Embedding it better...

µTasker Document

µTasker – MQTT/MQTTS

Table of Contents

1. Introduction

This document discusses the MQTT (Message Queuing Telemetry Transport) implementation in the µTasker project.

MQTT is an ISO standard (ISO/IEC PRF 20922) publish-subscribe based protocol that operates over TCP/IP whereby multiple clients can connect to a server (known as the broker) on the TCP port 1883 [or securely on port 8883] and subscribe to topics, publish their own messages and receive topic messages published from other clients.

MQTT is said to have been designed for a small code footprint and the µTasker implementation requires around 4k code and 1k of RAM for the protocol layer and a simple client interface.

The µTasker project also includes secure MQTT (over TLS), which is referenced to as MQTTS. Since the underlying MQTT operations are essentially identical over TCP/IP or secure socket layer MQTTS details are only mentioned when there is something of relevance to it – *otherwise all MQTT discussions can be considered to be valid for both connection types*.

The document is very practical of nature in that it illustrates how to use the features on MQTT through concrete operations and observations and then introduces and discusses interesting or useful points as they come up in each case. *For a more formal review of the MQTT protocol its specification and various standard texts can be used*.

2. Enabling MQTT

MQTT can be enabled with the define

```
USE_MQTT_CLIENT
```

in the project configuration file `config.h` as long as TCP/IP is also enabled in the project (*operating over Ethernet, USB RNDIS or other TCP/IP enabled connections*).

When enabled it adds a command line interface that can be used to immediately establish connections to a broker and verify various actions, such as subscribing to topics, publishing messages and viewing topic messages from other devices.

Secure MQTTS support is optionally enabled with the additional define

```
SECURE_MQTT
```

and client side authentication support optionally added using the define

```
SUPPORT_CLIENT_SIDE_CERTIFICATE
```

The following shows the MQTT command line menu (*available on the UART debug interface, USB-CDC or Telnet*):

```
FTP/TELNET/MQTT
===================
up              go to main menu
mqtts_con       Secure con. to MQTT broker [ip]
mqtt_con        Connect to MQTT broker [ip]
mqtt_sub        Subscribe [topic] <QoS>
mqtt_top        List sub. topics
mqtt_un         Unsubscribe [ref]
mqtt_pub        Publish <"topic"/ref> <QoS>
mqtt_pub_l      Publish (long) <"topic"/ref> <QoS>
mqtt_dis        Disconnect from MQTT broker
help            Display menu specific help
quit            Leave command mode
```

whereby `mqtts_con` is only available when configured for secure operation.

Using these commands all MQTT operations can be tested and the command line interface code also serves as reference for program control of application layer use of the protocol and its capabilities.

The following optional configuration parameters can be used to override basic settings if needed, whereby their specific use is described later in the document:

```
#define MQTT_KEEPALIVE_TIME_SECONDS    300 // the keep alive time announced to the
broker when the connection is established
#define MQTT_MAX_SUBSCRIPTIONS 8 // MQTT module manages up to this many subscriptions
#define MQTT_MAX_TOPIC_LENGTH  16 // MQTT module stores individual topic strings up to
this length
```

3. MQTT Operation and Interface

This chapter discusses the command line interface operation in order to test various aspects of MQTT operation. In each case the application interface is also explained so that the same operations can be easily achieved from general application code.

The Mosquitto test broker at `test.mosquitto.org` was used for these tests, whereby its fixed IPv4 IP address is used for connection. *DNS can be used to first used to discover the IP address of any other brokers to be tested with*.

3.1 Connecting to/Disconnecting from the Broker

```
#mqtt_con 37.187.106.16
MQTT client Connecting...
#MQTT connected

mqtt_dis
Disconnecting...
#MQTT closed
```

The connection and disconnection parts are quite standard TCP connections and closes (in case of a secure connection the connection will involve the TLS handshake but, apart from additional delays, the user interface will see the same general behaviour).

Once the TCP connection itself has been established the MQTT connection itself is established by the client sending the MQTT Connect command, which the broker will respond to with a Connect Ack (which causes the “MQTT connected” message).

The MQTT connection is disconnected by using a simple TCP close sequence (which causes the “MQTT close” message).

The MQTT connection between the initial connection and its termination is a persistent TCP connection and so could remain in the established state forever. If we however look at the Wireshark recording of a reference connection that remains active for 5 minutes it is seen that there are regular MQTT Ping messages being sent by the client, which are used to confirm to the broker that the connection is still active, whereby the broker could close the connection is there is no activity for a certain period of time.

No.	Time	Source	Destination	Protocol	Length	Info
47	19.669174	192.168.0.5	37.187.106.16	TCP	58	49165 → 1883 [SYN] Seq=0 Win=1460 Len=0 MSS=1460
48	19.696154	37.187.106.16	192.168.0.5	TCP	60	1883 → 49165 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1420
49	19.696203	192.168.0.5	37.187.106.16	TCP	54	49165 → 1883 [ACK] Seq=1 Ack=1 Win=1460 Len=0
50	19.696358	192.168.0.5	37.187.106.16	MQTT	75	Connect Command
51	19.723957	37.187.106.16	192.168.0.5	TCP	60	1883 → 49165 [ACK] Seq=1 Ack=22 Win=29200 Len=0
52	19.725038	37.187.106.16	192.168.0.5	MQTT	60	Connect Ack
53	19.726422	192.168.0.5	37.187.106.16	TCP	54	49165 → 1883 [ACK] Seq=22 Ack=5 Win=1460 Len=0
214	81.013499	192.168.0.5	37.187.106.16	MQTT	56	Ping Request
215	81.041088	37.187.106.16	192.168.0.5	MQTT	60	Ping Response
216	81.041570	192.168.0.5	37.187.106.16	TCP	54	49165 → 1883 [ACK] Seq=24 Ack=7 Win=1460 Len=0
374	142.572814	192.168.0.5	37.187.106.16	MQTT	56	Ping Request
375	142.600157	37.187.106.16	192.168.0.5	MQTT	60	Ping Response
376	142.600812	192.168.0.5	37.187.106.16	TCP	54	49165 → 1883 [ACK] Seq=26 Ack=9 Win=1460 Len=0
500	203.960210	192.168.0.5	37.187.106.16	MQTT	56	Ping Request
501	203.994367	37.187.106.16	192.168.0.5	MQTT	60	Ping Response
502	203.996310	192.168.0.5	37.187.106.16	TCP	54	49165 → 1883 [ACK] Seq=28 Ack=11 Win=1460 Len=0
610	265.866714	192.168.0.5	37.187.106.16	MQTT	56	Ping Request
611	265.897793	37.187.106.16	192.168.0.5	MQTT	60	Ping Response
612	265.899154	192.168.0.5	37.187.106.16	TCP	54	49165 → 1883 [ACK] Seq=30 Ack=13 Win=1460 Len=0
801	327.182466	192.168.0.5	37.187.106.16	MQTT	56	Ping Request
802	327.212404	37.187.106.16	192.168.0.5	MQTT	60	Ping Response
803	327.213374	192.168.0.5	37.187.106.16	TCP	54	49165 → 1883 [ACK] Seq=32 Ack=15 Win=1460 Len=0
867	358.369718	192.168.0.5	37.187.106.16	TCP	54	49165 → 1883 [FIN, ACK] Seq=32 Ack=15 Win=1460 Len=0
868	358.394777	37.187.106.16	192.168.0.5	TCP	60	1883 → 49165 [FIN, ACK] Seq=15 Ack=33 Win=29200 Len=0
869	358.395872	192.168.0.5	37.187.106.16	TCP	54	49165 → 1883 [ACK] Seq=33 Ack=16 Win=1460 Len=0

This first recording shows both TCP and MQTT frames and the following only MQTT frames by filtering out all non-MQTT messages – *in the following discussions the MQTT filtering is used unless there is a particular TCP layer interest*.

No.	Time	Source	Destination	Protocol	Length	Info
50	19.696358	192.168.0.5	37.187.106.16	MQTT	75	Connect Command
52	19.725038	37.187.106.16	192.168.0.5	MQTT	60	Connect Ack
214	81.013499	192.168.0.5	37.187.106.16	MQTT	56	Ping Request
215	81.041088	37.187.106.16	192.168.0.5	MQTT	60	Ping Response
374	142.572814	192.168.0.5	37.187.106.16	MQTT	56	Ping Request
375	142.600157	37.187.106.16	192.168.0.5	MQTT	60	Ping Response
500	203.960210	192.168.0.5	37.187.106.16	MQTT	56	Ping Request
501	203.994367	37.187.106.16	192.168.0.5	MQTT	60	Ping Response
610	265.866714	192.168.0.5	37.187.106.16	MQTT	56	Ping Request
611	265.897793	37.187.106.16	192.168.0.5	MQTT	60	Ping Response
801	327.182466	192.168.0.5	37.187.106.16	MQTT	56	Ping Request
802	327.212404	37.187.106.16	192.168.0.5	MQTT	60	Ping Response

To get an initial idea of the MQTT protocol itself it is very useful to use the MQTT interpretation feature of Wireshark, whereby the following shows the content of the MQTT `Connect Command`:

```
▼ MQ Telemetry Transport Protocol, Connect Command
  ▶ Header Flags: 0x10 (Connect Command)
    Msg Len: 19
    Protocol Name Length: 4
    Protocol Name: MQTT
    Version: MQTT v3.1.1 (4)
  ▶ Connect Flags: 0x02
    Keep Alive: 300
    Client ID Length: 7
    Client ID: KINETIS
```

```
0000  54 67 51 be 0a 57 00 00  00 00 00 55 08 00 45 10   TgQ..W.. ...U..E.
0010  00 3d 00 0e 00 00 80 06  ea 24 c0 a8 00 05 25 bb   .=...... .$....%.
0020  6a 10 c0 0d 07 5b 10 aa  bd 29 db 4a e0 d2 50 18   j....[.. .).J..P.
0030  05 b4 10 67 00 00 10 13  00 04 4d 51 54 54 04 02   ...g.... ..MQTT..
0040  01 2c 00 07 4b 49 4e 45  54 49 53                  .,..KINE TIS
```

and the next the content of the broker's MQTT `Connect Ack`:

```
MQ Telemetry Transport Protocol, Connect Ack
  > Header Flags: 0x20 (Connect Ack)
    Msg Len: 2
  > Acknowledge Flags: 0x00
    Return Code: Connection Accepted (0)
```

```
0000  00 00 00 00 00 55 54 67  51 be 0a 57 08 00 45 00   .....UTg Q..W..E.
0010  00 2c a2 3b 40 00 36 06  52 18 25 bb 6a 10 c0 a8   .,.;@.6. R.%.j...
0020  00 05 07 5b c0 0d db 4a  e0 d2 10 aa bd 3e 50 18   ...[...J .....>P.
0030  72 10 7b ce 00 00 20 02  00 00 20 20               r.{... . ..
```

And in similar fashions, the `Ping Request` and `Ping Response`:

```
MQ Telemetry Transport Protocol, Ping Request
  > Header Flags: 0xc0 (Ping Request)
    Msg Len: 0
```

```
0000  54 67 51 be 0a 57 00 00  00 00 00 55 08 00 45 10   TgQ..W.. ...U..E.
0010  00 2a 00 10 00 00 80 06  ea 35 c0 a8 00 05 25 bb   .*...... .5....%.
0020  6a 10 c0 0d 07 5b 10 aa  bd 3e db 4a e0 d6 50 18   j....[.. .>.J..P.
0030  05 b4 48 2a 00 00 c0 00                           ..H*....
```

Ping Request

```
MQ Telemetry Transport Protocol, Ping Response
  > Header Flags: 0xd0 (Ping Response)
    Msg Len: 0
```

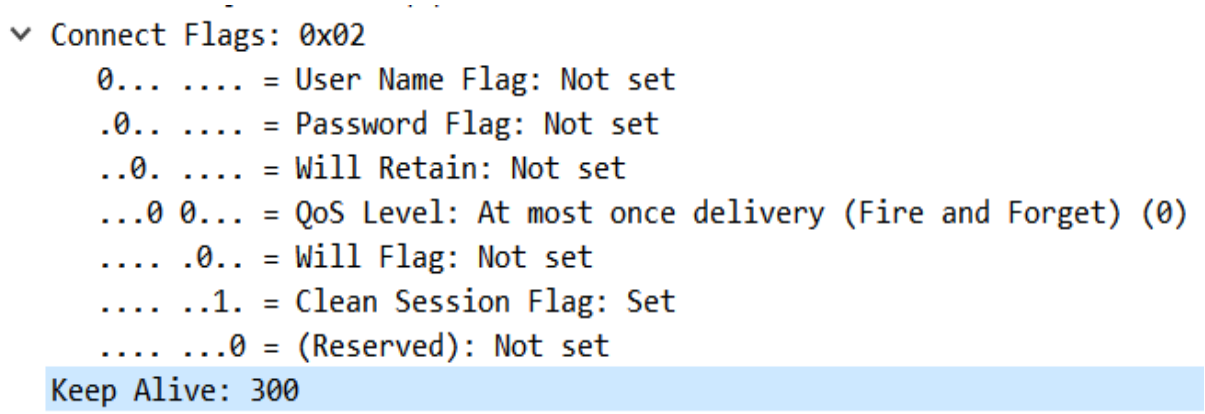
```
0000  00 00 00 00 00 55 54 67  51 be 0a 57 08 00 45 00   .....UTg Q..W..E.
0010  00 2a a2 3c 40 00 36 06  52 19 25 bb 6a 10 c0 a8   .*.<@.6. R.%.j...
0020  00 05 07 5b c0 0d db 4a  e0 d6 10 aa bd 40 50 18   ...[...J .....@P.
0030  72 10 cb cb 00 00 d0 00  20 20 20 20               r.....  ..
```

Ping Response

We notice that the client specifies a keep alive time of 300s*, and then sends pings (when no other activity) at a faster rate to ensure that the connection doesn't timeout when the device is still in good heath.
As well as allowing the broker to detect a device that has been powered down (not no longer contactable) this ping mechanism also enables the device to detect whether the broker is still available – if the broker doesn't respond to a ping (after the TCP layer has attempted delivery a number of times if needed) the broker can be declared as being no longer on line.

*See optional parameter `MQTT_KEEPALIVE_TIME_SECONDS`

In the initial connection message there is a field called “Connect Flags” that needs to be mentioned since it defines some basic, but important, properties of the MQTT connection being established. This is how it looks in this example case:



In this case the Clean Session Flag is set as standard since it ensures that the device's messages at the broker are deleted when the connection terminates.

The QoS (quality of service) level is set to 0, which means that the MQTT will not guarantee delivery but only ensure that a single attempt is made. This is the basic QoS used by the connection but doesn't restrict higher levels being used for individual messages during the connection session.

Finally the application interface can be shown that allows connection. The code source is available in the file `debug.c`, which controls this MQTT menu operation, whereby the code shown here is sometimes a simplified version of it to focus on the relevant points.

```
unsigned long ulModeFlags = (UNSECURE_MQTT_CONNECTION |
                             MQTT_CONNECT_FLAG_CLEAN_SESSION);
unsigned char ucDestinationIP[] = {37, 187, 106, 16};
if (fnConnectMQTT(ucDestinationIP, fnMQTT_callback, ulModeFlags) == 0) {
    fnDebugMsg("Connecting...");
}
else {
    fnDebugMsg("Can't connect!");
}
```

Here is is seen that the connection flags such as `MQTT_CONNECT_FLAG_CLEAN_SESSION` are defined by the application when it connections.
In case the connection is to be performed over a secure socket `SECURE_MQTT_CONNECTION` can be used instead of `UNSECURE_MQTT_CONNECTION` (*which is the default if not explicitly specified*).

We note that a call-back is specified which is required to respond to various event during general MQTT session operation. The complete callback is reproduced here as reference throughout this document:

```
static unsigned short fnMQTT_callback(signed char scEvent, unsigned char *ptrData, unsigned long
ulLength, unsigned char ucSubscriptionRef)
{
    CHAR *ptrBuf = (CHAR *)ptrData;
    int iAddRef = 0;
    switch (scEvent) {
    case MQTT_CLIENT_IDENTIFIER:
        ptrBuf = uStrcpy(ptrBuf, temp_pars->temp_parameters.cDeviceIDName); // supply a string to be
used as MQTT device identifier - this should be unique and normally contain only characters 0..9, a..z,
A..Z (normally up to 23 bytes)
        break;
    case MQTT_WILL_TOPIC:            // optional connection string fields
    case MQTT_WILL_MESSAGE:
    case MQTT_USER_NAME:
    case MQTT_USER_PASSWORD:
        ptrBuf = uStrcpy(ptrBuf, "string");
        break;
    case MQTT_CONNACK_RECEIVED:
        fnDebugMsg("MQTT connected\r\n");
        break;
    case MQTT_SUBACK_RECEIVED:
        fnDebugMsg("MQTT subscribed");
        iAddRef = 1;
        break;
    case MQTT_UNSUBACK_RECEIVED:
        fnDebugMsg("MQTT subscribed");
        iAddRef = 1;
        break;
    case MQTT_PUBLISH_ACKNOWLEDGED:
        fnDebugMsg("MQTT published - QoS");
        fnDebugDec(ulLength, 0);
        iAddRef = 1;
        break;
    case MQTT_PUBLISH_TOPIC:                        // add a default publish topic
        ptrBuf = uStrcpy(ptrBuf, "xyz/abc");
        break;
    case MQTT_PUBLISH_DATA:
        {
            static unsigned char ucDataCnt = 0;
            int i = 0;
            ptrBuf = uStrcpy(ptrBuf, "abcd");       // add string content
            while (i++ < usPubLength) {             // plus some binary content
                *ptrBuf++ = ucDataCnt++;
            }
        }
        break;
    case MQTT_HOST_CLOSED:
    case MQTT_CONNECTION_CLOSED:
        fnDebugMsg("MQTT closed\r\n");
        break;
    case MQTT_TOPIC_MESSAGE:
        fnDebugMsg("Message (");
        fnDebugDec(ulLength, 0);
        fnDebugMsg(")");
        iAddRef = 1;
        break;
    }
    if (iAddRef != 0) {
        fnDebugMsg(" [");
        fnDebugDec(ucSubscriptionRef, 0);
        fnDebugMsg("]\r\n");
    }
    return (unsigned short)((unsigned char *)ptrBuf - ptrData);
}
```

MQTT application callback reference

It may have been noticed that in the connect message the device sent a string to indicate its Client ID (“KINETIS”), which is the first reference to the use of the application call back to define specific details. When the call-back receives the event MQTT_CLIENT_IDENTIFIER it can enter its own ID as it desires, which is then appropriately inserted into this message.

Beware that each device should have a unique client ID. Trying to connection a second device with the same Client ID as another one will tend to cause the broker to close the connection to the client with the same ID that was connected to it first.

3.2 Subscribing to/Unsubscribing from a Topic

MQTT is subscription oriented and so it is usual for a client to subscribe to one or more topics. Once subscribed this client will then receive any message published for this topic, whether it be published by another client or itself.

The client is also free to later unsubscribe form topics that no linger interest it, after which no more topic messages will be sent to it by the broker.

The following shows the command line interface being used to subscribe to two topics (“uTasker/test1” and “uTasker/test2”) and then unsubscribing from them.

Beware that topics are case sensitive!

We note that a QoS is assigned to each individual topic which means that the subscription process itself is performed with this level. QoS 2 and QoS 1 are used, which differ in the guarantee of delivery made. QoS 2 means that it is to be guaranteed to be delivery to the broken a single time whereas QoS 1 is guaranteed to be delivered “at least once” but not excluding that it may be delivered a multiple number of times.

See the chapter on QoS for some examples of MQTT message exchanges with the three levels (0, 1 and 2).

```
#MQTT subscribed [1]
mqtt_sub uTasker/test1
Subscribing (ref=1)...
#MQTT subscribed [1]
mqtt_sub uTasker/test2 1
Subscribing (ref=2)...
#MQTT subscribed [2]
mqtt_un 2
Unsubscribing...
#MQTT subscribed [2]
mqtt_un 1
Unsubscribing...
#MQTT subscribed [1]
```

The application code responsible for subscribing to a topic may look as follows:

```
int iSubscriptionReference;
unsigned char ucQoS = 2;
iSubscriptionReference = fnSubscribeMQTT("TEST1", ucQoS);
if (iSubscriptionReference >= 0) {
    fnDebugMsg("Subscribing (ref=");
    fnDebugDec(iSubscriptionReference, 0);
    fnDebugMsg(")...");
}
else {
    fnDebugMsg("MQTT error!");
}
```

where it is seen that it defines the (default) QoS per topic, can specify directly the topic name and received a topic reference (for later use with respect to all messages received belonging to this topic or for publishing ones own messages related to it).

It is worth noting that the application doesn't need to pass the topic with the subscription call but can instead insert it via the call-back each time the event MQTT_PUBLISH_TOPIC is received (during the subscription process). In the second case a null pointer is passed instead and the MQTT module doesn't keep a local copy of the topic string and calls back the application each time it needs to know it.

The MQTT management interface associates the defined QoS with the subscription for later (default) use and can manage up to MQTT_MAX_SUBSCRIPTIONS* active subscriptions. It can save topic strings of up to MQTT_MAX_TOPIC_LENGTH* length each.

When unsubscribing, the subscription reference number is used, whereby the unsubscribe operation will be performed using the same QoS as used for its subscription.

```
if (fnUnsubscribeMQTT(2) >= 0) { // unsubscribe from topic ref. 2
    fnDebugMsg("Unsubscribing...");
}
else {
    fnDebugMsg("MQTT error!");
}
```

The command “mqtt_top” allows listing topics presently subscribed to, for example:

```
mqtt_top
Reference:1 QoS:2 Topic:uTasker/test1
Reference:2 QoS:1 Topic:uTasker/test2
Reference:3 QoS:2 Topic:uTasker/test3
```

*See optional parameters MQTT_MAX_SUBSCRIPTIONS and MQTT_MAX_TOPIC_LENGTH

3.2 Messages and Receiving Published Messages

The command line interface allows testing publishing predefined test message – either of short length or of a length almost filling up a standard TCP buffer (around 1k bytes). The message can be sent with a defined QoS and be sent to a specific topic. If sent to a topic that we are subscribed to we expect to receive the message back to ourselves from the broker too!

Examples of publishing messages are as follows:

```
mqtt_pub "uTasker/test1" 1
```

which publishes a *test message* to the topic `uTasker/test1` with QoS of 1. Note that the topic string needs to be enclosed in "..." in this command and a missing QoS value will cause QoS 2 to be used.

```
mqtt_pub_l "uTasker/test1" 0
```

would publish a *long* message to the same topic but using QoS 0 instead.

When the publish process completes the call back displays the fact along with details of the QoS and reference to a local subscription (0 mean that it is a one-off topic publication since we are not subscribed to the topic):

```
MQTT published - QoS2 [0]
```

The MQTT broker will pass on the topic message to any, and all, devices subscribed to it and a device receiving such a topic message will display the fact that it has as follows:

```
Message (14) [1]
```

The message length is 14 bytes in this case and the reception is on the subscription reference 1. This then allows the application callback to simply associate the message to a particular topic and then it can parse the content accordingly, whereby we note that the message content itself is not defined by the MQTT specification and can be either ASCII or binary. Depending on the overall application's communication between devices the content could be details about local measurements (values from sensors) or commands that one device sends to others.

A long test message would look like:

```
Message (1028) [3]
```

where it is seen that the message content is approximately 1k byte instead – in this case for topic reference 3.

If a device is subscribed to a topic that it sends a message to it can use its subscription reference instead as the topic string. For example

```
mqtt_pub 1 2
```

will cause a message to be published to the topic of subscription reference 1 with QoS 2.

If the QoS is not specified the value used will default to that in the subscription characteristics (the QoS defined when subscribing). *This behaviour is not specified by MQTT since every message has its own QoS level but this allows simply setting a QoS to be used as default for individual subscription topic messages.*

See the chapter on QoS for some examples of MQTT message exchanges with the three levels (0, 1 and 2).

The application interface for publishing messages may look like this:

```
unsigned char ucSubscriptionRef = 2; // send to subscription 2 topic
if (fnPublishMQTT(ucSubscriptionRef, 0, -1) == 0) {
    fnDebugMsg("Publishing...");
}
else {
    fnDebugMsg("MQTT error!");
}
```

In this example the message is sent to the topic that we are subscribed to at subscription reference 2. Not topic needs to be passed in this case and the QoS used is the subscription's default.

The MQTT application callback is used with the event number MQTT_PUBLISH_DATA (see reference earlier on in the document) for the application to insert the appropriate content. It is to be noted that a maximum of 1400 bytes may be inserted!

In order to sent a message to a non-subscribed topic the call may instead look like:

```
if (fnPublishMQTT(0, "uTasker/test5", 2) == 0) { // publish with QoS 2
    fnDebugMsg("Publishing...");
}
else {
    fnDebugMsg("MQTT error!");
}
```

or

```
if (fnPublishMQTT(0, 0, 1) == 0) { // publish with QoS 1
    fnDebugMsg("Publishing...");
}
else {
    fnDebugMsg("MQTT error!");
}
```

whereby the publish topic can be inserted by the MQTT module calls back with the event `MQTT_PUBLISH_TOPIC`.

Beware that publishing a message with an empty topic (length of zero) will tend to result in the broker closing the connection to the devices. The command line interface will report an error if this is attempted.

4. Quality of Service (QoS)

The chapters discussing subscribing and publishing have shown how to control the QoS or each exchange via the application interface. This section takes a brief look at the MQTT operation at the three levels based on publishing a message, whereby the principle is similar for subscribing and unsubscribing too. *By using Wireshark it is easy to monitor the operation and get familiar with the details involved*.

4.1 QoS 0 (Fire and Forget)

The lowest level of QoS is 0, whereby messages are sent without requiring them to be acknowledged. However this doesn't mean that the messages will often be lost due to the the fact that they are being transported by TCP/IP where the delivery is more or less ensured as long as the connection doesn't totally fail during the process.

This recording shows the MQTT protocol frame with the publish message whereby the TCP protocol's ACK is displayed since there is no further activity at the MQTT level:

```
2010.261663   192.168.0.5      37.187.106.16    MQTT    85 Publish Message
2010.344861   37.187.106.16    192.168.0.5      TCP     60 1883 → 49163 [ACK] Seq=92 Ack=155 Win=29200 Len=0
```

In this mode one sees that the MQTT message was delivered to the destination but there is no confirmation that it was actually processed by the MQTT level.

This level of quality is known as “Fire and Forget”.

The MQTT protocol level is shown as interpreted by Wireshark as follows:

```
MQ Telemetry Transport Protocol, Publish Message
  Header Flags: 0x32 (Publish Message)
      0011 .... = Message Type: Publish Message (3)
      .... 0... = DUP Flag: Not set
      .... .01. = QoS Level: At least once delivery (Acknowledged deliver) (1)
      .... ...0 = Retain: Not set
    Msg Len: 31
    Topic Length: 13
    Topic: uTasker/test1
    Message Identifier: 2
    Message: abcd\n\v\f\r\016\017\020\021\022\023
```

Here we see that the QoS is signaled in the header flags which precede the topic and message themselves.

4.2 QoS 1 (Acknowledged Delivery)

The next level of QoS is 1, whereby messages are acknowledged also by the MQTT layer.

This recording shows the MQTT protocol (without TCP layer):

2007.119301	37.187.106.16	192.168.0.6	MQTT	87 Publish Message
2007.120054	192.168.0.6	37.187.106.16	MQTT	60 Publish Ack

In this mode one sees that the MQTT message delivered is acknowledged by the receiving the MQTT level.

```
MQ Telemetry Transport Protocol, Publish Message
  Header Flags: 0x32 (Publish Message)
     0011 .... = Message Type: Publish Message (3)
     .... 0... = DUP Flag: Not set
     .... .01. = QoS Level: At least once delivery (Acknowledged deliver) (1)
     .... ...0 = Retain: Not set
  Msg Len: 31
  Topic Length: 13
  Topic: uTasker/test1
  Message Identifier: 27
  Message: abcd\n\v\f\r\016\017\020\021\022\023
```

The QoS 1 is seen in the header flags. In addition one notices that there is an additional message identification filed that is used to ensure that the acknowledge belongs to the published message; *the ack message contains the same identifier value*. This level guarantees delivery *at least once*, but not that it can't be delivered a multiple number of times.

4.3 QoS 2 (Assured Delivery)

The highest level of QoS is 2, whereby messages are acknowledged by the MQTT layer in a way that ensures that they are delievered only a single time..

This recording shows the MQTT protocol (without TCP layer):

2040.739129	192.168.0.5	37.187.106.16	MQTT	87 Publish Message
2040.769623	37.187.106.16	192.168.0.5	MQTT	60 Publish Received
2040.770682	192.168.0.5	37.187.106.16	MQTT	58 Publish Release
2040.798710	37.187.106.16	192.168.0.5	MQTT	60 Publish Complete

In this mode one sees that there is a 4-way handshake involved whereby the broker informs first that it has received the message, after which the client releases it, which the broker confirms with a complete.

The publish message is the same as the QoS 1 message, with the QoS flags in the header to to 2. The further messages are not particularly interesting apart from the fact that they all need to contain the same message identifier value, which ensures that they indeed belong to the correct one. Just the Publish Received message is shown to give an idea of its simplicity:

```
MQ Telemetry Transport Protocol, Publish Received
  Header Flags: 0x50 (Publish Received)
      0101 .... = Message Type: Publish Received (5)
      .... 0000 = Reserved: 0
  Msg Len: 2
  Message Identifier: 5
```

5. Conclusion

This document has given a brief introduction to MQTT (Message Queuing Telemetry Transport) and explained how to configure the µTasker project to efficiently start working with it.

The command line interface allows immediate testing of MQTT operations in combination with a MQTT broker, such as the on-line Mosquitto one which is foreseen for such purposes.

Subscribing and unsubscribing to topics and publishing and receiving topic messages has been demonstrated as well as how Quality of Service levels can be easily controlled.

As well as MQTT operation on the TCP port 1883 the µTasker project enables turn-key secure MQTTS operation on TCP port 8883 using its secure (socket) layer option.

Modifications:

V1.00 12.03.2018: First version