wolfMQTT User Manual

December 20, 2016

Version 0.12

Table of Contents

Chapter 1: Introduction

1.1 Protocol overview

Chapter 2: Building wolfMQTT

2.1 Getting the Source Code

2.2 Building on *nix

2.3 Building on Windows

2.4 Building in a Non-Standard Environment

2.5 Cross Compiling

2.6 Install to Custom Directory

Chapter 3: Getting Started

3.1 Client Example

3.2 Firmware Update Example

3.3 Azure IoT Hub Example

3.4 AWS IoT Example

Chapter 4: Library Design

4.1 Example Design

Chapter 5: API Reference

4.1 MqttPacketResponseCodes (enum)

4.2 MqttClient_Init

4.3 MqttClient_Connect

4.4 MqttClient_Publish

4.5 MqttClient_Subscribe

4.6 MqttClient_Unsubscribe

4.7 MqttClient_Ping

4.8 MqttClient_Disconnect

4.9 MqttClient_WaitMessage

4.10 MqttClient_NetConnect

4.11 MqttClient_NetDisconnect

4.12 MqttClient_ReturnCodeToString

Chapter 1: Introduction

This is an implementation of the MQTT (Message Queuing Telemetry Transport) Client written in C. This library was built from the ground up to be multi-platform, space conscience and extensible. It supports all Packet Types, all Quality of Service (QoS) levels 0-2 and supports SSL/TLS using the wolfSSL library. This implementation is based on the MQTT v3.1.1 specification.

1.1 Protocol Overview

MQTT is a lightweight open messaging protocol that was developed for constrained environments such as M2M (Machine to Machine) and IoT (Internet of Things), where a small code footprint is required. MQTT is based on the Pub/Sub messaging principle of publishing messages and subscribing to topics. The protocol efficiently packs messages to keep the overhead very low. The MQTT specification recommends TLS as a transport option to secure the protocol using port 8883 (secure-mqtt). Constrained devices can benefit from using TLS session resumption to reduce the reconnection cost.

MQTT defines QoS levels 0-2 to specify the delivery integrity required.
0 = At most once delivery: No acknowledgment.
1 = At least once delivery: Sends acknowledgment (PUBLISH_ACK).
2 = Exactly once delivery: Sends received (PUBLISH_REC), gets back released (PUBLISH_REL) and then sends complete (PUBLISH_COMP).

Highlights:

- A publish message payload can be up to 256MB (28 bits).
- Packet header remaining length is encoded using a scheme where the most significant bit (7) indicates an additional length byte.
- Packets which require a response must include a 16-bit packet Id. This needs to be unique for any outstanding transactions. Typically an incremented value.
- A client can provide a last will and testament upon connect, which will be delivered when the broker sees the client has disconnected or network keep-alive has expired.

- The packet types are: CONNECT, CONNECT_ACK, PUBLISH, PUBLISH_ACK, PUBLISH_REC, PUBLISH_REL, PUBLISH_COMP, SUBSCRIBE, SUBSCRIBE_ACK, UNSUBSCRIBE, UNSUBSCRIBE_ACK, PING_REQ, PING_RESP and DISCONNECT.
- The connect packet contains the ASCII string “MQTT” to define the protocol name. This can be useful for wireshark/sniffing.
- Multiple topics can be subscribed or unsubscribed in the same packet request.
- Each subscription topic must define a QoS level. The QoS level is confirmed in the subscription acknowledgment.
- A publish message can be sent or received by either the client or the broker.
- Publish messages can be flagged for retention on the broker.
- A QoS level 2 requires two round-trips to complete the delivery exchange confirmation.
- Strings are UTF-8 encoded.

See http://mqtt.org/documentation for additional MQTT documentation.

Chapter 2: Building wolfMQTT

wolfMQTT was written with portability in mind, and should generally be easy to build on most systems. If you have difficulty building, please don't hesitate to seek support through our **support forums** (http://www.wolfssl.com/forums) or contact us directly at **support@wolfssl.com**.

This chapter explains how to build wolfMQTT on Unix and Windows, and provides guidance for building in a non-standard environment. You will find a getting started guide and example client in **Chapter 3**.

When using the autoconf / automake system to build, wolfMQTT uses a single Makefile to build all parts and examples of the library, which is both simpler and faster than using Makefiles recursively.

If using the TLS features or the Firmware/Azure IoT Hub examples you'll need to have wolfSSL installed. For wolfSSL and wolfMQTT we recommend using config options "./configure --enable-ecc --enable-supportedcurves --enable-base64encode". For wolfSSL use "make && sudo make install". If you get an error locating the libwolfssl.so run "ldconfig" from the wolfSSL directory.

2.1 Getting the Source Code

The most recent version can be downloaded from the GitHub website here: https://github.com/wolfSSL/wolfMQTT

Either click the "Download ZIP" button or use the command "git clone git@github.com:wolfSSL/wolfMQTT.git"

2.2 Building on *nix

When building on Linux, *BSD, OS X, Solaris, or other *nix-like systems, use the

autoconf system. To build wolfMQTT you only need to run three commands:

```
./configure
make
```

You can append any number of build options to ./configure. For a list of available build options run:

```
./configure --help
```

from the command line to see a list of possible options to pass to the ./configure script. To build wolfMQTT, run:

```
make
```

To install wolfMQTT run:

```
make install
```

You may need superuser privileges to install, in which case precede the command with sudo:

```
sudo make install
```

To test the build, run the *mqttclient* program from the root wolfMQTT source directory:

```
./examples/mqttclient/mqttclient
```

If you want to build only the wolfMQTT library and not the additional items (examples), you can run the following command from the wolfMQTT root directory:

```
make src/libwolfmqtt.la
```

2.3 Building on Windows

Visual Studio 2015: The wolfmqtt.sln solution is included for Visual Studio 2015 in the root directory of the install.

To test each build, choose “Build All” from the Visual Studio menu and then run the mqttclient program. To edit build options in the Visual Studio project, select your desired project (wolfmqtt, mqttclient) and browse to the “Properties” panel.

For instructions on building the required wolfssl.dll see https://www.wolfssl.com/wolfSSL/Docs-wolfssl-visual-studio.html. When done copy the “wolfssl.dll” and “wolfssl.lib” into the wolfMQTT root. The project also assumes the wolfSSL headers are located “../wolfssl/”.

Cygwin: If using Cygwin, or other toolsets for Windows that provides *nix-like commands and functionality, please follow the instructions in section 2.2, above, for “Building on *nix”. If building wolfMQTT for Windows on a Windows development machine, we recommend using the included Visual Studio project files to build wolfMQTT.

2.4 Building in a non-standard environment

While not officially supported, we try to help users wishing to build wolfMQTT in a non-standard environment, particularly with embedded and cross-compilation systems. Below are some notes on getting started with this.

1. The source and header files need to remain in the same directory structure as they are in the wolfMQTT download package.

2. Some build systems will want to explicitly know where the wolfMQTT header files are located, so you may need to specify that. They are located in the <wolfmqtt_root>/wolfmqtt directory. Typically, you can add the <wolfmqtt_root> directory to your include path to resolve header problems.

3. wolfMQTT defaults to a little endian system unless the configure process detects big endian. Since users building in a non-standard environment aren't using the configure process, **BIG_ENDIAN_ORDER** will need to be defined if using a big endian system.

4. Try to build the library, and let us know if you run into any problems. If you need

help, contact us at *support@wolfssl.com*.

2.5 Cross Compiling

Many users on embedded platforms cross compile for their environment. The easiest way to cross compile the library is to use the ./configure system. It will generate a Makefile which can then be used to build wolfMQTT.

When cross compiling, you'll need to specify the host to ./configure, such as:

```
./configure --host=arm-linux
```

You may also need to specify the compiler, linker, etc. that you want to use:

```
./configure --host=arm-linux CC=arm-linux-gcc AR=arm-linux-ar
RANLIB=arm-linux
```

After correctly configuring wolfMQTT for cross-compilation, you should be able to follow standard autoconf practices for building and installing the library:

```
make
sudo make install
```

If you have any additional tips or feedback about cross compiling wolfMQTT, please let us know at info@wolfssl.com.

2.6 Install to Custom Directory

To setup a custom install directory for wolfSSL use the following:

In wolfSSL:

1. ./configure --prefix=~/wolfssl
2. Make
3. make install

This will place the libs in ~/wolfssl/lib and includes in ~/wolfssl/include

To setup a custom install directory for wolfMQTT and specify custom wolfSSL lib/include directories use the following:

In wolfMQTT:

1. ./configure --prefix=~/wolfmqtt --libdir=~/wolfssl/lib --includedir=~/wolfssl/include
2. Make
3. make install

Make sure the paths above match your actual location.

Chapter 3 : Getting Started

Here are the steps for creating your own implementation:

1. Create network callback functions for Connect, Read, Write and Disconnect. See `examples/mqttnet.c` and `examples/mqttnet.h` for reference implementation.
2. Define the network callback functions and context in a `MqttNet` structure.
3. Call `MqttClient_Init` passing in a `MqttClient` structure pointer, `MqttNet` structure pointer, `MqttMsgCb` function callback pointer, TX/RX buffers with maximum length and command timeout.
4. Call `MqttClient_NetConnect` to connect to broker over network. If `use_tls` is non-zero value then it will perform a TLS connection. The TLS callback `MqttTlsCb` should be defined for WolfSSL certificate configuration.
5. Call `MqttClient_Connect` passing pointer to `MqttConnect` structure to send MQTT connect command and wait for Connect Ack.
6. Call `MqttClient_Subscribe` passing pointer to `MqttSubscribe` structure to send MQTT Subscribe command and wait for Subscribe Ack (depending on QoS level).
7. Call `MqttClient_WaitMessage` passing pointer to `MqttMessage` to wait for incoming MQTT Publish message.

3.1 Client Example

The example MQTT client is located in /examples/mqttclient/. This example exercises all exposed API’s and prints any incoming publish messages for subscription topic “wolfMQTT/example/testTopic”.

Usage

./examples/mqttclient/mqttclient -?

mqttclient:

-?	Help, print this usage
-h <host>	Host to connect to, default iot.eclipse.org
-p <num>	Port to connect on, default: Normal 1883, TLS 8883
-t	Enable TLS
-c <file>	Use provided certificate file
-q <num>	Qos Level 0-2, default 0

-s Disable clean session connect flag
-k <num> Keep alive seconds, default 60
-i <id> Client Id, default WolfMQTTClient
-l Enable LWT (Last Will and Testament)
-u <str> Username
-w <str> Password
-n <str> Topic name, default wolfMQTT/example/testTopic
-r Set Retain flag on publish message
-C <num> Command Timeout, default 30000ms
-T Test mode

Output (no TLS):

```
./examples/mqttclient/mqttclient
MQTT Client: QoS 0
MQTT Net Init: Success (0)
MQTT Init: Success (0)
MQTT Socket Connect: Success (0)
MQTT Connect: Success (0)
MQTT Connect Ack: Return Code 0, Session Present 0
MQTT Subscribe: Success (0)
  Topic wolfMQTT/example/testTopic, Qos 0, Return Code 0
MQTT Publish: Topic wolfMQTT/example/testTopic, Success (0)
MQTT Waiting for message...
MQTT Message: Topic wolfMQTT/example/testTopic, Qos 0, Len 4
Payload (0 - 4): test
MQTT Message: Done
asdf
MQTT Publish: Topic wolfMQTT/example/testTopic, Success (0)
MQTT Message: Topic wolfMQTT/example/testTopic, Qos 0, Len 1
Payload (0 - 1): asdf
MQTT Message: Done
^CReceived SIGINT
MQTT Unsubscribe: Success (0)
MQTT Disconnect: Success (0)
MQTT Socket Disconnect: Success (0)
```

Output (with TLS - peer has self signed cert)

```
./examples/mqttclient/mqttclient -t
MQTT Client: QoS 0
MQTT Net Init: Success (0)
MQTT Init: Success (0)
MQTT TLS Setup (1)
MQTT TLS Verify Callback:
  PreVerify 0, Error -188 (ASN no signer error to confirm failure)
  Subject's domain name is iot.eclipse.org
  Allowing cert anyways
MQTT Socket Connect: Success (0)
MQTT Connect: Success (0)
MQTT Connect Ack: Return Code 0, Session Present 0
MQTT Subscribe: Success (0)
  Topic wolfMQTT/example/testTopic, Qos 0, Return Code 0
MQTT Publish: Topic wolfMQTT/example/testTopic, Success (0)
MQTT Waiting for message...
MQTT Message: Topic wolfMQTT/example/testTopic, Qos 0, Len 4
Payload (0 - 4): test
MQTT Message: Done
asdf
MQTT Publish: Topic wolfMQTT/example/testTopic, Success (0)
MQTT Message: Topic wolfMQTT/example/testTopic, Qos 0, Len 1
Payload (0 - 1):  asdf
MQTT Message: Done
^CReceived SIGINT
MQTT Unsubscribe: Success (0)
MQTT Disconnect: Success (0)
MQTT Socket Disconnect: Success (0)
```

3.2 Firmware Update Example

The MQTT firmware update is located in /examples/firmware/. This example has two parts. The first is called “fwpush”, which publishes a signed firmware image. The second is called “fwclient”, which receives the firmware image and verifies the signature. This example publishes message on the topic “wolfMQTT/example/firmware”.

Usage

./examples/firmware/fwpush -?
fwpush:
-? Help, print this usage
-h <host> Host to connect to, default iot.eclipse.org
-p <num> Port to connect on, default: Normal 1883, TLS 8883
-t Enable TLS
-c <file> Use provided certificate file
-q <num> Qos Level 0-2, default 0
-s Disable clean session connect flag
-k <num> Keep alive seconds, default 60
-i <id> Client Id, default WolfMQTTFwPush
-l Enable LWT (Last Will and Testament)
-u <str> Username
-w <str> Password
-n <str> Topic name, default wolfMQTT/example/firmware
-r Set Retain flag on firmware publish message
-C <num> Command Timeout, default 30000ms
-T Test mode
-f <file> Use file for publish, default README.md

fwpush output:
./examples/firmware/fwpush -t -f README.md
MQTT Firmware Push Client: QoS 2
Firmware Message: Sig 74 bytes, Key 65 bytes, File 4271 bytes
MQTT Net Init: Success (0)
MQTT Init: Success (0)
MQTT TLS Setup (1)
MQTT TLS Verify Callback: PreVerify 0,
Error -188 (ASN no signer error to confirm failure)
Subject's domain name is iot.eclipse.org
Allowing cert anyways
MQTT Socket Connect: Success (0)
MQTT Connect: Success (0)
MQTT Connect Ack: Return Code 0, Session Present 0
MQTT Publish: Topic wolfMQTT/example/firmware, Success (0)
MQTT Disconnect: Success (0)
MQTT Socket Disconnect: Success (0)
MQTT Net DeInit: Success (0)

fwclient output:

```
./examples/firmware/fwclient -t -f README.md
MQTT Firmware Client: QoS 2
MQTT Net Init: Success (0)
MQTT Init: Success (0)
MQTT TLS Setup (1)
MQTT TLS Verify Callback: PreVerify 0,
  Error -188 (ASN no signer error to confirm failure)
  Subject's domain name is iot.eclipse.org
  Allowing cert anyways
MQTT Socket Connect: Success (0)
MQTT Connect: Success (0)
MQTT Connect Ack: Return Code 0, Session Present 0
MQTT Subscribe: Success (0)
  Topic wolfMQTT/example/firmware, Qos 2, Return Code 2
MQTT Waiting for message...
MQTT Firmware Message: Qos 2, Len 4415
Firmware Signature Verification: Pass (0)
Saved 4271 bytes to README.md

^CReceived SIGINT
MqttSocket_NetRead: Error 0
MQTT Message Wait: Error (Network) (-8)
MQTT Disconnect: Success (0)
MQTT Socket Disconnect: Success (0)
MQTT Net DeInit: Success (0)
```

3.3 Azure IoT Hub Example

We setup a wolfMQTT IoT Hub on the Azure server for testing. We added a device called `demoDevice`, which you can connect and publish to. The example demonstrates creation of a SasToken, which is used as the password for the MQTT connect packet. It also shows the topic names for publishing events and listening to `devicebound` messages. This example only works with `ENABLE_MQTT_TLS` set and the wolfSSL library present because it requires Base64 Encode/Decode and HMAC-SHA256. Note: The wolfSSL library must be built with `./configure --enable-base64encode` or `#define WOLFSSL_BASE64_ENCODE`. The `wc_GetTime` API was added in 3.9.1 and if not

present you'll need to implement your own version of this to get current UTC seconds or update your wolfSSL library.

Usage

./examples/azure/azureiothub -?

azureiothub:

-?	Help, print this usage
-h <host>	Host to connect to, default wolfMQTT.azure-devices.net
-p <num>	Port to connect on, default: Normal 1883, TLS 8883
-t	Enable TLS
-c <file>	Use provided certificate file
-q <num>	Qos Level 0-2, default 1
-s	Disable clean session connect flag
-k <num>	Keep alive seconds, default 60
-i <id>	Client Id, default demoDevice
-l	Enable LWT (Last Will and Testament)
-u <str>	Username
-w <str>	Password
-n <str>	Topic name, default devices/demoDevice/messages/devicebound/#
-r	Set Retain flag on publish message
-C <num>	Command Timeout, default 30000ms
-T	Test mode

azureiothub output:

```
./examples/azure/azureiothub
AzureIoTHub Client: QoS 1, Use TLS 1
MQTT Net Init: Success (0)
SharedAccessSignature
sr=wolfMQTT.azure-devices.net%2fdevices%2fdemoDevice&sig=iy8al9ZPBLLZdMT38
SIGy8Qx7k5jY%2f5nTpBo8Mw84PA%3d&se=1482274308
MQTT Init: Success (0)
MQTT TLS Setup (1)
MQTT TLS Verify Callback: PreVerify 0, Error -188 (ASN no signer error to confirm
failure)
  Subject's domain name is *.azure-devices.net
  Allowing cert anyways
MQTT Socket Connect: Success (0)
MQTT Connect: Success (0)
MQTT Connect Ack: Return Code 0, Session Present 0
```

```
MQTT Subscribe: Success (0)
  Topic devices/demoDevice/messages/devicebound/#, Qos 1, Return Code 1
MQTT Publish: Topic devices/demoDevice/messages/events/, Success (0)
MQTT Waiting for message...
^CReceived SIGINT
MQTT Exiting...
MQTT Disconnect: Success (0)
MQTT Socket Disconnect: Success (0)
```

3.4 AWS IoT Example

We setup an AWS IoT endpoint and testing device certificate for testing. The AWS server uses TLS client certificate for authentication. The example is located in /examples/aws/. The example subscribes to `$aws/things/"AWSIOT_DEVICE_ID"/shadow/update/delta` and publishes to `$aws/things/"AWSIOT_DEVICE_ID"/shadow/update`. The AWS IoT broker requires TLS and only supports QoS levels 0-1.

Usage

```
./examples/aws/awsiot -?
awsiot:
-?          Help, print this usage
-h <host>   Host to connect to, default a2dujmi05ideo2.iot.us-west-2.amazonaws.com
-p <num>    Port to connect on, default: Normal 1883, TLS 8883
-t          Enable TLS
-c <file>   Use provided certificate file
-q <num>    Qos Level 0-2, default 1
-s          Disable clean session connect flag
-k <num>    Keep alive seconds, default 60
-i <id>     Client Id, default demoDevice
-l          Enable LWT (Last Will and Testament)
-u <str>    Username
-w <str>    Password
-n <str>    Topic name, default $aws/things/demoDevice/shadow/update/delta
-r          Set Retain flag on publish message
-C <num>    Command Timeout, default 30000ms
```

-T Test mode

awsiot output:

```
./examples/aws/awsiot
AwsIoT Client: QoS 1, Use TLS 1
MQTT Net Init: Success (0)
MQTT Init: Success (0)
MQTT TLS Setup (1)
MQTT Socket Connect: Success (0)
MQTT Connect: Success (0)
MQTT Connect Ack: Return Code 0, Session Present 0
MQTT Subscribe: Success (0)
  Topic $aws/things/demoDevice/shadow/update/delta, Qos 1, Return Code 1
MQTT Publish: Topic $aws/things/demoDevice/shadow/update, Success (0)
MQTT Waiting for message...

^CReceived SIGINT
MQTT Exiting...
MQTT Disconnect: Success (0)
MQTT Socket Disconnect: Success (0)
```

Chapter 4: Library Design

Library header files are located in the /wolfmqtt directory. Only the /wolfmqtt/mqtt_client.h header is required to be included:

```
#include <wolfmqtt/mqtt_client.h>
```

The library has three components:

1. mqtt_client
This is where the top level application interfaces for the MQTT client reside. If the API performs a network write it will block on a network read if an acknowledgment is expected.

2. mqtt_packet
This is where all the packet encoding/decoding is handled. This contains the MQTT Packet structures for:

- Connect: `MqttConnect`
- Publish / Message: `MqttPublish` / `MqttMessage` (they are the same)
- Subscribe: `MqttSubscribe`
- Unsubscribe: `MqttUnsubscribe`

3. mqtt_socket
This is where the transport socket optionally wraps TLS and uses the `MqttNet` callbacks for the platform specific network handling. This contains the MQTT Network structure `MqttNet` for network callback and context.

4.1 Example Design

The examples use a common “examples/mqttnet.c” to handle the network callbacks on the clients. This reference supports Linux (BSD sockets), FreeRTOS/LWIP, MQX RTCS, Harmony and Windows.

Chapter 5: API Reference

This describes the public application interfaces for the wolfMQTT library.

4.1 MqttPacketResponseCodes (enum)

These are the API response codes:
MQTT_CODE_SUCCESS = 0: Success
MQTT_CODE_ERROR_BAD_ARG = -1: Invalid argument provided
MQTT_CODE_ERROR_OUT_OF_BUFFER = -2: Rx or Tx buffer out of space
MQTT_CODE_ERROR_MALFORMED_DATA = -3: Malformed packet remaining length
MQTT_CODE_ERROR_PACKET_TYPE = -4: Invalid packet type in header
MQTT_CODE_ERROR_PACKET_ID = -5: Packet Id mismatch
MQTT_CODE_ERROR_TLS_CONNECT = -6: TLS connect error.
MQTT_CODE_ERROR_TIMEOUT = -7: Net read/write/connect timeout
MQTT_CODE_ERROR_NETWORK = -8: Network error
MQTT_CODE_ERROR_MEMORY = -9: Memory error
MQTT_CODE_ERROR_STAT = -10: State error
MQTT_CODE_CONTINUE = -101: Non-blocking mode, perform IO and call again.

4.2 MqttClient_Init

Synopsis:
```
#include <wolfmqtt/mqtt_client.h>

typedef int (*MqttMsgCb)(struct _MqttClient *client, MqttMessage *message, byte
is_new, byte is_done);

int MqttClient_Init(
        MqttClient *client,
        MqttNet *net,
        MqttMsgCb cb,
        byte *tx_buf, int tx_buf_len,
```

byte *rx_buf, int rx_buf_len,
int cmd_timeout_ms);

Description:

Initializes the wolfMQTT library for use. Must be called once per application and before any other calls to the library.

Return Values:

See MqttPacketResponseCodes in /wolfmqtt/mqtt_types.h
MQTT_CODE_SUCCESS - Success
MQTT_CODE_ERROR_BAD_ARG - Invalid argument provided

Parameters:

client - Pointer to MqttClient structure (okay if not initialized).
net - Pointer to MqttNet structure populated with network callbacks and context.
cb - Pointer to MqttMsgCb callback function.
tx_buf - Pointer to transmit buffer used during encoding.
tx_buf_len - Maximum length of the transmit buffer.
rx_buf - Pointer to receive buffer used during decoding.
rx_buf_len - Maximum length of the receive buffer.
connect_timeout_ms - Maximum command wait timeout in milliseconds.

Example:

```
#define MAX_BUFFER_SIZE         1024
#define DEFAULT_CMD_TIMEOUT_MS  1000

static int mqttclient_message_cb(MqttClient *client, MqttMessage *msg, byte
msg_new, byte msg_done)
{
    if (msg_new) {
        /* Message new */
    }
    if (msg_done) {
        /* Message done */
    }

    return MQTT_CODE_SUCCESS;
    /* Return negative to terminate publish processing */
}

int rc = 0;
MqttClient client;
```

```
MqttNet net;
byte *tx_buf = NULL, *rx_buf = NULL;

tx_buf = malloc(MAX_BUFFER_SIZE);
rx_buf = malloc(MAX_BUFFER_SIZE);
rc = MqttClient_Init(&client, &net, mqttclient_message_cb,
    tx_buf, MAX_BUFFER_SIZE, rx_buf, MAX_BUFFER_SIZE,
    DEFAULT_CMD_TIMEOUT_MS);
if (rc != MQTT_CODE_SUCCESS) {
    printf("MQTT Init: %s (%d)\n", MqttClient_ReturnCodeToString(rc), rc);
}
```

See Also:
None

4.3 MqttClient_Connect

Synopsis:
#include <wolfmqtt/mqtt_client.h>

int MqttClient_Connect(
MqttClient *client,
MqttConnect *connect);

Description:
Encodes and sends the MQTT Connect packet and waits for the Connect Acknowledgement packet. This can block in MqttNet.read data unless non-blocking is enabled (WOLFMQTT_NONBLOCK) and MQTT_CODE_CONTINUE is returned.

Return Values:
See MqttPacketResponseCodes in /wolfmqtt/mqtt_types.h
MQTT_CODE_SUCCESS - Success
MQTT_CODE_CONTINUE - Non-blocking mode, perform IO and call again.

Parameters:
client - Pointer to MqttClient structure already initialized using MqttClient_Init.
connect - Pointer to MqttConnect structure populated with connection options.

Example:

```
int rc = 0;
MqttClient client;
MqttConnect connect;
MqttMessage lwt_msg;

/* Define connect parameters */
connect.keep_alive_sec = keep_alive_sec;
connect.clean_session = clean_session;
connect.client_id = client_id;

/* Last will and testament sent by broker to subscribers of topic when broker
connection is lost */
memset(&lwt_msg, 0, sizeof(lwt_msg));
connect.lwt_msg = &lwt_msg;
connect.enable_lwt = enable_lwt;
if (enable_lwt) {
      lwt_msg.qos = qos;
      lwt_msg.retain = 0;
      lwt_msg.topic_name = "lwttopic";
      lwt_msg.message = (byte*)DEFAULT_CLIENT_ID;
      lwt_msg.message_len = strlen(DEFAULT_CLIENT_ID);
}
/* Optional authentication */
connect.username = username;
connect.password = password;

/* Send Connect and wait for Connect Ack */
rc = MqttClient_Connect(&client, &connect);
if (rc != MQTT_CODE_SUCCESS) {
    printf("MQTT Connect: %s (%d)\n", MqttClient_ReturnCodeToString(rc), rc);
}
```

See Also:
MqttClient_Init
MqttClient_Disconnect

4.4 MqttClient_Publish

Synopsis:
#include <wolfmqtt/mqtt_client.h>

```
int MqttClient_Publish(
        MqttClient *client,
        MqttPublish *publish);
```

Description:

Encodes and sends the MQTT Publish packet and waits for the Publish response (if QoS > 0). This can block in MqttNet.read data unless non-blocking is enabled (WOLFMQTT_NONBLOCK) and MQTT_CODE_CONTINUE is returned. If QoS level = 1 then will wait for PUBLISH_ACK. If QoS level = 2 then will wait for PUBLISH_REC then send PUBLISH_REL and wait for PUBLISH_COMP.

Return Values:

See enum MqttPacketResponseCodes in /wolfmqtt/mqtt_types.h
MQTT_CODE_SUCCESS - success
MQTT_CODE_CONTINUE - Non-blocking mode, perform IO and call again.

Parameters:

client - Pointer to MqttClient structure already initialized using MqttClient_Init.
publish - Pointer to MqttPublish structure initialized with message data. Note: MqttPublish and MqttMessage are same structure.

Example:

```
#define TEST_MESSAGE            "test" /* NULL */

int rc = 0;
MqttPublish publish;
word16 packet_id = 0;

/* Publish Topic */
publish.retain = 0;
publish.qos = qos;
publish.duplicate = 0;
publish.topic_name = "pubtopic";
publish.packet_id = ++packet_id;
publish.message = (byte*)TEST_MESSAGE;
publish.message_len = strlen(TEST_MESSAGE);
rc = MqttClient_Publish(&client, &publish);
if (rc != MQTT_CODE_SUCCESS) {
    printf("MQTT Publish: %s (%d)\n", MqttClient_ReturnCodeToString(rc), rc);
}
```

See Also:
MqttClient_Init
MqttClient_Subscribe

4.5 MqttClient_Subscribe

Synopsis:
#include <wolfmqtt/mqtt_client.h>

int MqttClient_Subscribe(
    MqttClient *client,
    MqttSubscribe *subscribe);

Description:
Encodes and sends the MQTT Subscribe packet and waits for the Subscribe Acknowledgement packet. This can block in MqttNet.read data unless non-blocking is enabled (WOLFMQTT_NONBLOCK) and MQTT_CODE_CONTINUE is returned.

Return Values:
See enum MqttPacketResponseCodes in /wolfmqtt/mqtt_types.h
MQTT_CODE_SUCCESS - Success
MQTT_CODE_CONTINUE - Non-blocking mode, perform IO and call again.

Parameters:
client - Pointer to MqttClient structure already initialized using MqttClient_Init.
subscribe - Pointer to MqttSubscribe structure initialized with subscription topic list and desired QoS.

Example:

```
#define TEST_TOPIC_COUNT        2

int rc = 0;
MqttSubscribe subscribe;
MqttTopic topics[TEST_TOPIC_COUNT], *topic;
word16 packet_id = 0;
```

```
/* Build list of topics */
topics[0].topic_filter = "subtopic1";
topics[0].qos = qos;
topics[1].topic_filter = "subtopic2";
topics[1].qos = qos;

/* Subscribe Topic */
subscribe.packet_id = ++packet_id;
subscribe.topic_count = TEST_TOPIC_COUNT;
subscribe.topics = topics;
rc = MqttClient_Subscribe(&client, &subscribe);

if (rc == MQTT_CODE_SUCCESS) {
      for (i = 0; i < subscribe.topic_count; i++) {
            topic = &subscribe.topics[i];
            printf("  Topic %s, Qos %u, Return Code %u\n",
                  topic->topic_filter, topic->qos, topic->return_code);
      }
}
else {
    printf("MQTT Subscribe: %s (%d)\n", MqttClient_ReturnCodeToString(rc), rc);
}
```

See Also:
MqttClient_Init
MqttClient_Unsubscribe

4.6 MqttClient_Unsubscribe

Synopsis:
#include <wolfmqtt/mqtt_client.h>

int MqttClient_Unsubscribe(
        MqttClient *client,
        MqttUnsubscribe *unsubscribe);

Description:
Encodes and sends the MQTT Unsubscribe packet and waits for the Unsubscribe Acknowledgement packet. This can block in MqttNet.read data unless non-blocking is enabled (WOLFMQTT_NONBLOCK) and MQTT_CODE_CONTINUE is returned.

Return Values:

See enum MqttPacketResponseCodes in /wolfmqtt/mqtt_types.h

MQTT_CODE_SUCCESS - Success

MQTT_CODE_CONTINUE - Non-blocking mode, perform IO and call again.

Parameters:

client - Pointer to MqttClient structure already initialized using MqttClient_Init.

unsubscribe - Pointer to MqttUnsubscribe structure initialized with topic list.

Example:

```
#define TEST_TOPIC_COUNT        2

int rc = 0;
MqttUnsubscribe unsubscribe;
MqttTopic topics[TEST_TOPIC_COUNT], *topic;
word16 packet_id = 0;

/* Build list of topics */
topics[0].topic_filter = "subtopic1";
topics[1].topic_filter = "subtopic2";

/* Unsubscribe Topics */
unsubscribe.packet_id = ++packet_id;
unsubscribe.topic_count = TEST_TOPIC_COUNT;
unsubscribe.topics = topics;
rc = MqttClient_Unsubscribe(&client, &unsubscribe);
if (rc != MQTT_CODE_SUCCESS) {
    printf("MQTT Unsubscribe: %s (%d)\n", MqttClient_ReturnCodeToString(rc),
rc);
}
```

See Also:

MqttClient_Init

MqttClient_Subscribe

4.7 MqttClient_Ping

Synopsis:

#include <wolfmqtt/mqtt_client.h>

int MqttClient_Ping(
        MqttClient *client);

Description:
Encodes and sends the MQTT Ping Request packet and waits for the Ping Response packet. This can block in MqttNet.read data unless non-blocking is enabled (WOLFMQTT_NONBLOCK) and MQTT_CODE_CONTINUE is returned.

Return Values:
See enum MqttPacketResponseCodes in /wolfmqtt/mqtt_types.h
MQTT_CODE_SUCCESS - Success
MQTT_CODE_CONTINUE - Non-blocking mode, perform IO and call again.

Parameters:
client - Pointer to MqttClient structure already initialized using MqttClient_Init.

Example:

```
/* Send Ping */
int rc = MqttClient_Ping(&client);
if (rc != MQTT_CODE_SUCCESS) {
    printf("MQTT Ping: %s (%d)\n", MqttClient_ReturnCodeToString(rc), rc);
}
```

See Also:
MqttClient_Init

4.8 MqttClient_Disconnect

Synopsis:
#include <wolfmqtt/mqtt_client.h>

int MqttClient_Disconnect(
        MqttClient *client);

Description:

Encodes and sends the MQTT Disconnect packet (no response).

Return Values:
See enum MqttPacketResponseCodes in /wolfmqtt/mqtt_types.h
MQTT_CODE_SUCCESS - Success
MQTT_CODE_CONTINUE - Non-blocking mode, perform IO and call again.

Parameters:
client - Pointer to MqttClient structure already initialized using MqttClient_Init.

Example:

```
int rc = MqttClient_Disconnect(&client);
if (rc != MQTT_CODE_SUCCESS) {
    printf("MQTT Disconnect: %s (%d)\n", MqttClient_ReturnCodeToString(rc),
      rc);
}
```

See Also:
MqttClient_Init
MqttClient_Connect

4.9 MqttClient_WaitMessage

Synopsis:
#include <wolfmqtt/mqtt_client.h>

int MqttClient_WaitMessage(
        MqttClient *client,
        MqttMessage *message,
        int timeout_ms);

Description:
Waits for Publish packets to arrive. This can block in MqttNet.read data unless non-blocking is enabled (WOLFMQTT_NONBLOCK) and MQTT_CODE_CONTINUE is returned. If a timeout_ms is provided it will be passed up to MqttNet.read which can be used for network select() with timeout or if non-blocking is enabled can return

MQTT_CODE_CONTINUE.

Return Values:

See enum MqttPacketResponseCodes in /wolfmqtt/mqtt_types.h
MQTT_CODE_SUCCESS - Success
MQTT_CODE_CONTINUE - Non-blocking mode, perform IO and call again.

Parameters:

client - Pointer to MqttClient structure already initialized using MqttClient_Init.
message - Pointer to MqttMessage structure (uninitialized is okay).
timeout_ms - Milliseconds until read timeout.

Example:

```
#define DEFAULT_CMD_TIMEOUT_MS  1000

int rc = 0;
MqttMessage msg;

/* Read Loop */
while (mStopRead == 0) {
      /* Try and read packet */
      rc = MqttClient_WaitMessage(&client, &msg, DEFAULT_CMD_TIMEOUT_MS);
      if (rc >= 0) {
            /* Print incoming message */
            printf("MQTT Message: Topic %s, Len %u\n", msg.topic_name,
            msg.message_len);
      }
      else if (rc != MQTT_CODE_ERROR_TIMEOUT) {
            /* There was an error */
            printf("MQTT Message Wait: %s (%d)\n",
      MqttClient_ReturnCodeToString(rc), rc);
            break;
      }
}
```

See Also:

MqttClient_Init
MqttClient_Publish

4.10 MqttClient_NetConnect

Synopsis:

```
#include <wolfmqtt/mqtt_client.h>

typedef int (*MqttTlsCb)(struct _MqttClient* client);

int MqttClient_NetConnect(
        MqttClient *client,
        const char *host,
        word16 port,
        int timeout_ms,
        int use_tls,
        MqttTlsCb cb);
```

Description:

Performs network connect with TLS (if use_tls is non-zero value). Will perform the MqttTlsCb callback if use_tls is non-zero value.

Return Values:

See enum MqttPacketResponseCodes in /wolfmqtt/mqtt_types.h
MQTT_CODE_SUCCESS - Success
MQTT_CODE_CONTINUE - Non-blocking mode, perform IO and call again.

Parameters:

client - Pointer to MqttClient structure already initialized using MqttClient_Init.
host - Address of the broker server
port - Optional custom port. If zero will use defaults (1883=normal, 8883=TLS)
use_tls - If non-zero value will connect with and use TLS for encryption of data.
cb - A function callback for configuration of the SSL context certificate checking.

Example:

```
#define DEFAULT_CON_TIMEOUT_MS  5000
#define DEFAULT_MQTT_HOST       "iot.eclipse.org"

word16 port = 0;
const char* host = DEFAULT_MQTT_HOST;

static int mqttclient_tls_cb(MqttClient* client)
{
```

```
    (void)client; /* Supress un-used argument */
    return SSL_SUCCESS;
}

/* Connect to broker */
int rc = MqttClient_NetConnect(&client, host, port, DEFAULT_CON_TIMEOUT_MS,
      use_tls, mqttclient_tls_cb);
if (rc != MQTT_CODE_SUCCESS) {
    printf("MQTT Net Connect: %s (%d)\n", MqttClient_ReturnCodeToString(rc),
      rc);
}
```

See Also:
MqttClient_NetDisconnect

4.11 MqttClient_NetDisconnect

Synopsis:
#include <wolfmqtt/mqtt_client.h>

int MqttClient_NetDisconnect(
        MqttClient *client);

Description:
Performs a network disconnect.

Return Values:
See enum MqttPacketResponseCodes in /wolfmqtt/mqtt_types.h
MQTT_CODE_SUCCESS - Success
MQTT_CODE_CONTINUE - Non-blocking mode, perform IO and call again.

Parameters:
client - Pointer to MqttClient structure already initialized using MqttClient_Init.

Example:

```
int rc = MqttClient_NetDisconnect(&client);
if (rc != MQTT_CODE_SUCCESS) {
    printf("MQTT Net Disconnect: %s (%d)\n", MqttClient_ReturnCodeToString(rc),
```

```
        rc);
}
```

See Also:
MqttClient_NetConnect

4.12 MqttClient_ReturnCodeToString

Synopsis:
#include <wolfmqtt/mqtt_client.h>

const char* MqttClient_ReturnCodeToString(
        int return_code);

Description:
Performs lookup of a wolfMQTT API return value.

Return Values:
String representation of the return code.

Parameters:
return_code - The return value from an API function.

Example:

```
printf("Return: %s (%d)\n", MqttClient_ReturnCodeToString(rc), rc);
```

See Also:
None