MQTT - User Manual

MU214606 Rev. C

June 5, 2020

According to current legislation in Brazil, the Consumer Defense Code, we are giving the following information to clients who use our products, regarding personal safety and premises.

The industrial automation equipment, manufactured by Altus, is strong and reliable due to the stringent quality control it is subjected to. However, any electronic industrial control equipment (programmable controllers, numerical commands, etc.) can damage machines or processes controlled by them when there are defective components and/or when a programming or installation error occurs. This can even put human lives at risk. The user should consider the possible consequences of the defects and should provide additional external installations for safety reasons. This concern is higher when in initial commissioning and testing.

The equipment manufactured by Altus does not directly expose the environment to hazards, since they do not issue any kind of pollutant during their use. However, concerning the disposal of equipment, it is important to point out that built-in electronics may contain materials which are harmful to nature when improperly discarded. Therefore, it is recommended that whenever discarding this type of product, it should be forwarded to recycling plants, which guarantee proper waste management.

It is essential to read and understand the product documentation, such as manuals and technical characteristics before its installation or use. The examples and figures presented in this document are solely for illustrative purposes. Due to possible upgrades and improvements that the products may present, Altus assumes no responsibility for the use of these examples and figures in real applications. They should only be used to assist user trainings and improve experience with the products and their features.

Altus warrants its equipment as described in General Conditions of Supply, attached to the commercial proposals.

Altus guarantees that their equipment works in accordance with the clear instructions contained in their manuals and/or technical characteristics, not guaranteeing the success of any particular type of application of the equipment.

Altus does not acknowledge any other guarantee, directly or implied, mainly when end customers are dealing with third-party suppliers. The requests for additional information about the supply, equipment features and/or any other Altus services must be made in writing form. Altus is not responsible for supplying information about its equipment without formal request. These products can use EtherCAT® technology (www.ethercat.org).

COPYRIGHTS

OPEN SOURCE SOFTWARE NOTICE

Contents

1. Introduction

This manual describes library **LibMQTT**. This library makes available one function block, called MQTT_CLIENT, and some related structures and enumerations, as shown in the figure below. The FB MQTT_CLIENT allows the user stablish a communication with a MQTT Broker, which can be online (cloud) or local.

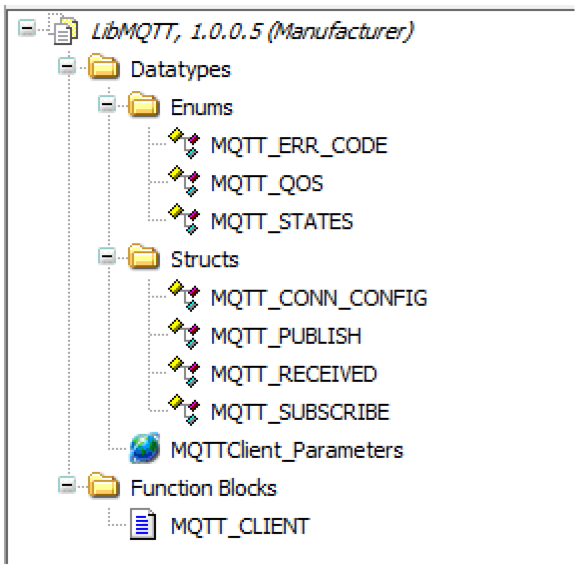


Figure 1: Components of library LibMQTT

ATTENTION:
This document doesn't have the objective to teach about the MQTT itself. It only explains some features and variables related to the protocol.

ATTENTION:
On systems with CPU redundancy, the MQTT Client Function Block must be called on the non-redundant part of the program (NonSkippedPrg).

Message Queuing Telemetry Transport (MQTT) is a protocol based on publish/subscribe. This protocol sends and receives messages asynchronously. There is a concept of client and broker. All clients connect to a broker and publish messages on topics. This messages are distributed by the broker according to the clients subscribed for that topic.

This document is organized in the following chapters:

- Chapter **MQTT Client Working Flow** describes how the function block communicate with the broker.
- Chapter **MQTT Client I/O Structure** shows all inputs and outputs necessary (and optional) to use the function block.
- Chapter **Application Example** gives an example to the user, improving the function block understand.

1.1. Technical Support

For Altus Technical Support contact in São Leopoldo, RS, call +55 51 3589-9500. For further information regarding the Altus Technical Support existent on other places, see http://www.altus.com.br/site_en/ or send an email to altus@altus.com.br.

If the equipment is already installed, you must have the following information at the moment of support requesting:

- The model of the used equipments and the installed system configuration
- The product serial number
- The equipment revision and the executive software version, written on the tag fixed on the product side
- CPU operation mode information, acquired through MasterTool IEC XE
- The application software content, acquired through MasterTool IEC XE
- Used programmer version

1.2. Warning Messages

In this manual, the warning messages will be presented in the following formats and meanings:

DANGER:
Reports potential hazard that, if not detected, may be harmful to people, materials, environment and production.

CAUTION:
Reports configuration, application or installation details that must be taken into consideration to avoid any instance that may cause system failure and consequent impact.

ATTENTION:
Identifies configuration, application and installation details aimed at achieving maximum operational performance of the system.

2. MQTT Client Working Flow

The figure below shows the function block interface. Note that the structure is very simple. However, it uses some structures to configure the MQTT client, which increase the complexity. Next chapter will describe in detail all inputs and outputs parameters.

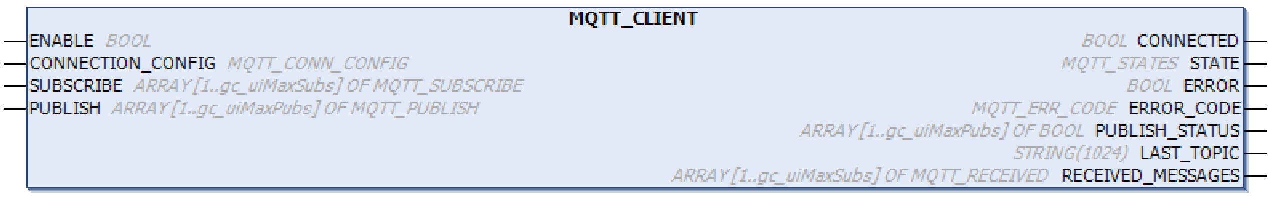

MQTT_CLIENT
ENABLE BOOL
CONNECTION_CONFIG MQTT_CONN_CONFIG
SUBSCRIBE ARRAY[1..gc_uiMaxSubs] OF MQTT_SUBSCRIBE
PUBLISH ARRAY[1..gc_uiMaxPubs] OF MQTT_PUBLISH
BOOL CONNECTED
MQTT_STATES STATE
BOOL ERROR
MQTT_ERR_CODE ERROR_CODE
ARRAY[1..gc_uiMaxPubs] OF BOOL PUBLISH_STATUS
STRING(1024) LAST_TOPIC
ARRAY[1..gc_uiMaxSubs] OF MQTT_RECEIVED RECEIVED_MESSAGES

Figure 2: MQTT Client Parameters

The working flow of the function block is based in a machine state, so it needs to be called periodically to work properly. Typically, the Function Block will be called on the main application cycle (MainTask).

CAUTION:
For CPU models NX3010, NX3020 and NX3030, the period of execution of the MQTT Client Function Block must be greater than 15 ms. For example, if the Function Block is called on MainTask context, the MainTask interval must be greater than this value.

The diagram below shows how this process is made. This is important to understand how the connection is made and the MQTT_CLIENT flow for subscribe and publish.

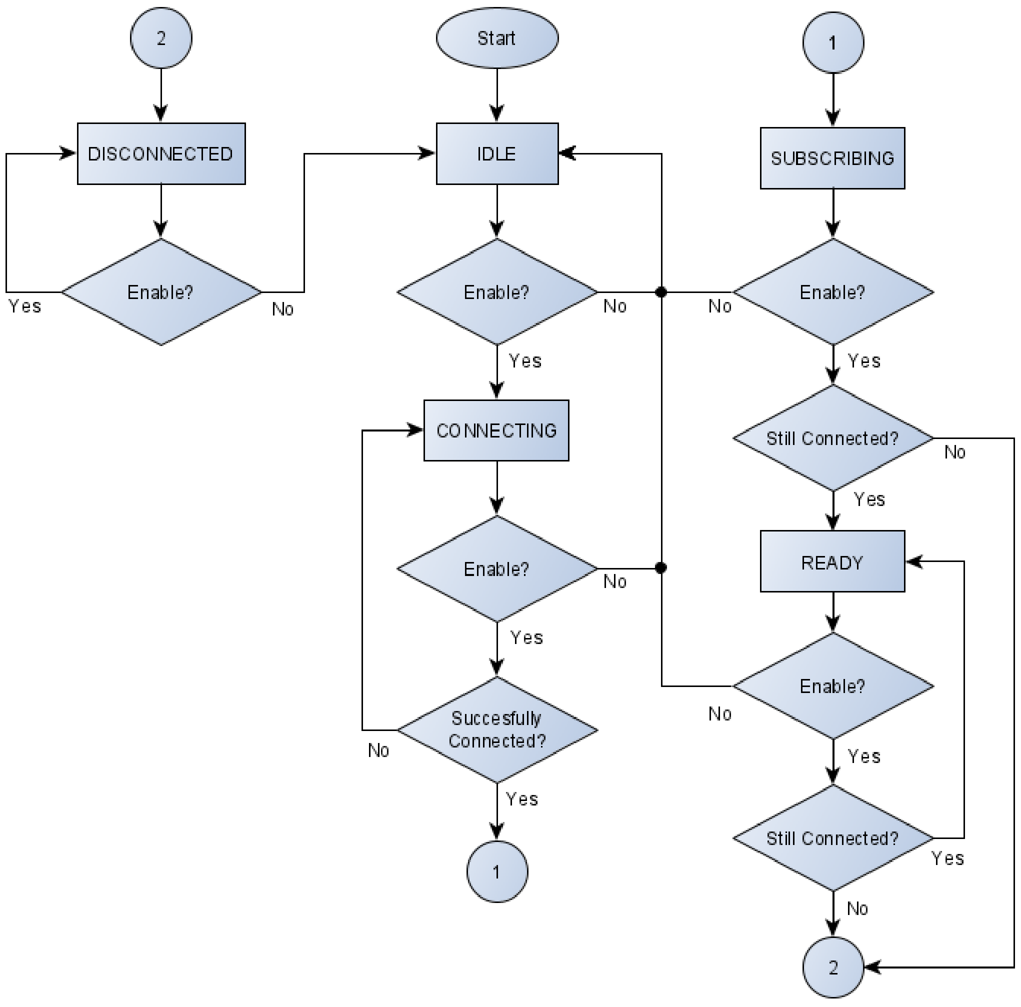


Figure 3: MQTT Client Flowchart

According the flowchart, the function block starts in IDLE. When ENABLE goes to TRUE, the MQTT client will try to connect to the broker, which is specified on the MQTT_CONN_CONFIG structure. After that, the client will subscribe to all topics entered in the MQTT_SUBSCRIBE structure array. Finally, it will be ready to receive and publish messages. A topic is like an address, where the broker and the client exchange messages, similar to a forum in a web site, which has a moderator (broker), the users (clients) and the specific conversation topic. The MQTT_RECEIVED_MESSAGES structure array contains details about all subscribed topics, while the MQTT_PUBLISH structure array contains the parameters needed to publish a message to the broker. It is possible to notice that the subscription is made only once during the connection. So, if you want to change the topics that the client will subscribe, or some other connection configuration, it is necessary re-enable the function block to validate this new information. Only the publish structure can be modified without disable the MQTT client.

3. MQTT Client I/O Structure

On the previous chapter, we understand how the function block works. In this chapter, we will see how to configure the MQTT client, which will organized on two sections. The first section will explain all input parameters. And the second section will describe all outputs.

3.1. Input Parameters

All function block inputs described in this section are shown in the list below, which will be organized on sections:

- ENABLE
- CONNECTION_CONFIG
- SUBSCRIBE
- PUBLISH

3.1.1. ENABLE - Enable the MQTT Client

- **Type:** BOOL
- **Default:** FALSE

This input is responsible to start the function block. When it is TRUE, the working flow will start, changing the MQTT client state from IDLE to READY, if the connection succeed (see Figure 3). If the client is connected to the broker and the ENABLE goes to FALSE, the client will disconnect.

3.1.2. CONNECTION_CONFIG

- **Type:** MQTT_CONN_CONFIG

This input allows the user to configure the connection. All variables in the structure are used only in the CONNECTING state.

CONNECTION_CONFIG.sClientId

- **Type:** STRING(255)
- **Default:** ''

This variable corresponds to the Client ID, which will be how the broker will identify the client. The input must be a string (up to 255 characters). This is an optional input. If an empty string is passed as argument, a random Client ID will be generated.

CONNECTION_CONFIG.bCleanSession

- **Type:** BOOL
- **Default:** TRUE

The Clean Session parameter indicates to the broker if the session for the respective Client ID must be saved or not. If this variable is set as FALSE, the broker will save the session for the Client ID. This parameter must be TRUE if sClientId is an empty string. This avoid the client to keep session trash on the broker, once the client not know how the broker identifies it.

CONNECTION_CONFIG.sUser

- **Type:** STRING(255)
- **Default:** ''

This parameter is the username (up to 255 characters), which will be required by the broker in the connection. It is an optional input. However, if the connection needs a specific credential, the client will be disconnected during the CONNECTING state when this input is set with the wrong username.

CONNECTION_CONFIG.sPass

- **Type:** STRING(255)
- **Default:** ''

This is the password that correspond to the username (up to 255 characters). If there is no password, only a username, it is possible to keep this input empty. However, if the broker credential required a password, you need to enter the right one to stablish the connection. The password is one way allowed by MQTT to guarantee some security, but it is not the most secure.

CONNECTION_CONFIG.sHostname

- **Type:** STRING(255)
- **Default:** ”

The hostname is the URL or IP address of the broker (up to 255 characters). The MQTT protocol uses the TCP/IP interface, so, it is necessary to inform the broker address. This input is mandatory, once the client must know the host it will try to connect. The broker can be online (e.g. Alibaba Broker) or local (e.g. Windows Mosquitto).

CONNECTION_CONFIG.uiPort

- **Type:** UINT
- **Default:** 1883

This parameter is the TCP/IP port that the broker and the client will use to communicate via MQTT. The standard port for MQTT is 1883. However, for TLS encrypted connection, the default is 8883. This input must be the same that the broker will listen.

CONNECTION_CONFIG.uiKeepAlive

- **Type:** UINT
- **Default:** 60

The Keep Alive is a parameter used to verify if the client is connected to the broker. This input is the interval of the ping request sent to the broker, in seconds. If the client does not receive any answer, it will be assumed the connection is not alive.

CONNECTION_CONFIG.sLastWillTopic

- **Type:** STRING(1024)
- **Default:** ”

The Last Will is a feature that allows the client to configure a message to be published when it is disconnected. The broker will publish the message to the topic specified on this parameter (up to 1024 characters). If this string is empty, the Last Will message will not be configured by the broker.

CONNECTION_CONFIG.pbLastWillPayload

- **Type:** POINTER TO BYTE
- **Default:** NULL

Besides the topic, the Last Will message must receive a variable address - ADR(variable), which is the message itself, also called payload. This variable can be of any type (e.g. STRING, DWORD, BYTE...). The address is necessary once the MQTT protocol allows any kind of variable. If this pointer is NULL, the Last Will message will not be configured by the broker, even if the topic was configured.

CAUTION:
It is mandatory that the variable has the same type used by others clients, just to guarantee the correct data exchange.

CONNECTION_CONFIG.uiLastWillPayloadSize

- **Type:** UINT
- **Default:** 0

This input is the message size, on bytes. This parameters must be at most the SIZEOF(variable), which is the memory size allocated for the variable. If a value lower than the variable size is configured, the message will be cut (e.g. a string with the message ’I am a message’, if the size is 4 bytes, the message will be received as ’I am’).

CAUTION:
Never use a size greater than the variable, this will send trash information to the broker, or worse, cause an access violation. It is strongly recommended to use always the SIZEOF function. When the client will send a STRING variable, use the LEN function instead of SIZEOF, which will count the number of characters in the STRING, avoiding the publishing of undesired data.

CONNECTION_CONFIG.eLastWillMessageQoS

- **Type:** MQTT_QOS
- **Default:** MQTT_QOS_0

This input is the Quality of Service (QoS) of the Last Will message. The QoS represents the message consistence. This value goes from 0 to 2, where 2 is the best quality but slower, and 0 the most unreliable but faster.

ATTENTION:
The QoS 0 is delivered at most once time, in other words, if the broker is not available during the message publish, the message will be lost. The QoS 1 should deliver the message at least once, what means the sender stores the message, while waits the receiver acknowledge, resending the message if it is not received. The QoS 2 should deliver the message exactly one time, which is done by a four-part handshake.

CONNECTION_CONFIG.bLastWillRetain

- **Type:** BOOL
- **Default:** FALSE

This parameter indicates to the broker that the Last Will message must be retained. A retain message will be received as any other message, however, when a new client subscribe to the topic (or after a disconnection), the message will be automatically send to it, marked as a retained message.

CONNECTION_CONFIG.bEnableTLS

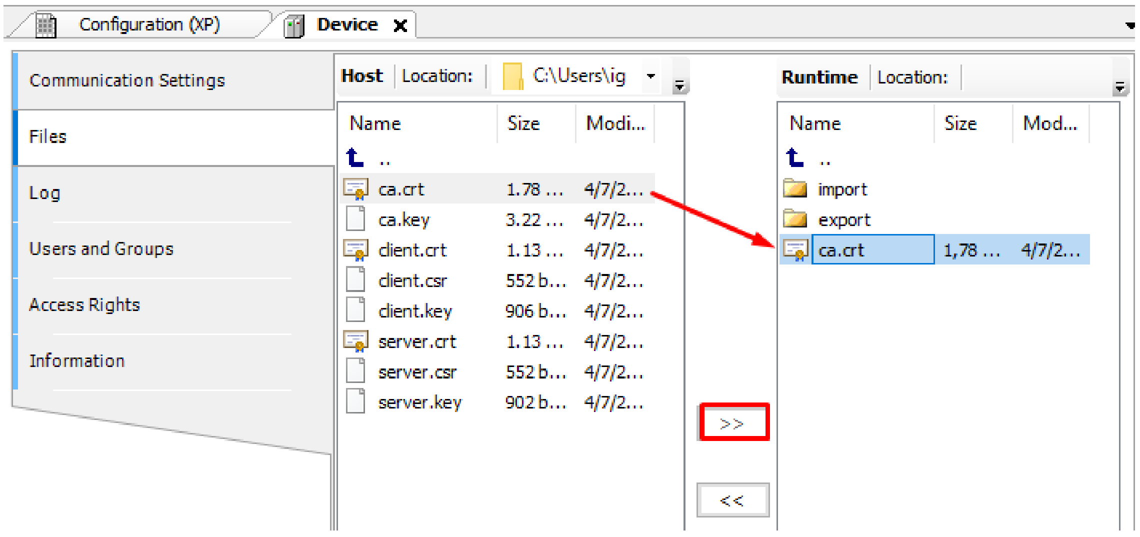
- **Type:** BOOL
- **Default:** FALSE

The MQTT client function block allows the use of TLS encryption (version 1.0, 1.1 and 1.2). This security method allows a very secure connection. This will be required by the broker, which must implement the TLS encryption (server certificate) with a Certificate Authority (CA) file. This parameter must match the configuration of the broker or the connection will not be established. If the broker uses TLS, must be TRUE, otherwise, keep the default. The TLS is based on certificates and keys, where the main certificate (the CA) is used to generate the server certificate (and the client, if desirable). An OpenSSL tool is able to create all certificates that you need to implement a TLS connection, since the CA file until the server and client certificates and keys. If you are using a broker on the cloud, you must download the CA file used on their TLS connection.

CONNECTION_CONFIG.sCertFilename

- **Type:** STRING(255)
- **Default:** ”

This input corresponds to the Certificate Authority (CA) file name (up to 255 characters). The CA must be pass to the PLC with the Device ->Files tool on the Master Tool (see the image below). It must be added to the folder "cert" on the controller’s internal memory. This CA file is the same used by the server. This input must receive the name of the file imported via Device ->Files (e.g. ’ca.crt’, in the image). If a wrong name is entered in this field or there is no file in the PLC, the connection will be denied and the client will be disconnected.



Host:

Name	Size	Modi...
..		
ca.crt	1.78 ...	4/7/2...
ca.key	3.22 ...	4/7/2...
client.crt	1.13 ...	4/7/2...
client.csr	552 b...	4/7/2...
client.key	906 b...	4/7/2...
server.crt	1.13 ...	4/7/2...
server.csr	552 b...	4/7/2...
server.key	902 b...	4/7/2...

Runtime:

Name	Size	Mod...
..		
import		
export		
ca.crt	1,78 ...	4/7/2...

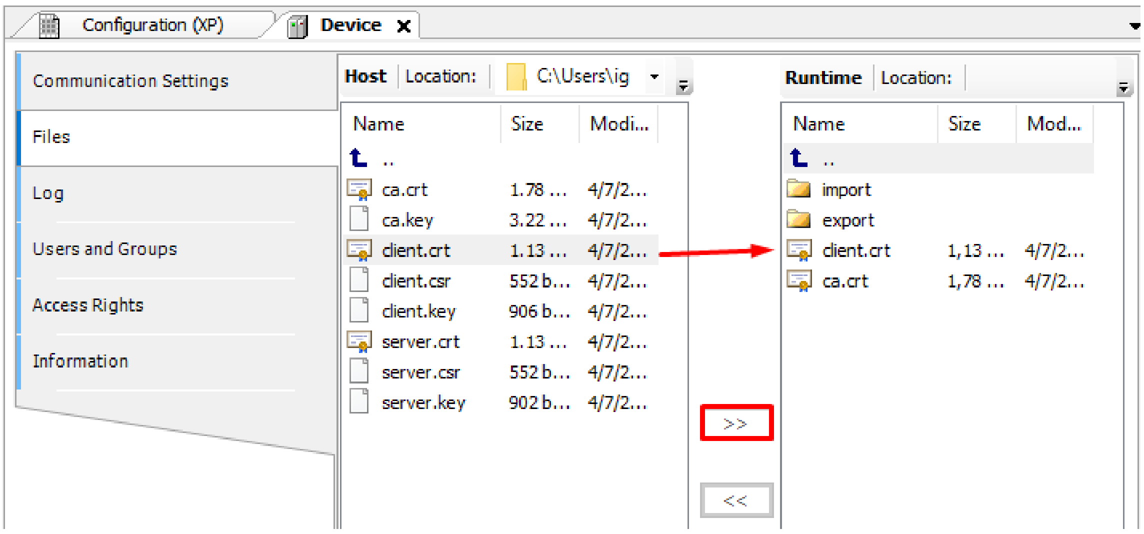
Figure 4: Sending TLS CA file to the PLC

ATTENTION:
A TLS Certificate Authority (CA) must not require a password to access it. All controllers can use a TLS version up to 1.2 since the firmware version 1.9.2.0, except the CPUs NX3010, NX3020 and NX3030 that only accept TLS version 1.0. The TLS version is selected automatically by the Function Block.

CONNECTION_CONFIG.sClientCert

- **Type:** STRING(255)
- **Default:** ”

This input is where you fill the name of the client certificate, generated from the CA file. To add the file, use the same method explained on the sCertFilename. Transfer your client certificate to the controller's memory in the "cert" folder, as shown in the image below. Then, fill this input with the file name used (e.g. 'client.crt'). You must not enter with the client certificate if the broker doesn't request it, this can cause a connection failure. On the other hand, if the broker requires a client certificate, you must enter with the certificate (this input) and the key (the next parameter) with the respective file names that were transferred to the controller.



Name	Size	Modi...
..		
ca.crt	1.78 ...	4/7/2...
ca.key	3.22 ...	4/7/2...
client.crt	1.13 ...	4/7/2...
client.csr	552 b...	4/7/2...
client.key	906 b...	4/7/2...
server.crt	1.13 ...	4/7/2...
server.csr	552 b...	4/7/2...
server.key	902 b...	4/7/2...

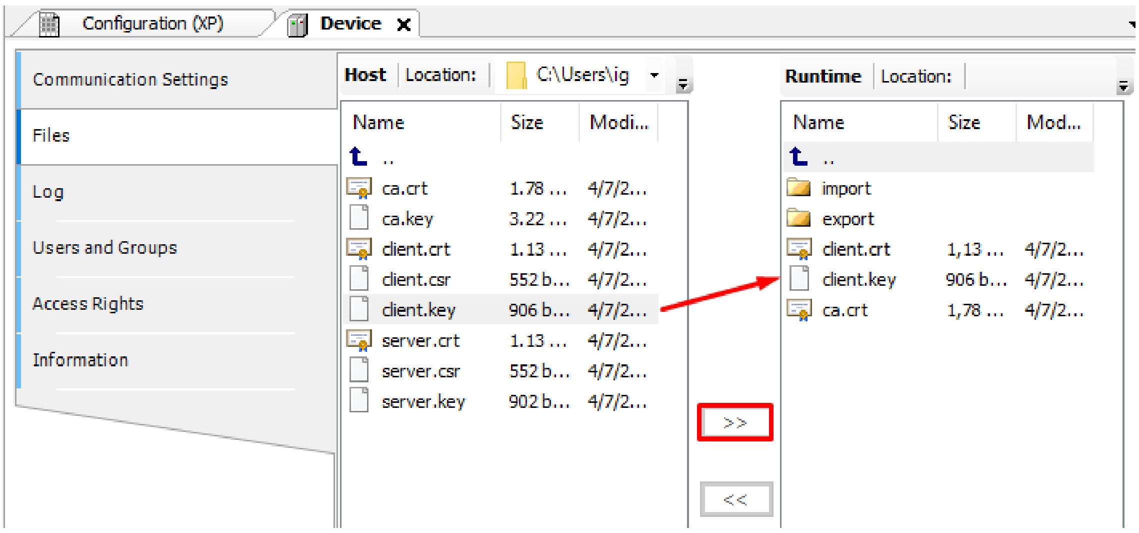
Name	Size	Mod...
..		
import		
export		
client.crt	1,13 ...	4/7/2...
ca.crt	1,78 ...	4/7/2...

Figure 5: Sending TLS client certificate file to the PLC

CONNECTION_CONFIG.sClientKey

- **Type:** STRING(255)
- **Default:** ”

This is where you put the client key name, which is part of the client certificate. Use the same method of the sCertFilename and the sClientCert to add the key to the MQTT Function Block: transfer it to the controller memory ("cert" folder, see the image below) and add its name to this input. If the broker doesn't require a client certificate, you must not fill this parameter or the connection will fail. The same behaviour will occur if a wrong name or the file doesn't exist in the correct folder.



Name	Size	Modi...
..		
ca.crt	1.78 ...	4/7/2...
ca.key	3.22 ...	4/7/2...
client.crt	1.13 ...	4/7/2...
client.csr	552 b...	4/7/2...
client.key	906 b...	4/7/2...
server.crt	1.13 ...	4/7/2...
server.csr	552 b...	4/7/2...
server.key	902 b...	4/7/2...

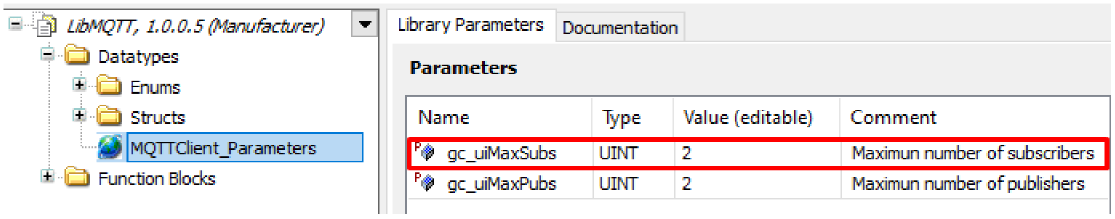
Name	Size	Mod...
..		
import		
export		
client.crt	1,13 ...	4/7/2...
client.key	906 b...	4/7/2...
ca.crt	1,78 ...	4/7/2...

Figure 6: Sending TLS client key file to the PLC

3.1.3. SUBSCRIBE

- **Type:** ARRAY [1..ui_gcMaxSubs] OF MQTT_SUBSCRIBE

This input is an array of structure for configure the topics that the client will subscribe. It is an array to allowed the user to monitor different topics at the same time. The parameter ui_gcMaxSubs is editable (LibMQTT ->MQTTClient_Parameters, see the image below). Remember that once this input is an array, it must be accessed like SUBS[1]. All variables in SUBSCRIBE are used only in the SUBSCRIBING state, described in the Figure 3.



Name	Type	Value (editable)	Comment
gc_uiMaxSubs	UINT	2	Maximun number of subscribers
gc_uiMaxPubs	UINT	2	Maximun number of publishers

Figure 7: Maximum Number of Subscribe Parameter

SUBSCRIBE[n].sSubscribeTopic

- **Type:** STRING(1024)
- **Default:** ”

This input is the specific topic that the client will subscribe (up to 1024 characters). If the topic is empty, the subscription will be ignored for this structure index. This behaviour is independent for other elements in the array.

SUBSCRIBE[n].eSubscribeQoS

- **Type:** MQTT_QOS
- **Default:** MQTT_QOS_0

The Quality of Service (QoS) in the subscription represents the QoS expected for the message that the client will receive. The best QoS is the 2 but is the slowest, while the 0 is the most unreliable but the fastest.

ATTENTION:
The QoS 0 is delivered at most once time, in other words, if the broker is not available during the message publish, the message will be lost. The QoS 1 should deliver the message at least once, what means the sender stores the message, while waits the receiver acknowledge, resending the message if it is not received. The QoS 2 should deliver the message exactly one time, which is done by a four-part handshake.

SUBSCRIBE[n].pbPayloadBuffer

- **Type:** POINTER TO BYTE
- **Default:** NULL

This parameter is the address of the buffer that will receive the newest message (also called payload) sent by the broker. The MQTT protocol allows to send and to receive any type of data (e.g. STRING, DWORD, BYTE...), this is why this input is a pointer. Use the ADR function (e.g. ADR(variable)) to pass the variable address. When the client receives a message, it will be copied to this variable. The RECEIVED_MESSAGES parameter contains the description of the message received in the topic.

> **CAUTION:**
> **It is mandatory that the variable has the same type used by others clients, just to guarantee the correct data exchange.**

SUBSCRIBE[n].uiMaxPayloadSize

- **Type:** UINT
- **Default:** 0

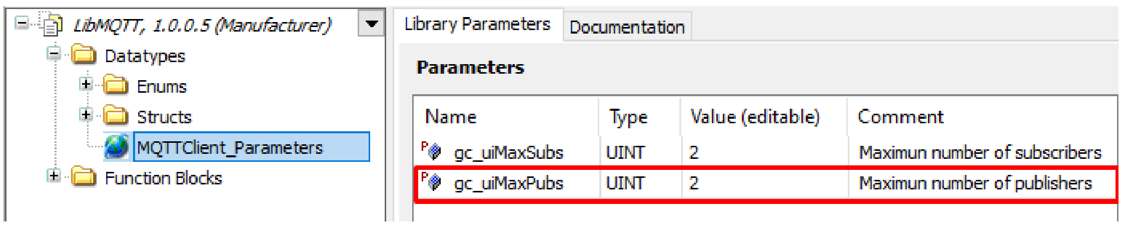
This size will be the maximum size, on bytes, that the message can be received. This information is necessary to avoid a memory invasion by some message bigger than the variable pointed in pbPayloadBuffer. If this size is lower than the size of the message received, the message in the buffer will be cut (e.g. a string with the message 'I am a message', if the size is 4 bytes, the message will be received as 'I am'). On the other hand, it the message size is lower or equal to the size specified in this input, the output uiReceivedPayloadSize will contain the original size of the message. Always use the SIZEOF function in this input, once it must be at most all the memory space available for the variable passed.

> **CAUTION:**
> **Never use a size greater than the variable, this will send trash information to the broker, or worse, cause an access violation. It is strongly recommended to use always the SIZEOF function. When the client will send a STRING variable, use the LEN function instead of SIZEOF, which will count the number of characters in the STRING, avoiding the publishing of undesired data.**

3.1.4. PUBLISH

- **Type:** ARRAY [1..ui_gcMaxPubs] OF MQTT_PUBLISH

This array of structure is responsible to configure the messages that will be published by the client. The parameter ui_gcMaxPubs is editable (LibMQTT ->MQTTClient_Parameters, see the image below). This input must be set as an array, like PUBS[1]. The PUBLISH is used during the READY state (Figure 3), which is the client loop.



Name	Type	Value (editable)	Comment
gc_uiMaxSubs	UINT	2	Maximun number of subscribers
gc_uiMaxPubs	UINT	2	Maximun number of publishers

Figure 8: Maximum Number of Publish Parameter

PUBLISH[n].bEnablePublish

- **Type:** BOOL
- **Default:** FALSE

This boolean will enable the function block to publish the configured message. It is developed as rising edge to avoid a burst of messages to the broker. Therefore, if it is TRUE, the message will be published once.

> **CAUTION:**
> **The MQTT protocol is processed on the background (low priority). For this reason, the publish may not occur immediately when it is enabled.**

PUBLISH[n].sPublishTopic

- **Type:** STRING(1024)
- **Default:** ''

It is the topic where the message will be published (up to 1024 characters). If this string is empty and the enable is TRUE, the publish will be ignored.

PUBLISH[n].pbPublishPayload

- **Type:** POINTER TO BYTE
- **Default:** NULL

This is the message (also called payload) that will be published in the specified topic. Enter with the address of the desired variable, using the ADR function. For the MQTT, the data published can be of any type (e.g. STRING, DWORD, BYTE...), this is why a pointer is necessary. If this input is NULL, the publish will be ignored.

CAUTION:
It is mandatory that the variable has the same type used by others clients, just to guarantee the correct data exchange.

PUBLISH[n].uiPublishPayloadSize

- **Type:** UINT
- **Default:** 0

It is the size of the message that will be published. This information complements the pbPublishPayload, which is a pointer to the variable that will be sent. The size will inform the MQTT client function block the amount of bytes of the message that will be published. Therefore, if this size is lower than the variable, the message will be cut (e.g. a string with the message 'I am a message', if the size is 4 bytes, the message will be received as 'I am'). If this size is keep as zero, an empty message will be publish on the topic.

CAUTION:
Never use a size greater than the variable, this will send trash information to the broker, or worse, cause an access violation. It is strongly recommended to use always the SIZEOF function. When the client will send a STRING variable, use the LEN function instead of SIZEOF, which will count the number of characters in the STRING, avoiding the publishing of undesired data.

PUBLISH[n].ePublishMessageQoS

- **Type:** MQTT_QOS
- **Default:** MQTT_QOS_0

This parameter is the Quality of Service (QoS) of the message that will be published. The QoS specifies the consistence of the message that the client will send. The best QoS is the 2 but is the slowest, while the QoS 0 is the most unreliable but the faster.

ATTENTION:
The QoS 0 is delivered at most once time, in other words, if the broker is not available during the message publish, the message will be lost. The QoS 1 should deliver the message at least once, what means the sender stores the message, while waits the receiver acknowledge, resending the message if it is not received. The QoS 2 should deliver the message exactly one time, which is done by a four-part handshake.

PUBLISH[n].bPublishRetain

- **Type:** BOOL
- **Default:** FALSE

Enable this input to retain the published in the broker. A retained message is sent when the client subscribes to the topic. When the client is already connected, the retained message is processed like a normal message.

3.2. Outputs Parameters

All outputs of the MQTT client function block are shown in the list below, which are described on the following sections:

- CONNECTED
- STATE
- ERROR
- ERROR_CODE
- PUBLISH_STATUS
- LAST_TOPIC
- RECEIVED_MESSAGES

3.2.1. CONNECTED

- **Type:** BOOL

This output shows when the client successfully connects to the broker. When the client loses the connection, this boolean goes to FALSE, being necessary to disable and enable the function block to reconnect.

3.2.2. STATE

- **Type:** MQTT_STATES

It is the current state of the function block. The states are IDLE, CONNECTING, SUBSCRIBING, READY and DISCONNECTED as shown in the Figure 3.

3.2.3. ERROR

- **Type:** BOOL

This boolean indicates when the MQTT client found an error. The error not necessarily disconnects the client, so it is possible to have both ERROR and CONNECTED on TRUE. The error is described by ERROR_CODE.

3.2.4. ERROR_CODE

- **Type:** MQTT_ERR_CODE

This output describes the error state of the MQTT client. See the enumerate MQTT_ERR_CODE to all possibilities. When the client is communicating without problems, the ERRO_CODE will shown MQTT_NO_ERROR.

3.2.5. PUBLISH_STATUS

- **Type:** ARRAY [1..gc_uiMaxPubs] OF BOOL

This boolean indicates when the message was published by the MQTT client function block for each element of the PUBLISH structure array. It goes to TRUE when bEnablePublish is TRUE and the publish was made successfully or not. When bEnablePublish returns to FALSE, this output also will be FALSE. This is useful to know if the message was published or not.

3.2.6. LAST_TOPIC

- **Type:** STRING(1024)

This string (up to 1024 characters) is the last topic received by the client. Only subscribed topics can appear in this field.

3.2.7. RECEIVED_MESSAGES

- **Type:** ARRAY [1..gc_uiMaxSubs] OF MQTT_RECEIVED

This array of structure relates all topics in SUBSCRIBE. In this structure, some informations are shown about the last message received for a specific topic. The RECEIVED_MESSAGES is updated during the READY state, see Figure 3.

RECEIVED_MESSAGES[n].udiMessageCounter

- **Type:** UDINT

This output shows how many times a message was received for the specific topic. When a new message is received, the counter is increased by one unit.

RECEIVED_MESSAGES[n].sReceivedTopic

- **Type:** STRING(1024)

It is the topic that client is subscribed. This output is loaded during the SUBSCRIBING state (see Figure 3) with the information entered in the SUBSCRIBE input. Therefore, the topic appears even when none message was received.

RECEIVED_MESSAGES[n].uiReceivedPayloadSize

- **Type:** UINT

This parameter shows the original size of the received message. However, the message itself is delimited by uiMaxPayloadSize in the SUBSCRIBE structure. It is useful to know that some client publish some message greater than expected.

RECEIVED_MESSAGES[n].eReceivedMessageQoS

- **Type:** MQTT_QOS

This output shows the Quality of Service (QoS) of the last message received for the specific topic. It will only receive messages that respect the QoS configuration entered in eSubscribeQoS (parameter of SUBSCRIBE structure).

ATTENTION:
The QoS 0 is delivered at most once time, in other words, if the broker is not available during the message publish, the message will be lost. The QoS 1 should deliver the message at least once, what means the sender stores the message, while waits the receiver acknowledge, resending the message if it is not received. The QoS 2 should deliver the message exactly one time, which is done by a four-part handshake.

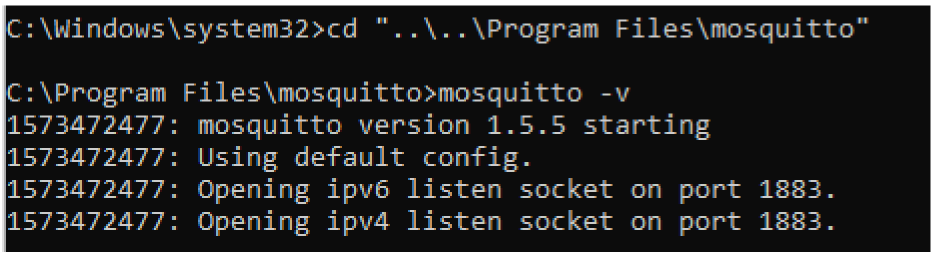
RECEIVED_MESSAGES[n].bReceivedRetain

- **Type:** BOOL

It is TRUE if the message received is retained. In other words, if some client publish a message as retain and other client subscribe to the topic that the message was published, this client will receive the message as soon as it subscribes. For this reason, the message will be marked as retained.

4. Application Example

This chapter will show an example of application developed on Continuous Function Chart (CFC), using a local broker with the Windows Mosquitto. First of all, download the broker in Eclipse Download. After that, install it and open a Windows prompt as administrator. Then, go to the Mosquitto root path and execute the command *mosquitto -v*, see the image below.

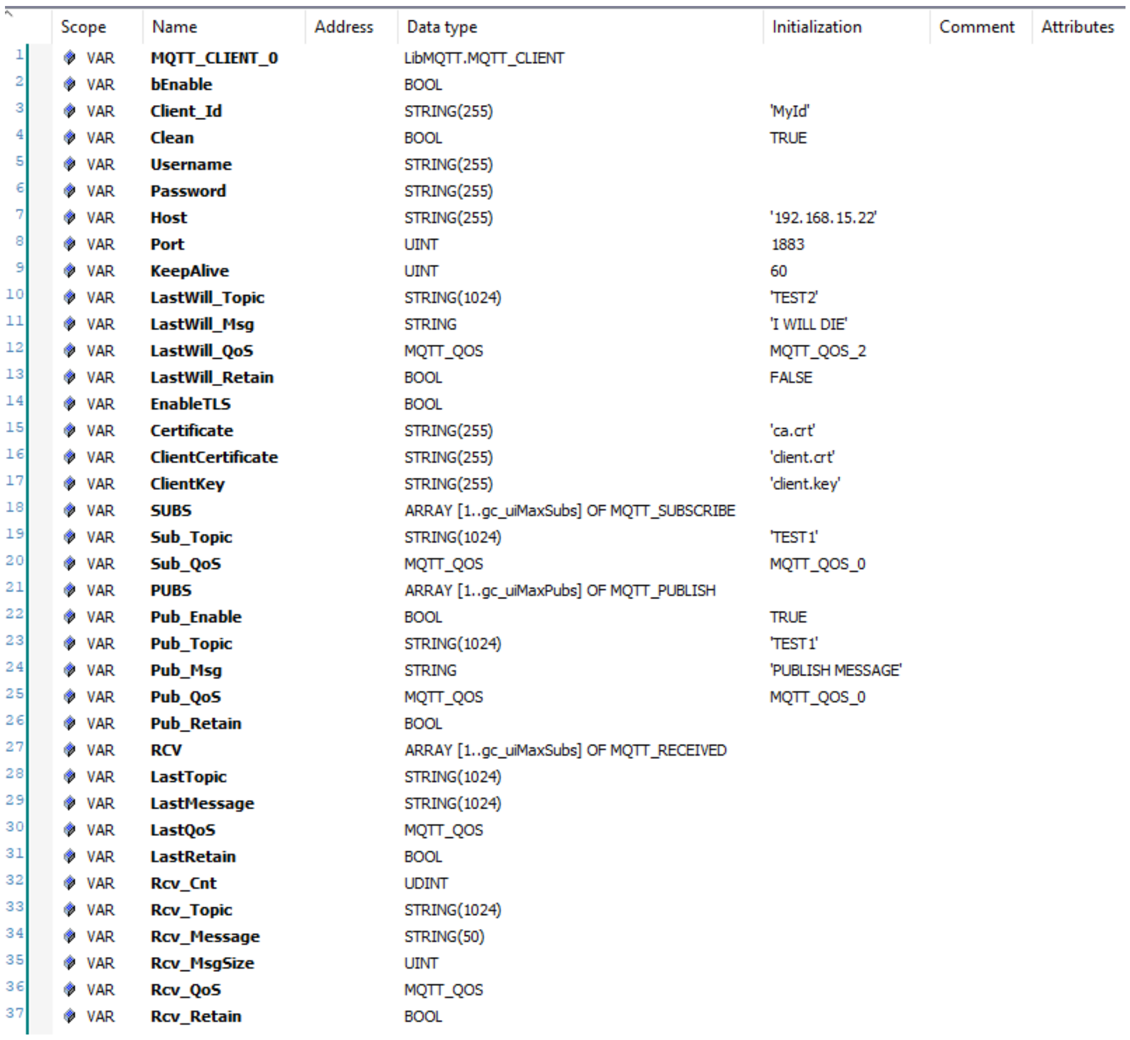


```
C:\Windows\system32>cd "..\..\Program Files\mosquitto"

C:\Program Files\mosquitto>mosquitto -v
1573472477: mosquitto version 1.5.5 starting
1573472477: Using default config.
1573472477: Opening ipv6 listen socket on port 1883.
1573472477: Opening ipv4 listen socket on port 1883.
```

Figure 9: Mosquitto Broker - Prompt

The command will use the default configuration for the broker. If you desired to change any settings, edit the file *mosquitto.conf* in the Mosquitto root path and use the command as *mosquitto -c mosquitto.conf -v*. At this point, the broker is online. So, in the application, we only need to enter the computer IP as hostname. The images below shows the function block configuration.



Scope	Name	Address	Data type	Initialization	Comment	Attributes
VAR	**MQTT_CLIENT_0**		LibMQTT.MQTT_CLIENT			
VAR	**bEnable**		BOOL			
VAR	**Client_Id**		STRING(255)	'MyId'		
VAR	**Clean**		BOOL	TRUE		
VAR	**Username**		STRING(255)			
VAR	**Password**		STRING(255)			
VAR	**Host**		STRING(255)	'192.168.15.22'		
VAR	**Port**		UINT	1883		
VAR	**KeepAlive**		UINT	60		
VAR	**LastWill_Topic**		STRING(1024)	'TEST2'		
VAR	**LastWill_Msg**		STRING	'I WILL DIE'		
VAR	**LastWill_QoS**		MQTT_QOS	MQTT_QOS_2		
VAR	**LastWill_Retain**		BOOL	FALSE		
VAR	**EnableTLS**		BOOL			
VAR	**Certificate**		STRING(255)	'ca.crt'		
VAR	**ClientCertificate**		STRING(255)	'client.crt'		
VAR	**ClientKey**		STRING(255)	'client.key'		
VAR	**SUBS**		ARRAY [1..gc_uiMaxSubs] OF MQTT_SUBSCRIBE			
VAR	**Sub_Topic**		STRING(1024)	'TEST1'		
VAR	**Sub_QoS**		MQTT_QOS	MQTT_QOS_0		
VAR	**PUBS**		ARRAY [1..gc_uiMaxPubs] OF MQTT_PUBLISH			
VAR	**Pub_Enable**		BOOL	TRUE		
VAR	**Pub_Topic**		STRING(1024)	'TEST1'		
VAR	**Pub_Msg**		STRING	'PUBLISH MESSAGE'		
VAR	**Pub_QoS**		MQTT_QOS	MQTT_QOS_0		
VAR	**Pub_Retain**		BOOL			
VAR	**RCV**		ARRAY [1..gc_uiMaxSubs] OF MQTT_RECEIVED			
VAR	**LastTopic**		STRING(1024)			
VAR	**LastMessage**		STRING(1024)			
VAR	**LastQoS**		MQTT_QOS			
VAR	**LastRetain**		BOOL			
VAR	**Rcv_Cnt**		UDINT			
VAR	**Rcv_Topic**		STRING(1024)			
VAR	**Rcv_Message**		STRING(50)			
VAR	**Rcv_MsgSize**		UINT			
VAR	**Rcv_QoS**		MQTT_QOS			
VAR	**Rcv_Retain**		BOOL			

Figure 10: MQTT Client Example - Variables

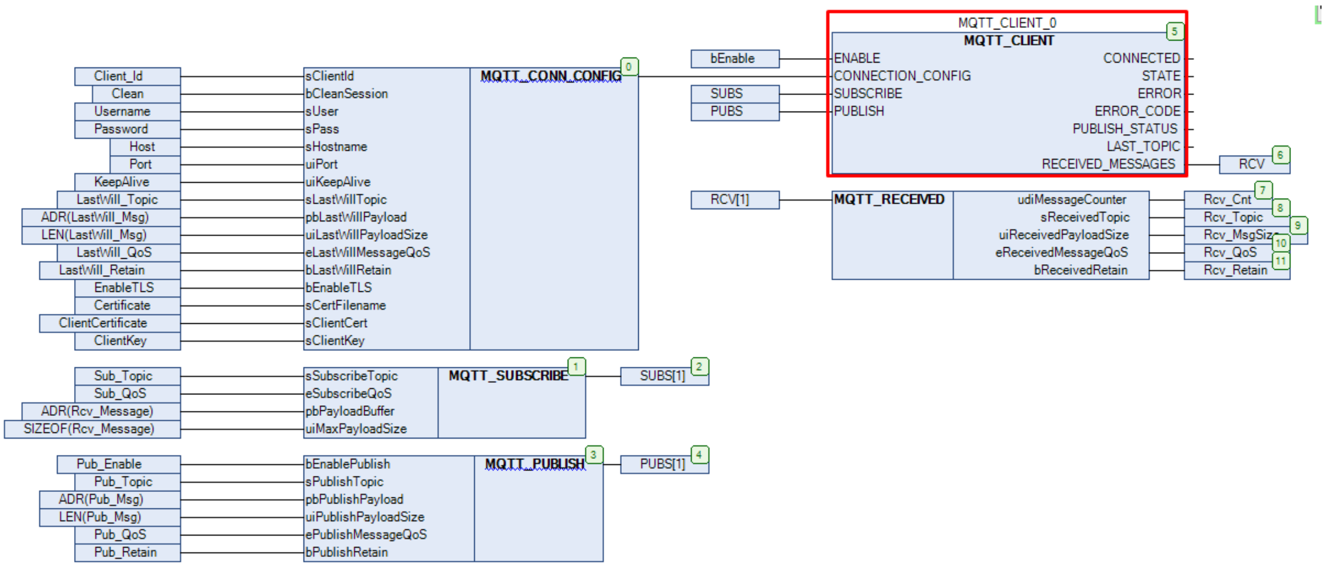
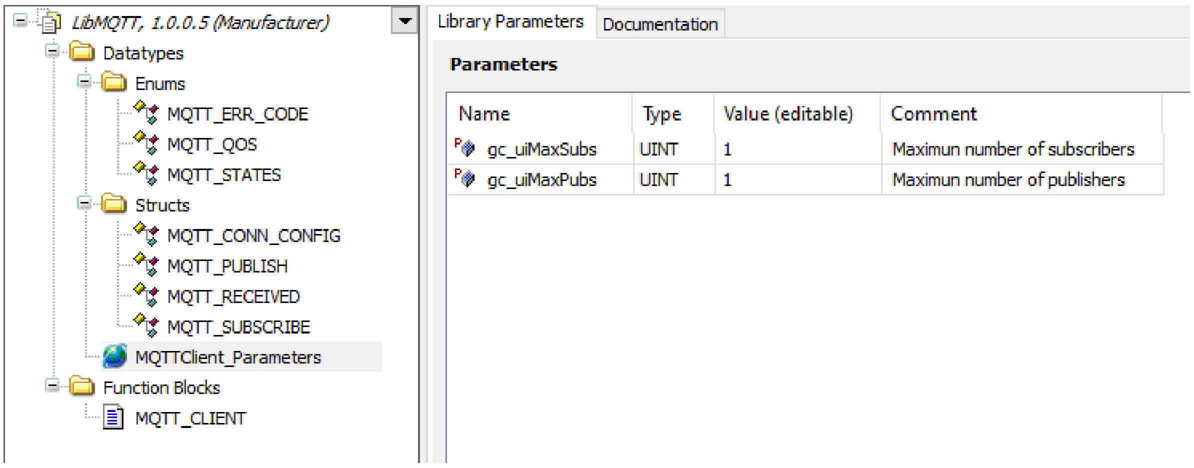


Figure 11: MQTT Client Example - Code

The function block is highlighted by the red rectangle. All others boxes are compounds and selectors used to access the function block structures (CONNECTION_CONFIG, SUBSCRIBE, PUBLISH and RECEIVED_MESSAGES). It is possible to notice that all array variables are using the index (e.g. SUBS[1]), even when ui_gcMaxSubs or ui_gcMaxPubs is one, see the image below.

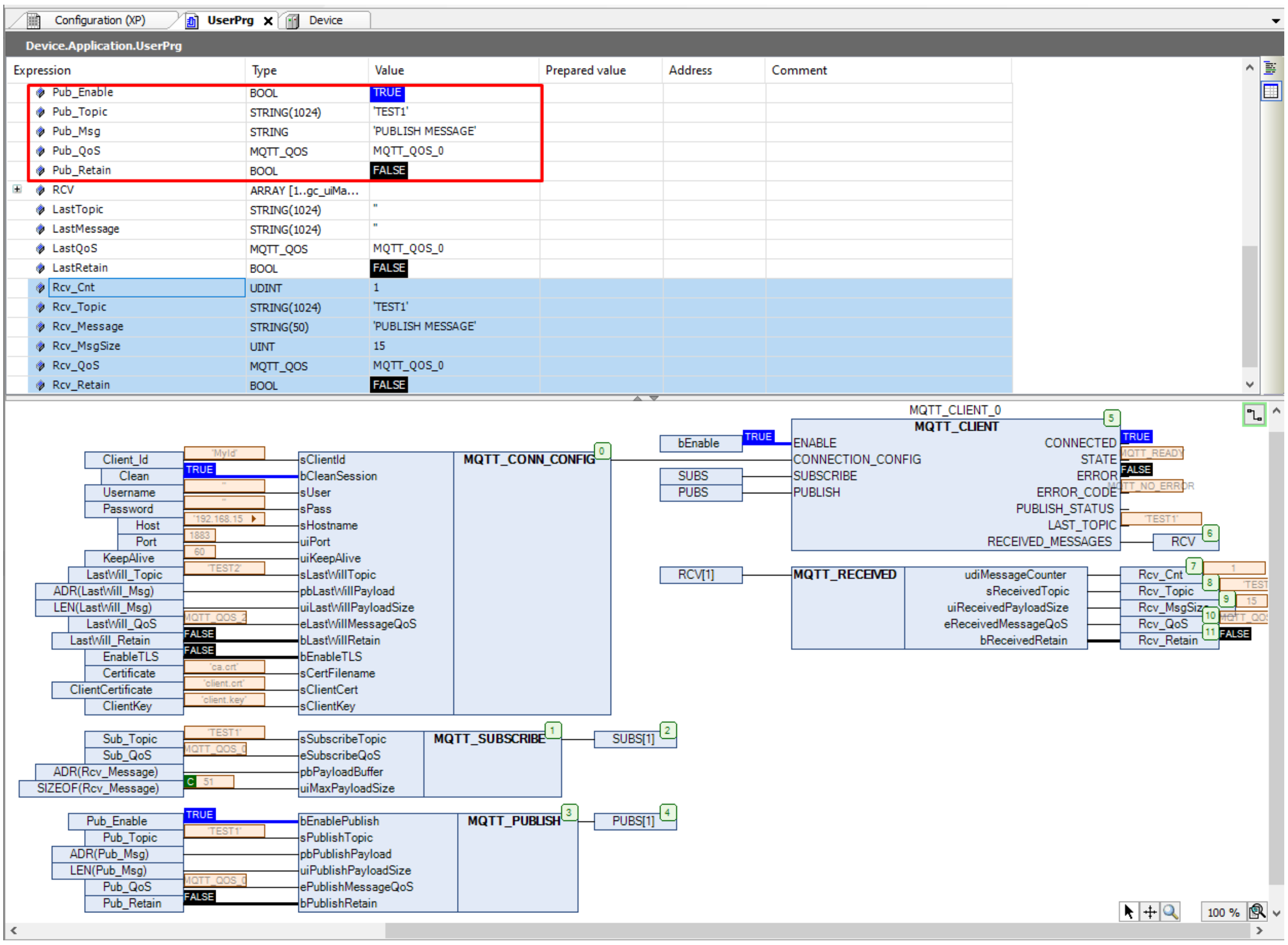


Parameters

Name	Type	Value (editable)	Comment
gc_uiMaxSubs	UINT	1	Maximun number of subscribers
gc_uiMaxPubs	UINT	1	Maximun number of publishers

Figure 12: MQTT Client Example - MQTTClient_Parameters

When everything is properly configured, it is time to download the application and run it. This application will connect to the local broker (e.g. ’192.168.16.106’) with the ID ’MyID’, and will subscribe to the topic ’TEST1’. Also, it will publish the string ’PUBLISH MESSAGE’ to the same topic. This will occur when *bEnable* is TRUE. Note that the same message published will be received on *Rcv_Message*. The prompt will show the message exchange. Finally, when the *bEnable* returns to FALSE, the client will disconnect and the broker will send the Last Will message ’I WILL DIE’ to the topic ’TEST2’. The images below shows the behaviour described.



Expression	Type	Value	Prepared value	Address	Comment
Pub_Enable	BOOL	TRUE			
Pub_Topic	STRING(1024)	'TEST1'			
Pub_Msg	STRING	'PUBLISH MESSAGE'			
Pub_QoS	MQTT_QOS	MQTT_QOS_0			
Pub_Retain	BOOL	FALSE			
RCV	ARRAY [1..gc_uiMa...				
LastTopic	STRING(1024)	''			
LastMessage	STRING(1024)	''			
LastQoS	MQTT_QOS	MQTT_QOS_0			
LastRetain	BOOL	FALSE			
Rcv_Cnt	UDINT	1			
Rcv_Topic	STRING(1024)	'TEST1'			
Rcv_Message	STRING(50)	'PUBLISH MESSAGE'			
Rcv_MsgSize	UINT	15			
Rcv_QoS	MQTT_QOS	MQTT_QOS_0			
Rcv_Retain	BOOL	FALSE			

Figure 13: MQTT Client Example - Publish and Subscribe

The red rectangle shows the message published, while the blue highlight indicates the received message. Note that both are the same. If you desired to see another example, see the Knowledge Base of Altus for an application based on Structured Text (ST) or Ladder Diagram (LD).