MQTT Sparkplug B Collector Connector

Devices compatible with the MQTT Sparkplug protocol

- Name: **MQTTspB**
- Version 1.0.2.26
- Protocol: MQTT 3.1.0 / 3.1.1 / 5.0.0
- Interface: TCP/IP
- Runtime: Multiplatform
- Configuration:
  - Devices / Protocols

Module Configuration
MQTT SpB Device
MQTT SpB TagProvider
Simulator
Troubleshooting
Driver Revision History

Module Configuration

This document has only information about the device connection settings and its address syntax.

For further Information, see:

- Devices Module
- Devices Module Reference

The software platform supports MQTT SparkplugB Protocol, configurable as a Device. Devices handle predefined, static data, ensuring control, security, and advanced metadata management, suitable for applications requiring strict validation like HMI and SCADA systems.

You can also configure this protocol as a TagProvider, which is recommended if you're working with dynamic data. Learn more at MQTT Client Service.

Implemented Methods

Connect, Disconnect, Subscribe, and Publish

The Connection Timeout for connections using the MQTTspB is set to 30 seconds, configured as a hard-coded value. This ensures that the connection attempt will be terminated if it is not completed within this period.

MQTT SpB Device

The initial configurations for devices in the software platform solutions involve defining a Channel using a target communication Protocol, which sets communication rules between the platform and devices and creates the base instance for linking a Node. Next, define the Node linked to the created Channel, and finally, define the Point you want to monitor.

Channel

Channels define global parameters for the usage of a communication protocol.

Protocol Options

Type: Defines how the channel works when connecting to the MQTT Broker.

> **Collector:** It is a primary MQTT client that consumes the real-time messages but also publishes commands (messages) to Publisher nodes and devices**. [Scada IIoT Host]**
>
> **Publisher**: It is a MQTT client that has as its main function to publish messages read from field equipment. It also can receive commands from Collectors. **[Eon Node]**

This Driver allows the Publisher type, only for compatibility with legacy applications, but there is a newer version for Publishing. The driver MQTT SparkplugB Publisher is newer and specifically designed and optimized to created to EON nodes publishing MQTT data using the SparkPlugB specification.

TagProperties: Defines the tag properties beyond the **Value**. The names of the properties should be separated by a **comma ' , '**.

TimePublishRate: Indicates the time for publishing, in milliseconds. The messages that will be published to the MQTT Broker are grouped and sent using this period.

StoreAndForward: When this option is enabled a database file, named as below, will be created at the same path of the project file.

> <ProjectName>.dbCh_N_<Device.Node.Id>_<GroupId>_<NodeId>
>
> Where:
>
> **<Device.Node.Id>** is the row value from the **ID** column in **Devices / Nodes**.

<GroupId> and **<NodeId>** come from the **Address** column in **Devices / Points.**

E.g.: MyMQTTProject.dbCh_N_0_MyGroupID_MyNodeID.

All data will be stored in this database before forwarding to Broker. The data that receives the Broker confirmation will be removed from this database. Then, when the communication with the Broker is lost, the data will stay in the local database until the connection is re-established to the Broker. The data-updating rate to the Broker is the time configured in the **TimePublishRate** field, even after a re-established connection and always using the first-in, first-out (FIFO) method to forward the data.

PublishDBIRTHAfterNBIRTH: Flag that indicates if a DBIRTH message will also be published when an NBIRTH message is requested.

UseParenthesesForArray: Flag that indicates if it should replace brackets by parentheses in the metric name.

The Built-in Broker can be started using the Run Broker button.

Node

Nodes represent the physical field devices (like a PLC) with a unique field address (PrimaryStation).

Station Configuration

The station syntax is:

BrokerURL ; Port ; [ClientID] ; [Username] ; [Password] ; [X509Certificate] ; [X509CertificatePassword] ; [NetworkSecurity] ; [WebSocket] ; [CollectorID] ; [QoS] ; [KeepAlive] ; [ReportOutOfRangeByException] ; [DisableRebirthRequest] ; [PublishSTATE]

Where:

BrokerURL: MQTT Broker (Server) URL. Defaults to *localhost*.

Port : MQTT Broker port. It must be the same port as configured in the Broker. Defaults to *1883*.

[**ClientID**]: A unique identifier for the client connecting to the MQTT broker. If left empty, a unique identified client will be assigned. This user-defined string can also include the following macros, intermixed with the text:

- _**Guid**_: Replaced by a dynamically generated Global Unique Identifier (GUID).
- _**SolutionName**_: Replaced by the name of the solution (without the path or file extension).
- _**NodeId**_: Replaced by the string defined as the NodeId.
- _**GroupId**_: Replaced by the string defined as the GroupId.

[Username]: Username defined on the MQTT. This is requested if the Broker needs this configuration (*optional*).

[Password]: Password defined on the MQTT. This is requested if the Broker needs this configuration (*optional*).

[X509Certificate]: Path of the X509 certificate. If using a server-side only certificate, this field does not need to be configured. This must be the complete path of the X509 certificate on the client computer. The certificate must be installed on the computer. Note: One of the easiest ways to install the certificate on the client computer is to use the wizard and use the *"Internet Options"* to import the certificate in the *"Trusted Root Certification Authorities"*. You need to import the certificate in the DER format (*optional*).

[X509Certificate Password]: Password created when X509Certificate was exported (*optional*).

[NetworkSecurity]: If the Broker works by using a certificate, this field is mandatory and must be the same as configured in MQTT. Options: None, Ssl2, Ssl3, Tls, Tls11, and Tls12 (*optional*).

[WebSocket]: When checked, the driver connects to the MQTT broker over WebSocket transport instead of native TCP. Network Security (TLS) applies independently: combine with a TLS option to obtain a secure WebSocket (WSS) connection. Use this when the broker is reachable only through a WebSocket endpoint, typical of HTTP-fronted brokers behind a reverse proxy (*optional*).

[CollectorID]: If channel works as Publisher, some Collector is active and other MQTT Brokers are also present in the network (redundancy, scalability, etc.,), this field is used to know that the MQTT Broker must be used. A Collector publishes its ID when connecting to MQTT Broker (optional).

QoS: Quality of Service (QoS) is an agreement between the sender and the receiver of messages regarding the guarantees of delivering a message. There are 3 QoS levels:

AtMostOnce: This service level guarantees a best-effort delivery. However, there is no guarantee of delivery, due to the recipient not acknowledging the receipt of the message.

AtLeastOnce: This service guarantees that a message is delivered at least one time to the receiver. *Default Value*.

ExactlyOnce: Highest level of service. It is safer, but slower when compared to the others. When the message flow is complete, all parties involved in the communication are sure the message was sent and delivered with success.

Choosing the Right QoS Level

Use QoS 0 when:

- The connection between sender and receiver is stable (e.g., a test client or front-end application connected via a wired connection).
- Occasional message loss is acceptable (e.g., when sending non-critical data at short intervals).
- Message queuing is **not** required.

Use QoS 1 when:

- Every message must be received, but duplicates are acceptable.
- The application **does not require** the overhead of QoS 2 but still needs message delivery guarantees.
- Faster delivery is preferred compared to QoS 2.

Use QoS 2 when:

- Receiving **every message exactly once** is critical (e.g., when duplicate messages could cause issues).
- Applications need the highest level of reliability, despite the additional overhead and latency.

KeepAlive: Ensures that the connection between the Broker and Client is open and both parties are aware of the connection. Default is *10 seconds*.

Report out of Range by Exception: When enabled, an item is published only when its value falls outside the Tag's ScaleMin / ScaleMax range.

DisableRebirthRequest: Disables NCMD/DCMD Rebirth solicitations to the broker. When *false* (default), the Collector requests fresh DBIRTH /NBIRTH from upstream Edge Nodes after sequence loss, per Sparkplug B host-state protocol. Set to *true* when a separate broker bridge owns rebirth orchestration, when a read-only Host policy forbids Host-originated commands, or when the broker or Edge-of-Network gateway is configured to reject NCMD / DCMD from the Host. Default: *false*.

PublishSTATE: Publishes the `STATE/<clientId>` host-state topic (*ONLINE* on connect, *OFFLINE* as Last-Will-and-Testament). When *false*, the Collector does not advertise host state — use only when a separate broker bridge owns STATE topics. Default: *true*.

If used 1.1.1.1 in the broker URL it will try establish the connection. This is to avoid timeouts, when creating solutions that the PrimaryStation IP will be set by script, using the Node properties.

If your client certificate is in .pem format with a .pem key, you can covert it into .pfx format using a built-in utility called CertificateConverter. exe that is located in your Product's installation directory (e.g.: ..\-9.3 \CertificateConverter.exe).

This utility requires **.NetCore 5.0** and **win-x64** operational system.
To use it, run the executable from the Command Line with the following input arguments:

- **/certificate:**
- **/key:**
- **/password:**

The syntax would be as detailed below:

```
CertificateConverter.exe /certificate:"<PathToCertificateFile.pem>"
                                                /key:"<PathToKeyFile.pem>"
                                                /password:<PasswordForExportedCertificate>
```

All nodes of the same channel share all the device points regardless of which node they were configured for. Setting more than one node for the same channel should be used for MQTT Broker redundancy.

When using the Test button in a Linux targeted project with TLS security, the certificate should be placed in the Project's directory in both Windows and Linux environments (for Engineering and Runtime). **Do not** use the full file path.

Points Configuration

Points represent tags within a device.

Address

The syntax for the MQTT Collector communication points is called ***Topic***:

- ***Topic*** is a string that is used by the broker to filter messages for each connected client. A topic consists of one or more topic levels. Each topic level is separated by a forward slash (topic level separator). Wildcards are **not** supported.

E.g.: ***USA/Houston/Station1/Level***

Access Type

AccessType defines how the device interacts with the platform (Read and/or Write).

This communication protocol works on an exception basis, using address subscriptions. In other words, when operating in **Collector** mode to **read** val ues, if the value changes in the broker, it will automatically be sent to our driver without the need for polling. Therefore, there is no need to configure read pooling for the value to be read. However, for **writing**, whether using **Collector** or **Publisher** mode, the write access type must be configured.

TagMapping

This protocol **DOES NOT** support the usage of simple Tags directly into the **Devices / Points** table. By simple Tags, we mean variables of common DataTypes (Integer, Digital, Text, ...).

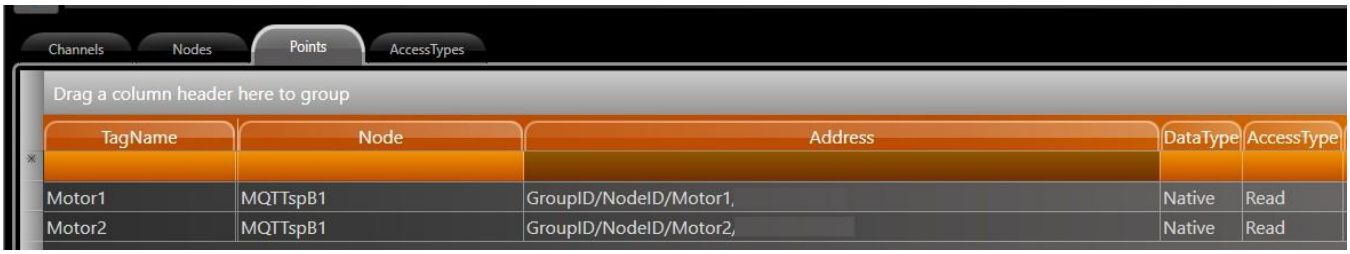
Instead of that, you need to build a DataModel / DataStucture using Templates, and use that Main element as your Communication Point. The example below illustrates that.

We have a Motor Template with members: Temperature and RPM. From that Template, two Tags are created, Motor1 and Motor2.



Once the Tags are created, we can map them to our DeviceIDs. In the **Devices / Points** tab, add just the main Tags (Motor1 and Motor2) to the table.

Both Tags are mapped to the same **NodeID**, but different **DeviceIDs**, as explained before



In Runtime, the Template members will be automatically expanded and your MQTTspB payload will look like the example below:


```
{
  "timestamp":1627559578321,
  "metrics":[
      {
        "name":"Temperature",
        "alias":1,
        "timestamp":1627559548277,
        "dataType":"Int32",
        "value":60
      },
      {
        "name":"RPM",
        "alias":2,
        "timestamp":1627559546172,
        "dataType":"Int32",
        "value":21
      },
      {
        "name":"Device Control/Rebirth",
        "timestamp":1627559578631,
        "dataType":"Boolean",
        "value":false
      }
  ],
  "seq":2
}
```

```
{
  "timestamp":1627559578321,
  "metrics":[
      {
        "name":"Temperature",
        "alias":1,
        "timestamp":1627559559829,
        "dataType":"Int32",
        "value":90
      },
      {
        "name":"RPM",
        "alias":2,
        "timestamp":1627559557398,
        "dataType":"Int32",
        "value":220
      },
      {
        "name":"Device Control/Rebirth",
        "timestamp":1627559578477,
        "dataType":"Boolean",
        "value":false
      }
  ],
  "seq":1
}
```


MQTT SpB TagProvider

This Communication Protocol supports the TagProvider feature, which is a tool that allows you to access your Communication Device Data Model without creating any Project Tags.

For more information, please refer to the *TagProvider* document.

How to Configure

To configure the MQTTspB protocol as a TagProvider, navigate to **Unified Namespace / TagProvider Connections** and create a new provider for the MQTT SparkplugB protocol.

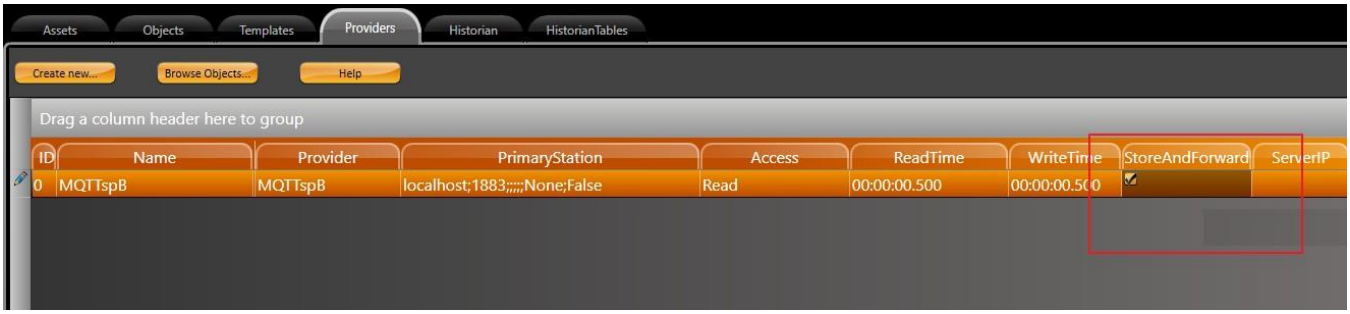
Configure the items under the **PrimaryStation** column the same way that was described in the Node Configuration.



The parameters configured in the **Channel** tab are not configured in this TagProvider feature. The default configuration is described below:

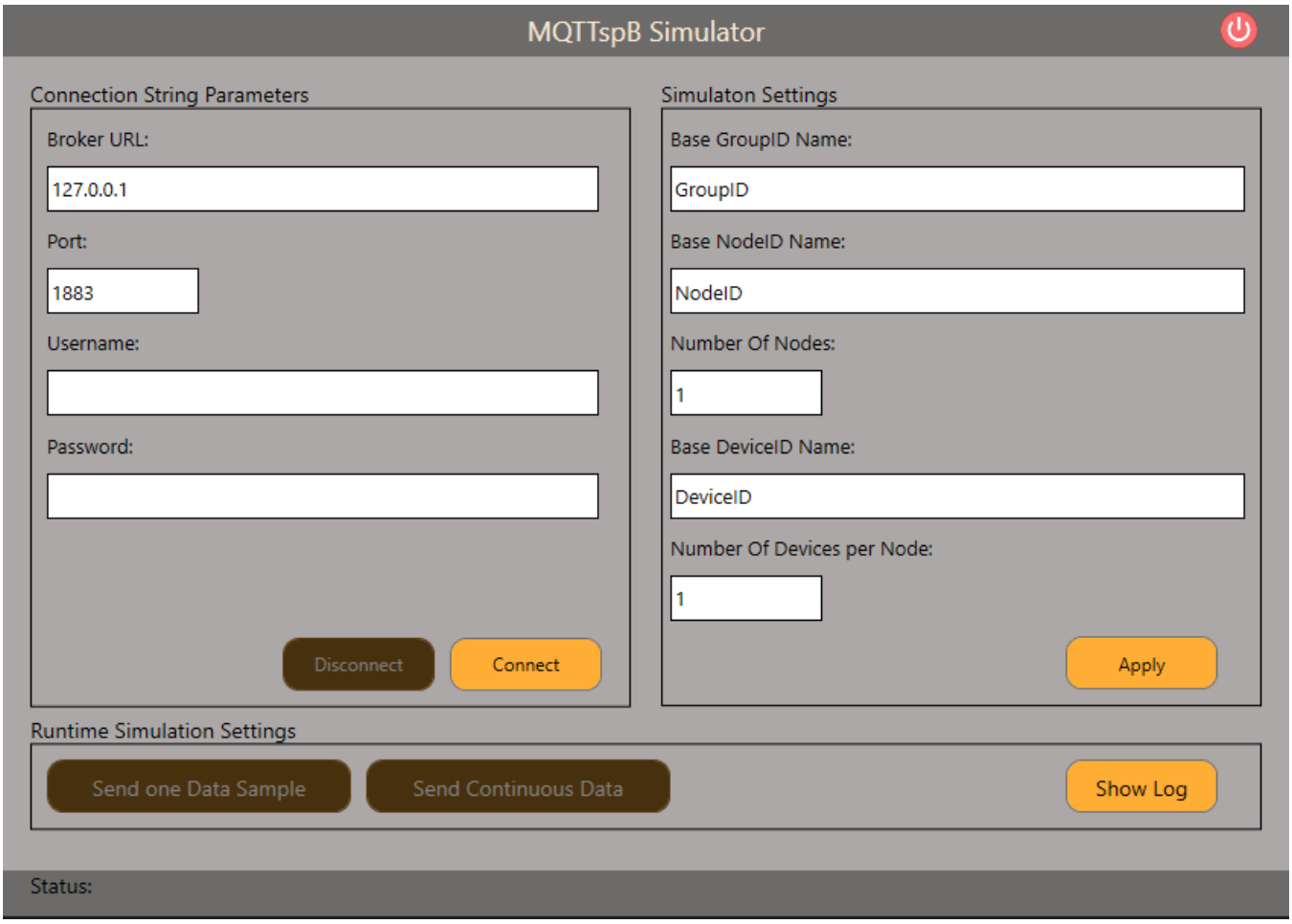
- **Type**: Collector.
- **TagProperties**: Only the value property is used in this communication.
- **Time Publish Rate**: 500ms.

The Store and Forward feature can be enabled for MQTTspB TagProvider at the column with the same name, as illustrated in the image below



> ⚠ The TagProvider for the MQTTspB protocol is a **Collector**, meaning you can publish **only** commands (NCMD and DCMD).

Simulator



Troubleshooting

The status of the driver execution can be observed through the diagnostic tools:

Diagnostic Tools

Diagnostics tools refer to the Property Watch, Trace Window, and Module Information. These tools identify communication blocks and address errors, allow property access to pinpoint error causes, and display system messages for device management and troubleshooting, offering a comprehensive approach to device management and troubleshooting for Solutions on the software platform.

Module Information

This tool provides insights into module operations and communication channels, offering information about virtual reading groups, error codes, and communication events. It helps users identify communication blocks and effectively address errors in industrial automation systems.

Property Watch

This tool enables users to access, read, and write tags and internal properties of the system. It assists in identifying the root cause of errors through negative error codes, allowing for real-time adjustments and optimizations to the system configuration.

Trace Window

This tool displays system messages in a data grid interface, providing information about the status of reads, writes, and communication frames. It is particularly useful for various PLCs and communication protocols, as it presents invalid addresses in the configuration, assisting in device management and troubleshooting.

> ⚠ All nodes of the same channel share all the device points regardless of which node they were configured for. Setting more than one node for the same channel should be used for MQTT Brokers redundancy.

Driver Revision History

MQTTspB Revision History	
Version	**Notes**
1.0.2.26	The Collector now acts as a Sparkplug 3.0 Primary Host: STATE is published as JSON on spBv1.0/STATE/<host id> (a legacy 2.x STATE option is available for older peers), and gaps in a node's message sequence trigger a rebirth request after a configurable reorder timeout; new station options Disable Rebirth Request and Publish STATE, and a fix for both options being read from the wrong field, which had silently disabled rebirth requests on default configurations; Report Out of Range by Exception now works: messages carry correct sequence numbers, the range is the tag's ScaleMin/ScaleMax (Min/Max clamps the value, so nothing was published), tags without a scale range are reported once in the Trace, and enabling the option no longer delays start-up; broker disconnects are now logged; station passwords can reference a Security Secret; a certificate not found at its configured path is also looked up in the machine Certificates folder; Test Connection in remote engineering states that the test ran on the Designer; tag importer: masked and encrypted passwords, SSL and QoS selections saved correctly, a browse dialog with configurable wait time, progress and cancel, and imported addresses without a QoS suffix.
1.0.2.16 (10.1.5)	Fixed off-by-one in PrimaryStation parser. **DisableRebirthRequest** and **PublishSTATE** now read correct segments. Behavioral note: rebirth solicitations resume per Sparkplug B spec for canonical 15-segment configs. Bridges that own rebirth orchestration should set **DisableRebirthRequest=true** explicitly to preserve pre-fix behavior.
1.0.2.10	Initial release on new documentation standards.

In this section...