MQTT Protocol and Its Implementation in pbsSoftLogic and pbsHMI

MQTT is a communication protocol defined by the OASIS standards, designed specifically for Internet of Things (IoT) applications.

It operates using a publish/subscribe messaging model and is ideal for connecting remote devices that have limited memory, low processing capability, or minimal available bandwidth.

Today, MQTT is widely used across many industries — including automotive systems, manufacturing lines, telecommunications, and the oil and gas sector — due to its lightweight design and reliability in distributed environments.

The MQTT protocol has the following characteristics:

- **Client/Server Architecture:**
  MQTT is based on a client/server model and typically operates over TCP/IP networks.
  A newer variant, **MQTT-SN**, is available for non-TCP/IP networks such as Zigbee.

- **No Data Model Awareness:**
  MQTT is not aware of the structure or meaning of the message payload.
  It simply transports the payload as-is, unlike protocols such as **DNP3** or **IEC-104**, which define their own data models.

- **Flexible Payload Format:**
  An MQTT message can contain any type of payload, including files, images, audio, JSON, XML, and more.

- **Three QoS (Quality of Service) Levels:**
  MQTT defines three delivery guarantees:

  - **QoS 0 – At most once:**
    The message may not reach the destination.
    This level is typically used for periodic data where losing a single message is acceptable because the next update will arrive in the next cycle.

  - **QoS 1 – At least once:**
    The message is guaranteed to reach the destination, but duplicates may occur.

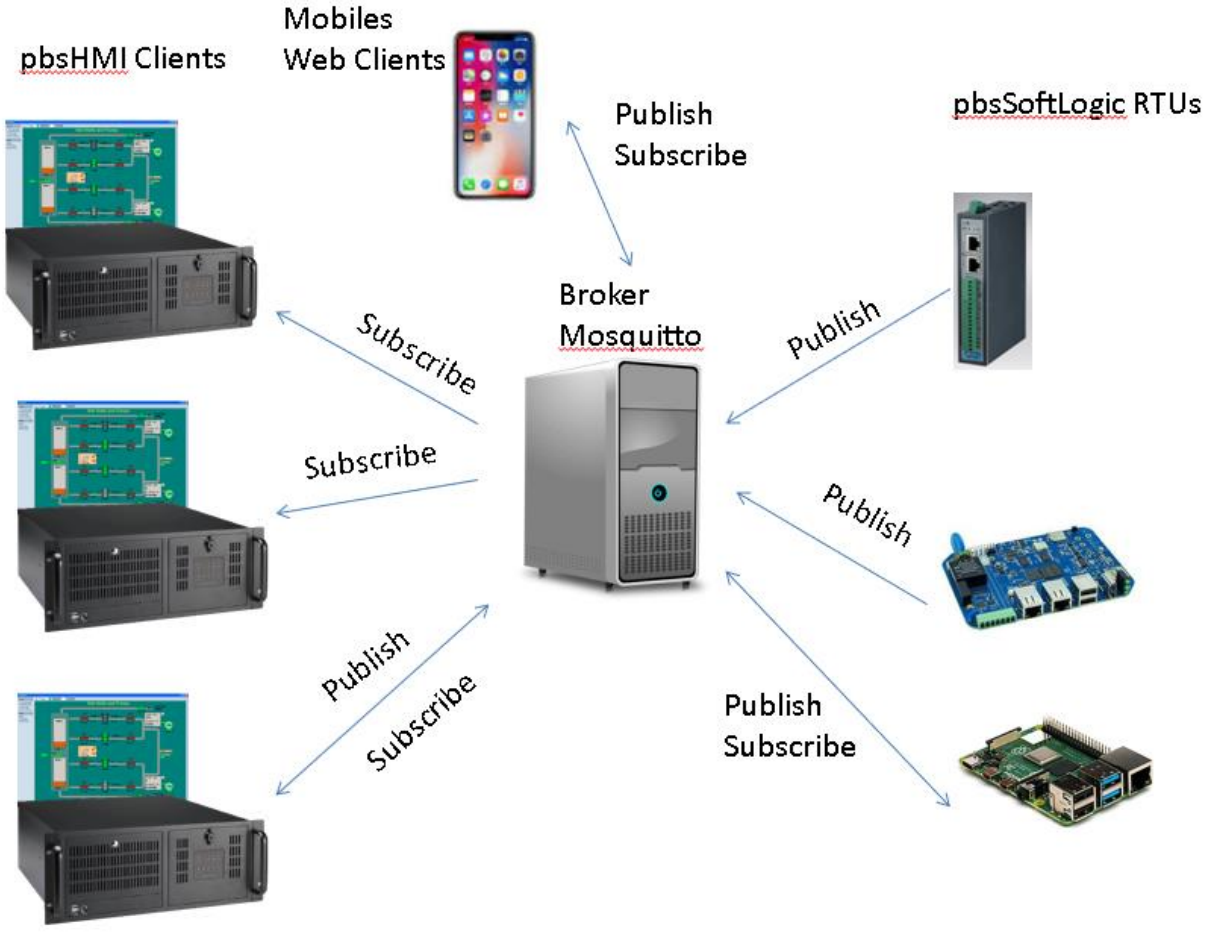
  - **QoS 2 – Exactly once:**
    The message is guaranteed to arrive **exactly once**, with no duplicates.
    This level is commonly used in payment or financial systems where message duplication is unacceptable.

An MQTT-based system consists of the following components:

- **Clients:**
  These can include computers, RTUs, mobile devices, sensors, and any equipment used to send or receive data.

- **Broker (Server):**
  The broker acts as the intermediary between clients and is responsible for routing messages among them.
  Unlike protocols such as **DNP3** or **IEC-104**, MQTT does **not** establish a direct connection between two endpoints.
  Instead, clients **publish** messages to the broker under a specific **topic**, and any client that has **subscribed** to that topic will receive the message.

As a result, a single message can be delivered to multiple clients simultaneously.
The presence of a broker is one of the key architectural features that distinguishes MQTT from many other communication protocols.



As shown in the system architecture, RTUs, stations running **pbsHMI**, and mobile devices all operate as **MQTT clients**.
Each client can **publish** and **subscribe** simultaneously.

RTUs publish the data they need to send to control centers, and they subscribe to **setpoints** and **commands**.
In **pbsHMI**, all data published by RTUs must be subscribed to, and all commands and setpoints intended for RTUs must be published by the HMI.

Mobile devices, using available applications for Android, iOS, and other platforms, can also receive data from the broker or send data to RTUs.

This architecture allows multiple control centers to operate simultaneously.
All of them can receive RTU data or send commands and setpoints to RTUs through the broker without conflict.

There are various MQTT broker implementations available, but our system uses **Mosquitto** on both Windows and Linux platforms.

The MQTT driver for **pbsSoftLogic** has been developed using Mosquitto, while **pbsHMI** uses the **MQTTnet** library.

In this document, the terms **Broker** and **Server** refer to the same component, and RTUs as well as pbsHMI instances are all considered **clients**.

To deploy this architecture over the public Internet, the broker must have a **valid static IP address**.
If the entire system operates within a private **APN**, then a valid public IP is not required.

Other clients do **not** need static IP addresses; they can connect to the broker using **dynamic IPs**.

Note that the connection between each client and the broker remains active throughout communication, but the **Connect** request is always initiated by the client toward the broker.

Types of Requests Between MQTT Clients and the Broker

In every MQTT message, there is a **Control Byte** that specifies the type of request being sent.
This control byte defines the following message types:

Name	Value	Direction of flow	Description
Reserved	0	Forbidden	Reserved
CONNECT	1	Client to Server	Connection request
CONNACK	2	Server to Client	Connect acknowledgment
PUBLISH	3	Client to Server or Server to Client	Publish message
PUBACK	4	Client to Server or Server to Client	Publish acknowledgment (QoS 1)
PUBREC	5	Client to Server or Server to Client	Publish received (QoS 2 delivery part 1)
PUBREL	6	Client to Server or Server to Client	Publish release (QoS 2 delivery part 2)
PUBCOMP	7	Client to Server or Server to Client	Publish complete (QoS 2 delivery part 3)
SUBSCRIBE	8	Client to Server	Subscribe request
SUBACK	9	Server to Client	Subscribe acknowledgment
UNSUBSCRIBE	10	Client to Server	Unsubscribe request
UNSUBACK	11	Server to Client	Unsubscribe acknowledgment
PINGREQ	12	Client to Server	PING request
PINGRESP	13	Server to Client	PING response
DISCONNECT	14	Client to Server or Server to Client	Disconnect notification
AUTH	15	Client to Server or Server to Client	Authentication exchange

At the beginning of communication, the client sends a **CONNECT** request to establish a connection with the broker.
If the broker accepts the request, it responds with a **CONNACK** message.
This indicates that the underlying **TCP/IP** connection between the client and the broker has been successfully established.

Once the TCP connection between the client and the broker is established, the first message the client sends is the **CONNECT** command.

Each client must have a **unique ClientID**.
Therefore, every RTU and every instance of pbsHMI must use its own distinct ClientID.
In pbsSoftLogic and pbsHMI, this is handled automatically by generating random identifiers.

The client also specifies a **Keep Alive** interval when connecting to the broker.
In pbsSoftLogic, the default value is **20 seconds**, although this parameter can be adjusted.

If the Keep Alive interval expires and the client has not sent any message to the broker, the client must send a **PING** request.
If it fails to do so, the connection between the client and the broker is considered lost, and a new connection must be established.
If the Keep Alive value is set to zero, this mechanism is disabled.

For the physical transport layer, MQTT can operate over **TCP/IP**, **TLS**, or **WebSocket**.

Port **1883** is used for non-TLS MQTT communication, and port **8883** is reserved for TLS-encrypted connections.
MQTT does **not** use the UDP protocol.

Topic names must **not** begin with the **$** symbol.
The dollar sign has a special meaning in MQTT and is reserved for system topics.
Topic names are **case-sensitive**.

If the **/** character is used in a topic name, it creates a hierarchical structure.
For example, if a topic is defined as:

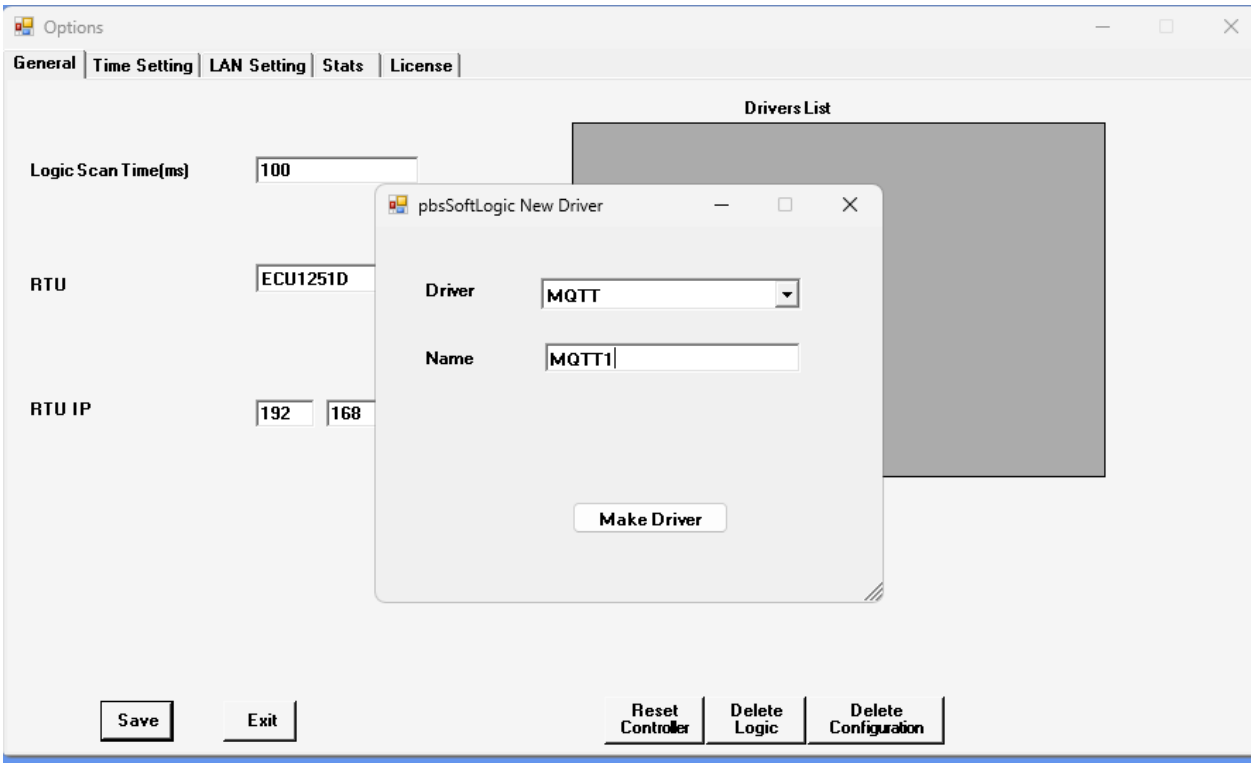
/group1/signal1

the broker interprets **group1** as a group (or level), and all topics beginning with /group1 are placed under this hierarchy.

MQTT Protocol in pbsSoftLogic

In both **pbsSoftLogic** and **pbsHMI**, the MQTT **client driver** has been fully implemented.

To add the MQTT driver to a pbsSoftLogic project, select **MQTT** from the list of available drivers:

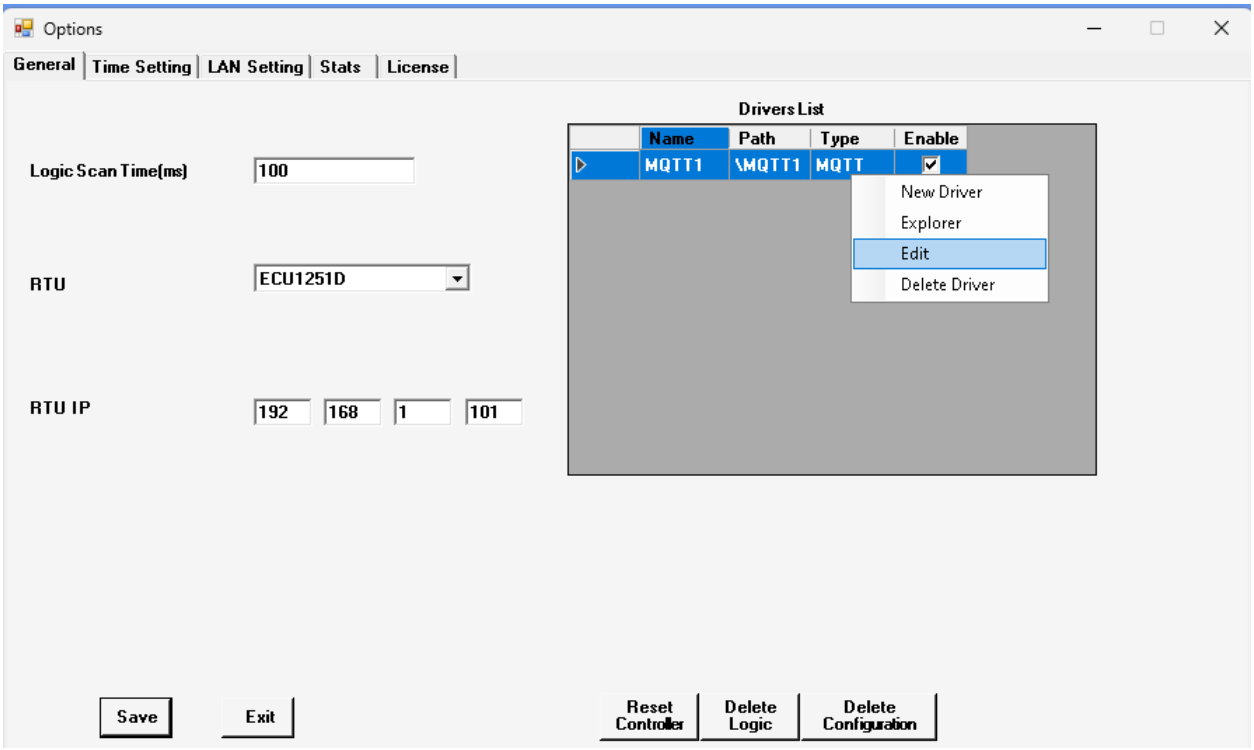


Choose a **unique name** for the driver.
In the latest version of **pbsSoftLogic**, the **Instance Number is assigned automatically**, so the user no longer needs to set it manually.

You can define up to **30 MQTT drivers** in a single project.
Each driver will automatically receive a unique Instance Number between **1 and 30**.
Multiple MQTT drivers are typically used for **gateway applications**, which will be explained in the next section.

After adding the driver, the software automatically generates the default configuration parameters and the initial set of tags.

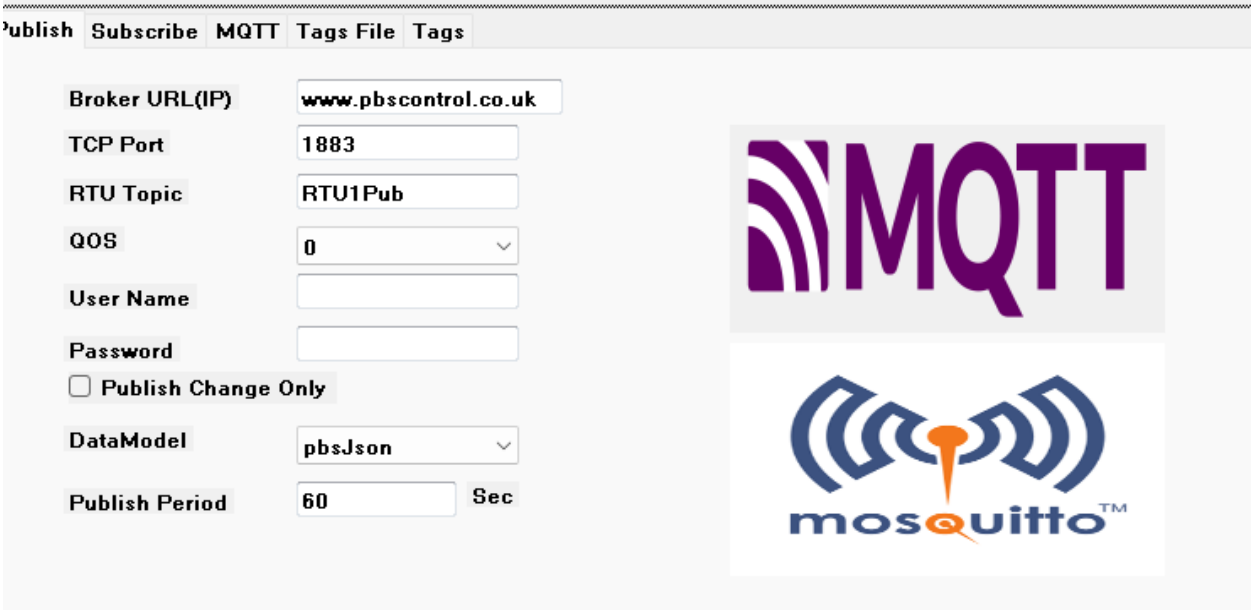
After adding the driver, you can open it and edit its configuration.



After editing the driver, the configuration GUI opens, allowing you to adjust all driver settings.

This window contains **four tabs**, used to configure **Publish**, **Subscribe**, **MQTT settings**, and **Tags**.

Publish Tab

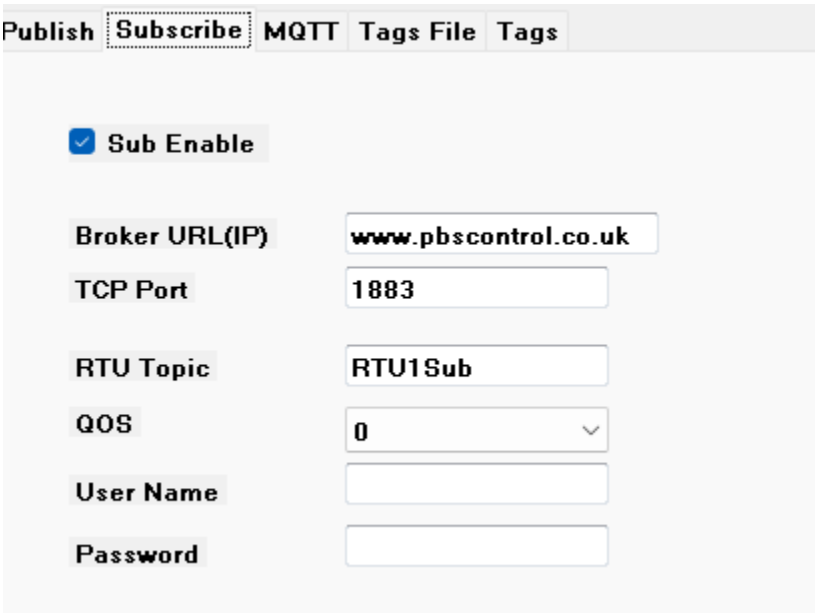


- **Broker IP:**
  Enter the IP address or hostname of the MQTT broker.
- **TCP Port:**
  The port number used to connect to the broker.
- **QoS:**
  Select the QoS level for publishing messages to the broker.
- **Username / Password:**
  If the broker requires authentication, enter the username and password here.
- **Publish Change Only:**
  Enable this option if you want the driver to publish **only when signal values change**.
  If this option is disabled, **all tags** will be published to the broker regardless of changes.
- **Data Model:**
  The MQTT protocol does **not** provide a built-in data model for SCADA systems.
  MQTT does not interpret or validate the structure of the transmitted data; it simply transports the payload as-is.

- In **pbsSoftLogic**, you cannot publish tags as **raw MQTT topics**.
  In other words, you cannot define a separate topic for each individual signal.
- The data model used in **pbsSoftLogic** and **pbsHMI** is based on **JSON**, and we refer to this structure as **pbsJson**.

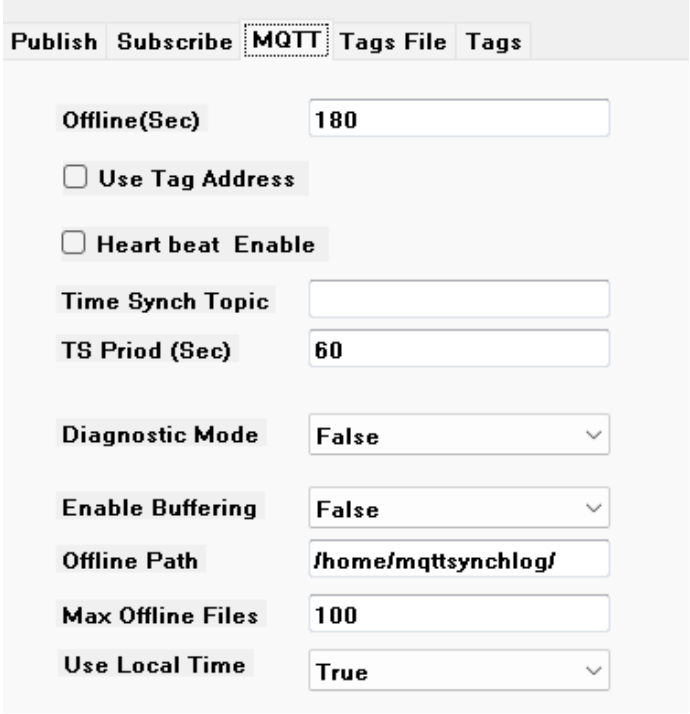
- **Publish Period:**
  This field specifies the interval, in seconds, at which tags are published to the broker.
- **RTU Topic:**
  As mentioned earlier, **pbsSoftLogic cannot publish tags as raw MQTT topics**.
  This means you cannot assign a separate topic for each individual signal.
  Instead, you must define **one topic for publishing** and **one topic for subscribing**.
  All tag data is sent to the broker in either **JSON** format, and **a single topic** is used to carry all information.

Subscribe Tab



In this section, if RTU wants to receive **setpoints** or **commands** from **pbsHMI** (the control center) you must enable **Subscribe**.
Enter the **Broker IP** and **TCP Port** and specify the **Topic** that the control center uses to publish its data.
Also enter the **QoS**, **Username**, and **Password** if authentication is required.
If the broker does not use authentication, leave the username and password fields empty.
The **data model used for subscribing** is the same as the data model used for publishing.

MQTT Tab



Offline Field

If this time interval expires and the driver is unable to send tags to the broker, the **Online** tag inside the program is set to **0**.

Otherwise, the **Online** tag remains **1**.

Use Tag Address

You can choose whether tags are sent to the control center using their **names** or their **addresses**.

Using **tag addresses** significantly reduces the size of the transmitted data and also improves security.

Each tag has a **unique address** within the driver.

Heart Beat Enable

When this option is enabled, a **Heart Beat** section is added at the beginning of every message sent to the broker.

The details of this mechanism will be explained in the next section.

Time Sync Topic

In the **pbsJson data model**, a dedicated topic must be defined for **time synchronization** between the RTU and the control center.

The details of this mechanism will be discussed in the Data Model section.

Diagnostic Mode

When this field is set to **True**, and the **psle kernel** is executed manually, you can view frame information in the Linux command window.

Enable Buffering

Enable this option if you want tag data to be automatically transferred from the MQTT drivers to **DNP3 Slave** or **IEC101/104 Slave** drivers.

Details of this mechanism will be explained in the next section.

Offline Path

If the RTU cannot send data to the broker, the data is stored in this directory.

As soon as the connection to the broker is restored, the stored data is transmitted.

Use Local Time

This field allows you to choose whether the driver uses **local time** or **UTC** for timestamps.

Tags Tab

Publish | Subscribe | MQTT | Tags

Name	Type	Address	SCADAAddress	Init	Log
SendByChange	SYS	0	-1	0	0
CommOnline	SYS	1	-1	0	0
DITag1	DI	1	-1	0	0
DITag2	DI	2	-1	0	0
DITag3	DI	3	-1	0	0
DITag16	DI	16	-1	0	0
AITag1	AI	17	-1	0	0
AITag2	AI	18	-1	0	0
AITag3	AI	19	-1	0	0
AITag4	AI	20	-1	0	0
DOTag1	DO	33	-1	0	0
DOTag2	DO	34	-1	0	0
AOTag1	AO	49	-1	0	0
AOTag2	AO	50	-1	0	0
AOTag3	AO	51	-1	0	0
AOTag4	AO	52	-1	0	0
AOTag5	AO	53	-1	0	0

To send data to the control center, you must define **tags**.

Tags are available in **five different types**.

SYS Tags are system-level tags, and currently two of them are defined.

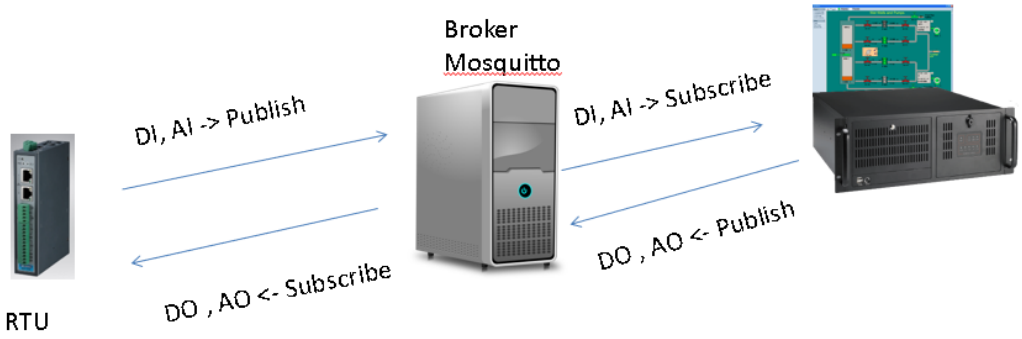
Whenever the **SendByChange** tag changes from **0 to 1**, a **publish operation** is triggered immediately, regardless of the publish interval configured in the driver.

The **CommOnline** tag becomes **1** when the driver is able to send data to the broker. If the **offline timeout** expires and the driver cannot transmit data, this tag becomes **0**.

You must **not change the name or address** of these two system tags.

To send data to the control center, you can define four types of tags: **DI, AI, DO, and AO**.

- **DI Tags:** Used to publish digital information to the broker. These tags can have values **0** or **1**.
- **AI Tags:** Used to publish any type of analog signal to the broker.
- **DO Tags:** These represent digital commands that the RTU must **subscribe** to. These commands are sent from the control center to the RTU.
- **AO Tags:** Used to write analog values from the control center to the RTU. Similar to DO tags, AO tags must also be **subscribed** by the RTU in the broker.

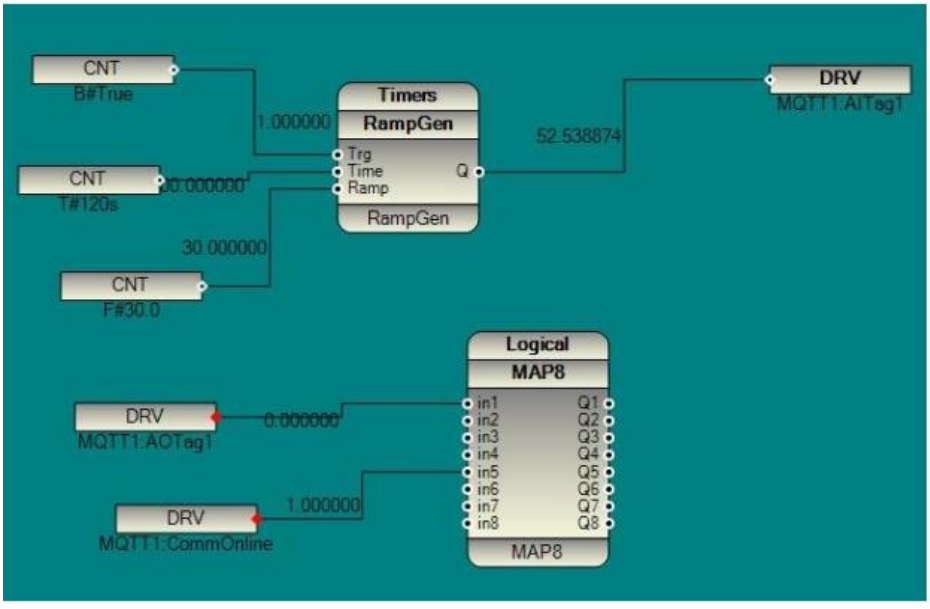


In the pbsHMI control-center software, **AI** and **DI** tags must be **subscribed**, and **DO** and **AO** tags must be **published**.

If a signal is configured as **log**, its latest value — typically the **setpoints** sent from the control center — is stored in the RTU's **flash memory**. After an RTU restart, this value is read back from flash so that the correct initial setpoint (the last value sent by the control center) is restored.

Each tag must be assigned a **unique address**. This allows the user to send data using tag **addresses** instead of tag names, and the address can then serve as the **ID** of the signal.

In the RTU logic, **DI** and **AI** tags must be **written**, and **DO** and **AO** tags must be **read** from the driver, as shown in the example below:



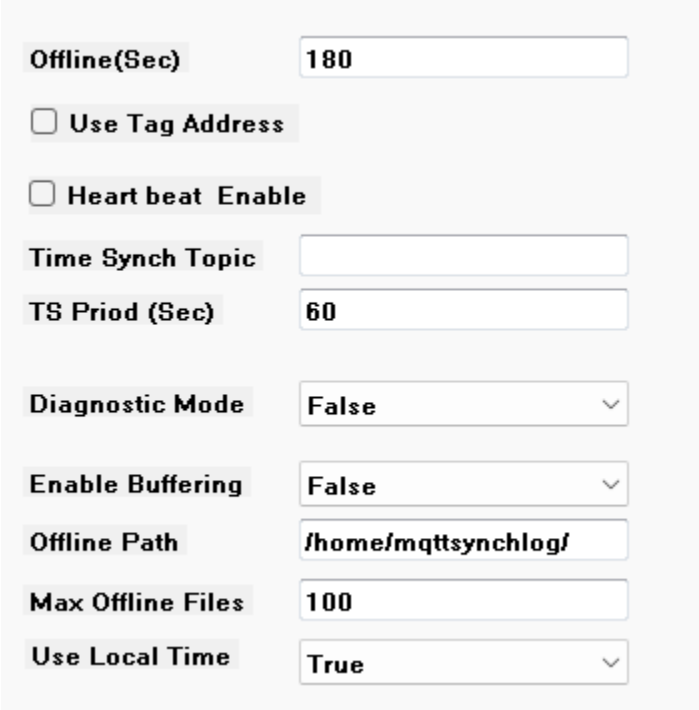
pbsJson Data Model

In this model, data is transmitted in the same **JSON format**, but each tag includes an **I field**, which indicates whether the data is **valid (0)** or **invalid (1)**.
When publishing only changed values to the broker, **empty tags** are sent as an empty JSON object {}. This prevents MQTT applications on other platforms (such as mobile devices) from encountering issues with missing tag addresses.
The **A field** in each tag represents the **signal name** or its **address**. The **V field** represents the **value** of the signal.
In the **rtu** tag at the beginning of the frame, there is a **status field** that indicates the RTU status. A value of **0** represents normal RTU operation.
If the RTU cannot communicate with the broker, the JSON file is stored in the RTU's flash memory at the configured path. In the **MQTT tab**, the **offline path** specifies this directory, and **max offline files** defines the maximum number of files that can be stored there.
When communication with the broker is restored, the files stored in this directory are sent to the control center **one by one**. It is important to note that each file contains the timestamp of when the frame was generated, and the control center should only **store these values in the database**, not treat them as real-time data.
In the following figure you can see a pbsJson Sample frame:


```
{
    "rtu":  {
        "topic":    "RTU74Pub",
        "time": 1686897160,
        "status":   0,
        "MqttHB":   "2"
    },
    "tags": [{
            "A":    "DITag1",
            "V":    1,
            "I":    0
        }, {
        }, {
        }, {
        }, {
            "A":    "AITag1",
            "V":    4.6188021535170058,
            "I":    0
        }, {
            "A":    "AITag2",
            "V":    14.641602826648908,
            "I":    0
        }, {
        }, {
        }, {
        }, {
        }]
}
```

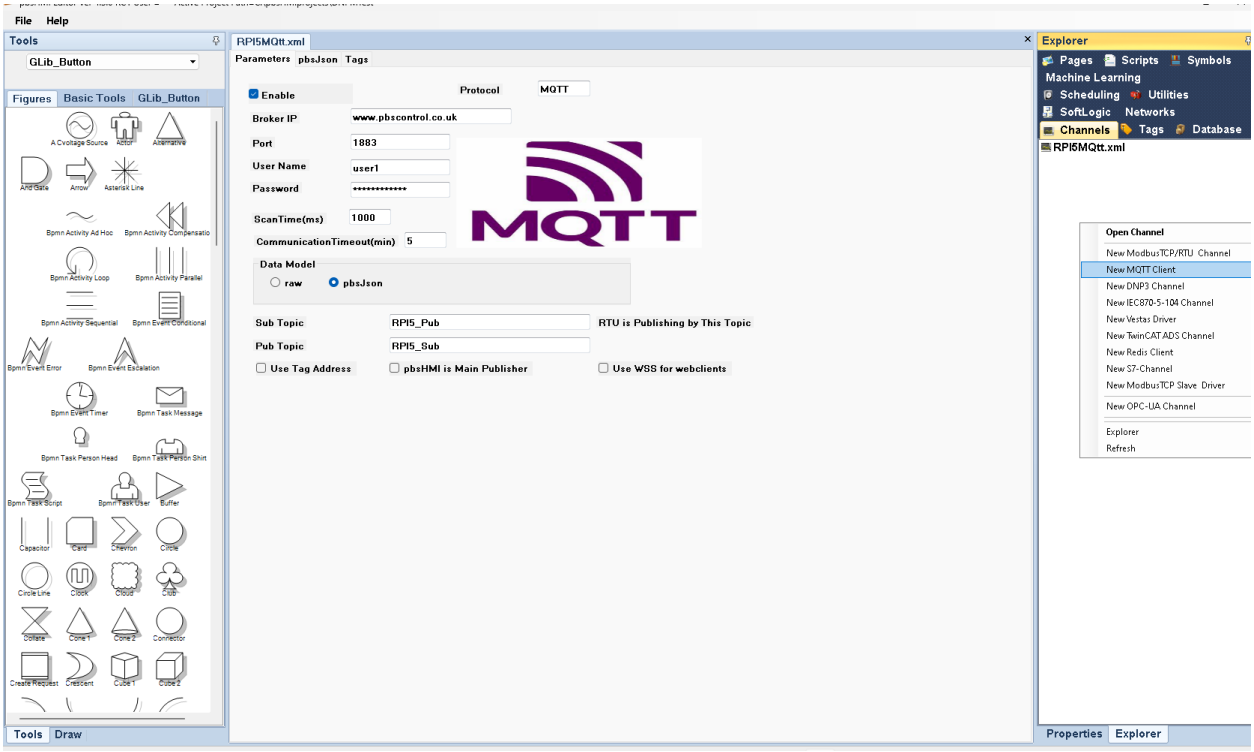
Time synch Synchronization

To synchronize time between the RTU and the Master SCADA system (pbsHMI), both sides must use the same **TimeSync Topic**. pbsHMI periodically updates this topic, and whenever the RTU needs to obtain the current time, it **subscribes** to the TimeSync Topic. After receiving the updated time value, the RTU immediately **unsubscribes** from the topic.

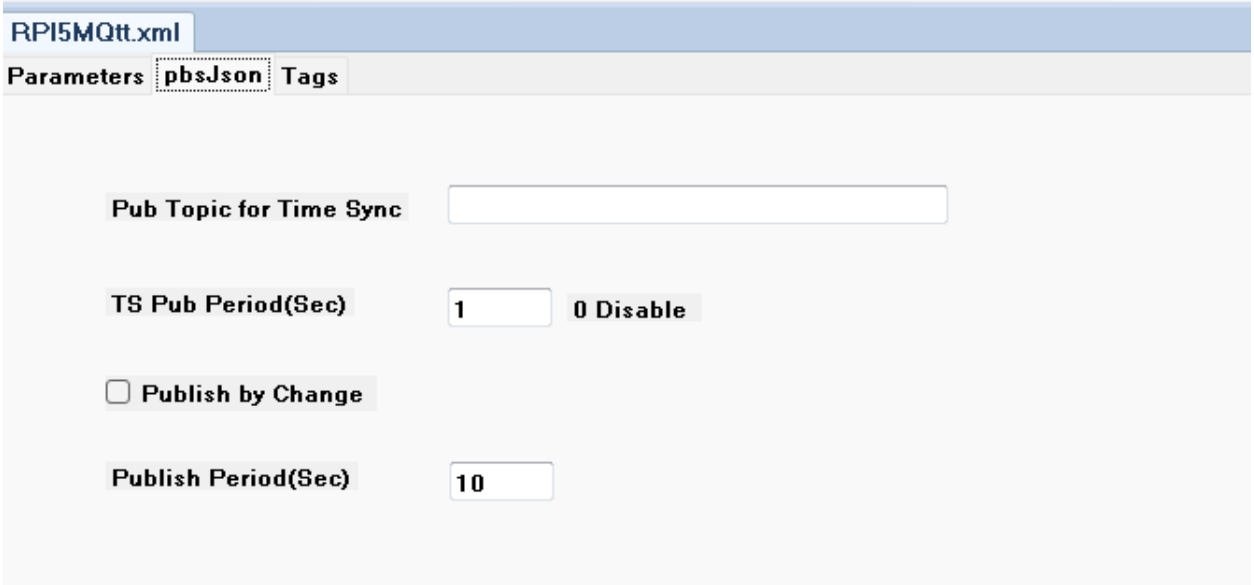


MQTT in pbsHMI Platform

To use MQTT in **pbsHMI**, you must create a separate **MQTT Channel** for each RTU. Open the **Channel** tab in the pbsHMI Editor, right-click, and select **"New MQTT Client."**
pbsHMI supports both **Raw** data and **pbsJson** data formats. For communication with RTUs based on **pbsSoftLogic**, you must use the **pbsJson** data model. All concepts and configuration principles are identical to those used in the pbsSoftLogic MQTT driver.

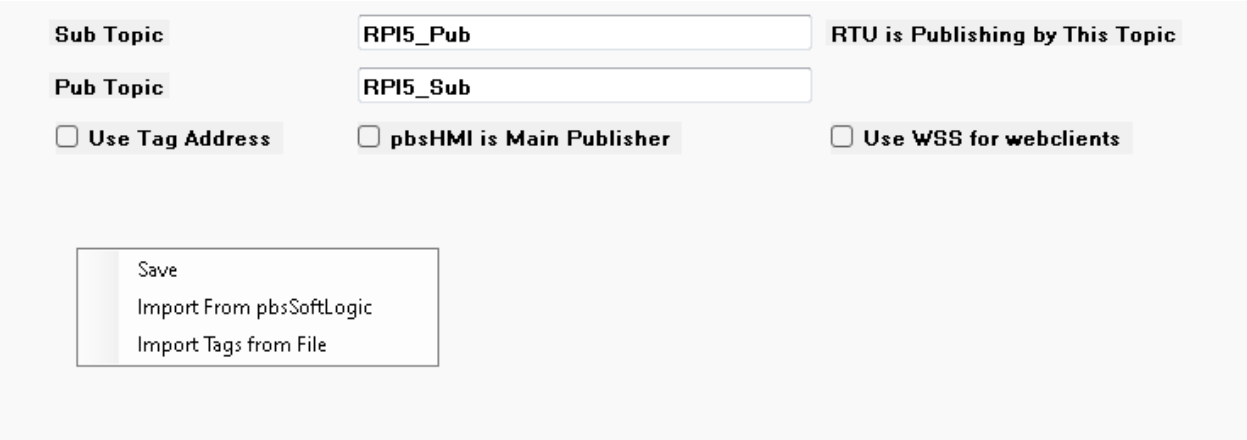


Communication Timeout (min): If pbsHMI does not receive any new **pbsJson** frame from the RTU within this period, it will disconnect from the broker and attempt to reconnect.
Sub Topic: The topic where the RTU publishes its data.
Pub Topic: The topic where the RTU subscribes to receive data.
Use Tag Address: When enabled, pbsHMI and pbsSoftLogic use **tag addresses** instead of tag names for communication. Therefore, each tag in pbsSoftLogic must have a **unique ID**.
pbsHMI is Main Publisher: pbsHMI can publish data (DO, AO) to the broker in the same way an RTU does. For example, if pbsHMI reads data from external devices via Modbus and needs to publish this data to the MQTT broker, this option allows it to act as the primary publisher.



In the **pbsJson** tab, you can configure the **Time Sync Topic** and its update period. These settings follow the same concepts used in pbsSoftLogic for time synchronization between the RTU and pbsHMI.
Publish Period defines how often pbsHMI publishes data to the broker. If **Publish by Change** is enabled, pbsHMI publishes data only when tag values change.
Both **Publish Period** and **Publish by Change** are applicable only when **pbsHMI is the main publisher**.

To define tags, you can either create them manually or import them from a pbsSoftLogic project or a sample configuration file. By right-clicking anywhere in the **MQTT Channel** area, you will see the available options for adding or importing tags.



Import From pbsSoftLogic: You can import tags and other parameters directly from the RTU project. You may select either the **Tag List file** or the **Option file**—pbsHMI will use both to import the required data.
Import Tags from File: pbsHMI can also load tags from the **cfg** folder inside the pbsHMI installation directory. The file name is **MQTTSample.xml**, and you may edit this file to match the requirements of different projects.

Parameters | pbsJson | Tags

Block Name	Type	Init Value	Address
SYS_Online	SYS	0	0
DITag1	DI	0	1
DITag2	DI	0	2
DITag3	DI	0	3
DITag4	DI	0	4
DITag5	DI	0	5
AITag1	AI	0	6
AITag2	AI	0	7
AITag3	AI	0	8
AITag4	AI	0	9
AITag5	AI	0	10
DOTag1	DO	0	11
DOTag2	DO	0	12
DOTag3	DO	0	13
AOTag1	AO	0	14
AOTag2	AO	0	15
AOTag3	AO	0	16
AOTag4	AO	0	17

DI and **AI** tags are received from the broker/RTU, while **DO** and **AO** tags are sent to the broker/RTU.

Sys_OnLine indicates that the MQTT driver is successfully communicating with the broker and is receiving new frames.

After creating the MQTT channel, you must define a **Channel–Device tag** in the **Tag** section of pbsHMI, just as you would for any other protocol supported by the system.