MQTT Client User Guide

Version 1.50

For use with MQTT Client module versions 1.41 and above

Date: 04-Sep-2017 17:15

Table of Contents

System Overview 4
Introduction 4
Feature Check 6
Publish and Subscribe 7
MQTT Features 8
Small Packet Overhead 8
Control Packets 9
Quality of Service - QoS 10
Topic-based Routing 10
Example 11
Client Down Notifications 11
Retained Messages 11
Clean Session or Continuous Session Awareness 12
MQTT Security 12
Use of TLS 12
Authentication and Authorization 12
Connection Setup 12
Packages and Documents 14
Packages 14
Documents 14
Change History 15
Source File List 16
API Header File 16
Configuration File 16
System Files 16
Demo Code 16
Version File 16
Configuration Options 17
Application Programming Interface 20
Module Management 20
mqttc_init 21
mqttc_start 22
mqttc_stop 23
mqttc_delete 24
Protocol Management 25
mqttc_connect 26
mqttc_disconnect 27
mqttc_reconnect 28
mqttc_publish 29
mqttc_subscribe 30
mqttc_unsubscribe 31
mqttc_ping 32

Callback Functions 33
t_mqttc_connect_callback 34
t_mqttc_subscribe_callback 35
t_mqttc_publish_callback 36
Error Codes 37
Types and Definitions 38
t_mqttc_connect_config 38
s_mqttc_client_config 38
s_mqttc_session_config 39
s_mqttc_will_config 39
t_mqttc_subscribe_data 40
t_mqttc_publish_data 40
t_mqttc_connection_report 41
t_mqttc_subscribe_report 42
QoS Levels 43
Code Examples 44
Connecting 45
Disconnecting 48
Reconnecting 50
Subscribing 52
Unsubscribing 54
Publishing 56
Ping 58
Integration 60
OS Abstraction Layer 60
Utilities 60
PSP Porting 61

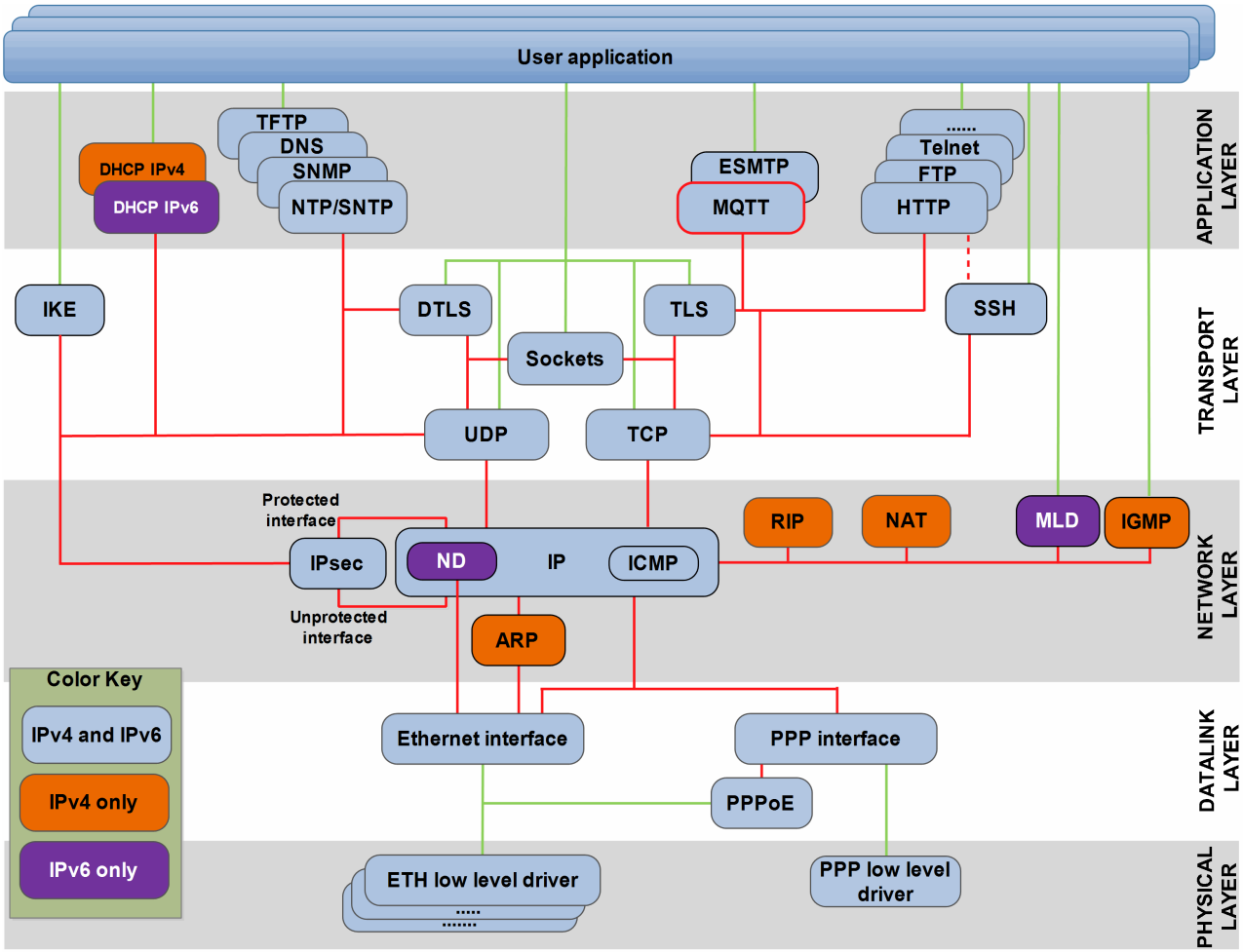
1 System Overview

1.1 Introduction

This guide is for those who want to implement an MQTT client as part of HCC Embedded's TCP/IP stack.

MQTT (originally termed Message Queueing Telemetry Transport) is a simple "publish and subscribe" messaging protocol for use over TCP/IP. It was designed to connect restricted devices in remote locations for sporadic messaging over low bandwidth, high-latency or unreliable networks, with minimal code size needed. Its original purpose was to collect data from multiple devices while using limited bandwidth and provide the information to several subscribers. It tries to ensure reliability and some degree of assurance of delivery. MQTT is now mainly used as a Machine-to-Machine (M2M) Internet of Things (IoT) connectivity protocol.

The MQTT Client module is part of the HCC MISRA-compliant TCP/IP stack, as shown below, and is designed specifically for use with it. (In this diagram green lines show interfaces available to users of the stack, red lines show interfaces internal to the TCP/IP system.)



MQTT is an Application Layer protocol that operates over TCP, normally using one of two ports: 1883 for clear data and 8883 for connections over Transport Layer Security (TLS). The protocol provides many useful capabilities, including different levels of "Quality of Service" (QoS), "client down" notification, automatic topic re-registration, and the ability to receive data from clients that have gone offline.

MQTT is ideal for mobile applications because of its small size, low power usage, minimized data packets, and efficient distribution of information to one or many receivers. It involves many "clients" communicating with a centralized server ("broker") that distributes messages among the interested clients that have subscribed to the appropriate "topic". For example, it is used in sensors communicating to a broker via satellite link, over occasional dial-up connections with healthcare providers, and in a range of home automation and small device scenarios.

MQTT enables an embedded device to publish and receive messages from the cloud using just a few lines of code. It has minimal packet overhead, compared to protocols like HTTP, and clients are easy to implement.

With an application-defined "topic", a client publishes free-format data to a broker. This data is transmitted by the broker to other client(s) that have subscribed to that topic. Through use of wildcards, a single subscription can result in data from many clients being received. Similarly, data from a single publish action may be provided to many clients.

1.2 Feature Check

The main features of the MQTT client package are the following:

- Conforms to the HCC Advanced Embedded Framework.
- Complies with the HCC MISRA-compliant TCP/IP stack.
- Designed for integration with both RTOS and non-RTOS based systems.
- Compliant with v3.1.1 of the specification, which is available at MQTT.org.
- Fully configurable.
- Users can publish and subscribe to topics.
- Supports Quality of Service (QoS) levels 0, 1 and 2.
- Supports Last Will and Testament/"Client Down" messages.
- Supports use of retained messages.
- Supports Clean Session/Continuous Session Awareness.
- Provides security using user name and password authorization.
- Provides security using TLS, with the additional option of authentication using X.509 certificates.

1.3 Publish and Subscribe

As stated above, MQTT is a "publish and subscribe" protocol. In this setup:

- A client is both the producer and consumer of MQTT data.
- The publishing and subscribing elements never need to be directly connected.
- One published message can be sent to many subscribers interested in receiving information on the topic.
- Topic subscription supports a wildcard capability that can be used to describe the subscriber's interest in types of messages, or messages from a particular sender, depending on how the application's data dictionary has been defined.
- MQTT is agnostic about the content of a message payload. It does not specify the payload layout or how data is represented in a message.

A subscriber:

- Subscribes to one or more topics.
- Can unsubscribe from a topic at any time.
- Can receive messages from multiple publishers.

A message broker:

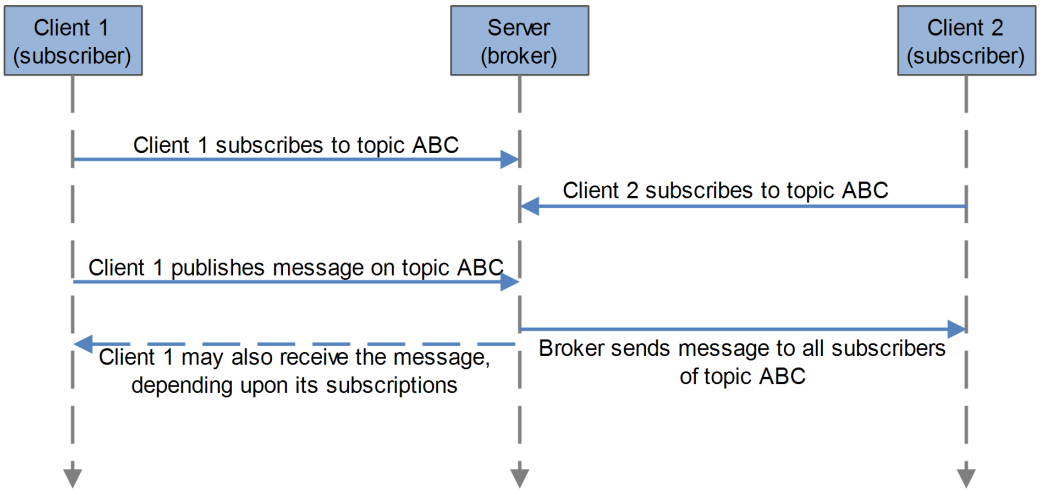
- Is an intermediary between subscribers. It is a server and the centralized system through which client data is communicated.
- Can be merely a 'Store and Forward' system or the main application that receives data and alerts from a sensor array.
- Does not just blindly pass data between the clients; it detects when a node drops off the network and can send memorized alerts to other entities interested in such notification.

With an application-defined "topic", a client publishes free-format data to a broker. This data is transmitted by the broker to other client(s) that have subscribed to that topic. Through use of wildcards, a single subscription can result in data from many clients being received. Similarly, data from a single publish action may be provided to many clients.

In a simplistic scenario, the process is as follows:

1. A client node comes online and connects to its broker, optionally providing id/password information and instructions stating what to do if fails to communicate within its keep-alive period. At this time the node may also subscribe to topics.
2. Client nodes publish messages that are received by a broker.
3. As the broker receives messages containing topics matching a node's subscription filters, the packets are transmitted to that node.
4. The broker stores and/or forwards the messages to all appropriate entities that have subscribed to the message's topic with an appropriate filter.

This diagram shows a simple publish and subscribe operation:



In an MQTT system:

- There is no inherent limitation to the number of clients the architecture can support so long as the broker can keep up with all of the traffic.
- The publisher and subscriber devices can be identical, acting as both sensors and controllers.
- The client can also provide a "Last Will and Testament" packet for transmission by the broker should communication be lost.

1.4 MQTT Features

The features of the MQTT protocol are described below.

Small Packet Overhead

MQTT control packet headers are kept as small as possible. A control packet has up to three parts:

- Two byte fixed header - this header is mandatory.
- Variable header - this may not be required, depending on the application. A variable header contains the packet identifier if the control packet uses these.
- Payload - if required, a payload up to 256 MB can be attached to a packet. Protocol defined values must be transmitted as UTF-8, but there are no requirements governing the format of published payloads.

This small header overhead makes MQTT appropriate for IoT applications as it reduces the amount of data transmitted over constrained networks.

It is easy to implement MQTT over a wide variety of IoT devices, platforms, and operating systems. Many MQTT applications can be developed by using just the CONNECT, DISCONNECT, PUBLISH and SUBSCRIBE control packets.

Control Packets

This table lists all types of control packet.

Packet	Function
CONNECT	Sent by a client requesting a connection to a server.
CONNACK	Sent by a server in response to a CONNECT received from a client.
PUBLISH	Sent by client to server or by server to client to transport an application message.
PUBACK	The response to a PUBLISH packet with QoS level 1.
PUBREC	The response to a PUBLISH packet with QoS 2. This is the second packet of the QoS 2 protocol exchange.
PUBREL	The response to a PUBREC. This is the third packet of the QoS 2 protocol exchange.
PUBCOMP	The response to a PUBREL. This is the fourth and last packet of the QoS 2 protocol exchange.
SUBSCRIBE	Sent from the client to a server to create one or more subscriptions. Each subscription registers a client's interest in one or more topics. A server sends PUBLISH packets to the client in order to forward application messages that were published to topics that match these subscriptions. The SUBSCRIBE packet also specifies (for each subscription) the maximum QoS with which the server can send application messages to the client.
SUBACK	Sent by a server to a client to confirm receipt and processing of a SUBSCRIBE packet.
UNSUBSCRIBE	Sent by a client to a server, to unsubscribe from one or more topics.
UNSUBACK	Sent by a server to a client to confirm receipt and processing of an UNSUBSCRIBE packet.
PINGREQ	Sent by a client to the server. This packet is used in Keepalive processing. It can: 1. Indicate to the server that the client is alive (if the client has sent no other control packets to the server). 2. Request that the server responds, confirming that it is alive. 3. Indicate that the network connection is active.
PINGRESP	Sent by a server to a client in response to a PINGREQ. This indicates that the server is alive.
DISCONNECT	The last control packet sent by the client to a server. This indicates that the client is disconnecting cleanly.

Quality of Service - QoS

Every message sent must have a QoS level specified. The three types of QoS are as follows:

- QoS 0 (at most once delivery) - the best effort is made to deliver messages, given the underlying network. No response is sent by the receiver and no retry is performed by the sender. The message arrives at the receiver either once or never. Messages may be lost but this level has the lowest performance impact.
- QoS 1 (at least once delivery) - messages are guaranteed to arrive at least once once but duplicates may occur.
- QoS 2 (exactly once delivery) - message are guaranteed to arrive exactly once. This is the most reliable QoS but has the greatest overhead.

The delivery protocol is symmetric; the client and server can each take the role of either sender or receiver.

When a server delivers an application message to multiple clients, each client is treated independently.

IoT devices should choose the correct QoS for their requirements. This is important for maximizing performance.

Topic-based Routing

MQTT uses a hierarchical topic-based routing scheme. This ensures fast data delivery. A topic is like a label added to every published message. It allows the broker to find all matching subscribers.

A "filter" is the string used to determine which topics are appropriate for delivery to a subscriber. Topic filters are provided by clients when they subscribe to topics and may contain topic wildcards, allowing access to multiple topics.

Topic subscription supports wildcards that can be used to describe the subscriber's interest in types of message, or messages from a specific sender, depending on how the application's data dictionary has been defined. Within topics:

- A forward slash (/) may be used to separate the levels in a topic tree and provide a hierarchical structure to topic names.
- A hash sign (#) acts as a multi-level wildcard character matching any number of levels in a topic. This wildcard represents the parent and any number of child levels.
- A plus sign (+) acts as a wildcard character that matches just one topic level.

Note: The following are the key points on wildcards:

- Where used, the '#' must be the final character in the topic filter.
- The single-level topic wildcard ('+') must occupy the entire level.

Example

In this example meters in LA and SF measure temperature and power consumption:

```
meter/CA/LA/temperature/meter-id-202
meter/CA/LA/usage/meter-id-202
meter/CA/LA/temperature/meter-id-203
meter/CA/LA/usage/meter-id-203
meter/CA/SF/temperature/meter-id-678
meter/CA/SF/usage/meter-id-678
meter/CA/SF/temperature/meter-id-701
meter/CA/SF/usage/meter-id-701
```

The topic hierarchy for these meters is shown below:

```
meter
  CA
    LA
      temperature
        meter-id-202
        meter-id-203
      usage
        meter-id-202
        meter-id-203
    SF
      temperature
        meter-id-678
        meter-id-701
      usage
        meter-id-678
        meter-id-701
```

To receive messages from all the usage meters, a subscriber could subscribe using wildcards as follows:

```
meter/CA/+/usage/#
```

Client Down Notifications

A client can provide a "Last Will and Testament" (LWT) message to a broker when it first connects to it. If that client is disconnected uncleanly in the future, for example due to a power failure, the broker delivers its LWT message to other clients. This can be used, for example, to detect when an IoT device goes offline from a network. The LWT messages can be used to notify a monitoring application on a server.

Retained Messages

Publishers can mark a message they send as "to be retained". Retained messages are stored permamently by a broker.

Clean Session or Continuous Session Awareness

MQTT sessions can survive disconnection/reconnection events. If a client device goes offline for any reason, when it reconnects the session between it and the broker is resumed. The session state can be preserved over these events and when the client reconnects any outstanding messages are delivered to it.

1.5 MQTT Security

There are two ways to obtain secure MQTT connections, as described below.

Use of TLS

When a secure connection is required, the MQTT client can use TLS to create this connection, then use that connection to perform I/O. Encryption is handled by TLS, independently of MQTT itself but with added overheads. An application may encrypt the data that it sends and receives, but again this is independent of MQTT.

Port 8883 is used for connections over TLS. The use of TLS is largely transparent to the user of the MQTT API. For more information on HCC's TLS, see the TLS and DTLS User Guide.

Authentication and Authorization

MQTT version 3.1 allows use of a user name and password within a packet to allow a client to authenticate itself with the broker.

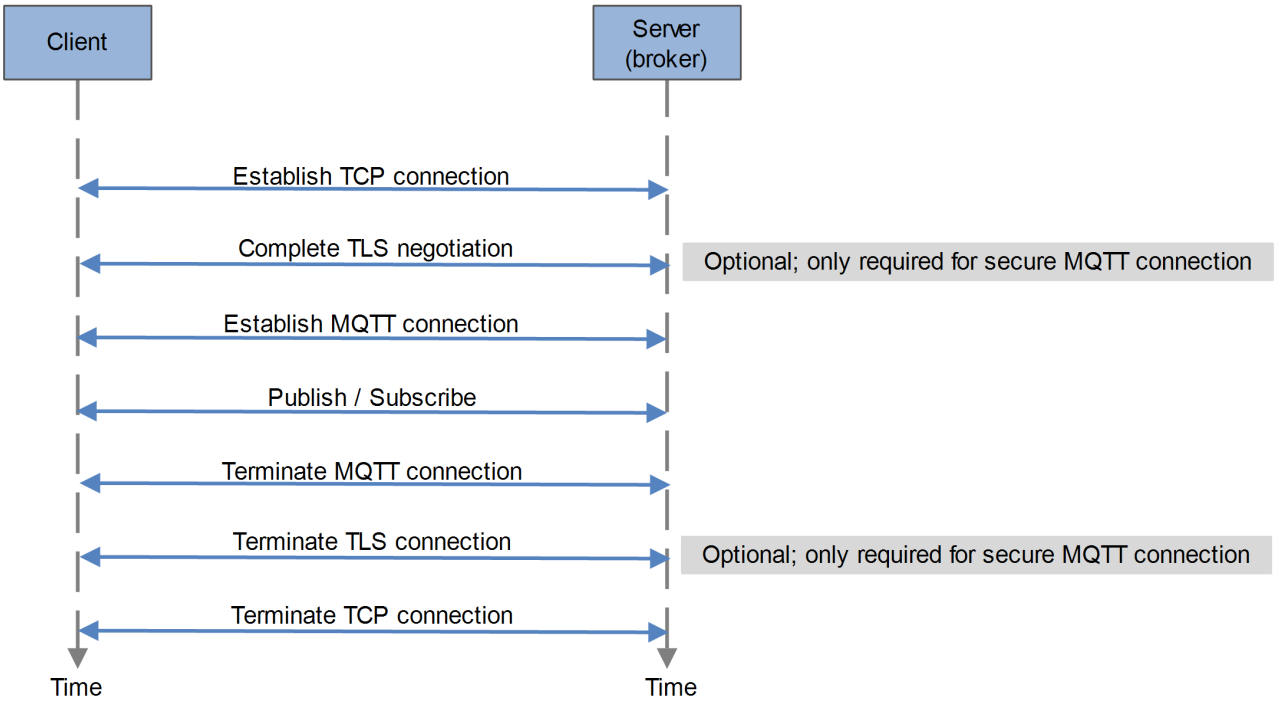
Additionally, when MQTT is run over a TLS connection, both the client and the server can authenticate each other by using X.509 certificates.

1.6 Connection Setup

This section describes MQTT's use with other protocols in the TCP/IP stack.

An MQTT client must first establish a TCP connection with the broker. After the TCP connection has been established, the MQTT client sends a CONNECT message to the broker, and waits for the receipt of a CONNACK, indicating a successful connection. A secure MQTT connection requires the successful completion of TLS negotiation between the client and the broker before the MQTT connection can be established.

This diagram shows the order in which various protocols involved in a MQTT connection exchange messages with their peer entities:


Client
Server (broker)
Establish TCP connection
Complete TLS negotiation
Optional; only required for secure MQTT connection
Establish MQTT connection
Publish / Subscribe
Terminate MQTT connection
Terminate TLS connection
Optional; only required for secure MQTT connection
Terminate TCP connection
Time
Time

1.7 Packages and Documents

Packages

This table lists the packages that need to be used with this module:

Package	Description
hcc_base_doc	This contains the two guides that will help you get started.
ip_app_mqttc	The MQTT client package (this package).
mip_tcp	The TCP package.
psp_template_base	The base Platform Support Package (PSP).
oal_base	The base OS Abstraction Layer (OAL) package.
mutil_timer	The MISRA-compliant timer utility.
ip_tls	The Transport Layer Security (TLS) package, needed if the option MQTT_SECURE_CONNECTION_ENABLED is set to 1.

Documents

For an overview of the HCC TCP/IP stack software, see Product Information on the main HCC website.

Readers should note the points in the HCC Documentation Guidelines on the HCC documentation website.

HCC Firmware Quick Start Guide

This document describes how to install packages provided by HCC in the target development environment. Also follow the *Quick Start Guide* when HCC provides package updates.

HCC Source Tree Guide

This document describes the HCC source tree. It gives an overview of the system to make clear the logic behind its organization.

HCC TCP/IP Dual Stack System User Guide

This is the core document that describes the complete TCP/IP stack. It covers both IPv4 and IPv6 systems.

HCC TLS and DTLS User Guide

This document describes HCC's Transport Layer Security (TLS) module.

HCC MQTT Client User Guide

This is this document.

1.8 Change History

This section describes past changes to this manual.

- To download earlier manuals, see Archive: MQTT Client User Guide.
- For the history of changes made to the package code itself, see History: ip_app_mqttc.

The current version of this manual is 1.50. The full list of versions is as follows:

Manual version	Date	Software version	Reason for change
1.50	2017-09-04	1.41	Corrected *Packages* list.
1.40	2017-06-23	1.41	Added refs. to demo/example code files. Added **memcmp()** to *PSP Porting*.
1.30	2017-06-20	1.01	New *Change History* format.
1.20	2017-04-06	1.01	Improved code examples.
1.00	2017-03-13	1.00	First online version.

Note: Version 1.10 was not released.

2 Source File List

This section describes all the source code files included in the system. These files follow the HCC Embedded standard source tree system, described in the *HCC Source Tree Guide*. All references to file pathnames refer to locations within this standard source tree, not within the package you initially receive.

> **Note:** Do not modify any files except the configuration file.

2.1 API Header File

The file **src/api/api_ip_app_mqttc.h** is the only file that should be included by an application using this module. For details of the API functions, see Application Programming Interface.

2.2 Configuration File

The file **src/config/config_ip_app_mqttc.h** contains all the configurable MQTT parameters. Configure these as required. For details of these options, see Configuration Options.

2.3 System Files

These files are in the directory **src/ip/apps/mqttc**. **These files should only be modified by HCC**.

File	Description
mqttc_api.c	Implements the API functions.
mqttc_io.c	Implements I/O.
mqttc_private.h	Contains module-specific information.
mqttc_task.c	Implements the task component.

2.4 Demo Code

Copies of the example code shown in Code Examples are supplied in the directory **hcc/doc/mqtt**. There is one **.c** file per example.

2.5 Version File

The file **src/version/ver_ip_app_mqttc.h** contains the version number of this module. This version number is checked by all modules that use this module to ensure system consistency over upgrades.

3 Configuration Options

Set the system configuration options in the file **src/config/config_ip_app_mqttc.h**. This section lists the available configuration options and their default values.

Note: The configuration parameters have the following effect on the client's RAM consumption:

- Packet size is directly affected by the options MQTT_CLIENT_MAX_MESSAGE_LENGTH (by default 32) and MQTT_CLIENT_MAX_TOPIC_LENGTH (by default 128). If the defaults are used for these options, the client requires approx. 900 bytes of RAM for each connection with 1 subscription and 4 packets stored.
- Every additional packet adds approx. 200 extra bytes; every additional subscription adds approx. 20 extra bytes.
- Alignment and pointer sizes have additional effects.

Be aware that the maximum packet size is limited by the maximum TCP buffer size. Using TLS further reduces the available buffer size.

MQTT_TASK_STACK_SIZE

The size of the MQTT stack. The default value is 1024. This includes the client's stack usage and the content of callbacks, so try to minimize callback implementations.

MQTT_CLIENT_MAX_ID_LENGTH

The maximum length of the client ID in bytes. The client ID is a unique identifier that the device manufacturer may give to the device. An MQTT server implementation must accept IDs of length 1-23 bytes but can optionally accept longer identifiers. Set this parameter to the maximum size you will use for this field on this range of devices. The default value is 48.

MQTT_CLIENT_MAX_USERNAME_LENGTH

The maximum length of the username in bytes. The default value is 48.

MQTT_CLIENT_MAX_PASSWORD_LENGTH

The maximum length of the password in bytes. The default value is 48.

MQTT_CLIENT_MAX_MESSAGE_LENGTH

The maximum length of a published message in bytes. The default value is 32.

MQTT_CLIENT_MAX_TOPIC_LENGTH

The maximum length of a topic (for publish and subscribe) in bytes. The default value is 128.

MQTT_CLIENT_MAX_NUMBER_OF_CONNECTIONS

The maximum number of connections that the client can maintain at the same time. The default value is 1. The range is from 1 to the limit of the RAM (see the note above).

MQTT_CLIENT_MAX_NUMBER_OF_PACKETS

The number of packets on one connection that the client can process at the same time. (The packet size always depends on the length of the parameters below). The default value is 4. The range is from 4 to the limit of the RAM (see the note above).

MQTT_CLIENT_MAX_NUMBER_OF_SUBSCRIPTIONS

The number of simultaneous subscriptions on one connection that the client supports. The default value is 1. The range is from 1 to the limit of the RAM (see the note above).

MQTT_CLIENT_MAX_BROKER_NAME_LENGTH

The maximum length of the broker name (for example, ***.broker.xively.com**) in bytes. The default value is 32. The range is from 1 to the maximum that TLS can handle (see the TLS options in the HCC TLS and DTLS User Guide).

MQTT_CLIENT_MAX_CONNECTION_TIME

The maximum time in seconds allowed for a connection procedure. The default value is 10.

MQTT_CLIENT_MAX_QUIET_TIME

The maximum time allowed for silence on the network in seconds. This is only used for cases where keep-alive is not provided. The default value is 30.

MQTT_CLIENT_MAX_RESTART_TIME

The maximum time in seconds allowed for a reconnection procedure. The default value is 10.

MQTT_CLIENT_MAX_DISCONNECT_TIME

The maximum time in seconds allowed for a disconnect procedure. The default value is 10.

MQTT_CLIENT_MAX_PACKET_TIMEOUT

The maximum time in seconds that a client can wait for a packet response to a resend event. The default value is 5.

MQTT_CLIENT_MAX_PACKET_SEND_COUNTER

The maximum number of times that a client can try to send a packet. The default value is 3.

MQTT_CLIENT_MAX_RECONNECT_COUNTER

The maximum number of reconnection attempts. The default value is 3.

MQTT_SECURE_CONNECTION_ENABLED

Set this to 1 if TLS is used. If the TLS package is not available, the module will only compile if this is set to 0 (the default value).

4 Application Programming Interface

This section describes the Application Programming Interface (API) functions. It includes all the functions that are available to an application program.

4.1 Module Management

The functions are the following:

Function	Description
mqttc_init()	Initializes the module and allocates the required resources.
mqttc_start()	Starts the module.
mqttc_stop()	Stops the module.
mqttc_delete()	Deletes the module and releases the resources it used.

mqttc_init

Use this function to initialize the MQTT Client module and allocate the required resources. Call this before any other MQTT Client function.

Format

```
t_mqttc_ret mqttc_init ( void )
```

Arguments

Arguments
None.

Return Values

Return value	Description
MQTTC_SUCCESS	Successful execution.
MQTTC_ERR_TASK	Operation failed.

mqttc_start

Use this function to start the MQTT Client module.

> **Note:** You must call **mqttc_init()** before you call this function.

Format

```
t_mqttc_ret mqttc_start ( void )
```

Arguments

Arguments
None.

Return Values

Return value	Description
MQTTC_SUCCESS	Successful execution.
MQTTC_ERR_TASK	Operation failed.

mqttc_stop

Use this function to stop the MQTT Client module.

Format

```
t_mqttc_ret mqttc_stop ( void )
```

Arguments

Arguments
None.

Return Values

Return value	Description
MQTTC_SUCCESS	Successful execution.
MQTTC_ERR_TASK	Operation failed.

mqttc_delete

Use this function to delete the MQTT Client module, releasing the associated resources.

Format

```
t_mqttc_ret mqttc_delete ( void )
```

Arguments

Arguments
None.

Return Values

Return value	Description
MQTTC_SUCCESS	Successful execution.
MQTTC_ERR_TASK	Operation failed.

4.2 Protocol Management

The functions are the following:

Function	Description
mqttc_connect()	Creates a connection and generates the CONNECT packet to send when the network connection is established.
mqttc_disconnect()	Terminates a connection.
mqttc_reconnect()	Re-establishes a previously opened connection.
mqttc_publish()	Publishes data on an available MQTT connection.
mqttc_subscribe()	Creates a new subscription.
mqttc_unsubscribe()	Cancels a subscription.
mqttc_ping()	Pings a broker.

mqttc_connect

Use this function to connect to the broker. This generates the CONNECT packet to send when the network connection is established.

This call provides the connection index that is used by other function calls. If packet creation fails, the connection is dropped immediately.

This call only starts the procedure; the connection is only considered ready for use when the broker accepts the request. At this point the user is notified by a mandatory connection callback function.

For example code, see Connecting.

Format

```
t_mqttc_ret mqttc_connect(
    t_mqttc_connect_config *   p_config,
    t_mqttc_connect_index *    p_conn_idx )
```

Arguments

Argument	Description	Type
p_config	A pointer to a connection descriptor.	t_mqttc_connect_config *
p_conn_idx	On return, a pointer to the connection index.	t_mqttc_connect_index *

Return Values

Return value	Description
MQTTC_SUCCESS	Successful execution.
MQTTC_ERR_PARAM	A parameter is invalid.
MQTTC_ERR_DUPLICATE_CONNECTION	The connection already exists.
MQTTC_ERR_NO_FREE_CONNECTION	There is no space left in the pool for the new connection.
MQTTC_ERR_NO_FREE_PACKET	There is no space left in the buffer for the new packet.

mqttc_disconnect

Use this function to terminate a connection. This call generates a DISCONNECT packet.

If the connection has any subscriptions remaining, these are cleared.

The user application is notified by a mandatory connection callback when the network connection is terminated.

For example code, see Disconnecting.

Format

```
t_mqttc_ret mqttc_disconnect( t_mqttc_connect_index conn_idx )
```

Arguments

Arguments	Description	Type
conn_idx	The connection index.	t_mqttc_connect_index

Return Values

Return value	Description
MQTTC_SUCCESS	Successful execution.
MQTTC_ERR_PARAM	The connection index is not valid.
MQTTC_ERR_INVALID_CONNECTION	The connection is not accepted.
MQTTC_ERR_STILL_CONNECTING	The connection is not yet established.
MQTTC_ERR_DISCONNECTING	The connection is terminating.
MQTTC_ERR_CONNECTION_SUSPENDED	The connection is not available at the moment.

mqttc_reconnect

Use this function to restore a connection manually.

When the connection is established, the user application is notified using the specified connection callback function.

For example code, see Reconnecting.

Format

```
t_mqttc_ret mqttc_reconnect(
    void *                    p_connection,
    t_mqttc_connect_index *   p_conn_idx )
```

Arguments

Argument	Description	Type
p_connection	A void pointer. This is the data retrieved in a connection callback function and used to restore a connection. This can contain multiple subscriptions that have already been accepted and packets that have not yet been transmitted or are waiting for responses.	void *
p_conn_idx	A pointer to the index of the restored connection.	t_mqttc_connect_index *

Return Values

Return value	Description
MQTTC_SUCCESS	Successful execution.
MQTTC_ERR_PARAM	A parameter is invalid.
MQTTC_ERR_DUPLICATE_CONNECTION	The connection already exists.
MQTTC_ERR_NO_FREE_CONNECTION	There is no space left in the pool for the new connection.
MQTTC_ERR_NO_FREE_PACKET	There is no space left in the buffer for the new packet.

mqttc_publish

Use this optional function to publish data on an available MQTT connection. It generates a PUBLISH packet.

A publication callback function can be provided to notify the user application when messages with QoS level 1 or 2 are delivered.

For example code, see Publishing.

Format

```
t_mqttc_ret mqttc_publish( t_mqttc_publish_data * p_pub_dsc )
```

Arguments

Arguments	Description	Type
p_pub_dsc	A pointer to a publication descriptor structure.	t_mqttc_publish_data *

Return Values

Return value	Description
MQTTC_SUCCESS	Successful execution.
MQTTC_ERR_PARAM	The parameter is not valid.
MQTTC_ERR_INVALID_CONNECTION	The connection is not ready for subscriptions.
MQTTC_ERR_STILL_CONNECTING	The connection is not yet established.
MQTTC_ERR_DISCONNECTING	The connection is terminating.
MQTTC_ERR_CONNECTION_SUSPENDED	The connection is not available at the moment.
MQTTC_ERR_NO_FREE_PACKET	There is no free packet in the buffer.

mqttc_subscribe

Use this function to create a new subscription (or update an existing subscription).

Subscriptions are identified by topics, so no other value than an error code is returned.

The user application is notified of the subscription status by a mandatory subscription callback function. This callback is invoked by the MQTT client when one of the following events occurs:

- The broker grants a subscription request.
- The broker denies a subscription request.
- A published message for a previously subscribed topic is received from the broker.

For example code, see Subscribing.

Format

```
t_mqttc_ret mqttc_subscribe( t_mqttc_subscribe_data * p_sub_dsc )
```

Arguments

Argument	Description	Type
p_sub_dsc	A pointer to a subscription descriptor.	t_mqttc_subscribe_data *

Return Values

Return value	Description
MQTTC_SUCCESS	Successful execution.
MQTTC_ERR_PARAM	The parameter is not valid.
MQTTC_ERR_INVALID_CONNECTION	The connection is not ready for subscriptions.
MQTTC_ERR_STILL_CONNECTING	The connection is not yet established.
MQTTC_ERR_DISCONNECTING	The connection is terminating.
MQTTC_ERR_CONNECTION_SUSPENDED	The connection is not available at the moment.
MQTTC_ERR_INVALID_SUBSCRIPTION	The subscription is either being created or terminated.
MQTTC_ERR_NO_FREE_SUBSCRIPTION	No free subscription is available.
MQTTC_ERR_NO_FREE_PACKET	There is no free packet in the buffer.

mqttc_unsubscribe

Use this function to cancel a subscription.

An application can unsubscribe from a topic if it is no longer interested in receiving messages about it.

For example code, see Unsubscribing.

Format

```
t_mqttc_ret mqttc_unsubscribe(
    t_mqttc_connect_index   conn_idx,
    char_t *                p_topic )
```

Arguments

Argument	Description	Type
conn_idx	The connection index.	t_mqttc_connect_index
p_topic	A pointer to the topic to unsubscribe from.	char_t *

Return Values

Return value	Description
MQTTC_SUCCESS	Successful execution.
MQTTC_ERR_PARAM	The parameter is not valid.
MQTTC_ERR_INVALID_CONNECTION	The connection is not ready for subscriptions.
MQTTC_ERR_STILL_CONNECTING	The connection is not yet established.
MQTTC_ERR_DISCONNECTING	The connection is terminating.
MQTTC_ERR_CONNECTION_SUSPENDED	The connection is not available at the moment.
MQTTC_ERR_INVALID_SUBSCRIPTION	The subscription is either being created or terminated.
MQTTC_ERR_NO_FREE_SUBSCRIPTION	No free subscription is available.
MQTTC_ERR_NO_FREE_PACKET	There is no free packet in the buffer.

mqttc_ping

Use this function to ping a broker. This generates PINGREQ packets.

An application can send PINGREQ packets on an established connection whenever required.

When a response from the broker arrives, the user application is notified using the connection callback.

For a usage example, see Ping.

Format

```
t_mqttc_ret mqttc_ping( t_mqttc_connect_index conn_idx )
```

Arguments

Argument	Description	Type
conn_idx	The connection index.	t_mqttc_connect_index

Return Values

Return value	Description
MQTTC_SUCCESS	Successful execution.
MQTTC_ERR_PARAM	The connection index is not valid.
MQTTC_ERR_INVALID_CONNECTION	The connection is not ready for subscriptions.
MQTTC_ERR_STILL_CONNECTING	The connection is not yet established.
MQTTC_ERR_DISCONNECTING	The connection is terminating.
MQTTC_ERR_CONNECTION_SUSPENDED	The connection is not available at the moment.
MQTTC_ERR_NO_FREE_PACKET	There is no free packet in the buffer.

4.3 Callback Functions

The functions are the following:

Function	Description
t_mqttc_connect_callback()	This is invoked by the MQTT Client when the user application has to be notified.
t_mqttc_subscribe_callback()	This is invoked by the MQTT Client when a subscription-related event occurs (subscription granted, subscription cancelled, or publish message received).
t_mqttc_publish_callback()	This is invoked by the MQTT Client when a publish message with QoS level greater than 0 is delivered.

Note:

- It is your responsibility to provide these callback functions, complying with the definitions described in this manual. The connection and subscription callbacks are mandatory.
- Callback function implementation has a direct impact on the client's stack usage, so minimize the code size.
- Use of MQTT Client API calls within callbacks is not supported.

t_mqttc_connect_callback

This callback function is invoked whenever the user application has to be notified.

This function displays the contents of the connection report structure.

For example code showing this callback used in different ways, see Connecting , Disconnecting, Reconnecting or Ping.

> **Note:** This callback is mandatory.

Format

```
void (* t_mqttc_connect_callback)( t_mqttc_connection_report * cb_report )
```

Arguments

Argument	Description	Type
cb_report	A pointer to the report containing the status returned by the connection request.	t_mqttc_connection_report *

Return Values

None.

t_mqttc_subscribe_callback

This callback function is invoked whenever a subscription-related event occurs (subscription granted, subscription cancelled, or publish message received).

This callback displays the contents of the subscription report structure.

For example code, see Subscribing or Unsubscribing.

> **Note:** This callback is mandatory.

Format

```
void (* t_mqttc_subscribe_callback)( t_mqttc_subscribe_report * cb_report )
```

Arguments

Argument	Description	Type
cb_report	A pointer to the report containing the status returned by the subscribe request.	t_mqttc_subscribe_report *

Return Values

None.

t_mqttc_publish_callback

This callback function is invoked whenever a publish message with QoS level greater than 0 is delivered.

> **Note:** This callback is optional.

For example code showing this callback, see Publishing.

Format

```
void (* t_mqttc_publish_callback)( t_mqttc_publish_info * cb_info )
```

Arguments

Argument	Description	Type
cb_info	A pointer to the report containing the status returned by the publish request. This has a status field with one of the following values: • MESSAGE_TRANSMITTED • MESSAGE_HANGUP	t_mqttc_publish_info *

Return Values

None.

4.4 Error Codes

If a function executes successfully, it returns with MQTTC_SUCCESS. The following table shows the meaning of the MQTT Client error codes.

Return Value	Description
MQTTC_SUCCESS	Successful execution.
MQTTC_ERR_PARAM	A parameter is invalid.
MQTTC_ERR_INVALID_CONNECTION	The connection is not accepted.
MQTTC_ERR_STILL_CONNECTING	The connection is not yet established.
MQTTC_ERR_NO_FREE_PACKET	There is no free packet in the buffer.
MQTTC_ERR_DISCONNECTING	The connection is terminating.
MQTTC_ERR_CONNECTION_SUSPENDED	The connection is not available at the moment.
MQTTC_ERR_DUPLICATE_CONNECTION	The connection already exists.
MQTTC_ERR_NO_FREE_CONNECTION	There is no space left in the pool for the new connection.
MQTTC_ERR_NO_FREE_PACKET	There is no space left in the buffer for the new packet.
MQTTC_ERR_INVALID_SUBSCRIPTION	The subscription is either being created or terminated.
MQTTC_ERR_NO_FREE_SUBSCRIPTION	No free subscription is available.
MQTTC_ERR_SUBSCRIPTION_NOT_ GRANTED	The subscription is not ready for termination.
MQTTC_ERR_IP_ERROR	IP error.
MQTTC_ERR_TASK	May be returned by Module Management functions.

4.5 Types and Definitions

t_mqttc_connect_config

The *t_mqttc_connect_config* structure describes a connection. It has the elements shown below.

Note: To meet size constraints, *p_broker_name* is handled by reference and it is the user's responsibility to keep this value intact for the whole session.

Element	Type	Description
broker_address	t_ip_port	The IP descriptor of the broker.
p_broker_name	char_t *	A pointer to the broker's name for certificate verification in secure connections.
client_info	s_mqttc_client_config	The client descriptor.
session_info	s_mqttc_session_config	The session descriptor.
will_data	s_mqttc_will_config	Last will and testament information.
conn_cb	t_mqttc_connect_callback	The connection callback function.

s_mqttc_client_config

The *s_mqttc_client_config* structure describes an MQTT client. It has the elements shown below.

Note: To meet size constraints, *p_clientid*, *p_username*, and *p_password* are handled by reference and it is the user's responsibility to keep these values intact for the whole session.

Element	Type	Description
client_address	t_ip_port	The IP descriptor of the broker.
p_clientid	char_t *	A pointer to the client ID.
p_username	char_t *	A pointer to the username.
p_password	uint8_t *	A pointer to the password.
password_length	uint16_t	The length of the password.

s_mqttc_session_config

The *s_mqttc_session_config* structure describes a session. It has the following elements:

Element	Type	Description
b_clean_session	uint8_t	The clean session flag.
b_secure_connection	uint8_t	The secure connection flag.
keep_alive_seconds	uint16_t	The keep-alive value in seconds.

s_mqttc_will_config

The *s_mqttc_will_config* structure describes the Last Will and Testament message. It has the elements shown below.

Note: To meet size constraints, *p_topic* and *p_message* are handled by reference and it is the user's responsibility to keep these values intact for the whole session.

Element	Type	Description
b_retain	uint8_t	The retain flag.
p_topic	char_t *	A pointer to the topic.
p_message	uint8_t *	A pointer to the message.
message_length	uint16_t	The length of the message.
qos_level	t_mqttc_qos_level	The QoS level of the message: 0, 1 or 2.

t_mqttc_subscribe_data

The *t_mqttc_subscribe_data* structure describes a subscription. It has the elements shown below.

> **Note:** To meet size constraints, *p_topic* is handled by reference and it is the user's responsibility to keep this value intact for the whole session.

Element	Type	Description
conn_idx	t_mqttc_connect_index	The index of the connection.
qos_level	t_mqttc_qos_level	The QoS level of the subscription.
subs_cb	t_mqttc_subscribe_callback	The connection's callback function.
p_topic	char_t *	A pointer to the topic.

t_mqttc_publish_data

The *t_mqttc_publish_data* structure describes a published message. It has the elements shown below.

> **Note:** To meet size constraints, *p_topic* and *p_message* are handled by reference and it is the user's responsibility to keep these intact for the whole session.

Element	Type	Description
b_retain	uint8_t	The retain flag.
p_topic	char_t *	A pointer to the publication topic.
p_message	uint8_t *	A pointer to the published message.
message_length	uint16_t	The length of the published message
conn_idx	t_mqttc_connect_index	The connection index.
qos_level	t_mqttc_qos_level	The QoS level of the published message.
pub_cb	t_mqttc_publish_callback	The publication callback function.

t_mqttc_connection_report

The *t_mqttc_connection_report* structure describes a connection-related event. It is used by the connection callback. It has the following elements:

Element	Type	Description
status	t_mqttc_connect_info	The status: see below.
connection_size	uint32_t	The size of the data pointed at. This is 0 if no data was provided. When it is not 0, the user can copy this number of bytes from the location pointed to by *connection_ptr*.
connection_ptr	void *	A pointer to the connection, unless connection size is 0.

The *status* values are as follows:

Element	Description
MQTTC_INTERNAL_ERROR	Unexpected system failure.
MQTTC_IP_ERROR	Network error.
MQTTC_DISCONNECTED	The connection terminated.
MQTTC_CONNECTION_HANGUP	No response from the broker.
MQTTC_CONNECTION_ACCEPTED	Connection accepted, status reset.
MQTTC_CONNECTION_ACCEPTED_SESSION_PRESENT	Connection accepted, status continues.
MQTTC_REFUSED_PROTOCOL	The protocol version used by the client is not supported.
MQTTC_REFUSED_ID	The client ID is not accepted.
MQTTC_SERVER_UNAVAILABLE	The broker cannot establish connection.
MQTTC_REFUSED_USER_OR_PASSWORD	Username or password not accepted.
MQTTC_INVALID_RETURN_CODE	The broker responded with invalid return code.
MQTTC_NOT_AUTHORIZED	The client is not allowed to connect.
MQTTC_BUFFER_FULL	There is no free packet.
MQTTC_PROTOCOL_ERROR	The content/packet is not supported.
MQTTC_PING_RESPONSE	The broker responded to a ping packet.

t_mqttc_subscribe_report

The *t_mqttc_subscribe_report* structure describes a subscription-related event (subscription granted, subscription cancelled, or publish message received). It is used by the subscription callback.

The structure has the following elements:

Element	Type	Description
status	t_mqttc_subscribe_info	The status: see below.
message	uint16_t	A byte stream containing the message.
message_length	uint8_t *	A pointer to the length of the message.
topic	char_t *	A pointer to the ASCII string containing the message topic that the received message has been sent to. This is not necessarily the same as the subscription topic (for example, subscription topics may use wildcards).
p_sub_filt	char_t *	A pointer to the subscription filter.

Status

The following *t_mqttc_subscribe_info* status values may be returned by the subscription callback:

Element	Description
MQTTC_SUBSCRIPTION_GRANTED_QOS0	Subscription granted with QoS level 0.
MQTTC_SUBSCRIPTION_GRANTED_QOS1	Subscription granted with QoS level 1.
MQTTC_SUBSCRIPTION_GRANTED_QOS2	Subscription granted with QoS level 2.
MQTTC_SUBSCRIPTION_REJECTED	Subscription rejected.
MQTTC_SUBSCRIPTION_CANCELLED	Subscription cancelled.
MQTTC_MESSAGE_RECEIVED	Message received.

QoS Levels

The *t_mqttc_qos_level* Quality of Service values are as follows:

Element	Description
MQTTC_QOS_LEVEL_INVALID	Invalid value.
MQTTC_QOS_LEVEL_0	QoS level 0.
MQTTC_QOS_LEVEL_1	QoS level 1.
MQTTC_QOS_LEVEL_2	QoS level 2.

4.6 Code Examples

This section gives examples showing how to code the functions and callbacks.

Copies of the example code shown here are supplied in the directory **hcc/doc/mqtt**. There is one **.c** file per example.

Connecting

This example shows ancillary code, a connection callback, and use of **mqttc_connect()**:

```
#include "api_ip_app_mqttc.h"
#include "api_ip.h"
#include "config_ip_app_mqttc.h"
#include <string.h>

static t_mqttc_connect_index  g_my_mqtt_connection;
static char                 * g_p_broker_name     = "broker.name.comes.here";
static char                 * g_p_client_id       = "my_clientID";
static char                 * g_p_username        = NULL;
static uint8_t              * g_p_password        = NULL;
static char                 * g_p_will_topic      = NULL;
static uint8_t              * g_p_will_message    = NULL;

static void my_error_parser ( t_mqttc_ret error_code )
{
  switch ( error_code )
  {
    case MQTTC_ERR_PARAM:
      /* User action */
      break;
    case MQTTC_ERR_DUPLICATE_CONNECTION:
      /* User action */
      break;
    case MQTTC_ERR_NO_FREE_CONNECTION:
      /* User action */
      break;
    case MQTTC_ERR_NO_FREE_PACKET:
      /* User action */
      break;
  } /* switch */
} /* my_error_parser */
```

```
static void my_connection_callback ( t_mqttc_connection_report * p_report )
{
  switch ( p_report->status )
  {
    case MQTTC_DISCONNECTED:
      /* User action */
      break;
    case MQTTC_INTERNAL_ERROR:
      /* User action */
      break;
    case MQTTC_IP_ERROR:
      /* User action */
      break;
    case MQTTC_PROTOCOL_ERROR:
      /* User action */
      break;
    case MQTTC_BUFFER_FULL:
      /* User action */
      break;
    case MQTTC_CONNECTION_ACCEPTED:
      /* User action */
      break;
    case MQTTC_CONNECTION_ACCEPTED_SESSION_PRESENT:
      /* User action */
      break;
    case MQTTC_REFUSED_PROTOCOL:
      /* User action */
      break;
    case MQTTC_REFUSED_ID:
      /* User action */
      break;
    case MQTTC_SERVER_UNAVAILABLE:
      /* User action */
      break;
    case MQTTC_REFUSED_USER_OR_PASSWORD:
      /* User action */
      break;
    case MQTTC_INVALID_RETURN_CODE:
      /* User action */
      break;
    case MQTTC_NOT_AUTHORIZED:
      /* User action */
      break;
    default:
      /* MQTTC_CONNECTION_HANGUP:
       * MQTTC_PING_RESPONSE: */
       break;
  } /* switch */
} /* my_connection_callback */
```

```
static void my_connect ( void )
{
  t_mqttc_connect_config  my_config;
  t_mqttc_ret             api_return;

  my_config.broker_addr.ipp_ip_addr.ipa_address[0] = 1;
  my_config.broker_addr.ipp_ip_addr.ipa_address[1] = 1;
  my_config.broker_addr.ipp_ip_addr.ipa_address[2] = 1;
  my_config.broker_addr.ipp_ip_addr.ipa_address[3] = 1;
  my_config.broker_addr.ipp_ip_addr.ipa_version = IPV_IP_V4;
  my_config.broker_addr.ipp_port = 1883;
  my_config.p_broker_name = g_p_broker_name;

  memset( &my_config.client_info.client_address, 0, sizeof( t_ip_port ) );
  my_config.client_info.client_address.ipp_ip_addr.ipa_version = IPV_IP_V4;
  my_config.client_info.client_address.ipp_port = 10500;
  my_config.client_info.p_clientid = g_p_client_id;
  my_config.client_info.p_username = g_p_username;
  my_config.client_info.p_password = g_p_password;
  my_config.client_info.password_length = strnlen( (char_t *)g_p_password
                                                  , MQTT_CLIENT_MAX_PASSWORD_LENGTH );

  my_config.session_info.b_clean_session = TRUE;
  my_config.session_info.b_secure_session = FALSE;
  my_config.session_info.keep_alive_seconds = 60;

  my_config.will_data.b_retain = FALSE;
  my_config.will_data.p_topic = g_p_will_topic;
  my_config.will_data.p_message = g_p_will_message;
  my_config.will_data.message_length = strnlen( (char_t *)g_p_will_message
                                               , MQTT_CLIENT_MAX_MESSAGE_LENGTH );
  my_config.will_data.qos_level = MQTTC_QOS_LEVEL_0;

  my_config.conn_cb = my_connection_callback;

  api_return = mqttc_connect( &my_config, &g_my_mqtt_connection );

  if ( api_return != MQTTC_SUCCESS )
  {
    my_error_parser( api_return );
  }
} /* my_connect */
```

Disconnecting

This example shows ancillary code, a connection callback, and use of **mqttc_disconnect()**:

```
#include "api_ip_app_mqttc.h"
#include "api_ip.h"

static t_mqttc_connect_index  g_my_mqtt_connection;

static void my_error_parser ( t_mqttc_ret error_code )
{
  switch ( error_code )
  {
    case MQTTC_ERR_PARAM:
      /* User action */
      break;
    case MQTTC_ERR_INVALID_CONNECTION:
      /* User action */
      break;
    case MQTTC_ERR_STILL_CONNECTING:
      /* User action */
      break;
    case MQTTC_ERR_DISCONNECTING:
      /* User action */
      break;
    case MQTTC_ERR_CONNECTION_SUSPENDED:
      /* User action */
      break;
    case MQTTC_ERR_DUPLICATE_CONNECTION:
      /* User action */
      break;
    case MQTTC_ERR_NO_FREE_PACKET:
      /* User action */
      break;
  } /* switch */
} /* my_error_parser */
```

```
static void my_connection_callback ( t_mqttc_connection_report * p_report )
{
  switch ( p_report->status )
  {
    case MQTTC_DISCONNECTED:
      /* User action */
      break;
    case MQTTC_INTERNAL_ERROR:
      /* User action */
      break;
    case MQTTC_IP_ERROR:
      /* User action */
      break;
    case MQTTC_PROTOCOL_ERROR:
      /* User action */
      break;
    default:
      /* MQTTC_CONNECTION_HANGUP:
       * MQTTC_BUFFER_FULL:
       * MQTTC_CONNECTION_ACCEPTED:
       * MQTTC_CONNECTION_ACCEPTED_SESSION_PRESENT:
       * MQTTC_REFUSED_PROTOCOL:
       * MQTTC_REFUSED_ID:
       * MQTTC_SERVER_UNAVAILABLE:
       * MQTTC_REFUSED_USER_OR_PASSWORD:
       * MQTTC_INVALID_RETURN_CODE:
       * MQTTC_NOT_AUTHORIZED:
       * MQTTC_PING_RESPONSE: */
      break;
  } /* switch */
} /* my_connection_callback */

static void my_disconnect ( void )
{
  t_mqttc_ret  api_return = mqttc_disconnect( g_my_mqtt_connection );

  if ( api_return != MQTTC_SUCCESS )
  {
    my_error_parser( api_return );
  }
}
```

Reconnecting

The **mqttc_reconnect()** function assumes that a user had a previously-saved MQTT connection and feeds this to the interface; the only place to save an MQTT connection is the connection callback function. This example code demonstrates this.

The size of the local space must always be at least as big as the connection descriptor, which consumes approx. 900 bytes with the default settings: 4 packets and 1 subscription.

Every additional packet adds approx. 200 extra bytes and every additional subscription adds approx. 20 extra bytes. The packet size is directly affected by the following configuration options:

- MQTT_CLIENT_MAX_MESSAGE_LENGTH (by default 32)
- MQTT_CLIENT_MAX_TOPIC_LENGTH (by default 128)

This example shows ancillary code, a connection callback, and use of **mqttc_reconnect()**:

```
#include "api_ip_app_mqttc.h"
#include <string.h>

static t_mqttc_connect_index  g_my_mqtt_connection;

static struct
{
  uint32_t  size;
  uint8_t   data[1024];
} g_connection_buffer;

static void my_error_parser ( t_mqttc_ret error_code )
{
  switch ( error_code )
  {
    case MQTTC_ERR_PARAM:
      /* User action */
      break;
    case MQTTC_ERR_DUPLICATE_CONNECTION:
      /* User action */
      break;
    case MQTTC_ERR_NO_FREE_CONNECTION:
      /* User action */
      break;
    case MQTTC_ERR_NO_FREE_PACKET:
      /* User action */
      break;
  } /* switch */
} /* my_error_parser */
```

```
static void my_connection_callback ( t_mqttc_connection_report * p_report )
{
  switch ( p_report->status )
  {
    case MQTTC_DISCONNECTED:
      if ( p_report->connection_ptr != NULL )
      {
        g_connection_buffer.size = p_report->connection_size;
        memcpy( &g_connection_buffer.data[0]
               , p_report->connection_ptr
               , p_report->connection_size );
      }
      else
      {
        g_connection_buffer.size = 0;
      }
      break;

    default:
      /* MQTTC_INTERNAL_ERROR
       * MQTTC_IP_ERROR:
       * MQTTC_CONNECTION_HANGUP:
       * MQTTC_REFUSED_PROTOCOL:
       * MQTTC_REFUSED_ID:
       * MQTTC_SERVER_UNAVAILABLE:
       * MQTTC_REFUSED_USER_OR_PASSWORD:
       * MQTTC_INVALID_RETURN_CODE:
       * MQTTC_NOT_AUTHORIZED:
       * MQTTC_BUFFER_FULL:
       * MQTTC_PROTOCOL_ERROR:
       * MQTTC_CONNECTION_ACCEPTED:
       * MQTTC_CONNECTION_ACCEPTED_SESSION_PRESENT:
       * MQTTC_PING_RESPONSE: */
      break;
  } /* switch */
} /* my_connection_callback */

static void my_reconnect ( void )
{
  t_mqttc_ret  api_return = mqttc_reconnect( g_connection_buffer.data
                                           , &g_my_mqtt_connection );

  if ( api_return != MQTTC_SUCCESS )
  {
    my_error_parser( api_return );
  }
}
```

Subscribing

This example shows ancillary code, a subscription callback, and use of **mqttc_subscribe()**. This assumes the connection is already established.

```
#include "api_ip_app_mqttc.h"

static t_mqttc_connect_index  g_my_mqtt_connection;
static char                 * p_my_topic = "user/topic/comes/here";

static void my_error_parser ( t_mqttc_ret error_code )
{
  switch ( error_code )
  {
    case MQTTC_ERR_PARAM:
      /* User action */
      break;
    case MQTTC_ERR_INVALID_CONNECTION:
      /* User action */
      break;
    case MQTTC_ERR_STILL_CONNECTING:
      /* User action */
      break;
    case MQTTC_ERR_DISCONNECTING:
      /* User action */
      break;
    case MQTTC_ERR_CONNECTION_SUSPENDED:
      /* User action */
      break;
    case MQTTC_ERR_INVALID_SUBSCRIPTION:
      /* User action */
      break;
    case MQTTC_ERR_NO_FREE_SUBSCRIPTION:
      /* User action */
      break;
    case MQTTC_ERR_NO_FREE_PACKET:
      /* User action */
      break;
  } /* switch */
} /* my_error_parser */
```

```
static void my_subscription_callback ( t_mqttc_subscribe_report * p_report )
{
  switch ( p_report->status )
  {
    case MQTTC_SUBSCRIPTION_GRANTED_QOS0:
      /* User action */
      break;
    case MQTTC_SUBSCRIPTION_GRANTED_QOS1:
      /* User action */
      break;
    case MQTTC_SUBSCRIPTION_GRANTED_QOS2:
      /* User action */
      break;
    case MQTTC_SUBSCRIPTION_REJECTED:
      /* User action */
      break;
    case MQTTC_MESSAGE_RECEIVED:
      /* p_report->topic, p_report->message_length and p_report->message[] fields are valid */
      break;
    default:
      /* MQTTC_SUBSCRIPTION_CANCELLED: */
      break;
  } /* switch */
} /* my_subscription_callback */

static void my_subscribe ( void )
{
  t_mqttc_ret             api_return;
  t_mqttc_subscribe_data  subscribe_config;

  subscribe_config.conn_idx = g_my_mqtt_connection;
  subscribe_config.subs_cb = my_subscription_callback;
  subscribe_config.p_topic = p_my_topic;
  subscribe_config.qos_level = MQTT_QOS_LEVEL_0; /* or MQTT_QOS_LEVEL_1 or MQTT_QOS_LEVEL_2 */

  api_return = mqttc_subscribe( &subscribe_config );

  if ( api_return != MQTTC_SUCCESS )
  {
    my_error_parser( api_return );
  }
}
```

Unsubscribing

This example shows ancillary code, a subscription callback, and use of **mqttc_unsubscribe()**. This assumes the connection is already established.

```
#include "api_ip_app_mqttc.h"

static t_mqttc_connect_index  g_my_mqtt_connection;
static char                 * p_my_topic = "user/topic/comes/here";

static void my_error_parser ( t_mqttc_ret error_code )
{
  switch ( error_code )
  {
    case MQTTC_ERR_PARAM:
      /* User action */
      break;
    case MQTTC_ERR_INVALID_SUBSCRIPTION:
      /* User action */
      break;
    case MQTTC_ERR_SUBSCRIPTION_NOT_GRANTED:
      /* User action */
      break;
    case MQTTC_ERR_INVALID_CONNECTION:
      /* User action */
      break;
    case MQTTC_ERR_STILL_CONNECTING:
      /* User action */
      break;
    case MQTTC_ERR_DISCONNECTING:
      /* User action */
      break;
    case MQTTC_ERR_CONNECTION_SUSPENDED:
      /* User action */
      break;
    case MQTTC_ERR_NO_FREE_PACKET:
      /* User action */
      break;
  } /* switch */
} /* my_error_parser */
```

```
 static void my_subscription_callback ( t_mqttc_subscribe_report * p_report )
{
  switch ( p_report->status )
  {
    case MQTTC_SUBSCRIPTION_CANCELLED:
      /* User action */
      break;
    default:
      /* MQTTC_MESSAGE_RECEIVED:
       * MQTTC_SUBSCRIPTION_GRANTED_QOS0:
       * MQTTC_SUBSCRIPTION_GRANTED_QOS1:
       * MQTTC_SUBSCRIPTION_GRANTED_QOS2:
       * MQTTC_SUBSCRIPTION_REJECTED: */
      break;
  }
}

static void my_unsubscribe ( void )
{
  t_mqttc_ret  api_return = mqttc_unsubscribe( g_my_mqtt_connection, p_my_topic );

  if ( api_return != MQTTC_SUCCESS )
  {
    my_error_parser( api_return );
  }
}
```

Publishing

This example shows ancillary code, a publication callback, and use of **mqttc_publish()**:

```
#include "api_ip_app_mqttc.h"

static t_mqttc_connect_index  g_my_mqtt_connection;
static char                 * g_p_publish_topic  = "user/topic/comes/here";
static uint8_t                g_publish_message[] = "user message in text";

static void my_error_parser ( t_mqttc_ret error_code )
{
  switch ( error_code )
  {
    case MQTTC_ERR_PARAM:
      /* User action */
      break;
    case MQTTC_ERR_INVALID_CONNECTION:
      /* User action */
      break;
    case MQTTC_ERR_STILL_CONNECTING:
      /* User action */
      break;
    case MQTTC_ERR_DISCONNECTING:
      /* User action */
      break;
    case MQTTC_ERR_CONNECTION_SUSPENDED:
      /* User action */
      break;
    case MQTTC_ERR_NO_FREE_PACKET:
      /* User action */
      break;
  } /* switch */
} /* my_error_parser */
```

```
static void my_publish_callback ( t_mqttc_publish_info cb_info )
{
  switch ( cb_info )
  {
    case MQTTC_MESSAGE_TRANSMITTED:
      /* User action */
      break;
    case MQTTC_MESSAGE_HANGUP:
      /* User action */
      break;
  } /* switch */
} /* my_publish_callback */

static void my_publish ( void )
{
  t_mqttc_publish_data  my_data;
  t_mqttc_ret           api_return;

  my_data.b_retain = FALSE;
  my_data.p_topic = g_p_publish_topic;
  my_data.p_message = g_publish_message;
  my_data.message_length = sizeof( g_publish_message ) - 1;
  my_data.conn_idx = g_my_mqtt_connection;
  my_data.qos_level = MQTTC_QOS_LEVEL_1; /* or MQTTC_QOS_LEVEL_0 or MQTTC_QOS_LEVEL_2*/
  my_data.pub_cb = my_publish_callback; /* Valid only if MQTTC_QOS_LEVEL_1 or MQTTC_QOS_LEVEL_2
*/

  api_return = mqttc_publish( &my_data );

  if ( api_return != MQTTC_SUCCESS )
  {
    my_error_parser( api_return );
  }
} /* my_publish */
```

Ping

This example shows ancillary code, a connection callback, and use of **mqttc_ping()**:

```
#include "api_ip_app_mqttc.h"

static t_mqttc_connect_index  g_my_mqtt_connection;

static void my_error_parser ( t_mqttc_ret error_code )
{
  switch ( error_code )
  {
    case MQTTC_ERR_PARAM:
      /* User action */
      break;
    case MQTTC_ERR_INVALID_CONNECTION:
      /* User action */
      break;
    case MQTTC_ERR_STILL_CONNECTING:
      /* User action */
      break;
    case MQTTC_ERR_DISCONNECTING:
      /* User action */
      break;
    case MQTTC_ERR_CONNECTION_SUSPENDED:
      /* User action */
      break;
    case MQTTC_ERR_NO_FREE_PACKET:
      /* User action */
      break;
  } /* switch */
} /* my_error_parser */
```

```
static void my_connection_callback ( t_mqttc_connection_report * p_report )
{
  switch ( p_report->status )
  {
    case MQTTC_PING_RESPONSE:
      /* User action */
      break;
    default:
      /* MQTTC_INTERNAL_ERROR:
       * MQTTC_IP_ERROR:
       * MQTTC_CONNECTION_HANGUP:
       * MQTTC_REFUSED_PROTOCOL:
       * MQTTC_REFUSED_ID:
       * MQTTC_SERVER_UNAVAILABLE:
       * MQTTC_REFUSED_USER_OR_PASSWORD:
       * MQTTC_INVALID_RETURN_CODE:
       * MQTTC_NOT_AUTHORIZED:
       * MQTTC_BUFFER_FULL:
       * MQTTC_PROTOCOL_ERROR:
       * MQTTC_CONNECTION_ACCEPTED:
       * MQTTC_CONNECTION_ACCEPTED_SESSION_PRESENT:
       * MQTTC_DISCONNECTED: */
       break;
  }
}

static void my_ping ( void )
{
  t_mqttc_ret  api_return = mqttc_ping( g_my_mqtt_connection );

  if ( api_return != MQTTC_SUCCESS )
  {
    my_error_parser( api_return );
  }
}
```

5 Integration

This section describes all aspects of the MQTT Client module that require integration with your target project. This includes porting and configuration of external resources.

5.1 OS Abstraction Layer

All HCC modules use the OS Abstraction Layer (OAL). This allows modules to run seamlessly with a wide variety of RTOSes, or without an RTOS.

The MQTT Client module uses the following OAL components:

OAL Resource	Number Required
Tasks	1
Mutexes	1
Events	1

5.2 Utilities

The MQTT Client module creates and uses a single timer in the **hcc_timer** module.

The **hcc_timer** module is included in your system when you install the base TCP/IP modules.

5.3 PSP Porting

The Platform Support Package (PSP) is designed to hold all platform-specific functionality, either because it relies on specific features of a target system, or because this provides the most efficient or flexible solution for the developer. For full details of its functions and macros, see the *HCC Base Platform Support Package User Guide*.

The MQTT Client module makes use of the following standard PSP functions:

Function	Package	Element	Description
psp_memcmp()	psp_base	psp_string	Compares two blocks of memory.
psp_memcpy()	psp_base	psp_string	Copies a block of memory. The result is a binary copy of the data.
psp_memset()	psp_base	psp_string	Sets the specified area of memory to the defined value.
psp_strncmp()	psp_base	psp_string	Compares two strings of defined length.
psp_strncpy()	psp_base	psp_string	Copies one string of defined length to another.
psp_strnlen()	psp_base	psp_string	Gets the length of a string.

The MQTT Client module makes use of the following standard PSP macros:

Macro	Package	Element	Description
PSP_RD_BE16	psp_base	psp_endianness	Reads a 16 bit value stored as big-endian from a memory location.
PSP_RD_BE32	psp_base	psp_endianness	Reads a 32 bit value stored as big-endian from a memory location.
PSP_WR_BE16	psp_base	psp_endianness	Writes a 16 bit value to be stored as big-endian to a memory location.