn_reg_dev	IOT_DynReg_Device
IOT_SYSTEM_GET_TIME	IOT_Get_SysTime

Compilation Configuration Description

Last updated：2023-07-27 10:41:13

This document describes the compilation methods and compilation configuration options of the SDK for C, as well as the compilation environment setup and compilation samples in the Linux and Windows development environments.

SDK for C Compilation Method Description

The SDK for C supports the following compilation methods.

CMake

- We recommend you use CMake, a cross-platform compilation tool, for compilation in the Linux and Windows development environments.
- Compilation with CMake uses `CMakeLists.txt` as the input file for compilation configuration options.

Makefile

- For environments that don't support CMake, Makefile can be used for compilation.
- As for SDK v3.0.3 or below, compilation with Makefile uses `make.settings` as the input file for compilation configuration options, and you only need to run `make` after the modification.

Code extraction

- This method allows you to select features based on your needs and extract the relevant code into a separate folder. The code hierarchy in the folder is concise, making it easy for you to copy and integrate it into your own development environment.
- This method relies on CMake. Configure relevant features in `CMakeLists.txt` , set `EXTRACT_SRC` to `ON` , and run the following command on Linux:

```
mkdir build
cd build
cmake ..
```

- You can find the relevant code files in `output/qcloud_iot_c_sdk` with the following directory hierarchy:

```
qcloud_iot_c_sdk
├── include
```

```
|    ├── config.h
|    ├── exports
├── platform
└── sdk_src
└── internal_inc
```

- The `include` directory contains the SDK APIs and variable parameters, where `config.h` is the compilation macros generated according to the compilation options.
- The `platform` directory contains platform-related code, which can be modified and adapted according to the specific conditions of the device.
- The `sdk_src` directory contains the SDK core logic and protocol-related code, which generally don't need to be modified, where `internal_inc` is the header file used internally by the SDK.

> Note：
> You can copy `qcloud_iot_c_sdk` to the compilation and development environment of your target platform and then modify the compilation options as needed.

SDK for C Compilation Option Description

Compilation configuration options

Most of the following configuration options apply to CMake and `make.setting` . The `ON` value in CMake corresponds to `y` in `make.setting` , and `OFF` to `n` .

Name	CMake Value	Description
BUILD_TYPE	release/debug	release: disable the `IOT_DEBUG` information (the compilation is output to the `release` directory). debug: enable the `IOT_DEBUG` information (the compilation is output to the `debug` directory).
EXTRACT_SRC	ON/OFF	Whether to enable code extraction, which takes effect only for CMake.

Name	CMake Value	Description
COMPILE_TOOLS	gcc	GCC and MSVC are supported. It can also be a cross compiler, such as `arm-none-linux-gnueabi-gcc` .
PLATFORM	Linux	Includes Linux/Windows/FreeRTOS/nonOS.
FEATURE_MQTT_COMM_ENABLED	ON/OFF	Whether to enable MQTT channel.
FEATURE_MQTT_DEVICE_SHADOW	ON/OFF	Whether to enable device shadow.
FEATURE_COAP_COMM_ENABLED	ON/OFF	Whether to enable CoAP channel.
FEATURE_GATEWAY_ENABLED	ON/OFF	Whether to enable gateway feature.
FEATURE_OTA_COMM_ENABLED	ON/OFF	Whether to enable OTA firmware update.
FEATURE_OTA_SIGNAL_CHANNEL	MQTT/CoAP	OTA signaling channel type.
FEATURE_AUTH_MODE	KEY/CERT	Connection authentication method.
FEATURE_AUTH_WITH_NOTLS	ON/OFF	OFF: TLS enabled; ON: TLS disabled.
FEATURE_DEV_DYN_REG_ENABLED	ON/OFF	Whether to enable dynamic device registration.
FEATURE_LOG_UPLOAD_ENABLED	ON/OFF	Whether to enable log reporting.
FEATURE_EVENT_POST_ENABLED	ON/OFF	Whether to enable event reporting.
FEATURE_DEBUG_DEV_INFO_USED	ON/OFF	Whether to enable device information source acquisition.
FEATURE_SYSTEM_COMM_ENABLED	ON/OFF	Whether to enable backend time acquisition.
FEATURE_AT_TCP_ENABLED	ON/OFF	Whether to enable TCP feature in AT module.
FEATURE_AT_UART_RECV_IRQ	ON/OFF	Whether to enable receipt interruption feature in AT module.
FEATURE_AT_OS_USED	ON/OFF	Whether to enable multithreaded feature in AT module.

Name	CMake Value	Description
FEATURE_AT_DEBUG	ON/OFF	Whether to enable debugging feature in AT module.
FEATURE_MULTITHREAD_TEST_ENABLED	ON/OFF	Whether to compile the Linux multithreaded test routine.

There is a dependency relationship between the configuration options. A configuration option is valid only when the value of its dependent option is valid as shown below:

Name	Dependent Option	Valid Value
FEATURE_MQTT_DEVICE_SHADOW	FEATURE_MQTT_COMM_ENABLED	ON
FEATURE_GATEWAY_ENABLED	FEATURE_MQTT_COMM_ENABLED	ON
FEATURE_OTA_SIGNAL_CHANNEL(MQTT)	FEATURE_OTA_COMM_ENABLED FEATURE_MQTT_COMM_ENABLED	ON ON
FEATURE_OTA_SIGNAL_CHANNEL(COAP)	FEATURE_OTA_COMM_ENABLED FEATURE_COAP_COMM_ENABLED	ON ON
FEATURE_AUTH_WITH_NOTLS	FEATURE_AUTH_MODE	KEY
FEATURE_AT_UART_RECV_IRQ	FEATURE_AT_TCP_ENABLED	ON
FEATURE_AT_OS_USED	FEATURE_AT_TCP_ENABLED	ON
FEATURE_AT_DEBUG	FEATURE_AT_TCP_ENABLED	ON

Device information options

After a device is created in the IoT Hub console, you need to configure its information (`ProductID/DeviceName/DeviceSecret/Cert/Key` file) in the SDK first before it can run properly. In the development phase, the SDK provides two methods of storing the device information:

- If the device information is stored in the code (compilation option `DEBUG_DEV_INFO_USED` = `ON`), you should modify the device information in `platform/os/xxx/HAL_Device_xxx.c` . This method can be used on platforms without a file system.
- If the device information is stored in the configuration file (compilation option `DEBUG_DEV_INFO_USED` = `OFF`), you should modify the device information in the `device_info.json` file with no need to recompile the SDK. This method is recommended for development on Linux and Windows.

Compilation Environment (Linux and Windows)

Last updated：2023-07-27 10:41:13

Linux (Ubuntu)

> Note：
> The Ubuntu version used for demonstration in this document is v16.04.

1. **Install the necessary software**
   The SDK requires CMake v3.5 or above. The CMake version installed by default is low. If compilation fails, download and install the specific version of CMake as instructed in Installation Instructions.

```
$ sudo apt-get install -y build-essential make git gcc cmake
```

2. **Modify the configuration**
   Modify the `CMakeLists.txt` file in the root directory of the SDK and make sure that the following options exist (with a key-authenticated device as example):

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

3. **Run the script for compilation**

4. Below is a complete compilation library and demo:

```
./cmake_build.sh
```

5. The output library files, header files, and samples are in the `output/release` folder.
   After the complete compilation, if you only need to compile the demo, then run the following code:

```
./cmake_build.sh samples
```

6. **Enter the device information**
   Enter the information of the device created on the IoT Hub platform (with a key-authenticated device as example) in `device_info.json` in the root directory of the SDK. Below is the sample code:

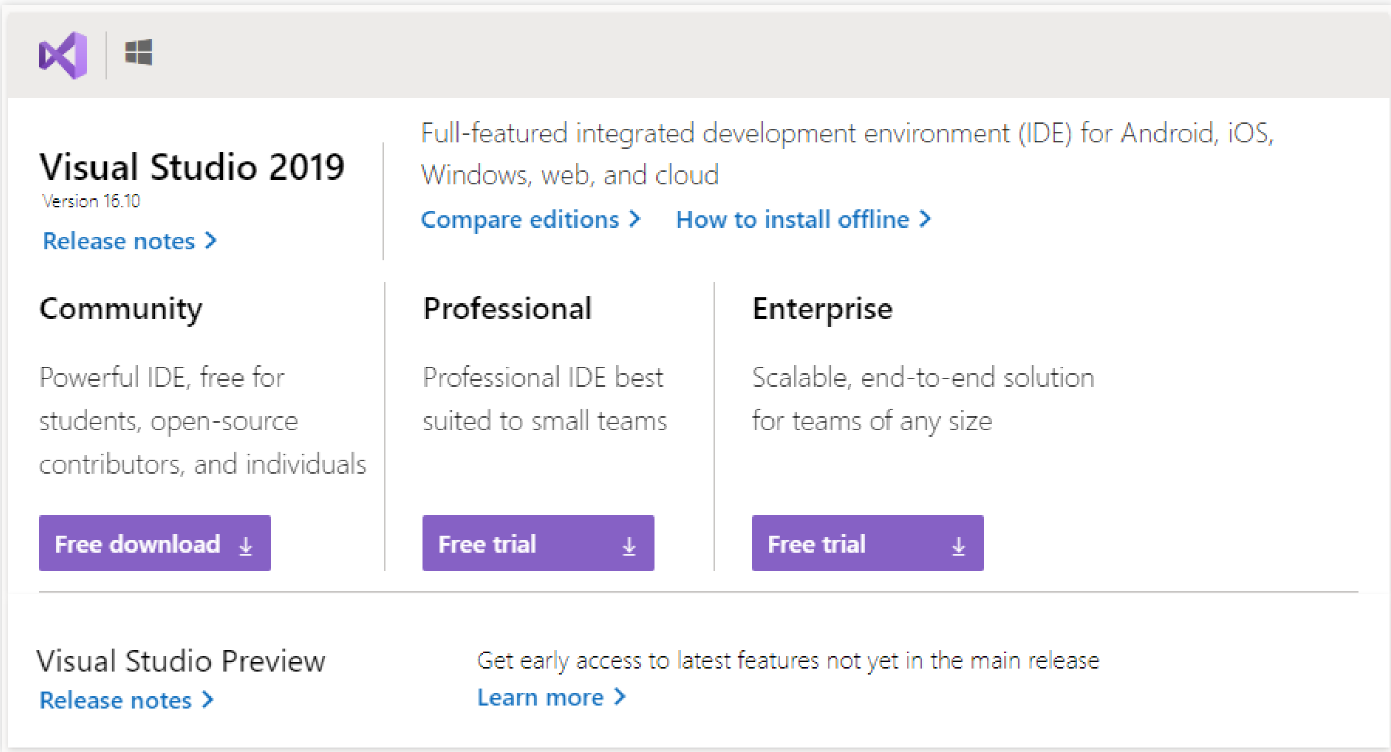
```
"auth_mode":"KEY",
"productId":"S3EUVBQAZW",
"deviceName":"test_device",
"key_deviceinfo":{
"deviceSecret":"vX6PQqazsGsMyf5SMfs6OA6y"
}
```

7. **Run the demo**
   The demo output is in the `output/release/bin` folder. For example, to run the `data_template_sample` demo, enter `./output/release/bin/data_template_sample` .

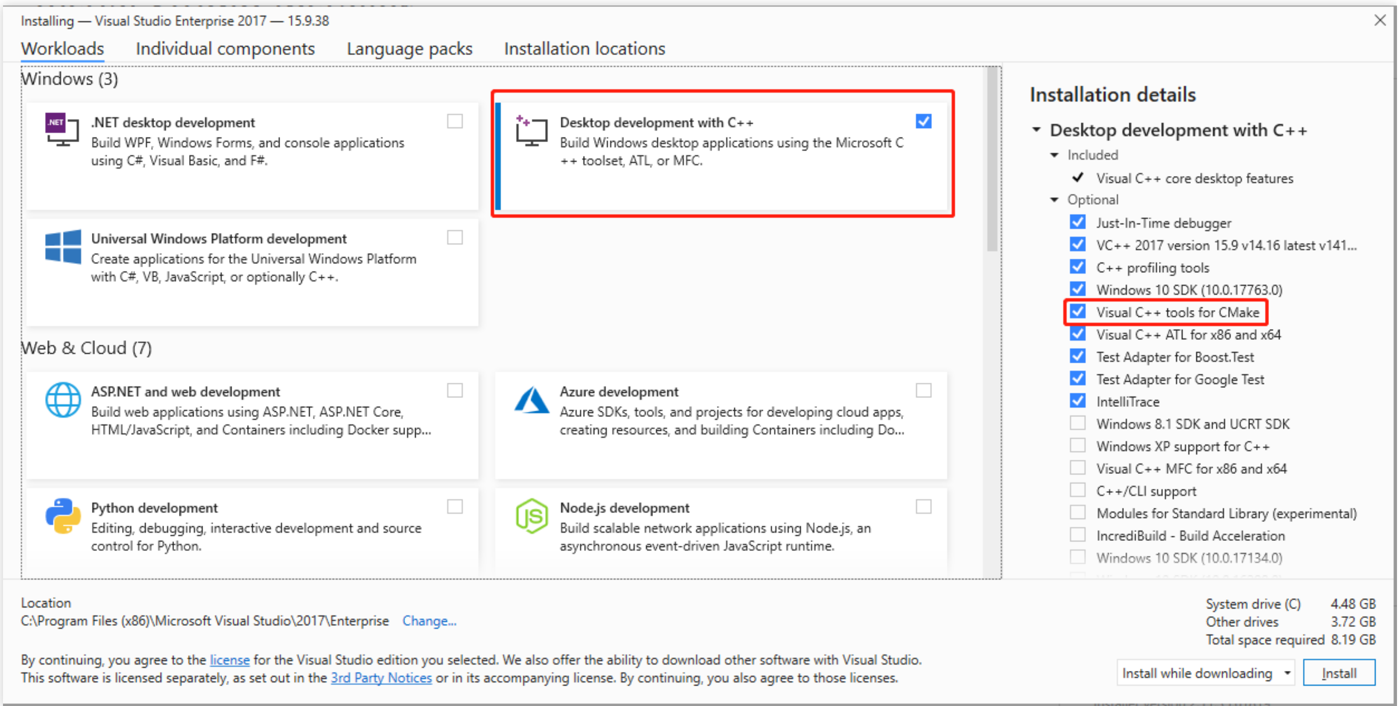
Windows

Getting and installing Visual Studio 2019

1. Download Visual Studio 2019 and install it. In this document, the downloaded and installed version is v16.2 Community.

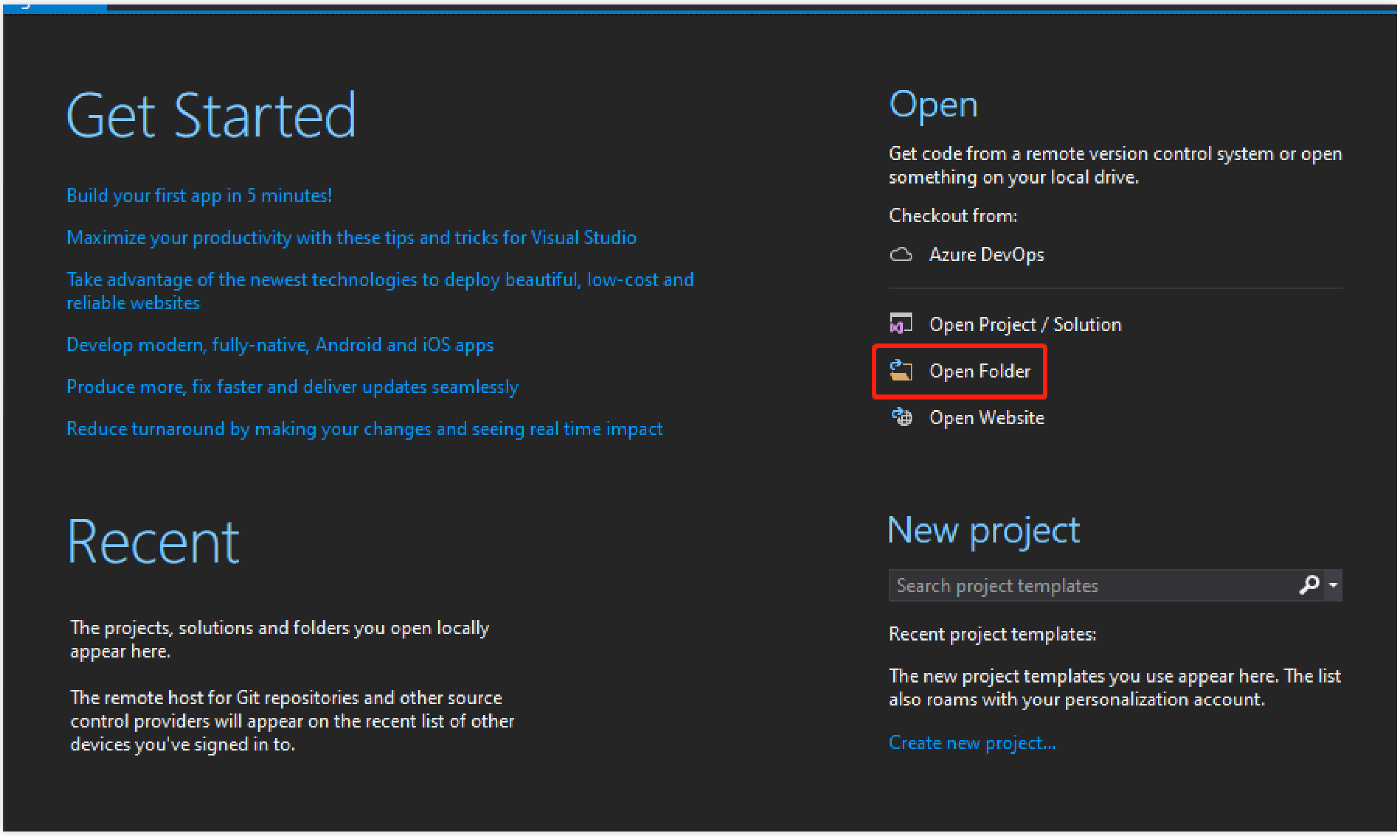


2. Select **Desktop development with C++** and **C++ CMake tools for Windows**.



Compilation and running

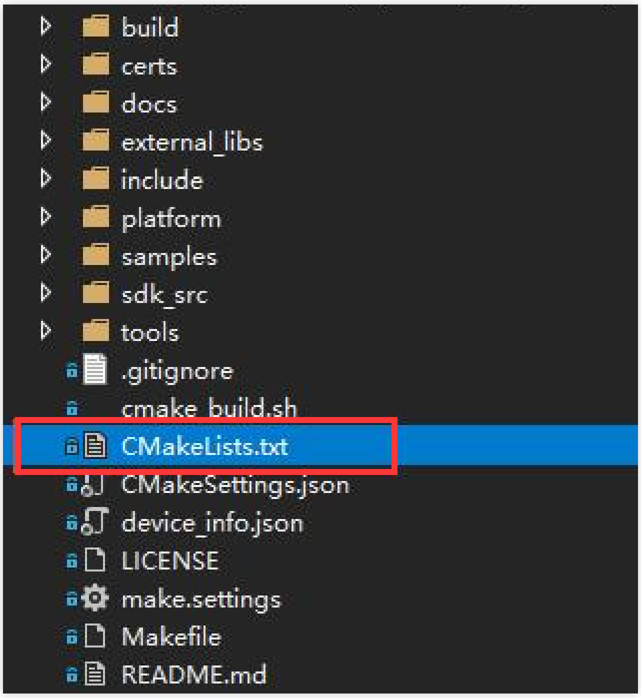
1. Run Visual Studio, select **Open a local folder**, and select the downloaded SDK for C directory.



2. Enter the information of the device created in the IoT Hub console (with a key-authenticated device as example) in `device_info.json`. Below is the sample code:

```
"auth_mode":"KEY",
"productId":"S3EUVBQAZW",
"deviceName":"test_device",
"key_deviceinfo":{
"deviceSecret":"vX6PQqazsGsMyf5SMfs6OA6y"
}
```

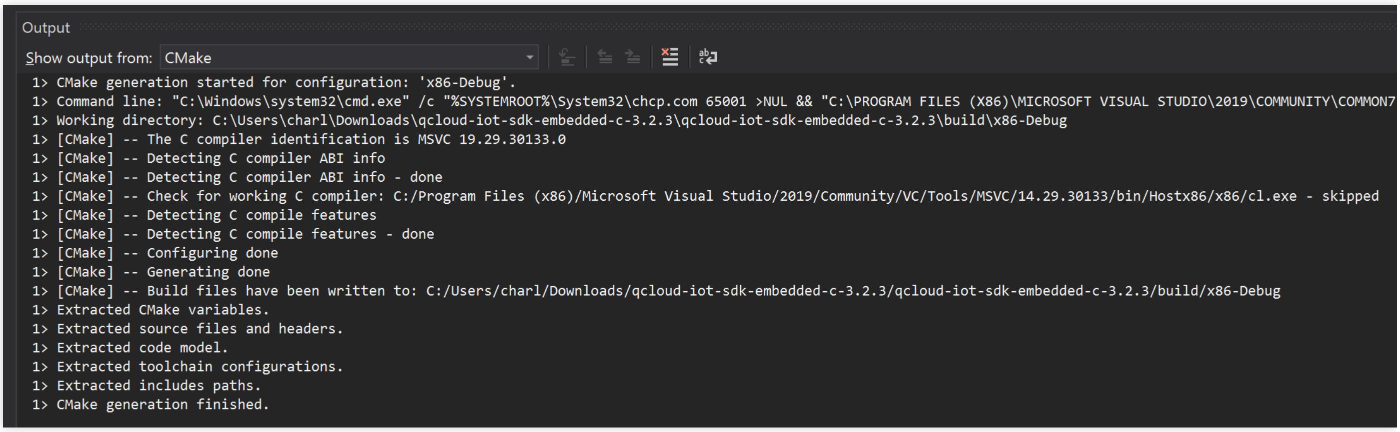
3. Double-click `CMakeLists.txt` in the root directory and make sure that the platform is set to **Windows** and the compilation tool is set to **MSVC** in the compilation toolchain.



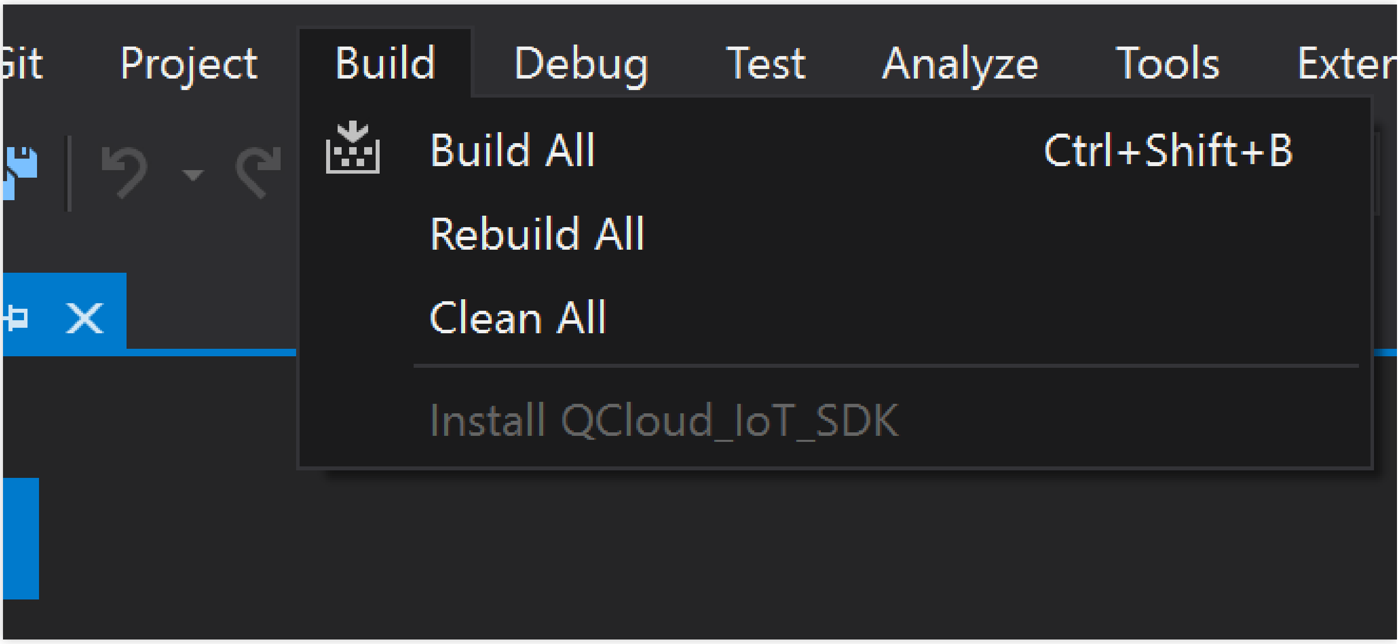

```
# Compilation toolchain
#set(COMPILE_TOOLS "gcc")
#set(PLATFORM "linux")

set(COMPILE_TOOLS "MSVC")
set(PLATFORM "windows")
```

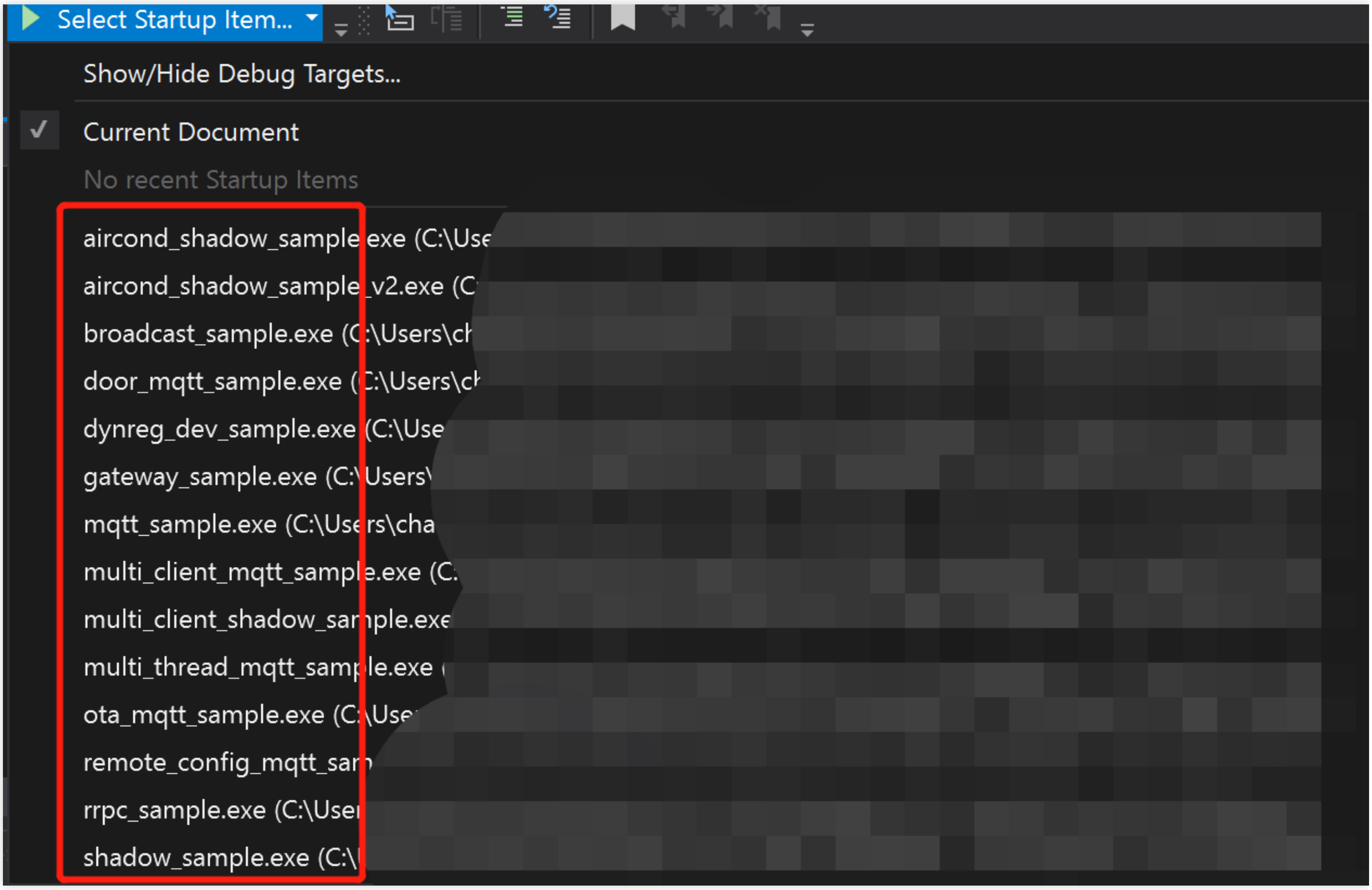
4. Visual Studio will automatically build the CMake cache. Just wait for the build to complete.



5. After the cache is generated, select **Build** > **Build All**.



6. Select the corresponding demo for running, which should correspond to the user information.



Getting Started with MQTT

Last updated：2023-07-27 10:41:13

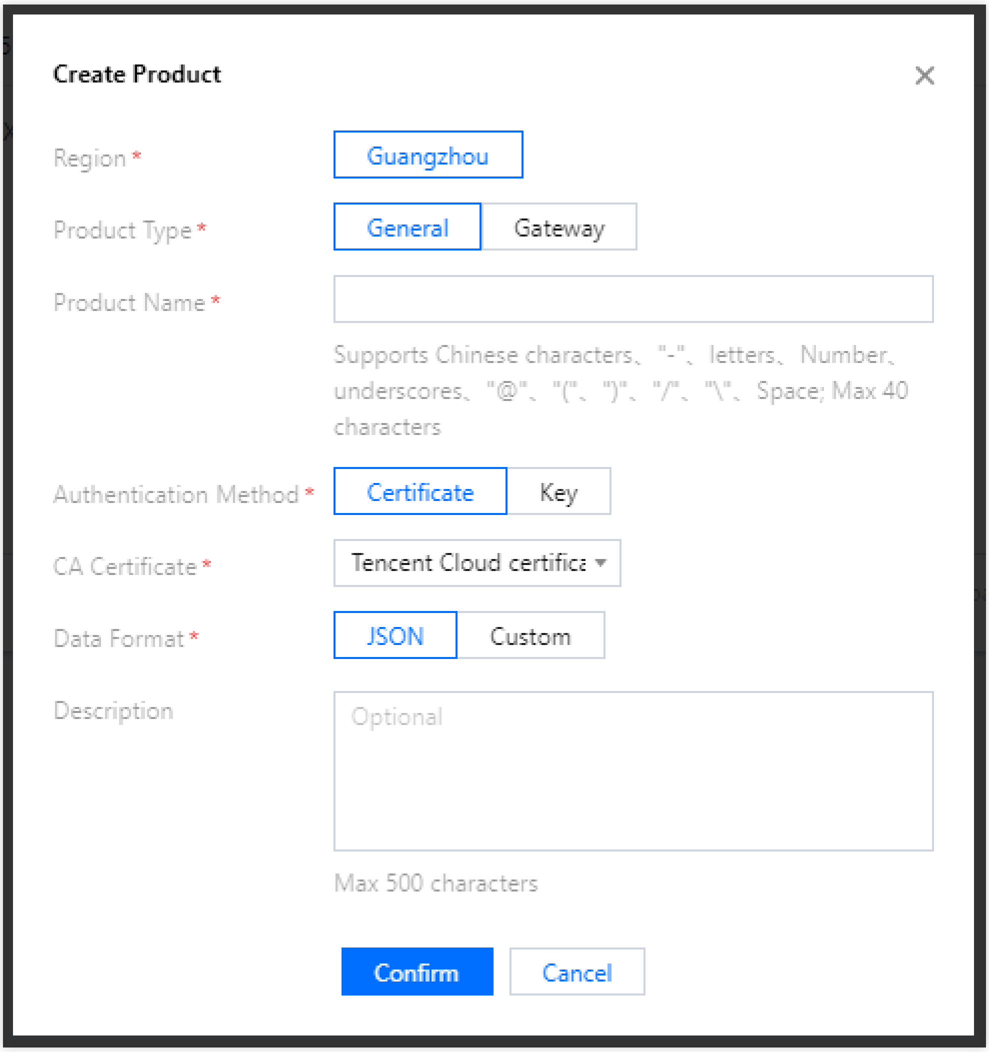
This document describes how to create devices and permissions in the IoT Hub console and quickly try out device connection to IoT Hub over the MQTT protocol for message sending and receiving based on the **mqtt_sample** of the C-SDK.

Operations in Console

Creating product and device

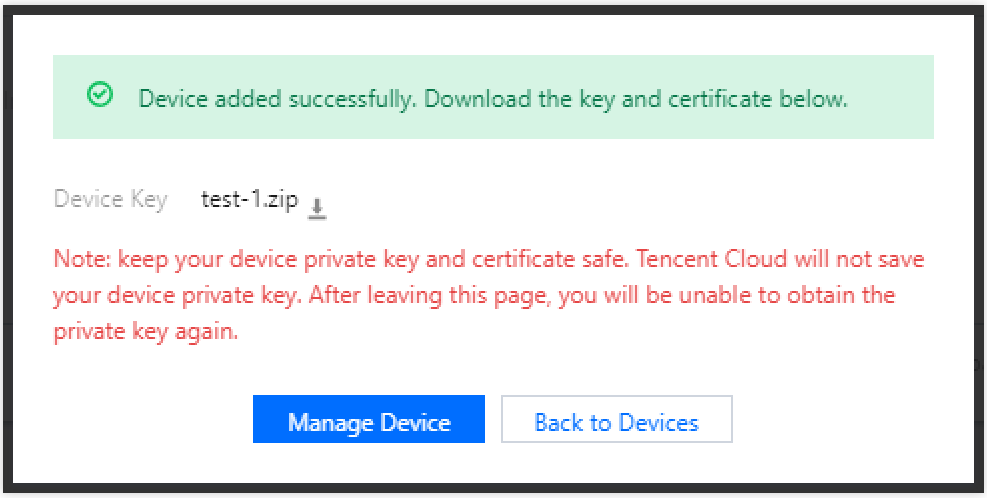
1. Log in to the IoT Hub console and click **Products** on the left sidebar.
2. On the product list page, click **Create Product**.
3. On the pop-up product adding page, select the node type and product type, enter the product name, select the authentication method and data format, and enter the product description.

Then, click **Confirm** (select as shown below for directly connected general devices).

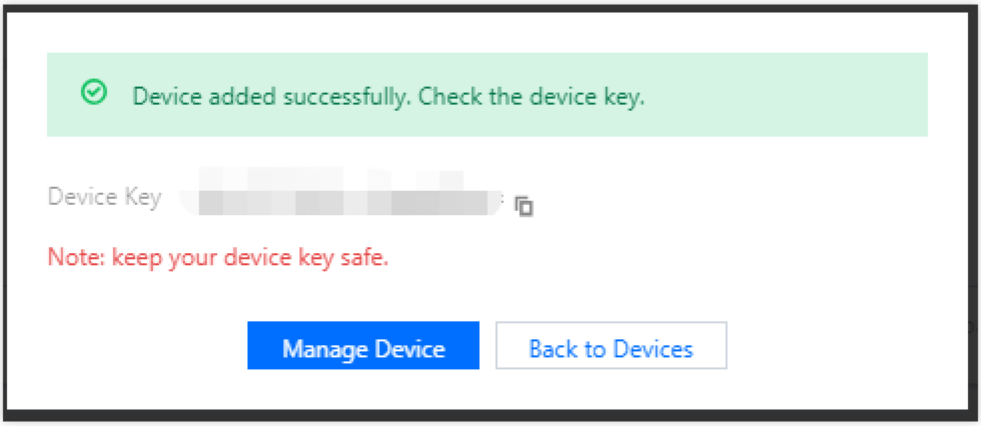


4. After the product is created, click **Devices** at the bottom of the generated product page.
5. On the device list page, click **Add Device**.

- If the authentication method is certificate authentication, after the device name is entered, be sure to click **Download** in the pop-up window. The device key and device certificate in the downloaded package are used for authenticating the device during connection to IoT Hub.

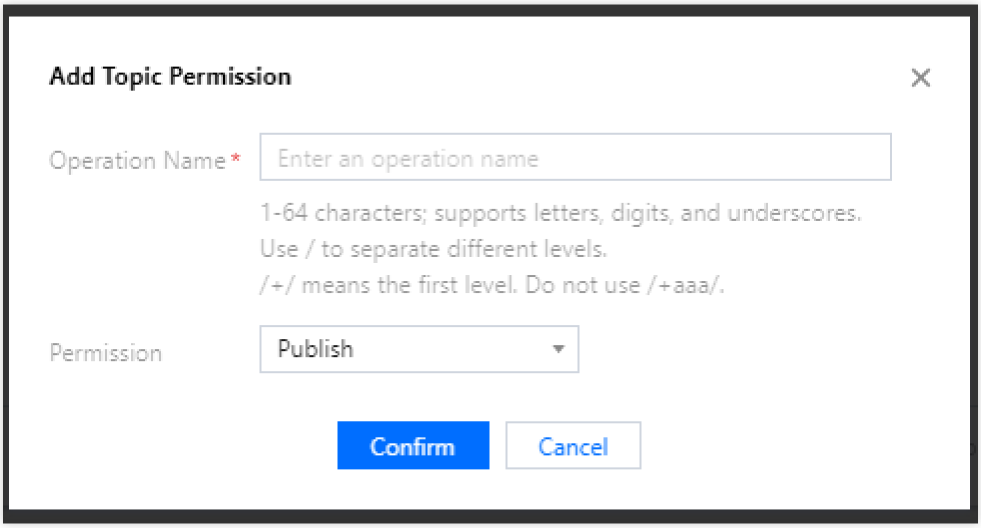


- If the authentication method is key authentication, after the device name is entered, the key of the added device will be displayed in the pop-up window.



Creating topic

1. On the generated product page, click **Permissions**.
2. On the permission list page, click **Add Topic Permission**.
3. In the topic permission pop-up window, enter `data` , set the operation permission to **Subscribe and Publish**, and click **Confirm**.



4. Then, the `productID/\${deviceName}/data` topic will be created, and you can view all permissions of the product in the permission list on the product page.

Compiling and Running Demo

The following describes how to compile and run the `mqtt_sample` demo in the Linux environment (with a key-authenticated device as example).

1. Compile the SDK

(1) Modify `CMakeLists.txt` to ensure that the following options exist:

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

(2) Run the following script for compilation.

```
./cmake_build.sh
```

(3) The demo output is in the `output/release/bin` folder.

2. Enter the device information

Enter the information of the device created above on the IoT Hub platform in `device_info.json` .

```
"auth_mode":"KEY",
"productId":"S3EUVBRJLB",
"deviceName":"test_device",
"key_deviceinfo":{
"deviceSecret":"vX6PQqazsGsMyf5SMfs6OA6y"
}
```

3. Run the `mqtt_sample` demo

```
./output/release/bin/mqtt_sample
INF|2019-09-12 21:28:20|device.c|iot_device_info_set(67): SDK_Ver: 3.1.0, Product_ID: S3EUVBRJLB, Device_Name: test_device
DBG|2019-09-12 21:28:20|HAL_TLS_mbedtls.c|HAL_TLS_Connect(204): Setting up the SSL/TLS structure...
DBG|2019-09-12 21:28:20|HAL_TLS_mbedtls.c|HAL_TLS_Connect(246): Performing the SSL/TLS handshake...
DBG|2019-09-12 21:28:20|HAL_TLS_mbedtls.c|HAL_TLS_Connect(247): Connecting to /S3EUVBRJLB.iotcloud.tencentdevices.com/8883...
INF|2019-09-12 21:28:20|HAL_TLS_mbedtls.c|HAL_TLS_Connect(269): connected with /S3EUVBRJLB.iotcloud.tencentdevices.com/8883...
INF|2019-09-12 21:28:20|mqtt_client.c|IOT_MQTT_Construct(125): mqtt connect with id: p8t0W success
INF|2019-09-12 21:28:20|mqtt_sample.c|main(303): Cloud Device Construct Success
DBG|2019-09-12 21:28:20|mqtt_client_subscribe.c|qcloud_iot_mqtt_subscribe(138): topicName=$sys/operation/result/S3EUVBRJLB/test_device|packet_id=1932
INF|2019-09-12 21:28:20|mqtt_sample.c|_mqtt_event_handler(71): subscribe success, packet-id=1932
DBG|2019-09-12 21:28:20|system_mqtt.c|_system_mqtt_sub_event_handler(80): mqtt sys topic subscribe success
DBG|2019-09-12 21:28:20|mqtt_client_publish.c|qcloud_iot_mqtt_publish(337): publish packetID=0|topicName=$sys/operation/S3EUVBRJLB/test_device|payload={"type": "get", "resource": ["time"]}
DBG|2019-09-12 21:28:20|system_mqtt.c|_system_mqtt_message_callback(63): Recv Msg Topic:$sys/operation/result/S3EUVBRJLB/test_device, payload:{"type":"get","time":1568294900}
INF|2019-09-12 21:28:21|mqtt_sample.c|main(316): system time is 1568294900
DBG|2019-09-12 21:28:21|mqtt_client_subscribe.c|qcloud_iot_mqtt_subscribe(138): topicName=S3EUVBRJLB/test_device/data|packet_id=1933
INF|2019-09-12 21:28:21|mqtt_sample.c|_mqtt_event_handler(71): subscribe success, packet-id=1933
DBG|2019-09-12 21:28:21|mqtt_client_publish.c|qcloud_iot_mqtt_publish(329): publish topic seq=1934|topicName=S3EUVBRJLB/test_device/data|payload={"action": "publish_test", "count": "0"}
INF|2019-09-12 21:28:21|mqtt_sample.c|_mqtt_event_handler(98): publish success, packet-id=1934
```

```
INF|2019-09-12 21:28:21|mqtt_sample.c|on_message_callback(195): Receive Message With topicName:S3EUVBRJLB/test_device/data, payload:{"action": "publish_test", "count": "0"}
INF|2019-09-12 21:28:22|mqtt_client_connect.c|qcloud_iot_mqtt_disconnect(437): mqtt disconnect!
INF|2019-09-12 21:28:22|system_mqtt.c|_system_mqtt_sub_event_handler(98): mqtt client has been destroyed
INF|2019-09-12 21:28:22|mqtt_client.c|IOT_MQTT_Destroy(186): mqtt release!
```

4. Observe message sending

The following log information shows that the demo reported data to `/{productID}/{deviceName}/data` through the `Publish` message type of MQTT, and that the server received and successfully processed the message.

```
INF|2019-09-12 21:28:21|mqtt_sample.c|_mqtt_event_handler(98): publish success, packet-id=1934
```

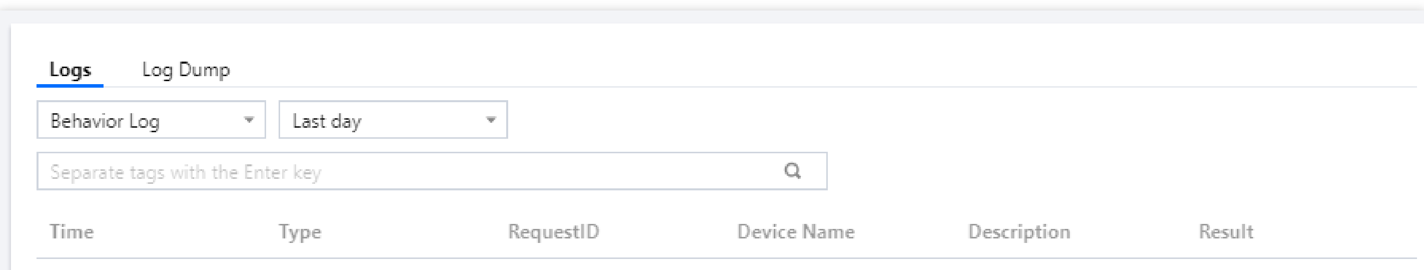
5. Observe message receiving

The following log information shows that as the message reached the subscribed topic, it was pushed to the demo as-is by the server and entered the corresponding callback function.

```
INF|2019-09-12 21:28:21|mqtt_sample.c|on_message_callback(195): Receive Message With topicName:S3EUVBRJLB/test_device/data, payload:{"action": "publish_test", "count": "0"}
```

6. Observe logs in the console

Log in to the IoT Hub console, click the product name, and click **Cloud Log** on the top to view the message just reported.



API and Variable Parameter Description

Last updated：2023-07-27 10:41:13

The header files of the device SDK for C provided for you to call such as API function declarations, constants, and variable parameter definitions are in the `include` directory. This document describes the variable parameters and API functions in the directory.

Variable Parameter Configuration

You can configure corresponding parameters in the SDK for C based on the needs in specific scenarios to ensure the smooth operations of your businesses. Variable connection parameters include:

1. Timeout period of blocking MQTT calls (including connection, subscribing, and publishing) in milliseconds. 5000 ms is recommended.
2. Size of the buffer for message sending and receiving over the MQTT protocol, which is 2,048 bytes by default and can be up to 16 KB.
3. Size of the buffer for message sending and receiving over the CoAP protocol, which is 512 bytes by default and can be up to 1 KB.
4. MQTT heartbeat message sending interval in milliseconds, which can be up to 690s.
5. Maximum waiting time for reconnection in milliseconds. When a device is reconnected after disconnection, the waiting time will double if reconnection fails, and reconnection will stop when the maximum waiting time is exceeded.

You can modify the configuration of the corresponding connection parameters by modifying the following macro definitions in the `include/qcloud_iot_export_variables.h` file.
You need to recompile the SDK after modification. Below is the sample code:

```
/* default MQTT/CoAP timeout value when connect/pub/sub (unit: ms) */
#define QCLOUD_IOT_MQTT_COMMAND_TIMEOUT (5 * 1000)

/* default MQTT keep alive interval (unit: ms) */
#define QCLOUD_IOT_MQTT_KEEP_ALIVE_INTERNAL (240 * 1000)

/* default MQTT Tx buffer size, MAX: 16*1024 */
#define QCLOUD_IOT_MQTT_TX_BUF_LEN (2048)

/* default MQTT Rx buffer size, MAX: 16*1024 */
#define QCLOUD_IOT_MQTT_RX_BUF_LEN (2048)
```

```
/* default COAP Tx buffer size, MAX: 1*1024 */
#define COAP_SENDMSG_MAX_BUFLEN (512)

/* default COAP Rx buffer size, MAX: 1*1024 */
#define COAP_RECVMSG_MAX_BUFLEN (512)

/* MAX MQTT reconnect interval (unit: ms) */
#define MAX_RECONNECT_WAIT_INTERVAL (60 * 1000)
```

API Function Description

The following describes the main features and corresponding APIs provided by the SDK for C v3.1.0 for you to compile business logic. For more information on API parameters and returned values, please see the comments in the header files of the SDK code such as `include/exports/qcloud_iot_export_*.h` .

MQTT APIs

No.	Function	Description
1	IOT_MQTT_Construct	Constructs `MQTTClient` and connects to MQTT cloud service
2	IOT_MQTT_Destroy	Closes MQTT connection and terminates `MQTTClient`
3	IOT_MQTT_Yield	Performs tasks such as reading MQTT messages, processing messages, timing out requests, and managing heartbeat packets and reconnection status in the current thread context
4	IOT_MQTT_Publish	Publishes MQTT message
5	IOT_MQTT_Subscribe	Subscribes to MQTT topic
6	IOT_MQTT_Unsubscribe	Unsubscribes from subscribed MQTT topic
7	IOT_MQTT_IsConnected	Queries whether MQTT is currently connected to
8	IOT_MQTT_GetErrCode	Gets the error code of `IOT_MQTT_Construct` failure

Notes on use in multithreaded environment

To use MQTT APIs in a multithreaded environment, you need to pay attention to the following:

- You cannot use multiple threads to call `IOT_MQTT_Yield` , `IOT_MQTT_Construct` , or `IOT_MQTT_Destroy` .

- You can use multiple threads to call `IOT_MQTT_Publish` , `IOT_MQTT_Subscribe` , and `IOT_MQTT_Unsubscribe` .
- As the function to read MQTT messages from `socket` and process them, `IOT_MQTT_Yield` should have a certain execution time to prevent it from being suspended or preempted for a long time.

Device shadow APIs

For more information on the device shadow feature, please see Device Shadow Details.

No.	Function	Description
1	IOT_Shadow_Construct	Constructs device shadow client `ShadowClient` and connects to MQTT cloud service
2	IOT_Shadow_Publish	The shadow client publishes an MQTT message
3	IOT_Shadow_Subscribe	The shadow client subscribes to an MQTT topic
4	IOT_Shadow_Unsubscribe	The shadow client unsubscribes from a subscribed MQTT topic
5	IOT_Shadow_IsConnected	Queries whether MQTT of the shadow client is currently connected to
6	IOT_Shadow_Destroy	Closes shadow MQTT connection and terminates `ShadowClient`
7	IOT_Shadow_Yield	Performs tasks such as reading MQTT messages, processing messages, timing out requests, and managing heartbeat packets and reconnection status in the current thread context
8	IOT_Shadow_Update	Updates device shadow document asynchronously
9	IOT_Shadow_Update_Sync	Updates device shadow document synchronously
10	IOT_Shadow_Get	Gets device shadow document asynchronously

No.	Function	Description
11	IOT_Shadow_Get_Sync	Gets device shadow document synchronously
12	IOT_Shadow_Register_Property	Registers the attribute of the current device
13	IOT_Shadow_UnRegister_Property	Deletes registered device attribute
14	IOT_Shadow_JSON_ConstructReport	Adds `reported` field to JSON document for update in a non-overwriting manner
15	IOT_Shadow_JSON_Construct_OverwriteReport	Adds `reported` field to JSON document for update in an overwriting manner
16	IOT_Shadow_JSON_ConstructReportAndDesireAllNull	Adds `reported` field to JSON document and empties `desired` field
17	IOT_Shadow_JSON_ConstructDesireAllNull	Adds `"desired": null` field to JSON document

CoAP APIs

No.	Function	Description
1	IOT_COAP_Construct	Constructs `CoAPClient` and completes CoAP connection
2	IOT_COAP_Destroy	Closes CoAP connection and terminates `CoAPClient`
3	IOT_COAP_Yield	Performs tasks such as reading CoAP messages and processing messages in the current thread context
4	IOT_COAP_SendMessage	Publishes CoAP message
5	IOT_COAP_GetMessageId	Gets `msgId` in `CoAP Response` message
6	IOT_COAP_GetMessagePayload	Gets the content of `CoAP Response` message
7	IOT_COAP_GetMessageCode	Gets the error code of `CoAP Response` message

OTA APIs

For more information on the OTA firmware download feature, please see Device Firmware Update.

No.	Function	Description

No.	Function	Description
1	IOT_OTA_Init	Initializes OTA module. The client needs to initialize MQTT/CoAP before calling this API
2	IOT_OTA_Destroy	Releases resources related to OTA module
3	IOT_OTA_ReportVersion	Reports local firmware version information to OTA server
4	IOT_OTA_IsFetching	Checks whether the firmware is being downloaded
5	IOT_OTA_IsFetchFinish	Checks whether the firmware has been downloaded
6	IOT_OTA_FetchYield	Gets firmware from remote server with specific timeout value
7	IOT_OTA_Ioctl	Gets specified OTA information
8	IOT_OTA_GetLastError	Gets the last error code
9	IOT_OTA_StartDownload	Establishes HTTP connection with firmware server according to obtained firmware update address and local firmware information offset (whether to support checkpoint restart)
10	IOT_OTA_UpdateClientMd5	Calculates the MD5 of local firmware before checkpoint restart
11	IOT_OTA_ReportUpgradeBegin	Reports the status of impending update to server before firmware update
12	IOT_OTA_ReportUpgradeSuccess	Reports the status of update success to server after successful firmware update
13	IOT_OTA_ReportUpgradeFail	Reports the status of update failure to server after failed firmware update

Log APIs

For more information on the device log reporting feature, please see the log reporting section of the IoT Hub documentation in the SDK `docs` directory.

No.	Function	Description
1	IOT_Log_Set_Level	Sets the printout level of SDK logs
2	IOT_Log_Get_Level	Returns the printout level of SDK logs
3	IOT_Log_Set_MessageHandler	Sets log callback function to redirect SDK logs to another output method

No.	Function	Description
4	IOT_Log_Init_Uploader	Enables SDK log reporting to the cloud and initializes resources
5	IOT_Log_Fini_Uploader	Disables SDK log reporting to the cloud and releases resources
6	IOT_Log_Upload	Reports SDK execution logs to the cloud
7	IOT_Log_Set_Upload_Level	Sets the reporting level of SDK logs
8	IOT_Log_Get_Upload_Level	Returns the reporting level of SDK logs
9	Log_d/i/w/e	Prints SDK logs by level

System time APIs

No.	Function	Description
1	IOT_Get_SysTime	Gets IoT Hub's backend time. Currently, the time sync feature is supported only for the MQTT channel

Gateway feature APIs

Fore more information on the gateway feature, please see the gateway product section of the IoT Hub documentation in the SDK `docs` directory.

No.	Function	Description
1	IOT_Gateway_Construct	Constructs gateway client and completes MQTT connection
2	IOT_Gateway_Destroy	Closes MQTT connection and terminates gateway client
3	IOT_Gateway_Subdev_Online	Connects subdevice
4	IOT_Gateway_Subdev_Offline	Disconnects subdevice
5	IOT_Gateway_Yield	Performs tasks such as reading MQTT messages, processing messages, timing out requests, and managing heartbeat packets and reconnection status in the current thread context
6	IOT_Gateway_Publish	Publishes MQTT message
7	IOT_Gateway_Subscribe	Subscribes to MQTT topic
8	IOT_Gateway_Unsubscribe	Unsubscribes from subscribed MQTT topic

Device Information Storage

Last updated：2023-07-27 10:41:13

Overview

IoT Hub assigns a unique product ID to each created product. You can customize the `DeviceName` to identify devices and use the product ID + device ID + device certificate/key to authenticate devices. Devices need to store such identity information. The C-SDK provides APIs for reading and writing the device information and reference implementations for adaptation as needed.

Device Identity Information

- Certificate-authenticated devices must carry the following four pieces of information before it can pass the authentication by the platform: product ID (ProductId), device name (DeviceName), device certificate (DeviceCert), and device private key (DevicePrivateKey), among which, the certificate and private key files are generated by the platform and correspond to each other.
- Key-authenticated devices must carry the following three pieces of information before it can pass the authentication by the platform: product ID (ProductId), device name (DeviceName), and device key (DeviceSecret), among which, the device key is generated by the platform.

Device Identity Information Burning

Device information burning is divided into preset burning and dynamic burning, which differ in terms of convenience and security.

Preset burning

After a product is created, you can create devices one by one in the IoT Hub console or through TencentCloud API, get their corresponding device information, and burn the above three or four pieces of information into a non-volatile medium in a specific step of device production, so that the device SDK can read the stored device information during running for device authentication.

Dynamic burning

- Preset burning: this involves performing personalized production actions in the mass production process and thus affects the production efficiency. To improve the ease of use, the platform supports dynamic burning. This feature is implemented as follows: after a product is created, its dynamic registration feature can be enabled to generate a product key (ProductSecret). Unified product information can be burned for all devices under it in the production process, i.e., product ID (ProductId) and product key (ProductSecret). After the devices are shipped, the device identity information can be obtained through dynamic registration and then saved, and then obtained three or four pieces of information can be used for device authentication.

- Device name (DeviceName) generation for dynamic burning: if automatic device creation is enabled during dynamic registration, device names can be generated by devices themselves, which are generally device IMEIs or MAC addresses but must be unique under the same product ID (ProductId). If automatic device creation is not enabled during dynamic registration, device names should be entered on the platform in advance, and the platform will verify whether the requested device names are validly entered during dynamic device registration. This can reduce the security risks in case of product key leakage.

> Note：
> For dynamic registration, you should ensure the security of the product key (ProductSecret); otherwise, major security risks may arise.

Device Information Read/Write HAL APIs

The SDK provides HAL APIs for reading and writing device information, which must be implemented. For more information on how to implement device information read/write, please see `HAL_Device_Linux.c` on Linux.

Device information HAL APIs:

HAL_API	Description
HAL_SetDevInfo	Writes device information
HAL_GetDevInfo	Reads device information

Device Information Configuration in Development Phase

After a device is created, you need to configure its information (`ProductID/DeviceName/DeviceSecret/Cert/Key` file) in the SDK first before the demo can run properly. In the development phase, the SDK provides two methods of storing the device information:

1. If the device information is stored in the code (compilation option `DEBUG_DEV_INFO_USED` = `ON`), you should modify the device information in `platform/os/xxx/HAL_Device_xxx.c` . This method can be used on platforms without a file system.

```
/* product Id */
static char sg_product_id[MAX_SIZE_OF_PRODUCT_ID + 1] = "PRODUCT_ID";

/* device name */
static char sg_device_name[MAX_SIZE_OF_DEVICE_NAME + 1] = "YOUR_DEV_NAME";

#ifdef DEV_DYN_REG_ENABLED
/* product secret for device dynamic Registration */
static char sg_product_secret[MAX_SIZE_OF_PRODUCT_SECRET + 1] = "YOUR_PRODUCT_SEC
RET";
#endif

#ifdef AUTH_MODE_CERT
/* public cert file name of certificate device */
static char sg_device_cert_file_name[MAX_SIZE_OF_DEVICE_CERT_FILE_NAME + 1] = "YO
UR_DEVICE_NAME_cert.crt";
/* private key file name of certificate device */
static char sg_device_privatekey_file_name[MAX_SIZE_OF_DEVICE_SECRET_FILE_NAME +
1] = "YOUR_DEVICE_NAME_private.key";
#else
/* device secret of PSK device */
static char sg_device_secret[MAX_SIZE_OF_DEVICE_SECRET + 1] = "YOUR_IOT_PSK";
#endif
```

2. If the device information is stored in the configuration file (compilation option `DEBUG_DEV_INFO_USED` = `OFF`), you should modify the device information in the `device_info.json` file with no need to recompile the SDK. This method is recommended for development on Linux and Windows.

```
{
"auth_mode":"KEY/CERT",

"productId":"PRODUCT_ID",
"productSecret":"YOUR_PRODUCT_SECRET",
"deviceName":"YOUR_DEV_NAME",

"key_deviceinfo":{
"deviceSecret":"YOUR_IOT_PSK"
},

"cert_deviceinfo":{
"devCertFile":"YOUR_DEVICE_CERT_FILE_NAME",
"devPrivateKeyFile":"YOUR_DEVICE_PRIVATE_KEY_FILE_NAME"
},

"subDev":{
```

```
"sub_productId":"YOUR_SUBDEV_PRODUCT_ID",
"sub_devName":"YOUR_SUBDEV_DEVICE_NAME"
}
}
```

Use Cases

- Initialize the connection parameters

```
static DeviceInfo sg_devInfo;

static int _setup_connect_init_params(MQTTInitParams* initParams)
{
int ret;

ret = HAL_GetDevInfo((void *)&sg_devInfo);
if(QCLOUD_ERR_SUCCESS != ret){
return ret;
}

initParams->device_name = sg_devInfo.device_name;
initParams->product_id = sg_devInfo.product_id;
......
}
```

- Generate the parameters for authenticating a key-authenticated device

```
static int _serialize_connect_packet(unsigned char *buf, size_t buf_len, MQTTConn
ectParams *options, uint32_t *serialized_len) {
......
......
int username_len = strlen(options->client_id) + strlen(QCLOUD_IOT_DEVICE_SDK_APPI
D) + MAX_CONN_ID_LEN + cur_timesec_len + 4;
options->username = (char*)HAL_Malloc(username_len);
get_next_conn_id(options->conn_id);
HAL_Snprintf(options->username, username_len, "%s;%s;%s;%ld", options->client_id,
QCLOUD_IOT_DEVICE_SDK_APPID, options->conn_id, cur_timesec);

#if defined(AUTH_WITH_NOTLS) && defined(AUTH_MODE_KEY)
if (options->device_secret != NULL && options->username != NULL) {
char sign[41] = {0};
utils_hmac_sha1(options->username, strlen(options->username), sign, options->devi
```



```
ce_secret, options->device_secret_len);
options->password = (char*) HAL_Malloc (51);
if (options->password == NULL) IOT_FUNC_EXIT_RC(QCLOUD_ERR_INVAL);
HAL_Snprintf(options->password, 51, "%s;hmacsha1", sign);
}
#endif
......
}
```

Connection Based on SDK for Android

SDK for Android Release Notes

Last updated：2023-07-27 10:51:42

Code Hosting

The code of the device SDK for Android has been hosted on GitHub since v1.0.0.

Version Information

For the version iteration information of the device SDK for Android since v1.0.0, please visit GitHub.

SDK for Android Project Configuration

Last updated：2023-07-27 10:51:43

IoT Hub device SDK for Android relies on a secure and powerful data channel to enable IoT developers to quickly connect devices to the cloud for two-way communication. You only need to complete the corresponding project configuration to connect devices.

Prerequisites

Products and devices have been created as instructed in Device Connection Preparations.

How to Import

SDK integration

If you don't need to run the IoT Hub SDK in the service component, only dependency on iot_core is required.

- Depend on Maven for remote build. Below is the sample code:

```
dependencies {
implementation 'com.tencent.iot.hub:hub-device-android-core:x.x.x'
implementation 'com.tencent.iot.hub:hub-device-android-service:x.x.x'
}
```

> Note：
>
> - You can set the above x.x.x to the latest version according to SDK for Android Release Notes.
> - If you don't need to run the IoT Hub SDK in the service component, only dependency on `iot_core` is required.
> - If you need to run the IoT Hub SDK in the service component, only dependency on `iot_service` is required.

- Depend on the local SDK source code for build:
  Modify the build.gradle of the application module to make it dependent on the iot_core and iot_service source code. Below is the sample code:

```
dependencies {
implementation project(':hub:hub-device-android:iot_core')
implementation project(':hub:hub-device-android:iot_service')
}
```

Connection Authentication

Edit the configuration information in the app-config.json file so that the following data in IoTMqttFragment.java can be read:

```
{
"PRODUCT_ID": "",
"DEVICE_NAME": "",
"DEVICE_PSK": "",
"SUB_PRODUCT_ID": "",
"SUB_DEV_NAME": "",
"SUB_PRODUCT_KEY": "",
"TEST_TOPIC": "",
"SHADOW_TEST_TOPIC": "",
"PRODUCT_KEY": ""
}
```

The SDK supports two authentication methods: certificate authentication and key authentication, which should be selected and set according to the authentication type of the created product.

- For key authentication, you need to enter the parameters corresponding to `PRODUCT_ID` , `DEVICE_NAME` , and `DEVICE_PSK` in the configuration information in app-config.json. The SDK will automatically generate a signature based on the device configuration information as a credential for connection to IoT Hub.
- For certificate authentication, you need to enter the `PRODUCT_ID` and `DEVICE_NAME` in the configuration information in app-config.json and read the content of the device certificate and private key files in either of the following two ways:
- Read through `AssetManager` . In this case, you need to create the `assets` directory under the `hub/hub-android-demo/src/main` path of the project and place the device certificate and private key files in it.
- Read through `InputStream` . In this case, you need to pass in the full path information of the device certificate and private key files.

i. After successfully reading the certificate and private key files, you need to set the `mDevCertName` certificate name and `mDevKeyName` private key name in IoTMqttFragment.java.

```
private String mDevCertName = "YOUR_DEVICE_NAME_cert.crt";
private String mDevKeyName = "YOUR_DEVICE_NAME_private.key";
```

ii. After the configuration is completed, call the MQTT connection APIs of the SDK in the project to complete device connection.

```
mMqttConnection = new TXGatewayConnection(mContext, mBrokerURL, mProductID, mDevName, mDevPSK,null,null ,mMqttLogFlag, mMqttLogCallBack, mMqttActionCallBack);
mMqttConnection.connect(options, mqttRequest);
```


SDK for Android Use Instructions

Last updated：2023-07-27 10:51:42

In addition to the device connection feature, the SDK for Android also provides gateway subdevice, device shadow, and OTA features with the following APIs.

MQTT APIs

TXMqttConnection

Method	Description
connect	Establishes MQTT connection
reconnect	Reestablishes MQTT connection
disConnect	Closes MQTT connection
publish	Publishes MQTT message
subscribe	Subscribes to MQTT topic
unSubscribe	Unsubscribes from MQTT topic
getConnectStatus	Gets MQTT connection status
setBufferOpts	Sets buffer for disconnection status
initOTA	Initializes OTA feature
reportCurrentFirmwareVersion	Reports current device version information to backend server
reportOTAState	Reports device update status to backend server

MQTT Gateway APIs

TXGatewayConnection

Method	Description
connect	Establishes gateway connection
reconnect	Reestablishes gateway connection
disConnect	Closes gateway MQTT connection

Method	Description
publish	Publishes MQTT message
subscribe	Subscribes to MQTT topic
unSubscribe	Unsubscribes from MQTT topic
getConnectStatus	Gets MQTT connection status
setBufferOpts	Sets buffer for disconnection status
initOTA	Initializes OTA feature
reportCurrentFirmwareVersion	Reports current device version information to backend server
reportOTAState	Reports device update status to backend server
addSubDev	Adds subdevice
removeSubdev	Removes subdevice
findSubdev	Finds subdevice
gatewaySubdevOffline	Connects subdevice
gatewaySubdevOnline	Disconnects subdevice

Device Shadow APIs

TXShadowConnection

Method	Description
connect	Establishes shadow connection
disConnect	Closes shadow connection
getConnectStatus	Gets MQTT connection status
update	Updates device shadow document
get	Gets device shadow document
registerProperty	Registers device attribute
unRegisterProperty	Unregisters device attribute

Method	Description
reportNullDesiredInfo	Reports empty `desired` information after updating `delta` information
setBufferOpts	Sets buffer for disconnection status
getMqttConnection	Gets `TXMqttConnection` instance

MQTT Remote Service Client

TXMqttClient

Method	Description
setMqttActionCallBack	Sets `MqttAction` callback API
setServiceConnection	Sets remote service connection callback API
init	Initializes remote service client
startRemoteService	Starts remote service
stopRemoteService	Stops remote service
connect	Establishes MQTT connection
disConnect	Closes MQTT connection
subscribe	Subscribes to MQTT topic
unSubscribe	Unsubscribes from MQTT topic
publish	Publishes MQTT message
setBufferOpts	Sets buffer for disconnection status
clear	Releases resource

Shadow Remote Service Client

TXShadowClient

Method	Description
setShadowActionCallBack	Sets `ShadowAction` callback API
setServiceConnection	Sets remote service connection callback API

Method	Description
init	Initializes remote service client
startRemoteService	Starts remote service
stopRemoteService	Stops remote service
connect	Establishes shadow connection
disConnect	Closes shadow connection
getMqttClient	Gets MQTT client instance
get	Gets device shadow
update	Updates device shadow
registerProperty	Registers device attribute
unRegisterProperty	Unregisters device attribute
reportNullDesiredInfo	Reports empty `desired` information after updating `delta` information
setBufferOpts	Sets buffer for disconnection status
clear	Releases resource

Firmware Update over MQTT Channel

TXMqttClient

Method	Description
initOTA	Initializes OTA feature
reportCurrentFirmwareVersion	Reports current device version information to backend server
reportOTAState	Reports device update status to backend server

Connection Based on SDK for Java

SDK for Java Release Notes

Last updated：2023-07-27 10:51:42

Code Hosting

The code of the device SDK for Java has been hosted on GitHub since v1.0.0.

Version Information

For the version iteration information of the device SDK for Java since v1.0.0, please visit GitHub.

SDK for Java Project Configuration

Last updated：2023-07-27 10:51:42

IoT Hub device SDK for Java relies on a secure and powerful data channel to enable IoT developers to quickly connect devices to the cloud for two-way communication. You only need to complete the corresponding project configuration to connect devices.

Prerequisites

Products and devices have been created as instructed in Device Connection Preparations.

How to Import

- If you need to use JAR import for project development, you can add dependencies in `build.gradle` in the `module` directory as follows:

```
dependencies {
...
implementation 'com.tencent.iot.hub:hub-device-java:x.x.x'
}
```

Note：
You can set the above x.x.x to the latest version according to SDK for Java Release Notes.

- If you need to develop a project through code integration, you can download the SDK for Java source code from GitHub.

Connection Authentication

Two device authentication methods are supported: key authentication and certificate authentication.

- Key authentication requires `ProductID` , `DevName` , and `DevPSK` .
- Certificate authentication requires `ProductID` , `CertFile` , and `PrivateKeyFile` .

Below is the sample code for connection authentication:

```
private String mProductID = "YOUR_PRODUCT_ID";
private String mDevName = "YOUR_DEVICE_NAME";
private String mDevPSK = "YOUR_DEV_PSK";
private String mCertFilePath = null;
private String mPrivKeyFilePath = null;
TXMqttConnection mqttconnection = new TXMqttConnection(mProductID, mDevName, mDev
PSK, new callBack());
mqttconnection.connect(options, null);
try {
Thread.sleep(20000);
} catch (InterruptedException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
mqttconnection.disConnect(null);
```

SDK for Java Use Instructions

Last updated：2023-07-27 10:51:42

In addition to the device connection feature, the SDK for Java also provides gateway subdevice and device shadow features with the following APIs.

MQTT APIs

The APIs related to MQTT are defined in the `TXMqttConnection` class and support publishing and subscribing. If you want to support the device shadow feature, you need to use the `TXShadowConnection` class and its methods. `TXMqttConnection` class APIs are as detailed below:

Method	Description
connect	Establishes MQTT connection
reconnect	Reestablishes MQTT connection
disConnect	Closes MQTT connection
publish	Publishes MQTT message
subscribe	Subscribes to MQTT topic
unSubscribe	Unsubscribes from MQTT topic
getConnectStatus	Gets MQTT connection status
setBufferOpts	Sets buffer for disconnection status

MQTT Gateway APIs

- 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, they need to be bound or unbound through the platform.

Note：

> After a subdevice is connected once, as long as the gateway is successfully connected subsequently, the backend will show that the subdevice is online until it is disconnected.

The APIs related to MQTT gateway are defined in the `TXGatewayConnection` class as detailed below:

Method	Description
connect	Establishes gateway MQTT connection
reconnect	Reestablishes gateway MQTT connection
disConnect	Closes gateway MQTT connection
publish	Publishes MQTT message
subscribe	Subscribes to MQTT topic
unSubscribe	Unsubscribes from MQTT topic
getConnectStatus	Gets MQTT connection status
setBufferOpts	Sets buffer for disconnection status
gatewaySubdevOffline	Connects subdevice
gatewaySubdevOnline	Disconnects subdevice
gatewayBindSubdev	Binds subdevice
gatewayUnbindSubdev	Unbinds subdevice

Device Shadow APIs

If you want to support the device shadow feature, you need to use the APIs in the `TXShadowConnection` class as detailed below:

Method	Description
connect	Establishes MQTT connection
reconnect	Reestablishes MQTT connection
disConnect	Closes MQTT connection

Method	Description
publish	Publishes MQTT message
subscribe	Subscribes to MQTT topic
unSubscribe	Unsubscribes from MQTT topic
update	Updates device shadow document
get	Gets device shadow document
reportNullDesiredInfo	Reports the empty `desired` information after updating `delta` information
setBufferOpts	Sets buffer for disconnection status
getMqttConnection	Gets `TXMqttConnection` instance
getConnectStatus	Gets MQTT connection status

Connection Based on SDK for Python

Python SDK Release Notes

Last updated：2023-07-27 10:51:42

Code Hosting

The code of the device SDK for Python has been hosted on GitHub since v1.0.0.

Version Information

For the version iteration information of the device SDK for Python since v1.0.0, see GitHub.

SDK for Python Project Configuration

Last updated：2023-07-27 10:51:43

IoT Hub device SDK for Python relies on a secure and powerful data channel to enable IoT developers to quickly connect devices to the cloud for two-way communication. You only need to complete the corresponding project configuration to connect devices.

Prerequisites

Products and devices have been created as instructed in Device Connection Preparations.

How to Import

- If you want to develop a project through import, you can install the SDK as follows:

  ```
  pip3 install tencent-iot-device
  ```

If you need to view the used SDK version, run the following command:

```
pip3 show --files tencent-iot-device
```

If you need to update the SDK version, run the following command:

```
pip3 install --upgrade tencent-iot-device
```

- If you want to develop a project through code integration, you can download the SDK for Python source code from Github.

SDK for Python Use Instructions

Last updated：2023-07-27 10:51:42

In addition to the device connection feature, the SDK for Python also provides gateway subdevice and device shadow features with the following APIs.

MQTT APIs

MQTT APIs are defined in the hub.py class and support publishing and subscribing. If you want to support the device shadow feature, you need to use the shadow.py class and its methods as detailed below:

Method	Description
connect	Establishes MQTT connection
disconnect	Closes MQTT connection
subscribe	Subscribes to topic over MQTT
unsubscribe	Unsubscribes from topic over MQTT
publish	Publishes message over MQTT
registerMqttCallback	Registers MQTT callback function
registerUserCallback	Registers user callback function
isMqttConnected	Checks whether MQTT is normally connected
getConnectState	Gets MQTT connection status
setReconnectInterval	Sets MQTT reconnection attempt interval
setMessageTimout	Sets message sending timeout period
setKeepaliveInterval	Sets MQTT keepalive interval

MQTT Gateway APIs

- 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, they need to be bound or unbound through the platform.

> Note：
> After a subdevice is connected once, as long as the gateway is successfully connected subsequently, the backend will show that the subdevice is online until it is disconnected.

MQTT gateway APIs are defined in the gateway.py class as detailed below:

Method	Description
gatewayInit	Initializes gateway
isSubdevStatusOnline	Determines whether subdevice is connected
updateSubdevStatus	Updates subdevice's connection status
gatewaySubdevGetConfigList	Gets subdevice list from configuration file
gatewaySubdevOnline	Proxies subdevice connection
gatewaySubdevOffline	Proxies subdevice disconnection
gatewaySubdevBind	Binds subdevice
gatewaySubdevUnbind	Unbinds subdevice
gatewaySubdevSubscribe	Proxies subdevice subscription

Dynamic Registration APIs

If you want to use the dynamic registration feature, you need to use the APIs in the hub.py class as detailed below:

Method	Description
dynregDevice	Gets the dynamic registration information of device

OTA APIs

If you want to use the OTA feature, you need to use the APIs in the hub.py class as detailed below:

Method	Description
otaInit	Initializes OTA
otaIsFetching	Determines whether the download is in progress
otaIsFetchFinished	Determines whether the download is completed
otaReportUpgradeSuccess	Reports update success message
otaReportUpgradeFail	Reports update failure message
otaIoctlNumber	Gets the information of the downloaded firmware in `int` type, such as the size
otaIoctlString	Gets the information of the downloaded firmware in `String` type, such as MD5
otaResetMd5	Resets MD5 information
otaMd5Update	Updates MD5 information
httpInit	Initializes HTTP
otaReportVersion	Reports the information of current firmware version
otaDownloadStart	Starts firmware download
otaFetchYield	Reads firmware