

Libraries

MQTT Client SL

JSON Utilities SL

PLC500, PLC500ED, PLC500MC
PLC410

Application Note



Application Note

PLC410, PLC500, PLC500ED, PLC500MC

Document: 10012146404

Revision: 00

Publication Date: 08/2024

SUMMARY OF REVISIONS

The information below describes the reviews made in this manual.

Version	Revision	Description
1.3.0	R00	First Edition.

CONTENTS

1	INTRODUCTION	1-1
1.1	ABBREVIATIONS AND DEFINITIONS	1-1
1.2	IMPORTANT NOTICE ABOUT CYBER SECURITY AND COMMUNICATIONS	1-2
1.3	RECOMMENDATIONS	1-2
1.4	INSTALLING LIBRARIES	1-3
1.5	THE MQTT PROTOCOL	1-4
2	MQTT CLIENT SL	2-1
2.1	WILDCARDS: #, + ...	2-2
2.2	MQTTCLIENT	2-2
2.3	MQTTPUBLISH	2-3
2.4	MQTTSUBSCRIBE	2-3
2.5	PUBLISHING AND SUBSCRIBING WITHOUT ENCRYPTION	2-5
2.6	PUBLISHING AND SUBSCRIBING WITH ENCRYPTION	2-8
3	JSON UTILITIES SL	3-1
3.1	INDEX OF ELEMENTS	3-3
3.2	CREATING AND WRITING A JSON	3-4
3.3	READING A JSON	3-7
3.4	SEARCHING FOR ELEMENTS, KEYS AND VALUES	3-9
4	MQTT CLIENT + JSON UTILITIES	4-1
4.1	CREATING AND PUBLISHING A JSON	4-1
4.2	SUBSCRIBING AND MANIPULATING A JSON	4-4
4.3	PUBLICATION AND SUBSCRIPTION	4-6

1 INTRODUCTION

This Application Note assists with the implementation of the **MQTT** protocol in WEG PLCs, models PLC410, PLC500, PLC500ED and PLC500MC. Please note that the data provided may change slightly due to the continuous development and updating of products and libraries.

It introduces the **MQTT Client SL** and **JSON Utilities SL** libraries, part of the **CODESYS IIoT Libraries SL**, where the main function blocks and methods that contribute to the development of IoT applications stand out. Additionally, practical examples are provided to demonstrate the capabilities of these libraries.

For more information regarding hardware, interfaces and communication protocols, refer to the **User manual** of the respective product, available at www.weg.net. For further details about libraries, function blocks and methods, see the **Codesys Online Help**, available at help.codesys.com.

**ATTENTION!**

This application note is intended for professionals trained in industrial networks. The installation and configuration of the devices must be conducted in accordance with the manufacturer's manual.

1.1 ABBREVIATIONS AND DEFINITIONS

Broker: Server that manages the receipt of messages sent by publishing clients (publishers), sending them to subscribed clients (subscribers), through the MQTT protocol.

MQTT Client: Any device or application that uses the MQTT protocol to connect to an MQTT broker. There are two main types of MQTT clients: publishers and subscribers.

Codesys: Programming platform that allows developing, configuring and monitoring solutions for industrial automation and system integration.

FB: *Function Block*. A specific function or a set of related functions with the ability to incorporate data and other elements.

IoT: Acronym that refers to technologies that simplify communication and data exchange between devices and the cloud, as well as between the devices themselves (*Internet of Things*).

IIoT: Acronym that refers to the application of IoT technologies in the industry. IIoT involves the interconnection of industrial devices, machines and systems via the Internet to collect, exchange and analyze data in order to improve efficiency, automation and monitoring in industrial operations.

IIoT Libraries SL: Set of Codesys libraries developed for IoT applications. IIoT involves the use of IoT technologies in industrial environments to enhance automation, monitoring and data exchange.

JSON: JSON (*JavaScript Object Notation*) is a lightweight data exchange format that is easy for humans to read and write, and easy for machines to interpret and generate. Its basic structure is made up of key and value pairs.

JSON Utilities SL: Specific Codesys library for reading and writing JSON type messages, as well as searching for attributes, values, and child and parent elements.

MQTT: Transport protocol that uses the publish/subscribe topology to transfer lightweight messages between devices.

MQTT Client SL: Specific Codesys library to handle the sending and receiving of messages via MQTT.

Payload: MQTT message content. It can contain several objects, attributes, values etc.

Publisher: Device or application that sends messages to a specific topic on the MQTT broker.

INTRODUCTION

QoS: Parameter used to determine the service quality level in the exchange of messages using the MQTT protocol (*Quality of Service*).

ST: *Structured text*. Programming language defined by IEC 61131-3. Provides a structured, procedural approach for developing control logic in industrial systems.

Subscriber: Device or application that connects to the broker and expresses interest in receiving messages from a specific topic.

TLS: TLS (*Transport Layer Security*) is a security protocol that guarantees the privacy and integrity of data transmitted over the internet. It is commonly used to encrypt communication between a client and a server to protect against man-in-the-middle and data manipulation attacks.

Topic: Way to categorize messages in MQTT. Publishers send messages to specific topics, and subscribers receive messages from topics to which they are subscribed.

1.2 IMPORTANT NOTICE ABOUT CYBER SECURITY AND COMMUNICATIONS

WEG PLCs, models PLC410, PLC500, PLC500ED and PLC500MC, can connect and exchange information through networks and communication protocols. Although they have been designed and subjected to testing to ensure correct operation with other automation systems using the protocols mentioned in this manual, it is essential that the customer understand the responsibilities associated with information and cyber security when using this device.

Consequently, it is the exclusive obligation of the customer to adopt in-depth defense strategies and implement policies and measures to ensure the security of the system as a whole, including with regard to communications sent and received by the equipment. Among such measures, we can point out the installation of firewalls, antivirus and malware protection applications, data encryption, authentication control and physical user access.

WEG and its affiliates take no liability for damages or losses arising from cyber security breaches, including, but not limited to, unauthorized access, intrusion, information or data leak and/or theft, denial-of-service attacks or any other form of security breach. Using this product under conditions for which it was not specifically designed is not recommended and may result in damage to the product, the network and the automation system. Thus, it is essential that the customer understand that the external intervention by third-party software applications, such as sniffers or applications with similar actions, has the potential to cause interruptions or restrictions in the functionality of the device.

1.3 RECOMMENDATIONS

At Codesys, the following precautions are recommended when implementing IoT applications:

- Use specific tasks for publishing and subscribing to MQTT payloads, when possible, and with low priority (20 - 31) in order to avoid interference in critical tasks. Note: the highest priority task is 0 (zero).
- Avoid operating critical loads through IoT platforms due to the possibility of high latency, variation in delay and dependence on a stable internet connection.
- The maximum frequency of payload publications must be based on the application needs and the capacity of the MQTT server.

1.4 INSTALLING LIBRARIES

The MQTT Client SL and JSON Utilities SL libraries, part of the **CODESYS IIoT Libraries SL**, are installed by default on the **WEG package** from version **1.3.x**. However, if necessary, follow the steps for installing these libraries.

To install the IIoT Libraries SL, open Codesys and access **Tools > CODESYS Installer**. In the new opened window, click **Browse** and search for “IIoT”. Select **IIoT Libraries SL** and click **Install**. When installing new libraries through the Codesys Installer, the Codesys software must be previously closed.

Figure 1.1 shows the **CODESYS installer** window, from where the CODESYS IIoT Libraries SL can be installed.

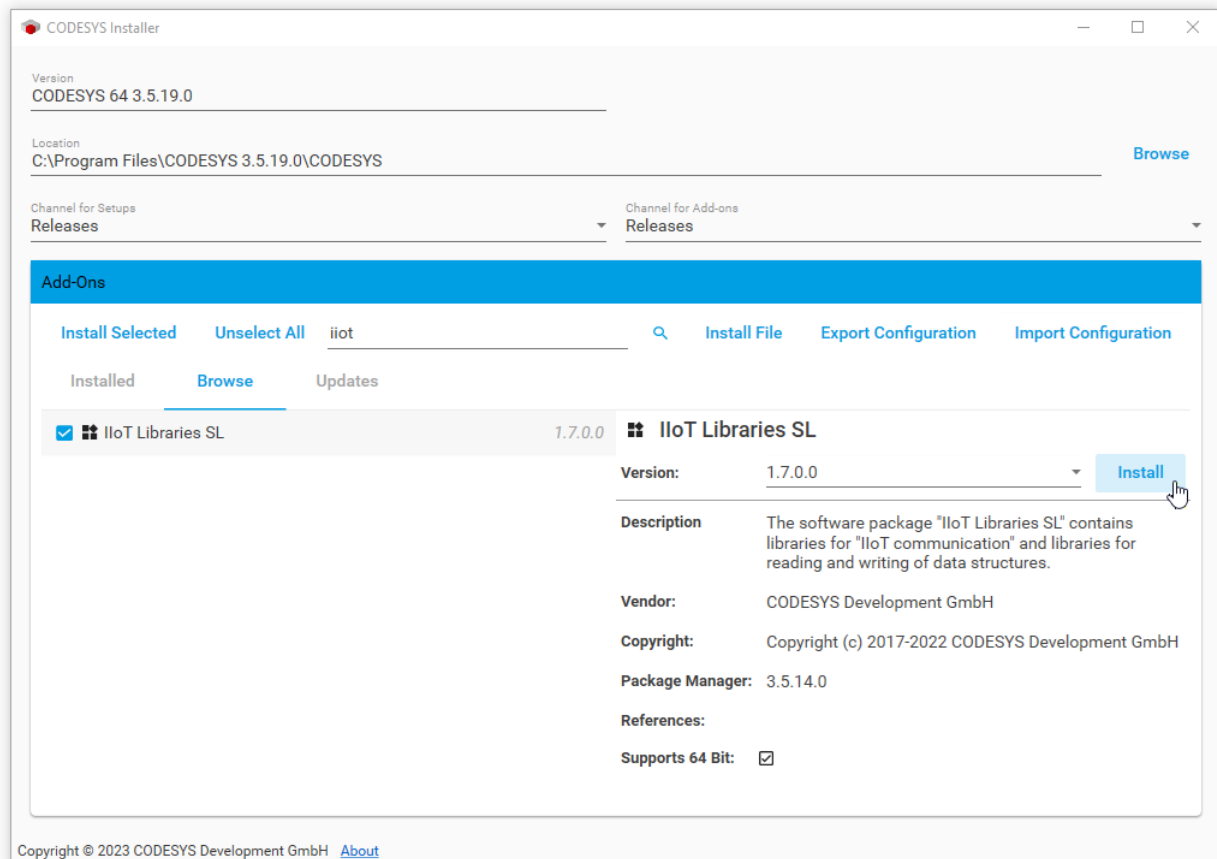


Figure 1.1: Installation of the CODESYS IIoT Libraries SL.

After installation, you will be able to add the **MQTT Client SL** and **JSON Utilities SL** libraries, as well as the other libraries present in IIoT Libraries SL, to Codesys applications.

1.5 THE MQTT PROTOCOL

MQTT (*Message Queuing Telemetry Transport*) is a lightweight and efficient communication protocol widely used in IoT applications (*Internet of Things*). Its characteristic is the *publisher/subscriber* message exchange pattern.

Messages, commonly called *payloads*, are published based on topics. Topics correspond to strings that represent paths with hierarchies. For example, in *factory1/lab2/temperature*, *factory1* is the top level, *lab2* is a sublevel, and *monitoring* is the characteristic that describes the last part of the topic. Whenever a device is subscribed to a topic, it will receive payloads published on that same topic.

Communication between devices occurs through an MQTT broker, which manages publications and subscriptions (for example, the [Mosquitto](#)).

Figure 1.2 shows an example of an IoT application using the MQTT protocol.

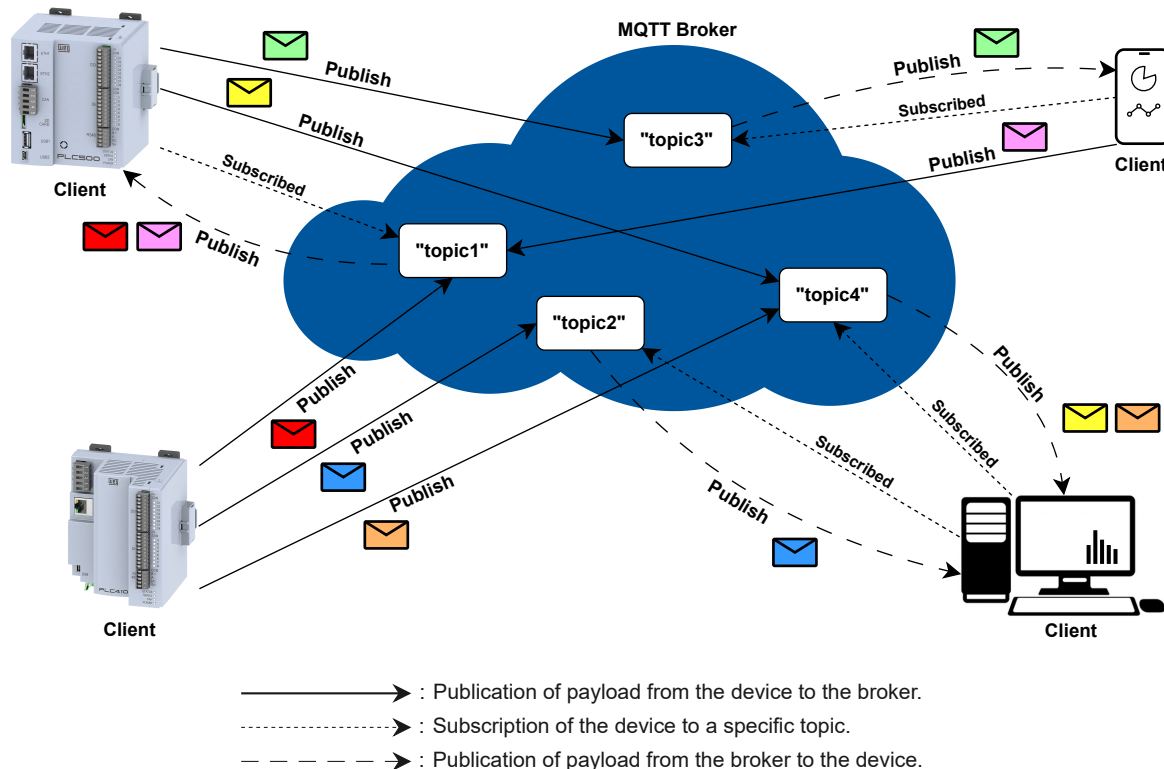


Figure 1.2: Schematic of an IoT application using the MQTT protocol.

Figure 1.2 contains four topics being used for publication, called *topic1*, *topic2*, *topic3* e *topic4*. The network is made up of a PLC500, a PLC410, a computer and a smartphone. The PLC500 publishes payloads in topics 3 and 4 and subscribes to topic 1. The PLC410 publishes payloads in topics 1, 2 and 4. The computer subscribes to topics 2 and 4. The smartphone publishes in topic 1 and subscribes to topic 3.

The data published in MQTT messages can vary widely depending on the application and the specific context of use. However, some general categories of commonly published data include: sensor readings, device states, production parameters, events, alerts, notifications, process control etc.



NOTE!

In many cases, the MQTT payload is formatted using JSON (*JavaScript Object Notation*). JSON syntax is lightweight and easy to read, making it easy to transport structured data.

2 MQTT CLIENT SL

The **MQTT Client SL** library allows connecting a Codesys controller to an MQTT broker. Then messages can be published and subscribed based on topics.

The message format is not fixed, which means that a JSON string or any data structure can be transmitted. The MQTT Client SL library has the following function blocks:

- **MQTTClient**: Establish the MQTT link with the broker. Every application using the MQTT Client SL library requires at least one MQTTClient FB. Only one instance of this block is used to connect to a single broker.
- **MQTTPublish**: Used to publish messages to the broker topics. One instance can be used for each topic to be published.
- **MQTTSubscribe**: Used to subscribe to the broker topics. One instance can be used for each topic to be subscribed.

Figure 2.1 shows the MQTT Client SL library on Codesys.

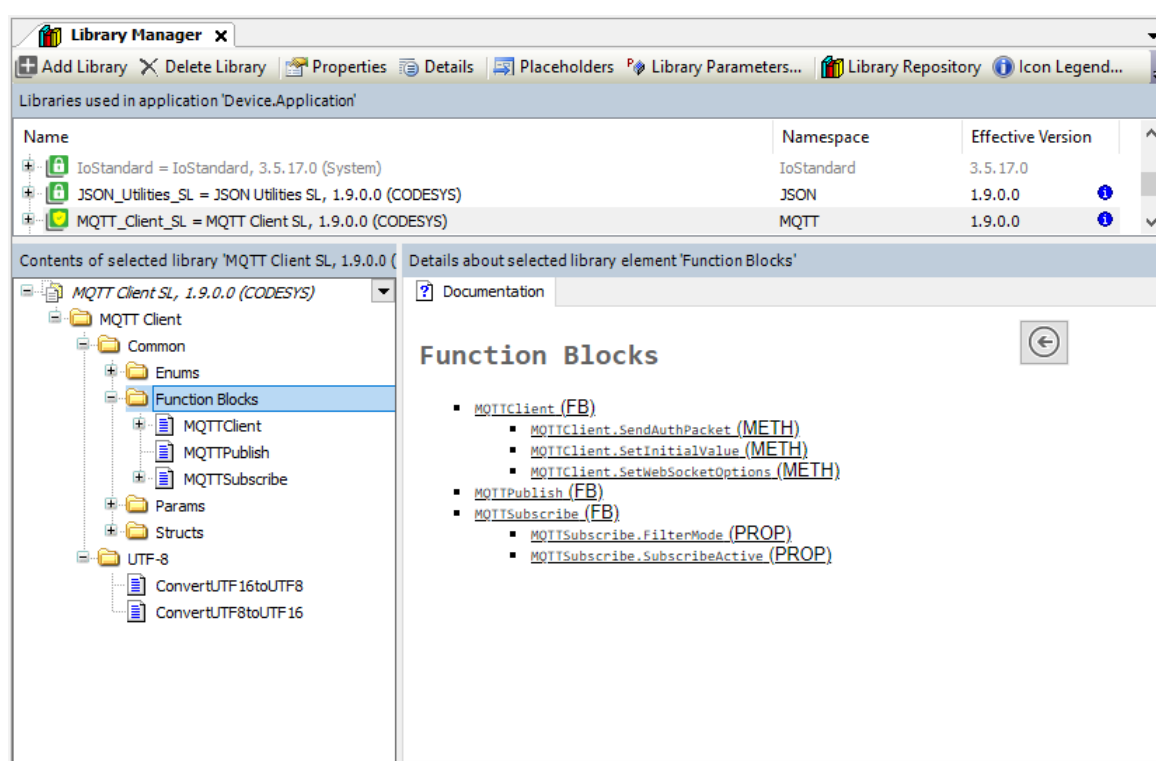


Figure 2.1: MQTT Client SL library on Codesys.

Supported Functions:

- Message publication and subscription based on MQTT V3.1.1 and MQTT V5.0
- TLS encryption
- Client certificates
- Quality of Service: 0, 1 and 2 (QoS0, QoS1, QoS2)
- Maximum packet size and payload size can be configured via a parameter list
- Multitasking support (MQTTClient, MQTTPublish and MQTTSubscribe can be run in different tasks)
- Last will messages (QoS0, QoS1, QoS2)
- Wildcards: Characters used to subscribe to multiple topics simultaneously



NOTE!

QoS levels determine the degree of guarantee in message delivery in MQTT. In QoS 0, delivery occurs at most once, without confirmation. In QoS 1, there is a guarantee of at least one delivery, with confirmation from the server. Finally, QoS 2 ensures delivery exactly once through a more complex *handshake* process between client and server.

The choice between these levels depends on the need for delivery reliability versus communication efficiency.

2.1 WILDCARDS: #, +

- # (multi-level): Used only at the end of topics. It allows all topic messages that begin with the pattern before the wildcard to be received. Example: **factory1/lab1/#**
 - factory1/lab1/temperature ✓
 - factory1/lab1/humidity ✓
 - factory1/lab2/temperature ✗
- + (single-level): It allows all topic messages that contain any string in place of the wildcard to be received. Example: **factory1/lab1/+/temperature**
 - factory1/lab1/room1/temperature ✓
 - factory1/lab1/room1/humidity ✗
 - factory1/lab2/room1/temperature ✗
 - factory1/lab1/room2/temperature ✓

2.2 MQTTCLIENT

Below are the main variables of the MQTTClient FB:

- **Inputs:**
 - **xEnable:** *True* = Activates the function block. *False* = Aborts/resets the block operation.
 - **uiPort:** Port used by the MQTT broker. Usually, the default port for unencrypted communications is **1883**, while port **8883** is generally used for encryption. However, you should check the characteristics of the specific broker.
 - **sHostname:** MQTT broker address.
 - **xUseTLS:** *True* = Connection encrypted using TLS. *False* = Connection not encrypted.
 - **xCleanSession:** Flag bit used to control the session state lifecycle. *True* = means that a new session will be created on the connection, and the session will be automatically dropped when the client disconnects. If *False*, it means it will try to reuse the previous session when connecting. If there is no matching session, a new session is created, which will always exist after the client disconnects.
 - **uiKeepAlive:** Time, in seconds, that the client and server agree to keep the connection open without any message exchanges. Initial value: 5 seconds. (optional)
 - **wsUsername:** Username (optional).
 - **wsPassword:** User password (optional).
 - **sClientId:** User identifier code (optional).

■ Outputs:

- **xDone**: *True* = Operation completed.
- **xBusy**: *True* = Operation running.
- **xError**: *True* = Operation error.
- **eMQTTError**: MQTT error type.
- **xConnectedToBroker**: *True* = Connected to the broker.



NOTE!

The main difference between encrypted (TLS) and unencrypted ports is the security of the transmitted data. When the MQTT operates over an unencrypted port, data is transmitted in simple text, which means it can be read by anyone with access to the network between the MQTT client and server. This poses a security risk, especially when sensitive information is being transmitted.

2.3 MQTTPUBLISH

Below are the main variables of the MQTTPublish FB:

■ Inputs:

- **xExecute**: *True* = Enables the function block.
- **udiTimeOut**: Maximum running time. An error is generated after this time.
- **eQoS**: QoS level of the message for publication.
- **pbPayload**: Pointer to the string containing the payload to be published. You can use the FBs of the **JSON Utilities** library to create a payload in JSON format.
- **udiPayloadSize**: Size of the payload to be published.
- **mqttClient**: Instance of the MQTTClient FB.
- **wsTopicName**: Name of the Topic for publication.

■ Outputs:

- **xDone**: *True* = Operation completed.
- **xBusy**: *True* = Operation running.
- **xError**: *True* = Operation error.
- **eMQTTError**: MQTT error type.

2.4 MQTTSUBSCRIBE

Below are the main variables of the MQTTSubscribe FB:

■ Inputs:

- **xEnable**: *True* = Enables the function block.
- **eSubscribeQoS**: Message QoS level for subscription.
- **pbPayload**: Pointer to the string that will store the received payload.
- **udiMaxPayloadSize**: Maximum length of the received string.
- **wsTopicFilter**: Topic filter. You can use a fixed topic or in conjunction with the characters # and +.

- **mqttClient**: Instance of the MQTTClient FB.
- **udiTimeOut**: Maximum running time. An error is generated after this time.

■ **Outputs:**

- **xDone**: *True* = Operation completed.
- **xBusy**: *True* = Operation running.
- **xError**: *True* = Operation error.
- **eMQTTErr**: MQTT error type.
- **xReceived**: *True* = Message received.
- **xSubscribeActive**: *True* = Subscribe is waiting for a payload on the specified topic.

2.5 PUBLISHING AND SUBSCRIBING WITHOUT ENCRYPTION

This application, called **MQTT_simple**, is a simple example of using function blocks from the MQTT library. It connects to the MQTT broker, publishes a string containing the value of a variable that represents the temperature of a device and, in the meantime, subscribes to the same topic, that is, it will receive the payload as soon as it is published. The communication port used is **1883**, no encryption and no authentication.

In detail, the **MQTTClient** FB connects the device to the test.mosquitto.org broker, which is a public server used for testing and experimenting with the MQTT protocol. The **RS** and **TON** FBs are used to disable the FB in case of an error from the MQTTClient, wait for one second, and then enable it again.

To publish payloads, the **MQTTPublish** FB is used within a simple state machine. In state 0, a counter is incremented every cycle, and after 50 cycles, it transitions to state 1. In state 1, the **uiTemperature** variable, of type unsigned integer, is incremented by one unit and then transitions to state 2. In state 2, the payload for publication is created. The payload consists of the string 'simple MQTT message: temperature = ' followed by the value of the **uiTemperature** variable.

In state 3, the publication itself is carried out. The **MQTTPublish** FB is enabled only if the MQTTClient has its output **xConnectedToBroker** in the high state. The topic defined for publication is "**weg/drivesbt/plc**". Note that the input for the payload is **pbPayload**, which is a pointer to the desired string. Also, the **mqttClient** input must define the instance of the MQTTClient FB used, which in this case is **MQTTClient_0**.

As soon as one of the **xDonePub** or **xErrorPub** outputs are in the high state, the MQTTPublish_0 FB is disabled, and the state machine returns to state 0, restarting the publishing process.

Outside of the state machine, we have the **MQTTSubscribe** FB. It also depends on the MQTTClient FB connection through the **xConnectedToBroker** variable. When enabled, this FB waits for the publication of payloads in any subtopic under "**weg/drivesbt/**", due to the use of the # character at the end of the topic. While the FB is active, the **xSubscribeActive** variable will be in high state. When a payload is received, the **xReceived** variable becomes high, allowing manipulation of the **sPayloadSub** string according to the application.



NOTE!

In this example, all the code of the **MQTTClient**, **MQTTPublisher** e **MQTTSubscribe** is performed in a single task in order to facilitate reproduction by the user. However, as highlighted in the recommendations in this note, you should try to use specific tasks with compatible priorities for each function block so as not to interfere with the main program.



NOTE!

To use the **MQTTClient**, **MQTTPublisher** e **MQTTSubscribe** function blocks, don't forget to add the **MQTT Client SL** library to the Codesys application.

Figure 2.2 shows the variable declaration, and Figure 2.3 shows the code in ST for the MQTT_simple application.

MQTT_simple - Declaration

```
PROGRAM MQTT_simple
VAR
    // Functions Blocks
    MQTTClient_0: MQTT.MQTTClient;
    MQTTPublish_0: MQTT.MQTTPublish;
    MQTTSubscribe_0: MQTT.MQTTSubscribe;

    // MQTTClient
    sHostName : STRING := 'test.mosquitto.org';
    uiPort : UINT := 1883;
    xConnectedToBroker, xDoneClient, xErrorClient : BOOL;
    eMQTTErrorClient : MQTT.MQTT_ERROR;
    uiQoS : MQTT.MQTT_QOS := 1;

    // MQTTPublish
    sPayloadPub : STRING(255);
    pbPayloadPub : POINTER TO BYTE := ADR(sPayloadPub);
    wsTopicPub : WSTRING(1024) := "weg/drivesbt/plc";
    udiPayloadSizePub : UDINT;
    xDonePub, xErrorPub : BOOL;
    eMQTTErrorPub : MQTT.MQTT_ERROR;

    // MQTTSubscribe
    sPayloadSub : STRING(255);
    pbPayloadSub : POINTER TO BYTE := ADR(sPayloadSub);
    wsTopicSub: WSTRING(1024) := "weg/drivesbt/#";
    udiMaxPayloadSize : UDINT := 1024;
    udiPayloadSizeSub : UDINT;
    xDoneSub, xErrorSub, xReceived, xSubscribeActive : BOOL;
    eMQTTErrorSub : MQTT.MQTT_ERROR;

    // PRG variables
    uiTemperature : UINT := 0;
    STATE, uiCount : UINT := 0;
    RS_0 : RS;
    TON_0 : TON;
END_VAR
```

Figure 2.2: Declaration of MQTT_simple variables.

MQTT_simple - Structured text (ST)

```

// Check MQTTClient error
RS_0(SET := NOT xErrorClient, RESET1 := xErrorClient); // Set Reset
TON_0(IN := RS_0.Q1, PT := T#1S); // Timer

// FB MQTTClient
MQTTClient_0(
  xEnable := TON_0.Q, sHostname := sHostName, uiPort := uiPort, uiKeepAlive := 60,
  xError => xErrorClient, eMQTTErr => eMQTTErrClient, xConnectedToBroker => xConnectedToBroker, xDone => xDoneClient);

// Publish State Machine
CASE STATE OF

0: // Increment uiCount
  uiCount := uiCount + 1;
  IF uiCount > 50 THEN uiCount := 0; STATE := 1; END_IF;

1: // Increment of uiTemperature variable
  uiTemperature := uiTemperature + 1;
  IF uiTemperature > 1000 THEN uiTemperature := 0; END_IF;
  STATE := 2;

2: // Creation of the Payload and size
  sPayloadPub := CONCAT('simple MQTT message: temperature = ',UINT_TO_STRING(uiTemperature));
  udiPayloadSizePub := DINT_TO_UDINT(StrLenA(pbPayloadPub));
  STATE := 3;

3: // Publish
  MQTTPublish_0(
    xExecute := ((NOT xErrorPub) AND (NOT xDonePub) AND (xConnectedToBroker)),
    wsTopicName := wsTopicPub, eQoS := uiQoS, mqttClient := MQTTClient_0,
    pbPayload := pbPayloadPub, udiPayloadSize := udiPayloadSizePub,
    xDone => xDonePub, xError => xErrorPub, eMQTTErr => eMQTTErrPub);

// Reset publisher
IF xDonePub = 1 OR xErrorPub = 1 THEN MQTTPublish_0.xExecute := 0; STATE := 0; END_IF

END_CASE;

// FB MQTTSubscribe
MQTTSubscribe_0(
  xEnable := ((NOT xErrorSub) AND (NOT xDoneSub) AND (xConnectedToBroker)),
  wsTopicFilter := wsTopicSub, eSubscribeQoS := uiQoS, mqttClient := MQTTClient_0,
  pbPayload := pbPayloadSub, udiMaxPayloadSize := udiMaxPayloadSize,
  xDone => xDoneSub, xError => xErrorSub, eMQTTErr => eMQTTErrSub,
  xReceived => xReceived, udiPayloadSize => udiPayloadSizeSub, xSubscribeActive => xSubscribeActive);

```

Figure 2.3: MQTT_simple program in structured text.

2.6 PUBLISHING AND SUBSCRIBING WITH ENCRYPTION

This section presents the **MQTT_encryption** examples, in which the previous application is modified so that communication between clients and the MQTT broker is encrypted, ensuring that the data involved cannot be intercepted or manipulated by unauthorized third parties. This communication uses the **TLS** (*Transport Layer Security*) protocol, which involves the use of digital certificates to authenticate the parties involved in the communication and encrypt the transmitted data.

Like the previous application, in this example, the **MQTTClient** FB connects the device to the test.mosquitto.org broker and publishes a string containing the value of the `uiTemperature` variable and, in the meantime, subscribes to the same topic. In this case, the **8883** port is used, which is the default port for secure MQTT communications. Furthermore, the **xUseTLS** input of the **MQTTClient** FB must be configured with the Boolean value **True**. The rest of the code is exactly the same as the previous example.



NOTE!

Usually, the default port for unencrypted communications is **1883**, while port **8883** is generally used for encryption. However, you should check the characteristics and ports available on the specific MQTT broker.

Figure 2.4 shows the variable declaration, and Figure 2.5 shows the ST code for the **MQTT_encryption** application.

MQTT_encryption - Declaration

```
PROGRAM MQTT_encryption
VAR
    // Functions Blocks
    MQTTClient_0: MQTT.MQTTClient;
    MQTTPublish_0: MQTT.MQTTPublish;
    MQTTSubscribe_0: MQTT.MQTTSubscribe;

    // MQTTClient
    sHostName : STRING := 'test.mosquitto.org';
    uiPort : UINT := 8883; // Encrypted port
    xUseTLS : BOOL := 1; // Using TLS
    xConnectedToBroker, xDoneClient, xErrorClient : BOOL;
    eMQTTErrorClient : MQTT.MQTT_ERROR;
    uiQoS : MQTT.MQTT_QOS := 1;

    // MQTTPublish
    sPayloadPub : STRING(255);
    pbPayloadPub : POINTER TO BYTE := ADR(sPayloadPub);
    wsTopicPub : WSTRING(1024) := "weg/drivesbt/plc";
    udiPayloadSizePub : UDINT;
    xDonePub, xErrorPub : BOOL;
    eMQTTErrorPub : MQTT.MQTT_ERROR;

    // MQTTSubscribe
    sPayloadSub : STRING(255);
    pbPayloadSub : POINTER TO BYTE := ADR(sPayloadSub);
    wsTopicSub: WSTRING(1024) := "weg/drivesbt/#";
    udiMaxPayloadSize : UDINT := 1024;
    udiPayloadSizeSub : UDINT;
    xDoneSub, xErrorSub, xReceived, xSubscribeActive : BOOL;
    eMQTTErrorSub : MQTT.MQTT_ERROR;

    // PRG variables
    STATE, uiCount, uiTemperature : UINT := 0;
    RS_0 : RS;
    TON_0 : TON;
END_VAR
```

Figure 2.4: Declaration of **MQTT_encryption** variables.

MQTT_encryption - Structured text (ST)

```

// Check MQTTClient error
RS_0(SET := NOT xErrorClient, RESET1 := xErrorClient); // Set Reset
TON_0(IN := RS_0.Q1, PT := T#1S); // Timer

// FB MQTTClient
MQTTClient_0(
  xEnable := TON_0.Q, sHostname := sHostName, uiPort := uiPort, uiKeepAlive := 60, xUseTLS := xUseTLS,
  xError => xErrorClient, eMQTTErr => eMQTTErrClient, xConnectedToBroker => xConnectedToBroker, xDone => xDoneClient);

// Publish State Machine
CASE STATE OF

0: // Increment uiCount
  uiCount := uiCount + 1;
  IF uiCount > 50 THEN uiCount := 0; STATE := 1; END_IF;

1: // Increment of uiTemperature variable
  uiTemperature := uiTemperature + 1;
  IF uiTemperature > 1000 THEN uiTemperature := 0; END_IF;
  STATE := 2;

2: // Creation of the Payload and size
  sPayloadPub := CONCAT('simple MQTT message: temperature = ',UINT_TO_STRING(uiTemperature));
  udiPayloadSizePub := DINT_TO_UDINT(StrLenA(pbPayloadPub));
  STATE := 3;

3: // Publish
  MQTTPublish_0(
    xExecute := ((NOT xErrorPub) AND (NOT xDonePub) AND (xConnectedToBroker)),
    wsTopicName := wsTopicPub, eQoS := uiQoS, mqttClient := MQTTClient_0,
    pbPayload := pbPayloadPub, udiPayloadSize := udiPayloadSizePub,
    xDone => xDonePub, xError => xErrorPub, eMQTTErr => eMQTTErrPub);

// Reset publisher
IF xDonePub = 1 OR xErrorPub = 1 THEN MQTTPublish_0.xExecute := 0; STATE := 0; END_IF

END_CASE;

// FB MQTTSubscribe
MQTTSubscribe_0(
  xEnable := ((NOT xErrorSub) AND (NOT xDoneSub) AND (xConnectedToBroker)),
  wsTopicFilter := wsTopicSub, eSubscribeQoS := uiQoS, mqttClient := MQTTClient_0,
  pbPayload := pbPayloadSub, udiMaxPayloadSize := udiMaxPayloadSize,
  xDone => xDoneSub, xError => xErrorSub, eMQTTErr => eMQTTErrSub,
  xReceived => xReceived, udiPayloadSize => udiPayloadSizeSub, xSubscribeActive => xSubscribeActive);

```

Figure 2.5: MQTT_encryption program in structured text.

Before MQTT communication is started, a TLS *handshake* occurs between the client and the server. During this process, the two parties negotiate security parameters, such as the encryption algorithm and key exchange. Next, the server sends a digital certificate to authenticate its identity to the client. All subsequent communication is encrypted using the shared keys established during the *handshake*. The server certificate can be found in the **Security Screen** of Codesys, being initially classified as **Quarantined Certificates**. You can drag the certificate to the **Trusted Certificates** folder in order to ensure for Codesys that the source is reliable.

Figure 2.6 shows the *Security Screen* from Codesys and the digital certificate belonging to the *Mosquitto* broker.

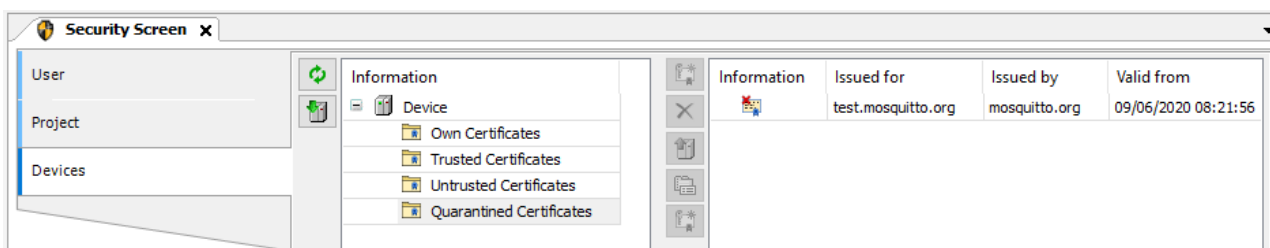


Figure 2.6: Security Screen of Codesys.

3 JSON UTILITIES SL

The JSON format is structured as a collection of key-value pairs, where a key is a string and the value can be a number, a string, a Boolean, an object, an array, or null. These key-value pairs are grouped into objects, which in turn can be nested in other objects or arrays. This hierarchical structure of JSON allows data to be represented in an organized way, making it easier for machines and humans to read and manipulate it.

The **JSON Utilities SL** library, in turn, is used to read and write data in JSON format from controllers using Codesys. In it, data is stored in an array of **JSONData** type structures, where elements are stored in the form of **JSONElements**.

A **JSONElement** has the following basic structure:

- **Value:** The data value is returned as **wsValue** (WSTRING), **xValue** (BOOL), **lrValue** (LREAL) or **liValue** (LINT).
- **eType:** Element type. It can return the following values:

Name	Value	Description
KEY	0	The element is a key stored as WSTRING.
WSTRING_VALUE	1	Element is a WSTRING value.
LINT_VALUE	2	Element is a LINT value.
LREAL_VALUE	3	Element is an LREAL value.
BOOL_VALUE	4	Element is an BOOL value.
NULL	5	Element is NULL.
JSON_ARRAY	6	Element is an ARRAY.
JSON_OBJECT	7	Element is an object.
NONE		No JSON type.

- **diIndex:** Index of the element in the data array. The type is DINT.
- **diParentIndex:** Index of the element that is one hierarchical level above the element in question. The type is DINT.

The JSON Utilities library has the following function blocks:

- **FindFirstValueByKey:** It finds the corresponding value linked to a given key.
- **JSONBuilder:** Simple and quick creation of a JSON file.
- **JSONByteArrayReader:** It reads data from a WSTRING in JSON format and saves it in JSONData format.
- **JSONByteArrayWriter:** It writes the content of a JSONData to a WSTRING.
- **JSONData:** Main function block of the library. This FB contains the JSON data itself and has methods to access, search and configure the data.
- **JSONDataFactory:** Required to allocate memory for creating a JSONData.
- **JSONFileReader:** It reads data from a file in JSON format and saves it in JSONData format.
- **JSONFileWriter:** It writes the content of a JSONData to a file.

Figure 3.1 shows the JSON Utilities SL library on Codesys.

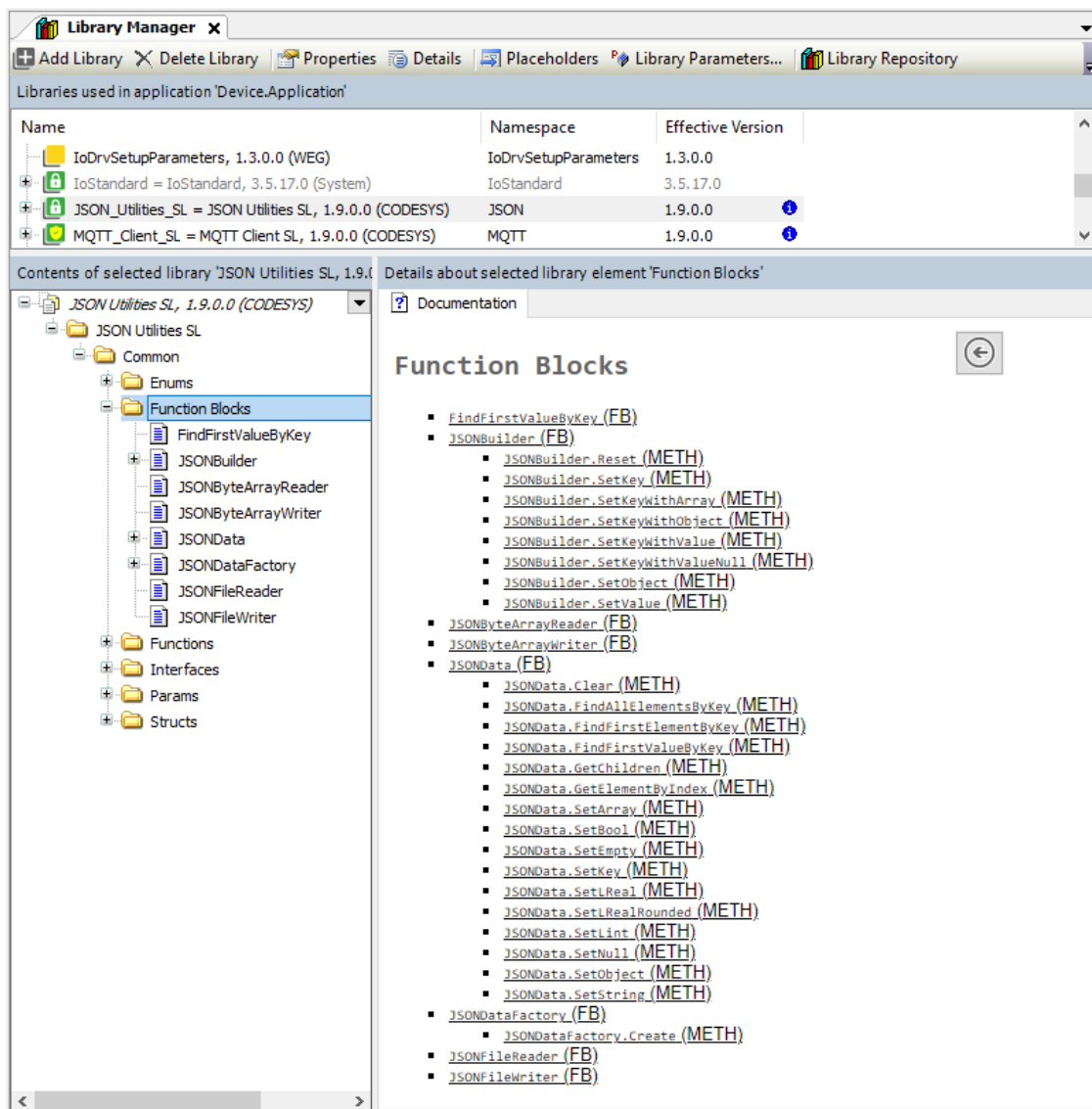


Figure 3.1: JSON Utilities SL library on Codesys.

3.1 INDEX OF ELEMENTS

On Codesys, it is necessary to understand the syntax of how JSON elements are organized in order to manipulate them in the best way. Figure 3.2 shows an example of JSON with the respective Index, Parent Index, Type and Value of each element.

```
{(0)
  "device1"(1): {(2)
    "temperature"(3): 39.2(4),
    "humidity"(5) : 25(6)
  },
  "device2"(7): {(8)
    "motor1"(9): {(10)
      "rotation"(11): "clockwise" (12),
      "fault"(13): 0(14)
    }
  }
  "time"(15): 1705424300(16)
}
```

Index	Parent Index	Type	Value
0	-1	Object	
1	0	Key	device1
2	1	Object	
3	2	Key	temperature
4	3	LREAL	39.2
5	2	Key	humidity
6	5	LINT	25
7	0	Key	device2
8	7	Object	
9	8	Key	motor1
10	9	Object	
11	10	Key	rotation
12	11	STRING	clockwise
13	10	Key	fault
14	13	BOOL	0
15	0	Key	time
16	15	LINT	1705424300

Figure 3.2: Example of JSON with the respective indexes of the elements.

The JSON above illustrates data being published from a device that is acquiring information from two different devices: *device1* and *device2*. *device1* has two keys, *temperature* and *humidity*, which contain the values 39.2 and 25, one of the LREAL type and the other of the LINT type, respectively. *device2*, in turn, contains information relating to the *motor1* key, which has two more internal keys, *rotation* and *fault*, representing the direction of rotation and fault state of this motor. These variables are of the STRING and BOOL types, respectively. Finally, there is the *time* key, containing the Unix time at which the data was published.

The Index of each element is shown in superscript, in parentheses, and it is increased with each new element created. The number -1 is considered the initial Index, or root.

Every element has a Parent Index, which is the index of the element that is one hierarchical level above the element in question. The Parent Index of a value is always its respective key.

The next section shows the ST code to create the JSON presented, detailing the function blocks and the main methods used.

3.2 CREATING AND WRITING A JSON

To create the JSON presented in the previous section, the function blocks **JSONDataFactory**, **JSONData**, and **JSONByteArrayWriter** are used. The **JSONDataFactory** is used only in variable declaration to allocate the necessary memory for creating the **JSONData** structure effectively. Therefore, for each JSON used, a specific pointer of type **POINTER TO JSONData** must be created. For error checking, it is necessary to add the **CAA FB Factory** library, since the output of the **Create** method of **JSONDataFactory** is of type **FBF.ERROR**.

The **JSONData** FB is responsible for creating the objects, keys, and values present in the JSON. There are specific methods in this FB for creating **BOOL**, **LINT**, **LREAL**, **STRING**, **ARRAY**, etc. Additionally, there are methods for searching for values, keys, and elements, which are covered in Section 3.4.

The **JSONByteArrayWriter** FB is used at the end of the code to write the contents of the **JSONData** created into a **WSTRING** variable. In this case, the content of the **pJsonData** pointer will be written to the **wsJsonData** string.

Most of the variables used in this example were declared with constant values. However, in real applications, this data is obtained from sensors, equipment, and other industrial devices, according to the designer's needs.

Although constants were used in most of the code, the **liDateTime** variable represents the current payload creation time in Unix time. It is calculated using the **SysTimeRtcHighResGet** function present in the **SysTimeRtc** library.

The **xStartWrite** variable starts with a high value and ensures that the program runs until the **xDoneWrite** variable also acquires the high value, which happens when the **wsJsonDataWrite** string is successfully written by the **jsonArrayWriter** FB.



NOTE!

To use the function blocks **JSONDataFactory**, **JSONData**, and **JSONByteArrayWriter**, don't forget to add the **JSON Utilities SL** library to the Codesys application.

Figure 3.3 shows the variable declaration, and Figure 3.4 shows the ST code for creating the JSON presented in the previous section.

WriteJsonData - Declaration
<pre> PROGRAM WriteJsonData VAR // Creating of the JSONData factory : JSON.JSONDataFactory; // Allocate memory to the JSONData eDataFactoryError : FBF.ERROR; // Error code: 0: OK, 30103: no more memory pJsonDataWrite : POINTER TO JSON.JSONData := factory.Create(eError => eDataFactoryError); // Function Blocks jsonArrayWriter : JSON.JSONByteArrayWriter; // PRG variables wsJsonDataWrite : WSTRING(1024); xStartWrite : BOOL := 1; xDoneWrite : BOOL := 0; diCounter, diObject1, diObject2, diObject3 : DINT; bFault : BOOL := 0; liDateTime : LINT := 0; lrTemperature : LREAL := 39.2; liHumidity : LINT := 25; Rtc : ULINT; END_VAR </pre>

Figure 3.3: Declaration of variables for the *WriteJsonData* program.

WriteJsonData - Structured text (ST)

```
IF xStartWrite = TRUE AND xDoneWrite = FALSE THEN
```

```
    // First element (object) - Index of the root object is -1.
```

```
    diCounter := 0;
```

```
    pJsonDataWrite^.SetObject(diIndex := diCounter, diParentIndex := -1);
```

```
    // Key inside an object - The parent index of the key is the index of the parent object.
```

```
    diCounter := diCounter + 1;
```

```
    pJsonDataWrite^.SetKey(wsKey := "device1", diParentIndex := 0, diIndex := diCounter);
```

```
    // Object inside "device1" - The parent index of the value is the index of corresponding key.
```

```
    diCounter := diCounter + 1; diObject1 := diCounter;
```

```
    pJsonDataWrite^.SetObject(diIndex := diCounter, diParentIndex := diCounter - 1);
```

```
    // Key inside an object - temperature (LREAL)
```

```
    diCounter := diCounter + 1;
```

```
    pJsonDataWrite^.SetKey(wsKey := "temperature", diParentIndex := diObject1, diIndex := diCounter);
```

```
    diCounter := diCounter + 1;
```

```
    pJsonDataWrite^.SetLReal(lrValue := lrTemperature, diIndex := diCounter, diParentIndex := diCounter - 1);
```

```
    // Key inside an object - humidity (LINT)
```

```
    diCounter := diCounter + 1;
```

```
    pJsonDataWrite^.SetKey(wsKey := "humidity", diParentIndex := diObject1, diIndex := diCounter);
```

```
    diCounter := diCounter + 1;
```

```
    pJsonDataWrite^.SetLInt(liValue := liHumidity, diIndex := diCounter, diParentIndex := diCounter - 1);
```

```
    // Key inside an object - device2
```

```
    diCounter := diCounter + 1;
```

```
    pJsonDataWrite^.SetKey(wsKey := "device2", diParentIndex := 0, diIndex := diCounter);
```

```
    // Object inside device2
```

```
    diCounter := diCounter + 1; diObject2 := diCounter;
```

```
    pJsonDataWrite^.SetObject(diIndex := diCounter, diParentIndex := diCounter - 1);
```

```
    // Key inside an object - motor1
```

```
    diCounter := diCounter + 1;
```

```
    pJsonDataWrite^.SetKey(wsKey := "motor1", diParentIndex := diObject2, diIndex := diCounter);
```

```
    // Object inside motor1
```

```
    diCounter := diCounter + 1; diObject3 := diCounter;
```

```
    pJsonDataWrite^.SetObject(diIndex := diCounter, diParentIndex := diCounter - 1);
```

```
    // Key inside an object - rotation (STRING)
```

```
    diCounter := diCounter + 1;
```

```
    pJsonDataWrite^.SetKey(wsKey := "rotation", diParentIndex := diObject3, diIndex := diCounter);
```

```
    diCounter := diCounter + 1;
```

```
    pJsonDataWrite^.SetString(wsValue := "clockwise", diIndex := diCounter, diParentIndex := diCounter - 1);
```

```
    // Key inside an object - fault (BOOL)
```

```
    diCounter := diCounter + 1;
```

```
    pJsonDataWrite^.SetKey(wsKey := "fault", diParentIndex := diObject3, diIndex := diCounter);
```

```
    diCounter := diCounter + 1;
```

```
    pJsonDataWrite^.SetBool(bValue := bFault, diIndex := diCounter, diParentIndex := diCounter - 1);
```

```
    // Timestamp (use the library: SysTimeRtc)
```

```
    SysTimeRtcHighResGet(pTimestamp := Rtc);
```

```
    liDateTime := ULINT_TO_LINT(Rtc);
```

```
    diCounter := diCounter + 1;
```

```
    pJsonDataWrite^.SetKey(wsKey := "time", diParentIndex := 0, diIndex := diCounter);
```

```
    diCounter := diCounter + 1;
```

```
    pJsonDataWrite^.SetLInt(liValue := liDateTime, diIndex := diCounter, diParentIndex := diCounter - 1);
```

```
    // Write the JSON String. The value will be written to the variable wsJsonDataWrite
```

```
    jsonArrayWriter(xExecute := xStartWrite, pwData := ADR(wsJsonDataWrite), udiSize := SIZEOF(wsJsonDataWrite),
```

```
        jsonData := pJsonDataWrite^, xDone => xDoneWrite);
```

```
    IF xDoneWrite = TRUE THEN xStartWrite := FALSE; END_IF
```

```
END_IF
```

Figure 3.4: WriteJsonData program in structured text.

With the program running, it is possible to check the JSONData formed, analyzing the index, parent index, type and value of each element. Figure 3.5 shows the variable declaration screen of the WriteJsonData program in *online* mode, where the current values of the variables can be checked.

Device.Application.WriteJsonData			
Expression	Type	Value	Comment
factory	JSON.JSONDataFactory		Allocate memory to the JSONData
eDataFactoryError	ERROR	NO_ERROR	Error code: 0: OK, 30103: no more memory
pJsonDataWrite	POINTER TO JSON.JSONData	16#74A77028	
jsonArrayWriter	JSON.JSONByteArrayWriter		Function Blocks
wsJsonDataWrite	WSTRING(1024)	"{\$r\$n \$"device1\$": \$r\$n { \$...	PGR Variables
xStartWrite	BOOL	FALSE	
xDoneWrite	BOOL	TRUE	
diCounter	DINT	16	
diStringLength	DINT	0	
diObject1	DINT	2	
diObject2	DINT	8	
diObject3	DINT	10	
bFault	BOOL	FALSE	
liDateTime	LINT	1709636173850	
lrTemperature	LREAL	39.2	
liHumidity	LINT	25	
Rtc	ULINT	1709636173850	

```

{
  "device1":
  {
    "temperature": 39.2,
    "humidity": 25
  },
  "device2":
  {
    "motor1":
    {
      "rotation": "clockwise",
      "fault": false
    }
  },
  "time": 1709636173750
}

```

Device.Application.WriteJsonData			
Expression	Type	Value	Comment
factory	JSON.JSONDataFactory		Allocate memory to the JSONData
eDataFactoryError	ERROR	NO_ERROR	Error code: 0: OK, 30103: no more memory
pJsonDataWrite	POINTER TO JSON.JSONData	16#74A77028	
pJsonDataWrite^	JSONData		
SUPER^	IInstance		
_aJSONData	ARRAY [0..GParams.g_diMaxElements...		
_aJSONData[0]	JSONElement		
_aJSONData[1]	JSONElement		
value	JSONValue		Value of the element
wsValue	WSTRING(GParams.g_diMaxStringSize)	"device1"	Value as WSTRING
xValue	BOOL	<Value of the expression canno...	Value as BOOL
lrValue	LREAL	1.1126170568611754E-306	Value as LREAL
liValue	LINT	29555379367379044	Value as LINT
eType	JSONTYPE	KEY	Type of the value
diIndex	DINT	1	Index of this element in the data array
diParentIndex	DINT	0	Index of the parent JSONElement, -1 for toplevel elements.
_aJSONData[2]	JSONElement		
_aJSONData[3]	JSONElement		
value	JSONValue		Value of the element
wsValue	WSTRING(GParams.g_diMaxStringSize)	"temperature"	Value as WSTRING
xValue	BOOL	<Value of the expression canno...	Value as BOOL
lrValue	LREAL	1.4241953019640074E-306	Value as LREAL
liValue	LINT	31525665549647988	Value as LINT
eType	JSONTYPE	KEY	Type of the value
diIndex	DINT	3	Index of this element in the data array
diParentIndex	DINT	2	Index of the parent JSONElement, -1 for toplevel elements.
_aJSONData[4]	JSONElement		
_aJSONData[5]	JSONElement		

Figure 3.5: Variable declaration screen for the WriteJsonData program in online mode.

3.3 READING A JSON

In this example, the existence of the **WriteJsonData** program, created in the previous section, is considered. **WriteJsonData** is executed as a function in this example to guarantee the creation of the **wsJsonDataWrite** string, which stores the contents of the **pJsonDataWrite** pointer.

Using the **JSONByteArrayReader** FB, you can read the contents of the **wsJsonDataWrite** string and assign this value to a structure of the **JSONData** type. Note that, initially, the **pJsonDataRead** pointer is declared from the **JSONDataFactory** FB, thus ensuring that there is enough memory to create the **JSONData** structure.

The **xStartRead** variable starts with a high value and ensures that the program runs until the **xDoneRead** variable also acquires the high value, which happens when the **wsJsonDataWrite** string is read successfully by the **jsonArrayReader** FB, storing its content in the **pJsonDataRead** pointer.



NOTE!

To use the **JSONDataFactory**, **JSONData**, and **JSONByteArrayReader** function blocks, don't forget to add the **JSON Utilities SL** library to the Codesys application.

The **StringUtils** library can be added to allow the use of the **StrLenW** function, which calculates the length of a WSTRING.

ReadJsonData - Declaration

```
PROGRAM ReadJsonData
VAR
    // Creating of the JSONData
    factory : JSON.JSONDataFactory; // Allocate memory to the JSONData
    eDataFactoryError : FBF.ERROR; // Error code: 0: OK, 30103: no more memory
    pJsonDataRead : POINTER TO JSON.JSONData := factory.Create(eError => eDataFactoryError);

    // Function Blocks
    jsonArrayReader: JSON.JSONByteArrayReader;

    // PRG variables
    diStringSize : DINT;
    xStartRead : BOOL := 1;
    xDoneRead : BOOL := 0;

END_VAR
```

Figure 3.6: Declaration of variables for the ReadJsonData program.

ReadJsonData - Structured text (ST)

```
// Execute the program only once
IF xStartRead = TRUE AND xDoneRead = FALSE THEN

    // Execute the WriteJsonData function
    WriteJsonData();

    // Read the JSON String. The value will be written to the pointer to JSONData called pJsonDataRead
    jsonArrayReader(xExecute := xStartRead, pwData := ADR(WriteJsonData.wsJsonDataWrite), jsonData := pJsonDataRead^,
        xDone => xDoneRead);

    // Calculate the length of a WSTRING
    diStringSize := StrLenW(ADR(WriteJsonData.wsJsonDataWrite));

    IF xDoneRead = TRUE THEN xStartRead := FALSE; END_IF

END_IF;
```

Figure 3.7: ReadJsonData program in structured text.

With the program running, it is possible to check the JSONData formed, analyzing the index, parent index, type and value of each element. Figure 3.8 shows the variable declaration screen of the ReadJsonData program in *online* mode, where the current values of the variables can be checked.

Device.Application.ReadJsonData			
Expression	Type	Value	Comment
factory	JSON.JSONDataFac...		Allocate memory to the JSONData
eDataFactoryError	ERROR	NO_ERROR	Error code: 0: OK, 30103: no more memory
pJsonDataRead	POINTER TO JSON.J...	16#0000026A68DDF080	
pJsonDataRead^	JSONData		
SUPER^	IInstance		
_aJSONData	ARRAY [0..GParams...		
_aJSONData[0]	JSONElement		
_aJSONData[1]	JSONElement		
value	JSONValue		Value of the element
wsValue	WSTRING(GParams...	"device1"	Value as WSTRING
xValue	BOOL	<Value of the expression cannot be retrie...	Value as BOOL
lrValue	LREAL	1.1126170568611754E-306	Value as LREAL
liValue	LINT	29555379367379044	Value as LINT
eType	JSONTYPE	KEY	Type of the value
diIndex	DINT	1	Index of this element in the data array
diParentIndex	DINT	0	Index of the parent JSONElement, -1 for toplevel elements.
_aJSONData[2]	JSONElement		
_aJSONData[3]	JSONElement		
value	JSONValue		Value of the element
wsValue	WSTRING(GParams...	"temperature"	Value as WSTRING
xValue	BOOL	<Value of the expression cannot be retrie...	Value as BOOL
lrValue	LREAL	1.4241953019640074E-306	Value as LREAL
liValue	LINT	31525665549647988	Value as LINT
eType	JSONTYPE	KEY	Type of the value
diIndex	DINT	3	Index of this element in the data array
diParentIndex	DINT	2	Index of the parent JSONElement, -1 for toplevel elements.
_aJSONData[4]	JSONElement		
_aJSONData[5]	JSONElement		

Figure 3.8: Variable declaration screen for the ReadJsonData program in online mode.

3.4 SEARCHING FOR ELEMENTS, KEYS AND VALUES

The JSON Utilities SL library also provides methods for searching values, keys and elements contained in the JSONData FB. The following methods stand out:

- **FindFirstElementByKey**: Searches for the first element in the data array based on a specific key.
- **FindFirstValueByKey**: Searches for the first value in the data array based on a specific key.
- **GetElementByIndex**: Gets the element based on the specific index.
- **JSONElementToString**: Function that allows you to transform the value of a JSONElement into a string.

The **FindGetJson** implements the methods mentioned above in order to obtain elements, keys and values contained in the **pJsonDataWrite**. To that end, the existence of the **WriteJsonData** program is considered, created previously. WriteJsonData is executed as a function in order to guarantee the creation of the pJsonDataWrite pointer, which contains a structure of the JSONData type.

Figure 3.9 shows the variable declaration, and Figure 3.10 the ST code of the FindGetJson application.

FindGetJson - Declaration
<pre> PROGRAM FindGetJson VAR // jsonElements jsonElement1, jsonElement2, jsonElement3, jsonElement4 : JSON.JSONElement; // jsonElement1 diElement1Type, diElement1Index, diElement1ParIndex : DINT; wsElement1Value : WSTRING(JSON.GParams.g_diMaxStringSize); // jsonElement2 diElement2Type, diElement2Index, diElement2ParIndex : DINT; lrElement2Value : LREAL; // jsonElement3 wsElement3Value : WSTRING(JSON.GParams.g_diMaxStringSize); // jsonElement4 wsElement4Value : WSTRING(JSON.GParams.g_diMaxStringSize); // PRG Variables xDoneFindGet : BOOL := 0; END_VAR </pre>

Figure 3.9: Declaration of FindGetJson program variables.

```

FindGetJson - Structured text (ST)
IF xDoneFindGet = FALSE THEN

    // Execute the WriteJsonData function
    WriteJsonData();

    // Search for the first element by Key -> device1
    WriteJsonData.pJsonDataWrite^.FindFirstElementByKey(wsKey := "device1", diStartIndex := 0, jsonElement => jsonElement1);

    // Search for the first value by Key -> temperature
    WriteJsonData.pJsonDataWrite^.FindFirstValueByKey(wsKey := "temperature", diStartIndex := 0, jsonElement => jsonElement2);

    // Get the Element by Index
    WriteJsonData.pJsonDataWrite^.GetElementByIndex(diIndex := 15, jsonElement => jsonElement3);
    WriteJsonData.pJsonDataWrite^.GetElementByIndex(diIndex := 16, jsonElement => jsonElement4);

    // jsonElement1 = element of the key
    diElement1Type := jsonElement1.eType;
    wsElement1Value := jsonElement1.value.wsValue;
    diElement1Index := jsonElement1.diIndex;
    diElement1ParIndex := jsonElement1.diParentIndex;

    // jsonElement2 = value of the key
    diElement2Type := jsonElement2.eType;
    lrElement2Value := jsonElement2.value.lrValue;
    diElement2Index := jsonElement2.diIndex;
    diElement2ParIndex := jsonElement2.diParentIndex;

    // jsonElement3
    JSON.JSONElementToString(element := jsonElement3, wsResult := wsElement3Value);

    // jsonElement4
    JSON.JSONElementToString(element := jsonElement4, wsResult := wsElement4Value);

END_IF

```

Figure 3.10: FindGetJson program in structured text.

In a similar way to what was shown in the previous section, with the program running it is possible to check the content of the elements obtained using the methods present in the JSONData FB. Figure 3.11 shows the variable declaration screen of the FindGetJson program in *online* mode, and Figure 3.12 shows the values assigned to the variables.

Device.Application.FindGetJson			
Expression	Type	Value	Comment
[-] jsonElement1	JSON.JSONElement		jsonElements
[-] value	JSONValue		Value of the element
[-] wsValue	WSTRING(GParams.g_dMaxStringSize)	"device1"	Value as WSTRING
[-] xValue	BOOL	<Value of the expression cannot ...	Value as BOOL
[-] lrValue	LREAL	1.1126170568611754E-306	Value as LREAL
[-] liValue	LINT	29555379367379044	Value as LINT
[-] eType	JSONTYPE	KEY	Type of the value
[-] diIndex	DINT	1	Index of this element in the data array
[-] diParentIndex	DINT	0	Index of the parent JSONElement, -1 for toplevel elements.
[-] jsonElement2	JSON.JSONElement		jsonElements
[-] value	JSONValue		Value of the element
[-] wsValue	WSTRING(GParams.g_dMaxStringSize)	"粉香香錶"	Value as WSTRING
[-] xValue	BOOL	<Value of the expression cannot ...	Value as BOOL
[-] lrValue	LREAL	39.2	Value as LREAL
[-] liValue	LINT	4630713726853028250	Value as LINT
[-] eType	JSONTYPE	LREAL_VALUE	Type of the value
[-] diIndex	DINT	4	Index of this element in the data array
[-] diParentIndex	DINT	3	Index of the parent JSONElement, -1 for toplevel elements.
[-] jsonElement3	JSON.JSONElement		jsonElements
[-] value	JSONValue		Value of the element
[-] wsValue	WSTRING(GParams.g_dMaxStringSize)	"time"	Value as WSTRING
[-] xValue	BOOL	<Value of the expression cannot ...	Value as BOOL
[-] lrValue	LREAL	9.3460503687416908E-307	Value as LREAL
[-] liValue	LINT	28429440806092916	Value as LINT
[-] eType	JSONTYPE	KEY	Type of the value
[-] diIndex	DINT	15	Index of this element in the data array
[-] diParentIndex	DINT	0	Index of the parent JSONElement, -1 for toplevel elements.
[-] jsonElement4	JSON.JSONElement		jsonElements
[-] value	JSONValue		Value of the element
[-] wsValue	WSTRING(GParams.g_dMaxStringSize)	"❖❖❖"	Value as WSTRING
[-] xValue	BOOL	<Value of the expression cannot ...	Value as BOOL
[-] lrValue	LREAL	8.44674349964981E-312	Value as LREAL
[-] liValue	LINT	1709639917438	Value as LINT
[-] eType	JSONTYPE	LINT_VALUE	Type of the value
[-] diIndex	DINT	16	Index of this element in the data array
[-] diParentIndex	DINT	15	Index of the parent JSONElement, -1 for toplevel elements.
[-] diElement1Type	DINT	0	jsonElement1

Figure 3.11: Variable declaration screen for the FindGetJson program in online mode.

```

// jsonElement1 = element of the key
diElement1Type := jsonElement1.eType KEY;
wsElement1Value := jsonElement1.value.wsValue "device1";
diElement1Index := jsonElement1.diIndex 1;
diElement1ParIndex := jsonElement1.diParentIndex 0;

// jsonElement2 = value of the key
diElement2Type := jsonElement2.eType LREAL_VALU;
lrElement2Value := jsonElement2.value.lrValue 39.2;
diElement2Index := jsonElement2.diIndex 4;
diElement2ParIndex := jsonElement2.diParentIndex 3;

// jsonElement3
JSON.JSONElementToString(element := jsonElement3, wsResult := wsElement3Value "time");

// jsonElement4
JSON.JSONElementToString(element := jsonElement4, wsResult := wsElement4Value "1705682686");

```

Figure 3.12: Values assigned to variables in the FindGetJson program in online mode.

4 MQTT CLIENT + JSON UTILITIES

In order to demonstrate the use of the **MQTT Client SL** and **JSON Utilities SL** libraries together, applications are created showing the creation and publication of a JSON, as well as the subscription and consequent manipulation of the received JSON.

4.1 CREATING AND PUBLISHING A JSON

In the **MQTT_JSON_Pub** application, two JSONs are created: one for status, containing the current value of the first two digital outputs of the PLC, and the other for command, which is used to change the state of these digital outputs. Publication topics are **"weg/drivesbt/state"** and **"weg/drivesbt/command"**, respectively. Two function blocks of the **MQTTPublish** type are used, each responsible for publishing a JSON. A state machine is used to manage these operations.

Figure 4.1 shows the variable declaration, and Figure 4.2 shows the ST code for the MQTT_JSON_Pub application.

MQTT_JSON_Pub - Declaration

```
PROGRAM MQTT_JSON_Pub
VAR
    // Functions Blocks MQTT
    MQTTClient_0: MQTT.MQTTClient;
    MQTTPublish_1, MQTTPublish_2: MQTT.MQTTPublish;

    // Creating of the JSONData
    factory : JSON.JSONDataFactory; // Allocate memory to the JSONData
    eDataFactoryError : FBF.ERROR; // Error code: 0: OK, 30103: no more memory
    pJsonData1 : POINTER TO JSON.JSONData := factory.Create(eError => eDataFactoryError);
    pJsonData2 : POINTER TO JSON.JSONData := factory.Create(eError => eDataFactoryError);

    // JSON
    jsonArrayWriter : JSON.JSONByteArrayWriter;
    wsJsonData1, wsJsonData2 : WSTRING(JSON.GParams.g_diMaxStringSize);
    xDoneWrite1, xDoneWrite2, xErrorWrite1, xErrorWrite2 : BOOL := 0;

    // MQTTClient
    sHostName : STRING := 'test.mosquitto.org';
    uiPort : UINT := 1883;
    xConnectedToBroker, xDoneClient, xErrorClient : BOOL;
    eMQTTErrorClient : MQTT.MQTT_ERROR;
    uiQoS : MQTT.MQTT_QOS := 1;

    // MQTTPublish
    sPayloadPub1, sPayloadPub2 : STRING(JSON.GParams.g_diMaxStringSize);
    pbPayloadPub1 : POINTER TO BYTE := ADR(sPayloadPub1);
    pbPayloadPub2 : POINTER TO BYTE := ADR(sPayloadPub2);
    wsTopicPub1 : WSTRING(1024) := "weg/drivesbt/state";
    wsTopicPub2 : WSTRING(1024) := "weg/drivesbt/command";
    udiPayloadSizePub1, udiPayloadSizePub2 : UDINT;
    xDonePub1, xErrorPub1, xDonePub2, xErrorPub2 : BOOL;
    eMQTTErrorPub1, eMQTTErrorPub2 : MQTT.MQTT_ERROR;

    // PRG variables
    diCounter, diObject1, diObject2 : DINT;
    liDateTime : LINT := 0;
    Rtc : ULINT;
    STATE, uiCount : UINT := 0;
    RS_0 : RS;
    TON_0 : TON;
END_VAR
```

Figure 4.1: Declaration of MQTT_JSON_Pub application variables.

MQTT_JSON_Pub - Structured text (ST) - Part 1

```
// Check MQTTClient error
RS_0(SET := NOT xErrorClient, RESET1 := xErrorClient); // Set Reset
TON_0(IN := RS_0.Q1, PT := T#1S); // Timer

// FB MQTTClient
MQTTClient_0(
  xEnable := TON_0.Q, sHostname := sHostName, uiPort := uiPort, uiKeepAlive := 60,
  xError => xErrorClient, eMQTTError => eMQTTErrorClient, xConnectedToBroker => xConnectedToBroker, xDone => xDoneClient);

// Publish State Machine
CASE STATE OF

0: // Increment uiCount
  uiCount := uiCount + 1;
  IF uiCount > 50 THEN uiCount := 0; STATE := 1; END_IF;

1: // Variables of the payload

  // Timestamp (use the library: SysTimeRtc)
  SysTimeRtcHighResGet(pTimestamp := Rtc);
  liDateTime := ULINT_TO_LINT(Rtc);

  // Switching DO states
  IF DO1 = 1 THEN DO1 := 0; ELSE DO1 := 1; END_IF;
  IF DO2 = 1 THEN DO2 := 0; ELSE DO2 := 1; END_IF;

STATE := 2;

2: // Creation of the first JSON (state)
  diCounter := 0;
  pJsonData1^.SetObject(diIndex := diCounter, diParentIndex := -1);
  diCounter := diCounter + 1;
  pJsonData1^.SetKey(wsKey := "state", diParentIndex := 0, diIndex := diCounter);
  diCounter := diCounter + 1; diObject1 := diCounter;
  pJsonData1^.SetObject(diIndex := diCounter, diParentIndex := diCounter - 1);

  diCounter := diCounter + 1;
  pJsonData1^.SetKey(wsKey := "DO1", diParentIndex := diObject1, diIndex := diCounter);
  diCounter := diCounter + 1;
  pJsonData1^.SetBool(bValue := DO1, diIndex := diCounter, diParentIndex := diCounter - 1);
  diCounter := diCounter + 1;
  pJsonData1^.SetKey(wsKey := "DO2", diParentIndex := diObject1, diIndex := diCounter);
  diCounter := diCounter + 1;
  pJsonData1^.SetBool(bValue := DO2, diIndex := diCounter, diParentIndex := diCounter - 1);

  diCounter := diCounter + 1;
  pJsonData1^.SetKey(wsKey := "time", diParentIndex := 0, diIndex := diCounter);
  diCounter := diCounter + 1;
  pJsonData1^.SetLInt(liValue := liDateTime, diIndex := diCounter, diParentIndex := diCounter - 1);

  STATE := 3;

3: // Creation of the second JSON (command)
  diCounter := 0;
  pJsonData2^.SetObject(diIndex := diCounter, diParentIndex := -1);
  diCounter := diCounter + 1;
  pJsonData2^.SetKey(wsKey := "command", diParentIndex := 0, diIndex := diCounter);
  diCounter := diCounter + 1; diObject2 := diCounter;
  pJsonData2^.SetObject(diIndex := diCounter, diParentIndex := diCounter - 1);

  diCounter := diCounter + 1;
  pJsonData2^.SetKey(wsKey := "DO1", diParentIndex := diObject2, diIndex := diCounter);
  diCounter := diCounter + 1;
  pJsonData2^.SetBool(bValue := DO1, diIndex := diCounter, diParentIndex := diCounter - 1);
  diCounter := diCounter + 1;
  pJsonData2^.SetKey(wsKey := "DO2", diParentIndex := diObject2, diIndex := diCounter);
  diCounter := diCounter + 1;
  pJsonData2^.SetBool(bValue := DO2, diIndex := diCounter, diParentIndex := diCounter - 1);

  diCounter := diCounter + 1;
  pJsonData2^.SetKey(wsKey := "time", diParentIndex := 0, diIndex := diCounter);
  diCounter := diCounter + 1;
  pJsonData2^.SetLInt(liValue := liDateTime, diIndex := diCounter, diParentIndex := diCounter - 1);

  STATE := 4;
```

MQTT_JSON_Pub - Structured text (ST) - Part 2

```

4: // Generate the wsJsonData1 JSON
jsonArrayWriter(xExecute := TRUE, pwData := ADR(wsJsonData1), udiSize := SIZEOF(wsJsonData1), jsonData := pJsonData1^,
  xError => xErrorWrite1, xDone => xDoneWrite1);

IF xDoneWrite1 = 1 OR xErrorWrite1 = 1 THEN
  jsonArrayWriter(xExecute := FALSE, pwData := ADR(wsJsonData1), udiSize := SIZEOF(wsJsonData1), jsonData := pJsonData1^,
    xError => xErrorWrite1, xDone => xDoneWrite1);
  STATE := 5;
END_IF

5: // Generate the wsJsonData2 JSON
jsonArrayWriter(xExecute := TRUE, pwData := ADR(wsJsonData2), udiSize := SIZEOF(wsJsonData2), jsonData := pJsonData2^,
  xError => xErrorWrite2, xDone => xDoneWrite2);

IF xDoneWrite2 = 1 OR xErrorWrite2 = 1 THEN
  jsonArrayWriter(xExecute := FALSE, pwData := ADR(wsJsonData2), udiSize := SIZEOF(wsJsonData2), jsonData := pJsonData2^,
    xError => xErrorWrite2, xDone => xDoneWrite2);
  STATE := 6;
END_IF

6: // Creation of the payloads and calculate their sizes

// Calculates the WSTRING sizes
udiPayloadSizePub1 := DINT_TO_UINT(StrLenW(pstData := ADR(wsJsonData1)));
udiPayloadSizePub2 := DINT_TO_UINT(StrLenW(pstData := ADR(wsJsonData2)));

// Conversion from UTF16 to UTF8 (WSTRING to STRING)
ConvertUTF16toUTF8(sourceStart := ADR(wsJsonData1), targetStart := ADR(sPayloadPub1),
  dwTargetBufferSize := udiPayloadSizePub1, bStrictConversion := 0);

ConvertUTF16toUTF8(sourceStart := ADR(wsJsonData2), targetStart := ADR(sPayloadPub2),
  dwTargetBufferSize := udiPayloadSizePub2, bStrictConversion := 0);

// Calculates the STRING sizes
udiPayloadSizePub1 := DINT_TO_UINT(StrLenA(pstData := pbPayloadPub1));
udiPayloadSizePub2 := DINT_TO_UINT(StrLenA(pstData := pbPayloadPub2));

STATE := 7;

7: // Publisher 1
MQTTPublish_1(
  xExecute := ((NOT xErrorPub1) AND (NOT xDonePub1) AND (xConnectedToBroker)), wsTopicName := wsTopicPub1,
  eQoS := uiQoS, mqttClient := MQTTClient_0, pbPayload := pbPayloadPub1, udiPayloadSize := udiPayloadSizePub1,
  xDone => xDonePub1, xError => xErrorPub1, eMQTTErr := eMQTTErrPub1);

// Reset publisher 1
IF xDonePub1 = 1 OR xErrorPub1 = 1 THEN MQTTPublish_1.xExecute := 0; STATE := 8; END_IF

8: // Publisher 2
MQTTPublish_2(
  xExecute := ((NOT xErrorPub2) AND (NOT xDonePub2) AND (xConnectedToBroker)), wsTopicName := wsTopicPub2,
  eQoS := uiQoS, mqttClient := MQTTClient_0, pbPayload := pbPayloadPub2, udiPayloadSize := udiPayloadSizePub2,
  xDone => xDonePub2, xError => xErrorPub2, eMQTTErr := eMQTTErrPub2);

// Reset publisher 2
IF xDonePub2 = 1 OR xErrorPub2 = 1 THEN MQTTPublish_2.xExecute := 0; STATE := 0; END_IF

END_CASE;

```

Figure 4.2: MQTT_JSON_Pub program in structured text.

4.2 SUBSCRIBING AND MANIPULATING A JSON

In this example, the **MQTTSubscribe_1** FB waits for payloads published in the **"weg/drivesbt/state"** topic, and the **MQTTSubscribe_2** waits for publications in **"weg/drivesbt/command"**. When a payload is received, the **xReceived** variable of the respective function block goes to the high state, and the content of the received pointer is transformed into JSON, which can be easily manipulated to acquire the keys and values of interest.

The values received in the status topic are only used to display the current status of the product's first two digital outputs. However, the values received in the command topic are used to define the state of these two digital outputs.

Figure 4.3 shows the variable declaration, and 4.4 shows the code in ST for the MQTT_JSON_Sub application.

MQTT_JSON_Sub - Declaration
<pre> PROGRAM MQTT_JSON_Sub VAR // Functions Blocks MQTT MQTTClient_0: MQTT.MQTTClient; MQTTSubscribe_1, MQTTSubscribe_2 : MQTT.MQTTSubscribe; // Creating of the JSONDatas factory : JSON.JSONDataFactory; // Allocate memory to the JSONData eDataFactoryError : FBF.ERROR; // Error code: 0: OK, 30103: no more memory pJsonDataState : POINTER TO JSON.JSONData := factory.Create(eError => eDataFactoryError); pJsonDataCommand : POINTER TO JSON.JSONData := factory.Create(eError => eDataFactoryError); // JSON jsonArrayReaderSub1, jsonArrayReaderSub2 : JSON.JSONByteArrayReader; wsJsonDataState, wsJsonDataCommand : WSTRING(JSON.GParams.g_diMaxStringSize); jsonElement1, jsonElement2, jsonElement3, jsonElement4 : JSON.JSONElement; // MQTTClient sHostName : STRING := 'test.mosquitto.org'; uiPort : UINT := 1883; xConnectedToBroker, xDoneClient, xErrorClient : BOOL; eMQTTErrorClient : MQTT.MQTT_ERROR; uiQoS : MQTT.MQTT_QOS := 1; // MQTTSubscribe sPayloadSubState, sPayloadSubCommand : STRING(JSON.GParams.g_diMaxStringSize); pbPayloadSubState : POINTER TO BYTE := ADR(sPayloadSubState); pbPayloadSubCommand : POINTER TO BYTE := ADR(sPayloadSubCommand); wsTopicSubState: WSTRING(1024) := "weg/drivesbt/state" ; wsTopicSubCommand: WSTRING(1024) := "weg/drivesbt/command" ; udiMaxPayloadSize : UDINT := DINT_TO_UDINT(JSON.GParams.g_diMaxStringSize); udiPayloadSizeSubState, udiPayloadSizeSubCommand : UDINT := DINT_TO_UDINT(JSON.GParams.g_diMaxStringSize); xDoneSub1, xErrorSub1, xReceived1, xSubscribeActive1 : BOOL; xDoneSub2, xErrorSub2, xReceived2, xSubscribeActive2 : BOOL; eMQTTErrorSub1, eMQTTErrorSub2: MQTT.MQTT_ERROR; // PRG variables liDateTime : LINT := 0; Rtc : ULINT; RS_0 : RS; TON_0 : TON; END_VAR </pre>

Figure 4.3: Declaration of MQTT_JSON_Sub variables.

MQTT_JSON_Sub - Structured text (ST)

```

// Check MQTTClient error
RS_0(SET := NOT xErrorClient, RESET1 := xErrorClient); // Set Reset
TON_0(IN := RS_0.Q1, PT := T#1S); // Timer

// FB MQTTClient
MQTTClient_0(
  xEnable := TON_0.Q, sHostname := sHostName, uiPort := uiPort, uiKeepAlive := 60,
  xError => xErrorClient, eMQTTError => eMQTTErrorClient, xConnectedToBroker => xConnectedToBroker, xDone => xDoneClient);

// FB MQTTSubscribe 1
MQTTSubscribe_1(
  xEnable := ((NOT xErrorSub1) AND (NOT xDoneSub1) AND (xConnectedToBroker)), wsTopicFilter := wsTopicSubState,
  eSubscribeQoS := uiQoS, mqttClient := MQTTClient_0, pbPayload := pbPayloadSubState,
  udiMaxPayloadSize := udiMaxPayloadSize, xDone => xDoneSub1, xError => xErrorSub1, eMQTTError => eMQTTErrorSub1,
  xReceived => xReceived1, udiPayloadSize => udiPayloadSizeSubState, xSubscribeActive => xSubscribeActive1);

// FB MQTTSubscribe 2
MQTTSubscribe_2(
  xEnable := ((NOT xErrorSub2) AND (NOT xDoneSub2) AND (xConnectedToBroker)), wsTopicFilter := wsTopicSubCommand,
  eSubscribeQoS := uiQoS, mqttClient := MQTTClient_0, pbPayload := pbPayloadSubCommand,
  udiMaxPayloadSize := udiMaxPayloadSize, xDone => xDoneSub2, xError => xErrorSub2, eMQTTError => eMQTTErrorSub2,
  xReceived => xReceived2, udiPayloadSize => udiPayloadSizeSubCommand, xSubscribeActive => xSubscribeActive2);

// Payload received from "weg/drivesbt/state"
IF xReceived1 = 1 THEN

  // Treating the pointer to byte
  ConvertUTF8toUTF16(sourceStart := pbPayloadSubState, targetStart := ADR(wsJsonDataState),
    dwTargetBufferSize := udiMaxPayloadSize, bStrictConversion := 0);

  // Calculates the STRING size
  udiPayloadSizeSubState := DINT_TO_UINT(StrLenW(pstData := ADR(wsJsonDataState)));

  // Read the WSTRING and trasform it to JSON
  jsonArrayReaderSub1(xExecute := TRUE, pwData := ADR(wsJsonDataState), jsonData := pJsonDataState^);
  IF jsonArrayReaderSub1.xError = 1 OR jsonArrayReaderSub1.xDone = 1 THEN
    jsonArrayReaderSub1(xExecute := FALSE, pwData := ADR(wsJsonDataState), jsonData := pJsonDataState^);
  END_IF

  // Find the values of the keys DO1 and DO2
  pJsonDataState^.FindFirstValueByKey(wsKey := "DO1", diStartIndex := 1, jsonElement => jsonElement1);
  pJsonDataState^.FindFirstValueByKey(wsKey := "DO2", diStartIndex := 1, jsonElement => jsonElement2);

END_IF

// Payload received from "weg/drivesbt/command"
IF xReceived2 = 1 THEN

  // Treating the pointer to byte
  ConvertUTF8toUTF16(sourceStart := pbPayloadSubCommand, targetStart := ADR(wsJsonDataCommand),
    dwTargetBufferSize := udiMaxPayloadSize, bStrictConversion := 0);

  // Calculates the STRING size
  udiPayloadSizeSubCommand := DINT_TO_UINT(StrLenW(pstData := ADR(wsJsonDataCommand)));

  // Read the WSTRING and trasform it to JSON
  jsonArrayReaderSub2(xExecute := TRUE, pwData := ADR(wsJsonDataCommand), jsonData := pJsonDataCommand^);
  IF jsonArrayReaderSub2.xError = 1 OR jsonArrayReaderSub2.xDone = 1 THEN
    jsonArrayReaderSub2(xExecute := FALSE, pwData := ADR(wsJsonDataCommand), jsonData := pJsonDataCommand^);
  END_IF

  // Find the values of the keys and add it to the DO1 and DO2 variables of the PLCx
  pJsonDataCommand^.FindFirstValueByKey(wsKey := "DO1", diStartIndex := 1, jsonElement => jsonElement3);
  pJsonDataCommand^.FindFirstValueByKey(wsKey := "DO2", diStartIndex := 1, jsonElement => jsonElement4);

  DO1 := jsonElement3.value.xValue;
  DO2 := jsonElement4.value.xValue;

END_IF

```

Figure 4.4: MQTT_JSON_Sub program in structured text.

4.3 PUBLICATION AND SUBSCRIPTION

Applications 4.1 and 4.2 were created to enable integration, allowing publication and subscription to the same topics, thus being able to be applied both in a single PLC and in different PLCs. Use both examples simultaneously for a complete publication and subscription process.



WEG Drives & Controls - Automation LTDA.
Jaraguá do Sul - SC - Brazil
Phone 55 (47) 3276-4000 - Fax 55 (47) 3276-4020
São Paulo - SP - Brazil
Phone 55 (11) 5053-2300 - Fax 55 (11) 5052-4212
automacao@weg.net
www.weg.net