MQTT Guide

Cross-Domain Development Kit XDK110
Platform for Application Development

Bosch Connected Devices and Solutions

BOSCH
Invented for life

XDK110: Data Sheet

Document revision	2.0
Document release date	17.08.17
Workbench version	3.0.0
Document number	BCDS-XDK110-GUIDE-MQTT
Technical reference code(s)	
Notes	Data in this document is subject to change without notice. Product photos and pictures are for illustration purposes only and may differ from the real product's appearance. **Subject to change without notice**

XDK MQTT Guide

Platform for Application Development

MQTT is a messaging protocol designed for lightweight M2M (Machine-to-Machine) communications and IoT (Internet of Things) applications. Although the XDK does not support MQTT natively this guide provides a convenient way to learn about the basics of MQTT and introduce it into XDK projects.

Table of Contents

1. THE MQTT PROTOCOL 3
1.1 Communication Architecture 3
1.2 Data Structure 6

2. API SETUP 8

3. NETWORK SETUP 9

4. SETUP OF XDK AS MQTT CLIENT 10
4.1 Initialization 10
4.2 Connection to MQTT Server 11
4.3 Subscribing to Topics 13
4.4 Publishing Data 14

5. DOCUMENT HISTORY AND MODIFICATION 16

This guide postulates a basic understanding of the XDK and according Workspace. For new users we recommend going through the following guides at xdk.io/guides first:

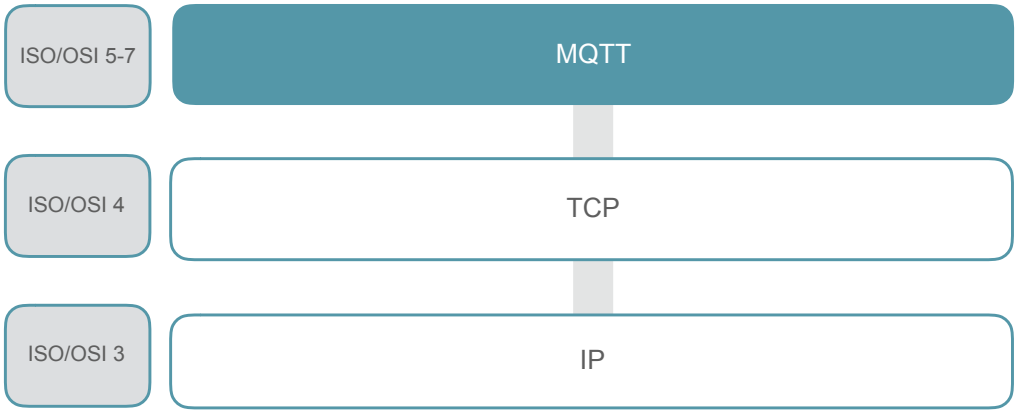
- *Workbench Installation*
- *Workbench First Steps*
- *XDK Guide FreeRTOS*
- *XDK Guide Wi-Fi*

1. The MQTT Protocol

Although MQTT is widely associated with the term Message Queuing Telemetry Transport it is not the correct meaning of the abbreviation: The name MQTT comes from an IBM product called MQseries and has nothing to do with traditional message queues. In general MQTT can be seen as a publish/subscribe messaging protocol based on TCP/IP using the client/server model.

As shown in Picture 1 the MQTT protocol is based on top of TCP/IP and all clients and the broker need to have a TCP/IP stack.

Picture 1. MQTT in the ISO/OSI layers



Initially developed by IBM to create a protocol for minimal battery loss and bandwidth, connecting oil pipelines via satellite connection, it is now an open standard. Gaining great acceptance in the IoT environment MQTT is persevering its initial goals like quality of service data delivery, continuos session awareness, simple implementation with a lightweight protocol stack especially useful on embedded devices with limited capacities.

This chapter will give an introduction into MQTT and especially its basic communication principles as well as data structures and methods that need to be understood in order to use the MQTT protocol in applications for lightweight and easy-to-use data exchange.

Readers who are already familiar with these topics or who are just looking for the XDK specific implementation details can safely skip to chapter 2.

Note: The MQTT standard is defined in the ISO/IEC PRF 20922.

1.1 Communication Architecture

The communication architecture is based on the client-server model where clients and servers are responsible for specific tasks.
In general the roles are defined as the following:
- server: awaits incoming connection requests by clients and provides resources or services
- client: initiates the communication and requests the server's content and functionalities

When talking about MQTT there are two kinds of client types:
- subscriber client: is interested in a special set of data from other publishers and registers as a potential recipient of data whenever it is published to the server
- publisher client: provides data that can be interesting to subscribers by pushing data to the server whenever the data is changed or needs to be updated

The clients, either subscriber, publisher or both, do not communicate directly with and do not know about each other but rather exchange data with the help of a common central server in the communication architecture, called the broker.
This principle is described as the clients being decoupled from each other. Decoupling happens on three stages:
- space decoupling: publisher and subscriber do not need to know each other (e.g. IP/port)
- time decoupling: publisher and subscriber do not need to run at the same time
- synchronization decoupling: operations on both sides are not halted during publishing / receiving

A MQTT client can be any device from a micro controller up to a full-blown server with a MQTT library running and is connecting to an MQTT broker over any kind of network. MQTT libraries are available for a huge variety of programming languages (e.g. C, C++, C#, Go, iOS, Java, JavaScript, .NET)

Each client that wants to talk to other clients needs to be connected to the same broker. The broker needs to receive and filter all incoming messages, check which clients apply as interested, connected and available recipients and distribute the updated message according to the subscriptions.
It is also an important task of the broker to provide the authentication and authorization of clients since the broker is exposed and accessible by many communication partners (e.g. via the internet). Whereas the MQTT client can be deployed on a great variety of devices with flexible performance and capabilities it is essential that the MQTT server can handle various client connections and data transmission between the publishers and subscribers.

As a premise for the communication architecture both clients, publisher and subscriber, initiate a connect message to the same broker which responds with an acknowledgement including a status code. Once the connection is established, the broker will keep it active as long as the client doesn't perform a disconnect or the connection is lost.

The connect request that is being sent by the clients contains mandatory elements to be seen in table 1.

Table 1. Client connect mandatory parameters

Parameter	Explanation
clientID	The client identifier (here: clientID) is a identifier of each MQTT client connecting to a MQTT broker. It needs to be unique for the broker to know the state of the client. For a stateless connection it is also possible to send an empty clientID together with the attribute of clean session to be true.
cleanSession	The clean session flag indicates to the broker whether the client wants to establish a clean session or a persistent session where all subscriptions and messages (QoS 1 & 2) are stored for the client. Note: QoS as part of message delivery will be explained in chapter 1.2.
keepAlive	The keep alive value is the time interval that the client commits to for when sending regular pings to the broker. The broker responds to the pings enabling both sides to determine if the other one is still alive and reachable

There are also optional elements that can be added to a connect request as listed in table 2.

Table 2. Client connect optional parameters

Parameter	Explanation
username/password	MQTT allows to send a username and password for authenticating the client. However the password is sent in plaintext if it isn't encrypted or hashed by implementation or TLS is used underneath.
lastWillTopic/ -QoS/ -Message/ -Retain	The last will message allows the broker to notify other clients when a client disconnects from the broker unexpectedly. Therefore the MQTT topic, QoS, message and retain flag can be sent to the broker during the client connect request.

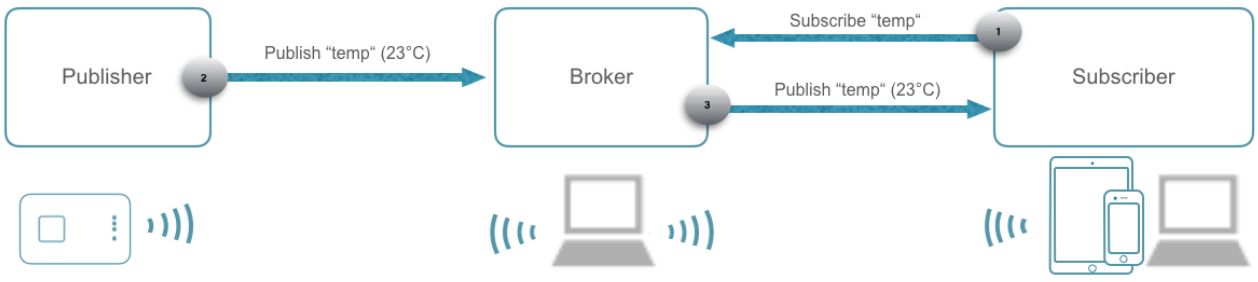
Upon receiving a client connect request the broker must respond with a connect acknowledgement containing the two elements described in table 3.

Table 3. Broker connect acknowledgement parameters

Parameter	Explanation
sessionPresent	The session present flag returns whether the broker has a persistent session of the client from previous connections. If a client previously connected to the broker with cleanSession set to false this indicates if it must subscribe to topics or if they are stored in its session.
returnCode	The return code signals if the connection attempt was successful or not (e.g. bad protocol version, identifier rejected, server unavailable, bad username/password or not authorized)

The minimal MQTT setup as an example of the communication architecture including the mandatory participants like the publisher, subscriber and broker can be seen in the Picture 2:

Picture 2. MQTT communication architecture



After both clients are successfully connected the following steps are performed on MQTT level:
1) The first step is done by the subscriber client by subscribing to a topic called "temp" that contains and represents the current value of the room temperature in this example. In general a topic can be seen as a data container that is identified by a unique name in the MQTT broker and will be explained in detail in chapter 1.2.
2) Secondly the publisher is ready to notify the broker about a change in the information it is providing to the MQTT ecosystem and publishes an update to the temperature value to the broker.
3) The third action is performed by the broker by forwarding the change in the temperature topic "temp" to each subscriber client that is interested in this topic and showed its interest by previously subscribing to the respective topic.

1.2 Data Structure

MQTT provides several mechanisms for data storage and exchange between clients. These tools and methods are ranging from providing data containers, also called topics, up until the usage of wild cards, quality of service agreement and adapting message options.

This chapter focuses on data structure in MQTT and shall provide an understanding of the ways of storing, sending and receiving data with options to match the application and use case environment in which MQTT is applied.

MQTT uses a topic-based filtering of the messages on the broker where each message must contain a topic which will be interpreted by the broker to forward the message to interested clients. Each message typically has a payload which contains the actual data to transmit in byte format. A publish message by the client consists of the following parameters seen in table 4.

Table 4. Client publish message parameters

Parameter	Explanation
packetID	The packet identifier (here: packetID) is a unique identifier set between client and broker to identify a message request/ack in a message flow (only relevant for QoS 1 & 2).
topicName	A simple string which is hierarchically structured with forward slashes „/“ as delimiters (e.g. “home/livingroom/temperature” or “Germany/Munich/Octoberfest/people”).
qos	A Quality of Service level (QoS) determines the guarantee of a message reaching the other client or the broker (e.g. 0 = fire and forget, 1 = at least once and 2 = exactly once)
retainFlag	The retain flag determines if the message will be saved by the broker for the specified topic as last known good value. New clients that subscribe to that topic will receive the last retained message on that topic instantly after subscribing.
payload	This is the actual content of the message. MQTT is data-agnostic and it totally depends on the use case how the payload is structured. It’s completely up to the sender if it wants to send binary data, textual data or even XML or JSON.
dupFlag	The duplicate flag is used on QoS levels 1 & 2 and indicates whether this message is a duplicate and is resent because the other end didn’t acknowledge the original message.

In order for a client to receive notifications about topics being updated by publish messages from other clients it needs to subscribe to topics it is interested in with a subscribe message to the broker. Subscribe messages contain the parameters visible in table 5.

Table 5. Client subscribe message parameters

Parameter	Explanation
packetID	The packet identifier (here: packetID) is a unique identifier set between client and broker to identify a message request/ack in a message flow (only relevant for QoS 1 & 2).
listTopics (qos1, topic1, qos2, topic2, qos3, topic3, …)	The subscribe message can contain a list subscriptions with a pair of topic and QoS level each. The topic in the subscribe message can also contain wildcards, which makes it possible to subscribe to certain topic patterns.

Topics are in general strings with an hierarchical structure, that allow filtering based on a limited number of expression. A topic consists of one or more topic levels which each level being separated by a forward slash „/“ like for example „*myserver/sensors/temperature*“.
Following characteristics apply to when choosing a topic name:
- at least one character
- short and simple yet understandable
- case-sensitivity

When a client subscribes to a topic it can use the exact topic name the message was published to or it can subscribe to more topics at once by using wildcards (only for subscribing and not for publishing). Wildcards can be used in the following ways:
- single level „+“: substitutes one topic level where any topic can match the level where the wildcard is placed (e.g. „*myserver/+/temperature*“ will match „*myserver/indoor/temperature*“ and „*myserver/outdoor/temperature*“)
- multi level „#“: covers an arbitrary number of topic levels as it is placed as the last character in a topic subscription (e.g. „myserver/sensors/#“ will match „*myserver/sensors/humidity*“ and „*myserver/sensors/gyro/x*“)

When choosing the QoS for the MQTT application it is very important to keep in mind that choosing higher QoS levels can have an impact on system performance. For instance when the MQTT communication partners are connected in a wired network with low data loss rates it can already work out with a QoS 0 fire-and-forget method. In wireless networks it might be beneficial to choose QoS 1 at-least-once methods to ensure data integrity where each message is definitely delivered but might be done more than once. This affects bandwidth, performance and the ability to cope with duplicates in the MQTT applications. QoS 2 increases the communication overhead to another level but is useful for when duplicates can not be handled individually on application level. If there are overlapping subscriptions for one client, the highest QoS level for that topic wins and will be used by the broker for delivering the message.

There is also the possibility to unsubscribe to topics if the client does not want to be updated about every change of the information represented in the given topic. The unsubscribe message consists of the packetID (QoS 1 & 2) and the topics to be unsubscribed which makes it rather simple and leads to not being described in detail here.

2. API Setup

Before working with MQTT there are several steps to take care of in terms of supporting MQTT functionality in the XDK in the form of APIs. For this the eclipse MQTT Paho Client library is used in this guide and throughout the implementation sections. To download this version, visit the website: http://git.eclipse.org/c/paho/org.eclipse.paho.mqtt.embedded-c.git/.

Under the „summary“ tab, look for the „Tag“ section and select one of the files under download (as of this release the latest file is *org.eclipse.paho.mqtt.embedded-c-1.0.0*). Save this file on the computer with the XDK Workbench. In a file explorer window, navigate to the folder where the Paho project is stored and extract the files. In the Paho project folder, navigate to the MQTTPacket/src folder and copy all the files over to the source/paho folder of the XDK application template created for this guide. Then navigate to the MQTTClient-C/src folder and copy the MQTTClient.c and MQTTClient.h files to the same source/paho folder of the XDK project - do not copy the subfolders into the project. For more information on the MQTT Paho Client see the website at http://www.eclipse.org/paho/.

The embedded C MQTT Paho Client configuration for the XDK must also be included in the XDK project for this guide and can be found in the „MQTT Paho“ implementation demo on the XDK homepage: xdk.io/demos. Once the demo is downloaded the folder „/paho“ inside the source path of the demo needs to be copied to the same location „/paho“ in the source path of the XDK project for this guide. Lastly the copied XDK specific MQTT header file „mqttXDK.h“ needs to be included in the “MQTTClient.h“ paho header file source code.

```
#include "MQTTPacket.h"
#include "stdio.h"
#include "mqttXDK.h" //Platform specific implementation header file
```

Once the source code is downloaded and placed the source files in the proper location, the project can be imported into the XDK workbench. Please see the Importing a project guide on the XDK website for more information.

At the end of this integration process the makefile of the XDK application template project created in this guide needs to be adapted as described in Code 1. The project folder structure of this XDK project is also shown there.

Code 1. Adaptation of the makefile in the XDK project of this guide

```
#All application header files under variable BCDS_XDK_INCLUDES
export BCDS_XDK_INCLUDES = \
-I$(BCDS_APP_SOURCE_DIR)/paho/XDK \
    -I$(BCDS_APP_SOURCE_DIR)/paho \
    -I$(BCDS_APP_SOURCE_DIR)

#Source files under variable BCDS_XDK_APP_SOURCE_FILES
export BCDS_XDK_APP_SOURCE_FILES = \
       $(BCDS_APP_SOURCE_DIR)/main.c \
       $(BCDS_APP_SOURCE_DIR)/XdkApplicationTemplate.c \
       $(BCDS_APP_SOURCE_DIR)/paho/XDK/mqttXDK.c \
       $(BCDS_APP_SOURCE_DIR)/paho/MQTTClient.c \
       $(BCDS_APP_SOURCE_DIR)/paho/MQTTConnectClient.c \
       $(BCDS_APP_SOURCE_DIR)/paho/MQTTConnectServer.c \
       $(BCDS_APP_SOURCE_DIR)/paho/MQTTDeserializePublish.c \
       $(BCDS_APP_SOURCE_DIR)/paho/MQTTFormat.c \
       $(BCDS_APP_SOURCE_DIR)/paho/MQTTPacket.c \
       $(BCDS_APP_SOURCE_DIR)/paho/MQTTSerializePublish.c \
       $(BCDS_APP_SOURCE_DIR)/paho/MQTTSubscribeClient.c \
       $(BCDS_APP_SOURCE_DIR)/paho/MQTTSubscribeServer.c \
       $(BCDS_APP_SOURCE_DIR)/paho/MQTTUnsubscribeClient.c \
       $(BCDS_APP_SOURCE_DIR)/paho/MQTTUnsubscribeServer.c
```

3. Network setup

To establish a MQTT communication architecture the clients and the broker need to be connected to a network. The easiest way to achieve this is to connect the XDK via its Wi-Fi interface to the same local network all the other MQTT communication partners are connected to. After providing the access point SSID and the password, the code can be compiled and flashed onto the XDK.
For further questions or problems in setting up please refer to the XDK Wi-Fi guide available at xdk.io/guides for further instructions. For just the minimal setup required to establish a Wi-Fi connection, an `appInitSystem()` function similar to Code 2 can be used.

Note: The code snippets in this guide do not cover error handling to keep them short and simple. Return codes can be found in the API documentation of the XDK library (see xdk.io/xdk-api).

Code 2. Minimal Wi-Fi setup

```
#include "BCDS_WlanConnect.h"
#include “BCDS_NetworkConfig.h"

void appInitSystem(void * CmdProcessorHandle, uint32_t param2){
    if (CmdProcessorHandle == NULL)
        printf("Command processor handle is null \n\r");
        assert(false);
    }
    BCDS_UNUSED(param2);

    WlanConnect_SSID_T connectSSID = (WlanConnect_SSID_T) "SSID";
    WlanConnect_PassPhrase_T connectPassPhrase = (WlanConnect_PassPhrase_T)
    "PW";
    WlanConnect_Init();
    NetworkConfig_SetIpDhcp(0);
    WlanConnect_WPA(connectSSID, connectPassPhrase, NULL);
    // following MQTT init procedures
}
```

4. Setup of XDK as MQTT Client

Now that the XDK application template project in this guide is provided with the needed prerequisites like the Paho MQTT client and the XDK adaptation of it in chapter 2 as well as the WiFi connection up and running in chapter 3, this chapter will focus on the MQTT implementation.

After basic steps for initializing the XDK as a standard-conform MQTT client the communication architecture will be built up between the XDK, an external MQTT broker and an external client. Apart from connecting to a server this chapter will show how to perform basic data exchange in both ways of sending information to and receiving information from external communication partners. Learning how the communication works in the introduction chapter 1 and applying it in this chapter will enable the introduction of MQTT into real-life applications developed with the XDK.

4.1 Initialization

Having set up the WiFi communication from the XDK to a local wireless network with Wi-Fi connection, this initialization part will focus on the initial MQTT configuration during startup.

Therefore it is advised to add the code lines displayed in Code 3 into the headers section and into the `appInitSystem()` function from Code 2. After connecting to the WiFi network a new network configuration is created and adjusted with some example data for the network host address of the MQTT broker (“broker.hivemq.com“ is used as broker in this guide) and the port on which the MQTT broker is listening for incoming MQTT connections from clients (here, common MQTT port 1883). The MQTT client is created with the prepared network information, the given command timeout duration until a response is awaited (here: 1000ms) and buffer sizes (here: 1000 bytes).

Code 3. MQTT network and client initialization

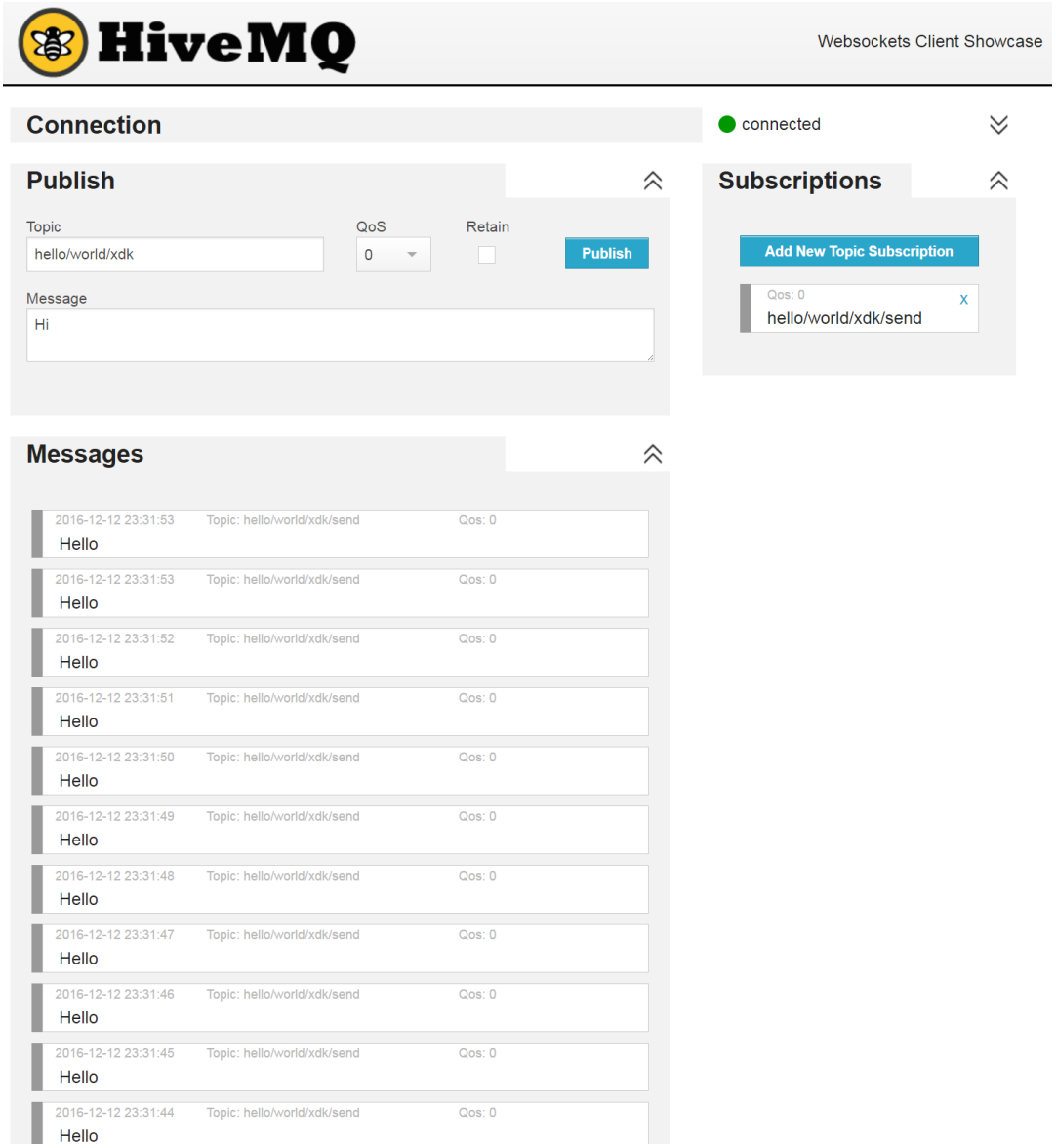
```
#include "MQTTClient.h"
#include "MQTTConnect.h"

#define CLIENT_BUFF_SIZE                                1000

static unsigned char buf[CLIENT_BUFF_SIZE];
static unsigned char readbuf[CLIENT_BUFF_SIZE];

Network n;
Client c;

void appInitSystem(void * CmdProcessorHandle, uint32_t param2)
{
    // previous WiFi Setup

    NewNetwork(&n);
    ConnectNetwork(&n, "broker.hivemq.com", 1883);

    MQTTClient(&c, &n, 1000, buf, CLIENT_BUFF_SIZE, readbuf,
       CLIENT_BUFF_SIZE);

    // following MQTT connect procedures
}
```

4.2 Connection to MQTT Server

As mentioned in chapter 4.1 an external MQTT server and client is used for validating if the XDK implementation for the MQTT client done in this guide is valid for talking to other standardized MQTT communication partners. To setup such a communication architecture a MQTT broker accessible via the host address “broker.hivemq.com“ is configured as the destination server for the XDK MQTT client in Code 3. For more details on how the MQTT broker is implemented the homepage can be visited under the following link: http://www.hivemq.com/try-out/.

Additionally it is suggested for our example MQTT communication architecture to have an external MQTT client connected to the broker for testing the publish / subscribe events. For that it is possible to work with an easy-to-use online client also provided by HiveMQ with a web socket solution: http://www.hivemq.com/demos/websocket-client/. The web client needs to be connected to the same MQTT broker as the XDK (here: “broker.hivemq.com“). The user interface of the web client can be seen in Picture 3 and enables interacting via publish / subscribe methods with other MQTT clients worldwide connected to this server and most importantly with the XDK MQTT client.

Picture 3. MQTT web socket client by HiveMQ



To perform the connection request to the MQTT broker, Code 4 needs to be added to the `appInitSystem()` function described in Code 2 and Code 3. The code shows how the connection parameters, as described in chapter 1, are prepared in one data structure and then passed on to the MQTT connect function with the previously configured MQTT client. It also shows that the lastWill flag is not used, MQTT version shall be 3.1, cleanSession flag is configured, the clientID string is given and the keepAlive interval is proposed with 100ms to the broker.

Code 4. MQTT client connecting to the broker

```
void appInitSystem(void * CmdProcessorHandle, uint32_t param2)
{
    // previous WiFi Setup
    // previous MQTT init procedures
    // MQTT connect procedures

    MQTTPacket_connectData data = MQTTPacket_connectData_initializer;
    data.willFlag = 0;
    data.MQTTVersion = 3;
    data.clientID.cstring = "xdk123";
    data.keepAliveInterval = 100;
    data.cleansession = 1;
    MQTTConnect(&c, &data);

    // following MQTT subscribe method
}
```

Note: It is also possible to connect to other MQTT brokers, e.g. local applications running on the same computer the XDK is connected to. To do so the connection data in Code 3 needs to be configured to the required broker host name or network address. For more information about TCP/IP sockets see the API documentation of the XDK library (see xdk.io/xdk-api).

4.3 Subscribing to Topics

Subscribing to a MQTT topic requires two things. First, the client needs to subscribe to a topic set by the string filter (here “hello/world/xdk“), the QoS level (here: 0) and a callback function (here: “clientRecvr“) to be executed whenever the subscribed topic received data. This part is shown in Code 5, appended into the appInitSystem() function.

Code 5. MQTT client subscribing to a topic

```
void appInitSystem(void * CmdProcessorHandle, uint32_t param2)
{
    // previous WiFi Setup
    // previous MQTT init procedures
    // previous MQTT connect procedures
    // MQTT subscribe method

       MQTTSubscribe(&c, (char*) "hello/world/xdk", QOS0, clientRecvr);

    // following MQTT publish task creation
}
```

Second, the callback function needs to be implemented in order to enable parsing of the arrived data messages on the subscribed topic or other incoming data handling.
The implementation can be seen in Code 6 where the incoming MQTT message is parsed into the subscribed topic the message is arriving for and the payload it is carrying in order to be printed to the XDK workbench console. Another client can now publish any data to the topic “hello/world/xdk“ which will lead to the execution of the XDK MQTT client callback function `clientRecvr()`.

Code 6. MQTT client incoming data handling

```
static void clientRecvr(MessageData* md)
{
       MQTTMessage* message = md->message;

       printf("Subscribed Topic, %.*s, Message Received: %.*s\r\n",
       md->topicName->lenstring.len, md->topicName->lenstring.data,
       (int)message->payloadlen, (char*)message->payload);
}
```

4.4 Publishing Data

The only thing missing in the previous steps is sending data from the XDK MQTT client to the connected MQTT broker by using the publishing mechanism and, from there, to subscribed clients. To do so a task needs to be created in the appInitSystem() function, which also completes it for the purpose of this guide.
The task creation and the definition of constant parameters used in the creation process can be seen in code 7. Creating a task requires a callback function to be called upon (here: “clientTask“) as well as the task stack size (here: 1024) and the priority (here: 1) as defined in the parameters.

Code 7. MQTT client publishing task creation

```
#define CLIENT_TASK_STACK_SIZE    1024
#define CLIENT_TASK_PRIORITY         1


void appInitSystem(void * CmdProcessorHandle, uint32_t param2)
{
    // previous WiFi Setup
    // previous MQTT init procedures
    // previous MQTT connect procedures
    // previous MQTT subscribe method
    // MQTT publish task creation

       xTaskCreate(clientTask, (const char * const) "Mqtt Client App",
       CLIENT_TASK_STACK_SIZE, NULL, CLIENT_TASK_PRIORITY, null);
}
```

In order to publish data when the task is created, the callback function to be executed needs to be detailed. The Implementation of the callback function is described in Code 8 where timer handling on basic XDK and MQTT level is performed as well as a MQTT message is prepared and published. First an endless loop is necessary for the freeRTOS task to continue running. Furthermore a task delay of 1 second helps with the XDK MQTT client not stressing the MQTT broker unnecessarily with aggressive timing.

Afterwards the MQTT message is built out of the single components like message ID (for QoS 1 and 2) as well as setting the QoS level itself. By adding the the payload bytes (here: string with “Hello“) and the payload length, the MQTT message data structure is completed and ready to be sent to the broker.

With the MQTT message being ready the MQTT client is able to publish it to the given topic. At the end the MQTT client needs to perform the yield method to ensure the keepAlive timing arrangement with the MQTT broker. Further optional parameters can be attached to the MQTT message object like the duplicate and retain flags described in the chapter 1 which are not used in this implementation guide.

Code 8. MQTT client publish task implementation

```
#define CLIENT_YIELD_TIMEOUT     10
#define SECONDS(x)               ((portTickType) (x * 1000) / portTICK_RATE_MS)

static uint32_t clientMessageId = 0;

static void clientTask(void *pvParameters)
{
       MQTTMessage msg;

      for(;;)
      {
          vTaskDelay(SECONDS(1));

          msg.id = clientMessageId++;
          msg.qos = 2;
          msg.payload = "Hello";
          msg.payloadlen = sizeof("Hello")-1;
          MQTTPublish(&c, "hello/world/xdk/send", &msg);

          MQTTYield(&c, CLIENT_YIELD_TIMEOUT);
       }
}
```

The data being published to the MQTT broker can be printed out in the HiveMQ web socket client by subscribing beforehand to the topic “hello/world/xdk/send“.

5. Document History and Modification

Rev. No.	Chapter	Description of modification/changes	Editor	Date
2.0		Version 2.0 initial release	AFS	2017-08-17