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 |
| 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.<br>**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. This guide gives an introduction into the MQTT protocol and the XDK MQTT API.

## Table of Contents



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:

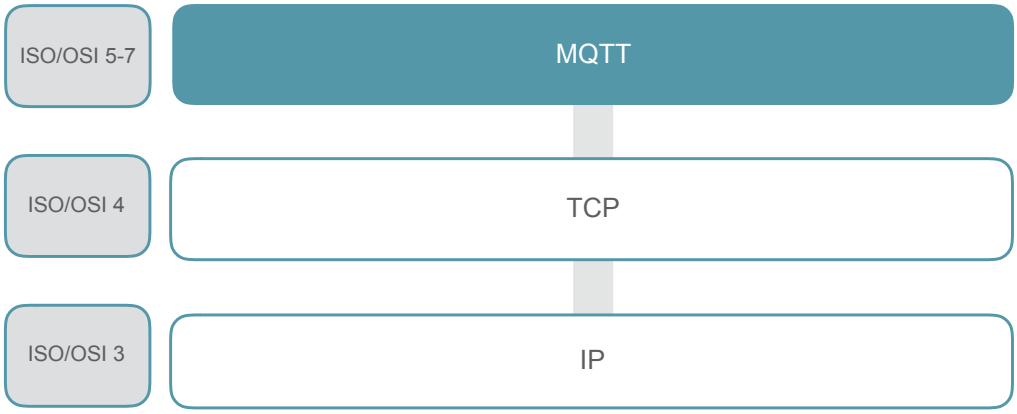
- *Workbench Installation*
- *Workbench First Steps*
- *XDK Guide FreeRTOS*
- *XDK Guide Wi-Fi*

# 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, 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 server which hosts the MQTT broker 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 request 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.<br>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/<br>-QoS/<br>-Message/<br>-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. |

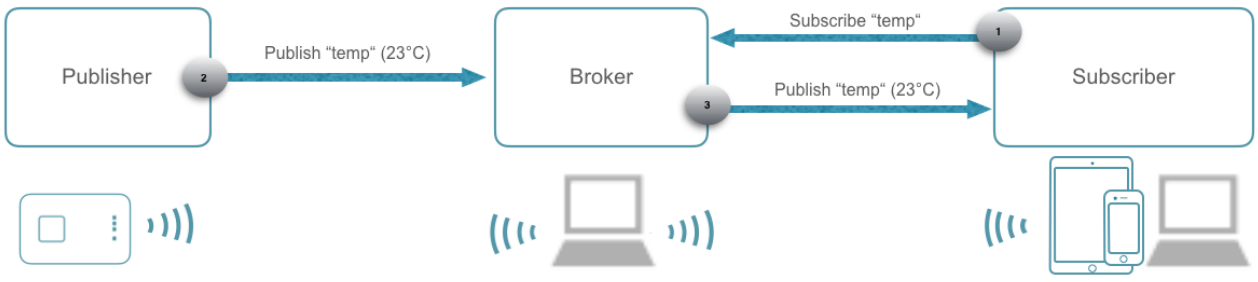
Upon receiving a client connect request the broker must respond with a connect acknowledgement containing the two elements described in table 3.

**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 (<br>qos1, topic1,<br>qos2, topic2,<br>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 and Overview

This guide presents a basic implementation using the MQTT API of the ServalStack. To make the API available to the application, the header file `Serval_Mqtt.h` has to be included in the implementation files. Additionally, a few other APIs are used in this guide. **Code 1** presents how to include the header files used in this guide.

**Code 1.** Including necessary header-files

```
// MQTT API
#include "Serval_Mqtt.h"

// WiFI / Networking API
#include "BCDS_NetworkConfig.h"
#include "BCDS_WlanConnect.h"
#include "PAL_initialize_ih.h"
#include "PAL_socketMonitor_ih.h"
```

Additionally, in your project browse to

SDK > xdk110 > Libraries > ServalStack > 3rd-party > Serval Stack > api > Serval_Defines.h

And set the define of `SERVAL_ENABLE_MQTT` to 1, if this is not already done. If this is set to 0, the API will not be available in the application.

Some important constants and variables will be used throughout the guide. They are listed in **Code 2.** The first two constants, `WIFI_SSID` and `WIFI_PW`, should contain your local WiFi credentials, to be able to connect to the MQTT broker. This broker can be either in your local network, or be accessed via an internet connection.

In either case, the broker's address has to be inserted into `MQTT_BROKER_HOST`. By default, this guide uses the broker located at broker.hivemq.com. This broker can be connected to via a websocket client here, to test out the application. The standard port for MQTT is 1883, and should not be changed.

**Code 2.** Constant definitions and variables

```
#define WIFI_SSID         "yourWifiSSID"
#define WIFI_PW           "yourWifiPW"

#define MQTT_BROKER_HOST  "broker.hivemq.com"
#define MQTT_BROKER_PORT  1883

static MqttSession_T session;
static MqttSession_T *session_ptr = &session;
```

Finally, the variable `session` of type `MqttSession_T` will contain all the connection and server information, and is used by nearly every function of the MQTT API. This variable's pointer should thus be global, to be accessed from any point of the implementation.

The following table lists some of the important functions of the MQTT API.

**Table 6.** MQTT API (excerpt)

| Function | Description |
|---|---|
| Mqtt_initialize() | This function is used to initialize the MQTT module and should be used before any other function of the MQTT API |
| Mqtt_initializeInternalSession() | This function is used to initialize the internal session. The session will hold all the information needed to connect to an MQTT broker |
| (*MqttEventCallback_T)() | This is a function that has to be implemented by the user of the API. On every MQTT Event, this function is called and will be used for handling the events. For more details, see chapter 4.2 |
| Mqtt_connect() | This function attempts to connect to the target host. The target host is a field of the MqttSession_T and should be configured before this function is called. |
| Mqtt_publish() | This function publishes data. The inputs are:<br>- Topic (of type StringDescr_T)<br>- Payload ( + Length)<br>- Quality of Service<br>- Retain Flag |
| Mqtt_subscribe() | This function subscribe to a list of topics with their respective QoS. The inputs are:<br>- List of Topics (of type StringDescr_T)<br>- List of QoS (of type Mqtt_qos_t) |

All these functions, except `Mqtt_initialize()`, receive the pointer to a `MqttSession_T` variable as well.

# 3. Network setup

To establish communication the XDK and the MQTT broker, they need to be connected to the same network, which is usually via internet. The easiest way to achieve this is to connect the XDK via its Wi-Fi interface to a local Wi-Fi network which provides an internet connection. 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, a function similar to the one in **Code 3** can be used. The code uses a WiFi SSID and PW, defined by `WIFI_SSID` and `WIFI_PW` respectively.

**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 3.** Minimal Network Connection

```
static void networkSetup(void){
	WlanConnect_SSID_T connectSSID = (WlanConnect_SSID_T) WIFI_SSID;
	WlanConnect_PassPhrase_T connectPassPhrase =
					(WlanConnect_PassPhrase_T) WIFI_PW;
	WlanConnect_Init();
	NetworkConfig_SetIpDhcp(0);
	WlanConnect_WPA(connectSSID, connectPassPhrase, NULL);

	PAL_initialize();
	PAL_socketMonitorInit();
}
```

# 4. Setup of XDK as MQTT Client

Now that the basic setup for using the API is done and the XDK is connected to a Wi-Fi network, this chapter will focus on implementing a MQTT Client on the XDK.

After basic steps for initializing the XDK as a standard-conform MQTT, client the communication architecture will be built up between the XDK, and an external MQTT broker. Apart from connecting to an MQTT broker, 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

Before the MQTT API can be used, the MQTT module must first be initialized. The initialization of the MQTT client is accomplished in two steps. For this, a function similar to the one implemented in **Code 4** can be used. First, the MQTT module itself is initialized by calling `Mqtt_initialize()`. Then, the variable session of type `MqttSession_T` is initialized. The declaration of this variable and its pointer is shown in chapter 2. It is crucial that this variable is global, since it will be used in nearly every other function of the MQTT API.

**Code 4.** Initializing the MQTT Client

```
retcode_t init(void){
	retcode_t rc_initialize = Mqtt_initialize();
	if (rc_initialize == RC_OK) {
		session_ptr = &session;
		Mqtt_initializeInternalSession(session_ptr);
	}
	return rc_initialize;
}
```

## 4.2 Event Handler

The current API is event-based. That means that the application reacts to every pre-defined event with a callback. Events can be Connection-events, incoming data, outgoing data, acknowledgements, and more. **Code 5** shows the general structure of this event callback. The handler checks the event-type and acts accordingly.

The type of the variable `eventData` depends on the event. For more information on this, refer to the header-file `Serval_Mqtt.h`. The events and the corresponding data types are listed there. The event `MQTT_CONNECTION_ESTABLISHED` holds special significance, because subscribing and publishing to topics can only be done after this event occured.

**Code 5.** Event Callback Implementation

```
retcode_t event_handler(MqttSession_T* session, MqttEvent_t event,
              const MqttEventData_t* eventData) {
       BCDS_UNUSED(session);
       switch(event){
              case MQTT_CONNECTION_ESTABLISHED:
                     handle_connection(eventData->connect);
                     // subscribing and publishing can now be done
                     // subscribe();
                     // publish();
                     break;
              case MQTT_CONNECTION_ERROR:
                     handle_connection(eventData->connect);
                     break;
              case MQTT_INCOMING_PUBLISH:
                     handle_incoming_publish(eventData->publish);
                     break;
              case MQTT_SUBSCRIPTION_ACKNOWLEDGED:
                     printf("Subscription Successful\n\r");
                     break;
              case MQTT_PUBLISHED_DATA:
                     printf("Publish Successful\n\r");
                     break;
              default:
                     printf("Unhandled MQTT Event: %d\n\r", event);
                     break;
       }
       return RC_OK;
}
```

The only events this guide handles are connection events and incoming publish events. There are prints for successful outgoing publish and subscription messages, as well. The functions in **Code 6** and **Code 7** are examples for how these events can be handled. For any other event, a new case has to be added, otherwise only the event's number will be printed in the default case.

The function `handle_connection()` is used in both `MQTT_CONNECTION_ESTABLISHED` and `MQTT_CONNECTION_ERROR`, because these events have the same data-type. It simply prints the connect return code sent by the server, at which the connection attempt directed.

**Code 6.** Handling Connection Events

```
static void handle_connection(MqttConnectionEstablishedEvent_T connectionData){
       int rc_connect = (int) connectionData.connectReturnCode;
       printf("Connection Event:\n\r"
              "\tServer Return Code: %d (0 for success)\n\r",
              (int) rc_connect);
}
```

The function, which handles incoming data from the topics to which the XDK subscribed, prints the message's payload and the topic it has been published on. The payload and the topic do not have a string-terminating character `"\0"`, which is why they are first written into a buffer using the function `snprintf()`, where a null-terminating character is automatically added.

**Code 7.** Handling Incoming Data

```
static void handle_incoming_publish(MqttPublishData_T publishData){
	int topic_length = publishData.topic.length + 1;
	int data_length = publishData.length + 1;
	char published_topic_buffer[topic_length];
	char published_data_buffer[data_length];

	snprintf(published_topic_buffer, topic_length, publishData.topic.start);
	snprintf(published_data_buffer, data_length,
			(char *) publishData.payload);

	printf("Incoming Published Message:\n\r"
			"\tTopic: %s\n\r"
			"\tPayload: %s\n\r",
			published_topic_buffer,
			published_data_buffer);
}
```

## 4.3 Configuring the Session

Now that the event handler is set up, the session has to be configured, to make a connection possible. The session holds the connect information, and thus has fields for all of the parameters from **Table 1** and **Table 2**. Additionally, this variable holds information about the target host. The target host is the broker the XDK will connect to. The codes **Code 8**, **Code 9** and **Code 10** show a minimal configuration necessary for a valid connection. Finally, the session also gets a reference to the event handler.

The first part is setting the target, which is shown in **Code 8**. The field `target` of the session variable is of type `SupportedUrl_T`, which is why `SupportedUrl_fromString()` is used to set the target-information. This function receives a full URL, such as "mqtt://255.255.255.255:1883/some/path" and parses the necessary information. Note that the host-adress has to be a valid IP-address.

**Code 8.** Configuring the session's target

```
void config_set_target(void){
	static char mqtt_broker[64];
	const char *mqtt_broker_format = "mqtt://%s:%d";
	char server_ip_buffer[13];
	Ip_Address_T ip;

	PAL_getIpaddress((uint8_t *) MQTT_BROKER_HOST, &ip);
	Ip_convertAddrToString(&ip, server_ip_buffer);
	sprintf(mqtt_broker, mqtt_broker_format,
			server_ip_buffer, MQTT_BROKER_PORT);

	SupportedUrl_fromString(mqtt_broker,
			(uint16_t) strlen(mqtt_broker), &session_ptr->target);
}
```

Since the host-address used in this guide is "broker.hivemq.com", the address is first resolved using the function `PAL_getIpaddress()` and the resulting IP-address is converted to a string (this function also works if the address is an IP-address). The IP-address-string and the port will then be inserted into the `mqtt_broker` buffer.

Finally, the string `mqtt_broker` can be used in `SupportedUrl_fromString()`, along with the buffer's length and a pointer to the session's `target` field. The buffer has to be static since the string will not be copied by the function.

Setting the connect data is relatively straight forward. `MQTTVersion` has to be set to 3 for the protocol's version 3.1. If the protocol version 3.1.1 should be used instead, set this field to 4. `cleanSession` and `keepAliveInterval` can be set to suit the use case. If `will.haveWill` is set to `true` instead of `false`, the other field of `will` have to be set as well. The `clientID` is arbitrary in this case. The following code shows example values for each field.

**Code 9**. Configuring the session's connect data

```
void config_set_connect_data(void){
	static char *device_name = "XDK110_Guide_Device";

	session_ptr->MQTTVersion = 3;
	session_ptr->keepAliveInterval = 100;
	session_ptr->cleanSession = true;
	session_ptr->will.haveWill = false;

	StringDescr_T device_name_descr;
	StringDescr_wrap(&device_name_descr, device_name);
	session_ptr->clientID = device_name_descr;
}
```

Finally, the session's event handler has to be set, as seen in the following code.

**Code 10.** Setting the event-handler

```
void config_set_event_handler(void){
	session_ptr->onMqttEvent = event_handler;
}
```

## 4.4 Connecting To MQTT Broker

Establishing a connection is straight-forward, since the target host and the connect details are already configured in chapter 4.3. To connect, a function such as seen in **Code 11** can be used. It is important to handle possible errors. One of the errors the function `Mqtt_connect()` throws, is when the XDK cannot find the target host. This can be either because the host does not exist, or if the XDK cannot access the host at all.

It is also crucial to understand that the errors of `Mqtt_connect()` are semantically different from the event `MQTT_CONNECTION_ERROR`. The function throws client-related errors, the event throws server-related errors, such as a connection refusal.

**Code 11.** Establishing a Connection

```
retcode_t connect(void){
	retcode_t rc = RC_INVALID_STATUS;
	rc = Mqtt_connect(session_ptr);
	if (rc != RC_OK) {
		printf("Could not connect, error 0x%04x\n", rc);
	}
	return rc;
}
```

To start the application by using the functions implemented in this chapter and the previous chapters, the following implementation of `appInitSystem()` can be used:

**Code 12.** appInitSystem implementation

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

    retcode_t rc = RC_MQTT_NOT_CONNECTED;
    networkSetup();
    rc = init();
    if(rc == RC_OK){
        config_set_target();
        config_set_connect_data();
        config_set_event_handler();
        connect();
    } else {
       printf("Initialize Failed\n\r");
    }
}
```

## 4.5 Subscribing to a Topic

As soon as the event `MQTT_CONNECTION_ESTABLISHED` occurs, it is possible to subscribe to a topic on the MQTT Server. The function `Mqtt_subscribe()` requires an array of topics and the corresponding Qualities of Service in another array. The first array is of type `StringDescr_T`, which is why each topic has to be wrapped into a `StringDescr_T` by using the function `StringDescr_wrap`. The array of QoS is of type `Mqtt_qos_t`, but the possible values are essentially numbers. The possible values are listed at the type definition of `Mqtt_qos_t` in `Serval_Mqtt.h`.

**Code 13** shows a function that subscribes to a topic. The variables `subscribe_topic` and `subscription_topics` should keep their values until a `MQTT_SUBSCRIPTION_ACKNOWLEDGED` event has occured. This is why the variables are declared static in this context, since the event is not handled by the callback that has been implemented in chapter 4.2.

**Code 13.** Subscribing to a single Topic

```
static void subscribe(void){
      static char *sub_topic = "your/subscribe/topic";
      static StringDescr_T subscription_topics[1];
      static Mqtt_qos_t qos[1];
      StringDescr_wrap(&(subscription_topics[0]), sub_topic);
      qos[0] = MQTT_QOS_AT_MOST_ONE;
      Mqtt_subscribe(session_ptr, 1, subscription_topics, qos);
}
```

Of course, the number of topics can be greater than one, but this example only subscribes to one topic. If more topics are added, each individual topic requires a corresponding QoS.

## 4.6 Publishing on a Topic

As soon as the event `MQTT_CONNECTION_ESTABLISHED` occurs, it is possible to publish on a topic on the MQTT Server to which the XDK is connected. The function `Mqtt_publish()` requires all of the parameters from **Table 4**, except the packetID and the duplicate Flag. The topic is wrapped in a variable of type `StringDescr_T`, the payload is an ordinary array of type char and its length. The `qos` is of type `Mqtt_qos_t`. The values the `qos` can have are listed in the typedef of `Mqtt_qos_t` in `Serval_Mqtt.h`. The retain flag is either true or false.

**Code 14** shows a function that publishes a message on a topic. Please note that the variables `pub_message` and `pub_topic` should keep their respective values until an `MQTT_PUBLISHED_DATA` event has occured. This is why the variables are declared static in this context, since the event is not handled by the callback that has been implemented in chapter 4.2.

**Code 14.** Publishing a Message on a Topic

```
static void publish(void){
	static char *pub_message = "Hello World";
	static char *pub_topic = "your/publish/topic";
	static StringDescr_T pub_topic_descr;
	StringDescr_wrap(&pub_topic_descr, pub_topic);

	Mqtt_publish(session_ptr, pub_topic_descr, pub_message,
			strlen(pub_message), MQTT_QOS_AT_MOST_ONE, false);
}
```

## 4.7 Recommended Application Flow

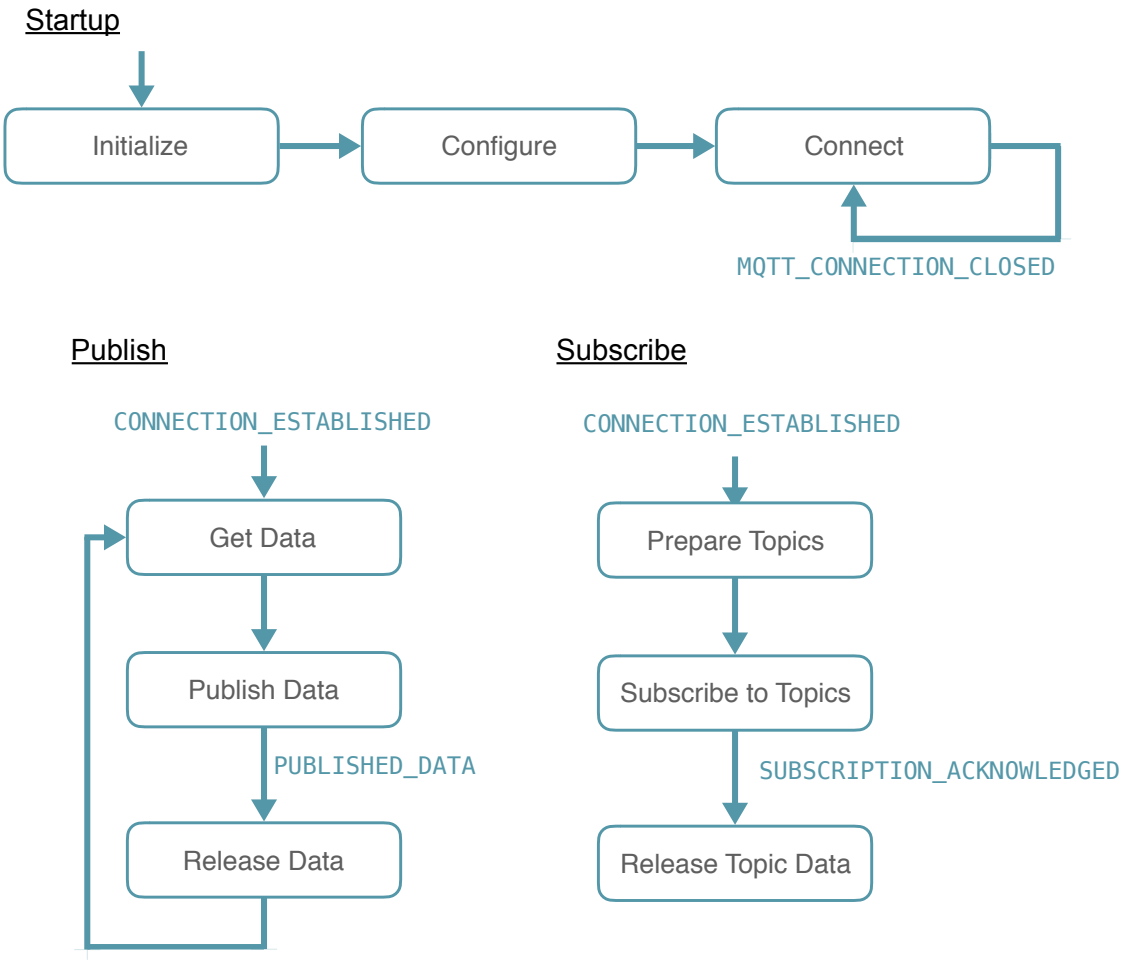
As a recommendation, an application based on the MQTT API would have a flow of event-handling that is similar to the graph in **Picture 5**.

The first step is to start up the application, which consists of initializing the module and internal session, configuring the session and then connecting. Alternatively, whenever the connection closes, a reconnect may be attempted.

For Publishing, it is recommended to first load the data into a buffer and then publish it. The buffer should not be changed until the event `MQTT_PUBLISHED_DATA` is fired, but afterwards, the data in the buffer may be released.

The same holds for Subscribing. The topic data should not be released until the event `MQTT_SUBSCRIPTION_ACKNOWLEDGED` has occured. Otherwise, the topic to which the XDK subscribed might be an arbitrary one in the worst case.

**Picture 3.** Application Flow as a Graph



# 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 |