User's Manual

NetX Duo™
MQTT (NetX Duo MQTT) for clients

User Guide

Renesas Synergy™ Platform

Renesas Electronics
www.renesas.com

Feb 25, 2019

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.
   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user's manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/.

Renesas Synergy Specific Information

If you are using NetX Duo MQTT for the Renesas Synergy platform, please use the following information.

MQTT Installation

Page 9: If you are using Renesas Synergy SSP and the e^2 studio ISDE, MQTT will already be installed. You can ignore the Installation section.

MQTT (NetX Duo MQTT) for clients

User Guide

Express Logic, Inc.

858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259

www.expresslogic.com

Part Number: 000-1054
Revision 5.11SP1

Contents

Chapter 1

Introduction to MQTT

NetX Duo MQTT Requirements

The NetX Duo MQTT client package requires that NetX Duo (version 5.10 or later) be installed, properly configured, and the IP instance has been created. The TCP module must be enabled in the system. In addition, if TLS security is required, the NetX Secure TLS module needs to be configured according to the security parameter required by the broker.

NetX Duo MQTT Specification

NetX Duo MQTT client implement is compliant with OASIS MQTT Version 3.1.1 Oct 29th 2014. The specification can be found at:

http://mqtt.org/

NetX Duo MQTT Basic Operation

MQTT (Message Queue Telemetry Transport) is based on publisher-subscriber model. A client can publish information to other clients through a broker. A client, if interested in a topic, can subscribe to the topic through the broker. A broker is responsible for delivering published messages to its clients who subscribe to the topic. In this publisher-subscriber model, multiple clients may publish data with the same topic. A client will receive a message it publishes if the client subscribes to the same topic.

Depending on the use case, a client may choose one of the 3 QoS levels when publishing a message:

QoS 0: The message is delivered at most once. Messages sent with QoS 0 may be lost.
QoS 1: The message is delivered at least once. Messages sent with QoS 1 may be delivered more than once.
QoS 2: The message is delivered exactly once. Messages sent with QoS 2 is guaranteed to be delivered, with no duplication.

Note: This implementation of MQTT client does not support QoS level 2 messages.

Since QoS 1 and QoS 2 are guaranteed to be delivered, the broker keeps track the state of QoS 1 and QoS 2 messages sent to each client. This is particularly important for clients that expect QoS1 or QoS 2 messages. The client may be disconnected from the broker (for example when the client reboots, or the communication link is temporarily lost). The broker must store QoS 1 and QoS 2 messages so the messages can be delivered later once the client is reconnected to the broker. However, the client may choose not to receive any stale messages from the broker after reconnection. The client can do so by initiating the connection with the *clean_session* flag set to ***NX_TRUE***. In this case, upon receiving the MQTT CONNECT message, the broker shall discard any session information associated with this client, including undelivered or unconfirmed QoS 1 or QoS 2 messages. If the *clean_session* flag is to ***NX_FALSE***, the server shall resend the QoS 1 and QoS 2 messages. The MQTT Client also resends any un-acknowledged messages if *clean_session* is set to ***NX_TRUE*.** This acknowledgment is different from the TCP layer ACK, although that happens as well. The MQTT client sends an acknowledgment to the broker.

An application creates an MQTT client instance by calling ***nxd_mqtt_client_create()***. Once the client is created, the application can connect to the broker by calling ***nxd_mqtt_client_connect()***. After connecting to the broker, the client can subscribe to a topic by calling ***nxd_mqtt_client_subscribe()***, or publish a topic by calling ***nxd_mqtt_client_publish()***.

Incoming MQTT messages are stored in the receive queue in the MQTT client instance. Application retrieves these message by calling ***nxd_mqtt_client_message_get()***. If there are messages in the receive queue, the first message (e.g. the oldest) from the queue is returned to the caller. The topic string from the message is also returned.

Note that the function ***nxd_mqtt_client_message_get()*** does not block if the MQTT client receive queue is empty. The function returns immediately with the return code ***NXD_MQTT_NO_MESSAGE***. The

application shall treat this return value as an indication that the receive queue is empty, not an error.

To avoid polling the receive queue for incoming messages, the application can register a callback function with the MQTT client by calling ***nxd_mqtt_client_recieve_notify_set()***. The callback function is declared as:

```
VOID (*receive_notify_callback)(NXD_MQTT_CLIENT *client_ptr,
                                UINT message_count);
```

As the MQTT client receives messages from the broker, it invokes the callback function if the function is set. The callback function passes the pointer to the client control block and a message count value. The message count value indicates the number of MQTT messages in the receive queue. Note that this callback function executes in the MQTT client thread context. Therefore, the callback function should not execute any procedures that may block the MQTT client thread. The callback function should trigger the application thread to call ***nxd_mqtt_client_message_get()*** to retrieve the messages.

To disconnect and terminate the MQTT client service, the application shall use the service ***nxd_mqtt_client_disconnect()*** and ***nxd_mqtt_client_delete().*** Calling ***nxd_mqtt_client_disconnect()*** simply disconnects the TCP connection to the broker. It releases messages already received and stored in the receive queue. However, it does not release QoS level 1 messages in the transmit queue. QoS level 1 messages are retransmitted upon connection, assuming the ***clean_session*** flag is set to ***NX_FALSE.***

The broker may also disconnect from the client. When the TCP connection between the client and the broker is terminated, the application can be notified by the disconnect notify function. To use the notification mechanism, application installs the disconnect notify function by calling ***nxd_mqtt_client_disconnect_notify_set*.** Once a TCP disconnect is observed and the MQTT session has been created, the notification function is invoked.

Calling ***nxd_mqtt_client_delete()*** releases all message blocks in the transmit queue and the receive queue. Unacknowledged QoS level 1 messages are also deleted.

Secure MQTT Connection

The MQTT client makes a secure connection to the broker using the NetX Secure TLS module. The default port number for MQTT with TLS security is 8883, defined in ***NXD_MQTT_TLS_PORT***.

To create a secure MQTT connection to the broker, a TLS session needs to be negotiated after a TCP connection is established, before MQTT CONNECT messages can be sent to the broker. The TLS session set up is accomplished by calling ***nxd_mqtt_client_secure_connect()*** and passing in a user-defined TLS setup callback function. During the MQTT connection phase, once the TCP connection is established, the client invokes the TLS setup callback function to start a proper TLS handshake process. After the TLS session is established, the client continues the MQTT CONNECT message over the secure channel.

The user defined callback function takes five input values and is declared as:

```
UINT tls_Setup_callback(NXD_MQTT_CLIENT *client_ptr,
     NX_SECURE_TLS_SESSION *session_ptr,
     NX_SECURE_TLS_CERTIFICATE *certificate_ptr,
     NX_SECURE_TLS_CERTIFICATE *trusted_cerfiticate);
```

Below is a description of the input parameters:

`client_ptr`: Pointer to the MQTT client control block.
`session_ptr`: Pointer to the TLS session control block.
`certificate_ptr`: Pointer to the certificate control block. The setup function configures this certificate before sending it to the broker.
`trusted_certificate_ptr`: Pointer to the trusted certificate. TLS setup function configures the trusted certificate to authenticate the server.

In the TLS setup function, the application is responsible for creating a TLS session, and configuring the session with a proper certificate. The following pseudo code outlines a typical TLS session start up procedure. The reader is referred to the NetX Secure TLS User Guide for details on using TLS APIs.

Below is an example TLS setup callback:

```
UINT tls_setup_callback(NXD_MQTT_CLIENT *client_pt
          NX_SECURE_TLS_SESSION *session_ptr,
          NX_SECURE_TLS_CERTIFICATE *certrifcate_ptr,
```

```
                NX_SECURE_TLS_CERTIFICATE
                                    *trusted_certificate_ptr)

{
     /* Initialize TLS module */
     nx_secure_tls_initialize();

     /* Create a TLS session */
     nx_secure_tls_session_create(session_ptr, …);

     /* Need to allocate space for the certificate
        coming in from the broker. */
     memset(certificate_ptr), 0,
            sizeof(NX_SECURE_TLS_CERTIFICATE));

     nx_secure_tls_remote_certificate_allocate(
            session_ptr, certificate_ptr);

     /* Add a CA Certificate to our trusted store for
       verifying incoming server certificates. */
     nx_secure_tls_certificate_initialize(
            trusted_certificate_ptr,
            ca_cert_der,
            ca_cert_der_len, NULL, 0);
     nx_secure_tls_trusted_certificate_add(session_ptr,
                           trusted_certificate));


 }
```

Known Limitations of the NetX Duo MQTT Client

- NetX Duo MQTT does not support sending or receiving QoS level 2 messages.
- NetX Duo MQTT does not support chained-packets.

Chapter 2

Installation and Use of NetX Duo MQTT Client

This chapter contains a description of various issues related to installation, setup, and usage of the NetX Duo MQTT Client component.

Product Distribution

MQTT Client for NetX Duo is shipped on a single CD-ROM compatible disk. The package includes two source files, one include file, and a file that contains this document, as follows:

nxd_mqtt_client.h	Header file for MQTT Client for NetX Duo
nxd_mqtt_client.c	C Source file for MQTT Client for NetX Duo
nxd_mqtt_client.pdf	Description of MQTT Client for NetX Duo
demo_netxduo_mqtt.c	NetX Duo MQTT demonstration

MQTT Client Installation

In order to use MQTT Client for NetX Duo, the entire distribution mentioned previously should be copied to the same directory where NetX Duo is installed. For example, if NetX Duo is installed in the directory "*threadx**arm7**green*" then the *nxd_mqtt_client.h* and *nxd_mqtt_client.*c for NetX Duo MQTT Client need to be copied into this directory.

Using MQTT Client

Using MQTT Client for NetX Duo is easy. Basically, the application code must include *nxd_mqtt_client.h* after it includes *tx_api.h* and *nx_api.h*, in order to use ThreadX, and NetX Duo, respectively. Once the MQTT Client header files are included, the application code is then able to use the MQTT services described later in this guide. The application must also include *nxd_mqtt_client.c* in the build process. These files must be compiled in the same manner as other application files and its object form must be linked along with the files of the application. This is all that is required to use NetX Duo MQTT Client.

Using MQTT Client with NetX Secure TLS

To use MQTT client with NetX Secure TLS module, application must have NetX Secure TLS module installed, and include *nx_secure_tls_api.h* and *nx_crypto.h*. The MQTT library must be built with the symbol ***NX_SECURE_ENABLE*** defined.

Configuration Options

There are several configuration options for building MQTT client for NetX Duo. Following is a list of all options, where each is described in detail. The default values are listed, but can be redefined prior to inclusion of *nxd_mqtt_client.h.*

Define	Meaning
NX_DISABLE_ERROR_CHECKING	Defined, this option removes the basic MQTT client error checking. It is typically used after the application has been debugged.
NX_SECURE_ENABLE	Defined, MQTT Client is built with TLS support. Defining this symbol requires NetX Secure TLS module to be installed. *NX_SECURE_ENABLE* is not enabled by default.
NXD_MQTT_REQUIRE_TLS	Defined, application must use TLS to connect to MQTT broker. This feature requires *NX_SECURE_ENABLE* defined. By default, this symbol is not defined.
NXD_MQTT_MAX_TOPIC_NAME_LENGTH	Defines the maximum topic length (in bytes) the application is going to subscribe to. The default is 12 bytes.
NXD_MQTT_MAX_MESSAGE_LENGTH	

Defines the maximum message length (in bytes) the application is going to send or receive. The default is 32 bytes.

NXD_MQTT_KEEPALIVE_TIMER_RATE

Defines the MQTT timer rate, in ThreadX timer ticks. This timer is used to keep track of the time since last MQTT control message was sent, and sends out an MQTT PINGREQ message before the keep-alive time expires. This timer is activated if the client connects to the broker with a keep-alive timer value set. The default value is TX_TIMER_TICKS_PER_SECOND, which is a one-second timer.

NXD_MQTT_PING_TIMEOUT_DELAY

Defines the time the MQTT client waits for PINGRESP from the broker after it sends out MQTT PINGREQ. If no PINGRESP is received after this timeout delay, the client treats the broker as non-responsive and disconnects itself from the broker. The default PING timeout delay is TX_TIMER_TICKS_PER_SECOND, which is one second.

NXD_MQTT_SOCKET_TIMEOUT

Defines the time out in the TCP socket disconnect call when disconnecting from the MQTT server in timer ticks. The default value is NX_WAIT_FOREVER.

Memory Considerations

The NetX Duo MQTT client requires the application to pass in a block of memory space when creating a client instance. The MQTT client divides this memory space into equal sized message blocks, each block holding a message to be published (for QoS level 1 messages) or a message received from the broker. The message block can store a message up to ***NXD_MQTT_MAX_MESSAGE_LENGTH*** bytes, with a topic string up to ***NXD_MQTT_MAX_TOPIC_NAME_LENGTH*****.** The message block also contains internal state information of about 16 bytes. When publishing a QoS level 1 message, a message block is allocated to store the message. The message block is later released on receiving a corresponding PUBACK from the broker. For the reception path, each incoming message is stored in a message block. After the message is transferred to the application with the service ***nxd_mqtt_client_message_get***, the message block is released.

The amount of space the application creates for the MQTT client depends on the size of the topic and message, and the anticipated number of messages being transmitted and received. Once the system runs out of memory space, the application is unable to transmit QoS level 1 messages, or receive incoming messages.

Sample MQTT program

The following program illustrates a simple MQTT application. For simplicity, the return codes are assumed to be successful, therefore no further error checking is done.

```
#define  LOCAL_SERVER_ADDRESS (IP_ADDRESS(192, 168, 1, 81))


/****************************************************/
*                  IOT MQTT Client Example             */
/****************************************************/
#define  DEMO_STACK_SIZE            2048
#define  CLIENT_ID_STRING           "mytestclient"
#define  MQTT_CLIENT_STACK_SIZE     4096


/* Declare the MQTT thread stack space. */
static ULONG       mqtt_client_stack[MQTT_CLIENT_STACK_SIZE / sizeof(ULONG)];

/* Declare the MQTT client control block. */
static NXD_MQTT_CLIENT             mqtt_client;

/* Declare a 2000-byte memory space the application supplies to the MQTT
client instance. */
static ULONG                       client_memory[2000 / sizeof(ULONG)];

/* Define the symbol for signaling a received message. */
```

```
/* Define the test threads.  */
#define TOPIC_NAME                  "topic"
#define MESSAGE_STRING              "This is a message. "

/* Message buffer stores messages received from the broker. */
#define DEMO_MESSAGE_EVENT          1

/* Define the priority of the MQTT internal thread. */
#define MQTT_THREAD_PRIORTY         2

/* Define the MQTT keep alive timer for 5 minutes */
#define MQTT_KEEP_ALIVE_TIMER       300

#define QOS0                        0
#define QOS1                        1

/* Declare event flag, which is used in this demo. */
TX_EVENT_FLAGS_GROUP                mqtt_app_flag;
#define DEMO_MESSAGE_EVENT          1
#define DEMO_ALL_EVENTS             3
/* Declare buffers to hold message and topic. */
static UCHAR message_buffer[NXD_MQTT_MAX_MESSAGE_LENGTH];
static UCHAR topic_buffer[NXD_MQTT_MAX_TOPIC_NAME_LENGTH];

/* Declare the disconnect notify function. */
static VOID my_disconnect_func(NXD_MQTT_CLIENT *client_ptr)
{
    printf("client disconnected from server\n");
}


static VOID my_notify_func(NXD_MQTT_CLIENT* client_ptr, UINT
number_of_messages)
{

    tx_event_flags_set(&mqtt_app_flag, DEMO_MESSAGE_EVENT, TX_OR);
    return;

}


static ULONG    error_counter;
void demo_mqtt_client_local(NX_IP *ip_ptr, NX_PACKET_POOL *pool_ptr)
{
UINT status;
NXD_ADDRESS server_ip;
ULONG events;
UINT topic_length, message_length;

    /* Create MQTT client instance. */
    nxd_mqtt_client_create(&mqtt_client, "my_client", CLIENT_ID_STRING,
                   strlen(CLIENT_ID_STRING), ip_ptr, pool_ptr,
                   (VOID*)mqtt_client_stack, sizeof(mqtt_client_stack),
                   MQTT_THREAD_PRIORTY,
                   (UCHAR*)client_memory, sizeof(client_memory));


    /* Register the disconnect notification function. */
    nxd_mqtt_client_disconnect_notify_set(&mqtt_client, my_disconnect_func);

    /* Create an event flag for this demo. */
```

```
    status = tx_event_flags_create(&mqtt_app_flag, "my app event");

    server_ip.nxd_ip_version = 4;
    server_ip.nxd_ip_address.v4 = LOCAL_SERVER_ADDRESS;


    /* Start the connection to the server. */
    nxd_mqtt_client_connect(&mqtt_client, &server_ip, NXD_MQTT_PORT,
                            MQTT_KEEP_ALIVE_TIMER, 0, NX_WAIT_FOREVER);

    /* Subscribe to the topic with QoS level 0. */
    nxd_mqtt_client_subscribe(&mqtt_client, TOPIC_NAME, strlen(TOPIC_NAME),
                              QOS0);

    /* Set the receive notify function. */
    nxd_mqtt_client_receive_notify_set(&mqtt_client, my_notify_func);

    /* Publish a message with QoS Level 1. */
    nxd_mqtt_client_publish(&mqtt_client, TOPIC_NAME, strlen(TOPIC_NAME),
                            (CHAR*)MESSAGE_STRING, strlen(MESSAGE_STRING), 0,
                            QOS1, NX_WAIT_FOREVER);


    /* Now wait for the broker to publish the message. */

    tx_event_flags_get(&mqtt_app_flag, DEMO_ALL_EVENTS, TX_OR_CLEAR, &events,
                       TX_WAIT_FOREVER);
    if(events & DEMO_MESSAGE_EVENT)
    {
        nxd_mqtt_client_message_get(&mqtt_client, topic_buffer,
                                    sizeof(topic_buffer), &topic_length,
                                    message_buffer, sizeof(message_buffer),
                                    &message_length);
        topic_buffer[topic_length] = 0;
        message_buffer[message_length] = 0;
        printf("topic = %s, message = %s\n", topic_buffer, message_buffer);
    }

    /* Now unsubscribe the topic. */
    nxd_mqtt_client_unsubscribe(&mqtt_client, TOPIC_NAME, strlen(TOPIC_NAME));

    /* Disconnect from the broker. */
    nxd_mqtt_client_disconnect(&mqtt_client);

    /* Delete the client instance, release all the resources. */
    nxd_mqtt_client_delete(&mqtt_client);

    return;

}
```

Chapter 3

Description of NetX Duo MQTT Client Services

This chapter contains a description of all NetX Duo MQTT Client services (listed below) in alphabetical order.

In the “Return Values” section in the following API descriptions, values in **BOLD** are not affected by the **NX_DISABLE_ERROR_CHECKING** define that is used to disable API error checking, while non-bold values are completely disabled.

nxd_mqtt_client_create
Create MQTT client instance

nxd_mqtt_client_will_message_set
Set the will message

nxd_mqtt_client_client_login_set
Set MQTT client login username and password

nxd_mqtt_client_connect
Connect MQTT Client to the broker

nxd_mqtt_client_secure_connect
Connect MQTT client to the broker with TLS security

nxd_mqtt_client_publish
Publish a message through the broker.

nxd_mqtt_client_subscribe
Subscribe to a topic

nxd_mqtt_client_unsubscribe
Unsubscribe from a topic

nxd_mqtt_client_receive_notify_set
Set MQTT message receive notify callback function

nxd_mqtt_client_message_get

Retrieve a message from the broker

nxd_mqtt_client_disconnect_notify_set
Set MQTT message disconnect notify callback function

nxd_mqtt_client_disconnect
Disconnect MQTT client from the broker

nxd_mqtt_client_delete
Delete the MQTT client instance

nxd_mqtt_client_create

Create MQTT Client Instance

Prototype

```
UINT nxd_mqtt_client_create(NXD_MQTT_CLIENT *client_ptr,
                            CHAR *client_name, CHAR *client_id,
                            UINT client_id_length, NX_IP *ip_ptr,
                            NX_PACKET_POOL *pool_ptr, VOID *stack_ptr,
                            ULONG stack_size, UINT mqtt_thread_priority,
                            VOID *memory_ptr, ULONG memory_size);
```

Description

This service creates an MQTT Client instance on the specified IP instance. The *client_id* string is passed to the server during MQTT connection phase as the *Client Identifier (ClientId)*. It also creates the necessary ThreadX resources (MQTT Client task thread, mutex, event flag group, and TCP socket).

Input Parameters

client_ptr	Pointer to MQTT Client control block.
client_name	Client name string.
client_id	Client ID string used during connection phase. MQTT broker uses this client_id to uniquely identify a client.
client_id_length	Length of the client ID string, in bytes.
ip_ptr	Pointer to IP instance.
pool_ptr	Pointer to a packet pool MQTT client uses for its operation.
stack_ptr	Stack area for the MQTT Client thread.
stack_size	Size of the stack area, in bytes.
mqtt_thread_priority	The priority of the MQTT Thread.
memory_ptr	Memory area supplied to the client thread. MQTT client uses this memory space to store QoS 1 messages being published to the broker, for possible retransmission. It also stores incoming QoS 0 and QoS 1 messages.
memory_size	Size of the memory passed to the client instance, in bytes. Memory size needed for MQTT client operation depends on the amount of data being sent

or received. The minimal memory size is the size of the ***MQTT_MESSAGE_BLOCK*** structure.

Return Values

NXD_MQTT_SUCCESS	(0x00)	Successfully created MQTT client.
NXD_MQTT_INTERNAL_ERROR	(0x10004)	Internal logic error
NX_PTR_ERROR	(0x07)	Invalid MQTT control block, ip_ptr, or packet pool pointer.
NXD_MQTT_INVALID_PARAMETER	(0x10009)	Invalid will topic string, will_retrain_flag, or will_QoS value.

Allowed From

Threads

Example

```
#define CLIENT_ID_STRING "My Test Client"
#define MQTT_THREAD_PRIORITY 2
NXD_MQTT_CLIENT my_client;
NX_IP ip_0;   /* Assume ip_0 is created prior to MQTT client
                 creation. */
NX_PACKET_POOL pool_0;/* Assume pool_0 is created prior to MQTT
                         client creation. */
UCHAR  mqtt_thread_stack[STACK_SIZE];
UCHAR  mqtt_client_memory_buffer[BUFFER_SIZE];

/* Create the MQTT Client instance on "ip_0".  */
status =  nxd_mqtt_client_create(&my_client, "my client",
                          CLIENT_ID_STRING,
                          stlren(CLIENT_ID_STRING),
                          &ip_0, &pool_0,
                          (VOID*)mqtt_thread_stack, STACK_SIZE,
                          MQTT_THREAD_PRIORITY,
                          (VOID*)mqtt_client_memory_buffer,
                          BUFFER_SIZE);


/* If status is NXD_MQTT_SUCCESS an MQTT Client instance was
   successfully created.  */
```

nxd_mqtt_client_will_message_set

Sets the Will message

Prototype

```
UINT nxd_mqtt_client_will_message_set(NXD_MQTT_CLIENT *client_ptr,
                                      Const UCHAR *will_topic,
                                      UINT will_topic_length
                                      Const UCHAR *will_message,
                                      UINT will_message_length,
                                      UINT will_retain_flag,
                                      UINT will_QoS);
```

Description

This service sets the optional will topic and will message before the client connects to the server. Will topic must be UTF-8 encoded string.

The will message, if set, is transmitted to the broker as part of the CONNECT message. Therefore application wishing to use will message must use this service before the MQTT connection is make.

Input Parameters

client_ptr	Pointer to MQTT Client control block.
will_topic	UTF-8 encoded will topic string. Will topic must be present. Caller must keep the will_topic string valid till the ***nx_mqtt_client_connect*** call is made.
will_topic_length	Number of bytes in the will topic string
will_message	Application defined will message. If will message is not required, application can set this field to *NX_NULL.*
will_message_length	Number of bytes in the will message string. If will_message is set to NULL, will_message_length must be set to 0.
will_retain_flag	Whether the server publishes the will message as a retained message. Valid values are *NX_TRUE* or *NX_FALSE.*
will_QoS	QoS value used by the server when sending will message. Valid values are 0 or 1.

Return Values

NXD_MQTT_SUCCESS	(0x00)	Successfully sets the will message.
NXD_MQTT_QOS2_NOT_SUPPORTED	(0x1000C)	QoS level 2 messages are not supported.
NX_PTR_ERROR	(0x07)	Invalid MQTT control block.
NXD_MQTT_INVALID_PARAMETER	(0x10009)	Invalid will topic string, will_retrain_flag, or will_QoS value.

Allowed From

Threads

Example

```
#define WILL_TOPIC  "my_will_topic"
#define WILL_MESSAGE "my will message"


/* Create the MQTT Client instance "my_client" on "ip_0".  */
status =  nxd_mqtt_client_will_message_set(&my_client,
                          WILL_TOPIC, strlen(WILL_TOPIC),
                          WILL_MESSAGE, strlen(WILL_MESSAGE),
                          NX_TRUE, 0);

/* If status is NXD_MQTT_SUCCESS the will message is properly
   configured for the session.  It will be transmitted to the
   server during MQTT connection. */
```

nxd_mqtt_client_login_set

Sets MQTT client login username and password

Prototype

```
UINT nxd_mqtt_client_login_set(NXD_MQTT_CLIENT *client_ptr,
                               Const UCHAR *username,
                               UINT username_length
                               Const UCHAR *password,
                               UINT password_length);
```

Description

This service sets the username and password, which is used during MQTT connection phase for log in authentication purpose.

The MQTT client login with username and password is optional. In situations where the server requires a user name and password, the user name and password must be set before the connection is established.

Input Parameters

client_ptr	Pointer to MQTT Client control block.
username	UTF-8 encoded user name string. Caller must keep the username string valid till the ***nx_mqtt_client_connect*** call is made.
username_length	Number of bytes in the username string
password	Password string. If password is not required, this field may be set to NX_NULL.

Return Values

NXD_MQTT_SUCCESS	(0x00)	Successfully sets the will message.
NX_PTR_ERROR	(0x07)	Invalid MQTT control block.
NXD_MQTT_INVALID_PARAMETER	(0x10009)	Invalid username string or the password string.

Allowed From

Threads

Example

```
#define USERNAME "MY_NAME"
#define PASSWORD "MY_LOGIN_SECRET"


/* Create the MQTT Client instance "my_client" on "ip_0".  */
status =  nxd_mqtt_client_login_set(&my_client,
                            USERNAME, strlen(USERNAME),
                            PASSWORD, strlen(PASSWORD));

/* If status is NXD_MQTT_SUCCESS the username and the password
   are set for the session.  This information will be
   transmitted to the server during MQTT connection. */
```

nxd_mqtt_client_connect

Connect MQTT Client to the broker

Prototype

```
UINT nxd_mqtt_client_connect(NXD_MQTT_CLIENT *client_ptr,
                             NXD_ADDRESS *server_ip, UINT server_port,
                             UINT keepalive, UINT clean_session,
                             ULONG wait_option));
```

Description

This service initiates a connection to the broker. First it binds a TCP socket, then makes a TCP connection. Assuming that succeeds, it creates a timer if the MQTT keep alive feature is enabled. Then it connects with the MQTT server (broker).

Note that this service creates an MQTT connection with no TLS protection. To create a secure MQTT connection, the application shall use the service ***nxd_mqtt_client_secure_connect().***

Upon the connection, if the client sets the *clean_session* to NX_FALSE, the client will retransmit any messages stored that have not been acknowledged yet.

Input Parameters

client_ptr	Pointer to MQTT Client control block.
server_ip	Broker IP address.
server_port	Broker port number. The default port for MQTT is defined as ***NXD_MQTT_PORT*** (1883).
keep_alive	The keep alive value, in seconds, to be used during the session. The value indicates the maximum time between two MQTT control messages being sent to the broker before the broker times out this client. The value 0 turns off the keep-alive feature.
clean_session	Whether the server shall start this session clean. Valid options are ***NX_TRUE*** or ***NX_FALSE.***
wait_option	Connection wait time.

Return Values

NXD_MQTT_SUCCESS (0x00) Successful MQTT connection
NXD_MQTT_ALREADY_CONNECTED

	(0x10001)	The client is already connected to the broker.
NXD_MQTT_MUTEX_FAILURE		
	(0x10003)	Failed to obtain MQTT mutex
NXD_MQTT_INTERNAL_ERROR		
	(0x10004)	Internal logic error
NXD_MQTT_CONNECT_FAILURE		
	(0x10005)	Failed to connect to the broker.
NXD_MQTT_COMMUNICATION_FAILURE		
	(0x10007)	Unable to send messages to the broker.
NXD_MQTT_SERVER_MESSAGE_FAILURE		
	(0x10008)	Server responded with error
NXD_MQTT_ERROR_UNACCEPTABLE_PROTOCOL		
	(0x10081)	Server response code
NXD_MQTT_ERROR_IDENTIFYIER_REJECTED		
	(0x10082)	Server response code
NXD_MQTT_ERROR_SERVER_UNAVAILABLE		
	(0x10083)	Server response code
NXD_MQTT_ERROR_BAD_USERNAME_PASSWORD		
	(0x10084)	Server response code
NXD_MQTT_ERROR_NOT_AUTHORIZED		
	(0x10085)	Server response code
NX_PTR_ERROR	(0x07)	Invalid MQTT control block, ip_ptr, or packet pool pointer

Allowed From

Threads

Example

```
NXD_ADDRESS broker_address;

/* Set up broker IP address */
broker_address.nxd_ip_version = 4;
broker_address.nxd_ip_address.v4 = MQTT_BROKER_ADDRESS;

/* Create the MQTT Client instance "my_client" on "ip_0".  */
status =  nxd_mqtt_client_connect(&my_client, &broker_address,
                          NXD_MQTT_PORT,
                          0, /* Turn off keepalive */
                          NX_TRUE, /* Clean session flag set */
                          NX_WAIT_FOREVER);

/* If status is NXD_MQTT_SUCCESS a connection to the broker is
  successfully established. */
```

nxd_mqtt_client_secure_connect

Connect MQTT client to the broker with TLS security

Prototype

```
UINT nxd_mqtt_client_secure_connect(NXD_MQTT_CLIENT *client_ptr,
                                    NXD_ADDRESS *server_ip, UINT server_port,
                                    UINT (*tls_setup)(NXD_MQTT_CLIENT *,
                                                      NX_SECURE_TLS_SESISON *,
                                                      NX_SECURE_TLS_CERTIFICATE *,
                                                      NX_SECURE_TLS_CERTIFICATE *),
                                    UINT keepalive, UINT connection_flag,
                                    UINT clean_session, ULONG wait_option));
```

Description

This service is identical to ***nxd_mqtt_client_connect*** except that the connection goes through TLS layer instead of TCP. Therefore, communication between the client and the broker is secured.

The user-defined *tls_setup* is a callback function that the MQTT client uses prior to making a MQTT client connection. The application shall initialize NetX Secure TLS, configure security parameters, and load relevant certificates to be used during TLS handshake. The actual TLS handshake happens after a TCP connection is established on the broker's MQTT TLS port (default TCP port 8883). Once the TLS handshake is successful, the MQTT CONNECT control packet is sent via TLS.

To make secure connections, the NetX Secure TLS library must be available, and the NetX Duo MQTT client must be built with ***NX_SECURE_ENABLE*** defined.

Input Parameters

client_ptr	Pointer to MQTT Client control block.
server_ip	Broker IP address.
server_port	Broker port number. The default port for MQTT is defined as ***NXD_MQTT_TLS_PORT*** (8883).
tls_setup	User-provided TLS Setup callback function. This callback function is invoked to set up TLS client connection parameters.
keep_alive	The keep-alive value to be used during the session. The value 0 turns off the keep-alive feature.
clean_session	Whether or not the server shall start this session clean. Valid options are ***NX_TRUE*** or ***NX_FALSE.***
wait_option	Connection wait time.

Return Values

NXD_MQTT_SUCCESS	(0x00)	Successful MQTT client connection established via TLS.
NXD_MQTT_ALREADY_CONNECTED	(0x10001)	The client is already connected to the broker.
NXD_MQTT_MUTEX_FAILURE	(0x10003)	Failed to obtain MQTT mutex
NXD_MQTT_INTERNAL_ERROR	(0x10004)	Internal logic error
NXD_MQTT_CONNECT_FAILURE	(0x10005)	Failed to connect to the broker.
NXD_MQTT_COMMUNICATION_FAILURE	(0x10007)	Unable to send messages to the broker.
NXD_MQTT_SERVER_MESSAGE_FAILURE	(0x10008)	Server responded with error message.
NXD_MQTT_ERROR_UNACCEPTABLE_PROTOCOL	(0x10081)	Server response code
NXD_MQTT_ERROR_IDENTIFYIER_REJECTED	(0x10082)	Server response code
NXD_MQTT_ERROR_SERVER_UNAVAILABLE	(0x10083)	Server response code
NXD_MQTT_ERROR_BAD_USERNAME_PASSWORD	(0x10084)	Server response code
NXD_MQTT_ERROR_NOT_AUTHORIZED	(0x10085)	Server response code

NX_PTR_ERROR	(0x07)	Invalid MQTT control block or sever address structure.
NX_INVALID_PORT	(0x46)	Server port cannot be 0.
NXD_MQTT_INVALID_PARAMETER	(0x10009)	Input parameter error
NXD_MQTT_CLIENT_NOT_RUNNING	(0x1000E)	MQTT Thread has not started running yet.

Allowed From

Threads

Example

```
/* TLS setup routine. This function is responsible for setting
   up TLS parameters.*/
UINT tls_setup_callback(NXD_MQTT_CLIENT *client_ptr,
                NX_SECURE_TLS_SESSION *session_ptr,
                NX_SECURE_TLS_CERTIFICATE *certificate_ptr,
                NX_SECURE_TLS_CERTIFICATE *trusted_certificate,
                UINT timeout)
{
    /* Note this routine is simplified to highlight the
       necessary steps to setup a TLS session.  Each
       application may employ different procedures suitable for
       its TLS settings, such as cipher suite, certificates. */

    /* Create a TLS session for the MQTT connection, and pass
       in various crypto methods this session can use for the
       initial TLS handshake. */

    /* Load appropriate certificates, or set up session key for
       Pre-share key operation. */

    /* Start the TLS session */

    /* Return NX_SUCCESS if the TLS session is established. */

    return(NX_SUCCESS);
}

NXD_ADDRESS broker_address;

/* Set up broker IP address */
broker_address.nxd_ip_version = 4;
broker_address.nxd_ip_address.v4 = MQTT_BROKER_ADDRESS;
```

```
/* Create the MQTT Client instance “my_client” on “ip_0”.  */
status =  nxd_mqtt_client_secure_connect(&my_client,
                                &server_address, NXD_MQTT_TLS_PORT,
                                tls_setup_callback,
                                0, /* Turn off keepalive */
                                NX_TRUE, /* Clean session set */
                                NX_WAIT_FOREVER);

 /* If status is NXD_MQTT_SUCCESS the MQTT Client was
    successfully connected to the broker via TLS.  */
```

nxd_mqtt_client_publish

Publish a message through the broker

Prototype

```
UINT nxd_mqtt_client_publish(NXD_MQTT_CLIENT *client_ptr,
                             CHAR *topic_name, UINT topic_name_length,
                             CHAR *message, UINT message_length,
                             UINT retain, UINT QoS, ULONG timeout);
```

Description

This service publishes a message through the broker. Publishing QoS level 2 messages is not supported yet.

Input Parameters

client_ptr	Pointer to MQTT Client control block.
topic_name	Topic to publish to.
topic_name_length	Length of the topic, in bytes.
message	Pointer to the message buffer.
message_length	Size of the message, in bytes
retain	Determines if the broker shall retain the message.
QoS	The desired QoS value: 0 or 1.
timeout	Timeout value

Return Values

NXD_MQTT_SUCCESS	(0x00)	Successful MQTT Client create
NXD_MQTT_INTERNAL_ERROR	(0x10004)	Internal logic error.
NXD_MQTT_PACKET_POOL_FAILURE	(0x10006)	Failed to obtain packet from the packet pool.
NXD_MQTT_COMMUNICATION_FAILURE	(0x10007)	Failed to communication with the broker.
NXD_MQTT_QOS2_NOT_SUPPORTED	(0x1000C)	QoS level 2 messages are not supported.
NX_PTR_ERROR	(0x07)	Invalid MQTT control block, ip_ptr, or packet pool pointer
NXD_MQTT_INVALID_PARAMETER	(0x10009)	Input parameter error

Allowed From

Threads

Example

```
CHAR *topic = "temperature";
CHAR *message = "100";

/* Publish the temperature value.   */
status =  nxd_mqtt_client_publish(&my_client,
                    topic, strlen(topic),
                    message, strlen(message),
                    NX_TRUE, /* Server retains message. */);
                    0, /* QOS */
                    NX_WAIT_FOREVER);

/* If status is NXD_MQTT_SUCCESS the message has been
   successfully sent to the broker.   */
```

nxd_mqtt_client_subscribe

Subscribe to a topic

Prototype

```
UINT nxd_mqtt_client_subscribe(NXD_MQTT_CLIENT *mqtt_client_ptr,
                               CHAR *topic_name,
                               UINT topic_name_length, UINT QoS);
```

Description

This service subscribes to a specific topic. Subscribing to QoS level 2 messages is not supported yet.

Input Parameters

client_ptr	Pointer to MQTT Client control block.
topic_name	Topic to publish to.
topic_name_length	Length of the topic, in bytes.
QoS	The desired QoS level: 0 or 1.

Return Values

NXD_MQTT_SUCCESS	(0x00)	Successfully subscribed to the topic.
NXD_MQTT_NOT_CONNECTED	(0x10002)	The client is not connected to the broker.
NXD_MQTT_MUTEX_FAILURE	(0x10003)	Failed to obtain MQTT mutex
NXD_MQTT_INTERNAL_ERROR	(0x10004)	Internal logic error
NXD_MQTT_COMMUNICATION_FAILURE	(0x10007)	Unable to send messages to the broker.
NXD_MQTT_QOS2_NOT_SUPPORTED	(0x1000C)	QoS level 2messages are not supported.
NX_PTR_ERROR	(0x07)	Invalid MQTT control block, ip_ptr, or packet pool pointer
NXD_MQTT_INVALID_PARAMETER		

(0x10009)	topic_name is not set, or topic_name_length is zero, or QoS is value is invalid.

Allowed From

Threads

Example

```
/* Subscribe to the topic "temperature" with QoS level 0   */
CHAR *topic = "temperature";

status =  nxd_mqtt_client_subscribe(&my_client, topic,
                                    strlen(topic), 0);

/* If status is NXD_MQTT_SUCCESS, the client successfully
   subscribes to the topic "temperate". At this point the client
   is ready for receiving messages from the broker. */
```

nxd_mqtt_client_unsubscribe

Unsubscribe from a topic

Prototype

```
UINT nxd_mqtt_client_unsubscribe(NXD_MQTT_CLIENT *mqtt_client_pr,
                                 CHAR *topic_name,
                                 UINT topic_name_length);
```

Description

This service unsubscribes from a topic.

Input Parameters

client_ptr	Pointer to MQTT Client control block.
topic_name	Topic to unsubscribe from.
topic_name_length	Length of the topic, in bytes.

Return Values

NXD_MQTT_SUCCESS	(0x00)	Successfully unsubscribed from the topic.
NXD_MQTT_NOT_CONNECTED	(0x10002)	The client is not connected to the broker.
NXD_MQTT_MUTEX_FAILURE	(0x10003)	Failed to obtain MQTT mutex.
NXD_MQTT_INTERNAL_ERROR	(0x10004)	Internal logic error
NXD_MQTT_COMMUNICATION_FAILURE	(0x10007)	Unable to send messages to the broker.
NX_PTR_ERROR	(0x07)	Invalid MQTT control block pointer
NXD_MQTT_INVALID_PARAMETER	(0x10009)	topic_name is not set, or topic_name_length is zero.

Allowed From

Threads

Example

```
/* Subscribe to the topic “temperature” with QoS level 0   */
CHAR *topic = “temperature”;

status =  nxd_mqtt_client_unsubscribe(&my_client, topic,
                                      strlen(topic));

/* If status is NXD_MQTT_SUCCESS, the client successfully
   unsubscribes the topic “temperate”. */
```

nxd_mqtt_client_receive_notify_set

Set MQTT message receive notify callback function

Prototype

```
UINT nxd_mqtt_client_receive_notify_set(NXD_MQTT_CLIENT *client_ptr,
       VOID(*receive_notify)(NXD_MQTT_CLIENT* client_ptr,
                             UINT message_count));
```

Description

This service registers a callback function with the MQTT client. Upon receiving a message published by the broker, MQTT client stores the message in the receive queue. If the callback function is set, the callback function is invoked to notify the application that a message is ready to be retrieved. The receive notify function takes a pointer to the MQTT client control block, and a *message_count* indicating the number of messages available in the receive queue. Note that the number may change between the receive notication and when the application retrieves these messages, as new messages may have arrived in the interval.

Input Parameters

client_ptr	Pointer to MQTT Client control block.
receive_notify	User supplied callback function to be invoked on receiving a message.

Return Values

NXD_MQTT_SUCCESS	(0x00)	Successfully set the receive notify function.
NX_PTR_ERROR	(0x07)	Invalid MQTT control block.

Allowed From

Threads

Example

```
/* Sample MQTT receive notify function.  */
VOID my_notify_func(NXD_MQTT_CLIENT* client_ptr,
                    UINT message_count)
{
   /* On receiving a message, set an event flag to wake up the
      application thread.  The message will be received and
      processed in the application thread. */
   tx_event_flags_set(&mqtt_app_flag,
                      MESSAGE_RECEIVED_EVENT, TX_OR);

   /* All done.  Return to the caller. */
   return;
}


/* Set the receive callback function.  */
status =  nxd_mqtt_client_receive_notify_set(&my_client,
                                             my_notify_func);


/* If status is NXD_MQTT_SUCCESS the notify function is properly
   set.  */
```

nxd_mqtt_client_message_get

Retrieve a message from the broker

Prototype

```
UINT nxd_mqtt_client_message_get(NXD_MQTT_CLIENT *client_ptr,
            UCHAR *topic_buffer, UINT topic_buffer_size,
            UINT *actual_topic_length, UCHAR *message_buffer,
            UINT message_buffer_size, UINT *actual_message_length)
```

Description

This service retrieves a message published by the broker. All incoming messages are stored in the receive queue. The application uses this service to retrieve these messages. This call is non-blocking. If the receive queue is empty, this service returns ***NXD_MQTT_NO_MESSAGE***. An application wishing to be notified of incoming message can call the service ***nxd_mqtt_client_receive_notify_set*** to register a receive callback function.

The caller needs to provide memory space for the topic string and the message body. The sizes of these two buffers are passed in using *topic_buffer_size* and *message_buffer_size*. The actual number of bytes in the topic string and the message body are returned in *actual_topic_length* and *actual_message_length*. If topic length or massage length is greater than the buffer space provided, this service returns error code *NXD_MQTT_INSUFFICIENT_BUFFER_SIZE*. The application shall allocate a bigger buffer and try again.

Input Parameters

client_ptr Pointer to MQTT Client control block.
topic_buffer Pointer to the memory location where the topic string is copied into.
topic_buffer_size Size of the topic buffer.
actual_topic_length Pointer to the memory location where the actual topic length is returned.
message_buffer Pointer to the memory location where the message string is copied into.
message_buffer_size Size of the message buffer.
actual_message_length Pointer to the memory location where the message length is returned.

Return Values

NXD_MQTT_SUCCESS	(0x00)	Successfully retrieved message.
NXD_MQTT_INTERNAL_ERROR	(0x10004)	Internal logic error
NXD_MQTT_NO_MESSAGE	(0x1000A)	The receive queue is empty.
NXD_MQTT_INSUFFICIENT_BUFFER_SIZE	(0x1000D)	Topic buffer or message buffer is too small for the topic or the message.
NX_PTR_ERROR	(0x07)	Invalid MQTT control block, ip_ptr, or packet pool pointer
NXD_MQTT_INVALID_PARAMETER	(0x10009)	message_buffer or topic_buffer pointer is NULL

Allowed From

Threads

Example

```
UCHAR topic[MAX_TOPIC_SIZE];
UCHAR message[MAX_TOPIC_SIZE];;
UINT topic_length;
UINT message_length;

/* Retrieve a message from MQTT client receive queue.  */
status =  nxd_mqtt_client_message_get(&my_client, topic,
                    sizeof(toipic), &topic_length, message,
                    sizeof(message), &message_length);

/* Check the return value. */
if(status == NXD_MQTT_SUCCESS)
{
   /* A message is received. All done. */
}
else if (status == NXD_MQTT_NO_MESSAGE)
{
    /* No more messages in the receive queue.  All done. */

}
else
{
     /* Receive error. */
}
```

nxd_mqtt_client_disconnect_notify_set

Set MQTT message disconnect notify callback function

Prototype

```
UINT nxd_mqtt_client_disconnect_notify_set(
       NXD_MQTT_CLIENT *client_ptr,
       VOID(*disconnect_notify)(NXD_MQTT_CLIENT* client_ptr));
```

Description

This service registers a callback function with the MQTT client. When MQTT detects the connection to the broker is lost, it calls this notify function to alert the application. Therefore, the application can use this callback function to detect a lost connection, and to be able to re-establish connection to the broker.

```
VOID callback_func(NXD_MQTT_CLIENT *client_ptr);
```

Input Parameters

client_ptr	Pointer to MQTT Client control block.
disconnect_notify	User supplied callback function to be invoked when MQTT detects the connection to the broker is lost.

Return Values

NXD_MQTT_SUCCESS	(0x00)	Successfully set the disconnect notify function.
NX_PTR_ERROR	(0x07)	Invalid MQTT control block.

Allowed From

Threads

Example

```
VOID disconnect_notify(NXD_MQTT_CLIENT *client_ptr)
{
     /* MQTT client is disconnected from the broker.  Notify the
        application so it can re-connect to the broker. */
}

status = nxd_mqtt_client_disconnect_notify_set(client_ptr,
                                               disconnect_notify);
```

nxd_mqtt_client_disconnect

Disconnect MQTT client from the broker

Prototype

```
UINT nxd_mqtt_client_disconnect(NXD_MQTT_CLIENT *client_ptr);
```

Description

This service disconnects the client from the broker. Note that messages on the receive queue are released. Messages with QoS 1 in the transmit queue are not released. After the client reconnects to the server, QoS 1 messages can be processed, unless the client reconnects to the server with *clean_session* flag set to ***NX_TRUE***.

If the connection was made with TLS security protection, this service will close the TLS session before disconnecting the TCP connection.

The actual TCP socket disconnect call has a wait option defined by NXD_MQTT_SOCKET_TIMEOUT (timer ticks). The default value is NX_WAIT_FOREVER. To avoid indefinite suspension in the event that the network connection is lost or the server is not responding, set this option to a finite value.

Input Parameters

client_ptr	Pointer to MQTT Client control block.

Return Values

NXD_MQTT_SUCCESS	(0x00)	Successfully disconnected from broker
NXD_MQTT_MUTEX_FAILURE	(0x10003)	Failed to obtain MQTT mutex.
NX_PTR_ERROR	(0x07)	Invalid MQTT control block

Allowed From

Threads

Example

```
/* Disconnect from the broker.  */
status =  nxd_mqtt_client_disconnect(&my_client);
```

```
/* If status is NXD_MQTT_SUCCESS the client is successfully
   disconnected from the broker.  */
```

nxd_mqtt_client_delete

Delete the MQTT client instance

Prototype

```
UINT nxd_mqtt_client_delete(NXD_MQTT_CLIENT *client_ptr);
```

Description

This service deletes the MQTT client instance and releases internal resources. This service automatically disconnects the client from the broker if it is still connected. Messages not yet transmitted or not been acknowledged are released. Messages received but not retrieved by the application are also released.

If the connection was made with TLS security protection, this service closes the TLS session before disconnecting the TCP connection.

After the client is deleted, an application wishing to use MQTT service needs to create a new instance.

Input Parameters

client_ptr	Pointer to MQTT Client control block.

Return Values

NXD_MQTT_SUCCESS	(0x00)	Successfully deleted MQTT client.
NX_PTR_ERROR	(0x07)	Invalid MQTT control block

Allowed From

Threads

Example

```
/* Delete the MQTT client instance.  */
status = nxd_mqtt_client_delete(&my_client);
```

```
/* If status is NXD_MQTT_SUCCESS the client is successfully
   deleted from the system.  */
```

MQTT (NetX Duo MQTT) for clients User Guide

Publication Date: Rev.5.13 Feb 25, 2019

Published by: Renesas Electronics Corporation

MQTT (NetX Duo MQTT) for clients
User Guide

Renesas Electronics Corporation

R11UM0068EU0513